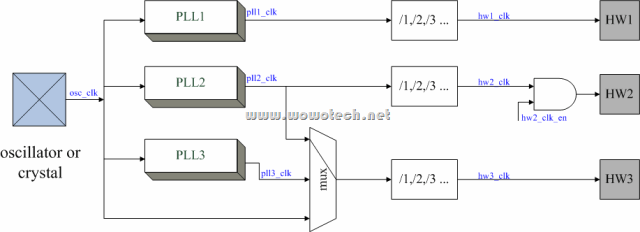
# Linux驱动\_clk子系统

## 1. clk子系统简介

### 1.1 clk子系统概述

如今，可运行Linux的主流处理器平台，都有非常复杂的clock tree，我们随便拿一个处理器的spec，查看clock相关的章节，一定会有一个非常庞大和复杂的树状图，这个图由clock相关的器件，以及这些器件输出的clock组成。下图是一个示例：

[](http://www.wowotech.net/content/uploadfile/201410/8321e8ac90bb43c1f00543082b74fbd820141020150656.gif)

clock相关的器件包括：用于产生clock的Oscillator（有源振荡器，也称作谐振荡器）或者Crystal（无源振荡器，也称晶振）；用于倍频的PLL（锁相环，Phase Locked Loop）；用于分频的divider；用于多路选择的Mux；用于clock enable控制的与门；使用clock的硬件模块（可称作consumer）等等。

common clock framework的管理对象，就是上图蓝色字体描述的clock（在软件中用struct clk抽象，以后就简称clk），主要内容包括（不需要所有clk都支持）：

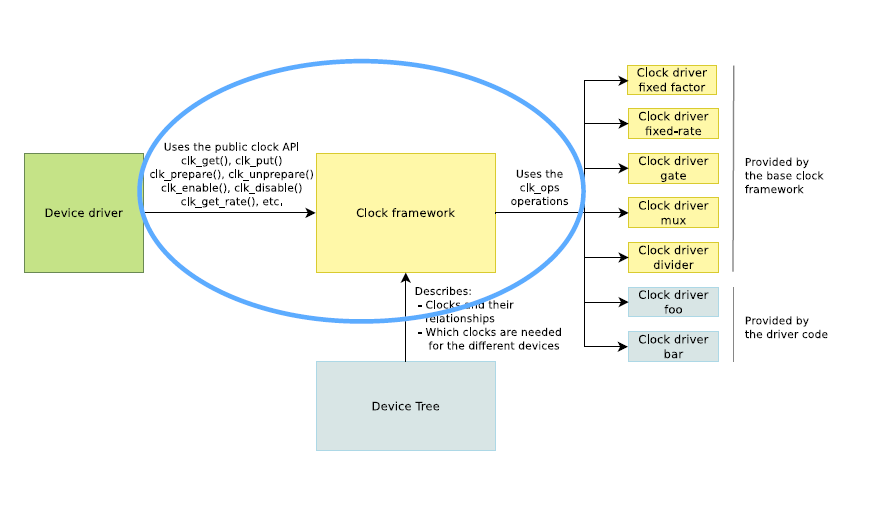
1）enable/disable clk。

2）设置clk的频率。

3）选择clk的parent，例如hw3\_clk可以选择osc\_clk、pll2\_clk或者pll3\_clk作为输入源。

### 1.2 clk子系统框图

clk framework是内核中用来统一管理clock的子系统。代码存在于kernel/driver/clk目录中。



上图中的黄色区域都是clk core所实现的功能，灰色区域是clock驱动开发需要做的事情，而绿色区域是其他device driver需要使用clock时要调用到的clk功能。

### 1.3 clock consumer通用API

kernel称clock driver为clock provider，相应的，clock的使用者为clock consumer。

这节主要描述clock consumer的用法。

管理clock的最终目的，是让device driver可以方便的使用，这些是通过include/linux/clk.h中的通用API实现的，如下：

#### 1.3.1 struct clk结构

一个系统的clock tree是固定的，因此clock的数目和用途也是固定的。假设上面图片所描述的是一个系统，它的clock包括osc\_clk、pll1\_clk、pll2\_clk、pll3\_clk、hw1\_clk、hw2\_clk和hw3\_clk。我们完全可以通过名字，抽象这7个clock，进行开/关、rate调整等操作。但这样做有一个缺点：不能很好的处理clock之间的级联关系，如hw2\_clk和hw3\_clk都关闭后，pll2\_clk才能关闭。因此就引入struct clk结构，以链表的形式维护这种关系。

同样的道理，系统的struct clk，也是固定的，由clock driver在系统启动时初始化完毕，需要访问某个clock时，只要获取它对应的struct clk结构即可。怎么获取呢？可以通过名字索引啊！很长一段时间内，kernel及driver就是使用这种方式管理和使用clock的。

最后，设备（由struct device表示）对应的clock（由struct clk表示）也是固定的啊，可不可以找到设备就能找到clock？可以，不过需要借助device tree。

对使用者（device driver）来说，struct clk只是访问clock的一个句柄，不用关心它内部的具体形态。

#### 1.3.2 clk获取相关的API

device driver在操作设备的clock之前，需要先获取和该clock关联的struct clk指针，获取的接口如下：

struct clk **\***clk\_get**(**struct device **\***dev**,** const char **\***id**);**

struct clk **\***devm\_clk\_get**(**struct device **\***dev**,** const char **\***id**);**

void clk\_put**(**struct clk **\***clk**);**

void devm\_clk\_put**(**struct device **\***dev**,** struct clk **\***clk**);**

struct clk **\***clk\_get\_sys**(**const char **\***dev\_id**,** const char **\***con\_id**);**

struct clk **\***of\_clk\_get**(**struct device\_node **\***np**,** int index**);**

struct clk **\***of\_clk\_get\_by\_name**(**struct device\_node **\***np**,** const char **\***name**);**

struct clk **\***of\_clk\_get\_from\_provider**(**struct of\_phandle\_args **\***clkspec**);**

**说明：**

a）clk\_get，以device指针或者id字符串（可以看作name）为参数，查找clock。

   a1）dev和id的任意一个可以为空。如果id为空，则必须有device tree的支持才能获得device对应的clk；

   a2）根据具体的平台实现，id可以是一个简单的名称，也可以 是一个预先定义的、唯一的标识（一般在平台提供的头文件中定义，如mach**/**clk**.**h）；

   a3）不可以在中断上下文调用。

b）devm\_clk\_get，和clk\_get一样，只是使用了device resource management，可以自动释放。

c）clk\_put、devm\_clk\_put，get的反向操作，一般和对应的get API成对调用。

d）clk\_get\_sys，类似clk\_get，不过使用device的name替代device结构。

e）of\_clk\_get、of\_clk\_get\_by\_name、of\_clk\_get\_from\_provider，device tree相关的接口，直接从相应的DTS node中，以index、name等为索引，获取clk，后面会详细说明。

#### 1.3.3 clk控制有关的API

int clk\_prepare**(**struct clk **\***clk**)**

void clk\_unprepare**(**struct clk **\***clk**)**

static inline int clk\_enable**(**struct clk **\***clk**)**

static inline void clk\_disable**(**struct clk **\***clk**)**

static inline unsigned long clk\_get\_rate**(**struct clk **\***clk**)**

static inline int clk\_set\_rate**(**struct clk **\***clk**,** unsigned long rate**)**

static inline long clk\_round\_rate**(**struct clk **\***clk**,** unsigned long rate**)**

static inline int clk\_set\_parent**(**struct clk **\***clk**,** struct clk **\***parent**)**

static inline struct clk **\***clk\_get\_parent**(**struct clk **\***clk**)**

static inline int clk\_prepare\_enable**(**struct clk **\***clk**)**

static inline void clk\_disable\_unprepare**(**struct clk **\***clk**)**

a）clk\_enable**/**clk\_disable，启动**/**停止clock。不会睡眠。

b）clk\_prepare**/**clk\_unprepare，启动clock前的准备工作**/**停止clock后的善后工作。可能会睡眠。

c）clk\_get\_rate**/**clk\_set\_rate**/**clk\_round\_rate，clock频率的获取和设置，其中clk\_set\_rate可能会不成功（例如没有对应的分频比），此时会返回错误。如果要确保设置成功，则需要先调用clk\_round\_rate接口，得到和需要设置的rate比较接近的那个值。d）获取**/**选择clock的parent clock。

e）clk\_prepare\_enable，将clk\_prepare和clk\_enable组合起来，一起调用。clk\_disable\_unprepare**,**将clk\_disable和clk\_unprepare组合起来，一起调用。

prepare/unprepare，enable/disable的说明：

这两套API的本质，是把clock的启动/停止分为atomic和non-atomic两个阶段，以方便实现和调用。因此上面所说的“不会睡眠/可能会睡眠”，有两个角度的含义：一是告诉底层的clock driver，请把可能引起睡眠的操作，放到prepare/unprepare中实现，一定不能放到enable/disable中；二是提醒上层使用clock的driver，调用prepare/unprepare接口时可能会睡眠哦，千万不能在atomic上下文（例如中断处理中）调用哦，而调用enable/disable接口则可放心。

另外，clock的开关为什么需要睡眠呢？这里举个例子，例如enable PLL clk，在启动PLL后，需要等待它稳定。而PLL的稳定时间是很长的，这段时间要把CPU交出（进程睡眠），不然就会浪费CPU。

最后，为什么会有合在一起的clk\_prepare\_enable/clk\_disable\_unprepare接口呢？如果调用者能确保是在non-atomic上下文中调用，就可以顺序调用prepare/enable、disable/unprepared，为了简单，framework就帮忙封装了这两个接口。

#### 1.3.4 其他接口

int clk\_notifier\_register**(**struct clk **\***clk**,** struct notifier\_block **\***nb**);**

int clk\_notifier\_unregister**(**struct clk **\***clk**,** struct notifier\_block **\***nb**);**

这两个notify接口，用于注册/注销 clock rate改变的通知。例如某个driver关心某个clock，期望这个clock的rate改变时，通知到自己，就可以注册一个notify。后面会举个例子详细说明。

#### 1.3.5 使用说明

首先，在DTS（device tree source）中，指定device需要使用的clock，如下：

/\* DTS \*/

device **{**

clocks **=** **<&**osc 1**>,** **<&**ref 0**>;**

clock**-**names **=** "baud"**,** "register"**;**

**};**

该DTS的含义是：

device需要使用两个clock，“baud”和“regitser”，由clock-names关键字指定；baud取自“osc”的输出1，register取自“ref”的输出0，由clocks关键字指定。那么问题来了，clocks关键字中，<&osc 1>样式的字段是怎么来的？是由clock的provider，也就是底层clock driver规定的。所以使用clock时，一定要找clock driver拿相关的信息（一般会放在“Documentation/devicetree/bindings/clock/”目录下）。

系统启动后，device tree会解析clock有关的关键字，并将解析后的信息放在platform\_device相关的字段中。

具体的driver可以在probe时，以clock的名称（不提供也行）为参数，调用clk get接口，获取clock的句柄，然后利用该句柄，可直接进行enable、set rate等操作，如下：

/\* driver \*/

int xxx\_probe**(**struct platform\_device **\***pdev**)**

**{**

struct clk **\***baud\_clk**;**

int ret**;**

baud\_clk **=** devm\_clk\_get**(&**pdev**->**dev**,** “baud”**);**

**if** **(**IS\_ERR**(**clk**))** **{**

…

**}**

ret **=** clk\_prepare\_enable**(**baud\_clk**);**

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

…

**}**

**}**

### 1.4 clock provider重要的数据结构

#### 1.4.1 struct clk\_hw

clock framework使用struct clk结构抽象clock，但该结构对clock consumer是透明的（不需要知道它的内部细节）。同样，struct clk对clock provider也是透明的。framework提供了struct clk\_hw结构，从clock provider的角度，描述clock，该结构的定义如下：

struct clk\_hw **{**

struct clk\_core **\***core**;**

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

const struct clk\_init\_data **\***init**;**

**};**

core：指向对应clock device实例。由于系统中的clk\_hw和clk\_core实例是一一对应的，因此，struct clk\_core中也有指回clk\_hw的数据成员。

clk：访问clk\_core的实例，每当consumer通过clk\_get一个clock device（也就是clk\_core）发起访问的时候都需要获取一个句柄，也就是clk。每一个用户访问都会有一个clk句柄，同样的，底层模块对其访问亦然。因此，这里clk是底层clk\_hw访问clk\_core的句柄实例。

init**:**在底层clock provider driver初始化的过程中，会调用clk\_register接口函数注册clk\_hw。当然，这时候需要设定一些初始数据，而这些初始数据被抽象成一个struct clk\_init\_data数据结构。在初始化过程中，clk\_init\_data的数据被用来初始化clk\_hw对应的clk\_core数据结构，当初始化完成之后，clk\_init\_data则没有存在的意义了。

init静态数据的数据结构如下：

struct clk\_init\_data **{**

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

const struct clk\_ops **\***ops**;**

const char **\*\***parent\_names**;**

u8 num\_parents**;**

unsigned long flags**;**

**};**

name：该clock的名称；

ops：该clock相关的操作函数集，具体参考下面的描述；

parent\_names：该clock所有的parent clock的名称。这是一个字符串数组，保存了所有可能的parent；

num\_parents：parent的个数；

flags：一些framework级别的flags，后面会详细说明。

clk\_ops数据结构如下：

struct clk\_ops **{**

int **(\***prepare**)(**struct clk\_hw **\***hw**);**

void **(\***unprepare**)(**struct clk\_hw **\***hw**);**

int **(\***is\_prepared**)(**struct clk\_hw **\***hw**);**

void **(\***unprepare\_unused**)(**struct clk\_hw **\***hw**);**

int **(\***enable**)(**struct clk\_hw **\***hw**);**

void **(\***disable**)(**struct clk\_hw **\***hw**);**

int **(\***is\_enabled**)(**struct clk\_hw **\***hw**);**

void **(\***disable\_unused**)(**struct clk\_hw **\***hw**);**

unsigned long **(\***recalc\_rate**)(**struct clk\_hw **\***hw**,**

unsigned long parent\_rate**);**

long **(\***round\_rate**)(**struct clk\_hw **\***hw**,** unsigned long**,**

unsigned long **\*);**

int **(\***set\_parent**)(**struct clk\_hw **\***hw**,** u8 index**);**

u8 **(\***get\_parent**)(**struct clk\_hw **\***hw**);**

int **(\***set\_rate**)(**struct clk\_hw **\***hw**,** unsigned long**,**

unsigned long**);**

void **(\***init**)(**struct clk\_hw **\***hw**);**

**};**

这是clock的操作函数集，和clock consumer中的通用API一致。

is\_prepared：判断clock是否已经prepared。可以不提供，clock framework core会维护一个prepare的计数（该计数在clk\_prepare调用时加一，在clk\_unprepare时

减一），并依据该计数判断是否prepared;

unprepare\_unused：自动unprepare unused clocks；

is\_enabled：和is\_prepared类似；

disable\_unused：自动disable unused clocks；

clock framework core提供一个clk\_disable\_unused接口，在系统初始化的late\_call中调用，用于关闭unused clocks，这个接口会调用相应clock的**.**unprepare\_unused和**.**disable\_unused函数。

recalc\_rate：以parent clock rate为参数，重新计算并返回clock rate；

该结构没有提供get\_rate函数，因为会有一个rate变量缓存，另外可以使用recalc\_rate。

round\_rate：该接口有点特别，在返回rounded rate的同时，会通过一个指针，返回round后parent的rate。这和CLK\_SET\_RATE\_PARENT flag有关；

init：clock的初始化接口，会在clock被register到内核时调用。

flags标志位定义如下。

#define CLK\_SET\_RATE\_GATE BIT(0) /\* must be gated across rate change \*/

#define CLK\_SET\_PARENT\_GATE BIT(1) /\* must be gated across re-parent \*/

#define CLK\_SET\_RATE\_PARENT BIT(2) /\* propagate rate change up one level \*/

#define CLK\_IGNORE\_UNUSED BIT(3) /\* do not gate even if unused \*/

#define CLK\_IS\_ROOT BIT(4) /\* root clk, has no parent \*/

#define CLK\_IS\_BASIC BIT(5) /\* Basic clk, can't do a to\_clk\_foo() \*/

#define CLK\_GET\_RATE\_NOCACHE BIT(6) /\* do not use the cached clk rate

上面是framework级别的flags，可以使用或的关系，指定多个flags，解释如下：

CLK\_SET\_RATE\_GATE：表示在改变该clock的rate时，必须gated（关闭）；

CLK\_SET\_PARENT\_GATE：表示在改变该clock的parent时，必须gated（关闭）；

CLK\_SET\_RATE\_PARENT：表示改变该clock的rate时，要将该改变传递到上层parent（下面再详细说明）；

CLK\_IGNORE\_UNUSED：忽略disable unused的调用；

CLK\_IS\_ROOT：该clock为root clock，没有parent；

CLK\_IS\_BASIC：不再使用了；

CLK\_GET\_RATE\_NOCACHE：get rate时，不要从缓存中拿，而是从新计算。

#### 1.4.2 **clk\_register**

系统中，每一个clock都有一个struct clk\_hw变量描述，clock provider需要使用register相关的接口，将这些clock注册到kernel，clock framework的核心代码会把它们转换为struct clk变量，并以tree的形式组织起来。这些接口的原型如下：

struct clk **\***clk\_register**(**struct device **\***dev**,** struct clk\_hw **\***hw**);**

struct clk **\***devm\_clk\_register**(**struct device **\***dev**,** struct clk\_hw **\***hw**);**

void clk\_unregister**(**struct clk **\***clk**);**

void devm\_clk\_unregister**(**struct device **\***dev**,** struct clk **\***clk**);**

这些API比较简单（复杂的是怎么填充struct clk\_hw变量），register接口接受一个填充好的struct clk\_hw指针，将它转换为sruct clk结构，并根据parent的名字，添加到clock tree中。

不过，clock framework所做的远比这周到，它基于clk\_register，又封装了其它接口，使clock provider在注册clock时，连struct clk\_hw都不需要关心，而是直接使用类似人类语言的方式。

#### 1.4.3 clock分类及注册

根据clock的特点，clock framework将clock分为fixed rate、gate、devider、mux、fixed factor、composite六类，每一类clock都有相似的功能、相似的控制方式，因而可以使用相同的逻辑，统一处理，这充分体现了面向对象的思想。

##### 1.4.3.1 fixed rate clock

这一类clock具有固定的频率，不能开关、不能调整频率、不能选择parent、不需要提供任何的clk\_ops回调函数，是最简单的一类clock。

可以直接通过DTS配置的方式支持，clock framework core能直接从DTS中解出clock信息，并自动注册到kernel，不需要任何driver支持。

clock framework使用struct clk\_fixed\_rate结构抽象这一类clock，另外提供了一个接口，可以直接注册fixed rate clock。

struct clk\_fixed\_rate **{**

struct clk\_hw hw**;**

unsigned long fixed\_rate**;**

u8 flags**;**

**};**

extern const struct clk\_ops clk\_fixed\_rate\_ops**;**

clk **=** clk\_register\_fixed\_rate\_with\_accuracy**(NULL,** clk\_name**,** **NULL,** 0**,** rate**,** accuracy**);**

clock provider一般不需要直接使用struct clk\_fixed\_rate结构，因为clk\_register\_fixed\_rate接口是非常方便的；

clk\_register\_fixed\_rate接口以clock name、parent name、fixed\_rate为参数，创建一个具有固定频率的clock，该clock的clk\_ops也是clock framework提供的，不需要provider关心；

如果使用DTS的话，clk\_register\_fixed\_rate都不需要，直接在DTS中配置即可，后面会说明。

##### 1.4.3.2 gate clock

这一类clock只可开关（会提供.enable/.disable回调），可使用下面接口注册：

struct clk **\***clk\_register\_gate**(**struct device **\***dev**,** const char **\***name**,**

const char **\***parent\_name**,** unsigned long flags**,**

void \_\_iomem **\***reg**,** u8 bit\_idx**,**

u8 clk\_gate\_flags**,** spinlock\_t **\***lock**);**

需要提供的参数包括：

name：clock的名称；

parent\_name：parent clock的名称，没有的话可留空；

flags：可参考前面的说明；

reg：控制该clock开关的寄存器地址（虚拟地址）；

bit\_idx：控制clock开关的bit位（是1开，还是0开，可通过下面gate特有的flag指定）；

clk\_gate\_flags：gate clock特有的flag，当前只有一种：CLK\_GATE\_SET\_TO\_DISABLE：clock开关控制的方式，如果置位，表示写1关闭clock，反之亦然；

lock：如果clock开关时需要互斥，可提供一个spinlock。

##### 1.4.3.3 divider clock

这一类clock可以设置分频值（因而会提供.recalc\_rate/.set\_rate/.round\_rate回调），可通过下面两个接口注册：

struct clk **\***clk\_register\_divider**(**struct device **\***dev**,** const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,** void \_\_iomem **\***reg**,** u8 shift**,** u8 width**,** u8 clk\_divider\_flags**,** spinlock\_t **\***lock**);**

该接口用于注册分频比规则的clock：

reg**:**控制clock分频比的寄存器；

shift**:**控制分频比的bit在寄存器中的偏移；

width**:**控制分频比的bit位数，默认情况下，实际的divider值是寄存器值加1。如果有其它例外，可使用下面的的flag指示；

clk\_divider\_flags**:**divider clock特有的flag，包括：

CLK\_DIVIDER\_ONE\_BASED，实际的divider值就是寄存器值（0是无效的，除非设置CLK\_DIVIDER\_ALLOW\_ZERO flag）；

CLK\_DIVIDER\_POWER\_OF\_TWO，实际的divider值是寄存器值得2次方；

CLK\_DIVIDER\_ALLOW\_ZERO，divider值可以为0（不改变，视硬件支持而定）。

如有需要其他分频方式，就需要使用另外一个接口，如下：

struct clk **\***clk\_register\_divider\_table**(**struct device **\***dev**,** const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,** void \_\_iomem **\***reg**,** u8 shift**,** u8 width**,** u8 clk\_divider\_flags**,** const struct clk\_div\_table **\***table**,** spinlock\_t **\***lock**);**

该接口用于注册分频比不规则的clock，和上面接口比较，差别在于divider值和寄存器值得对应关系由一个table决定，该table的原型为：

struct clk\_div\_table **{**

unsigned int val**;**

unsigned int div**;**

**};**

其中val表示寄存器值，div表示分频值，它们的关系也可以通过clk\_divider\_flags改变。

##### 1.4.3.4 mux clock

这一类clock可以选择多个parent，因为会实现.get\_parent/.set\_parent/.recalc\_rate回调，可通过下面两个接口注册：

struct clk **\***clk\_register\_mux**(**struct device **\***dev**,** const char **\***name**,**const char **\*\***parent\_names**,** u8 num\_parents**,** unsigned long flags**,**void \_\_iomem **\***reg**,** u8 shift**,** u8 width**,** u8 clk\_mux\_flags**,** spinlock\_t **\***lock**);**

该接口可注册mux控制比较规则的clock（类似divider clock）：

parent\_names**:**一个字符串数组，用于描述所有可能的parent clock；

num\_parents**:**parent clock的个数；

reg、shift、width**:**选择parent的寄存器、偏移、宽度，默认情况下，寄存器值为0时，对应第一个parent，依此类推。如有例外，可通过下面的flags，以及另外一个接口实现；

clk\_mux\_flags**:**mux clock特有的flag：

        CLK\_MUX\_INDEX\_ONE，寄存器值不是从0开始，而是从1开始；

        CLK\_MUX\_INDEX\_BIT，寄存器值为2的幂。

同样还需要注册分频比不规则的的函数。

struct clk **\***clk\_register\_mux\_table**(**struct device **\***dev**,** const char **\***name**,**const char **\*\***parent\_names**,** u8 num\_parents**,** unsigned long flags**,** void \_\_iomem **\***reg**,** u8 shift**,** u32 mask**,** u8 clk\_mux\_flags**,** u32 **\***table**,** spinlock\_t **\***lock**);**

该接口通过一个table，注册mux控制不规则的clock，原理和divider clock类似，不再详细介绍。

##### 1.4.3.5 fixed factor clock

这一类clock具有固定的factor（即multiplier和divider），clock的频率是由parent clock的频率，乘以mul，除以div，多用于一些具有固定分频系数的clock。由于parent clock的频率可以改变，因而fix factor clock也可该改变频率，因此也会提供.recalc\_rate/.set\_rate/.round\_rate等回调。可通过下面接口注册：

struct clk **\***clk\_register\_fixed\_factor**(**struct device **\***dev**,** const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,** unsigned int mult**,** unsigned int div**);**

另外，这一类接口和fixed rateclock类似，不需要提供driver，只需要配置dts即可。

##### 1.4.3.6 composite clock

顾名思义，就是mux、divider、gate等clock的组合，可通过下面接口注册：

struct clk **\***clk\_register\_composite**(**struct device **\***dev**,** const char **\***name**,** const char **\*\***parent\_names**,** int num\_parents**,** struct clk\_hw **\***mux\_hw**,** const struct clk\_ops **\***mux\_ops**,** struct clk\_hw **\***rate\_hw**,** const struct clk\_ops **\***rate\_ops**,** struct clk\_hw **\***gate\_hw**,** const struct clk\_ops **\***gate\_ops**,** unsigned long flags**);**

#### 1.4.4 dts相关的API

clock driver使用一个DTS node描述一个clock provider，而clock consumer则会使用类似“clocks = <&clock  32>, <&clock 45>;”的形式引用，clock framework会自行把这些抽象的数字转换成实际的struct clk结构，怎么做的呢？肯定离不开clock provider的帮助。

上面描述的regitser接口，负责把clocks抽象为一个一个的struct clock，与此同时，clock provider需要把这些struct clk结构保存起来，并调用clock framework的接口，将这些对应信息告知framework的OF模块，这样才可以帮助将clock consumer的DTS描述转换为struct clk结构。该接口如下：

int of\_clk\_add\_provider**(**struct device\_node **\***np**,** struct clk **\*(\***clk\_src\_get**)(**struct of\_phandle\_args **\***args**,**void **\***data**),** void **\***data**);**

参数解析如下：

np**:**device\_node指针，clock provider在和自己的DTS匹配时获得；

clk\_src\_get**:**获取struct clk指针的回调函数，由clock provider根据实际的逻辑实现，参数说明如下：

    args**:**struct of\_phandle\_args类型的指针，由DTS在解析参数时传递。例如上面的“clocks **=** **<&**clock  32**>,** **<&**clock 45**>;**”，32、45就是通过这个指针传进来的；

    data**:**保存struct clk结构的指针，通常是一个数组，具体由provider决定。

data**:**和回调函数中的data意义相同，只是这里由provider提供，get时由clock framework core传递给回调函数。

对于常用的one cell clock provider，clock framework core提供一个默认的回调用函数，如下：

struct clk\_onecell\_data **{**

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

unsigned int clk\_num**;**

**};**

struct clk **\***of\_clk\_src\_onecell\_get**(**struct of\_phandle\_args **\***clkspec**,** void **\***data**);**

其中data指针为struct clk\_onecell\_data结构，该结构提供了clk指针和clk\_num的对应，clock provider在regitser clocks时，同时维护一个clk和num对应的数组，并调用of\_clk\_add\_provider接口告知clock framework core即可。

#### 1.4.5编写clock驱动的步骤

编写clock driver的步骤大概如下：

1）分析硬件的clock tree，按照上面所描述的分类，将这些clock分类。

2）将clock tree在DTS中描述出来，需要注意以下几2点：

a）对于fixed rate clocks，.compatible固定填充"fixed-clock"，并提供"clock-frequency"和"clock-output-names"关键字。之后不需要再driver中做任何处理，clock framework core会帮我们搞定一切。

b）同样，对于fixed factor clock，.compatible为"fixed-factor-clock"，并提供"clock-div"、"clock-mult"和"clock-output-names"关键字。clock framework core会帮我们搞定一切。

3）对于不能由clock framework core处理的clock，需要在driver中使用struct of\_device\_id进行匹配，并在初始化时，调用OF模块，查找所有的DTS匹配项，并执行合适的regitser接口，注册clock。

4）注册clock的同时，将返回的struct clk指针，保存在一个数组中，并调用of\_clk\_add\_provider接口，告知clock framework core。

### 1.5 两种形式的clock DTS

**方式1：**将系统所有的clock，抽象为一个虚拟的设备，用一个DTS node表示。这个虚拟的设备称作clock controller，参考如下例子：

clock**:** clock**-**controller@0x10030000 **{**

compatible **=** "samsung,exynos4210-clock"**;**

reg **=** **<**0x10030000 0x20000**>;**

#clock-cells = <1>;

**};**

clock，该clock设备的名称，clock consumer可以根据该名称引用clock；

#clock-cells，该clock的cells，1表示该clock有多个输出，clock consumer需要通过ID值指定所要使用的clock（很好理解，系统那么多clock，被抽象为1个设备，因而需要额外的ID标识）。

**方式2：**每一个可输出clock的器件，如Oscillator、PLL、Mux等等，都是一个设备，用一个DTS node表示。每一个器件，即是clock provider，也是clock consumer（根节点除外，如OSC），因为它需要接受clock输入，经过处理后，输出clock。

clocks **{**

#address-cells = <1>;

#size-cells = <1>;

ranges**;**

dummy**:** dummy **{**

#clock-cells = <0>;

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

clock**-**frequency **=** **<**0**>;**

**};**

osc24M**:** osc24M@01c20050 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-osc-clk"**;**

reg **=** **<**0x01c20050 0x4**>;**

clock**-**frequency **=** **<**24000000**>;**

**};**

osc32k**:** osc32k **{**

#clock-cells = <0>;

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

clock**-**frequency **=** **<**32768**>;**

**};**

pll1**:** pll1@01c20000 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-pll1-clk"**;**

reg **=** **<**0x01c20000 0x4**>;**

clocks **=** **<&**osc24M**>;**

**};**

/\* dummy is 200M \*/

cpu**:** cpu@01c20054 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-cpu-clk"**;**

reg **=** **<**0x01c20054 0x4**>;**

clocks **=** **<&**osc32k**>,** **<&**osc24M**>,** **<&**pll1**>,** **<&**dummy**>;**

**};**

axi**:** axi@01c20054 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-axi-clk"**;**

reg **=** **<**0x01c20054 0x4**>;**

clocks **=** **<&**cpu**>;**

**};**

axi\_gates**:** axi\_gates@01c2005c **{**

#clock-cells = <1>;

compatible **=** "allwinner,sun4i-axi-gates-clk"**;**

reg **=** **<**0x01c2005c 0x4**>;**

clocks **=** **<&**axi**>;**

clock**-**output**-**names **=** "axi\_dram"**;**

**};**

ahb**:** ahb@01c20054 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-ahb-clk"**;**

reg **=** **<**0x01c20054 0x4**>;**

clocks **=** **<&**axi**>;**

**};**

ahb\_gates**:** ahb\_gates@01c20060 **{**

#clock-cells = <1>;

compatible **=** "allwinner,sun4i-ahb-gates-clk"**;**

reg **=** **<**0x01c20060 0x8**>;**

clocks **=** **<&**ahb**>;**

clock**-**output**-**names **=** "ahb\_usb0"**,** "ahb\_ehci0"**,**

"ahb\_ohci0"**,** "ahb\_ehci1"**,** "ahb\_ohci1"**,** "ahb\_ss"**,**

"ahb\_dma"**,** "ahb\_bist"**,** "ahb\_mmc0"**,** "ahb\_mmc1"**,**

"ahb\_mmc2"**,** "ahb\_mmc3"**,** "ahb\_ms"**,** "ahb\_nand"**,**

"ahb\_sdram"**,** "ahb\_ace"**,** "ahb\_emac"**,** "ahb\_ts"**,**

"ahb\_spi0"**,** "ahb\_spi1"**,** "ahb\_spi2"**,** "ahb\_spi3"**,**

"ahb\_pata"**,** "ahb\_sata"**,** "ahb\_gps"**,** "ahb\_ve"**,**

"ahb\_tvd"**,** "ahb\_tve0"**,** "ahb\_tve1"**,** "ahb\_lcd0"**,**

"ahb\_lcd1"**,** "ahb\_csi0"**,** "ahb\_csi1"**,** "ahb\_hdmi"**,**

"ahb\_de\_be0"**,** "ahb\_de\_be1"**,** "ahb\_de\_fe0"**,**

"ahb\_de\_fe1"**,** "ahb\_mp"**,** "ahb\_mali400"**;**

**};**

apb0**:** apb0@01c20054 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-apb0-clk"**;**

reg **=** **<**0x01c20054 0x4**>;**

clocks **=** **<&**ahb**>;**

**};**

apb0\_gates**:** apb0\_gates@01c20068 **{**

#clock-cells = <1>;

compatible **=** "allwinner,sun4i-apb0-gates-clk"**;**

reg **=** **<**0x01c20068 0x4**>;**

clocks **=** **<&**apb0**>;**

clock**-**output**-**names **=** "apb0\_codec"**,** "apb0\_spdif"**,**

"apb0\_ac97"**,** "apb0\_iis"**,** "apb0\_pio"**,** "apb0\_ir0"**,**

"apb0\_ir1"**,** "apb0\_keypad"**;**

**};**

/\* dummy is pll62 \*/

apb1\_mux**:** apb1\_mux@01c20058 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-apb1-mux-clk"**;**

reg **=** **<**0x01c20058 0x4**>;**

clocks **=** **<&**osc24M**>,** **<&**dummy**>,** **<&**osc32k**>;**

**};**

apb1**:** apb1@01c20058 **{**

#clock-cells = <0>;

compatible **=** "allwinner,sun4i-apb1-clk"**;**

reg **=** **<**0x01c20058 0x4**>;**

clocks **=** **<&**apb1\_mux**>;**

**};**

apb1\_gates**:** apb1\_gates@01c2006c **{**

#clock-cells = <1>;

compatible **=** "allwinner,sun4i-apb1-gates-clk"**;**

reg **=** **<**0x01c2006c 0x4**>;**

clocks **=** **<&**apb1**>;**

clock**-**output**-**names **=** "apb1\_i2c0"**,** "apb1\_i2c1"**,**

"apb1\_i2c2"**,** "apb1\_can"**,** "apb1\_scr"**,**

"apb1\_ps20"**,** "apb1\_ps21"**,** "apb1\_uart0"**,**

"apb1\_uart1"**,** "apb1\_uart2"**,** "apb1\_uart3"**,**

"apb1\_uart4"**,** "apb1\_uart5"**,** "apb1\_uart6"**,**

"apb1\_uart7"**;**

**};**

**};**

osc24M和osc32k是两个root clock，因此只做clock provider功能。它们的cells均为0，因为直接使用名字即可引用。另外，增加了“clock-frequency”自定义关键字，这样在板子使用的OSC频率改变时，如变为12M，不需要重新编译代码，只需更改DTS的频率即可（这不正是Device Tree的核心思想吗！）。

pll1即是clock provider（cell为0，直接用名字引用），也是clock consumer（clocks关键字，指定输入clock为“osc24M”）；

再看一个复杂一点的，ahb\_gates，它是clock provider（cell为1），通过clock-output-names关键字，描述所有的输出时钟。同时它也是clock consumer（由clocks关键字可知输入clock为“ahb”）。需要注意的是，clock-output-names关键字只为了方便clock provider编程方便，clock consumer不能使用（或者可理解为不可见）；

我们在上面提到了clock provider的两种DTS定义方式，哪一种好呢？

从规范化、条理性的角度，毫无疑问方式2是好的，它真正理解了Device Tree的精髓，并细致的执行。且可以利用很多clock framework的标准实现。

而方式1的优点是，DTS容易写，相应的clock driver也较为直观，只是注册一个一个clock provider即可，没有什么逻辑可言。换句话说，方式1比较懒。

## 2. 2440的clk子系统代码分析

### 2.1 fixed-clock

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>;

};

};

2440外部有一个外部晶振，固定频率为12M。该clock driver对应的代码如下：

void \_\_init of\_fixed\_clk\_setup**(**struct device\_node **\***node**)**

**{**

\_of\_fixed\_clk\_setup**(**node**);**

**}**

CLK\_OF\_DECLARE**(**fixed\_clk**,** "fixed-clock"**,** of\_fixed\_clk\_setup**);**

这部分代码由linux内核提供，下面为\_of\_fixed\_clk\_setup代码。

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**;**

//获取dts节点中的频率12000000

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

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

//获取dts节点中的clock-accuracy属性

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

//获取dts节点中的clock-output-names属性xti

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

//注册fixed-rate clock

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

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

//将该clock添加到provider list中，方便后续的查找使用

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

**return** clk**;**

**}**

#### 2.1.1 clk\_register\_fixed\_rate\_with\_accuracy

clk\_register\_fixed\_rate\_with\_accuracy函数代码如下：

struct clk **\***clk\_register\_fixed\_rate\_with\_accuracy**(**struct device **\***dev**,**const char **\***name**,** const char **\***parent\_name**,** unsigned long

flags**,**unsigned long fixed\_rate**,** unsigned long fixed\_accuracy**)**

**{**

struct clk\_hw **\***hw**;**

hw **=** clk\_hw\_register\_fixed\_rate\_with\_accuracy**(**dev**,** name**,** parent\_name**,** flags**,** fixed\_rate**,** fixed\_accuracy**);**

**return** hw**->**clk**;**

**}**

继续追踪。

struct clk\_hw **\***clk\_hw\_register\_fixed\_rate\_with\_accuracy**(**struct device **\***dev**,** const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,** unsigned long fixed\_rate**,** unsigned long fixed\_accuracy**)**

**{**

struct clk\_fixed\_rate **\***fixed**;**

struct clk\_hw **\***hw**;**

struct clk\_init\_data init**;**

int ret**;**

//分配一个clk\_fixed\_rate结构体

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

//初始化clk\_init\_data

init**.**name **=** name**;**

init**.**ops **=** **&**clk\_fixed\_rate\_ops**;**

init**.**flags **=** flags **|** CLK\_IS\_BASIC**;**

init**.**parent\_names **=** **(**parent\_name **?** **&**parent\_name**:** **NULL);**

init**.**num\_parents **=** **(**parent\_name **?** 1 **:** 0**);**

//初始化clk\_fixed\_rate

fixed**->**fixed\_rate **=** fixed\_rate**;**

fixed**->**fixed\_accuracy **=** fixed\_accuracy**;**

fixed**->**hw**.**init **=** **&**init**;**

//注册clk

hw **=** **&**fixed**->**hw**;**

clk\_register**(**dev**,** hw**)**

**return** hw**;**

**}**

这里有三个结构体的关系如下：

struct clk\_fixed\_rate **{**

struct clk\_hw hw**;**

unsigned long fixed\_rate**;**

unsigned long fixed\_accuracy**;**

u8 flags**;**

**};**

struct clk\_hw **{**

struct clk\_core **\***core**;**

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

const struct clk\_init\_data **\***init**;**

**};**

struct clk\_init\_data **{**

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

const struct clk\_ops **\***ops**;**

const char **\*** const **\***parent\_names**;**

u8 num\_parents**;**

unsigned long flags**;**

**};**

内核使用clk\_fixed\_rate来表示fixed-clock。

clk\_fixed\_rate\_ops定义了fixed-clock的操作函数，可以用来获取fixed-clock的频率和accuracy。

#define to\_clk\_fixed\_rate(\_hw) container\_of(\_hw, struct clk\_fixed\_rate, hw)

static unsigned long clk\_fixed\_rate\_recalc\_rate**(**struct clk\_hw **\***hw**,**

unsigned long parent\_rate**)**

**{**

**return** to\_clk\_fixed\_rate**(**hw**)->**fixed\_rate**;**

**}**

static unsigned long clk\_fixed\_rate\_recalc\_accuracy**(**struct clk\_hw **\***hw**,**

unsigned long parent\_accuracy**)**

**{**

**return** to\_clk\_fixed\_rate**(**hw**)->**fixed\_accuracy**;**

**}**

const struct clk\_ops clk\_fixed\_rate\_ops **=** **{**

**.**recalc\_rate **=** clk\_fixed\_rate\_recalc\_rate**,**

**.**recalc\_accuracy **=** clk\_fixed\_rate\_recalc\_accuracy**,**

**};**

最关键的函数就是clk\_register了，clk\_register负责将clock注册到kernel，并返回代表该clock的struct clk指针。

##### 2.1.1.1 clk\_register

该接口接受一个struct clk\_hw指针，该指针包含了将要注册的clock的信息。clk\_hw有两个核心数据结构，如下：

(1) struct clk\_core：这个数据结构是clock device的抽象，每一个实际的硬件clock device（struct clk\_hw）都会对应一个clk\_core，clk子系统负责建立整个抽象的clock tree的树状结构并维护这些数据。

clk子系统有两条全局的链表：

static HLIST\_HEAD**(**clk\_root\_list**);**

static HLIST\_HEAD**(**clk\_orphan\_list**);**

父节点个数为0的话，clock会挂在clk\_root\_list上；如果父节点个数不为0，但是并没有获取到valid的parent，则会挂到clk\_orphan\_list，表示无父节点。如果有valid的parent，则会挂到parent的“children”链表中。

struct clk\_core **{**

const char **\***name**;** //名字

const struct clk\_ops **\***ops**;** //操作函数集

struct clk\_hw **\***hw**;** //对应的clk\_hw

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

struct clk\_core **\***parent**;** //当前对应的parent

const char **\*\***parent\_names**;** //parent\_name名字集合

struct clk\_core **\*\***parents**;** //parent clk\_core结构体集合

u8 num\_parents**;** //有几个parent

u8 new\_parent\_index**;**

unsigned long rate**;** //当前频率

unsigned long req\_rate**;**

unsigned long new\_rate**;** //新设置的clock rate之所要保存在这里，是因为set rate过程中有一些中间计算

struct clk\_core **\***new\_parent**;**

struct clk\_core **\***new\_child**;**

unsigned long flags**;** //flag集合

bool orphan**;** //是否孤儿节点

unsigned int enable\_count**;** //enable的次数

unsigned int prepare\_count**;** //prepare的次数

unsigned long min\_rate**;**

unsigned long max\_rate**;**

unsigned long accuracy**;** //当前的accuracy

int phase**;** //当前的phase

struct hlist\_head children**;** //保存子节点链表头

struct hlist\_node child\_node**;** //加入到别人的子节点

struct hlist\_head clks**;** //保存clk结构体链表头

unsigned int notifier\_count**;** //记录注册到notifier的个数

#ifdef CONFIG\_DEBUG\_FS

struct dentry **\***dentry**;**

struct hlist\_node debug\_node**;**

#endif

struct kref ref**;**

**};**

(2)struct clk：这个数据结构和consumer的访问有关，基本上，每一个user对clock device的访问都会创建一个访问句柄，这个句柄就是clk。不同的user访问同样的clock device的时候，虽然是同一个struct clk\_core实例。但是其访问的句柄（clk）是不一样的。

struct clk **{**

struct clk\_core **\***core**;** //保存对应的clk\_core

const char **\***dev\_id**;**

const char **\***con\_id**;**

unsigned long min\_rate**;**

unsigned long max\_rate**;**

struct hlist\_node clks\_node**;** //链表操作，加入到clk\_core中

**};**

clk\_register源码分析如下：

struct clk **\***clk\_register**(**struct device **\***dev**,** struct clk\_hw **\***hw**)**

**{**

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

struct clk\_core **\***core**;**

//分配clk\_core结构体

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

//分配字符串空间，并内容拷贝到分配的字符串空间中。

core**->**name **=** kstrdup\_const**(**hw**->**init**->**name**,** GFP\_KERNEL**);**

//获取clk\_init\_data中的ops操作函数

core**->**ops **=** hw**->**init**->**ops**;**

**if** **(**dev **&&** dev**->**driver**)**

core**->**owner **=** dev**->**driver**->**owner**;**

//core关联hw

core**->**hw **=** hw**;**

//获取clk\_init\_data中的flags

core**->**flags **=** hw**->**init**->**flags**;**

//获取clk\_init\_data中的number\_parents

core**->**num\_parents **=** hw**->**init**->**num\_parents**;**

core**->**min\_rate **=** 0**;**

core**->**max\_rate **=** ULONG\_MAX**;**

//hw关联core

hw**->**core **=** core**;**

//分配空间存储parents\_names

core**->**parent\_names **=** kcalloc**(**core**->**num\_parents**,** **sizeof(**char **\*),**

GFP\_KERNEL**);**

//拷贝clk\_init\_data中的parent\_names到core中的parent\_names

**for** **(**i **=** 0**;** i **<** core**->**num\_parents**;** i**++)** **{**

core**->**parent\_names**[**i**]** **=** kstrdup\_const**(**hw**->**init**->**parent\_names**[**i**],**GFP\_KERNEL**);**

**}**

//分配存储parents的空间,parents为clk\_core类型

core**->**parents **=** kcalloc**(**core**->**num\_parents**,** **sizeof(\***core**->**parents**),**GFP\_KERNEL**);**

//初始化链表，clks用于存储clk结构体，每次访问device都会获得一个clk句柄，clks保存这些句柄

INIT\_HLIST\_HEAD**(&**core**->**clks**);**

//分配一个clk结构体，并添加到core->clks链表中。

hw**->**clk **=** \_\_clk\_create\_clk**(**hw**,** **NULL,** **NULL);**

//初始化clk\_core

ret **=** \_\_clk\_core\_init**(**core**);**

//最后返回clk结构体

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

**return** hw**->**clk**;**

**}**

核心函数为\_\_clk\_core\_init。

static int \_\_clk\_core\_init**(**struct clk\_core **\***core**)**

**{**

int i**,** ret **=** 0**;**

struct clk\_core **\***orphan**;**

struct hlist\_node **\***tmp2**;**

unsigned long rate**;**

clk\_prepare\_lock**();**

//遍历clk\_root\_list和clk\_orphan\_list查找是否有相同名字的clk

clk\_core\_lookup**(**core**->**name**);**

//如果有改变rate的回调，则recalc\_rate也一定需要，round\_rate determine\_rate必须具备一个

**if** **(**core**->**ops**->**set\_rate **&&!((**core**->**ops**->**round\_rate **||** core**->**ops**->**determine\_rate**)** **&&** core**->**ops**->**recalc\_rate**))** **{**

ret **=** **-**EINVAL**;**

**goto** out**;**

**}**

//有set\_parent就需要有get\_parent

**if** **(**core**->**ops**->**set\_parent **&&** **!**core**->**ops**->**get\_parent**)** **{**

ret **=** **-**EINVAL**;**

**goto** out**;**

**}**

//如果父clk>1，则需要get\_parent

**if** **(**core**->**num\_parents **>** 1 **&&** **!**core**->**ops**->**get\_parent**)** **{**

ret **=** **-**EINVAL**;**

**goto** out**;**

**}**

//set\_rate\_and\_parent则需要有set\_parent和set\_rate

**if** **(**core**->**ops**->**set\_rate\_and\_parent **&&** **!(**core**->**ops**->**set\_parent **&&** core**->**ops**->**set\_rate**))** **{**

ret **=** **-**EINVAL**;**

**goto** out**;**

**}**

//找到parent对应的clk\_core结构体。

core**->**parent **=** \_\_clk\_init\_parent**(**core**);**

//如果有一个parent，则添加到子列表中

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

hlist\_add\_head**(&**core**->**child\_node**,**

**&**core**->**parent**->**children**);**

core**->**orphan **=** core**->**parent**->**orphan**;**

**}**

//number\_parents为0，表示为根节点

**else** **if** **(!**core**->**num\_parents**)** **{**

hlist\_add\_head**(&**core**->**child\_node**,** **&**clk\_root\_list**);**

core**->**orphan **=** **false;**

**}**

//虽然number\_parents > 0但是没找到parent，孤儿节点

**else** **{**

hlist\_add\_head**(&**core**->**child\_node**,** **&**clk\_orphan\_list**);**

core**->**orphan **=** **true;**

**}**

//计算accuracy

**if** **(**core**->**ops**->**recalc\_accuracy**)**

core**->**accuracy **=** core**->**ops**->**recalc\_accuracy**(**core**->**hw**,**

\_\_clk\_get\_accuracy**(**core**->**parent**));**

**else** **if** **(**core**->**parent**)**

core**->**accuracy **=** core**->**parent**->**accuracy**;**

**else**

core**->**accuracy **=** 0**;**

//计算phase

**if** **(**core**->**ops**->**get\_phase**)**

core**->**phase **=** core**->**ops**->**get\_phase**(**core**->**hw**);**

**else**

core**->**phase **=** 0**;**

//计算rate

**if** **(**core**->**ops**->**recalc\_rate**)**

rate **=** core**->**ops**->**recalc\_rate**(**core**->**hw**,**

clk\_core\_get\_rate\_nolock**(**core**->**parent**));**

**else** **if** **(**core**->**parent**)**

rate **=** core**->**parent**->**rate**;**

**else**

rate **=** 0**;**

core**->**rate **=** core**->**req\_rate **=** rate**;**

//遍历clk\_orphan\_list，如果能找到parent节点，则设置parent

hlist\_for\_each\_entry\_safe**(**orphan**,** tmp2**,** **&**clk\_orphan\_list**,** child\_node**)** **{**

struct clk\_core **\***parent **=** \_\_clk\_init\_parent**(**orphan**);**

**if** **(**parent**)** **{**

\_\_clk\_set\_parent\_before**(**orphan**,** parent**);**

\_\_clk\_set\_parent\_after**(**orphan**,** parent**,** **NULL);**

\_\_clk\_recalc\_accuracies**(**orphan**);**

\_\_clk\_recalc\_rates**(**orphan**,** 0**);**

**}**

**}**

//初始化函数调用

**if** **(**core**->**ops**->**init**)**

core**->**ops**->**init**(**core**->**hw**);**

//如果flags有CLK\_IS\_CRITICAL，则需要开启该clock

**if** **(**core**->**flags **&** CLK\_IS\_CRITICAL**)** **{**

unsigned long flags**;**

clk\_core\_prepare**(**core**);**

flags **=** clk\_enable\_lock**();**

clk\_core\_enable**(**core**);**

clk\_enable\_unlock**(**flags**);**

**}**

kref\_init**(&**core**->**ref**);**

out**:**

clk\_prepare\_unlock**();**

**return** ret**;**

**}**

##### 2.1.1.2 of\_clk\_add\_provider

调用of\_clk\_add\_provider接口，将该clock添加到provider list中，方便后续的查找使用。

static LIST\_HEAD**(**of\_clk\_providers**);**

struct of\_clk\_provider **{**

struct list\_head link**;**

struct device\_node **\***node**;**

struct clk **\*(\***get**)(**struct of\_phandle\_args **\***clkspec**,** void **\***data**);**

struct clk\_hw **\*(\***get\_hw**)(**struct of\_phandle\_args **\***clkspec**,** void **\***data**);**

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

**};**

int of\_clk\_add\_provider**(**struct device\_node **\***np**,**

struct clk **\*(\***clk\_src\_get**)(**struct of\_phandle\_args **\***clkspec**,**void **\***data**),** void **\***data**)**

**{**

struct of\_clk\_provider **\***cp**;**

int ret**;**

cp **=** kzalloc**(sizeof(**struct of\_clk\_provider**),** GFP\_KERNEL**);**

//保存device\_node节点

cp**->**node **=** of\_node\_get**(**np**);**

//保存私有数据

cp**->**data **=** data**;**

//根据dts获取clk的方法函数

cp**->**get **=** clk\_src\_get**;**

//将新建的of\_clk\_provider结构体添加到of\_clk\_providers链表

mutex\_lock**(&**of\_clk\_mutex**);**

list\_add**(&**cp**->**link**,** **&**of\_clk\_providers**);**

mutex\_unlock**(&**of\_clk\_mutex**);**

//解析'assigned-{clocks/clock-parents/clock-rates}' properties

ret **=** of\_clk\_set\_defaults**(**np**,** **true);**

**return** ret**;**

**}**

#### 2.1.2 fixed-clock的使用

下面分享一个例子，如何获取fixed-clock的频率。

##### 2.1.2.1 dts添加xti的引用

/\* DTS \*/

device **{**

clocks **=** **<&**xti**>;**

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

**};**

##### 2.1.2.2 代码中对xti的操作

/\* driver \*/

int xxx\_probe**(**struct platform\_device **\***pdev**)**

**{**

struct clk **\***xti\_clk**;**

unsigned long rate**;**

xti\_clk **=** devm\_clk\_get**(&**pdev**->**dev**,** “xti”**);**

**if** **(**IS\_ERR**(**clk**))** **{**

…

**}**

rate **=** clk\_get\_rate**(**xti\_clk**);**

**}**

##### 2.1.2.3 devm\_clk\_get函数分析

devm\_clk\_get获取名字为xti对应的clk结构体。

流程如下：

devm\_clk\_get

clk\_get

\_\_of\_clk\_get\_by\_name

\_\_of\_clk\_get

\_\_of\_clk\_get\_from\_provider

devm\_clk\_get主要作用是调用了clk\_get函数。

struct clk **\***devm\_clk\_get**(**struct device **\***dev**,** const char **\***id**)**

**{**

struct clk **\*\***ptr**,** **\***clk**;**

ptr **=** devres\_alloc**(**devm\_clk\_release**,** **sizeof(\***ptr**),** GFP\_KERNEL**);**

**if** **(!**ptr**)**

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

clk **=** clk\_get**(**dev**,** id**);**

**if** **(!**IS\_ERR**(**clk**))** **{**

**\***ptr **=** clk**;**

devres\_add**(**dev**,** ptr**);**

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

devres\_free**(**ptr**);**

**}**

**return** clk**;**

**}**

clk\_get继续调用\_\_of\_clk\_get\_by\_name。

struct clk **\***clk\_get**(**struct device **\***dev**,** const char **\***con\_id**)**

**{**

//con\_id为clk的名字这里为xti

//dev\_id为使用clk device的名字

const char **\***dev\_id **=** dev **?** dev\_name**(**dev**)** **:** **NULL;**

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

**if** **(**dev**)** **{**

clk **=** \_\_of\_clk\_get\_by\_name**(**dev**->**of\_node**,** dev\_id**,** con\_id**);**

**if** **(!**IS\_ERR**(**clk**)** **||** PTR\_ERR**(**clk**)** **==** **-**EPROBE\_DEFER**)**

**return** clk**;**

**}**

**return** clk\_get\_sys**(**dev\_id**,** con\_id**);**

**}**

\_\_of\_clk\_get\_by\_name会寻找dts中clock-names对应的选项。

static struct clk **\***\_\_of\_clk\_get\_by\_name**(**struct device\_node **\***np**,**

const char **\***dev\_id**,** const char **\***name**)**

**{**

//np为使用clk的device的device\_node

struct clk **\***clk **=** ERR\_PTR**(-**ENOENT**);**

**while** **(**np**)** **{**

int index **=** 0**;**

//查找是否有clock-names为xti的选项，如果有，返回index

**if** **(**name**)**

index **=** of\_property\_match\_string**(**np**,** "clock-names"**,** name**);**

clk **=** \_\_of\_clk\_get**(**np**,** index**,** dev\_id**,** name**);**

**if** **(!**IS\_ERR**(**clk**))** **{**

**break;**

**}** **else** **if** **(**name **&&** index **>=** 0**)** **{**

**if** **(**PTR\_ERR**(**clk**)** **!=** **-**EPROBE\_DEFER**)**

pr\_err**(**"ERROR: could not get clock %s:%s(%i)\n"**,**

np**->**full\_name**,** name **?** name **:** ""**,** index**);**

**return** clk**;**

**}**

np **=** np**->**parent**;**

**if** **(**np **&&** **!**of\_get\_property**(**np**,** "clock-ranges"**,** **NULL))**

**break;**

**}**

**return** clk**;**

**}**

\_\_of\_clk\_get会解析clocks的中的参数，并从of\_clk\_providers链表中找到对应clk。

static struct clk **\***\_\_of\_clk\_get**(**struct device\_node **\***np**,** int index**,**

const char **\***dev\_id**,** const char **\***con\_id**)**

**{**

struct of\_phandle\_args clkspec**;**

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

int rc**;**

//分析clocks index选项，解析<&xti>，clock-cells这里为0

//解析出来的结果存储在clkspec中。

rc **=** of\_parse\_phandle\_with\_args**(**np**,** "clocks"**,** "#clock-cells"**,** index**,&**clkspec**);**

//从of\_clk\_providers链表中得到clk信息

clk **=** \_\_of\_clk\_get\_from\_provider**(&**clkspec**,** dev\_id**,** con\_id**);**

of\_node\_put**(**clkspec**.**np**);**

**return** clk**;**

**}**

这里需要分别分析两个重要的函数调用of\_parse\_phandle\_with\_args和\_\_of\_clk\_get\_from\_provider。

###### 2.1.2.3.1 of\_parse\_phandle\_with\_args

of\_parse\_phandle\_with\_args的使用范例如下，dts中有如下定义：

phandle1**:** node1 **{**

#list-cells = <2>;

**}**

phandle2**:** node2 **{**

#list-cells = <1>;

**}**

node3 **{**

list **=** **<&**phandle1 1 2 **&**phandle2 3**>;**

**}**

如果要获取node2的device\_node和参数，调用如下：

of\_parse\_phandle\_with\_args**(**node3**,** "list"**,** "#list-cells"**,** 1**,** **&**args**);**

同样的，2440中xti对应的dts代码：

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

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

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

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

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

#clock-cells = <0>;

**};**

device **{**

clocks **=** **<&**xti**>;**

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

**};**

那么调用如下代码将分析的结果存入clkspec中。

of\_parse\_phandle\_with\_args**(**np**,** "clocks"**,** "#clock-cells"**,** index**,&**clkspec**);**

clkspec对应的结构体为：

struct of\_phandle\_args **{**

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

int args\_count**;**

uint32\_t args**[**MAX\_PHANDLE\_ARGS**];**

**};**

那么最后结果是clkspec 的np中存储了xti对应的device\_node，args\_count为0，args为空。

###### 2.1.2.3.2 \_\_of\_clk\_get\_from\_provider

上面的步骤已经获取到了xti对应的device\_node，那么下面就需要获取xti对应的clk结构体了。

struct clk **\***\_\_of\_clk\_get\_from\_provider**(**struct of\_phandle\_args **\***clkspec**,** const char **\***dev\_id**,** const char **\***con\_id**)**

**{**

struct of\_clk\_provider **\***provider**;**

struct clk **\***clk **=** ERR\_PTR**(-**EPROBE\_DEFER**);**

struct clk\_hw **\***hw**;**

**if** **(!**clkspec**)**

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

mutex\_lock**(&**of\_clk\_mutex**);**

//遍历of\_clk\_providers链表

list\_for\_each\_entry**(**provider**,** **&**of\_clk\_providers**,** link**)** **{**

//比较device\_node是否一致

**if** **(**provider**->**node **==** clkspec**->**np**)** **{**

//如果一致，得到对应clk\_hw，这里需要调用of\_clk\_src\_simple\_get

hw **=** \_\_of\_clk\_get\_hw\_from\_provider**(**provider**,** clkspec**);**

//创建clk\_hw对应的clk

clk **=** \_\_clk\_create\_clk**(**hw**,** dev\_id**,** con\_id**);**

**}**

**if** **(!**IS\_ERR**(**clk**))** **{**

**if** **(!**\_\_clk\_get**(**clk**))** **{**

\_\_clk\_free\_clk**(**clk**);**

clk **=** ERR\_PTR**(-**ENOENT**);**

**}**

**break;**

**}**

**}**

mutex\_unlock**(&**of\_clk\_mutex**);**

**return** clk**;**

**}**

\_\_of\_clk\_get\_hw\_from\_provider会直接调用of\_clk\_src\_simple\_get函数。

static struct clk\_hw **\*** \_\_of\_clk\_get\_hw\_from\_provider**(**struct of\_clk\_provider **\***provider**,** struct of\_phandle\_args **\***clkspec**)**

**{**

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

//如果定义了get\_hw，则调用，这里未定义。

**if** **(**provider**->**get\_hw**)**

**return** provider**->**get\_hw**(**clkspec**,** provider**->**data**);**

//这里会调用of\_clk\_src\_simple\_get

clk **=** provider**->**get**(**clkspec**,** provider**->**data**);**

//clk->core->hw

**return** \_\_clk\_get\_hw**(**clk**);**

**}**

of\_clk\_src\_simple\_get的定义如下，也就是直接返回data数据，这里的data传入的就是xti对应的clk。

struct clk **\***of\_clk\_src\_simple\_get**(**struct of\_phandle\_args **\***clkspec**,** void **\***data**)**

**{**

**return** data**;**

**}**

##### 2.1.2.4 clk\_get\_rate

上面已经通过devm\_clk\_get得到了xti对应的clk结构体，那么可以调用clk\_get\_rate得到xti的频率了。

unsigned long clk\_get\_rate**(**struct clk **\***clk**)**

**{**

**return** clk\_core\_get\_rate**(**clk**->**core**);**

**}**

static unsigned long clk\_core\_get\_rate**(**struct clk\_core **\***core**)**

**{**

unsigned long rate**;**

clk\_prepare\_lock**();**

//CLK\_GET\_RATE\_NOCACHE表示需要重新计算频率recalc\_rate

**if** **(**core **&&** **(**core**->**flags **&** CLK\_GET\_RATE\_NOCACHE**))**

\_\_clk\_recalc\_rates**(**core**,** 0**);**

//获取core->rate

rate **=** clk\_core\_get\_rate\_nolock**(**core**);**

clk\_prepare\_unlock**();**

**return** rate**;**

**}**

### 2.2 s3c2440-clock

2440其他所有的clk dts。

clocks**:** clock**-**controller@4c000000 **{**

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

reg **=** **<**0x4c000000 0x20**>;**

#clock-cells = <1>;

**};**

对应的代码如下：

static void \_\_init s3c2440\_clk\_init**(**struct device\_node **\***np**)**

**{**

s3c2410\_common\_clk\_init**(**np**,** 0**,** S3C2440**,** 0**);**

**}**

CLK\_OF\_DECLARE**(**s3c2440\_clk**,** "samsung,s3c2440-clock"**,** s3c2440\_clk\_init**);**

s3c2410\_common\_clk\_init将所有的clock添加到一个device\_node上。

void \_\_init s3c2410\_common\_clk\_init**(**struct device\_node **\***np**,** unsigned long xti\_f**,** int current\_soc**,** void \_\_iomem **\***base**)**

**{**

//直接通过设备节点进行设备内存区间的ioremap，index是内存段索引。

//reg\_base为虚拟寄存器起始地址

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

//创建并初始化samsung\_clk\_provider结构体

struct samsung\_clk\_provider **\***ctx**;**

ctx **=** samsung\_clk\_init**(**np**,** reg\_base**,** NR\_CLKS**);**

//找到xti对应的clk结构体，并获取rate

**if** **(**\_get\_rate**(**"xti"**)** **==** 12 **\*** MHZ**)** **{**

//如果mpll要设置为400M，mdiv pdiv sdiv的取值

s3c244x\_common\_plls**[**mpll**].**rate\_table **=** pll\_s3c244x\_12mhz\_tbl**;**

//如果upll要设置为48M，mdiv pdiv sdiv的取值

s3c244x\_common\_plls**[**upll**].**rate\_table **=** pll\_s3c2410\_12mhz\_tbl**;**

**}**

//注册pll(mpll upll)，并添加到ctx中，pll为自定义的clock

samsung\_clk\_register\_pll**(**ctx**,** s3c244x\_common\_plls**,** ARRAY\_SIZE**(**s3c244x\_common\_plls**),** reg\_base**);**

//注册fclk，并添加到ctx中,fclk为mux类型，可以选择时钟源

samsung\_clk\_register\_mux**(**ctx**,** s3c2410\_common\_muxes**,** ARRAY\_SIZE**(**s3c2410\_common\_muxes**));**

//注册pclk，并添加到ctx中，pclk为divider类型，可以通过分频改变频率

samsung\_clk\_register\_div**(**ctx**,** s3c2410\_common\_dividers**,**ARRAY\_SIZE**(**s3c2410\_common\_dividers**));**

//注册spi i2s i2c等clk,这些clk类型为gate，只能控制开关

samsung\_clk\_register\_gate**(**ctx**,** s3c2410\_common\_gates**,**ARRAY\_SIZE**(**s3c2410\_common\_gates**));**

//还有其他一些clk的注册，这里未全部列出来,后面有讲解

**...**

samsung\_clk\_register\_alias**(**ctx**,** s3c2410\_common\_aliases**,** ARRAY\_SIZE**(**s3c2410\_common\_aliases**));**

//和sleep模式相关的设置，保存寄存器值，方便恢复

s3c2410\_clk\_sleep\_init**();**

//将clocks对应的device\_node添加到of\_clk\_providers中。

samsung\_clk\_of\_add\_provider**(**np**,** ctx**);**

**}**

samsung\_clk\_provider结构体的定义如下：

struct samsung\_clk\_provider **{**

//存储clk寄存器起始地址

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

//保存clk列表

struct clk\_onecell\_data clk\_data**;**

spinlock\_t lock**;**

**};**

初始化代码如下：

struct samsung\_clk\_provider **\***\_\_init samsung\_clk\_init**(**struct device\_node **\***np**,**

void \_\_iomem **\***base**,** unsigned long nr\_clks**)**

**{**

struct samsung\_clk\_provider **\***ctx**;**

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

int i**;**

//分配samsung\_clk\_provider结构体

ctx **=** kzalloc**(sizeof(**struct samsung\_clk\_provider**),** GFP\_KERNEL**);**

//分配clk\_table空间

clk\_table **=** kcalloc**(**nr\_clks**,** **sizeof(**struct clk **\*),** GFP\_KERNEL**);**

//初始化clk\_table

**for** **(**i **=** 0**;** i **<** nr\_clks**;** **++**i**)**

clk\_table**[**i**]** **=** ERR\_PTR**(-**ENOENT**);**

//初始化samsung\_clk\_provider

ctx**->**reg\_base **=** base**;**

ctx**->**clk\_data**.**clks **=** clk\_table**;**

ctx**->**clk\_data**.**clk\_num **=** nr\_clks**;**

spin\_lock\_init**(&**ctx**->**lock**);**

**return** ctx**;**

**}**

#### 2.2.1 2440 clock简介

S3C2440有两个PLL（phase locked loop）一个是MPLL，一个是UPLL。MPLL用于CPU及其他外围器件，UPLL用于USB**.**

MPLL用来产生FCLK**,**HCLK**,**PCLK的高频工作时钟**,**UPLL用来为USB提供工作频率

FCLK主要为ARM920T内核提供工作频率

HCLK主要为S3C2440 AHB总线（Advanced High performance Bus）上挂接硬件提供工作频率，AHB总线主要挂接有内存，NAND，LCD控制器等硬件

PCLK主要为APB总线提供工作频率，APB总线主要挂接UART串口，Watchdog等硬件控制器。

#### 2.2.2 samsung\_clk\_register\_pll

samsung\_clk\_register\_pll用来注册PLL，这里有MPLL和UPLL两个PLL。

samsung的pll有一个通用的结构体。

struct samsung\_clk\_pll **{**

struct clk\_hw hw**;**

void \_\_iomem **\***lock\_reg**;**

void \_\_iomem **\***con\_reg**;**

enum samsung\_pll\_type type**;**

unsigned int rate\_count**;**

const struct samsung\_pll\_rate\_table **\***rate\_table**;**

**};**

其中rate\_table定义了需要设置的频率对应的寄存器设置值。

mpll和upll的定义如下：

static struct samsung\_pll\_clock s3c244x\_common\_plls**[]** \_\_initdata **=** **{**

**[**mpll**]** **=** PLL**(**pll\_s3c2440\_mpll**,** MPLL**,** "mpll"**,** "xti"**,**LOCKTIME**,** MPLLCON**,** **NULL),**

**[**upll**]** **=** PLL**(**pll\_s3c2410\_upll**,** UPLL**,** "upll"**,** "xti"**,**LOCKTIME**,** UPLLCON**,** **NULL),**

**};**

pll\_s3c2440\_mpll type

MPLL**/**UPLL id

"mpll"**/**"upll" name

"xti" parent\_name

MPLLCON**/**UPLLCON 寄存器偏移

rate\_table为pll\_s3c244x\_12mhz\_tbl**/**pll\_s3c2410\_12mhz\_tbl

pll注册代码如下：

void \_\_init samsung\_clk\_register\_pll**(**struct samsung\_clk\_provider **\***ctx**,** const struct samsung\_pll\_clock **\***pll\_list**,** unsigned int nr\_pll**,**

void \_\_iomem **\***base**)**

**{**

int cnt**;**

**for** **(**cnt **=** 0**;** cnt **<** nr\_pll**;** cnt**++)**

\_samsung\_clk\_register\_pll**(**ctx**,** **&**pll\_list**[**cnt**],** base**);**

**}**

实际的注册代码为：

static void \_\_init \_samsung\_clk\_register\_pll**(**struct samsung\_clk\_provider **\***ctx**,** const struct samsung\_pll\_clock **\***pll\_clk**,**void \_\_iomem **\***base**)**

**{**

//创建samsung\_clk\_pll,用于保存mpll upll操作的相关参数和clk\_hw。

struct samsung\_clk\_pll **\***pll**;**

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

struct clk\_init\_data init**;**

//"mpll"或"upll"

init**.**name **=** pll\_clk**->**name**;**

init**.**flags **=** pll\_clk**->**flags**;**

//parent\_names为"xti"

init**.**parent\_names **=** **&**pll\_clk**->**parent\_name**;**

init**.**num\_parents **=** 1**;**

**if** **(**pll\_clk**->**rate\_table**)**

**{**

//rate\_table里面有多少个

**for** **(**len **=** 0**;** pll\_clk**->**rate\_table**[**len**].**rate **!=** 0**;** **)**

len**++;**

pll**->**rate\_count **=** len**;**

//分配存储rate\_table的空间

pll**->**rate\_table **=** kmemdup**(**pll\_clk**->**rate\_table**,**

pll**->**rate\_count **\*** **sizeof(**struct samsung\_pll\_rate\_table**),**

GFP\_KERNEL**);**

**}**

//设置操作函数，根据rate\_table设置mpll upll频率等操作。

**switch** **(**pll\_clk**->**type**)**

**{**

**case** pll\_s3c2440\_mpll**:**

init**.**ops **=** **&**samsung\_s3c2440\_mpll\_clk\_ops**;**

**break;**

**case** pll\_s3c2410\_upll**:**

init**.**ops **=** **&**samsung\_s3c2410\_upll\_clk\_ops**;**

**break;**

**}**

//clk\_hw的初始化代码

pll**->**hw**.**init **=** **&**init**;**

pll**->**type **=** pll\_clk**->**type**;**

pll**->**lock\_reg **=** base **+** pll\_clk**->**lock\_offset**;**

//mpll upll的操作寄存器

pll**->**con\_reg **=** base **+** pll\_clk**->**con\_offset**;**

//注册clk

clk **=** clk\_register**(NULL,** **&**pll**->**hw**);**

//将clk添加到ctx的clk table中，方便dts查找

samsung\_clk\_add\_lookup**(**ctx**,** clk**,** pll\_clk**->**id**);**

//创建clk\_lookup，并注册到clocks全局链表

clk\_register\_clkdev**(**clk**,** pll\_clk**->**alias**,** pll\_clk**->**dev\_name**);**

**}**

mpll upll的操作函数集为：

static const struct clk\_ops samsung\_s3c2440\_mpll\_clk\_ops **=** **{**

**.**recalc\_rate **=** samsung\_s3c2440\_mpll\_recalc\_rate**,**

**.**enable **=** samsung\_s3c2410\_mpll\_enable**,**

**.**disable **=** samsung\_s3c2410\_mpll\_disable**,**

**.**round\_rate **=** samsung\_pll\_round\_rate**,**

**.**set\_rate **=** samsung\_s3c2410\_pll\_set\_rate**,**

**};**

static const struct clk\_ops samsung\_s3c2410\_upll\_clk\_ops **=** **{**

**.**recalc\_rate **=** samsung\_s3c2410\_pll\_recalc\_rate**,**

**.**enable **=** samsung\_s3c2410\_upll\_enable**,**

**.**disable **=** samsung\_s3c2410\_upll\_disable**,**

**.**round\_rate **=** samsung\_pll\_round\_rate**,**

**.**set\_rate **=** samsung\_s3c2410\_pll\_set\_rate**,**

**};**

这里对mpll的set rate操作进行分析：

static int samsung\_s3c2410\_pll\_set\_rate**(**struct clk\_hw **\***hw**,** unsigned long drate**,** unsigned long prate**)**

**{**

//根据clk\_hw找到samsung\_clk\_pll结构体

struct samsung\_clk\_pll **\***pll **=** to\_clk\_pll**(**hw**);**

const struct samsung\_pll\_rate\_table **\***rate**;**

u32 tmp**;**

//从samsung\_clk\_pll中的rate table中找到对应频率的table

rate **=** samsung\_get\_pll\_settings**(**pll**,** drate**);**

//读出对应的寄存器值

tmp **=** readl\_relaxed**(**pll**->**con\_reg**);**

//改变mdiv pdiv sdiv

tmp **&=** **~((**PLLS3C2410\_MDIV\_MASK **<<** PLLS3C2410\_MDIV\_SHIFT**)** **|**

**(**PLLS3C2410\_PDIV\_MASK **<<** PLLS3C2410\_PDIV\_SHIFT**)** **|**

**(**PLLS3C2410\_SDIV\_MASK **<<** PLLS3C2410\_SDIV\_SHIFT**));**

tmp **|=** **(**rate**->**mdiv **<<** PLLS3C2410\_MDIV\_SHIFT**)** **|**

**(**rate**->**pdiv **<<** PLLS3C2410\_PDIV\_SHIFT**)** **|**

**(**rate**->**sdiv **<<** PLLS3C2410\_SDIV\_SHIFT**);**

writel\_relaxed**(**tmp**,** pll**->**con\_reg**);**

udelay**(**300**);**

**return** 0**;**

**}**

#### 2.2.3 samsung\_clk\_register\_mux

samsung\_clk\_register\_mux用来注册mux类型的时钟，以fclk为例，fclk可以有两个源可以选择，在slow mode中，fclk = xti/divider ratio。在normal mode，fclk = mpll。所以fclk为mux型。

fclk的定义如下：

PNAME**(**fclk\_p**)** **=** **{** "mpll"**,** "div\_slow" **};**

struct samsung\_mux\_clock s3c2410\_common\_muxes**[]** \_\_initdata **=** **{**

MUX**(**FCLK**,** "fclk"**,** fclk\_p**,** CLKSLOW**,** 4**,** 1**),**

**};**

FCLK id

"flck" name

"mpll"**,** "div\_slow" parent\_name

CLKSLOW fclk操作寄存器

4 fclk操作寄存器偏移

1 fclk操作寄存器宽度

samsung\_clk\_register\_mux代码如下：

void \_\_init samsung\_clk\_register\_mux**(**struct samsung\_clk\_provider **\***ctx**,** const struct samsung\_mux\_clock **\***list**,** unsigned int nr\_clk**)**

**{**

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

unsigned int idx**,** ret**;**

//注册mux clock

**for** **(**idx **=** 0**;** idx **<** nr\_clk**;** idx**++,** list**++)** **{**

clk **=** clk\_register\_mux**(NULL,** list**->**name**,** list**->**parent\_names**,**

list**->**num\_parents**,** list**->**flags**,**

ctx**->**reg\_base **+** list**->**offset**,**

list**->**shift**,** list**->**width**,** list**->**mux\_flags**,** **&**ctx**->**lock**);**

//将mux clock添加到ctx中

samsung\_clk\_add\_lookup**(**ctx**,** clk**,** list**->**id**);**

**if** **(**list**->**alias**)** **{**

ret **=** clk\_register\_clkdev**(**clk**,** list**->**alias**,**list**->**dev\_name**);**

**}**

**}**

**}**

注册mux clk函数clk\_register\_mux代码如下：

struct clk\_hw **\***clk\_hw\_register\_mux\_table**(**struct device **\***dev**,** const char **\***name**,** const char **\*** const **\***parent\_names**,** u8 num\_parents**,**

unsigned long flags**,** void \_\_iomem **\***reg**,** u8 shift**,** u32 mask**,**

u8 clk\_mux\_flags**,** u32 **\***table**,** spinlock\_t **\***lock**)**

**{**

struct clk\_mux **\***mux**;**

struct clk\_hw **\***hw**;**

struct clk\_init\_data init**;**

u8 width **=** 0**;**

int ret**;**

**if** **(**clk\_mux\_flags **&** CLK\_MUX\_HIWORD\_MASK**)** **{**

width **=** fls**(**mask**)** **-** ffs**(**mask**)** **+** 1**;**

**if** **(**width **+** shift **>** 16**)** **{**

pr\_err**(**"mux value exceeds LOWORD field\n"**);**

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

**}**

**}**

mux **=** kzalloc**(sizeof(**struct clk\_mux**),** GFP\_KERNEL**);**

init**.**name **=** name**;**

**if** **(**clk\_mux\_flags **&** CLK\_MUX\_READ\_ONLY**)**

init**.**ops **=** **&**clk\_mux\_ro\_ops**;**

**else**

init**.**ops **=** **&**clk\_mux\_ops**;**

init**.**flags **=** flags **|** CLK\_IS\_BASIC**;**

init**.**parent\_names **=** parent\_names**;**

init**.**num\_parents **=** num\_parents**;**

mux**->**reg **=** reg**;**

mux**->**shift **=** shift**;**

mux**->**mask **=** mask**;**

mux**->**flags **=** clk\_mux\_flags**;**

mux**->**lock **=** lock**;**

mux**->**table **=** table**;**

mux**->**hw**.**init **=** **&**init**;**

hw **=** **&**mux**->**hw**;**

ret **=** clk\_hw\_register**(**dev**,** hw**);**

**return** hw**;**

**}**

和注册fixed-clock的代码很类似，定义了一个clk\_mux辅助mux的操作。

struct clk\_mux **{**

struct clk\_hw hw**;**

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

u32 **\***table**;**

u32 mask**;**

u8 shift**;**

u8 flags**;**

spinlock\_t **\***lock**;**

**};**

那么mux有哪些关键的操作函数呢？

const struct clk\_ops clk\_mux\_ops **=** **{**

**.**get\_parent **=** clk\_mux\_get\_parent**,**

**.**set\_parent **=** clk\_mux\_set\_parent**,**

**.**determine\_rate **=** \_\_clk\_mux\_determine\_rate**,**

**};**

mux操作有获得parent，设置parent，根据频率选择parent三种操作。

get\_parent从寄存器中读取value。

static u8 clk\_mux\_get\_parent**(**struct clk\_hw **\***hw**)**

**{**

//得到clk\_mux结构体

struct clk\_mux **\***mux **=** to\_clk\_mux**(**hw**);**

//获得parent的数量

int num\_parents **=** clk\_hw\_get\_num\_parents**(**hw**);**

//读取对应的寄存器，得到parent的index

u32 val**;**

val **=** clk\_readl**(**mux**->**reg**)** **>>** mux**->**shift**;**

val **&=** mux**->**mask**;**

**return** val**;**

**}**

set\_parent则是根据index设置对应的寄存器，代码不贴出来了。

determine\_rate作用为提供一个频率时，找到最合适的parent。当set\_rate时，如果设置的频率并不符合要求，则会调用round\_rate或者determine\_rate得到近似的频率或者parent。

#### 2.2.4 samsung\_clk\_register\_div

以PCLK为例，PCLK为HCLK分频而来。所以PCLK属于div。

struct samsung\_div\_clock s3c2410\_common\_dividers**[]** \_\_initdata **=** **{**

DIV**(**PCLK**,** "pclk"**,** "hclk"**,** CLKDIVN**,** 0**,** 1**),**

**};**

PCKL id

"pclk" name

"hclk" parent\_name

CLKDIVN 分配寄存器

0 偏移

1 宽度

samsung\_clk\_register\_div代码如下：

void \_\_init samsung\_clk\_register\_div**(**struct samsung\_clk\_provider **\***ctx**,** const struct samsung\_div\_clock **\***list**,** unsigned int nr\_clk**)**

**{**

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

unsigned int idx**,** ret**;**

**for** **(**idx **=** 0**;** idx **<** nr\_clk**;** idx**++,** list**++)** **{**

**if** **(**list**->**table**)**

clk **=** clk\_register\_divider\_table**(NULL,** list**->**name**,**

list**->**parent\_name**,** list**->**flags**,**

ctx**->**reg\_base **+** list**->**offset**,**

list**->**shift**,** list**->**width**,** list**->**div\_flags**,**

list**->**table**,** **&**ctx**->**lock**);**

**else**

clk **=** clk\_register\_divider**(NULL,** list**->**name**,**

list**->**parent\_name**,** list**->**flags**,**

ctx**->**reg\_base **+** list**->**offset**,** list**->**shift**,**

list**->**width**,** list**->**div\_flags**,** **&**ctx**->**lock**);**

samsung\_clk\_add\_lookup**(**ctx**,** clk**,** list**->**id**);**

**if** **(**list**->**alias**)** **{**

ret **=** clk\_register\_clkdev**(**clk**,** list**->**alias**,**

list**->**dev\_name**);**

**}**

**}**

**}**

和注册mux clock代码类似，主要分析clk\_register\_divider。

struct clk **\***clk\_register\_divider**(**struct device **\***dev**,** const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,** void \_\_iomem **\***reg**,** u8 shift**,** u8 width**,** u8 clk\_divider\_flags**,** spinlock\_t **\***lock**)**

**{**

struct clk\_hw **\***hw**;**

hw **=** \_register\_divider**(**dev**,** name**,** parent\_name**,** flags**,** reg**,** shift**,**

width**,** clk\_divider\_flags**,** **NULL,** lock**);**

**return** hw**->**clk**;**

**}**

继续分析\_register\_divider。

static struct clk\_hw **\***\_register\_divider**(**struct device **\***dev**,** const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,**void \_\_iomem

**\***reg**,** u8 shift**,** u8 width**,**u8 clk\_divider\_flags**,** const struct

clk\_div\_table **\***table**,**spinlock\_t **\***lock**)**

**{**

struct clk\_divider **\***div**;**

struct clk\_hw **\***hw**;**

struct clk\_init\_data init**;**

int ret**;**

**if** **(**clk\_divider\_flags **&** CLK\_DIVIDER\_HIWORD\_MASK**)** **{**

**if** **(**width **+** shift **>** 16**)** **{**

pr\_warn**(**"divider value exceeds LOWORD field\n"**);**

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

**}**

**}**

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

**if** **(!**div**)**

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

init**.**name **=** name**;**

**if** **(**clk\_divider\_flags **&** CLK\_DIVIDER\_READ\_ONLY**)**

init**.**ops **=** **&**clk\_divider\_ro\_ops**;**

**else**

init**.**ops **=** **&**clk\_divider\_ops**;**

init**.**flags **=** flags **|** CLK\_IS\_BASIC**;**

init**.**parent\_names **=** **(**parent\_name **?** **&**parent\_name**:** **NULL);**

init**.**num\_parents **=** **(**parent\_name **?** 1 **:** 0**);**

/\* struct clk\_divider assignments \*/

div**->**reg **=** reg**;**

div**->**shift **=** shift**;**

div**->**width **=** width**;**

div**->**flags **=** clk\_divider\_flags**;**

div**->**lock **=** lock**;**

div**->**hw**.**init **=** **&**init**;**

div**->**table **=** table**;**

/\* register the clock \*/

hw **=** **&**div**->**hw**;**

ret **=** clk\_hw\_register**(**dev**,** hw**);**

**return** hw**;**

**}**

这些代码看起来已经很熟悉了，使用clk\_divider结构体来保存divider clock对应的参数，这里主要分析clk\_divider\_ops。

const struct clk\_ops clk\_divider\_ops **=** **{**

**.**recalc\_rate **=** clk\_divider\_recalc\_rate**,**

**.**round\_rate **=** clk\_divider\_round\_rate**,**

**.**set\_rate **=** clk\_divider\_set\_rate**,**

**};**

clk\_divider\_set\_rate流程如下：

static int clk\_divider\_set\_rate**(**struct clk\_hw **\***hw**,** unsigned long rate**,** unsigned long parent\_rate**)**

**{**

//获取clk\_divider结构体

struct clk\_divider **\***divider **=** to\_clk\_divider**(**hw**);**

unsigned int value**;**

unsigned long flags **=** 0**;**

u32 val**;**

//根据rate parent\_rate计算需要分频的值

value **=** divider\_get\_val**(**rate**,** parent\_rate**,** divider**->**table**,** divider**->**width**,** divider**->**flags**);**

spin\_lock\_irqsave**(**divider**->**lock**,** flags**);**

//读出分频寄存器

val **=** clk\_readl**(**divider**->**reg**);**

val **&=** **~(**div\_mask**(**divider**->**width**)** **<<** divider**->**shift**);**

//写入分频寄存器

val **|=** value **<<** divider**->**shift**;**

clk\_writel**(**val**,** divider**->**reg**);**

spin\_unlock\_irqrestore**(**divider**->**lock**,** flags**);**

**return** 0**;**

**}**

计算近似频率的函数clk\_divider\_round\_rate代码如下：

static long clk\_divider\_round\_rate**(**struct clk\_hw **\***hw**,** unsigned long rate**,** unsigned long **\***prate**)**

**{**

struct clk\_divider **\***divider **=** to\_clk\_divider**(**hw**);**

**return** divider\_round\_rate**(**hw**,** rate**,** prate**,** divider**->**table**,** divider**->**width**,** divider**->**flags**);**

**}**

long divider\_round\_rate**(**struct clk\_hw **\***hw**,** unsigned long rate**,** unsigned long **\***prate**,** const struct clk\_div\_table **\***table**,** u8 width**,**

unsigned long flags**)**

**{**

int div**;**

//根据rate, parent\_rate talbe width计算出最合适的分频系数

div **=** clk\_divider\_bestdiv**(**hw**,** rate**,** prate**,** table**,** width**,** flags**);**

//根据parent div计算出接近的rate

**return** DIV\_ROUND\_UP\_ULL**((**u64**)\***prate**,** div**);**

**}**

获取当前频率函数clk\_divider\_recalc\_rate代码如下：

static unsigned long clk\_divider\_recalc\_rate**(**struct clk\_hw **\***hw**,** unsigned long parent\_rate**)**

**{**

struct clk\_divider **\***divider **=** to\_clk\_divider**(**hw**);**

unsigned int val**;**

//读出寄存器中的分频系数。

val **=** clk\_readl**(**divider**->**reg**)** **>>** divider**->**shift**;**

val **&=** div\_mask**(**divider**->**width**);**

//根据parent\_rate 分频系数计算rate

**return** divider\_recalc\_rate**(**hw**,** parent\_rate**,** val**,** divider**->**table**,**

divider**->**flags**);**

**}**

#### 2.2.5 samsung\_clk\_register\_gate

gate表示该clock能开关，以PCLK\_SPI为例。

struct samsung\_gate\_clock s3c2410\_common\_gates**[]** \_\_initdata **=** **{**

GATE**(**PCLK\_SPI**,** "spi"**,** "pclk"**,** CLKCON**,** 18**,** 0**,** 0**),**

**};**

PCLK\_SPI id

"spi" name

"pclk" parent\_name

CLKCON 操作寄存器

18 offset

samsung\_clk\_register\_gate代码如下：

void \_\_init samsung\_clk\_register\_gate**(**struct samsung\_clk\_provider **\***ctx**,** const struct samsung\_gate\_clock **\***list**,** unsigned int nr\_clk**)**

**{**

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

unsigned int idx**,** ret**;**

**for** **(**idx **=** 0**;** idx **<** nr\_clk**;** idx**++,** list**++)** **{**

clk **=** clk\_register\_gate**(NULL,** list**->**name**,** list**->**parent\_name**,**

list**->**flags**,** ctx**->**reg\_base **+** list**->**offset**,**

list**->**bit\_idx**,** list**->**gate\_flags**,** **&**ctx**->**lock**);**

**if** **(**list**->**alias**)** **{**

ret **=** clk\_register\_clkdev**(**clk**,** list**->**alias**,**

list**->**dev\_name**);**

**}**

samsung\_clk\_add\_lookup**(**ctx**,** clk**,** list**->**id**);**

**}**

**}**

继续来分析clk\_register\_gate。

struct clk **\***clk\_register\_gate**(**struct device **\***dev**,** const char **\***name**,**

const char **\***parent\_name**,** unsigned long flags**,**

void \_\_iomem **\***reg**,** u8 bit\_idx**,**

u8 clk\_gate\_flags**,** spinlock\_t **\***lock**)**

**{**

struct clk\_hw **\***hw**;**

hw **=** clk\_hw\_register\_gate**(**dev**,** name**,** parent\_name**,** flags**,** reg**,**

bit\_idx**,** clk\_gate\_flags**,** lock**);**

**return** hw**->**clk**;**

**}**

struct clk\_hw **\***clk\_hw\_register\_gate**(**struct device **\***dev**,** const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,** void \_\_iomem **\***reg**,** u8 bit\_idx**,** u8 clk\_gate\_flags**,** spinlock\_t **\***lock**)**

**{**

struct clk\_gate **\***gate**;**

struct clk\_hw **\***hw**;**

struct clk\_init\_data init**;**

int ret**;**

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

init**.**name **=** name**;**

init**.**ops **=** **&**clk\_gate\_ops**;**

init**.**flags **=** flags **|** CLK\_IS\_BASIC**;**

init**.**parent\_names **=** **(**parent\_name **?** **&**parent\_name**:** **NULL);**

init**.**num\_parents **=** **(**parent\_name **?** 1 **:** 0**);**

gate**->**reg **=** reg**;**

gate**->**bit\_idx **=** bit\_idx**;**

gate**->**flags **=** clk\_gate\_flags**;**

gate**->**lock **=** lock**;**

gate**->**hw**.**init **=** **&**init**;**

hw **=** **&**gate**->**hw**;**

ret **=** clk\_hw\_register**(**dev**,** hw**);**

**return** hw**;**

**}**

同样的分析方法，gate型的clk定义了clk\_gate结构体。下面看一下clk\_gate\_ops。

const struct clk\_ops clk\_gate\_ops **=** **{**

**.**enable **=** clk\_gate\_enable**,**

**.**disable **=** clk\_gate\_disable**,**

**.**is\_enabled **=** clk\_gate\_is\_enabled**,**

**};**

这些操作就比较简单的了，基本就是读写对应寄存器对应的bit位，这里不再详细分析。

#### 2.2.6 samsung\_clk\_register\_fixed\_factor

fixed\_factor固定了分频和倍频系数，这里以div\_hclk\_2为例进行分析。

struct samsung\_fixed\_factor\_clock s3c244x\_common\_ffactor**[]** \_\_initdata **=** **{**

FFACTOR**(**0**,** "div\_hclk\_2"**,** "fclk"**,** 1**,** 2**,** 0**),**

**};**

div\_hclk\_2 **=** fclk**\***1**/**2

clk\_hw\_register\_fixed\_factor代码如下：

struct clk\_hw **\***clk\_hw\_register\_fixed\_factor**(**struct device **\***dev**,**

const char **\***name**,** const char **\***parent\_name**,** unsigned long flags**,**

unsigned int mult**,** unsigned int div**)**

**{**

struct clk\_fixed\_factor **\***fix**;**

struct clk\_init\_data init**;**

struct clk\_hw **\***hw**;**

int ret**;**

fix **=** kmalloc**(sizeof(\***fix**),** GFP\_KERNEL**);**

/\* struct clk\_fixed\_factor assignments \*/

fix**->**mult **=** mult**;**

fix**->**div **=** div**;**

fix**->**hw**.**init **=** **&**init**;**

init**.**name **=** name**;**

init**.**ops **=** **&**clk\_fixed\_factor\_ops**;**

init**.**flags **=** flags **|** CLK\_IS\_BASIC**;**

init**.**parent\_names **=** **&**parent\_name**;**

init**.**num\_parents **=** 1**;**

hw **=** **&**fix**->**hw**;**

ret **=** clk\_hw\_register**(**dev**,** hw**);**

**return** hw**;**

**}**

这里只需重点分析fixed factor clk的操作函数了。

const struct clk\_ops clk\_fixed\_factor\_ops **=** **{**

**.**round\_rate **=** clk\_factor\_round\_rate**,**

**.**set\_rate **=** clk\_factor\_set\_rate**,**

**.**recalc\_rate **=** clk\_factor\_recalc\_rate**,**

**};**

clk\_factor\_round\_rate获取最接近的rate，从fixed factor clk定义上看，这个rate是唯一的。即parent clk\*mul/div。

static long clk\_factor\_round\_rate**(**struct clk\_hw **\***hw**,** unsigned long rate**,** unsigned long **\***prate**)**

**{**

struct clk\_fixed\_factor **\***fix **=** to\_clk\_fixed\_factor**(**hw**);**

//如果有CLK\_SET\_RATE\_PARENT标志，表示先需要获取parent clock的round rate。

**if** **(**clk\_hw\_get\_flags**(**hw**)** **&** CLK\_SET\_RATE\_PARENT**)** **{**

unsigned long best\_parent**;**

best\_parent **=** **(**rate **/** fix**->**mult**)** **\*** fix**->**div**;**

**\***prate **=** clk\_hw\_round\_rate**(**clk\_hw\_get\_parent**(**hw**),** best\_parent**);**

**}**

**return** **(\***prate **/** fix**->**div**)** **\*** fix**->**mult**;**

**}**

clk\_factor\_set\_rate为空函数，因为rate不能设置。

clk\_factor\_recalc\_rate则直接返回parent\_clk\*mul/div。

#### 2.2.7 samsung\_clk\_of\_add\_provider

void \_\_init samsung\_clk\_of\_add\_provider**(**struct device\_node **\***np**,**struct samsung\_clk\_provider **\***ctx**)**

**{**

**if** **(**np**)** **{**

**if** **(**of\_clk\_add\_provider**(**np**,** of\_clk\_src\_onecell\_get**,&**ctx**->**clk\_data**))**

panic**(**"could not register clk provider\n"**);**

**}**

**}**

这里将clocks这个dts节点添加到of\_clk\_providers链表中。

#### 2.2.8 2440 clocks的使用

那么dts中如何是如何查找和使用对应的clock呢？

/\* DTS \*/

i2c-device **{**

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

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

**};**

/\* driver \*/

int xxx\_probe**(**struct platform\_device **\***pdev**)**

**{**

struct clk **\***i2c\_clk**;**

unsigned long rate**;**

i2c\_clk **=** devm\_clk\_get**(&**pdev**->**dev**,** “i2c”**);**

**if** **(**IS\_ERR**(**clk**))** **{**

…

**}**

rate **=** clk\_enable**(**i2c\_clk**);**

**}**

devm\_clk\_get获取到i2c clk对应的clk结构体流程如下：

1.devm\_clk\_get

2.clk\_get

//获取i2c-device节点下的clock

3. \_\_of\_clk\_get\_by\_name

//获取clock-names为i2c对应的index

4.\_\_of\_clk\_get

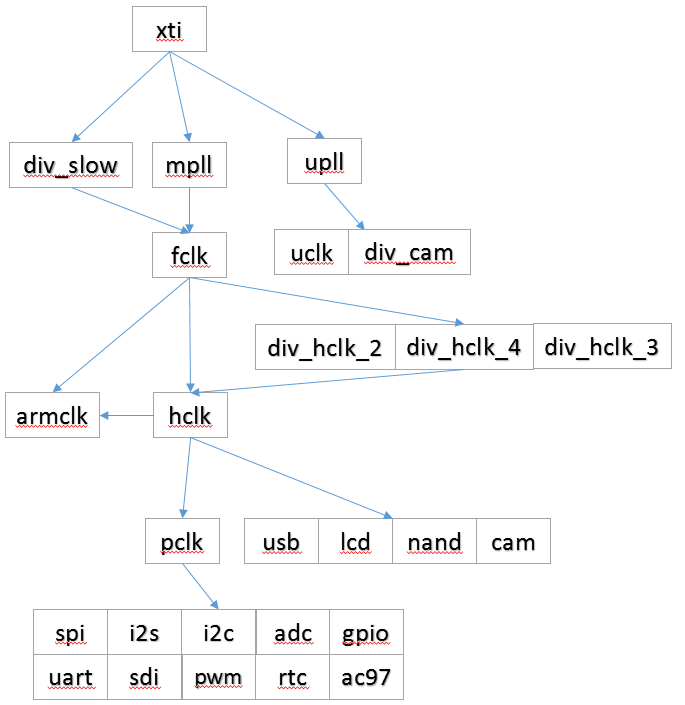
//解析clocks对应index的内容，也就是<&clocks PCLK\_I2C

//得出clocks对应的device\_node为clocks，arg为PCLK\_I2C

5.\_\_of\_clk\_get\_from\_provider

//调用of\_clk\_src\_onecell\_get从clk table中得到对应的clk结构体

#### 2.2.9 2440 clock树分析



##### 2.2.9.1 PLL类型clk相关操作分析

pll主要有mpll和upll，分别通过MPLLCON和UPLLCON完成相关操作。

**(1)clk\_enable/clk\_disable**

直接调用mpll/upll注册的enable/diable函数。

操作CLKSLOW中的bit5和bit7可以打开或者关闭mpll/upll。

**(2)clk\_round\_rate**

获得parent\_clock xti的频率12Mhz。

调用mpll/upll注册的round\_rate函数。

根据频率表找到最接近传入参数的频率(这里并未使用到xti的频率)。

**(3)clk\_set\_rate**

调用round\_rate计算离设置频率最合适的频率。

通知subtree rate改变了。

调用set\_rate设置新的频率。

操作MPLLCON/UPLLCON，并根据当前的parent\_rate和频率表进行比较，得到合适的寄存器值并设置。

**(4)clk\_get\_rate**

由于pll在注册的时候设置了CLK\_GET\_RATE\_NOCACHE标志，所以获取rate的时候需要重新计算。

调用recalc\_rate，根据parent\_clock的rate值和相关寄存器值，结合给定公式计算出mpll/upll的值。

##### 2.2.9.2 MUX类型clk相关操作

以fclk为例，fclk有两个parent为mpll或者div\_slow。

**(1)clk\_round\_rate**

因为有determine\_rate的存在，所以调用fclk注册的determine\_rate函数。

根据给定的频率，遍历number\_parents，得到mpll或div\_slow的rate。

靠近哪个频率则返回对应的频率。

**(2)clk\_set\_rate**

调用determine\_rate获取给定频率相近的频率和parent。

调用set\_parent改变parent达到设置频率的效果。

更新缓存中的rate值。

**(3)clk\_get\_rate**

没有设置CLK\_GET\_RATE\_NOCACHE标志，直接获取缓存中的rate。

**(4)clk\_set\_parent**

直接调用set\_parent。

从原来的parent subtree中删除自己并加入到新的subtree中。

**(5)clk\_get\_parent**

parent对应的clk结构体为clk->core->parent->hw->clk。

##### 2.2.9.3 DIV类型clk相关操作

以pclk为例，pclk是从hclk分频而来。

**(1)clk\_round\_rate**

调用pclk注册的round\_rate函数。

根据rate, parent\_rate talbe width计算出最合适的分频系数并得到round rate。

**(2)clk\_set\_rate**

调用round\_rate计算最接近设置频率的rate。

调用pclk注册的set\_rate函数。

根据parent\_rate table width计算寄存器值并设置。

**(3)clk\_get\_rate**

没有设置CLK\_GET\_RATE\_NOCACHE标志，直接获取缓存中的rate。

**其他两种类型clk就比较简单了，这里不再详细分析了。**

### 2.3 cpufreq流程猜测

clk子系统将所有的clk按照树的形式组合起来了，并且提供了通用的操作函数。

当cpufreq改变cpu频率时，整个系统的时钟会做些什么呢？

(1)获取需要设置的cpu频率

(2)计算当前分频值

(3)根据分频值计算新的mpll/fclk/hclk/pclk值。

(4)调用clk\_set\_rate设置mpll/fclk/hclk/pclk。

(5)改变sdram的刷新时间。

(6)如果有关注cpufreq变化的回调，则调用它们。例如i2c注册了相关的回调，当cpu频率改变时，需要重新设置i2c对应的寄存器，让i2c master clock保持原来的频率。