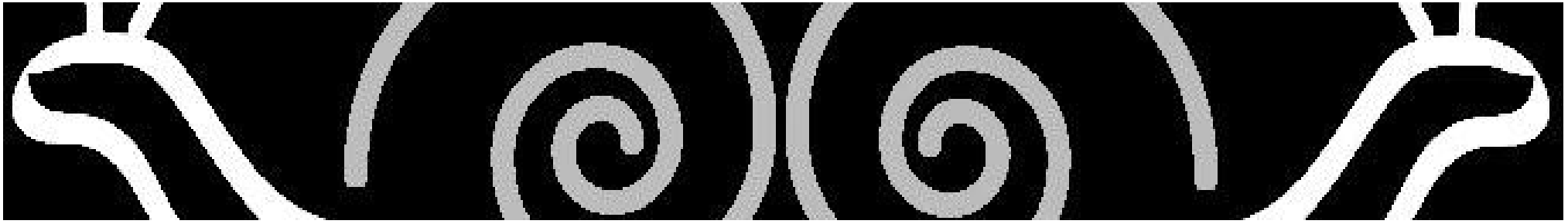


Linux common clock framework(2)_clock provider

作者：wowo 发布于：2014-10-23 23:49 分类：电源管理子系统

1. 前言

本文接上篇文章，从clock driver的角度，分析怎么借助common clock framework管理系统的时钟资源。换句话说，就是怎么编写一个clock driver。

由于kernel称clock driver为clock provider（相应的，clock的使用者为clock consumer），因此本文遵循这个规则，统一以clock provider命名。

2. clock有关的DTS

我们在“Linux common clock framework(1)_概述”中讲述clock consumer怎么使用clock时，提到过clock consumer怎么在DTS中指定所使用的clock。这里再做进一步说明。

2.1 clock provider的DTS

我们知道，DTS（Device Tree Source）是用来描述设备信息的，那系统的clock资源，是什么设备呢？换句话，用什么设备表示呢？这决定了clock provider的DTS怎么写。

通常有两种方式：

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：

```
1: /* arch/arm/boot/dts/exynos4210.dtsi */
2: clock: clock-controller@0x10030000 {
3:         compatible = "samsung,exynos4210-clock";
4:         reg = <0x10030000 0x20000>;
5:         #clock-cells = <1>;
6: };
```

> 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“Linux common clock framework(1)_概述”所提及的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。参考如下例子（如果能拿到对应的datasheet会更容易理解）：

```
44:         clocks = <&osc32k>, <&osc24M>, <&pll1>, <&dummy>;
45:     };
46:
47:     axi: axi@01c20054 {
48:         #clock-cells = <0>;
```

如果当前VA_B...

- Nicole
  期待接下来的精彩故事
- wowo
  @chenchuang：这种情况要由应用程序自己想办法，例如...
- chenchuang
  1. wowo大神，我想问下，如果几个ad struct 合...

文章分类

- Linux内核分析(11)
  - 统一设备模型(15)
  - 电源管理子系统(42)
  - 中断子系统(15)
  - 进程管理(17)
  - 内核同步机制(18)
  - GPIO子系统(5)
  - 时间子系统(14)
  - 通信类协议(7)
  - 内存管理(27)
  - 图形子系统(1)
  - 文件系统(4)
  - TTY子系统(6)
- u-boot分析(3)
- Linux应用技巧(13)
- 软件开发(6)
- 基础技术(13)
  - 蓝牙(16)
  - ARMv8A Arch(13)
  - 显示(3)
  - USB(1)
- 基础学科(10)
- 技术漫谈(12)
- 项目专区(0)

```
49:         compatible = "allwinner,sun4i-axi-clk";
50:         reg = <0x01c20054 0x4>;
51:         clocks = <&cpu>;
52:     };
53:
54:     axi_gates: axi_gates@01c2005c {
55:         #clock-cells = <1>;
56:         compatible = "allwinner,sun4i-axi-gates-clk";
```

> 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！）。话说回来了，osc24M的命名不是很好，如果频率改变，名称也得改吧，clock consumer的引用也得改吧；
>
> 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（后面会讲），clock consumer不能使用（或者可理解为不可见）；

也许您会问，这些DTS描述，怎么使用？怎么和代码关联起来？先不着急，我们慢慢看。

2.2 clock consumer的DTS

在2.1中的方法二，我们已经看到clock consumer的DTS了，因为很多clock provider也是clock consumer。这里再举几个例子，做进一步说明。

例子1（对应2.1中的方式1，来自同一个DTS文件）：

- X Project(28)

随机文章

- Concurrency Managed Workqueue之（一）：workqueue的基本概念
- Linux TTY framework(3)_从应用的角度看TTY设备
- Linux时间子系统之（一）：时间的基本概念
- Linux操作命令记录
- ARM Linux上的系统调用代码分析

文章存档

- 2018年6月(1)
- 2018年5月(1)
- 2018年4月(7)
- 2018年2月(4)
- 2018年1月(5)
- 2017年12月(2)
- 2017年11月(2)
- 2017年10月(1)
- 2017年9月(5)
- 2017年8月(4)
- 2017年7月(4)
- 2017年6月(3)
- 2017年5月(3)
- 2017年4月(1)
- 2017年3月(8)
- 2017年2月(6)
- 2017年1月(5)
- 2016年12月(6)
- 2016年11月(11)
- 2016年10月(9)
- 2016年9月(6)

```
/* arch/arm/boot/dts/exynos4210.dtsi */
mct@10050000 {
        compatible = "samsung,exynos4210-mct";
        ...
        clocks = <&clock 3>, <&clock 344>;
        clock-names = "fin_pll", "mct";
        ...
};
```

> clocks，指明该设备的clock列表，clk_get时，会以它为关键字，去device_node中搜索，以得到对应的struct clk指针；
>
> clocks需要指明的信息，由clock provider的“#clock-cells”规定：为0时，只需要提供一个clock provider name（称作phandle）；为1时，表示phandle有多个输出，则需要额外提供一个ID，指明具体需要使用那个输出。这个例子直接用立即数表示，更好的做法是，将系统所有clock的ID，定义在一个头文件中，而DTS可以包含这个头文件，如“clocks = <&clock CLK_SPI0>”；
>
> clock-names，为clocks指定的那些clock分配一些易于使用的名字，driver可以直接以名字为参数，get clock的句柄（具体可参考“Linux common clock framework(1)_概述”中clk_get相关的接口描述）。

例子2，如果clock provider的“#clock-cells”为0，可直接引用该clock provider的名字，具体可参考2.1中的方式2。

例子3，2.1中方式2有一个clock provider的名字为apb0_gates，它的“#clock-cells”为1，并通过clock-output-names指定了所有的输出clock，那么，clock consumer怎么引用呢？如下（2和.1中的方式2，来自同一个DTS文件）：

```
/* arch/arm/boot/dts/sun4i-a10.dtsi */
soc@01c20000 {
        compatible = "simple-bus";
        ...

        pio: pinctrl@01c20800 {
                compatible = "allwinner,sun4i-a10-pinctrl";
                reg = <0x01c20800 0x400>;
                clocks = <&apb0_gates 5>;
```

```
10:                 ...
11:         }
12: }
```

> 和例子1一样，指定phandle为“aph0_gates”，ID为5。

2.3 DTS相关的讨论和总结

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（后面会讲）。

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。

后面的API描述，蜗蜗会着重从方式2的角度，因为这样才能体会到软件设计中的美学。

注1：上面例子中用到了两个公司的代码，方式1是三星的，方式2是全志的。说实话，全志的代码写的真漂亮，一个默默无闻的白牌公司，比三星这种国际大公司强多了。从这里，我们可以看到中国科技业的未来，还是很乐观的。

3. clock provider有关的API汇整

clock provider的API位于include/linux/clk_provider.h。

3.1 struct clk_hw

由“Linux common clock framework(1)_概述”可知，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：

```
1: struct clk_hw {
2:         struct clk *clk;
3:         const struct clk_init_data *init;
4: };
```

> clk，struct clk指针，由clock framework分配并维护，并在需要时提供给clock consumer使用；
>
> init，描述该clock的静态数据，clock provider负责把系统中每个clock的静态数据准备好，然后交给clock framework的核心逻辑，剩下的事情，clock provider就不用操心了。这个过程，就是clock driver的编写过程，简单吧？该静态数据的数据结构如下。

```
1: struct clk_init_data {
2:         const char              *name;
3:         const struct clk_ops    *ops;
4:         const char              **parent_names;
5:         u8                      num_parents;
6:         unsigned long           flags;
7: };
```

> 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。

```
1: struct clk_ops {
2:         int             (*prepare)(struct clk_hw *hw);
```

```
3:          void            (*unprepare)(struct clk_hw *hw);
4:          int             (*is_prepared)(struct clk_hw *hw);
5:          void            (*unprepare_unused)(struct clk_hw *hw);
6:          int             (*enable)(struct clk_hw *hw);
7:          void            (*disable)(struct clk_hw *hw);
8:          int             (*is_enabled)(struct clk_hw *hw);
9:          void            (*disable_unused)(struct clk_hw *hw);
10:         unsigned long   (*recalc_rate)(struct clk_hw *hw,
11:                                        unsigned long parent_rate);
12:         long            (*round_rate)(struct clk_hw *hw, unsigned long,
```

这是clock的操作函数集， 很多和“Linux common clock framework(1)_概述”中的clock framework通用API一致（通用API会直接调用相应的操作函数）：

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；

注2：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，从新计算并返回clock rate；

注3：细心的读者可能会发现，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，后面会详细解释；

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

```
/*
 * flags used across common struct clk.  these flags should only affect the
 * top-level framework.  custom flags for dealing with hardware specifics
 * belong in struct clk_foo
 */
#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。

注4：round_rate和CLK_SET_RATE_PARENT
当clock consumer调用clk_round_rate获取一个近似的rate时，如果该clock没有提供.round_rate函数，有两种方法：
1）在没有设置CLK_SET_RATE_PARENT标志时，直接返回该clock的cache rate
2）如果设置了CLK_SET_RATE_PARENT标志，则会询问parent，即调用clk_round_rate获取parent clock能提供的、最接近该rate的值。这是什么意思呢？也就是说，如果parent clock可以得到一个近似的rate值，那么通过改变parent clock，就能得到所需的clock。
在后续的clk_set_rate接口中，会再次使用该flag，如果置位，则会在设置rate时，传递到parent clock，因此parent clock的rate可能会重设。

讲的很拗口，我觉得我也没说清楚，那么最好的方案就是：在写clock driver时，最好不用这个flag，简单的就是最好的（前提是能满足需求）。

3.2 clock tree建立相关的API

3.2.1 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：

```
/**
 * clk_register - allocate a new clock, register it and return an opaque cookie
 * @dev: device that is registering this clock
 * @hw: link to hardware-specific clock data
 *
 * clk_register is the primary interface for populating the clock tree with new
 * clock nodes.  It returns a pointer to the newly allocated struct clk which
 * cannot be dereferenced by driver code but may be used in conjuction with the
 * rest of the clock API.  In the event of an error clk_register will return an
 * error code; drivers must test for an error code after calling clk_register.
 */
struct clk *clk_register(struct device *dev, struct clk_hw *hw);
struct clk *devm_clk_register(struct device *dev, struct clk_hw *hw);
```

> 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，下面继续。

3.2.2 clock分类及register

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s，统一处理，这充分体现了面向对象的思想。

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，如下：

```
1: /**
2:  * struct clk_fixed_rate - fixed-rate clock
3:  * @hw:          handle between common and hardware-specific interfaces
4:  * @fixed_rate: constant frequency of clock
5:  */
6: struct clk_fixed_rate {
7:         struct          clk_hw hw;
8:         unsigned long   fixed_rate;
9:         u8              flags;
10: };
11:
12: extern const struct clk_ops clk_fixed_rate_ops;
13: struct clk *clk_register_fixed_rate(struct device *dev, const char *name,
```

> 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。

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：

```
1: struct clk *clk_register_gate(struct device *dev, const char *name,
2:                 const char *parent_name, unsigned long flags,
3:                 void __iomem *reg, u8 bit_idx,
4:                 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3.1中的说明；
>
> 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。

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：

```
1: struct clk *clk_register_divider(struct device *dev, const char *name,
```

```
2:                 const char *parent_name, unsigned long flags,
3:                 void __iomem *reg, u8 shift, u8 width,
4:                 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：

```
1: struct clk *clk_register_divider_table(struct device *dev, const char *name,
2:                 const char *parent_name, unsigned long flags,
3:                 void __iomem *reg, u8 shift, u8 width,
4:                 u8 clk_divider_flags, const struct clk_div_table *table,
5:                 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。

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。

```
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,
```

```
4:                  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。

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：

```
1: struct clk *clk_register_fixed_factor(struct device *dev, const char *name,
2:                 const char *parent_name, unsigned long flags,
3:                 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。

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：

```
1: struct clk *clk_register_composite(struct device *dev, const char *name,
2:                 const char **parent_names, int num_parents,
3:                 struct clk_hw *mux_hw, const struct clk_ops *mux_ops,
4:                 struct clk_hw *rate_hw, const struct clk_ops *rate_ops,
5:                 struct clk_hw *gate_hw, const struct clk_ops *gate_ops,
6:                 unsigned long flags);
```

> 看着有点复杂，但理解了上面1~5类clock，这里就只剩下苦力了，耐心一点，就可以了。

3.2.3 DTS相关的API

再回到第2章DTS相关的介绍，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。

3.2.1和3.2.2小节所描述的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：

```
1: int of_clk_add_provider(struct device_node *np,
2:                         struct clk *(*clk_src_get)(struct of_phandle_args *args,
3:                                                    void *data),
4:                         void *data);
```

> 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（第2章的例子），clock framework core提供一个默认的会调用函数，如下：

```
1: struct clk_onecell_data {
```

```
2:          struct clk **clks;
3:          unsigned int clk_num;
4: };
5: 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。

4. 使用clock framework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，讲这些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。

切记，尽量利用kernel已有资源，不要多写一行代码，简洁的就是美的！

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。

5）最后，也是最重要的一点，多看kernel源代码，多模仿，多抄几遍，什么都熟悉了！

原创文章，转发请注明出处。蜗窝科技，www.wowotech.net。

标签: Linux framework clock provider dts

« 防冲突机制介绍 | arm64 linux移植»

评论：

zaixi
2018-03-10 16:49

CLK_DIVIDER_ONE_BASED，实际的divider值就是寄存器值

这里有笔误，应该是实际值加上1

回复

> **zaixi**
> 2018-03-10 16:57
>
> @zaixi：sorry，没问题，是我自己看错了
>
> 回复

小亮
2017-10-10 14:45

@wowo 现在内核又加入了 fractional-divider 类型的clock支持。
http://www.spinics.net/lists/u-boot-v2/msg22600.html，这是加入该类型clock的patch链接

回复

> **wowo**
> 2017-10-10 16:34
>
> @小亮：多谢分享。能否帮忙解释一下这种类型的clock的涵义以及应用场景呢?
>
> 回复

wink
2016-12-06 14:58

看到了我司的代码 ^_^

回复

wowo

2016-12-06 17:04

@wink：哈哈～～

回复

jim

2016-09-08 14:01

"注1：上面例子中用到了两个公司的代码，方式1是三星的，方式2是全志的。说实话，全志的代码写的真漂亮，一个默默无闻的白牌公司，比三星这种国际大公司强多了。从这里，我们可以看到中国科技业的未来，还是很乐观的。"
全志的code并不全是自己写的，很大一部分都是外包给free electrons的，比如下面这个ppt的上就写着，链接：
http://events.linuxfoundation.org/sites/events/files/slides/ripard-dmaengine.pdf
我还在很多其他地方的ppt中看到过类似的owner维护sunxi soc的信息，就不一一列举了

回复

wowo

2016-09-08 18:19

@jim：确实如此，我也是后来才知道的，呵呵~~~

回复

Renkangshuo

2016-09-01 17:43

现在正在学习linux clock subsystem ，看了您的博客， 受益很大， 谢谢您！！！！

回复

wowo

2016-09-02 08:45

@Renkangshuo：不客气，希望可以多交流~~~

回复

ooonebook

2016-08-26 14:45

之前看高通clock框架的时候，看这篇文章感觉差异挺大，就没细看。
现在在移植tiny210的过程发现开机过程中获取PLL的频率出来都是0导致crash，
看了一下代码懵圈了，再来看这篇文章，瞬间理解了好多~膜拜一下wowo~哈哈

回复

wowo
2016-08-26 15:26

@ooonebook：结合实际的需求去看代码，效果才是比较好的~~~

回复

heavenward
2016-03-01 16:13

有个小疑问。像uart这种需要不同工作频率的设备，在get source clk时，也有可能需要不同的source clk 频率来得到正确的工作频率。如果uart的clk source是个mux clk，可以从frac，divider，fix 3种类型获取时钟频率的话，获取与fix clk不同的时钟频率时，recal 函数是会尝试从frac，divider，这2个父得到请求的时钟么?

回复

wowo
2016-03-01 20:58

@heavenward：recal只是用来计算当前的频率值，不会涉及clock路径的选择，如果需要使用不同的parent，还是需要通过set parent接口主动设置的。

回复

zbunix
2016-11-04 23:53

@wowo：例如uart 962100频率，可以分出给uart路径有7种比较复杂，每种路径分出误差都不同，但肯定存在一种路径分出误差最小，要7种路径手工逐个计算出来？还是有现成算法代码函数呢?

回复

wowo
2016-11-05 21:19

@zbunix：这样的算法有点复杂，没必要由kernel来做，因此没有。clock framework只是提供一种手段而已，具体用什么值，还是要人先心里有数。

回复

super-
2015-05-12 18:15

@wowo：“ 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。” 对于这句话的“ 并在初始化时，调用OF模块，查找所有的DTS匹配项” 是什么意思？ 在kernel下有没有具体的clock driver的例子？

回复

super-

2015-05-12 18:19

@super-：还有 struct clk *(*clk_src_get)()这个回调函数，我的理解是在clock framework解析具体的使用clock device时，解析这个device的DTS节点的clocks属性时，需要调用这个函数向clock provider查询。

回复

super-

2015-05-12 18:57

@super-：/*
 * Returning -EPROBE_DEFER here is inefficient due to
 * destroying work 'unnecessarily'
 */
for_each_available_child_of_node(dev->of_node, child) {
result = msmclk_parse_dt_node(dev, child);
if (!IS_ERR(result))
continue;
if (!msmclk_debug)
return PTR_ERR(result);
/*
 * Parse and report all errors instead of immediately
 * exiting. Return the first error code.
 */
if (!rc)
rc = PTR_ERR(result);
}
那个查找所有的DTS匹配项是这个操作吗？ 这个操作就是解析某个结点的所有子结点

回复

super-

2015-05-12 19:06

@super-：&soc {
clock_rpm: qcom,rpmcc@1800000 {
compatible = "qcom,msm-clock-controller";
reg = <0x1800000 0x80000>;

```
            reg-names = "cc-base";
            #clock-cells = <1>;
        };

        clock_a7pll: qcom,a7pll@0xB008018 {
            compatible = "qcom,msm-clock-controller";
            reg = <0xB008018 0x9020>;
            reg-names = "cc-base";
            #clock-cells = <1>;

            clock-names = "a7_xo_a";
            clocks = <&clock_rpm clk_xo_a_clk_src>;

            qcom,regulator-names = "a7pll_vdd_dig";
            a7pll_vdd_dig-supply = <&pmd9635_s5_corner_ao>;

            a7pll_vdd_dig: a7pll_vdd_dig {
                compatible = "qcom,simple-vdd-class";
                qcom,regulators = <&pmd9635_s5_corner_ao>;
                qcom,uV-levels =
                        <RPM_REGULATOR_CORNER_NONE>,
                        <RPM_REGULATOR_CORNER_SVS_SOC>,
                        <RPM_REGULATOR_CORNER_NOMINAL>,
                        <RPM_REGULATOR_CORNER_SUPER_TURBO>;
            };
        };
} ;
```

我在kernel中找到了匹配这个node的driver

回复

super-

2015-05-12 19:30

@super-：@wowo： 我在driver 代码里没有找到调用 xxx_register() 的代码。 怎么回事?

回复

super-

2015-05-12 19:32

@super-：这个是xx_probe()函数：

```
static int msm_clock_dummy_probe(struct platform_device *pdev)
{
    int ret;

    ret = of_clk_add_provider(pdev->dev.of_node, of_dummy_get, NULL);
    if (ret)
        return -ENOMEM;

    dev_info(&pdev->dev, "Registered DUMMY provider.\n");
    return ret;
}
```

这个是回调函数：

```
 static struct clk *of_dummy_get(struct of_phandle_args *clkspec,
                void *data)
{
    return &dummy_clk;
}
```

回复

wowo

2015-05-13 08:51

@super-：这个probe应该是一个示例性质的（dummy），不知道是否会有register接口。
你可以看看其它例子，如：drivers/clk/tegra/clk-tegra114.c

wowo

2015-05-12 20:32

@super-：super兄，你的问题比较多，我一时不知道怎么回答了啊。或者你只是自言自语，不需要我回答？呵呵。

回复

super-

2015-05-12 21:04

@wowo：@wowo: 我现在疑惑的是static int msm_clock_dummy_probe(struct platform_device *pdev)
{
    int ret;

    ret = of_clk_add_provider(pdev->dev.of_node, of_dummy_get, NULL);

```
    if (ret)
        return -ENOMEM;

    dev_info(&pdev->dev, "Registered DUMMY provider.\n");
    return ret;
} 这个函数里为什么没有类似 clk_register_divider( )的调用?
```

回复

puppypyb
2014-11-13 09:22

还有clk_root_list, clk_orphan_list, clkdev_add 这些都有必要给大家讲讲

期待第三篇 :)

回复

> **wowo**
> 2014-11-13 09:46
>
> @puppypyb：好的，我把pm domain的先写一篇之后，再回到clock上面。多谢关注~~
>
> 回复

puppypyb
2014-11-12 19:04

clock tree中某clock的类型是怎么确定的?

我认为是根据该clock前面的时钟部件（mux、dvider、gate等）决定的。 例如mux类型的clock，是因为这个clock是某mux的输出。所以说，给你一个clock tree，怎么调用对应的注册函数（clk_register_divider还是clk_register_mux）去注册clock就要看该clock前面是什么部件。

wowo怎么看?

回复

> **wowo**
> 2014-11-12 19:30
>
> @puppypyb：您的理解是对的。
> 弄清楚这个问题，首先要定义我们这里讨论的“clock”到底是什么？clock framework把某个器件的输出定义为“clock”，就决定了clock是一个虚拟的存在。

例如OSC的输出，在clock framework中可以定义为"osc_clk"，很显然，这个clock为fixed rate clock。那么，“osc_clk”的作用范围是什么呢？怎么派生出其它的clock呢？
显然，如果某个clock不经过可以改变它的器件，包括mux、gate、divider等等，那么这个clock一直存在。只要经过某一个器件，我们就认为一个新的clock产生了。至于这个clock的类型，就是产生这个新的clock器件的类型。

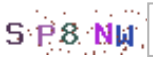
回复

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)



发表评论