Linux common clock framework(1)_概述

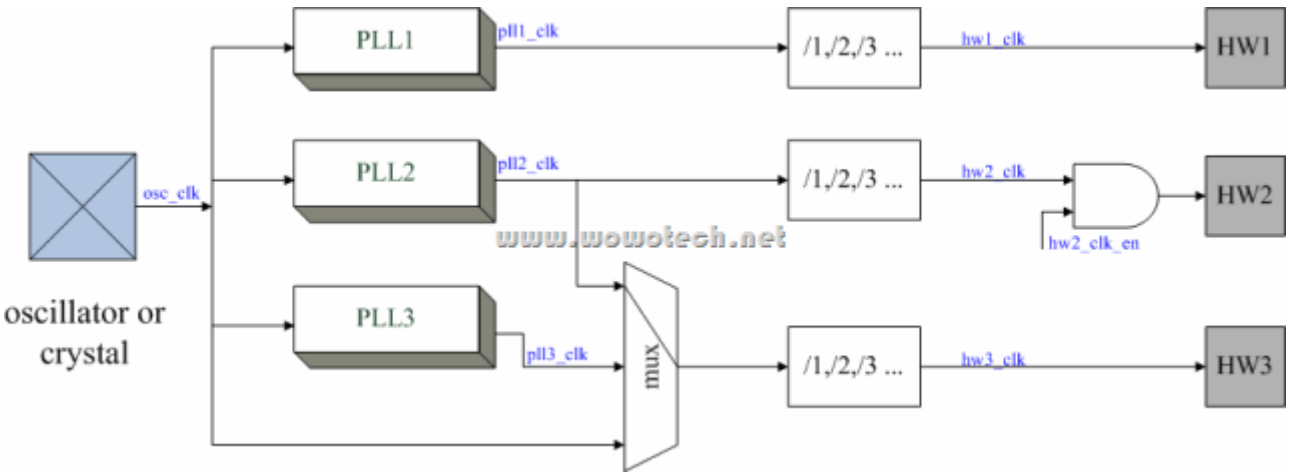
作者：wowo 发布于：2014-10-20 23:06 分类：电源管理子系统

1. 前言

common clock framework 是用来管理系统 clock 资源的子系统，根据职能，可分为三个部分：
1）向其它 driver 提供操作 clocks 的通用 API。
2）实现 clock 控制的通用逻辑，这部分和硬件无关。
3）将和硬件相关的 clock 控制逻辑封装成操作函数集，交由底层的 platform 开发者实现，由通用逻辑调用。
因此，蜗蜗会将 clock framework 的分析文章分为 3 篇：
第一篇为概述和通用 API 的使用说明，面向的读者是使用 clock 的 driver 开发者，目的是掌握怎么使用 clock framework（就是本文）；
第二篇为底层操作函数集的解析和使用说明，面向的读者是 platform clock driver 的开发者，目的是掌握怎么借助 clock framework 管理系统的时钟资源；
第三篇为 clock framework 的内部逻辑解析，面向的读者是 linux kernel 爱好者，目的是理解怎么实现 clock framework。
注 1：任何 framework 的职能分类都是如此，因此都可以按照这个模式分析。

2. 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：



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。

3. common clock framework 提供的通用 API

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）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。

注 2：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。

2）clock 获取有关的 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：

```
1: struct clk *clk_get(struct device *dev, const char *id);
2: struct clk *devm_clk_get(struct device *dev, const char *id);
3: void clk_put(struct clk *clk);
4: void devm_clk_put(struct device *dev, struct clk *clk);
5: struct clk *clk_get_sys(const char *dev_id, const char *con_id);
6:
7: struct clk *of_clk_get(struct device_node *np, int index);
8: struct clk *of_clk_get_by_name(struct device_node *np, const char *name);
9: struct clk *of_clk_get_from_provider(struct of_phandle_args *clkspec);
```

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。

3）clock 控制有关的 API

```
1: int clk_prepare(struct clk *clk)
2: void clk_unprepare(struct clk *clk)
3:
4: static inline int clk_enable(struct clk *clk)
5: static inline void clk_disable(struct clk *clk)
6:
7: static inline unsigned long clk_get_rate(struct clk *clk)
8: static inline int clk_set_rate(struct clk *clk, unsigned long rate)
9: static inline long clk_round_rate(struct clk *clk, unsigned long rate)
10:
11: static inline int clk_set_parent(struct clk *clk, struct clk *parent)
12: static inline struct clk *clk_get_parent(struct clk *clk)
13:
14: static inline int clk_prepare_enable(struct clk *clk)
15: 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。
注 2：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。
4）其它接口

```
1: int clk_notifier_register(struct clk *clk, struct notifier_block *nb);
2: 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: int clk_add_alias(const char *alias, const char *alias_dev_name, char *id,
2:                   struct device *dev);
```

这是一个非主流接口，用于给某个 clk 起个别名。无论出于何种原因，尽量不要它，保持代码的简洁，是最高原则！

4. 通用 API 的使用说明

结合一个例子（摘录自“Documentation/devicetree/bindings/clock/clock-bindings.txt”），说明 driver 怎么使用 clock 通用 API。
1）首先，在 DTS（device tree source）中，指定 device 需要使用的 clock，如下：

```
1: /* DTS */
2: device {
3:     clocks = <&osc 1>, <&ref 0>;
4:     clock-names = "baud", "register";
5: };
```

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（具体会在下一篇文章讲述）。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）。
2）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。
3）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：

```
1: /* driver */
2: int xxx_probe(struct platform_device *pdev)
3: {
4:         struct clk *baud_clk;
5:         int ret;
6:
7:         baud_clk = devm_clk_get(&pdev->dev, “baud”);
```

```
 8:         if (IS_ERR(clk)) {
 9:                 …
10:         }
11:
12:         ret = clk_prepare_enable(baud_clk);
13:         if (ret) {
14:                 …
15:         }
16: }
```