Linux common clock framework(3)_实现逻辑分析

作者：wowo 发布于：2014-11-24 22:31 分类：电源管理子系统

1. 前言

前面两篇clock framework的分析文章，分别从clock consumer和clock provider的角度，介绍了Linux kernel怎么管理系统的clock资源，以及device driver怎么使用clock资源。本文将深入到clock framework的内部，分析相关的实现逻辑。

注：本文使用的kernel版本为linux-3.10.29。虽然最新版本的kernel增加了一些内容，但主要逻辑没有改变，就不紧跟kernel的步伐了。

2. struct clk结构

到目前为止，我们还没有仔细介绍过struct clk这个代表了一个clock的数据结构呢。对consumer和provider来说，可以不关心，但对内部实现逻辑来说，就不得不提了：

```
/* include/linux/clk-private.h */
struct clk {
        const char              *name;
        const struct clk_ops    *ops;
        struct clk_hw           *hw;
        struct clk              *parent;
        const char              **parent_names;
        struct clk              **parents;
        u8                      num_parents;
        unsigned long           rate;
        unsigned long           new_rate;
        unsigned long           flags;
        unsigned int            enable_count;
```

> name, ops, hw, parents_name, num_parents, flags, 可参考“Linux common clock framework(2)_clock provider”中的相关描述；
>
> parent，保存了该clock当前的parent clock的struct clk指针；
>
> parents，一个指针数组，保存了所有可能的parent clock的struct clk指针；
>
> rate，当前的clock rate；
>
> 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，后面再详解；
>
> enable_count, prepare_count，该clock被enable和prepare的次数，用于确保enable/disable以及prepare/unprepare的成对调用；
>
> children，该clock的children clocks（孩儿们），以链表的形式组织；
>
> child_node，一个list node，自己作为child时，挂到parent的children list时使用；
>
> notifier_count，记录注册到notifier的个数。

3. clock regitser/unregister

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)

在“Linux common clock framework(2)_clock provider”中已经讲过，clock provider需要将系统的clock以tree的形式组织起来，分门别类，并在系统初始化时，通过provider的初始化接口，或者clock framework core的DTS接口，将所有的clock注册到kernel。

clock的注册，统一由clk_regitser接口实现，但基于该接口，kernel也提供了其它更为便利注册接口，下面将会一一描述。

3.1 clk_regitser

clk_register是所有register接口的共同实现，负责将clock注册到kernel，并返回代表该clock的struct clk指针。分析该接口之前，我们先看一下下面的内容：

```
1: 1 F   f    clk_register      .\arch\arm\mach-at91\clock.c
2:              int __init clk_register(struct clk *clk)
3: 2 F   v    clk_register      .\arch\arm\mach-davinci\clock.c
4:              EXPORT_SYMBOL(clk_register);
5: 3 F   f    clk_register      .\arch\arm\mach-davinci\clock.c
6:              int clk_register(struct clk *clk)
7: 4 F   v    clk_register      .\arch\arm\mach-omap1\clock.c
8:              EXPORT_SYMBOL(clk_register);
9: 5 F   f    clk_register      .\arch\arm\mach-omap1\clock.c
10:             int clk_register(struct clk *clk)
11: 6 F   v    clk_register      .\arch\c6x\platforms\pll.c
12:             EXPORT_SYMBOL(clk_register);
13: 7 F   f    clk_register      .\arch\c6x\platforms\pll.c
```

> 上面是kernel中clk_register接口可能的实现位置，由此可以看出，clk_register在“include/linux/clk-provider.h”中声明，却可能在不同的C文件中实现。其它clock API也类似。这说明了什么？
>
> 这恰恰呼应了“Linux common clock framework”中“common”一词。
>
> 在旧的kernel中，clock framework只是规定了一系列的API声明，具体API的实现，由各个machine代码完成。这就导致每个machine目录下，都有一个类似clock.c的文件，以比较相似的逻辑，实现clock provider的功能。显然，这里面有很多冗余代码。

- X Project(28)

随机文章

- 显示技术介绍(3)_CRT技术
- Linux时间子系统之（十三）：Tick Device layer综述
- ARM64架构下地址翻译相关的宏定义
- Linux MMC framework(1)_软件架构
- linux kernel的中断子系统之（七）：GIC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)
- 2016年8月(9)

后来，kernel将这些公共代码，以clock provider的形式（上面drivers/clk/clk.c文件）抽象出来，就成了我们所说的common clock framework。

后面所有的描述，都会以common clock framework的核心代码为基础，其它的，就不再涉及了。

下面是clk_register的实现：

```
1: /**
2:  * clk_register - allocate a new clock, register it and return an opaque cookie
3:  * @dev: device that is registering this clock
4:  * @hw: link to hardware-specific clock data
5:  *
6:  * clk_register is the primary interface for populating the clock tree with new
7:  * clock nodes.  It returns a pointer to the newly allocated struct clk which
8:  * cannot be dereferenced by driver code but may be used in conjuction with the
9:  * rest of the clock API.  In the event of an error clk_register will return an
10:  * error code; drivers must test for an error code after calling clk_register.
11:  */
12: struct clk *clk_register(struct device *dev, struct clk_hw *hw)
13: {
```

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（具体可参考“Linux common clock framework(2)_clock provider”），在内部分配一个struct clk变量后，将clock信息保存在变量中，并返回给调用者。实现逻辑如下：

分配struct clk空间；

根据struct clk_hw指针提供的信息，初始化clk的name、ops、hw、flags、num_parents、parents_names等变量；

调用内部接口__clk_init，执行后续的初始化操作。这个接口包含了clk_regitser的主要逻辑，具体如下。

```
1: /**
2:  * __clk_init - initialize the data structures in a struct clk
3:  * @dev:    device initializing this clk, placeholder for now
4:  * @clk:    clk being initialized
5:  *
6:  * Initializes the lists in struct clk, queries the hardware for the
7:  * parent and rate and sets them both.
8:  */
9: int __clk_init(struct device *dev, struct clk *clk)
10: {
11:     int i, ret = 0;
12:     struct clk *orphan;
13:     struct hlist_node *tmp2;
```

__clk_init接口的实现相当繁琐，做的事情包括：

20~26行，以clock name为参数，调用__clk_lookup接口，查找是否已有相同name的clock注册，如果有，则返回错误。由此可以看出，clock framework以name唯一识别一个clock，因此不能有同名的clock存在；

28~42行，检查clk ops的完整性，例如：如果提供了set_rate接口，就必须提供round_rate和recalc_rate接口；如果提供了set_parent，就必须提供get_parent。这些逻辑背后的含义，会在后面相应的地方详细描述；

50~73行，分配一个struct clk *类型的数组，缓存该clock的parents clock。具体方法是根据parents_name，查找相应的struct clk指针；

75行，获取当前的parent clock，并将其保存在parent指针中。具体可参考下面“说明2”；

77~93行，根据该clock的特性，将它添加到clk_root_list、clk_orphan_list或者parent->children三个链表中的一个，具体请参考下面“说明1”；

95~107行，计算clock的初始rate，具体请参考下面“说明3”；

109~126行，尝试reparent当前所有的孤儿（orphan）clock，具体请参考下面“说明4”；

128~137行，如果clock ops提供了init接口，执行之（由注释可知，kernel不建议提供init接口）。

上面的clock init流程，有下面4点补充说明：

说明1：clock的管理和查询

> clock framework有2条全局的链表：clk_root_list和clk_orphan_list。所有设置了CLK_IS_ROOT属性的clock都会挂在clk_root_list中。其它clock，如果有valid的parent ，则会挂到parent的“children”链表中，如果没有valid的parent，则会挂到clk_orphan_list中。
>
> 查询时（__clk_lookup接口做的事情），依次搜索：clk_root_list-->root_clk-->children-->child's children，clk_orphan_list-->orphan_clk-->children-->child's children，即可。

说明2：当前parent clock的选择（__clk_init_parent）

> 对于没有parent，或者只有1个parent 的clock来说，比较简单，设置为NULL，或者根据parent name获得parent的struct clk指针接。
>
> 对于有多个parent的clock，就必须提供.get_parent ops，该ops要根据当前硬件的配置情况，例如寄存器值，返回当前所有使用的parent的index（即第几个parent）。然后根据index，取出对应parent clock的struct clk指针，作为当前的parent。

说明3：clock的初始rate计算

> 对于提供.recalc_rate ops的clock来说，优先使用该ops获取初始的rate。如果没有提供，退而求其次，直接使用parent clock的rate。最后，如果该clock没有parent，则初始的rate只能选择为0。

> .recalc_rate ops的功能，是以parent clock的rate为输入参数，根据当前硬件的配置情况，如寄存器值，计算获得自身的rate值。

说明4：orphan clocks的reparent

> 有些情况下，child clock会先于parent clock注册，此时该child就会成为orphan clock，被收养在clk_orphan_list中。
>
> 而每当新的clock注册时，kernel都会检查这个clock是否是某个orphan的parent，如果是，就把这个orphan从clk_orphan_list中移除，放到新注册的clock的怀抱。这就是reparent的功能，它的处理逻辑是：
>
> a） 遍历orphan list，如果orphan提供了.get_parent ops，则通过该ops得到当前parent的index，并从parent_names中取出该parent的name，然后和新注册的clock name比较，如果相同，呵呵，找到parent了，执行__clk_reparent，进行后续的操作。
>
> b） 如果没有提供.get_parent ops，只能遍历自己的parent_names，检查是否有和新注册clock匹配的，如果有，执行__clk_reparent，进行后续的操作。
>
> c） __clk_reparent会把这个orphan从clk_orphan_list中移除，并挂到新注册的clock上。然后调用__clk_recalc_rates，重新计算自己以及自己所有children的rate。计算过程和上面的clock rate设置类似。

3.2 clk_unregister/devm_clk_register/devm_clk_unregister

clock的regitser和init，几乎占了clock framework大部分的实现逻辑。clk_unregister是regitser接口的反操作，不过当前没有实现（不需要）。而devm_clk_register/devm_clk_unregister则是clk_register/clk_unregister的device resource manager版本。

3.3 fixed rate clock的注册

“Linux common clock framework(2)_clock provider”中已经对fixed rate clock有过详细的介绍，这种类型的clock有两种注册方式，通过API注册和通过DTS注册，具体的实现位于“drivers/clk/clk-fixed-rate.c”中，介绍如下。

1）通过API注册

```
1: struct clk *clk_register_fixed_rate(struct device *dev, const char *name,
2:                 const char *parent_name, unsigned long flags,
3:                 unsigned long fixed_rate)
4: {
5:         struct clk_fixed_rate *fixed;
6:         struct clk *clk;
7:         struct clk_init_data init;
8:
9:         /* allocate fixed-rate clock */
10:        fixed = kzalloc(sizeof(struct clk_fixed_rate), GFP_KERNEL);
11:        if (!fixed) {
12:                pr_err("%s: could not allocate fixed clk\n", __func__);
13:                return ERR_PTR(-ENOMEM);
```

> clk_register_fixed_rate API用于注册fixed rate clock，它接收传入的name、parent_name、flags、fixed_rate等参数，并转换为struct clk_hw结构，最终调用clk_register接口，注册clock。大致的逻辑如下：
>
> 16~20行，定义一个struct clk_init_data类型的变量（init），并根据传入的参数以及fixed rate clock的特性，初始化该变量；
>
> 22~30行，分配一个私有的数据结构（struct clk_fixed_rate），并将init的指针保存在其中，最后调用clk_regitser注册该clock。

说明1：struct clk_init_data类型的变量

struct clk_init_data类型的变量（init），是一个局部变量，传递给clk_regitser使用时，用的是它的指针，说明了什么？说明该变量不会再后面使用了。再回忆一下clk_regitser的实现逻辑，会把所有的信息copy一遍，这里就好理解了。后面其它类型的clock注册时，道理相同。

说明2：fixed rate clock的实现思路

私有数据结构的定义如下：

```
1: struct clk_fixed_rate {
2:         struct          clk_hw hw;
3:         unsigned long   fixed_rate;
4:         u8              flags;
5: };
```

包含一个struct clk_hw变量，用于clk_regitser。另外两个变量，则为该类型clock特有的属性。私有数据结构变量（fixed）是通过kzalloc分配的，说明后续还需要使用。那怎么用呢？

由clk_regitser的实现可知，fixed rate clock注册时hw);>，把fixed指针中hw变量的地址保存在了struct clk指针中了。因此，在任何时候，通过struct clk指针（clock的代表），就可以找到所对应clock的struct clk_hw指针，从而可以找到相应的私有变量（fixed）的指针以及其中的私有数据。

基于此，fixed rate ops的实现就顺利成章了：

```
1: #define to_clk_fixed_rate(_hw) container_of(_hw, struct clk_fixed_rate, hw)
2:
3: static unsigned long clk_fixed_rate_recalc_rate(struct clk_hw *hw,
4:                 unsigned long parent_rate)
5: {
6:         return to_clk_fixed_rate(hw)->fixed_rate;
7: }
8:
9: const struct clk_ops clk_fixed_rate_ops = {
10:         .recalc_rate = clk_fixed_rate_recalc_rate,
11: };
12: EXPORT_SYMBOL_GPL(clk_fixed_rate_ops);
```

2）通过DTS注册

fixed rate clock是非常简单的一种clock，因而可以直接通过DTS的形式注册，clock framework负责解析DTS，并调用API注册clock，如下：

```
#ifdef CONFIG_OF
/**
 * of_fixed_clk_setup() - Setup function for simple fixed rate clock
 */
void of_fixed_clk_setup(struct device_node *node)
{
        struct clk *clk;
        const char *clk_name = node->name;
        u32 rate;

        if (of_property_read_u32(node, "clock-frequency", &rate))
                return;

```

首先看一下CLK_OF_DECLARE宏，它的定义位于“include/linux/clk-provider.h”中，负责在指定的section中（以__clk_of_table开始的位置），定义struct of_device_id类型的变量，并由of_clk_init接口解析、匹配，如果匹配成功，则执行相应的回调函数（这里为of_fixed_clk_setup）；

初始化的时候，device tree负责读取DTS，并和这些变量的名字（这里为"fixed-clock"）匹配，如果匹配成功，则执行相应的回调函数（这里为of_fixed_clk_setup）；

of_fixed_clk_setup会解析两个DTS字段"clock-frequency"和"clock-output-names"，然后调用clk_register_fixed_rate，注册clock。注意，注册时的flags为CLK_IS_ROOT，说明目前只支持ROOT类型的clock通过DTS注册；

最后，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。该接口会在后面再详细介绍。

of_clk_init负责从DTS中扫描并初始化clock provider，如下：

```
/* drivers/clk/clk.c */
/**
 * of_clk_init() - Scan and init clock providers from the DT
 * @matches: array of compatible values and init functions for providers.
 *
 * This function scans the device tree for matching clock providers and
 * calls their initialization functions
 */
void __init of_clk_init(const struct of_device_id *matches)
{
        struct device_node *np;

        if (!matches)
```

> 该接口有一个输入参数，用于指定需要扫描的OF id，如果留空，则会扫描__clk_of_table，就是通过CLK_OF_DECLARE宏指定的fixed rate等类型的clock。
>
> 在最新的kernel中，会在初始化代码（time_init）中以NULL为参数调用一次of_clk_init，以便自动匹配并初始化DTS中的描述的类似fixed rate的clock。

注2：这里使用大量篇幅描述一个简单的fixed rate clock的注册方式，主要目的是给大家介绍一种通用的实现方式，或者说通用思路。后面其它类型的clock，包括我们自定义的类型，实现方法都是一样的。这里就不罗嗦了，大家看代码就可以了。

3.4 gate、devider、mux、fixed factor、composite以及自定义类型clock的注册

和fixed rate类似，不再一一说明。

4. 通用API的实现

4.1 clock get

clock get是通过clock名称获取struct clk指针的过程，由clk_get、devm_clk_get、clk_get_sys、of_clk_get、of_clk_get_by_name、of_clk_get_from_provider等接口负责实现，这里以clk_get为例，分析其实现过程（位于drivers/clk/clkdev.c中）。

1）clk_get

```
1: struct clk *clk_get(struct device *dev, const char *con_id)
2: {
3:         const char *dev_id = dev ? dev_name(dev) : NULL;
4:         struct clk *clk;
5:
6:         if (dev) {
7:                 clk = of_clk_get_by_name(dev->of_node, con_id);
8:                 if (!IS_ERR(clk) && __clk_get(clk))
9:                         return clk;
10:        }
11:
12:        return clk_get_sys(dev_id, con_id);
13: }
```

> 如果提供了struct device指针，则调用of_clk_get_by_name接口，通过device tree接口获取clock指针。否则，如果没有提供设备指针，或者通过device tree不能正确获取clock，则进一步调用clk_get_sys。
>
> 这两个接口的定义如下。

2) of_clk_get_by_name

我们在“Linux common clock framework(2)_clock provider”中已经提过，clock consumer会在本设备的DTS中，以clocks、clock-names为关键字，定义所需的clock。系统启动后，device tree会简单的解析，以struct device_node指针的形式，保存在本设备的of_node变量中。

而of_clk_get_by_name，就是通过扫描所有“clock-names”中的值，和传入的name比较，如果相同，获得它的index（即“clock-names”中的第几个），调用of_clk_get，取得clock指针。

```
1: struct clk *of_clk_get_by_name(struct device_node *np, const char *name)
2: {
3:         struct clk *clk = ERR_PTR(-ENOENT);
4:
5:         /* Walk up the tree of devices looking for a clock that matches */
6:         while (np) {
7:                 int index = 0;
8:
9:                 /*
10:                 * For named clocks, first look up the name in the
11:                 * "clock-names" property.  If it cannot be found, then
12:                 * index will be an error code, and of_clk_get() will fail.
13:                 */
```

6~33行，是一个while循环，用于扫描所有的device_node；

14~15行，只要name不为空，管它三七二十一，直接以name为参数，去和“clock-names”匹配，获得一个index；

16~18行，以返回的index为参数，调用of_clk_get。这个index可能是invalid，不过无所谓，最糟糕就是不能获得clock指针。如果成功获取，则退出，或者继续；

19~22行，一个警告，如果name和index均合法，但是不能获得指针，则视为异常状况；

25~32行，尝试”clock-ranges“熟悉，比较冷门，不介绍它。

再看一下of_clk_get接口。

```
1: struct clk *of_clk_get(struct device_node *np, int index)
2: {
```

```
3:          struct of_phandle_args clkspec;
4:          struct clk *clk;
5:          int rc;
6:
7:          if (index < 0)
8:                  return ERR_PTR(-EINVAL);
9:
10:         rc = of_parse_phandle_with_args(np, "clocks", "#clock-cells", index,
11:                                         &clkspec);
12:         if (rc)
```

> 10~13行，通过of_parse_phandle_with_args接口，将index转换为struct of_phandle_args类型的参数句柄；
>
> 15行，调用of_clk_get_from_provider，获取clock指针。

of_clk_get_from_provider的实现位于drivers/clk/clk.c，通过便利of_clk_providers链表，并调用每一个provider的get回调函数，获取clock指针。如下：

```
1: struct clk *of_clk_get_from_provider(struct of_phandle_args *clkspec)
2: {
3:          struct of_clk_provider *provider;
4:          struct clk *clk = ERR_PTR(-ENOENT);
5:
6:          /* Check if we have such a provider in our array */
7:          mutex_lock(&of_clk_lock);
8:          list_for_each_entry(provider, &of_clk_providers, link) {
9:                  if (provider->node == clkspec->np)
10:                         clk = provider->get(clkspec, provider->data);
11:                 if (!IS_ERR(clk))
12:                         break;
13:         }
```

注3：分析到这里之后，consumer侧的获取流程已经很清晰，再结合“Linux common clock framework(2)_clock provider”中所介绍的of_clk_add_provider接口，整个流程都融汇贯通了。篇幅所限，有关of_clk_add_provider接口的实现，本文就不再分析了，感兴趣的读者可以自行阅读kernel代码。

3）clk_get_sys

clk_get_sys接口是在调用者没有提供struct device指针或者通过of_clk_get_xxx获取clock失败时，获取clock指针的另一种手段。基于kernel大力推行device tree的现状，蜗蜗不建议使用这种过时的手段，就不分析了。

4.2 clk_prepare/clk_unprepare

prepare和unprepare的的代码位于drivers/clk/clk.c中，分别由内部接口__clk_prepare和__clk_unprepare实现具体动作，如下：

```
1: int __clk_prepare(struct clk *clk)
2: {
3:         int ret = 0;
4:
5:         if (!clk)
6:                 return 0;
7:
8:         if (clk->prepare_count == 0) {
9:                 ret = __clk_prepare(clk->parent);
10:                if (ret)
11:                        return ret;
12:
13:                if (clk->ops->prepare) {
```

> prepare会维护一个prepare_count，用于记录prepare的次数。且在prepare_count为零时：
>
> 递归prepare自己的parent（有的话）；
>
> 调用clk ops中的prepare回调函数（有的话）。

unprepare类似，不再分析。

4.3 clk_enable/clk_disable

enable/disable和prepare/unprepare的实现逻辑基本一致，需要注意的是，enable/disable时如果prepare_count为0，则会报错并返回。

4.4 clock rate有关的实现

clock rate有关的实现包括get、set和round三类，让我们依次说明。

1）clk_get_rate负责获取某个clock的当前rate，代码如下：

```
1: /**
2:  * clk_get_rate - return the rate of clk
3:  * @clk: the clk whose rate is being returned
4:  *
5:  * Simply returns the cached rate of the clk, unless CLK_GET_RATE_NOCACHE flag
6:  * is set, which means a recalc_rate will be issued.
7:  * If clk is NULL then returns 0.
8:  */
9: unsigned long clk_get_rate(struct clk *clk)
10: {
11:         unsigned long rate;
12:
13:         clk_prepare_lock();
```

> a）如果该clock设置了CLK_GET_RATE_NOCACHE标志，获取rate前需要先调用__clk_recalc_rates接口，根据当前硬件的实际情况，重新计算rate。
> __clk_recalc_rates的逻辑是：如果提供了recalc_rate ops，以parent clock的rate为参数，调用该ops，否则，直接获取parent的clock值；然后，递归recalc所有child clock。
>
> b）调用__clk_get_rate返回实际的rate值。

2）clk_round_rate，返回该clock支持的，和输入rate最接近的rate值（不做任何改动），实际是由内部函数__clk_round_rate实现，代码如下：

```
1: unsigned long __clk_round_rate(struct clk *clk, unsigned long rate)
2: {
3:         unsigned long parent_rate = 0;
4:
5:         if (!clk)
6:                 return 0;
7:
8:         if (!clk->ops->round_rate) {
9:                 if (clk->flags & CLK_SET_RATE_PARENT)
10:                        return __clk_round_rate(clk->parent, rate);
11:                else
12:                        return clk->rate;
13:        }
```

a）18行，如果该clock提供了round_rate ops，直接调用该ops。

需要说明的是，round_rate ops接受两个参数，一个是需要round的rate，另一个时parent rate（以指针的形式提供）。它的意义是，对有些clock来说，如果需要得到一个比较接近的值，需要同时round parent clock，因此会在该指针中返回round后的parent clock。

b）9~10行，如果clock没有提供round_rate ops，且设置了CLK_SET_RATE_PARENT标志，则递归round parent clock，背后的思考是，直接使用parent clock所能提供的最接近的rate。

c）11~12，最后一种情况，直接返回原值，意味着无法round。

3）clk_set_rate

set rate的逻辑比较复杂，代码如下：

```
1: int clk_set_rate(struct clk *clk, unsigned long rate)
2: {
3:         struct clk *top, *fail_clk;
4:         int ret = 0;
5:
6:         /* prevent racing with updates to the clock topology */
7:         clk_prepare_lock();
8:
9:         /* bail early if nothing to do */
10:        if (rate == clk->rate)
11:                goto out;
12:
13:        if ((clk->flags & CLK_SET_RATE_GATE) && clk->prepare_count) {
```

a）9~16，进行一些合法性判断。

b）19~23行，调用clk_calc_new_rates接口，将需要设置的rate缓存在new_rate字段。

同时，获取设置该rate的话，需要修改到的最顶层的clock。背后的逻辑是：如果该clock的rate改变，有可能需要通过改动parent clock的rate来实现，依次递归。

c）25~23，发送rate将要改变的通知。如果有clock不能接受改动，即set rate失败，再发送rate更改停止的通知。

d）调用clk_change_rate，从最top的clock开始，依次设置新的rate。

注4：clock rate set有2种场景，一是只需修改自身的配置，即可达到rate set的目的。第二种是需要同时修改parent的rate（可向上递归）才能达成目的。看似简单的逻辑，里面却包含非常复杂的系统设计的知识。大家在使用clock framework，知道有这回事即可，并尽可能的不要使用第二种场景，以保持系统的简洁性。

4.5 clock parent有关的实现

parent操作包括get parent和set parent两类。
get parent的逻辑非常简单，直接从clk->parent指针中获取即可。

set parent稍微复杂，需要执行reparent和recalc_rates动作，具体不再描述了。

原创文章，转发请注明出处。蜗窝科技，www.wowotech.net。

标签: Linux framework clock 实现

« Linux时间子系统之（十五）： clocksource | 一个技术男眼中的Smartisan T1手机»

评论:

zoro
2017-03-28 00:13

请教一下学习方法，大神的3篇文章我都看了，对这个框架也有了些了解。
但是对于部分细节还是不清楚，于是打算在MTK平台上去实例分析。
看了一下MTK的clk类型有fixed rate、gate、mux、fixed factor还有一个pll不知道算不算。
本来是想把每个类型的注册都看看的，但是发现其实实现还是比较复杂的，也比较耗时间，
所以这里请教一下，对于每个类型的注册是不是都要去看？还是说了解了大体框架就可以呢?

回复

passerby
2015-04-16 10:28

wowo，我看了高通的code。发现高通好像没有做clk_register而是自己弄了一个msm_clock_register，用clk_lookup进行注册，你看过高通的这部分代码吗。

回复

wowo
2015-04-16 11:23

@passerby：内核版本是多少?

回复

passerby
2015-04-16 14:40

@wowo：linux版本是 3.10.49

回复

wowo

2015-04-16 15:15

@passerby：它确实没有使用clock框架，自己搞的！

回复

passerby

2015-04-16 16:01

@wowo：msm_clock_register是不是实现了clk_register里面的clock device tree呢?

回复

wowo

2015-04-16 17:14

@passerby：我只在arch/arm/mach-msm/中见到过msm的clock driver，但您这里所说的这个接口：msm_clock_register，我一直没有找到，不会是你们自己写的吧?

回复

passerby

2015-04-16 17:50

@wowo：。。。我们只是ODM厂商，没那个能力啊。你可以看看clock-cpu-8939.c，这个是高通CPU时钟注册，我从msm8939.dtsi中根据compatible找到的。今天我仔细看了一下，确实是高通自己弄了一套出来。

passerby

2015-04-16 18:11

@wowo：在/driver/clk/qcom下面可以看到clk_enable跟标准linux定义的都不同了，高通做了很多改动。把clk_prepare都自己重新写了一遍，但不知道为什么高通会花这些力气改内核都已经支持的框架。

```
int clk_enable(struct clk *clk)
{
    int ret = 0;
    unsigned long flags;
    struct clk *parent;
    const char *name;

    if (!clk)
        return 0;
    if (IS_ERR(clk))
```

```
        return -EINVAL;
    name = clk->dbg_name;

    spin_lock_irqsave(&clk->lock, flags);
    WARN(!clk->prepare_count,
         "%s: Don't call enable on unprepared clocks\n", name);
    if (clk->count == 0) {
        parent = clk->parent;

        ret = clk_enable(parent);
        if (ret)
            goto err_enable_parent;
        ret = clk_enable(clk->depends);
        if (ret)
            goto err_enable_depends;

        trace_clock_enable(name, 1, smp_processor_id());
        if (clk->ops->enable)
            ret = clk->ops->enable(clk);
        if (ret)
            goto err_enable_clock;
    }
    clk->count++;
    spin_unlock_irqrestore(&clk->lock, flags);

    return 0;

err_enable_clock:
    clk_disable(clk->depends);
err_enable_depends:
    clk_disable(parent);
err_enable_parent:
    spin_unlock_irqrestore(&clk->lock, flags);
    return ret;
}
```

wowo

2015-04-16 18:36

@passerby：单单从clk_enable看，貌似高通有parent不只一个的clock：
ret = clk_enable(clk->depends);
if (ret)
goto err_enable_depends;
可能是硬件比较奇怪吧！

回复

wzw200
2015-03-06 10:38

WO 你好：
kernel\arch\arm\XX\clock.c
1：clk_register(struct clk *clk)

\kernel\drivers\clk\clk.c
2：struct clk *clk_register(struct device *dev, struct clk_hw *hw)

函数1是旧内核，2是新内核

我现在用的板子，我看他们用函数1，自己维护了时钟树，
函数2，是不是系统帮我们维护时钟树了，我们只管调用这个函数注册就可以了是吗

回复

wowo
2015-03-06 10:54

@wzw200：是的，最好都用kernel的框架，因为它帮我们做了很多事情。

回复

wzw200
2015-03-06 11:08

@wowo：板级代码里面 clock.c 有一个这样的函数
int clk_register(struct clk *clk,const char *parent)
{.....}
EXPORT_SYMBOL(clk_register);

驱动\drivers\clk\clk.c
struct clk *clk_register(struct device *dev, struct clk_hw *hw)
{.....}

EXPORT_SYMBOL_GPL(clk_register);
都用EXPORT_SYMBOL导出了，我系统里面这两个函数都在使用，想不明白他们区别
在看一个SOC amlogic厂的代码

回复

wowo
2015-03-06 12:21

@wzw200：应该不会这样，你再看看，是否有一个没有编译。

回复

linux初学者
2014-12-02 16:50

对于fixed rate clocks，若使用DTS，kernel是如何“自动”注册的？能否再详细一点啊？是否还需要用户的driver去调用of_clk_init（NULL）？

回复

wowo
2014-12-02 17:26

@linux初学者：抱歉，我写的不清楚。在我用的ARM64里面，kernel的setup代码会帮忙调用这个接口，如下：

```
static int __init arm64_device_init(void)
{
	of_clk_init(NULL);
	of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);
	return 0;
}
arch_initcall_sync(arm64_device_init);
```

可是对于其它的ARCH，则需要自行调用。
非常感谢，我会再更新我的描述。

回复

wowo
2014-12-02 18:02

@wowo：不过在最新的kernel里，都在time_init中调用了。

```
void __init time_init(void)
{
	u32 arch_timer_rate;
```

```
        of_clk_init(NULL);
        clocksource_of_init();

        tick_setup_hrtimer_broadcast();

        arch_timer_rate = arch_timer_get_rate();
        if (!arch_timer_rate)
                panic("Unable to initialise architected timer.\n");

        /* Calibrate the delay loop directly */
        lpj_fine = arch_timer_rate / HZ;
}
```

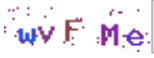
回复

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)



发表评论