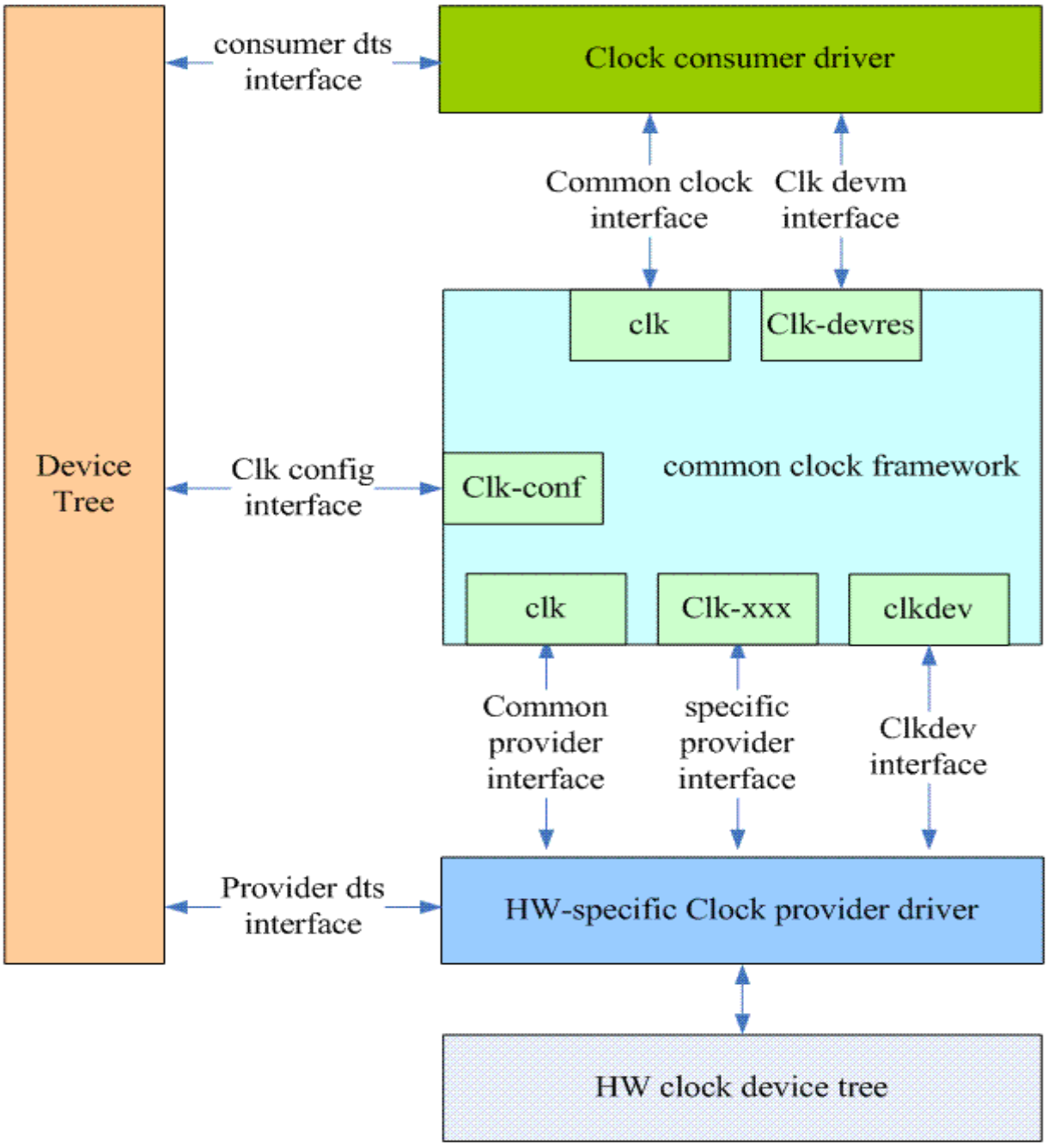
Common Clock Framework 系统结构

一、前言

之前，wowo 同学已经发表了关于 CCF（Common Clock Framework）的三份文档，相信大家对 CCF 有一定的了解了，本文就是在阅读那三份文档的基础上，针对 Linux 4.4.6 内核的内核代码实现，记录自己对 CCF 的理解，并对 CCF 进行系统结构层面的归纳和整理。
本文内容包括三个部分，第二章给出了整个 CCF 相关的 block diagram 图，随后在第三章对各个模块进行功能层面的描述。最后，第四章给出了各个 block 之间的接口描述。
另外，在阅读 CCF 代码的过程中，我准备用两份文档来分享我对 CCF 的理解。这一份是系统结构，另外一份是逻辑解析。

二、软件框架

CCF 的框架如下图所示：



上图中，位于图片中心的那个青色的 Block 就是 CCF 模块，和硬件无关，实现了通用的 clock 设备的逻辑。其他的软件模块都不属于 CCF 的范畴，但是会和 CCF 有接口进行通信。随后的两个章节会对各个 block 以及接口进行描述。

三、软件框架图中各个 block 的功能描述

1、HW clock device tree

Clock device 是指那些能够产生 clock 信号、控制 clock 信号的设备，例如晶振、分频器什么的，本小节主要是从 HW 工程师的角度来描述这些设备极其拓扑结构。系统中的 clock distribution 形成了类似文件系统那样的树状结构，和文件系统树不同的是：clock tree 有多个根节点，形成多个 clock tree，而文件系统树只有一个根节点；文件系统树的中间节点是目录，叶节点是文件，而 clock tree 相对会复杂一些，它包括如下的的节点：
（1）根节点一般是 Oscillator（有源振荡器）或者 Crystal（无源振荡器，即大家经常说的晶振）。
（2）中间节点有很多种，包括 PLL（锁相环，用于提升频率的），Divider（分频器，用于降频的），mux（从多个 clock path 中选择一个），开关（用来控制 ON/OFF 的）。
（3）叶节点是使用 clock 做为输入的、有具体功能的 HW block。
了解了 clock tree 的结构之后，我们来看看具体的操作。对于叶节点，或者说 clock consumer 而言，没有什么可以控制的，HW block 只是享用这个 clock source 而已。这里需要注意的是：虽然 HW clock 的 datasheet 往往有 clock gating 的内容（或者一些针对 clock source 进行分频的内容，概念类似），但是，本质上负责 clock gating 的那个硬件模块需要独立出来，称为 clock tree 中的一个中间节点。而对中间节点的设定包括：
（1）ON/OFF 控制
（2）频率设定、相位控制等
（3）从众多输入的 clock source 中选择一个做为输出。

2、HW-specific Clock provider driver

这个模块是真正和系统中实际的 clock device 打交道的模块。与其说 Clock provider driver，不如说是 clock provider drivers（复数），主要用来驱动系统中的 clock tree 中的各个节点上 clock device（不包括 clock consumer device），只要该 HW block 对外提供时钟信号，那么它就是一个 clock provider，就有对应的 clock provider driver。
上一小节我们已经了解到，系统中的 clock device 非常多种，什么 VCO 啦、什么分频器啦，什么复用器啦，其功能各不相同，但是本质上都属于 clock device，Linux kernel 把这些 clock HW block 的特性抽取出来，用 struct clk_hw 来表示，具体如下：

```
struct clk_hw {
    struct clk_core *core;————————（1）
    struct clk *clk;————————————（2）
    const struct clk_init_data *init;————（3）
};
```

（1）指向 CCF 模块中对应 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。
（2）clk 是访问 clk_core 的实例，每当 consumer 通过 clk_get 对 CCF 中的 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。
（3）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，具体 struct clk_init_data 的定义如下：

```
struct clk_init_data {
    const char      *name;————————————————（1）
    const struct clk_ops    *ops;————————————————（2）
```

```
    const char        * const *parent_names;————————（3）
    u8            num_parents;
    unsigned long         flags;————————————————（4）
};
```

（1）该 clock 设备的名字。

（2）consumer 访问 clock 设备的时候往往是首先获取 clk 句柄，然后找到 clk_core，之后即可通过 clk_core 的 struct clk_ops 中的 callback 函数可以进入具体的 clock provider driver 层进行具体的 HW 操作。struct clk_ops 这个数据结构是底层驱动必须要准备好的数据之一，在注册的时候通过 clk_init_data 带入 CCF 层。

（3）描述该 clk_hw 的拓扑结构，通过这样的信息，CCF 层可以建立 clk_core 的拓扑结构来跟踪实际 clock 设备的拓扑。

（4）CCF 级别的 flag。

struct clk_hw 应该是 clock device 的基类，所有的具体的 clock device 都应该由它派生出来，例如固定频率的振动器，它的数据结构是：

```
struct    clk_fixed_rate {
    struct        clk_hw hw;————————————基类
    unsigned long     fixed_rate;————————下面是 fixed rate 这种 clock device 特有的成员
    unsigned long     fixed_accuracy;
    u8        flags;
};
```

其他的特定的 clock device 大概都是如此，这里就不赘述了。在构建自己硬件平台的 clock provider 驱动的时候，如果能够使用 clock device 的派生类，尽量不要使用 struct clk_hw 这个基类，尽量不要重复造轮子。

3、CCF 模块

CCF 模块，wowo 的文章已经说了很多，我这里从文件的角度来说一说该模块。CCF 模块的文件汇整如下：

文件名	类别	对应的框架图中的元素	描述
clk.h	接口	common clock interface clk devm interface	为各种 clock consumer 模块提供通用的 clock 控制接口
clkdev.h	接口	clkdev interface	为各种 clock provider 模块提供的 clkdev 接口，主要用于 consumer 寻找 CCF 中的 clk 数据结构。
clk-provider.h	接口	common provider interface specific provider interface	为各种硬件相关的的 clock provider 模块提供注册、注销的接口。
clk.c	模块	clk	CCF 的核心模块，对上提供查找，管理各种 clk 实例的功能，对内，提供管理和各种 clock device（struct clk_core）的功能，对下，接收来自底层硬件的注册注销请求，维护 clk、clk_core 和 clk_hw 之间的连接。
clkdev.c	模块	devclk	该模块主要用来维护 clk 结构和其对应的 clk 名字之间的关系，用于 clk 的 lookup。
clk-devres.c	模块	clk-devres	设备自动管理 clk 资源模块（参考 device resource management）
clk-conf.c	模块	clk-conf	该模块主要用来实现在启动阶段的时候，通过分析 DTS 来设定一下 default 参数
clk-xxx.c	模块	clk-xxx	这些模块包括：clk-divider.c clk-fixed-factor.c clk-fixed-rate.c clk-gate clk-multiplier.c clk-mux.c clk-composite.c clk-fractional-divider.c clk-gpio.c。这些都是 CCF 模块对某一类 clock 设备进行抽象，方便具体 clock provider driver 工程师撰写 driver 的时候不至于从头开始。

drivers/clk 目录其他文件	N/A	N/A	这些文件都是具体硬件平台上的各种 clock provider 驱动

此外，CCF 模块的核心数据结构就有两个：

（1）struct clk_core。这个数据结构是 CCF 层对 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，CCF 模块负责建立整个抽象的 clock tree 的树状结构并维护这些数据。具体如何维护呢？这里不得不给出几个链表头的定义，如下：

```
static HLIST_HEAD(clk_root_list);
static HLIST_HEAD(clk_orphan_list);
```

CCF layer 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，而这个链表中的每一个阶段又展成一个树状结构。（这和硬件拓扑是吻合的，clock tree 实际上是有多个根节点的，多条树状结构）。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。

（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。CCF 如何管理 clk 呢？这里说来就话长了，在 DTS 还没有引入 kenrel 的时代，struct clk_lookup 用来管理 clk 数据，具体该数据结构定义如下：

```
struct clk_lookup {
    struct list_head    node;－－－－－－－挂入 clocks 链表
    const char      *dev_id;－－－－－－－dev_id 和 con_id 是用来寻找适合的 clk 的
    const char      *con_id;
    struct clk      *clk;－－－－－－－－－对应的 clk
    struct clk_hw       *clk_hw;
};
```

对于底层的 clock provider 驱动而言，除了调用 clk_register 函数注册到 common clock framework 中，还会调用 clk_register_clkdev 将该 clk 和一个名字捆绑起来（否则 clock consumer 并不知道如何定位到该 clk），在 CCF layer，clocks 全局链表用来维护系统中所有的 struct clk_lookup 实例。通过这样的机制，clock consumer 可以通过名字获取 clk。引入 device tree 之后，情况发生了一些变化。基本上每一个 clock provider 都会变成 dts 中的一个节点，也就是说，每一个 clk 都有一个设备树中的 device node 与之对应。在这种情况下，与其捆绑 clk 和一个“名字”，不如捆绑 clk 和 device node，具体的数据结构如下：

```
struct of_clk_provider {
    struct list_head link;  －－－－－－－挂入 of_clk_providers 全局链表
    struct device_node *node;－－－－该 clock device 的 DTS 节点
    struct clk *(*get)(struct of_phandle_args *clkspec, void *data);－－获取对应 clk 数据结构的函数
    void *data;
};
```

因此，对于底层 provider driver 而言，原来的 clk_register + clk_register_clkdev 的组合变成了 clk_register + of_clk_add_provider 的组合，在 CCF layer 保存了 of_clk_providers 全局链表来管理所有的 DTS 节点和 clk 的对应关系。我们再看 clock consumer 这一侧：这时候，使用名字检索 clk 已经过时了，毕竟已经有了强大的 device tree。我们可以通过 clock consumer 对应的 struct device_node 寻找为他提供 clock signal 那个 clock 设备对应的 device node（clock 属性和 clock-names 属性），当然，如果 consumer 有多个 clock signal 来源，那么在寻找的时候需要告知是要找哪一个时钟源（用 connection ID 标记）。当找了 provider 对应的 device node 之后，一切都变得简单了，从全局的 clock provide 链表中找到对应 clk 就 OK 了。在引入强大的设备树之后，clkdev 模块按理说应该退出历史舞台了。

4、clock consumer driver

对于 clock consumer driver 而言，它其实就是调用 CCF 接口和 DTS 的接口来完成下面的功能：

（1）初始化的时候，调用 common clk interface 来对其 clock 设备（如果有需要，可能波及 clock 设备的上游设备）进行设定，以便让该 HW block 正常运作起来。

（2）根据用户需求（例如用户修改波特率），在运行时，clock consumer driver 也会调用 common clk interface 来对其 clock 设备进行修改（例如修改 clock rate，相位等）
（3）配合系统电源管理（TODO）。

5、设备树

具体请参考本站的相关文档。

四、接口描述

1、consumer dts interface

clock consumer 的属性列表如下：

属性	是否必须提供？	描述
clocks	必须	该属性描述 clock consumer 设备使用的 clock source，或者 clock input（可能不止一个哦）。 该属性是一个数组，数组中每一个具体的 entry 对应一个 clock source。而 clock source 是由 phandle 和 clock specifier 来描述。phandle 指向一个 clock provider 的 device node，如果该 provider 的#clock-cells 等于 0，那么说明该 provider 就一个 output，那么就不需要 clock specifier 来进一步描述。如果该 provider 的#clock-cells 不等于 0，那么 clock specifier 必须提供，以便指明本设备到底使用 provider 输出时钟源的哪一路。
clock-names	可选	同样的，该属性也似描述设备使用的 clock source 信息的，也是一个数组，是一个字符串数组，每一个字符串描述一个 clock source，对应着 clocks 中 phandle 和 clock specifier。 之所以提供 clock-names 这个属性其实是为了编程方便，驱动程序可以通过比较直观的 clock name 来找到该设备的输入时钟源信息。
clock-ranges	可选	该属性值为空，主要用来说明该设备的下级设备可以继承该设备的 clock source。例如 B 设备是 A 设备的 sub node，A 设备如果有 clock-ranges 属性，那么 B 设备在寻找其 clock source 的时候，如果在本 node 定义的 clock 相关属性中没有能够找到，那么可以去 A 设备去继续寻找（也就是说，B 设备会继承 A 设备的 clock source 相关的属性，也就是 clocks 或者 clock-names 这两个属性）。

2、provider dts interface

clock provider 的属性列表如下：

属性	是否必须提供？	描述
#clock-cells	必须	我们上面说过了，一个 HW block（clock consumer）的时钟源可以通过 phandle 和 clock specifier 来描述，phandle 指向一个 clock provider 的 device node，很明确，但是定位到 clock provider 并不行，因此有的 provider 会提供多路 clock output 给其他 HW block 使用，因此需要所谓的 clock specifier 来进一步描述。这里#clock-cells 就是说明使用多少个 cell（u32）来描述 clock

		specifier。 如果等于 0，说明 provider 就一个 clock output，不需要 specifier，如果等于 1，说明 provider 有多个 clock output（能用 u32 标识）。等于 2 或者更大的情况应该不存在，一个 provider 不可能提供超过 2^32 个 clock output。
clock-output-names	可选	如果 clock provider 能提供多路时钟输出，那么给每一个 clock output 起个适合人类阅读的名字是不错的选择，这也就是 clock-output-names 的目的。clock consumer 中提供的 clock specifier 是一个 index，通过这个 index 可以在 clock-output-names 属性值中找到对应的时钟源的名字。
clock-indices	可选	如果不提供这个属性，那么 clock-output-names 和 index 的对应关系就是 0，1，2……。如果这个对应关系不是线性的，那么可以通过 clock-indices 属性来定义映射到 clock-output-names 的 index。

3、clock config interface

初始化时可以通过 dts 来设定 clock parent 以及 clock rate，具体属性如下：

属性	是否必须提供？	描述
assigned-clocks	可选	这个属性列出了需要进行设定的 clock，其值是一个 phandle＋clock specifier 数组
assigned-clock-parent	可选	准备要设定的 parent 列表。“儿子”在哪里呢？assigned-clocks 中定义的，注意，是一一对应的。例如： assigned-clocks：A， B，C； assigned-clock-parent：A_parent，B_parent，C_parent；
assigned-clock-rate	可选	要设定的频率列表，同样的，和 assigned-clocks 也是一一对应的。

4、其他接口

请参考窝窝同学的文档，这里不再赘述。

五、参考文献

[1] Documentation/devicetree/bindings/clock/clock-bindings.txt

[2] Documentation/clk.txt