# 系统框架

CCF（CommonClock Framework）子系统的框架如下。时钟源驱动作为clk providr完成时钟的注册，设备驱动则作为clk consumer获取并使用时钟，而core则完成clk providr与clk consumer的隔离。



# 基本数据结构

硬件时钟源即clock provider被抽象成如下结构：

struct clk\_hw {

struct clk\_core \*core;

struct clk \*clk;

const struct clk\_init\_data \*init;//用于初始化clk\_core，注册clock时需提供

};

struct clk\_init\_data {

const char \*name;//时钟名称

const struct clk\_ops \*ops;//时钟的操作集

const char \* const \*parent\_names;//父时钟名称列表

u8 num\_parents;//父时钟个数

unsigned long flags;//标志，CLK\_IS\_BASIC等

};

clock的操作集定义如下：

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);

int (\*save\_context)(struct clk\_hw \*hw);

void (\*restore\_context)(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 rate, unsigned long \*parent\_rate);

int (\*determine\_rate)(struct clk\_hw \*hw,struct clk\_rate\_request \*req);

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 rate, unsigned long parent\_rate);

int (\*set\_rate\_and\_parent)(struct clk\_hw \*hw, unsigned long rate, unsigned long parent\_rate, u8 index);

unsigned long (\*recalc\_accuracy)(struct clk\_hw \*hw,unsigned long parent\_accuracy);

int (\*get\_phase)(struct clk\_hw \*hw);

int (\*set\_phase)(struct clk\_hw \*hw, int degrees);

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

int (\*debug\_init)(struct clk\_hw \*hw, struct dentry \*dentry);

};

clk\_core结构在CCF框架内部使用，包含了clock provider的操作方法、支持的时钟频率信息等：

struct clk\_core {

const char \*name;

const struct clk\_ops \*ops;

struct clk\_hw \*hw;

struct module \*owner;

struct clk\_core \*parent;

const char \*\*parent\_names;

struct clk\_core \*\*parents;

u8 num\_parents;

u8 new\_parent\_index;

unsigned long rate;

unsigned long req\_rate;

unsigned long new\_rate;

struct clk\_core \*new\_parent;

struct clk\_core \*new\_child;

unsigned long flags;

bool orphan;

unsigned int enable\_count;

unsigned int prepare\_count;

unsigned long min\_rate;

unsigned long max\_rate;

unsigned long accuracy;

int phase;

struct hlist\_head children;

struct hlist\_node child\_node;

struct hlist\_head clks;

unsigned int notifier\_count;

struct kref ref;

};

时钟的consumer由如下结构表示：

struct clk {

struct clk\_core \*core;

const char \*dev\_id;

const char \*con\_id;

unsigned long min\_rate;

unsigned long max\_rate;

struct hlist\_node clks\_node;

};

上述结构之间的关系如下：



# clock注册

clock的注册通过如下函数完成，将struck clk\_hw注册到内核，此时就会创建struct clk\_core：

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

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

相应的注销函数如下：

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

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

clock的注册还完成了父子clk provider的关联，最终就会形成时钟树：

* 如果该clk\_core父节点存在，则将该clk\_core插入到父节点的children链表中
* 如果该clk\_core为root节点，则将该clk\_core插入到clk\_root\_list链表中
* 如果该clk\_core父节点不存在且不为root节点，则将该clk\_core插入到clk\_orphan\_list链表中
* 遍历链表clk\_orphan\_list中的clk\_core，若其父节点为该clk\_core，则将其从clk\_orphan\_list链表上移除，并加入到该clk\_core的children链表中

根据时钟种类的不同，CCF子系统将时钟分为fixed rate、gate、divider、mux、fixed factor、fractional divider、composite clock，不同种类的时钟注册方法不一致，当然底层还是调用clk\_register()，但是此时无需我们自己实现struct clk\_ops了，内核为每类时钟提供了统一的操作方法，为每一类时钟自动填充clk\_ops的此类时钟所需的成员。

## fixed rate clock

这一类时钟具有固定的频率，不能开关、不能更改频率、不能设置parent、无需提供任何的clk\_ops回调函数，是最简单的一类时钟，晶振就是常用的fixed-rate clock。

fixed-rate clock被抽象成如下结构：

struct clk\_fixed\_rate {

struct clk\_hw hw;

unsigned long fixed\_rate;

unsigned long fixed\_accuracy;

u8 flags;

};

相应的注册与注销方法如下，具体实现在clk-fixed-rate.c中：

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

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

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 \*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);

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

void clk\_hw\_unregister\_fixed\_rate(struct clk\_hw \*hw);

clk\_register\*和clk\_hw\_register\*区别不大，只是一个返回的是struct clk另一返回的是struct clk\_hw，下同。

## gate clock

这一类时钟只可开关，clk\_ops需提供enable、disable、is\_enabled回调。

gate clock被抽象成如下结构：

struct clk\_gate {

struct clk\_hw hw;

void \_\_iomem \*reg;

u8 bit\_idx;

u8 flags;

spinlock\_t \*lock;

};

相应的注册与注销方法如下，具体实现在clk-gate.c中：

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 \*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);

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

void clk\_hw\_unregister\_gate(struct clk\_hw \*hw);

## divider clock

这一类时钟可以设置分频值，clk\_ops需提供recalc\_rate、set\_rate、round\_rate回调。

divider clock被抽象成如下结构：

struct clk\_divider {

struct clk\_hw hw;

void \_\_iomem \*reg;

u8 shift;

u8 width;

u8 flags;

const struct clk\_div\_table \*table;

spinlock\_t \*lock;

};

相应的注册与注销方法如下，具体实现在clk-divider.c中：

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 \*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, spinlock\_t \*lock);

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);

struct clk\_hw \*clk\_hw\_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);

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

void clk\_hw\_unregister\_divider(struct clk\_hw \*hw);

## mux clock

这一类时钟可以选择多个parent，clk\_ops需提供get\_parent、set\_parent、recalc\_rate回调。

mux clock被抽象成如下结构：

struct clk\_mux {

struct clk\_hw hw;

void \_\_iomem \*reg;

u32 \*table;

u32 mask;

u8 shift;

u8 flags;

spinlock\_t \*lock;

u8 saved\_parent;

};

相应的注册与注销方法如下，具体实现在clk-mux.c中：

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

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

struct clk \*clk\_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\_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);

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

void clk\_hw\_unregister\_mux(struct clk\_hw \*hw);

## fixed factor clock

这一类时钟具有固定的倍频mul和分频div，clock的频率等于parent clock的频率乘以mul除以div，clk\_ops需提供recalc\_rate、set\_rate、round\_rate回调。

fixed-factor clock被抽象成如下结构：

struct clk\_fixed\_factor {

struct clk\_hw hw;

unsigned int mult;

unsigned int div;

};

相应的注册与注销方法如下，具体实现在clk-fixed-factor.c中：

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);

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);

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

void clk\_hw\_unregister\_fixed\_factor(struct clk\_hw \*hw);

## fractional divider clock

这一类时钟与divider clock的区别是Output rate = (m / n) \* parent\_rate，分频数是分数，clk\_ops需提供recalc\_rate、set\_rate、round\_rate回调。

fractional-divider clock被抽象成如下结构：

struct clk\_fractional\_divider {

struct clk\_hw hw;

void \_\_iomem \*reg;

u8 mshift;

u8 mwidth;

u32 mmask;

u8 nshift;

u8 nwidth;

u32 nmask;

u8 flags;

spinlock\_t \*lock;

};

相应的注册与注销方法如下，具体实现在clk-fractional-divider.c中：

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

struct clk\_hw \*clk\_hw\_register\_fractional\_divider(struct device \*dev, const char \*name, const char \*parent\_name, unsigned long flags, void \_\_iomem \*reg, u8 mshift, u8 mwidth, u8 nshift, u8 nwidth, u8 clk\_divider\_flags, spinlock\_t \*lock);

void clk\_hw\_unregister\_fractional\_divider(struct clk\_hw \*hw);

## composite clock

这一类时钟是上述mux、divider、gate时钟的组合。

composite clock被抽象成如下结构：

struct clk\_composite {

struct clk\_hw hw;

struct clk\_ops ops;

struct clk\_hw \*mux\_hw;

struct clk\_hw \*rate\_hw;

struct clk\_hw \*gate\_hw;

const struct clk\_ops \*mux\_ops;

const struct clk\_ops \*rate\_ops;

const struct clk\_ops \*gate\_ops;

};

相应的注册与注销方法如下，具体实现在clk-fractional-divider.c中，clk\_ops可直接使用内核已经定义好的clk\_mux\_ops、clk\_fixed\_rate\_ops、clk\_gate\_ops：

struct clk \*clk\_register\_composite(struct device \*dev, const char \*name, const char \* const \*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);

struct clk\_hw \*clk\_hw\_register\_composite(struct device \*dev, const char \*name, const char \* const \*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);

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

void clk\_hw\_unregister\_composite(struct clk\_hw \*hw);

# clock使用

## 获取

使用clock前需要获取时钟，即得到struct clk。

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

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

dev和id任意一个可以为空，如果id为空，则必须有dts的支持。

相比clk\_get把device替换为device的name：

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

dts相关接口：

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);

clk的释放如下：

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

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

## 配置

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

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

int clk\_enable(struct clk \*clk);

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

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

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

时钟的使能/禁用分为两步，可能会睡眠的放到prepare里，不能睡眠的放到enable里。clk\_prepare\_enable是prepare和enable的组合。

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

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

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

时钟频率的获取和设置。其中clk\_set\_rate可能会因为没有对应分频数而设置失败，可以先调用clk\_round\_rate获取与所需rate最近的值在进行设置。

int clk\_set\_parent(struct clk \*clk, struct clk \*parent);

struct clk \*clk\_get\_parent(struct clk \*clk);

设置/获取时钟的parent clock。

## clock查找

我们按照上述的方法注册clock provider，然后clock consumer调用clk\_get来获取clk，发现并不能获取到clk。此时就需要分析一下clk\_get了。

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

{

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

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

}

clk\_get中还有使用dts时的clk获取方法，暂时按下不表，可以看到clk\_get其实也是通过clk\_get\_sys获取。

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

{

struct clk\_lookup \*cl = clk\_find(dev\_id, con\_id);

clk = \_\_clk\_create\_clk(cl->clk\_hw, dev\_id, con\_id);

}

clk\_find如下，其实就是遍历clocks链表，匹配dev\_id或con\_id名称一致的clk\_lookup：

static struct clk\_lookup \*clk\_find(const char \*dev\_id, const char \*con\_id)

{

list\_for\_each\_entry(p, &clocks, node)

if (p->dev\_id) {

if (!dev\_id || strcmp(p->dev\_id, dev\_id))

continue;

match += 2;

}

if (p->con\_id) {

if (!con\_id || strcmp(p->con\_id, con\_id))

continue;

match += 1;

}

}

而向clocks添加链表项需要额外调用函数：

clk\_register\_clkdev

vclkdev\_create

vclkdev\_alloc

\_\_clkdev\_add

list\_add\_tail(&cl->node, &clocks);

**所以我们在注册完clock后还需要调用clk\_register\_clkdev向时钟的查找表上添加项**。

# clock与DTS

## 配置与获取

clock provider的dts示例如下，ref25是该clock设备的名称，clock consumer通过该名称引用clock，#clock-cells如果为0表示该clock只有1个输出，如果为1表示该clock有多个输出，clock consumer需要通过ID指定需要使用的clock。

ref25: ref25M {

compatible = "fixed-clock";

#clock-cells = <0>;

clock-frequency = <25000000>;

};

clock consumer的dts示例如下，clocks为该设备的clock列表，clocks需提供的信息由clock provider的#clock-cells决定，为0时直接引用，为1时还需额外提供ID指定使用的是哪路输出。clock-names是clocks中的clock的别名，clock\_get时可通过此名称获取clk。

si5351a: clock-generator@60 {

clocks = <&ref25>;

clock-names = "xtal";

};

上面在分析clk\_get跳过了使用dts时的clk获取方式，现在来分析：

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

{

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

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

while (np) {

if (name)

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

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

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

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

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

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

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

}

}

}

}

可以看到会遍历设备树节点，根据名称获取节点信息，然后遍历of\_clk\_providers找到device\_node一致的provider，然后调用\_\_of\_clk\_get\_hw\_from\_provider()获取clk\_hw：

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

{

struct clk \*clk;

if (provider->get\_hw)

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

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

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

}

如果of\_clk\_provider提供了get\_hw方法则使用get\_hw，可直接获取clk\_hw，否则使用get方法，获取完clk再转化成clk\_hw。所以现在的问题就是of\_clk\_providers的成员在哪添加以及of\_clk\_provider的get\_hw/get方法如何实现。

向of\_clk\_providers添加链表项的方法有如下两个，区别就是一个提供of\_clk\_provider的get\_hw方法，另一个则是get方法。

int of\_clk\_add\_hw\_provider(struct device\_node \*np, struct clk\_hw \*(\*get)(struct of\_phandle\_args \*clkspec, void \*data), void \*data)

{

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

}

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

{

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

}

**所以在dts时，注册完clock还需调用of\_clk\_add\_hw\_provider或者of\_clk\_add\_provider向查找表上添加项。**

内核已提供了基本的get\_hw/get方法，可在调用of\_clk\_add\_hw\_provider/of\_clk\_add\_provider时直接使用：

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

如果只是将1个clk添加到查找表上，of\_clk\_add\_provider提供的data为clk指针，此函数直接将data返回。

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

同上，只是clk变为clk\_hw。

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

对于将多个clk添加到查找表的情况，需定义struct clk\_onecell\_data，of\_clk\_add\_provider提供的data就是struct clk\_onecell\_data指针，of\_clk\_src\_onecell\_get根据clkspec->args[0]即索引返回对应clk。

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

同上，只是clk\_onecell\_data变成了clk\_hw\_onecell\_data。

当然get\_hw/get方法可以自己实现，根据clkspec->args[0]提供的索引返回对应clk/clk\_hw即可。

## CLK\_OF\_DECLARE

时钟初始化经常会用到CLK\_OF\_DECLARE宏，比如clk-fixed-rate.c中有CLK\_OF\_DECLARE(fixed\_clk, "fixed-clock", of\_fixed\_clk\_setup)：

#define CLK\_OF\_DECLARE(name, compat, fn) OF\_DECLARE\_1(clk, name, compat, fn)

OF\_DECLARE\_1 的定义：

#define OF\_DECLARE\_1(table, name, compat, fn) \_OF\_DECLARE(table, name, compat, fn, of\_init\_fn\_1)

如果定义了CONFIG\_OF，\_OF\_DECLARE定义如下：

#define \_OF\_DECLARE(table, name, compat, fn, fn\_type) \

static const struct of\_device\_id \_\_of\_table\_##name \

\_\_used \_\_section(\_\_##table##\_of\_table) \

= { .compatible = compat, \

.data = (fn == (fn\_type)NULL) ? fn : fn }

所以CLK\_OF\_DECLARE(fixed\_clk, "fixed-clock", of\_fixed\_clk\_setup)展开后为：

static const struct of\_device\_id \_\_of\_table\_fixed\_clk \_\_used \_\_section(\_\_clk\_of\_table) \

= { .compatible = "fixed-clock", \

.data = of\_fixed\_clk\_setup }

其中\_\_section为\_\_attribute\_\_ ((\_\_section\_\_(#S)))，表示将变量定义到指定的输入段中，\_\_used为\_\_attribute\_\_((\_\_used\_\_))，表示告诉编译器在目标文件中保留一个静态函数或者静态变量，即使它没有被引用，避免被编译器优化掉。所以CLK\_OF\_DECLARE(fixed\_clk, "fixed-clock", of\_fixed\_clk\_setup)就是定义了struct of\_device\_id \_\_of\_table\_fixed\_clk，并将其放到\_\_clk\_of\_table下，\_\_clk\_of\_table会在clk.c的 of\_clk\_init()中引用。

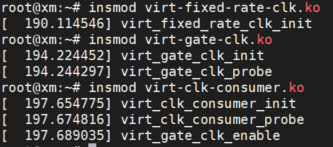
内核已经完成了fixed-clock以及fixed-factor-clock的驱动。对于fixed clock，设备树的compatible固定填充"fixed-clock"，并提供"clock-frequency"、"clock-accuracy"和"clock-output-names"关键字。对于fixed factor clock，设备树的compatible固定填充"fixed-factor-clock"，并提供"clock-div"、"clock-mult"和"clock-output-names"关键字。

# 示例说明

示例虚拟一个简单的时钟树：



如果不使用设备树，virt-gate-clk.c和virt-clk-consumer.c中USE\_DTS宏要注释掉：



如果使用设备树，virt-gate-clk.c和virt-clk-consumer.c中USE\_DTS宏要放开，此时virt-fixed-rate-clk.ko就无须insmod了，上面已经讲过内核已经实现fixed-clock的驱动，设备树信息如下：

virt\_osc: virt\_osc{

compatible = "fixed-clock";

#clock-cells = <0>;

clock-frequency = <25000000>;

};

virt\_gate\_clk: gate\_clk{

compatible = "xm,virt-gate-clk";

clocks = <&virt\_osc>;

#clock-cells = <0>;

clock-names = "osc";

};

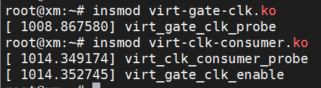
virt\_clk\_consumer {

compatible = "xm,virt-clk-consumer";

clocks = <&virt\_gate\_clk>;

clock-names = "virt\_gate";

};



通过clk\_summary可以查看到当前的时钟树信息：

