

# Rockchip

## 时钟子模块 开发指南

发布版本:1.00

日期:2016.06

# 前言

## 概述

本文档主要介绍 RK 平台时钟子系统框架介绍以及配置。

## 产品版本

与本文档相对应的产品版本如下。

产品名称	内核版本
RK3288	Linux4.4
RK3366	Linux4.4
RK3368	Linux4.4
RK3399	Linux4.4

## 读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

## 修订记录

日期	版本	作者	修改说明
2016-06-30	V1.0	Elaine	第一次临时版本发布

# 目录

前言 .....	I
目录 .....	II
插图目录.....	III
表格目录.....	IV
1 方案概述 .....	1-1
1.1 概述.....	1-1
1.2 重要概念 .....	1-1
1.3 时钟方案 .....	1-1
1.4 总体流程 .....	1-2
1.5 代码结构 .....	1-3
2 CLOCK 开发指南 .....	2-1
2.1 概述.....	2-1
2.2 时钟的相关概念 .....	2-1
2.2.1 PLL.....	2-1
2.2.2 ACLK、PCLK、HCLK.....	2-1
2.2.3 GATING.....	2-2
2.3 时钟配置 .....	2-2
2.3.1 时钟初始化配置 .....	2-2
2.3.2 时钟 ID.....	2-4
2.3.3 主要的 CLK 注册类型函数 .....	2-4
2.3.4 Driver 的时钟配置 .....	2-5
2.4 CLOCK API 接口 .....	2-5
2.4.1 主要的 CLK API .....	2-5
2.4.2 示例 .....	2-6
2.5 CLOCK 调试.....	2-7

# 插图目录

图 1-1 CLK 时钟树的示例图..... 1-1

图 1-2 时钟分配示例图..... 1-2

图 1-3 时钟配置流程图..... 1-2

图 2-1 总线时钟结构 ..... 2-1

图 2-2 GATING 示例图..... 2-2

# 表格目录

表 1-1 CLK 代码构成.....	1-3
表 2-1 CLK PLL 描述 .....	2-1

# 1 方案概述

## 1.1 概述

本章主要描述时钟子系统的相关的重要概念、时钟方案、总体流程、代码结构。

## 1.2 重要概念

### ● 时钟子系统

这里讲的时钟是给 SOC 各组件提供时钟的树状框架，并不是内核使用的时间，和其他模块一样，CLK 也有框架，用以适配不同的平台。适配层之上是客户代码和接口，也就是各模块（如需要时钟信号的外设，USB 等）的驱动。适配层之下是具体的 SOC 台的时钟操作细节。

### ● 时钟树结构

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。

### ● 相关器件

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。

## 1.3 时钟方案

每一个 SOC 都有自己的时钟分配方案，主要是包括 PLL 的设置，各个 CLK 的父属性、DIV、MUX 等。芯片不同，时钟方案是有差异的。

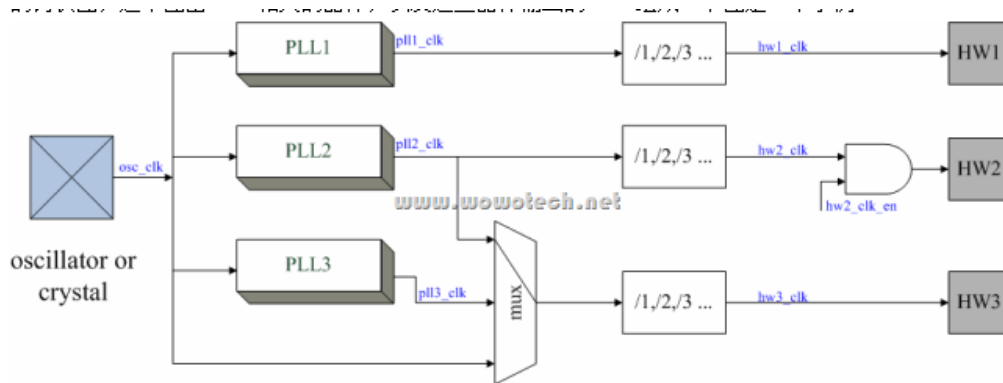


图 1-1 CLK 时钟树的示例图

TP	最高频率与特殊频率	配置方案				
		APLL、DPLL、HDMI PHY 分别专供 CPU、DDR 以及 LCD 使用; CPPLL 专供 GMAC; GPPLL 专供 WiFi;				
		ARM PLL	DDR PLL	HDMI PHY	CODEC PLL	GENERAL PLL
		850MHz	1333MHz/1600MHz	594MHz	500MHz	600M
A7	800	850MHz ARM PLL 1分频				
WiFi	37.5					16分频
DDR	400		DDR PLL 4分频			
LCD (HDMI)	74.25			74.25MHz NPLL 8分频	备份	备份
	148.5			148.5MHz NPLL 4分频	备份	备份
	297			297MHz NPLL 2分频	备份	备份
	594			594MHz NPLL 1分频	备份	备份
I2S	24.576				24.576MHz CODEC PLL 小数分频	备份
	22.5792				22.5792MHz CODEC PLL 小数分频	备份
	12.288				12.288MHz CODEC PLL 小数分频	备份

图 1-2 时钟分配示例图

1.4 总体流程

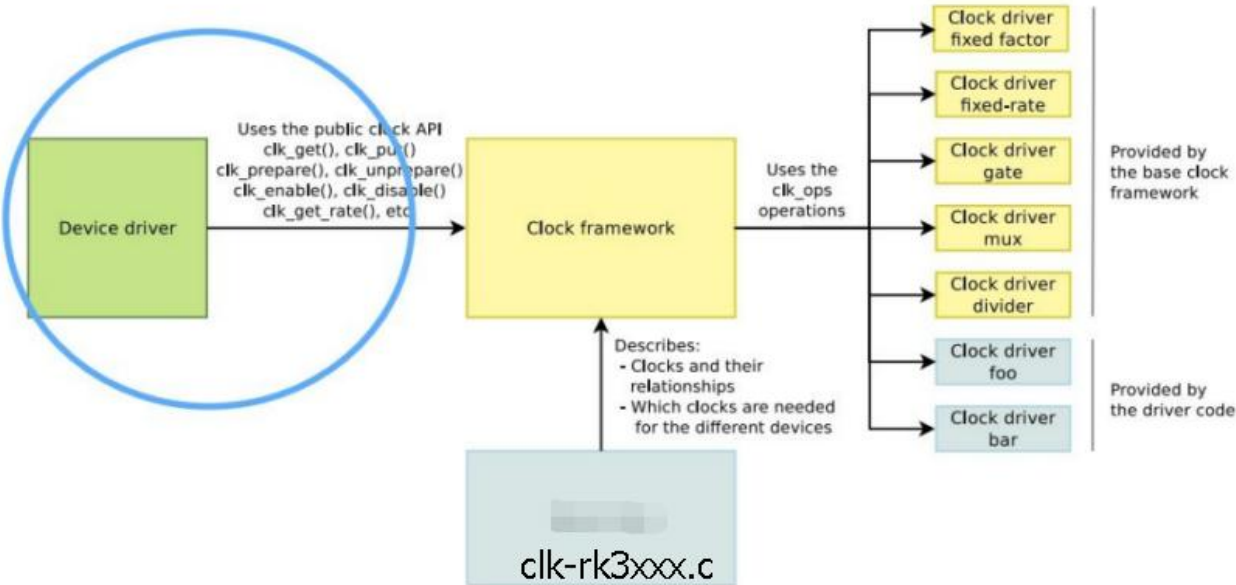


图 1-3 时钟配置流程图

- 主要包括（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。

1.5 代码结构

CLOCK 的软件框架由 CLK 的 clk-rk3xxx.c（clk 的寄存器描述、clk 之间的树状关系等）、Device driver 的 CLK 配置和 CLK API 三部分构成。这三部分的功能、CLK 代码路径如表 1-1 所示。

表 1-1 CLK 代码构成

项目	功能	路径
Clk-rk3xxx.c Rk3xxx-cru.h	.c 中主要是 clk 的寄存器描述、clk 之间的树状关系描述等。 .h 中是 clk 的 ID 定义，通过 ID 匹配 clk name。	Drivers/clk/rockchip/clk-rk3xxx.c Include/dt-bindings/clock/rk3xxx-cru.h
RK 特别的处理	1、处理 RK 的 PLL 时钟 2、处理 RK 的 CPU 时钟等	Drivers/clk/rockchip/clk-xxx.c
CLK API	提供 linux 环境下供 driver 调用的接口	Drivers/clk/clk-xxx.x



# 2 CLOCK 开发指南

## 2.1 概述

本章描述如何修改时钟配置、使用 API 接口及调试 CLK 程序。

## 2.2 时钟的相关概念

### 2.2.1 PLL

锁相环，是由 24M 的晶振输入，然后内部锁相环锁出相应的频率。这个是 SOC 所有 CLOCK 的时钟的源。SOC 的所有总线及设备的时钟都是从 PLL 分频下来的。RK 平台主要 PLL 有：

表 2-1 CLK PLL 描述

PLL	子设备	用途	备注
APLLL	ARMCLKL	CPU 小核的时钟	一般只给 CPU 使用，因为 CPU 会变频，PLL 会根据 CPU 要求的频率变化
APLLB	ARMCLKB	CPU 大核的时钟	一般只给 CPU 使用，因为 CPU 会变频，PLL 会根据 CPU 要求的频率变化
DPLL	Clk_DDR	DDR 的时钟	一般只给 DDR 使用，因为 DDR 会变频，DPLL 会根据 DDR 要求变化
GPLL		提供总线、外设时钟做备份	一般设置在 594M 或者 1200M，保证基本的 100、200、300、400M 的时钟都有输出
CPLL		其他设备做备份	一般可能是 400、500、800、1000M。
NPLL		GMAC 或者给其他设备做备份	一般可能是 1000M。
VPLL		HDMI、VOP 使用	频率根据显示分辨率定，支持任意频率对 jitter 有要求。
PPLL		PMUCRU 时钟，给 PMU 模块提供时钟	一般可能是 676M，可以出 26M。

### 2.2.2 ACLK、PCLK、HCLK

ACLK 是设备的总线的 CLK，PCLK 跟 HCLK 一般是用于寄存器读写的。而像 CLK\_GPU 是 GPU 的控制器的时钟。

我们 SOC 的总线有 ACLK\_PERI、HCLK\_PERI、PCLK\_PERI、ACLK\_BUS、HCLK\_BUS、PCLK\_BUS.各个设备的总线时钟会挂在上面这些时钟下面，如下图结构：

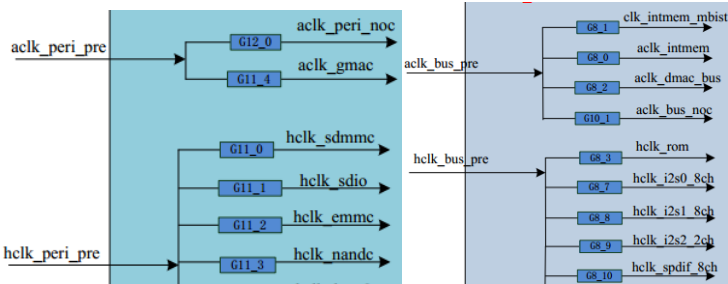


图 2-1 总线时钟结构

（如：EMMC 想提高自己设备的总线频率以实现其快速的数据拷贝或者搬移，可以提高 ACLK\_PERI 来实现）

RK3399 上设计将高速和低速总线彻底分开，分成高速：ACLK\_PERIHP、HCLK\_PERIHP、PCLK\_PERIHP；低速：ACLK\_PERILP0、HCLK\_PERILP0、PCLK\_PERILP0、HCLK\_PERILP1、PCLK\_PERILP1。这样做是为了功耗最优，根据不同的需求可以设置不同的总线频率。（具体每个设备在哪条总线下详细见时钟图）

可以参考如下（EMMC、GMAC、USB 等有自己的 ACLK）：

pd_perilp	cm0, crypto, dcf, imem, dmac, bootrom, efuse_con, spi, i2c, uart, saradc, tsadc
pd_perihp	pcie, usb2, hsic

2.2.3 GATING

Clock 的框架中有很多的 Gating，这个主要是为了降低功耗使用，在一些设备关闭，Clock 不需要维持的时候就可以关闭 Gating，来节省功耗。

我们 Clock 的框架的 Gating 是按照树的结构，有父子属性。Gating 的开关是有一个引用计数机制的，使用这个计数来实现 Clock 打开时，会遍历打开其父 Clock。在子 Clock 关闭时，父 Clock 会遍历所有的子 Clock，在所有的子都关闭的时候才会关闭父 Clock。

（如：I<sup>2</sup>S2 在使用的时候，必须要打开下面这三个 Gating，但是软件上只需要开最后一级的 Gating，我们的时钟结构会自动的打开其 Parent 的 Gating）

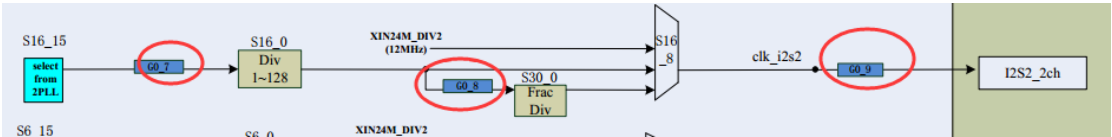


图 2-2 GATING 示例图

2.3 时钟配置

2.3.1 时钟初始化配置

与 LINUX3.10 不同的，4.4 内核时钟的初始化使用 of\_clk\_set\_defaults 然后解析 assigned-clocks 获取 CLK 的 ID，然后获取 clk name，解析 assigned-clock-parents 获取需要设置的 parent，解析 assigned-clock-rates 获取需要设置的频率。

Rk3xxx.dtsi 中：

```
cru: clock-controller@ff760000 {
    compatible = "rockchip,rk3399-cru";
    reg = <0x0 0xff760000 0x0 0x1000>;
    #clock-cells = <1>;
    #reset-cells = <1>;
    assigned-clocks =
        <&cru ACLK_VOP0>, <&cru HCLK_VOP0>,
        <&cru ACLK_VOP1>, <&cru HCLK_VOP1>,
        <&cru ARMCLKL>, <&cru ARMCLKB>,
        <&cru PLL_GPLL>, <&cru PLL_CPLL>,
        <&cru PLL_NPLL>,
        <&cru ACLK_PERIHP>, <&cru HCLK_PERIHP>,
        <&cru PCLK_PERIHP>,
        <&cru ACLK_PERILP0>, <&cru HCLK_PERILP0>,
        <&cru PCLK_PERILP0>,
```

```

        <&cru HCLK_PERILP1>, <&cru PCLK_PERILP1>;
    };}

```

### 1. 频率

CLOCK TREE 初始化时设置的频率:

```

    assigned-clock-rates =
        <400000000>, <200000000>,
        <400000000>, <200000000>,
        <816000000>, <816000000>,
        <594000000>, <800000000>,
        <1000000000>,
        <150000000>, <75000000>,
        <37500000>,
        <100000000>, <100000000>,
        <50000000>,
        <100000000>, <50000000>;
};

```

### 2. Parent

CLOCK TREE 初始化时设置的 parent:

```

    assigned-clock-parents =
        <&cru VPLL>, <&cru VPLL>,
        <&cru CPLL>, <&cru CPLL>,
        <&cru APLLL>, <&cru APLLB>,
        <&cru GPLL>, <&cru GPLL>,
        <&cru GPLL>,
        <&cru GPLL>, <&cru GPLL>;

```

注意:

Assinged 的配置 Parent 和 Rate 时候, 需要跟 Assigned-Clocks 一一对应, 因为设置是按照 Assigned-Clocks 的 CLK ID 进行查找并设置的。

### 3. Gating

CLOCK TREE 初始化时是否默认 enable:

- (1) 需要在 clk-rk3xxx.c 中增加 critical 配置, 主要在 rk3399\_cru\_critical\_clocks 中增加需要默认打开的 CLK name, 一旦增加 CLK 的计数被加 1, 后面这个 CLK 将不能被关闭。

```

static const char *const rk3399_cru_critical_clocks[] __initconst = {
    "aclk_usb3_noc",
    "aclk_gmac_noc",
    "pclk_gmac_noc",
    "pclk_center_main_noc",
    "aclk_cci_noc0",
    "aclk_cci_noc1",
    "clk_dbg_noc",
    "hclk_vcodec_noc",
    "aclk_vcodec_noc",
    "hclk_vdu_noc",
    "aclk_vdu_noc",
};

```

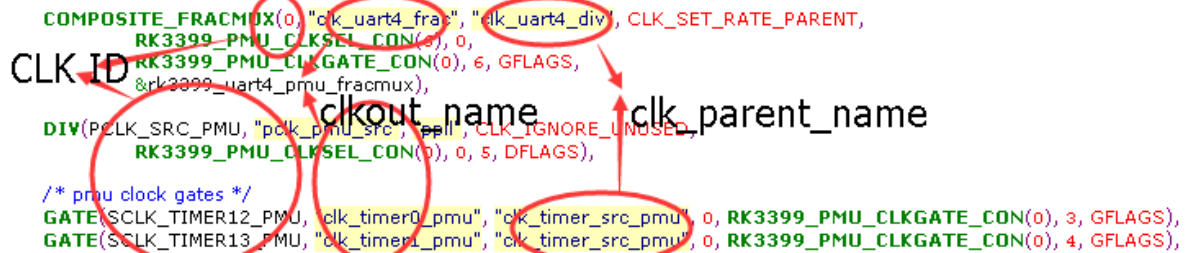
(2) CLK 的定义时候增加 flag 属性 CLK\_IGNORE\_UNUSED, 这样即使这个 CLK 没有使用, 在最后 CLK 关闭没有用的 CLK 时也不会关闭这个。但是在 CLK TREE 上看到的 enable cnt 还是 0, 但是 CLK 是开启的。

```
GATE(PCLK_PMUGRF_PMU, "pclk_pmugrf_pmu", "pclk_pmu_src",
CLK_IGNORE_UNUSED, RK3399_PMU_CLKGATE_CON(1), 1, GFLAGS),
```

### 2.3.2 时钟 ID

LINUX4.4 上对 CLK 的操作都是引用 CLK ID, 而 ID 如何获取?

在 clk-rk3xxx.c 中找到需要控制的 clk(可以用过 name 查找):



```
COMPOSITE_FRACMUX(0, "clk_uart4_frac", "clk_uart4_div", CLK_SET_RATE_PARENT,
RK3399_PMU_CLKSEL_CON(5), 0,
RK3399_PMU_CLKGATE_CON(0), 6, GFLAGS,
&rk3399_uart4_pmu_fracmux),
CLK ID
clkout_name
clk_parent_name
DIV(PCLK_SRC_PMU, "pclk_pmu_src", "pclk_pmu_src", CLK_IGNORE_UNUSED,
RK3399_PMU_CLKSEL_CON(0), 0, 5, DFLAGS),
/* pmu clock gates */
GATE(SCLK_TIMER12_PMU, "clk_timer0_pmu", "clk_timer_src_pmu", 0, RK3399_PMU_CLKGATE_CON(0), 3, GFLAGS),
GATE(SCLK_TIMER13_PMU, "clk_timer1_pmu", "clk_timer_src_pmu", 0, RK3399_PMU_CLKGATE_CON(0), 4, GFLAGS),
```

有一些是没有 ID 的, 因为暂时不需要引用并控制的时钟就没有增加 ID。

### 2.3.3 主要的 CLK 注册类型函数

常用的有如下几种:

**GATE:** 描述 GATING, 主要包括 CLK ID、类型、GATING 的寄存器偏移地址、BIT 位等。

**MUX:** 描述 SLECT, 主要包括 CLK ID、类型、MUX 的寄存器偏移地址、BIT 位等。

**COMPOSITE:** 描述有 MUX、DIV、GATING 的 CLK, 主要包括 CLK ID、类型、MUX、DIV、GARING 的寄存器偏移地址、BIT 位等。

```
#define MUX(_id, cname, pnames, f, o, s, w, mf) \
{ \
    .id = _id, \
    .branch_type = branch_mux, \
    .name = cname, \
    .parent_names = pnames, \
    .num_parents = ARRAY_SIZE(pnames), \
    .flags = f, \
    .muxdiv_offset = o, \
    .mux_shift = s, \
    .mux_width = w, \
    .mux_flags = mf, \
    .gate_offset = -1, \
} \

#define DIV(_id, cname, pname, f, o, s, w, df) \
{ \
    .id = _id, \
    .branch_type = branch_divider, \
    .name = cname, \
    .parent_names = (const char *[]){ pname }, \
    .num_parents = 1, \
    .flags = f, \
    .muxdiv_offset = o, \
    .div_shift = s, \
    .div_width = w, \
    .div_flags = df, \
    .gate_offset = -1, \
} \

#define COMPOSITE(_id, cname, pnames, f, mo, ms, mw, mf, ds, dw, \
df, go, gs, gf) \
{ \
    .id = _id, \
    .branch_type = branch_composite, \
    .name = cname, \
    .parent_names = pnames, \
    .num_parents = ARRAY_SIZE(pnames), \
    .flags = f, \
    .muxdiv_offset = mo, \
    .mux_shift = ms, \
    .mux_width = mw, \
    .mux_flags = mf, \
    .div_shift = ds, \
    .div_width = dw, \
    .div_flags = df, \
    .gate_offset = go, \
    .gate_shift = gs, \
    .gate_flags = gf, \
} \
? end COMPOSITE ?
```

Clk-rk3xxx.c 中的使用, 使用这些 CLK 的注册函数, 描述此 CLK 的类型, 寄存器及父子关系等。

```

3:
4: /* usbphy */
5: GATE(SCLK_USB2PHY0_REF, "clk_usb2phy0_ref", "xin24m", CLK_IGNORE_UNUSED,
6:      RK3399_CLKGATE_CON(6), 5, GFLAGS),
7: GATE(SCLK_USB2PHY1_REF, "clk_usb2phy1_ref", "xin24m", CLK_IGNORE_UNUSED,
8:      RK3399_CLKGATE_CON(6), 6, GFLAGS),
9:
10: GATE(0, "clk_usbphy0_480m_src", "clk_usbphy0_480m", CLK_IGNORE_UNUSED,
11:      RK3399_CLKGATE_CON(13), 12, GFLAGS),
12: GATE(0, "clk_usbphy1_480m_src", "clk_usbphy1_480m", CLK_IGNORE_UNUSED,
13:      RK3399_CLKGATE_CON(13), 12, GFLAGS),
14: MUX(0, "clk_usbphy_480m", mux_usbphy_480m_p, CLK_IGNORE_UNUSED,
15:      RK3399_CLKSEL_CON(14), 6, 1, MFLAGS),
16:
17: MUX(0, "upll", mux_pll_src_24m_usbphy480m_p, 0,
18:      RK3399_CLKSEL_CON(14), 15, 1, MFLAGS),
19:
20: COMPOSITE_NODIV(SCLK_HSICPHY, "clk_hsicphy", mux_pll_src_cppll_gppll_nppll_usbphy480m_p, 0,
21:                 RK3399_CLKSEL_CON(19), 0, 2, MFLAGS,
22:                 RK3399_CLKGATE_CON(6), 4, GFLAGS),
23:
24: COMPOSITE(ACLK_USB3, "aclk_usb3", mux_pll_src_cppll_gppll_nppll_p, 0,
25:           RK3399_CLKSEL_CON(39), 6, 2, MFLAGS, 0, 5, DFLAGS,
26:           RK3399_CLKGATE_CON(12), 0, GFLAGS),

```

## 2.3.4 Driver 的时钟配置

### 1. 获取 CLK 指针

- (1) DTS 设备结点里添加 clock 引用信息（推荐）

DTS:

```
clocks = <&cru SCLK_TSADC>, <&cru PCLK_TSADC>;
```

```
clock-names = "tsadc", "apb_pclk";
```

（CLK 引用的是 ID，通过 ID 找到 CLK NAME）

Driver code:

```
dev->pclk = devm_clk_get(&pdev->dev, "tsadc");
```

```
dev->clk = devm_clk_get(&pdev->dev, "apb_pclk");
```

## 2.4 CLOCK API 接口

### 2.4.1 主要的 CLK API

#### 1. 头文件: #include <linux/clock.h>

clk\_prepare/ clk\_unprepare

clk\_enable/ clk\_disable

clk\_prepare\_enable / clk\_disable\_unprepare

clk\_get/ clk\_put

devm\_clk\_get/ devm\_clk\_put

clk\_get\_rate / clk\_set\_rate

clk\_round\_rate

#### 2. 获取 CLK 指针

```
struct clk *devm_clk_get(struct device *dev, const char *id)（推荐）
```

```
struct clk *clk_get(struct device *dev, const char *id)
```

#### 3. 准备/使能 CLK

```
int clk_prepare(struct clk *clk)
```

/\*开时钟前调用，可能会造成休眠，所以把休眠部分放到这里，可以原子操作的放到 enable 里\*/

```
void clk_unprepare(struct clk *clk)
/*prepare 的反操作*/
int clk_enable(struct clk *clk)
/*原子操作，打开时钟，这个函数必须在产生实际可用的时钟信号后才能返回*/
void clk_disable(struct clk *clk)
/*原子操作，关闭时钟*/
```

- (1) clk\_enable/clk\_disable, 启动/停止 clock。不会睡眠。
- (2) 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。
- (3) 可以使用 clk\_prepare\_enable / clk\_disable\_unprepare, clk\_prepare\_enable / clk\_disable\_unprepare(或者 clk\_enable / clk\_disable) 必须成对，以使引用计数正确。

#### 注意:

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，千万不能在 **atomic** 上下文(例如内部包含 **mutex** 锁、中断关闭、**spinlock** 锁保护的区域)调用，而调用 **enable/disable** 接口则可放心。

另外，clock 的操作为什么需要睡眠呢？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/unprepare**，为了简单，**framework** 就帮忙封装了这两个接口。

#### 4. 设置 CLK 频率

```
int clk_set_rate(struct clk *clk, unsigned long rate) (单位 Hz)
```

### 2.4.2 示例

#### DTS

```
tsadc: tsadc@ff260000 {
    compatible = "rockchip,rk3399-tsadc";
    reg = <0x0 0xff260000 0x0 0x100>;
    interrupts = <GIC_SPI 97 IRQ_TYPE_LEVEL_HIGH>;
    rockchip,grf = <&grf>;
    clocks = <&cru SCLK_TSADC>, <&cru PCLK_TSADC>;
    clock-names = "tsadc", "apb_pclk";
    assigned-clocks = <&cru SCLK_TSADC>;
    assigned-clock-rates = <750000>;
    resets = <&cru SRST_TSADC>;
    reset-names = "tsadc-apb";
    pinctrl-names = "init", "default", "sleep";
    pinctrl-0 = <&otp_gpio>;
    pinctrl-1 = <&otp_out>;
    pinctrl-2 = <&otp_gpio>;
```

```

        #thermal-sensor-cells = <1>;
        rockchip,hw-tshut-temp = <95000>;
        status = "disabled";
    };

Driver code
static int rockchip_thermal_probe(struct platform_device *pdev)
{
    ...
    thermal->clk = devm_clk_get(&pdev->dev, "tsadc");
    if (IS_ERR(thermal->clk)) {
        error = PTR_ERR(thermal->clk);
        dev_err(&pdev->dev, "failed to get tsadc clock: %d\n", error);
        return error;
    }
    thermal->pclk = devm_clk_get(&pdev->dev, "apb_pclk");
    if (IS_ERR(thermal->pclk)) {
        error = PTR_ERR(thermal->pclk);
        dev_err(&pdev->dev, "failed to get apb_pclk clock: %d\n",
            error);
        return error;
    }
    error = clk_prepare_enable(thermal->clk);
    if (error) {
        dev_err(&pdev->dev, "failed to enable converter clock: %d\n",
            error);
        return error;
    }
    error = clk_prepare_enable(thermal->pclk);
    if (error) {
        dev_err(&pdev->dev, "failed to enable pclk: %d\n", error);
        goto err_disable_clk;
    }
}

static int rockchip_thermal_remove(struct platform_device *pdev)
{
    clk_disable(thermal->pclk);
    clk_disable(thermal->clk);
}

```

## 2.5 CLOCK 调试

### CLOCK DEBUGS:

打印当前时钟树结构:

cat d/clock/clock\_summary

### CLOCK 设置节点:

配置选项:

勾选 RK\_PM\_TESTS

```

.config - Linux/arm64 4.4.11 Kernel Configuration
> Platform selection

There is no help available for this option.
Symbol: RK_PM_TESTS [=y]
Type : boolean
Prompt: /sys/pm_tests/ support
Location:
  -> Platform selection
  -> Rockchip Platforms (ARCH_ROCKCHIP [=y])
Defined at arch/arm64/mach-rockchip/kconfig:3
Depends on: ARCH_ROCKCHIP [=y]

```

节点命令:

get rate:

```
echo get [clk_name] > /sys/pm_tests/clk_rate
```

set rate:

```
echo set [clk_name] [rate(Hz)] > /sys/pm_tests/clk_rate
echo rawset [clk_name] [rate(Hz)] > /sys/pm_tests/clk_rate
```

open rate:

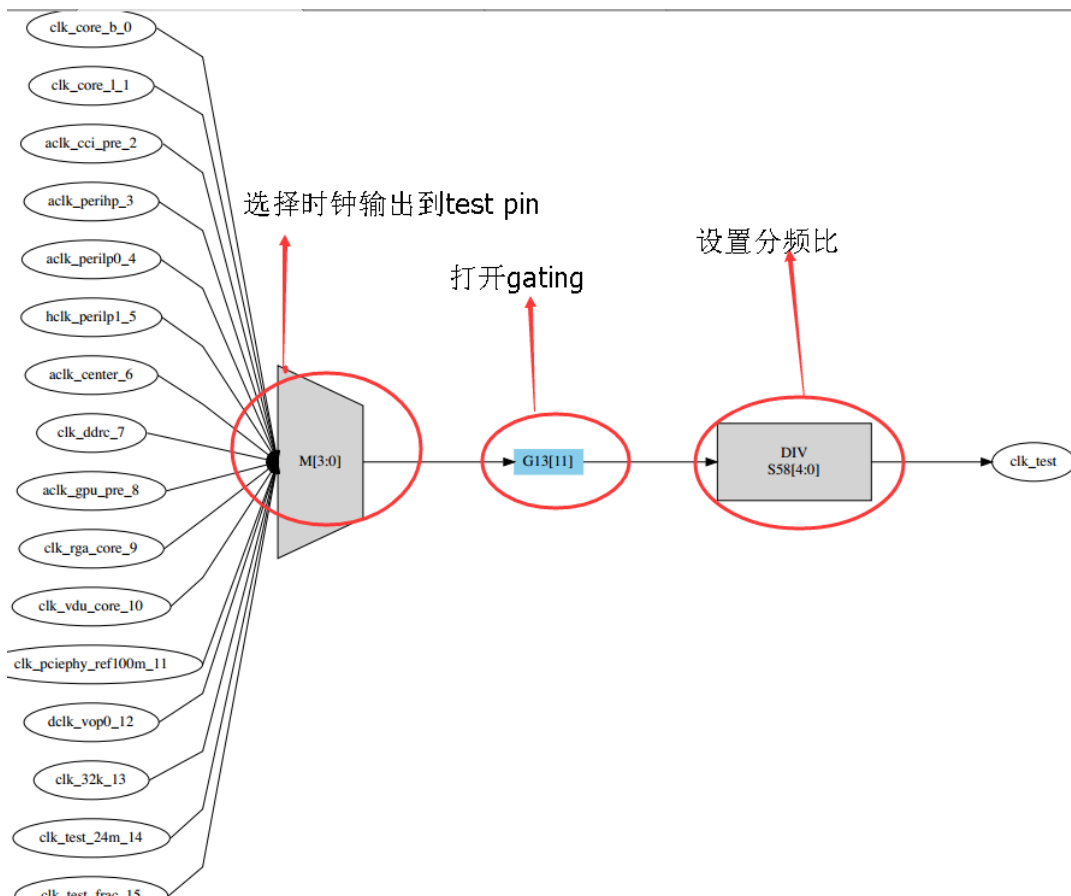
```
echo open [clk_name] > /sys/pm_tests/clk_rate
```

close rate:

```
echo close [clk_name] > /sys/pm_tests/clk_rate
```

### TEST\_CLK\_OUT 测试:

部分时钟是可以输出到 test\_clk\_out，直接测试 clk 输出频率，用于确认某些时钟波形是否正常。配置方法(以 RK3399 为例):



(1) 设置 CLK 的 MUX

CRU\_MISC\_CON

Address: Operational Base + offset (0x050c)



			testclk_sel
			4'h9: clk_rga_core_2wrap
			4'ha: clk_vdu_core_2wrap
			4'hb: clk_pciephy_ref100m
			4'hc: dclk_vop0_2wrap
			4'hd: clk_rtc
			4'he: clkout_24m
			4'hf: clk_wifi
			4'h0: clk_core_b_2wrap
			4'h1: clk_core_l_2wrap
			4'h2: aclk_cci_2wrap
			4'h3: aclk_perihp_2wrap
			4'h4: aclk_perilp0_2wrap
			4'h5: hclk_perilp1_2wrap
			4'h6: aclk_center_2wrap
			4'h7: clk_ddrc_2wrap
			4'h8: aclk_gpu_2wrap
			4'h9: clk_rga_core_2wrap
			4'ha: clk_vdu_core_2wrap
			4'hb: clk_pciephy_ref100m
			4'hc: dclk_vop0_2wrap
			4'hd: clk_rtc
			4'he: clkout_24m
			4'hf: clk_wifi

(2) 设置 CLK 的 DIV

CRU\_CLKSEL58\_CON

Address: Operational Base + offset (0x01e8)

4:0	RW	0x1f	clk_test_div_con test divider control register clk=clk_src/(div_con+1)
-----	----	------	------------------------------------------------------------------------------

(3) 设置 CLK 的 GATING

CRU\_CLKGATE13\_CON

Address: Operational Base + offset (0x0334)