Rockchip Clock 开发指南

发布版本：1.0

作者邮箱：zhangqing@rock-chips.com

日期：2018.6

文件密级：公开资料

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

芯片名称	内核版本
RK303X	LINUX3.10
RK312X	LINUX3.10
RK322X	LINUX3.10
RK3288X	LINUX3.10
RK3328	LINUX3.10
RK3368	LINUX3.10

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
2016.6.6	1.0.	Elaine	第一次临时版本发布
2017.2.10	1.1.	Elaine	Add soc RK3328

图表目录

---

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

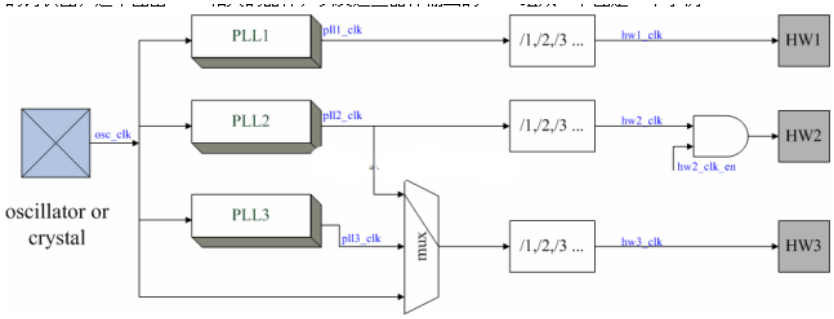
可运行Linux的主流处理器平台，都有非常复杂的clock tree，我们随便拿一个处理器的spec，查看clock相关的章节，一定会有一个非常庞大和复杂的树状图，这个图由clock相关的器件，以及这些器件输出的clock组成。

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。

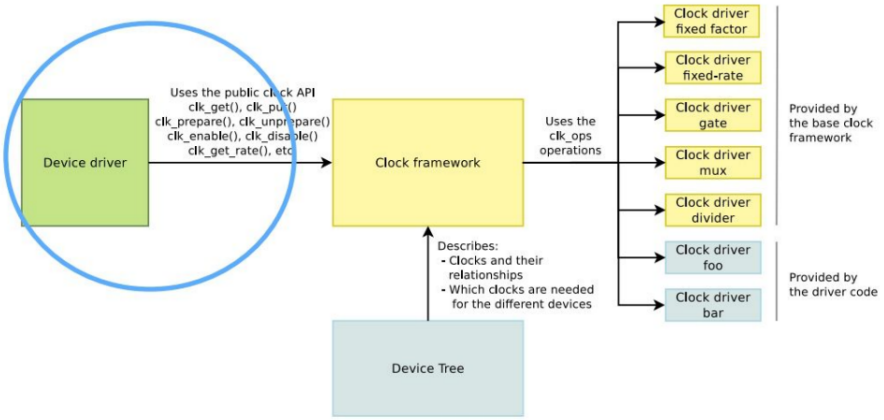


图表 1 - 1 clk 时钟树的示例图

TP	Max freq and	Configuration scheme: APLL、DPLL、HDMIPHY is just for CPU、DDR and LCDC DCLK; CPLLis just for GMAC; GPLLis for wifi and other clks;					
		850MHz	1332MHz/1600MHz	594MHz	500MHz	600M	480MHz
A7	800	850MHz ARM PLL div = 1					
WiFi	37.5					div = 16	40MHz
DDR	400		DDR PLL div = 4				
LCDC(HDMI)	74.25			74.25MHz NPLL div = 8	alternative	alternative	
	148.5			148.5MHz NPLL div = 4	alternative	alternative	
	297			297MHz NPLL div = 2	alternative	alternative	
	594			594MHz NPLL div = 1	alternative	alternative	
I2S	24.576				24.576MHz CODEC PLL fractional divider	alternative	
	22.5792				22.5792MHz CODEC PLL fractional divider	alternative	
	12.288				12.288MHz CODEC PLL fractional divider	alternative	
	8.192				8.192MHz CODEC PLL fractional divider	alternative	

图表 1 - 2时钟分配示例图

1.4 总体流程



图表 1 - 3 时钟配置流程图

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。

1.5 代码结构

CLOCK的软件框架由CLK的Device Tree（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。

项目	功能	路径
Device Tree	clk的寄存器描述、clk之间的树状关系描述等	Arch/arm/boot/dts/rk3xxx-clocks.dtsi
RK PLL及特殊CLK的处理	1.处理RK的PLL时钟2. 处理RK的一些特殊时钟，如LCDC、I2S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

表格 1 - 1CLK代码构成

---

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:

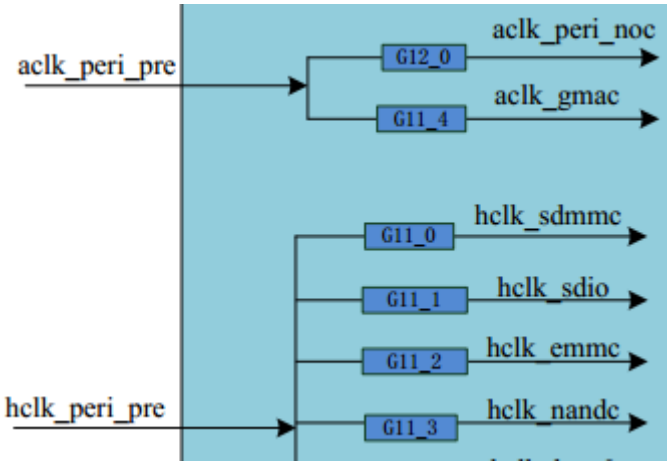
表格 - 1PLL描述

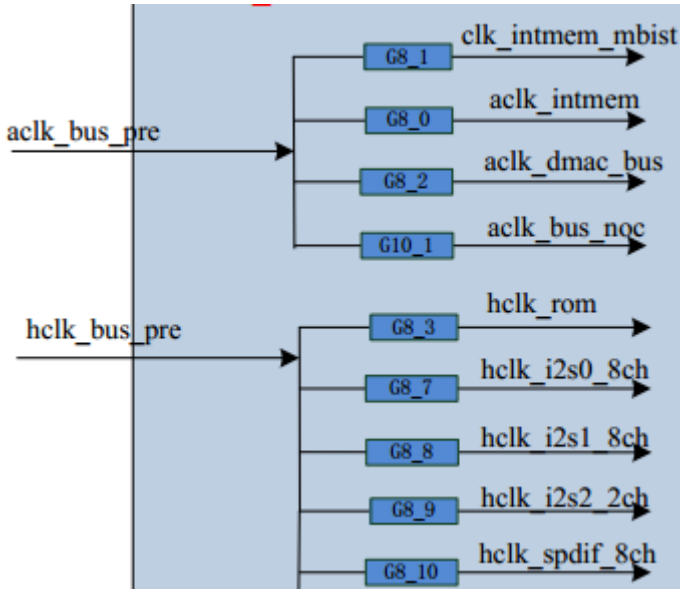
PLL	子设备	用途	备注
APLL	CLK_CORE	CPU的时钟	一般只给CPU使用，因为CPU会变频，APLL会根据CPU要求的频率变化
DPLL	Clk_DDR	DDR的时钟	一般只给DDR使用，因为DDR会变频，DPLL会根据DDR要求变化
GPLL		提供总线、外设时钟做备份	一般设置在594M或者1200M，保证基本的100、200、300、400M的时钟都有输出
CPLL		GMAC或者其他设备做备份	一般可能是400、500、800、1000M。或者是给lcdc独占使用。
NPLL		给其他设备做备份	一般可能是1188M，或者给lcdc独占使用。

我们SOC的总线有ACLK_PERI、HCLK_PERI、PCLK_PERI、ACLK_BUS、HCLK_BUS、PCLK_BUS.（ACLK用于数据传输，PCLK跟HCLK一般是用于寄存器读写）

> *而区分BUS跟PERI主要是为了做高速和低速总线的区分，ACLK范围100-300M，PCLK范围50M~150M，HCLK范围37M~150M。BUS下面主要是一些低速的设备，如I2C、I2S、SPI等，PERI下面一般是EMMC、GMAC、USB等。不同的芯片在设计时会有一些差异。例如：对于某些对总线速度要求较高时，可能单独给此设备设计一个独立的ACLK（如ACLK_EMMC或者ACLK_USB等）。*

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：





（如：GMAC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）

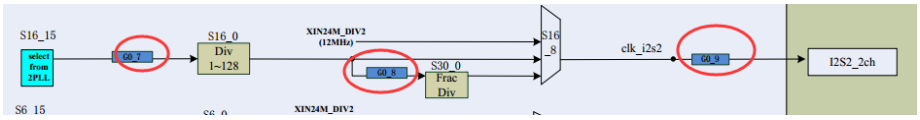
图表 - 1总线时钟结构

2.2.2 GATING

CLOCK的框架中有很多的GATING，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，可以关闭GATING，节省功耗。

RK CLOCK的框架的GATING是按照树的结构，有父子属性。GATING的开关有一个引用计数机制，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：I2S2在使用的时候，必须要打开图中三个GATING（如图2-2），但是软件上只需要开最后一级的GATING，时钟结构会自动的打开其parent的GATING）



图表 - 2GATING示例图

2.3 时钟配置

2.3.1 时钟初始化配置

```
arch/arm64/dts/rockchip/rk33xx.dtsi

    rockchip_clocks_init: clocks-init{
        compatible = "rockchip,clocks-init";
    }
```

1. 频率

CLOCK TREE初始化时设置的频率：

```
    rockchip,clocks-init-rate =
            <&clk_gpll 1200000000>, <&clk_core
700000000>,
            <&clk_cpll 500000000>, <&aclk_bus
150000000>,
            <&hclk_bus 150000000>, <&pclk_bus
75000000>,
            <&aclk_peri 150000000>, <&hclk_peri
150000000>,
            <&pclk_peri 75000000>, <&clk_mac
125000000>,
            <&aclk_iep 250000000>, <&hclk_vio
125000000>,
            <&aclk_rga 250000000>, <&clk_gpu
250000000>,
            <&aclk_vpu 25000000>, <&clk_vdec_core
250000000>,
            <&clk_vdec_cabac 250000000>,
<&aclk_rkvdec 16000000>,
            <&aclk_vop 400000000>, <&clk_gmac_div
25000000>;
```

2. **Parent**

CLOCK TREE初始化时设置的parent：

```
    rockchip,clocks-init-parent =
            <&clk_i2s0_pll &clk_gpll>, <&clk_i2s1_pll
&clk_gpll>,
            <&clk_i2s2_pll &clk_gpll>, <&clk_spdif_pll
&clk_gpll>,
            <&clk_gpu &clk_cpll>, <&dclk_vop0
&hdmi_phy_clk>,
            <&aclk_bus &clk_gpll>, <&aclk_peri
&clk_gpll>,
            <&clk_sdmmc0 &clk_cpll>, <&clk_emmc
&clk_cpll>,
            <&clk_sdio &clk_cpll>, <&aclk_vpu
&clk_cpll>,
            <&hdmi_phy_clk &hdmiphy_out>, <&usb480m
&usb480m_phy>,
            <&aclk_rkvdec &clk_cpll>, <&clk_gmac
&clk_cpll>;
```

3. **Gating**

CLOCK TREE初始化时是否默认enable：

注意：对于没有默认初始化enable，且设备没有引用去enable的时钟，在clk初始化完成之后，会被关闭。

```
    rockchip_clocks_enable: clocks-enable {
        compatible = "rockchip,clocks-enable";
        clocks =
            /*PLL*/
            <&clk_apll>,
            <&clk_dpll>,
            <&clk_gpll>,
            <&clk_cpll>,

            /*PD_CORE*/
            <&clk_core>,
            <&pclk_dbg>,
            <&aclk_core>,
            <&clk_gates4 2>,

            /*PD_BUS*/
            <&clk_gates6 3>,/*pclk_bus_pre*/
            <&clk_gates7 1>,/*clk4x_ddrphy*/
            <&clk_gates8 5>,/*clk_ddrupctl*/
            <&clk_gates8 0>,/*aclk_intmem*/
            <&clk_gates8 1>,/*clk_intmem_mbist*/
            <&clk_gates8 2>,/*aclk_dmac_bus*/
}
```

2.3.2 Driver的时钟配置

1. 获取CLK指针

DTS设备结点里添加clock引用信息（推荐）

```
    clocks = <&clk_saradc>, <&clk_gates7 14>;
    clock-names = "saradc", "pclk_saradc";
```

```
    dev->pclk = devm_clk_get(&pdev->dev,
"pclk_saradc");
    dev->clk = devm_clk_get(&pdev->dev, "saradc");
```

DTS设备结点里未添加clock引用信息

```
Driver code:

    dev->pclk = devm_clk_get(NULL, "g_p_saradc");
    dev->clk = devm_clk_get(NULL, "clk_saradc");
```

2.4 CLOCK API接口

2.4.1 主要的CLK API

1. 头文件

```
#include <linux/clk.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)（推荐使用，可以自动释放）
    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*/
    clk_enable/clk_disable，启动/停止clock。不会睡眠。
    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。
```

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enable/disable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. 设置CLK频率

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

（返回值小于0，设置CLK失败）

2.4.2 示例

DTS

```
    adc: adc@2006c000 {
        compatible = "rockchip,saradc";
        reg = <0x2006c000 0x100>;
        interrupts = <GIC_SPI 26 IRQ_TYPE_LEVEL_HIGH>;
        #io-channel-cells = <1>;
        io-channel-ranges;
        rockchip,adc-vref = <1800>;
        clock-frequency = <1000000>;
        clocks = <&clk_saradc>, <&clk_gates7 14>;
        clock-names = "saradc", "pclk_saradc";
        status = "disabled";
    };
```

Driver code

```
    static int rk_adc_probe(struct platform_device *pdev)
    {
        info->clk = devm_clk_get(&pdev->dev, "saradc");
        if (IS_ERR(info->clk)) {
            dev_err(&pdev->dev, "failed to get adc clock\n");
            ret = PTR_ERR(info->clk);
            goto err_pclk;
        }
```

```
        if(of_property_read_u32(np, "clock-frequency", &rate)) {
            dev_err(&pdev->dev, "Missing clock-frequency property in the DT.\n");
            goto err_pclk;
        }
        ret = clk_set_rate(info->clk, rate);
        if(ret < 0) {
            dev_err(&pdev->dev, "failed to set adc clk\n");
            goto err_pclk;
        }
        clk_prepare_enable(info->clk);
    }

    static int rk_adc_remove(struct platform_device *pdev)
    {
        struct iio_dev *indio_dev = platform_get_drvdata(pdev);
        struct rk_adc *info = iio_priv(indio_dev);
        device_for_each_child(&pdev->dev, NULL, rk_adc_remove_devices);
        clk_disable_unprepare(info->clk);
        clk_disable_unprepare(info->pclk);
        iio_device_unregister(indio_dev);
        free_irq(info->irq, info);
        iio_device_free(indio_dev);
        return 0;
    }
```

2.5 CLOCK调试

1. **CLOCK DEBUGS:**

打印当前时钟树结构：

```
    cat /sys/kernel/debug/clk/clk_summary
```

2. **CLOCK** 设置节点：

配置选项：

勾选RK_PM_TESTS

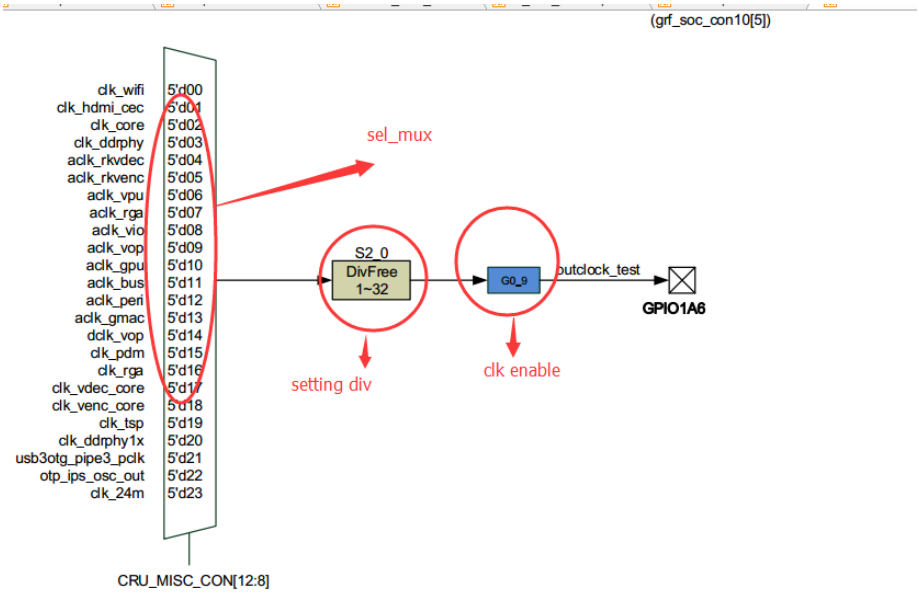
```
    There is no help available for this option.
    Symbol: RK_PM_TESTS [=y]
    Type  : boolean
    Prompt: /sys/pm_tests/ support
    Location:
     -> System Type
     -> Rockchip SoCs (ARCH_ROCKCHIP [=y])
    Defined at arch/arm/mach-
rockchip/Kconfig.common:41
    Depends on: ARCH_ROCKCHIP [=y]
    Selects: DVFS [=y] && WATCHDOG [=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
```

3. 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RK3228为例)：



设置CLK的MUX

CRU_MISC_CON

Address: Operational Base + offset (0x0134)

11:8	RW	0x0	testclk_sel Output clock selection for test 3'b000: outclock_test_t = buf_clk_wifi 3'b001: outclock_test_t = buf_clk_hdmi_cec 3'b010: outclock_test_t = buf_clk_core_2wrap_pre 3'b011: outclock_test_t = buf_clk_ddrphy 3'b100: outclock_test_t = buf_aclk_iep_2wrap_pre 3'b101: outclock_test_t = buf_aclk_gpu_2wrap_pre 3'b110: outclock_test_t = buf_aclk_peri_2wrap_pre 3'b111: outclock_test_t = buf_aclk_cpu_2wrap_pre default: outclock_test_t = buf_clk_wifi

设置CLK的DIV

CRU_CLKSEL4_CON

Address: Operational Base + offset (0x0054)

4:0	RW	0x03	clk_monitor_div_con Control clk_monitor divider freuency clk=clk_src/(div_con+1)

设置CLK的GATING

CRU_CLKGATE0_CON

Address: Operational Base + offset (0x00d0)

15	RW	0x0	testclk_gate_en test output clock disable When HIGH, disable clock

---

3 常见问题分析

3.1 PLL设置

3.1.1 PLL类型查找

不同芯片PLL相关的寄存器、PLL计算公式等会有一些差异，使用PLL类型来区分芯片不同，去计算并设置PLL的参数。

在rk3xxx-clocks.dtsi中，找到PLL，确认其类型。

```
    clk_cpll: pll-clk@0018 {
                        compatible = "rockchip,rk3188-pll-
clk";
                        clock-output-names = "clk_cpll";
                        rockchip,pll-type =
<CLK_PLL_312XPLUS>;
};
```

根据PLL的类型，在clk-pll.c中查找PLL频率支持的表格：

```
    case CLK_PLL_312XPLUS:

    return &clk_pll_ops_312xplus;
```

3.1.2 PLL回调函数的定义

```
    static const struct clk_ops clk_pll_ops_312xplus =
{
    .recalc_rate = clk_pll_recalc_rate_3036_apll,
    .round_rate = clk_cpll_round_rate_312xplus,
    .set_rate = clk_cpll_set_rate_312xplus,
    };
```

3.1.3 PLL频率表格定义

```
    struct pll_clk_set *clk_set = (struct pll_clk_set
*)(rk312xplus_pll_com_table);
    static const struct pll_clk_set
rk312xplus_pll_com_table[] = {
        _RK3036_PLL_SET_CLKS(1000000, 3, 125, 1,  1, 1,
0),
        _RK3036_PLL_SET_CLKS(800000, 1, 100, 3, 1, 1,
0),
        _RK3036_PLL_SET_CLKS(594000, 2, 99, 2, 1, 1,
0),
        _RK3036_PLL_SET_CLKS(500000, 1, 125, 3, 2, 1,
0),
        _RK3036_PLL_SET_CLKS(416000, 1, 104, 3, 2, 1,
0),
        _RK3036_PLL_SET_CLKS(400000, 3, 200, 2, 2, 1,
0),
    };
```

3.1.4 PLL计算公式

```
    VCO = 24M * FBDIV / REFDIV (450M ~ 2200M)
    /*VCO越大jitter越小，功耗越大; REFDIV越小PLL LOCK时间越短*/
    FOUT = VCO / POSTDIV1/ POSTDIV2 /
    /* POSTDIV1 > = POSTDIV2*/
```

如： VCO = 24M * 99 / 2 = 1188M

FOUT = 1188 / 2/ 1 = 594M

如果需要增加其他的PLL频率，按照上述公式补齐表格即可。

有一个PLL类型是特殊的，查表查不到，会自动去计算PLL的参数。如：

```
    CLK_PLL_3036PLUS_AUTO
    CLK_PLL_312XPLUS
    CLK_PLL_3188PLUS_AUTO
```

（注意：但是使用自动计算的时候，VCO不能保证尽量大，如果对PLL的jitter有要求的不建议使用。）

3.2 部分特殊时钟的设置

3.2.1 LCDC显示相关的时钟

LCDC的DCLK是根据当前屏幕的分辨率决定的，所以不同产品间会有很大差异。所以RK平台上LCDC的DCLK一般是独占一个PLL的。由于要独占一个PLL，所以这个PLL的频率会根据屏的要求变化。所以一般此PLL要求是可以自动计算PLL参数的。而且一些其他对时钟有要求的时钟尽量不要挂在此PLL下面。如下表中：

表格 - 1

产品名称	PLL
RK303X	独占CPLL
RK312X	独占CPLL
RK322X	独占HDMIPHY PLL
RK3288X	独占CPLL
RK3368	独占NPLL

对于显示的CLOCK的设置，不同的平台差异很大，在此以RK322X和RK3288为例。

RK322X：

使用HDMIPHY PLL给DCLK LCDC，所以就比较简单，DCLK LCDC需要多少，就按照HDMIPHY输出多少的时钟就可以了，这个是HDMIPHY内部实现PLL的锁相输出。

RK3288：

RK3288的就比较麻烦了，虽然也是CPLL独占使用，但是CPLL下面还有其他的时钟，而且3288是支持双显，也就是有DCLK_LCDC0和DCLK_LCDC1，一个做主显一个做HDMI显示。主显跟HDMI显示都跟实际屏的分辨率有关系，所以理论上需要两个独立的PLL的，但是3288设计上只有一个PLL给显示用，那么我们就只能要求主显的是可以修改CPLL的频率，满足任意分辨率的屏，而另一个lcdc只能是在当前GPLL和CPLL已有的频率下分频出就近的频率。

```
drivers/clk/rockchip/clk-ops.c

    const struct clk_ops clkops_rate_3288_dclk_lcdc0 = {
    .determine_rate = clk_3288_dclk_lcdc0_determine_rate,
    .set_rate = clk_3288_dclk_lcdc0_set_rate,
    .round_rate = clk_3288_dclk_lcdc0_round_rate,
    .recalc_rate = clk_divider_recalc_rate,
    };

    const struct clk_ops clkops_rate_3288_dclk_lcdc1 = {
    .determine_rate = clk_3288_dclk_lcdc1_determine_rate,
    .set_rate = clk_3288_dclk_lcdc1_set_rate,
    .round_rate = clk_3288_dclk_lcdc1_round_rate,
    .recalc_rate = clk_divider_recalc_rate,
    };
```

3.2.2 EMMC、SDIO、SDMMC

这几个时钟有要求必须是偶数分频得到的，而且控制器内部还有默认的二分频。也就是说如果EMMC需要50M，那边CLOCK要给EMMC提供100M的时钟。并且100M是由PLL偶数分频得到的。

偶数分频的时钟有一个标志：rockchip,clkops-idx = <CLKOPS_RATE_MUX_EVENDIV>;

如果修改此类时钟的频率需要如下步骤：

1. 确认此时钟的**parent**。

```
arch/arm/dts/rk3xxx-clocks.dtsi

    clk_emmc: clk_emmc_mux {
            compatible = "rockchip,rk3188-mux-con";
            rockchip,bits = <14 2>;
            clocks = <&cpll>, <&clk_gpll>, <&xin24m>;
            clock-output-names = "clk_emmc";
            #clock-cells = <0>;
    };
    /*有CPLL、GPLL、24M三个parent可以选择*/
```

2. 确认其**parent**的频率

```
cat /sys/kernel/debug/clk/clk_summary | grep gpll
```

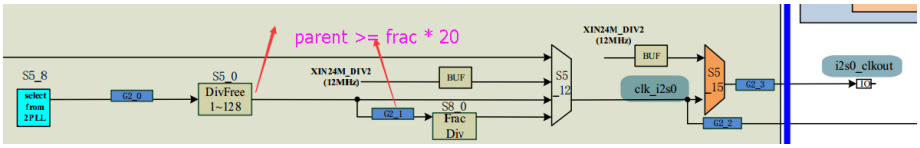
按照上面说的二倍的关系，及偶数分频确认是否可以分出需求的频率。如果不能分出，是否可以将PLL的频率倍频上去（但是一般不建议超过1200M）。

3. 设置频率

EMMC的驱动中可以通过clk_set_rate接口去修改频率。

3.2.3 小数分频

I2S、UART等有小数分频的。对于小数分频设置时有一个要求，就是小数分频的频率跟小数分频的parent有一个20倍的关系，如果不满足20倍关系，输出的CLK会有较大的抖动及频偏。



图表 3 - 1 小数分频时钟示意图

3.2.4 以太网时钟

对于以太网的时钟，一般要求是精准的，百兆以太网要求50M精准的频率，千兆以太网要求125M精准的频率。一般有以太网需求的，PLL也要是精准的时钟输出。如果说当前的时钟方案由于其他的原因不能出精准的时钟，那么以太网就要使用外部时钟晶振。这个是根据项目需求及实际的产品方案定的。