全志科技
Allwinner Technology

# Tina

## clk 模块使用说明 v1.0

# 文档履历

| 版本号 | 日期 | 制/修订人 | 制/修订记录 |
|---|---|---|---|
| V1.0 | 2017/10/25 | | 初始版本 |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# 目　录



# 1. 概述

## 1.1. 编写目的

介绍 Linux 内核中时钟管理接口及使用方法，为时钟操作开发提供参考。

## 1.2. 适用范围

适用于 R6/R16/R18/R40/F35/R30 硬件平台。

## 1.3. 相关人员

公司开发人员、客户。

# 2. 模块介绍

时钟管理模块是 linux 系统为统一管理各硬件的时钟而实现管理框架，负责所有模块的时钟调节和电源管理。

## 2.1. 模块功能介绍

时钟管理模块主要负责处理各硬件模块的工作频率调节及电源切换管理。一个硬件模块要正常工作，必须先配置好硬件的工作频率、打开电源开关、总线访问开关等操作，时钟管理模块为设备驱动提供统一的操作接口，使驱动不用关心时钟硬件实现的具体细节。

## 2.2. 相关术语介绍

| 术语 | 介绍 |
| --- | --- |
| 晶振 | 晶体振荡器的简称，晶振有固定的振荡频率，如 32K/24Mhz 等，是芯片所有时钟的源头。 |
| PLL | 锁相环，利用输入信号和反馈信号差异提升频率输出 |
| 时钟 | 驱动数字电路运转是的时钟信号。芯片内部的各硬件模块都需要时序控制，因此理解时钟信号对于底层编程非常重要。 |

## 2.3. 模块配置介绍

ccmu 初始化时，会设置一些系统时钟的频率，如 PLL4、PLL6 等等，在 sys_config1.fex 中，增加了相关的频率配置，如：

```
[clock]
pll4_freq     = 960
pll6_freq     = 720
```

对于没有配置的，系统设置频率为默认值

# 3. 接口描述

Linux 系统为时钟管理定义了标准的 API 接口，详见内核接口头文件《include/linux/clk.h》。

## 3.1. 系统时钟定义

系统时钟主要是指一些源时钟，为其它硬件模块提供时钟源输入。系统时钟一般为多个硬件模块共享，不允许随意调节。时钟源可以如下定义：

```
#define PLL1_CLK "pll1"
#define PLL2_CLK "pll2"
#define PLL3_CLK "pll3"
#define PLL4_CLK "pll4"
#define PLL5_CLK "pll5"
#define PLL6_CLK "pll6"
#define PLL7_CLK "pll7"
#define PLL8_CLK "pll8"
#define PLL9_CLK "pll9"
```

```
#define PLL10_CLK "pll10"
#define CPU_CLK "cpu"
#define AXI_CLK "axi"
#define AHB1_CLK "ahb1"
#define APB1_CLK "apb1"
#define APB2_CLK "apb2"
```

各 PLL 的分工如下

| PLLx | 分工 |
|---|---|
| PLL1 | PLL1 只作为 CPU 的时钟源，不作他用； |
| PLL2 | 只作为音频模块（如 codec、iis、spdif 等）的时钟源，不作他用； |
| PLL3、PLL7 | 一般作为显示相关模块（如 de、csi、hdmi 等）的时钟源； |
| PLL4 | 一般只作为视频解码模块（ve）的时钟源； |
| PLL5 | 一般只作为 DDR 的时钟源； |
| PLL6 | 用作一些外设接口模块（如 nand、sdmmc、usb 等）的时钟源； |
| PLL8 | 一般只作为 GPU 模块的时钟源； |
| PLL9/PLL10 | 是两个通用时钟源，可以为多个模块共享； |

## 3.2. 模块时钟定义

模块时钟主要是针对一些具体模块（如：gpu、de），在时钟频率配置、电源控制、访问控制等方面进行管理。一个典型的模块如图 3.1 所示，包含 module gateing、ahb gating、dram gating，以及 reset 控制。要想一个模块能够正常工作，必须在这几个方面作好相关的配置。

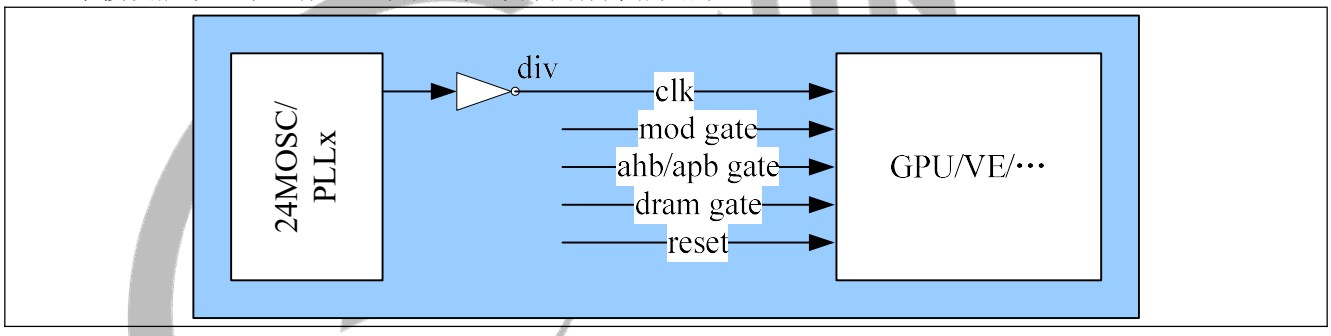


图 3.1 clock 模块

硬件设计时，为每个硬件模块定义好了可选的时钟源（有些默认使用总线的工作时钟作时钟源），时钟源的定义如上节所述，模块只能在相关可能的时钟源间作选择。

模块的电源管理体现在两个方面：模块的时钟使能和模块控制器复位，相关驱动需要通过以下所列的时钟进行控制。如下：

```
#define NAND0_CLK "nand0"
#define NAND1_CLK "nand1"
#define SDMMC0_CLK "sdmmc0"
#define SDMMC1_CLK "sdmmc1"
#define SDMMC2_CLK "sdmmc2"
#define SDMMC3_CLK "sdmmc3"
#define TS_CLK "ts"
#define SS_CLK "ss"
#define SPI0_CLK "spi0"
#define SPI1_CLK "spi1"
#define SPI2_CLK "spi2"
#define SPI3_CLK "spi3"
#define I2S0_CLK "i2s0"
#define I2S1_CLK "i2s1"
```

```
#define SPDIF_CLK "spdif"
#define USBOHCI0_CLK "usbohci0"
#define USBOHCI1_CLK "usbohci1"
#define USBOHCI2_CLK "usbohci2"
#define USBEHCI0_CLK "usbehci0"
#define USBEHCI1_CLK "usbehci1"
#define USBOTG_CLK "usbotg"
#define USBPHY0_CLK "usbphy0"
#define USBPHY1_CLK "usbphy1"
#define USBPHY2_CLK "usbphy2"
#define MDFS_CLK "mdfs"
#define DEBE0_CLK "debe0"
#define DEBE1_CLK "debe1"
#define DEFE0_CLK "defe0"
#define DEFE1_CLK "defe1"
#define MP_CLK "mp"
#define LCD0CH0_CLK "lcd0ch0"
#define LCD0CH1_CLK "lcd0ch1"
#define LCD1CH0_CLK "lcd1ch0"
#define LCD1CH1_CLK "lcd1ch1"
#define CSI0_S_CLK "csi0_s"
#define CSI0_M_CLK "csi0_m"
#define CSI1_S_CLK "csi1_s"
#define CSI1_M_CLK "csi1_m"
#define VE_CLK "ve"
#define ADDA_CLK "adda"
#define AVS_CLK "avs"
#define DMIC_CLK "dmic"
#define HDMI_CLK "hdmi"
#define HDMI_DDC_CLK "hdmi_ddc"
#define PS_CLK "ps"#define MTCACC_CLK "mtcacc"
#define MBUS0_CLK "mbus0"
#define MBUS1_CLK "mbus1"
#define MIPIDSI_CLK "mipidsi"
#define MIPIDPHY_CLK "mipidphy"
#define MIPICSI_CLK "mipicsi"
#define DRC0_CLK "drc0"
#define DRC1_CLK "drc1"
#define DEU0_CLK "deu0"
#define DEU1_CLK "deu1"
#define GPUCORE_CLK "gpucore"
#define GPUMEM_CLK "gpumem"
#define GPUHYD_CLK "gpuhyd"
#define DMA_CLK "dma"
#define GMAC_CLK "gmac"
#define HSTMR_CLK "hstmr"
#define MSBBOX_CLK "msbbox"
#define SPINLOCK_CLK "spinlock"
#define LVDS_CLK "lvds"
#define PIO_CLK "pio"
#define TWI0_CLK "twi0"
#define TWI1_CLK "twi1"
#define TWI2_CLK "twi2"
#define TWI3_CLK "twi3"
#define UART0_CLK "uart0"
#define UART1_CLK "uart1"
#define UART2_CLK "uart2"
```

```
#define UART3_CLK "uart3"
#define UART4_CLK "uart4"
#define UART5_CLK "uart5"
```

## 3.3. 时钟 API 接口定义

使用系统的时钟操作接口，必须引用 Linux 系统提供的时钟接口头文件，引用方式为“#include <linux/clk.h>”

### 3.3.1. clk_get

| 类别 | 介绍 |
|---|---|
| 函数原型 | struct clk *clk_get(struct device *dev, const char *id); |
| 参数 | dev,申请时钟的设备句柄；id,要申请的时钟名； |
| 返回 | 如果申请时钟成功，返回时钟句柄，否则返回 NULL。 |
| 功能描述 | 该函数用于申请指定时钟名的时钟句柄，所有的时钟操作都基于该时钟句柄来实现。 |

DEMO:

```
//打开”nand”的时钟句柄
h_nand = clk_get(NULL, “nand”);
if(!h_nand) {
        printk(“try to get nand clock failed!\n”);
        ……
}
```

### 3.3.2. devm_clk_get

| 类别 | 介绍 |
|---|---|
| 函数原型 | struct clk *devm_clk_get(struct device *dev, const char *id); |
| 参数 | dev,申请时钟的设备句柄；id,要申请的时钟名； |
| 返回 | 如果申请时钟成功，返回时钟句柄，否则返回 NULL。 |
| 功能描述 | 该函数用于申请指定时钟名的时钟句柄，所有的时钟操作都基于该时钟句柄来实现。<br>和 clk_get 的区别在于:一般用在 driver 的 probe 函数里申请时钟句柄,而当 driver probe 失败或者 driver remove 时，devres 会自动释放对应的时钟句柄（即相当于系统自动调用 clk_put） |

DEMO:

```
//打开“sdmmc0”的时钟句柄
struct    clk        *sdmmc_clk
sdmmc_clk = devm_clk_get(&pdev->dev, “hosc”);
if(!h_hosc) {
        printk(“try to get hosc clock failed!\n”);
        ……
}
```

### 3.3.3. clk_put

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | void clk_put(struct clk *clk); |
| 参数 | clk，待释放的时钟句柄； |
| 返回 | 无 |
| 功能描述 | 该函数用于释放成功申请到的时钟句柄，当不再使用时钟时，需要释放时钟句柄。 |

DEMO:

```
///释放 h_hosc 时钟句柄
clk_put(h_nand);“sdmmc0”的时钟句柄
```

### 3.3.4. clk_set_parent

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | int clk_set_parent(struct clk *clk, struct clk *parent); |
| 参数 | clk，待操作的时钟句柄；parent，父时钟的时钟句柄。 |
| 返回 | 如果设置父时钟成功，返回 0；否则，返回-1 |
| 功能描述 | 该函数用于设定指定时钟的父时钟，即将 parent 作为 clk 的时钟源。 |

DEMO:

```
//设置 nand 的父时钟为的 hosc
if(clk_set_parent(h_nand, h_hosc)) {
        printk(“try to set parent of nand to hosc failed!\n”);
        ……
    }
```

### 3.3.5. clk_get_parent

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | struct clk * clk_get_parent(struct clk *clk); |
| 参数 | clk ，待操作的时钟句柄； |
| 返回 | 如果获取父时钟成功，返回父时钟句柄；否则，返回-1。 |
| 功能描述 | 该函数用于获取指定时钟的父时钟。 |

DEMO:

```
//获取 nand 的父时钟
Struct clk* hparent;
hparent = clk_get_parent(h_nand);
if(IS_ERR(hparent)) {
        printk(“try to getparent of nand failed!\n”);
        ……
    }
```

### 3.3.6. clk_prepare

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | int clk_prepare(struct clk *clk); |

| | |
|---|---|
| 参数 | clk　　待操作的时钟句柄； |
| 返回 | 如果时钟 prepare 成功，返回 0；否则，返回-1。 |
| 功能描述 | 该函数用于 prepare 使能指定的时钟<br>(Note:旧版本 kernel 的 clk_enable 在新 kernel 中分解成不可在原子上下文调用的 clk_prepare（该函数可能睡眠）和可以在原子上下文调用的 clk_enable。而 clk_prepare_enable 则同时完成 prepare 和 enable 的工作，只能在可能睡眠的上下文调用该 API) |

DEMO:

```
///prepare nand 时钟
if(clk_prepare(h_nand)) {
        printk(“try to prepare nand failed!\n”);
        ……
    }
```

### 3.3.7. clk_enable

| 类别 | 介绍 |
|---|---|
| 函数原型 | int clk_enable(struct clk *clk); |
| 参数 | clk　　待操作的时钟句柄； |
| 返回 | 如果时钟使能成功，返回 0；否则，返回-1。 |
| 功能描述 | 该函数用于使能指定的时钟。<br>(Note:旧版本 kernel 的 clk_enable 在新 kernel 中分解成不可在原子上下文调用的 clk_prepare(该函数可能睡眠)和可以在原子上下文调用的 clk_enable。因此在 clk_enable 之前至少调用了一次 clk_prepare,也可用 clk_prepare_enable 同时完成 prepare 和 enable 的工作，只能在可能睡眠的上下文调用该 API) |

DEMO:

```
//使能 nand 时钟
if(clk_enable(h_nand)) {
        printk(“try to enable nand failed!\n”);
        ……
    }
```

### 3.3.8. clk_prepare_enable

| 类别 | 介绍 |
|---|---|
| 函数原型 | int clk_prepare_enable(struct clk *clk); |
| 参数 | clk　　待操作的时钟句柄； |
| 返回 | 如果时钟使能成功，返回 0；否则，返回-1。 |
| 功能描述 | 该函数用于 prepare 并使能指定的时钟。<br>(Note:旧版本 kernel 的 clk_enable 在新 kernel 中分解成不可在原子上下文调用的 clk_prepare（该函数可能睡眠）和可以在原子上下文调用的 clk_enable,clk_prepare_enable 同时完成 prepare 和 enable 的工作,只能在可能睡眠的上下文调用该 API) |

DEMO:

```
///使能 nand 时钟
if(clk_prepare_enable(h_nand)) {
        printk(“try to prepare_enable nand failed!\n”);
        ……
    }
```

### 3.3.9. clk_disable

| 类别 | 介绍 |
|---|---|
| 函数原型 | void clk_disable(struct clk *clk); |
| 参数 | clk 待操作的时钟句柄； |
| 返回 | 无 |
| 功能描述 | 该函数用于关闭指定的时钟。 |

DEMO:

```
//关闭 nand 时钟
clk_disable(h_nand);
```

### 3.3.10. clk_unprepare

| 类别 | 介绍 |
|---|---|
| 函数原型 | void clk_unprepare(struct clk *clk); |
| 参数 | clk 待操作的时钟句柄； |
| 返回 | 无 |
| 功能描述 | 该函数用于释放指定的时钟 prepare 动作。<br>(Note:旧版本 kernel 的 clk_disable 在新 kernel 中分解成可以在原子上下文调用的 clk_disable 和不可在原子上下文调用的 clk_unprepare（该函数可能睡眠）和,clk_disable_unprepare 同时完成 disable 和 unprepare 的工作，只能在可能睡眠的上下文调用该 API) |

DEMO:

```
//关闭 nand 时钟
clk_disable(h_nand);
```

### 3.3.11. clk_disable_unprepare

| 类别 | 介绍 |
|---|---|
| 函数原型 | void clk_disable_unprepare(struct clk *clk); |
| 参数 | clk 待操作的时钟句柄； |
| 返回 | 无 |
| 功能描述 | 该函数用于关闭指定的时钟并且释放指定的时钟的 prepare 工作。<br>(Note:旧版本 kernel 的 clk_disable 在新 kernel 中分解成可以在原子上下文调用的 clk_disable 和不可在原子上下文调用的 clk_unprepare（该函数可能睡眠）和,clk_disable_unprepare 同时完成 disable 和 unprepare 的工作，只能在可能睡眠的上下文调用该 API) |

DEMO:

```
//关闭 nand 时钟
clk_disable_unprepare(h_nand);
```

### 3.3.12. clk_get_rate

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | int clk_set_rate(struct clk *clk, unsigned long rate); |
| 参数 | clk 待操作的时钟句柄； |
| 返回 | 指定时钟的当前频率值。 |
| 功能描述 | 该函数用于获取指定时钟当前的频率，无论时钟是否已经使能。 |

DEMO:

```
//获取 hosc 的时钟频率
unsigned long rate;
rate = clk_get_rate(h_hosc);
printk(“rate of hosc is:%ld”, rate);
```

### 3.3.13. clk_set_rate

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | int clk_set_rate(struct clk *clk, unsigned long rate); |
| 参数 | clk 待操作的时钟句柄；rate 时钟的目标频率值，以 Hz 为单位； |
| 返回 | 如果设置时钟频率成功，返回 0；否则，返回-1。 |
| 功能描述 | 该函数用于设置指定时钟的频率。 |

DEMO:

```
//设置 nand 时钟的频率
unsigned long rate;
rate =clk_get_rate(h_hosc);
if(clk_set_rate(h_nand, rate/2)) {
        printk(“set nand clock freq to 1/2 of hosc failed!\n”);
}
```

### 3.3.14. sunxi_periph_reset_assert

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | void sunxi_periph_reset_assert(struct clk *c); |
| 参数 | clk 待 assert 的时钟句柄； |
| 返回 | 设置模块的 assert 状态成功，返回 0；否则，返回-1。 |
| 功能描述 | 该函数用于设置指定时钟的 assert 状态(相当于旧版本的 reset )。 |

### 3.3.15. sunxi_periph_reset_deassert

| 类别 | 介绍 |
| --- | --- |
| 函数原型 | void sunxi_periph_reset_deassert(struct clk *c); |
| 参数 | clk 待 deassert 的时钟句柄； |
| 返回 | 设置 deassert 的复位状态成功，返回 0；否则，返回-1。 |
| 功能描述 | 该函数用于设置指定时钟的 deassert 状态。 |

# 4. Demo

## 4.1. 时钟 API 调用格式要求

（1）一定要判断 clk_get 返回句柄的有效性. 比如:
规范写法

```
struct clk *pll = clk_get(NULL, "sys_pll3");
if(!pll || IS_ERR(pll)){
        /* 获取时钟句柄失败 */
        printk("try to get pll3 failed!\n");
}
...
```

不规范写法

```
struct clk *pll = clk_get(NULL, "sys_pll3");
...(访问 pll 句柄)
```

（2）有返回值的 clk api 一定要判断返回值，返回失败时建议加打印，比如:

规范写法：

```
if(clk_enable(pll)) {
        /* 使能 PLL3 输出失败 */
        printk("try to enable pll3 output failed!\n");
}
```

不规范写法：

```
clk_enable(pll);
```

（3）clk_disable 或 clk_put 之前，先检查句柄的有效性; clk_put 之后将句柄清空．比如:

```
if(NULL == sdc0_clk || IS_ERR(sdc0_clk)) {
        printk("sdc0_clk handle is invalid, just return!\n");
        return;
} else {
        clk_disable(sdc0_clk);
        clk_put(sdc0_clk);
        sdc0_clk = NULL;
}
```

## 4.2. 设置 PLL3 频率

```
struct clk *pll = clk_get(NULL, "sys_pll3");
if(!pll || IS_ERR(pll)){
        /* 获取时钟句柄失败 */
            printk("try to get pll3 failed!\n");
}

/* 使能时钟 */
if(clk_prepare_enable(pll)) {
            /* 使能 PLL3 输出失败 */
            printk("try to enable pll3 output failed!\n");
}

/* 设置 PLL3 的频率为 270Mhz */
if(clk_set_rate(pll, 270000000)) {
            /* 设置 PLL3 频率失败 */
            printk("try to set rate to 270000000 failed!\n");
}
```

## 4.3. 配置 SDMMC 控制器时钟

配置 SDMMC 的时钟，使期以 50Mhz 的工作频率正常工作，其示例代码如下：

```
struct clk *pll6 = clk_get(NULL, "pll6");
struct clk *sdc0_clk = clk_get(NULL, "sdmmc0");
```

```
long       rate;

/* 检查 clock 句柄有效性 */
if(NULL == pll6 || IS_ERR(pll6)
            || NULL == sdc0_clk || IS_ERR(sdc0_clk) ){
                printk("%s: clock handle invalid!\n", __func__);
                return;
}

/* 设置 SDMMC0 控制器的父时钟为 PLL6 */
if(clk_set_parent(sdc0_clk, pll6))
                printk("%s: set sdc0_clk parent to pll6 failed!\n", __func__);

            /* 设置 SDMMC0 控制器的频率为 50Mhz */
            rate = clk_round_rate(sdc0_clk, 50000000);
if(clk_set_rate(sdc0_clk, rate))
                printk("%s: set sdc0_clk rate to %dHZ failed!\n", __func__, rate);

/* 使能 SDMMC0 */
if(clk_prepare_enable(sdc0_clk))
                printk("%s: enablesdc0_clk failed!\n", __func__);

/* SDMMC0 控制器可以正常工作 */
            ……
```

系统卸载 SDMMC0 以后，关闭 SDMMC0 的时钟，使其处如非工作状态，其示例代码如下：

```
    if(NULL != sdc0_clk && !IS_ERR(sdc0_clk)) {
                    /* 关闭 SDMMC0 的模块时钟 */
                    clk_disable_unprepare(sdc0_clk);
                    /* 释放 sdc0_clk 句柄 */
                    clk_put(sdc0_clk);
                    sdc0_clk = NULL;
}

if(NULL != pll6 && !IS_ERR(pll6)) {
                    /* 释放 pll6 句柄 */
                    clk_put(pll6);
                    pll6 = NULL;
}
```

# 5. Declaration