H616-clock 接口使用说明书

1.0

2019.12.10

文档履历

版本号	日期	制/修订人	内容描述
1.0	2019.12.10	AW1226	
		AW1440	

目录

1. 概述

1.1 编写目的

本文档对 Sunxi 平台的时钟管理接口使用进行详细的阐述，让用户明确掌握时钟操作的编程方法。

1.2 适用范围

本文档适用于 Linux-4.9 内核。

1.3 相关人员

本文档适用于所有需要开发设备驱动的人员。

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

- 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的差异提升频率输出。
- 时钟：驱动数字电路运转时的时钟信号。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

主要 device tree 配置方式。

```
pll_ve: pll_ve_clk {
#clock-cells = <0>;
compatible = "allwinner,pll-clock";
device_type = "clk_pll_ve";
lock-mode = "new";
assigned-clock-rates = <460000000>;
clock-output-names = "pll_ve";
```

```
};

clk_de: de {
    #clock-cells = <0>;
    compatible = "allwinner,periph-clock";
    assigned-clocks = <&clk_de>;
    assigned-clock-parents = <&clk_pll_de>;
    assigned-clock-rates = <300000000>;
    clock-output-names = "de";
};
```

对于没有配置的，系统设置频率为默认值。

2.4 源码结构介绍

CCU 的源码结构如下图所示：

```
|---drivers
||---clk
|||---sunxi
||||---clk-sun8iw18.h
||||---clk-periph.h
||||---clk-sunxi.h
||||---clk-factors.h
||||---clk-factors.c
||||---clk-periph.c
||||---clk-sun8iw18.c
||||---clk-default.c
||||---clk-debugfs.c
|---include
||---linux
|||---clk.h
|||---clk
```

clk.h：时钟子系统提供的 API 接口定义。

clk-factors.c：针对 PLLx 等系统时钟 (以 HOSC 为 source) 的公用代码。

clk-periph.c：针对各模块时钟的公用代码。

clk-sun8iw18.c：具体平台的驱动实现。

clk-default.c：第二级初始化操作。

clk-debugfs.c：针对 debugfs 的调试接口。

2.5 系统时钟结构

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。

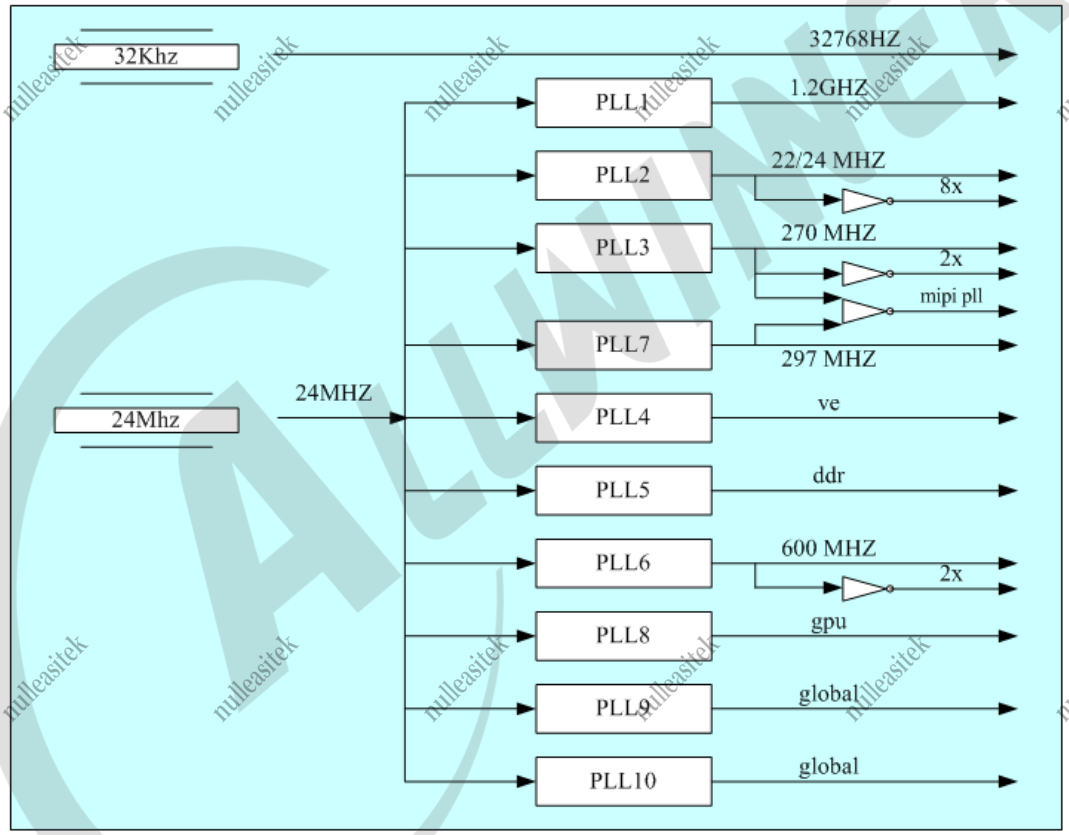

图 1: 系统时钟结构图

系统上一般只有两个时源头：低频晶振（LOSC）32KHz 和高频晶振（HOSC）24MHz，系统在 HOSC 的基础上，增加一些锁相环电路，实现更高的时钟频率输出。为了便于控制一些模块的时钟频率，系统对时钟源进行了分组，实现较多的锁相环电路，以实现分路独立调节。由于 CPU、总线的时钟比较特殊，其工作时钟也经常会输出作为某些其它模块的时钟源，因此，我们也将此类时钟归结为系统时钟。其结构图如下：

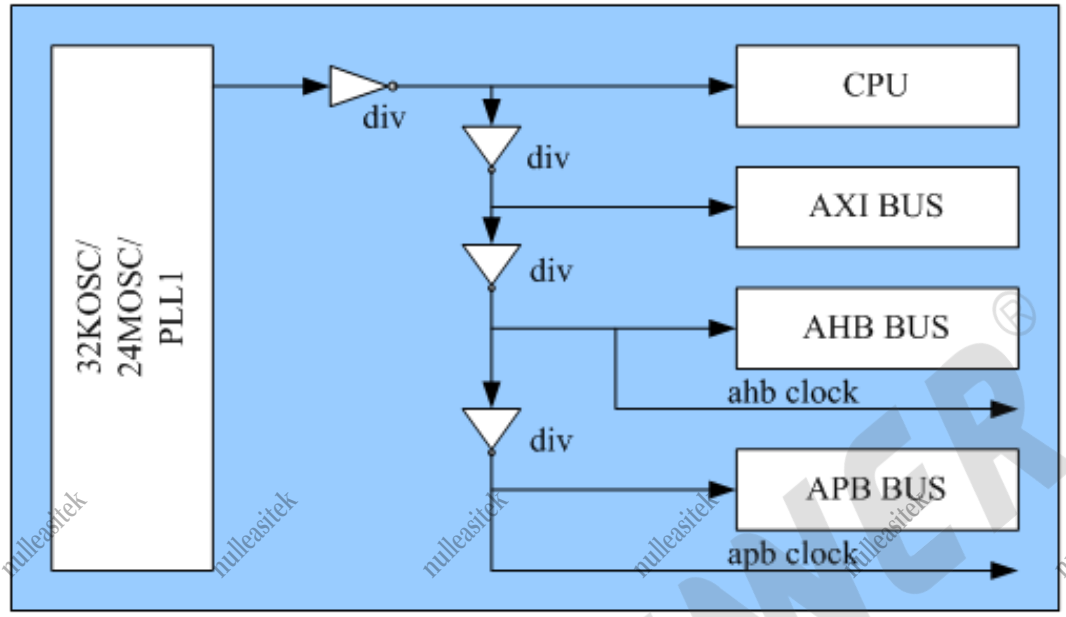

图 2: 总线时钟结构图

以 sun8iw18p1 平台为例，定义系统时钟源的设备树文件为 sun8iw18p1-clk.dtsi，其中 pll 的定义如下：

```
clk_pll_cpu: pll_cpu {
    #clock-cells = <0>;
    compatible = "allwinner,pll-clock";
    lock-mode = "new";
    clock-output-names = "pll_cpu";
};
clk_pll_ddr: pll_ddr {
    #clock-cells = <0>;
    compatible = "allwinner,pll-clock";
    lock-mode ="new";
    clock-output-names = "pll_ddr";
};
clk_pll_periph0: pll_periph0 {
    #clock-cells = <0>;
    compatible = "allwinner,pll-clock";
    assigned-clock-rates = <600000000>;
    lock-mode = "new";
    clock-output-names = "pll_periph0";
};
clk_pll_periph1: pll_periph1 {
    #clock-cells = <0>;
    compatible = "allwinner,pll-clock";
```

```
    assigned-clock-rates = <600000000>;
    lock-mode = "new";
    clock-output-names = "pll_periph1";
  };
  clk_pll_audio: pll_audio {
    #clock-cells = <0>;
    compatible = "allwinner,pll-clock";
    lock-mode = "new";
    assigned-clock-rates = <24576000>;
    clock-output-names = "pll_audio";
  };
  clk_pll_32k: pll_32k {
    #clock-cells = <0>;
    compatible = "allwinner,pll-clock";
    lock-mode = "new";
    assigned-clock-rates = <24576000>;
    clock-output-names = "pll_32k";
  };
  ...
```

各 PLL 的分工如下：

pll_cpu 只作为 CPU 的时钟源，不作他用。

pll_audio 只作为音频模块（如 codec、iis、spdif 等）的时钟源，不作他用。

pll_video0、pll_video1 一般作为显示相关模块（如 de、csi、hdmi 等）的时钟源。

pll_ve 一般只作为视频解码模块（ve）的时钟源。

pll_ddr0、pll_ddr1 一般只作为 DDR 的时钟源。

hosc 用作一些外设接口模块（如 nand、sdmmc、usb 等）的时钟源。

pll_gpu 一般只作为 GPU 模块的时钟源。

pll_periph0、pll_periph1 是两个通用时钟源，可以为多个模块共享。

2.6 模块时钟结构

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。一个典型的模块如下图所示，包含 module gating、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。

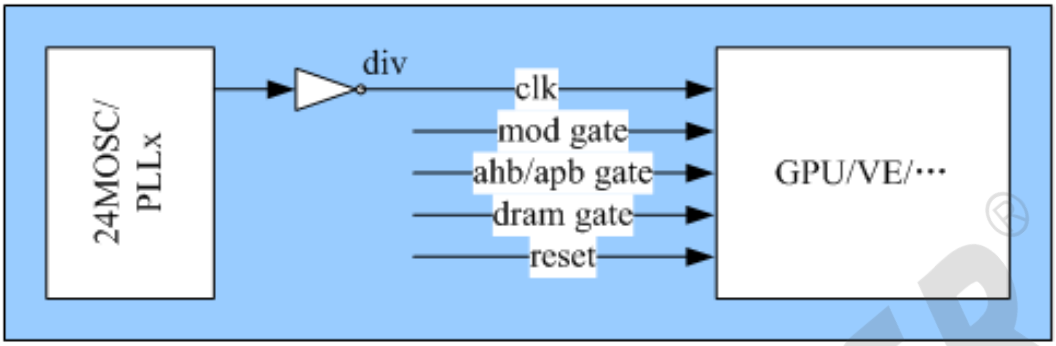

图 3: 模块时钟结构图

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。以 sun8iw18 平台为例, 模块时钟 sun8iw18p1-clk.dtsi 清单如下：

```
clk_cpu: cpu {
    #clock-cells = <0>;
    compatible = "allwinner,periph-clock";
    clock-output-names = "cpu";
};
其他类似的还有：clk_axi、clk_cpuapb、clk_psi、clk_ahb1、clk_ahb2、clk_ahb3、clk_apb1、clk_apb2、
clk_ce、clk_dma、clk_hstimer、clk_avs、clk_dbgsys、clk_pwm、clk_sdram、clk_nand0、clk_nand1、
clk_sdmmc1_mod、clk_sdmmc1_bus、clk_sdmmc1_rst、clk_uart0、clk_uart1、clk_uart2、clk_uart3、
clk_twi0、clk_twi1、clk_spi0、clk_spi1、clk_gpadc、clk_ths、clk_i2s0: i2s0、clk_dmic、clk_mad、
clk_ledc、clk_pio、clk_losc_out、clk_losc_ext ...
```

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 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>
```

Linux 系统为时钟管理定义了一套标准和 API 接口，Sunxi 平台的时钟 API 遵循该 API 规范。

3.2 时钟 API 说明

3.2.1 clk_get

- PROTOTYPE

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

- ARGUMENTS

dev: 申请时钟的设备句柄；

id: 要申请的时钟名；

- RETURNS

如果申请时钟成功，返回时钟句柄，否则返回 NULL。

- DESCRIPTION

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”);
        ……
}
```

3.2.2 devm_clk_get

- PROTOTYPE

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

- ARGUMENTS

dev: 申请时钟的设备句柄；

id: 要申请的时钟名；

- RETURNS

如果申请时钟成功，返回时钟句柄，否则返回 NULL。

- DESCRIPTION

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。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，driver 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”);
        ……
}
```

3.2.3 clk_put

- PROTOTYPE

```
void clk_put(struct clk *clk);
```

- ARGUMENTS

clk: 待释放的时钟句柄;

- RETURNS

无。

- DESCRIPTION

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);
```

3.2.4 of_clk_get(推荐使用）

- PROTOTYPE

```
struct clk *of_clk_get(struct device_node *np, int index)
```

- ARGUMENTS

  np: 设备的 device_node

  index: 在 dts 中的索引值

- RETURNS

  如果申请时钟成功，返回时钟句柄，否则返回 NULL。

- DESCRIPTION

  获取设备的时钟。

- DEMO

```
sw_uport->mclk = of_clk_get(np, 0); //用序号0访问设定的唯一uart1 clk
    if (IS_ERR(sw_uport->mclk)) {
        SERIAL_MSG("uart%d error to get clk\n", pdev->id);
        return -EINVAL;
    }
```

3.2.5 clk_set_parent

- PROTOTYPE

```
int clk_set_parent(struct clk *clk, struct clk *parent)
```

- ARGUMENTS

clk: 待操作的时钟句柄；

parent: 父时钟的时钟句柄；

- RETURNS

如果设置父时钟成功，返回 0；否则，返回 -1。

- DESCRIPTION

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”);
        ……
    }
```

3.2.6 clk_get_parent

- PROTOTYPE

```
struct clk * clk_get_parent(struct clk *clk);
```

- ARGUMENTS

clk: 待操作的时钟句柄；

- RETURNS

如果获取父时钟成功，返回父时钟句柄；否则，返回 -1。

- DESCRIPTION

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”);
        ……
    }
```

3.2.7 clk_get_parent

- PROTOTYPE

```
struct clk * clk_get_parent(struct clk *clk);
```

- ARGUMENTS

  clk: 待操作的时钟句柄；

- RETURNS

  如果获取父时钟成功，返回父时钟句柄；否则，返回 -1。

- DESCRIPTION

  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”);
        ……
    }
```

3.2.8 clk_prepare

- PROTOTYPE

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

- ARGUMENTS

clk: 待操作的时钟句柄；

- RETURNS

如果时钟 prepare 成功，返回 0；否则，返回 -1。

- DESCRIPTION

该函数用于 prepare 指定的时钟 (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”);
        ……
    }
```

3.2.9 clk_enable

- PROTOTYPE

```
int clk_enable(struct clk *clk);
```

- ARGUMENTS

  clk: 待操作的时钟句柄；

- RETURNS

  如果时钟使能成功，返回 0；否则，返回 -1。

- DESCRIPTION 该函数用于使能指定的时钟。(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”);
        ……
    }
```

3.2.10 clk_prepare_enable（推荐使用）

- PROTOTYPE

```
int clk_prepare_enable(struct clk *clk);
```

- ARGUMENTS

clk: 待操作的时钟句柄；

- RETURNS

如果时钟使能成功，返回 0；否则，返回 -1。

- DESCRIPTION

该函数用于 prepare 并使能指定的时钟。(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”);
        ……
    }
```

3.2.11 clk_disable

- PROTOTYPE

```
void clk_disable(struct clk *clk);
```

- ARGUMENTS

  clk 待操作的时钟句柄；

- RETURNS

  无。

- DESCRIPTION

  该函数用于关闭指定的时钟。

- DEMO

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

3.2.12 clk_unprepare

- PROTOTYPE

```
void clk_unprepare(struct clk *clk);
```

- ARGUMENTS

clk: 待操作的时钟句柄；

- RETURNS

无。

- DESCRIPTION

该函数用于释放指定的时钟 prepare 动作。(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);
```

3.2.13 clk_disable_unprepare

- PROTOTYPE

```
void clk_disable_unprepare(struct clk *clk);
```

- ARGUMENTS

clk: 待操作的时钟句柄；

- RETURNS

无。

- DESCRIPTION

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。(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);
```

3.2.14 clk_get_rate

- PROTOTYPE

```
unsigned long clk_get_rate(struct clk *clk);
```

- ARGUMENTS

clk: 待操作的时钟句柄；

- RETURNS

指定时钟的当前频率值。

- DESCRIPTION

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);
```

3.2.15 clk_set_rate

- PROTOTYPE

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

- ARGUMENTS

clk: 待操作的时钟句柄；

rate: 时钟的目标频率值，以 Hz 为单位；

- RETURNS

如果设置时钟频率成功，返回 0；否则，返回 -1。

- DESCRIPTION

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”);
}
```

3.2.16 sunxi_periph_reset_assert

- PROTOTYPE

```
void sunxi_periph_reset_assert(struct clk *c);
```

- ARGUMENTS

  c: 待 assert 的时钟句柄；

- RETURNS

  设置模块的 assert 状态成功，返回 0；否则，返回 -1。

- DESCRIPTION

  该函数用于设置指定时钟的 assert 状态 (相当于旧版本的 reset)。

- DEMO

3.2.17 sunxi_periph_reset_deassert

- PROTOTYPE

```
void sunxi_periph_reset_deassert(struct clk *c);
```

- ARGUMENTS

  c: 待 deassert 的时钟句柄；

- RETURNS

  设置 deassert 的复位状态成功，返回 0；否则，返回 -1。

- DESCRIPTION

  该函数用于设置指定时钟的 deassert 状态。

- DEMO

4. Demo

以 spi 模块的时钟处理部分作为 demo 分析：

```
static int sunxi_spi_clk_init(struct sunxi_spi *sspi, u32 mod_clk)
{
    int ret = 0;
    long rate = 0;

    /* 获取index = 0的clk句柄 */
    sspi->pclk = of_clk_get(sspi->pdev->dev.of_node, 0);
    /* 对clk句柄的有效性进行判断 */
    if (IS_ERR_OR_NULL(sspi->pclk)) {
        SPI_ERR("[spi-%d] Unable to acquire module clock '%s', return %x\n",
            sspi->master->bus_num, sspi->dev_name, PTR_RET(sspi->pclk));
        return -1;
    }
    /* 获取index = 1的clk句柄并判断有效性 */
    sspi->mclk = of_clk_get(sspi->pdev->dev.of_node, 1);
    if (IS_ERR_OR_NULL(sspi->mclk)) {
        SPI_ERR("[spi-%d] Unable to acquire module clock '%s', return %x\n",
            sspi->master->bus_num, sspi->dev_name, PTR_RET(sspi->mclk));
        return -1;
    }
    /* 设置clk的父时钟并判断有效性 */
    ret = clk_set_parent(sspi->mclk, sspi->pclk);
    if (ret != 0) {
        SPI_ERR("[spi-%d] clk_set_parent() failed! return %d\n",
            sspi->master->bus_num, ret);
        return -1;
    }

    rate = clk_round_rate(sspi->mclk, mod_clk);
    /* 设置clk的频率并判断有效性 */
    if (clk_set_rate(sspi->mclk, rate)) {
        SPI_ERR("[spi-%d] spi clk_set_rate failed\n", sspi->master->bus_num);
        return -1;
    }

    SPI_INF("[spi-%d] mclk %u\n", sspi->master->bus_num, (unsigned)clk_get_rate(sspi->mclk));
    /* 使能clk并判断有效性 */
    if (clk_prepare_enable(sspi->mclk)) {
        SPI_ERR("[spi-%d] Couldn't enable module clock 'spi'\n", sspi->master->bus_num);
        return -EBUSY;
    }
    /* 获取clk的频率值 */
    return clk_get_rate(sspi->mclk);
}
```

5. Declaration

This document is the original work and copyrighted property of Allwinner Technology (“Allwinner”). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner. The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This document neither states nor implies warranty of any kind, including fitness for any particular application.tates nor implies warranty of any kind, including fitness for any particular application.