版本号:

发布日期: 2025-04-15

版本历史

版本号	日期	制/修订人	内容描述

目　录

title: Linux CCU subtitle: 开发指南 author: Allwinner changelog:

- ver: 1.0 date: 2022.08.04 author: XAA0249 desc: | 初始版本
- ver: 1.1 date: 2024.1.26 author: XAA0249 desc: | 增加框图、流程等详细介绍
- ver: 1.2 date: 2024.10.30 author: XAA0312 desc: | 刷新文档格式
- ver: 1.3 date: 2025.1.3 author: XAA0329 desc: | 刷新板级设备树路径
- ver: 1.4 date: 2025.2.21 author: XAA0338 desc: | 1. 更新 menuconfig 配置路径; 2. 更新源码结构; 3. 更新模块接口相关描述; 4. 更新模块使用范例; 5. 增加 linux6.6 内核 clk tree 新特性描述;
- ver: 1.5 date: 2025.3.26 author: XAA0329 desc: | 修改设备树配置说明

1 前言

1.1 文档简介

本文档对 Sunxi 平台的操作与配置进行详细的阐述，让用户明确掌握时钟管理接口及其使用方法

1.2 目标读者

本文档适用于所有需要开发驱动设备的人员。

1.3 适用范围

适用于使用 bsp 独立仓库的所有内核及版本，linux-5.4 及以上。

1.4 相关术语介绍

1.4.1 硬件术语

表 1-1: CCU 模块相关术语介绍

术语	解释说明
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，因此理解时钟信号很重要。

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的一套管理框架，负责所有模块的时钟调节。

2.1 模块功能与结构

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。

此外，由于 sunxi 的 clk 驱动中同时集成了 reset 相关驱动，因此，在文章中会包含 reset 子系统的使用说明。

说明

本节中出现的图片仅为示例 demo，不同型号会有区别，具体信息请以 user manual 为准

2.1.1 系统时钟结构

系统时钟主要是指一些为其他硬件模块提供时钟源输入的时钟源，为其它硬件模块提供时钟源输入。系统时钟一般由多个硬件模块共享，不允许随意调节。如下图所示，该平台有 12 路 PLL 作为系统时钟为其它模块提供时钟源，图中是 HOSC，也可以直接作为系统时钟源给给其他硬件模块提供时钟。

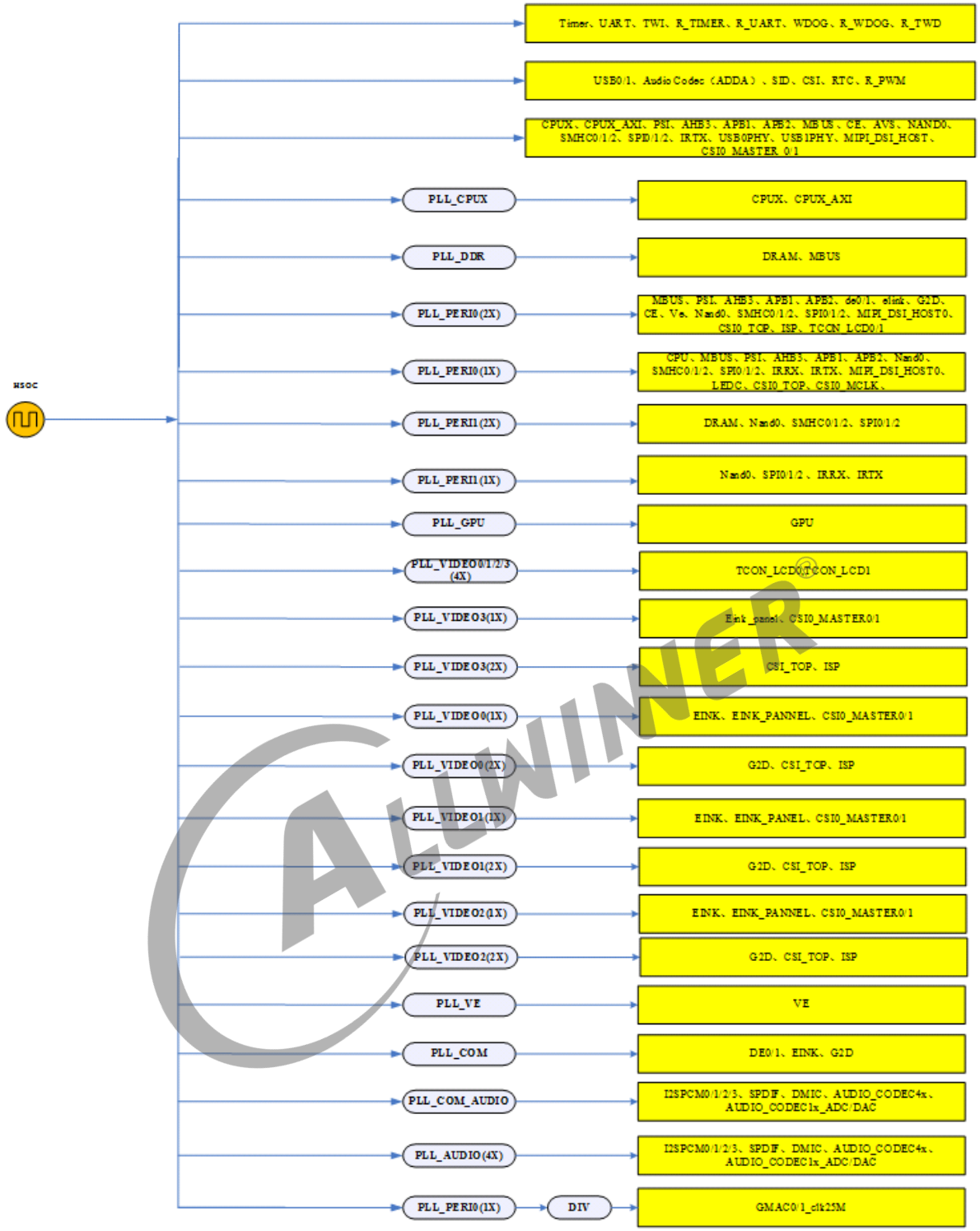

图 2-1: pic0

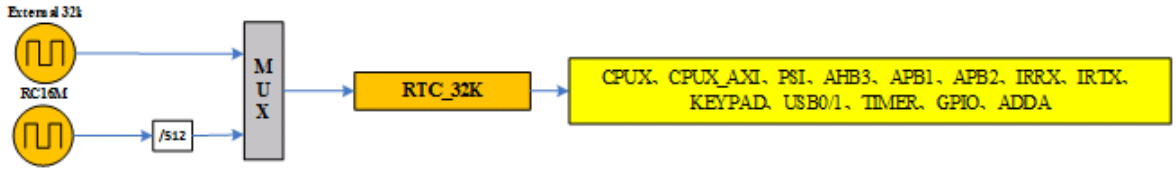

图 2-2: pic6

系统时钟来源：32k(losc)、16M(iosc)、24M(hosc)，32k 时钟也可以从 16M 分频得来。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、BUS 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：

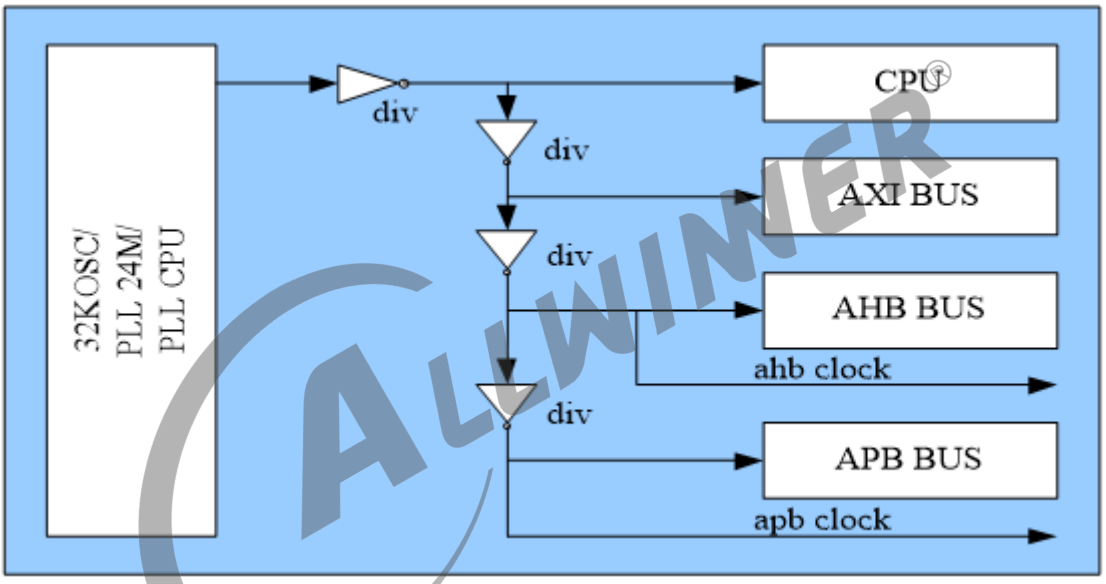

图 2-3: pic1

2.1.2 模块时钟结构

模块时钟主要是针对一些具体模块（如：pio、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、bus 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。

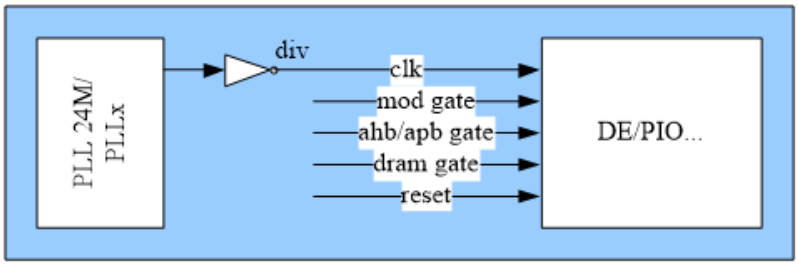

图 2-4: 模块时钟

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。

2.1.3 System bus 结构

系统总线是连接不同模块的桥梁，整个系统中存在不同的总线，不同的总线挂载了不同的硬件IP。需要注意的是，对于某根总线以及其下面挂载的 IP，它们的时钟源没有必然联系，即总线时钟源与相应的 IP 时钟源并不一定为同一个。

对于 bus gate 控制功能，我们一般也将其抽象为一个时钟，这个时候，我们会人为的将某些模块时钟挂载在其总线时钟下，这样在通过 debug 打印时钟树时，可以较为方便的查看其总线结构，对于部分时钟来说，还能够拿到当前模块的真实频率。

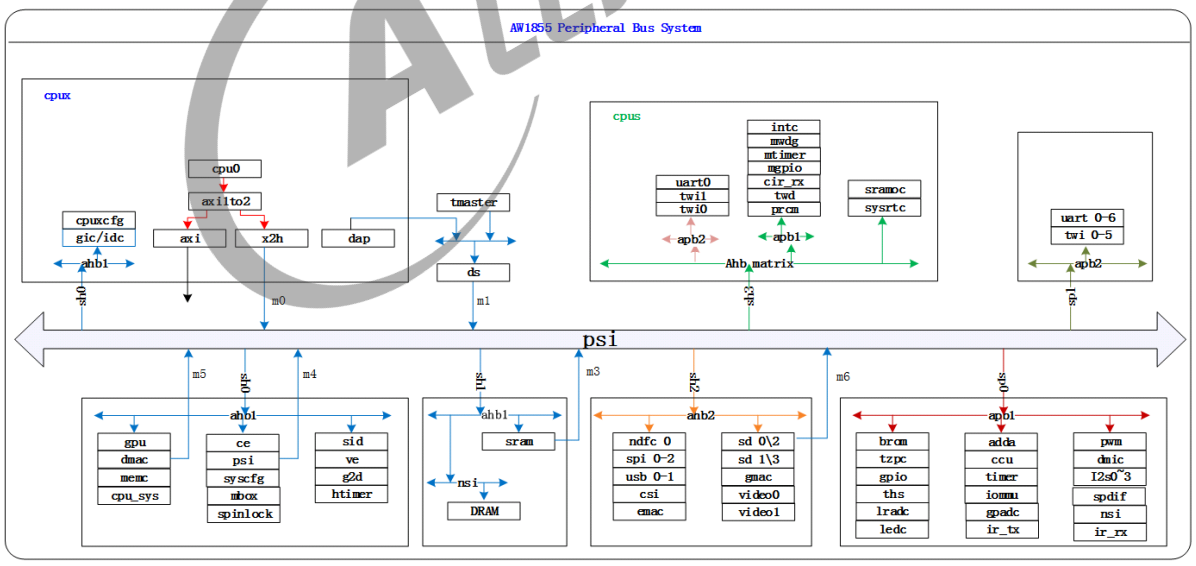

图 2-5: pic3

2.2 模块配置

2.2.1 Device tree 源码结构和路径

驱动使用或应用层开发人员一般不需要修改此文件中 clk 相关配置；驱动开发人员需要在 dtsi 文档中配置 clk，具体配置方法详见 2.2.3 章节。

以 linux-5.10 内核版本为例：

- SoC 级设备树路径：

bsp/configs/linux-5.10/sun*.dtsi

- 板级设备树 (board.dts) 路径：

device/config/chips/{PRODUCT}/configs/{BOARD}/linux-5.10/board.dts

- device tree 的源码包含关系如下：

board.dts `--------sun*.dtsi

2.2.2 menuconfig 配置

- SDK 开发环境

首先进入 SDK 开发环境根目录执行：

```
source build/envsetup.sh ----配置SDK环境变量。
./build.sh config      ----选择对应平台。
./build.sh menuconfig   ----进入内核配置主界面。

-> Allwinner BSP  --->
   -> Device Drivers  --->
      -> Clock Drivers  --->
         -> <*> Clock Support for Allwinner SoCs    /* 选中 */
         -> /* 选中当前平台对应选项 */
```

进入配置主界面，勾选 AW_CCU 配置项即可。

2.2.3 dts 配置

各个驱动模块作为 CLK 的使用者，可以通过 CLK 子系统配置各自模块需要的时钟。通常情况下，各个驱动模块需要在各自的 dts 节点下配置 CLK 相关的节点，同时在驱动代码中调用相应的 CLK 接口，配置各自的时钟。

以 g2d 模块为例，说明如下:

```
g2d: g2d@6480000 {
    compatible = "allwinner,sunxi-g2d";
    reg = <0x0 0x06480000 0x0 0x3ffff>;
    interrupts = <GIC_SPI 91 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&ccu CLK_BUS_G2D>, <&ccu CLK_G2D>, <&ccu CLK_MBUS_G2D>;
    clock-names = "bus", "g2d", "mbus_g2d";
    resets = <&ccu RST_BUS_G2D>;
    reset-names = "g2d"
    iommus = <&mmu_aw 5 1>;
    assigned-clocks = <&ccu CLK_G2D>; /* 指定CLK_G2D这个时钟 */
    assigned-clock-rates = <300000000>; /* 指定时钟频率 */
    assigned-clock-parents = <&ccu xxx>; /* 指定GLK_G2D的父时钟 */
};
```

dts 中配置方式有两种：

- 1. 使用 “assigned-clock*” 关键字进行配置：如果使用这种方式配置对应的时钟，那么在模块驱动加载时，内核框架就会帮助模块进行时钟配置。
- 2. 使用 “clocks” 以及 “clock-names” 关键字进行配置：模块需要在自己的驱动中调用相应的 CLK 接口进行时钟配置（对应的时钟接口查看 3.2）。

说明

“resets” 与 “reset-names” 关键字与 reset 子系统相关，模块在驱动代码中需要调用 reset 子系统接口进行相应的复位操作（对应的时钟接口查看 3.2）。

2.2.4 CLK 子系统使用说明

对于大多数模块，时钟与复位单元是必要的硬件资源。时钟单元负责为当前模块提供必要的外部时钟源，复位单元负责为当前模块提供复位信号与解复位信号。

对于一般的模块驱动来说，都需要进行如下的时钟和复位操作：

1. 获取当前模块的 clk 句柄。
2. 调整对应时钟的时钟频率并开启对应时钟。
3. 获取当前模块的 reset 句柄。
4. 解除当前模块的复位状态（模块在上电后一般都为复位状态，因此需要解复位后才能对模块进行寄存器操作）。

对于以上操作，clk 以及 reset 子系统都已经提供对应的操作接口。

2.3 源码结构

以 sun50iw9 平台为例，CCU 的源码结构如下图所示：

```
.bsp
 ├──── drivers
 │   ├──── clk
| | ├──── sunxi-ng
| | | ├──── ccu_common.c
| | | ├──── ccu_common.h
| | | ├──── ccu_div.c
| | | ├──── ccu_div.h
| | | ├──── ccu_frac.c
| | | ├──── ccu_frac.h
| | | ├──── ccu_gate.c
| | | ├──── ccu_gate.h
| | | ├──── ccu_mp.c
| | | ├──── ccu_mp.h
| | | ├──── ccu_mult.c
| | | ├──── ccu_mult.h
| | | ├──── ccu_mux.c
| | | ├──── ccu_mux.h
| | | ├──── ccu_nk.c
| | | ├──── ccu_nk.h
| | | ├──── ccu_nkm.c
| | | ├──── ccu_nkm.h
| | | ├──── ccu_nkmp.c
| | | ├──── ccu_nkmp.h
| | | ├──── ccu_nm.c
| | | ├──── ccu_nm.h
| | | ├──── ccu_phase.c
| | | ├──── ccu_phase.h
| | | ├──── ccu_reset.c
| | | ├──── ccu_reset.h
| | | ├──── ccu_sdm.c
| | | ├──── ccu_sdm.h
| | | ├──── ccu-sun50iw9.c    //sun50iw9平台cpux域时钟实现代码
| | | ├──── ccu-sun50iw9.h    //上述文件头文件，定义时钟数量，具体的时钟宏定义在bsp/include/dt-bindings/clock
      中
| | | ├──── ccu-sun50iw9-r.c   //sun50iw9平台cpus域时钟实现代码
| | | ├──── ccu-sun50iw9-r.h   //上述文件头文件，定义时钟数量，具体的时钟宏定义在bsp/include/dt-bindings/clock
      中
| | | ├──── ccu-sun50iw9-rtc.c //sun50iw9平台rtc域时钟实现代码
| | | ├──── ccu-sun50iw9-rtc.h //上述文件头文件，定义时钟数量，具体的时钟宏定义在bsp/include/dt-bindings/
      clock中
| | | ├──── ccu-sunxi-rtc.c
| | | ├──── clk-debugfs.c
| | | ├──── clk-debugfs.h
| | | ├──── Kconfig
| | | └──── Makefile
```

3 模块接口说明

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

3.1 时钟 API 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的 API，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

- 函数原型: struct clk *clk_get(struct device *dev, const char *id)
- 作用：获取管理时钟的时钟句柄。
- 参数：
  - dev: 指向申请时钟的设备句柄。
  - id: 指向要申请的时钟名，可以为 NULL。
- 返回：
  - 成功，返回时钟句柄。
  - 失败，返回错误指针。

⚠ 注意

该接口要与 clk_put 成对使用。

📖 说明

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。

3.2.2 devm_clk_get(推荐使用)

- 函数原型: struct clk *devm_clk_get(struct device *dev, const char *id)
- 作用：获取管理时钟的时钟句柄。
- 参数：
  - dev: 指向申请时钟的设备句柄。
  - id: 指向要申请的时钟名（字符串），可以为 NULL。
- 返回：
  - 成功，返回时钟句柄。
  - 失败，返回错误指针。

说明

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)。

3.2.3 clk_put

- 函数原型: void clk_put(struct clk *clk)
- 作用：释放管理时钟的时钟句柄。
- 参数：
  - clk: 指向申请时钟的设备句柄。
- 返回：
  - 没有返回值。

⚠ 注意

该接口要与 clk_get 成对使用。

说明

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。

3.2.4 of_clk_get(推荐使用)

- 函数原型: struct clk *of_clk_get(struct device_node *np, int index)
- 作用：获取管理时钟的时钟句柄。
- 参数：
  - np: 指向设备的 device_node 节点的指针。

- index: 在 dts 中属性的索引值。

- 返回：
  - 成功，返回时钟句柄。
  - 失败，返回错误指针。

3.2.5 clk_set_parent

- 函数原型:int clk_set_parent(struct clk *clk, struct clk *parent)
- 作用：用于设定指定时钟的父时钟。
- 参数：
  - clk: 待操作的时钟句柄。
  - parent: 父时钟的时钟句柄。
- 返回：
  - 成功，返回 0。
  - 失败，返回负值的错误码。

3.2.6 clk_get_parent

- 函数原型: struct clk *clk_get_parent(struct clk *clk)
- 作用：用于获取指定时钟的父时钟。
- 参数：
  - clk: 待操作的时钟句柄。
- 返回：
  - 成功，返回父时钟句柄。
  - 失败，返回 NULL。

3.2.7 clk_prepare

- 函数原型: int clk_prepare(struct clk *clk)
- 作用：用于 prepare 指定的时钟。需要与 enable 接口联合使用，先 prepare 再 enable。
- 参数：
  - clk: 待操作的时钟句柄。
- 返回：
  - 成功，返回 0。

— 失败，返回错误码。

说明

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。

3.2.8 clk_enable

- 函数原型：int clk_enable(struct clk *clk)
- 作用：用于 enable 指定的时钟。
- 参数：
  - clk: 待操作的时钟句柄。
- 返回：
  - 成功，返回 0。
  - 失败，返回错误码。

说明

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，只能在可以睡眠的上下文调用该 API。

3.2.9 clk_prepare_enable（推荐使用）

- 函数原型: int clk_prepare_enable(struct clk *clk)
- 作用：用于 prepare 并使能指定的时钟。
- 参数：
  - clk: 待操作的时钟句柄。
- 返回：
  - 成功，返回 0。
  - 失败，返回错误码。

说明

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，只能在可以睡眠的上下文调用该 API。

3.2.10 clk_disable

- 函数原型: void clk_disable(struct clk *clk)
- 作用：用于关闭指定的时钟。
- 参数：
  - clk: 待操作的时钟句柄。
- 返回：
  - 无返回值。

⚠ 注意

该接口要与 clk_enable 成对使用。

3.2.11 clk_unprepare

- 函数原型: void clk_unprepare(struct clk *clk)
- 作用：用于 unprepare 指定的时钟。
- 参数：
  - clk: 待操作的时钟句柄。
- 返回：
  - 无返回值。

⚠ 注意

该接口要与 clk_prepare 成对使用。

📖 说明

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 同时成 disable 和 unprepare 的工作，只能在可能睡眠的上下文调用该 API。

3.2.12 clk_disable_unprepare（推荐使用）

- 函数原型: void clk_disable_unprepare(struct clk *clk)
- 作用：用于 unprepare 和关闭指定的时钟。
- 参数：
  - clk: 待操作的时钟句柄。

- 返回：
  - 无返回值。

⚠ 注意

该接口要与 clk_prepare_enable 成对使用。

说明

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），clk_disable_unprepare 同时完成 disable 和 unprepare 的工作，只能在可睡眠的上下文调用该 API。

3.2.13 clk_get_rate

- 函数原型: unsigned long clk_get_rate(struct clk *clk)
- 作用：用于获取指定时钟当前的频率，无论时钟是否已经使能。
- 参数：
  - clk: 待操作的时钟句柄。
- 返回：
  - 成功，返回指定时钟的频率。
  - 失败，返回 0。

3.2.14 clk_set_rate

- 函数原型：int clk_set_rate(struct clk *clk, unsigned long rate)
- 作用：用于设置时钟频率成功，无论时钟是否已经使能。
- 参数：
  - clk: 待操作的时钟句柄。
  - rate: 希望设置的频率。
- 返回：
  - 成功，返回 0。
  - 失败，返回错误码。

3.2.15 devm_reset_control_get

- 函数原型：struct reset_control *devm_reset_control_get(struct device *dev, const char *id)
- 作用：获取相应的 reset 句柄。

- 参数：
  - dev: 指向申请 reset 资源的设备句柄。
  - id: 指向要申请的 reset 的资源名（字符串），可以为 NULL。
- 返回：
  - 成功，返回 reset 句柄。
  - 失败，返回错误指针。

3.2.16 reset_control_deassert

- 函数原型: int reset_control_deassert(struct reset_control *rstc)
- 作用：对传入的 reset 资源进行解复位操作。
- 参数：
  - dev: 指向申请 reset 资源的设备句柄。
- 返回：
  - 成功，返回 0。
  - 失败，返回错误指针。

3.2.17 reset_control_assert

- 函数原型: int reset_control_assert(struct reset_control *rstc)
- 作用：对传入的 reset 资源进行复位操作。
- 参数：
  - dev: 指向申请 reset 资源的设备句柄。
- 返回：
  - 成功，返回 0。
  - 失败，返回错误码。

3.2.18 reset_control_reset

- 函数原型: int reset_control_reset(struct reset_control *rstc)
- 作用：对传入的 reset 资源先进行复位操作，然后等待 10us，再进行解复位操作。
- 参数：
  - dev: 指向申请 reset 资源的设备句柄。
- 返回：
  - 成功，返回 0。
  - 失败，返回错误码。

3.3 重要流程介绍

3.3.1 set_parent

- set_parent 调用过程

```
clk_set_parent
  ->clk_core_set_parent_nolock(clk, parent)
    ->__clk_set_parent(clk, parent, index)
      ->ops->set_parent(core, index)
```

以上为 set_parent 的重要接口，接下来介绍每个函数的具体实现：

- clk_core_set_parent_nolock

```
/* verify ops for multi-parent clks */
if (core->num_parents > 1 && !core->ops->set_parent)
    return -EPERM;

/* check that we are allowed to re-parent if the clock is in use */
if ((core->flags & CLK_SET_PARENT_GATE) && core->prepare_count)
    return -EBUSY;

if (clk_core_rate_is_protected(core))
    return -EBUSY;

/* try finding the new parent index */
if (parent) {
    p_index = clk_fetch_parent_index(core, parent);
    if (p_index < 0) {
        pr_debug("%s: clk %s can not be parent of clk %s\n",
                __func__, parent->name, core->name);
        return p_index;
    }
    p_rate = parent->rate;
}
```

在此函数中会检查当前时钟是否有父时钟、父时钟个数是否大于 1，是否已实现 set_parent 接口以及是否被设置 CLK_SET_PARENT_GATE 标志位，并且对特殊情况进行处理。

利用当前父时钟信息得到父时钟的 index

- *__clk_set_parent*

```
old_parent = __clk_set_parent_before(core, parent);

trace_clk_set_parent(core, parent);

/* change clock input source */
if (parent && core->ops->set_parent)
    ret = core->ops->set_parent(core->hw, p_index);
```

```
trace_clk_set_parent_complete(core, parent);

if (ret) {
      flags = clk_enable_lock();
      clk_reparent(core, old_parent);
      clk_enable_unlock(flags);
      __clk_set_parent_after(core, old_parent, parent);

      return ret;
}
__clk_set_parent_after(core, parent, old_parent);
```

__clk_set_parent_before：在这个函数中获取旧的父时钟并赋予新父时钟

__clk_set_parent_after：关闭旧的父时钟

- *__set_parent*

接下来是 sunxi-ng 的框架，在设置父时钟时，所有的类型统一会调到 ccu_mux_helper_set_parent，重要流程如下：

```
if (cm->table)
      index = cm->table[index];

spin_lock_irqsave(common->lock, flags);

reg = readl(common->base + common->reg);
if (common->features & CCU_FEATURE_KEY_FIELD_MOD) {
      reg = reg | common->key_value;
}
reg &= ~GENMASK(cm->width + cm->shift - 1, cm->shift);
writel(reg | (index << cm->shift), common->base + common->reg);
/* some clks need set mux reg repeatedly to fix ic bug */
if (common->features & CCU_FEATURE_REPEAT_SET_MUX)
      writel(reg | (index << cm->shift), common->base + common->reg);

spin_unlock_irqrestore(common->lock, flags);
```

拿出在时钟定义时 mux 的偏移量以及位宽，写入寄存器。

3.3.2 get_parent

- 主要逻辑

```
parent = !clk->core->parent ? NULL : clk->core->parent->hw->clk;
return parent;
```

即之前从 core 结构体中拿取 parent 即可。

3.3.3 set_rate

- 调用过程

```
clk_set_rate
  ->clk_core_set_rate_nolock
    ->clk_core_req_round_rate_nolock
      ->clk_core_round_rate_nolock
    ->clk_change_rate
```

以上为在 set_rate 过程中比较重要的接口，接下来对重点接口做详细的描述：

- clk_change_rate

此接口是关键，修改频率的操作在这个接口中进行

首先会判断当前时钟有没有被赋予新的父时钟，若有，则刷新当前父时钟并将父时钟的频率赋给当前时钟。之后会调到 ng 框架中的 set_rate 流程里，在 set_rate 之后会重新计算当前时钟的频率并刷新 core 结构体中的 rate 变量，最后会刷新当前时钟下的所有子时钟频率。

接下来讲解 ng 框架中 set_rate 的具体逻辑，ng 框架根据时钟计算公式的不同封装了多种类型，每种类型设置频率的方式都不一致，这里对经常用到的 ccu_mp 类型做一简单讲解：

- set_rate

主要逻辑是根据当前需要设置的频率以及时钟的计算过程去推算写入寄存器的具体数值，最后在写入寄存器。

需要重点注意：CCU_FEATURE_MP_NO_INDEX_MODE 这个宏比较常用，用来区分 p 因子是以指数形式增长还是以整数增长。

3.3.4 get_rate

- 调用过程

```
clk_get_rate
  ->clk_core_get_rate_recalc
    ->__clk_recalc_rates
    ->clk_core_get_rate_nolock
```

- 主要逻辑代码如下：

```
if (core && (core->flags & CLK_GET_RATE_NOCACHE))
    __clk_recalc_rates(core, 0);

return clk_core_get_rate_nolock(core);
```

- CLK_GET_RATE_NOCACHE：此标志位表示每次获取频率的时候不从缓存的结构体中拿取，而是重新根据寄存器的值计算频率。

3.3.5 round_rate

- 调用过程

```
clk_round_rate
  -> clk_core_round_rate_nolock
    ->clk_core_determine_round_nolock
    ->clk_core_round_rate_nolock
```

- clk_core_round_rate_nolock 的主要逻辑如下：

```
if(clk_core_can_round(core))
      return clk_core_determine_round_nolock(core, req);
else if (core->flags & CLK_SET_RATE_PARENT)
      return clk_core_round_rate_nolock(core->parent, req);
```

在 round_rate 时，如果具体时钟实现了determine_rate或者round_rate接口，则会继续执行clk_core_determine_round_nolock函数。

若需要计算频率的时钟中设置了CLK_SET_RATE_PARENT标志位，则表示在计算频率的时候需要更换父时钟频率，此时会往上查找并重新计算父时钟频率。

clk_core_determine_round_nolock 的主要逻辑如下：

```
if (clk_core_rate_is_protected(core)) {
        req->rate = core->rate;
  } else if (core->ops->determine_rate) {
        return core->ops->determine_rate(core->hw, req);
   } else if (core->ops->round_rate) {
        rate = core->ops->round_rate(core->hw, req->rate,
                        &req->best_parent_rate);
```

在 ng 框架中，nkmp、nk、nm 类型的时钟设置了round_rate属性，而 mp 类型的时钟设置了determine_rate属性，mp 类型为模块时钟最常见的类型。

4 调试方法

4.1 常用 debug 方法说明

4.1.1 clk tree

1. 需要开启 DEBUG_FS：

```
make kernel_menuconfig

---> Kernel hacking
 ---> Compile-time checks and compiler options
  ---> Debug Filesystem (选中)
```

2. 打印 clk tree，查看 clk 频率和父时钟是否正确，按以下步骤操作：

```
# 挂载debug节点
mount -t debugfs none /sys/kernel/debug
console:/ # ls sys/kernel/debug/clk/
clk_dump         clk_orphan_dump    clk_orphan_summary  clk_summary
console:/ # cat sys/kernel/debug/clk/clk_summary
  clock                enable_cnt  prepare_cnt      rate   accuracy   phase
----------------------------------------------------------------------------------
 pll_periph0div25m              0        0   25000000        0 0
   ephy_25m                     0        0   25000000        0 0
 hoscdiv32k                     1        1      32768        0 0
   hosc32k                      1        1      32768        0 0
     losc_out                   2        2      32768        0 0
 osc48m                         0        0   48000000        0 0
   osc48md4                     0        0   12000000        0 0
     usbohci3_12m               0        0   12000000        0 0
     usbohci2_12m               0        0   12000000        0 0
     usbohci1_12m               0        0   12000000        0 0
     usbohci0_12m               0        0   12000000        0 0
 hosc                          20       21   24000000        0 0
   sdmmc0_mod                   0        0     800000        0 0
   cpurcir                      1        1   24000000        0 0
   dcxo_out                     0        0   24000000        0 0
   cpurapbs2                    0        0   24000000        0 0
     cpurcan                    0        0   24000000        0 0
   cpurcpus                     1        1   24000000        0 0
     cpurahbs                   1        1   24000000        0 0
       cpurapbs1                2        2   24000000        0 0
         stwi                   1        1   24000000        0 0
         cpurpio                1        1   24000000        0 0
   csi_master1                  0        0   24000000        0 0
   csi_master0                  0        0   24000000        0 0
   usbphy3                      2        2   24000000        0 0
   usbphy2                      2        2   24000000        0 0
```

```
   usbphy1                2       2   24000000       0 0
   usbphy0                1       1   24000000       0 0
   ths                  1       1   24000000       0 0
   ts                   0       0   24000000       0 0
   gpadc                  0       0   24000000       0 0
   spi1                 0       0   24000000       0 0
   spi0                 0       0   24000000       0 0
   ...
```

3.Linux-6.6 内核 clk tree 新特性

```
/sys/kernel/debug # cat clk/clk_summary
     enable  prepare  protect                duty   hardware                connection
clock   count   count   count   rate   accuracy   phase  cycle    enable   consumer            id
---------------------------------------------------------------------------------------------------------------
rc-16m     0     0       0  16000000 300000000  0     50000    Y      deviceless       no_connection_id
dcxo24M   40    45       0  24000000 0       0     50000    Y      8220000.timestamp   hosc
                                                  3000000.pinctrl    hosc
                                                  3000000.pinctrl    hosc
                                                  timer0@3208000     parent
```

上述新特性说明如下：

```
deviceless       no_connection_id:   表示没有模块使用rc-16m时钟

8220000.timestamp   hosc:          表示timestamp模块在使用dcxo24M时钟，并且该模块将 dcxo24M时钟命名为hosc

timer0@3208000     parent:         表示timer0模块在时钟在使用dcxo24M时钟，并且该模块将dcxo24M时钟命名为parnet
```

Linux-6.6 内核的 clk_summary 新增了 **consumer** 和 **connection id** 两个维度，其中：

- **consumer**：显示哪些模块获取了当前时钟，即当前时钟的消费者

模块驱动程序会在 probe 阶段请求并启用所需的时钟，这通常通过调用 clk_get() 或 devm_clk_get() 函数来实现。如果成功获取到时钟，则说明消费者与提供者之间的连接已经建立，当前时钟的 consumer 就会显示该模块驱动的名称。

- **connection id**：显示使用当前时钟的模块对该时钟的命名，对应 dtsi 中的clk-names；

举例，当时钟选择dcxo24M时，对应的connection id即为parent：

```
timer0: timer0@3208000{
   ......
   clock-names = "parent", "bus", "timer0-mod", "timer1-mod";
   clocks = <&dcxo24M>, <&ccu CLK_BUS_TIMER0_AHB>,
         <&ccu CLK_TIMER0_0>, <&ccu CLK_TIMER0_1>;
   ......
};
```

4.1.2 clk debugfs

利用 debugfs 提供的节点测试 clk 接口是否存在问题，按以下步骤操作：

1. 在内核菜单项打开 clk debugfs 的配置：AW_CCU_DEBUG。

2. 利用 clk debugfs 提供的节点测试。

通过步骤 1 中在内核菜单项打开CONFIG_AW_CCU_DEBUG这个配置项后，挂载上 debugfs，可以看到 debugfs 目录下存在 ccudbg 目录，则可以进行 debug 了，如下所示：

```
mount -t debugfs none /sys/kernel/debug
console:/ # ls sys/kernel/debug/ccudbg
command info name param start

/*
 * debug clk_get_parent()接口。
 */
echo getparent > sys/kernel/debug/ccudbg/command
echo cpuapb > sys/kernel/debug/ccudbg/name    /*cpuapb为需要调试的时钟*/
echo 1 > sys/kernel/debug/ccudbg/start
cat sys/kernel/debug/ccudbg/info          /*查看返回的父时钟*/
结果如下：
console:/ # cat sys/kernel/debug/ccudbg/info
cpu

/*
 * debug clk_set_rate()接口
 */
 echo setrate > sys/kernel/debug/ccudbg/command
 echo pll_csi > sys/kernel/debug/ccudbg/name    /* pll_csi为需要调试的时钟 */
 echo 600000000 > sys/kernel/debug/ccudbg/param  /* 设置期望设置的频率 */
 echo 1 > sys/kernel/debug/ccudbg/start

查看结果如下：
console:/ # cat sys/kernel/debug/ccudbg/info
600000000

console:/ # cat sys/kernel/debug/clk/clk_summary | grep "pll_csi"
   pll_csi                    0        0   600000000        0 0

clk debugfs提供的常用测试命令如下所示：
getparents：获取某个时钟的所有父时钟。
getparent：获取某个时钟当前的父时钟。
setparent:设置某个时钟的父时钟。
getrate：获取某个时钟的频率。
setrate:设置某个时钟的频率。
is_enabled:判断某个时钟是否enable。
enable：使能某个时钟。
disable：关闭某个时钟。
```

4.1.3 利用 sunxi_dump 读写相应寄存器

需要开启 SUNXI_DUMP 模块：

```
make kernel_menuconfig

-> Allwinner BSP  --->
   -> Device Drivers  --->
      -> Dump-Reg Drivers  --->
```

```
        -> <*> Dump-Reg Support for Allwinner SoCs     /* 选中 */
        -> [*]   Enable dumpreg dynamic debug         /* 选中 */
```

使用方法：

```
cd /sys/class/sunxi_dump/
1.查看一个寄存器，如查看DE时钟寄存器，根据spec，看寄存器含义。
 echo 0x03001600 > dump ;cat dump

结果如下：
cupid-p1:/sys/class/sunxi_dump # echo 0x03001600 > dump ;cat dump
0x80000000

2.写值到寄存器上，如关闭DE时钟。
 echo 0x03001600 0x00000000 > write ;cat write

结果如下：
reg             to_write    after_write
0x0000000003001600  0x00000000  0x00000000

3.查看一片连续寄存器。
 echo 0x03001000,0x03001fff > dump;cat dump

结果如下：
ccupid-p1:/sys/class/sunxi_dump # echo 0x03001000,0x03001fff > dump;cat dump

0x0000000003001000: 0x8a003a00 0x00000000 0x00000000 0x00000000
0x0000000003001010: 0xb8003900 0x00000000 0x08002301 0x00000000
0x0000000003001020: 0xb8003100 0x00000000 0x89003100 0x00000000
0x0000000003001030: 0x80003203 0x00000000 0x00000000 0x00000000
0x0000000003001040: 0x88006203 0x00000000 0x88004701 0x00000000
0x0000000003001050: 0x88006213 0x00000000 0x80001700 0x00000000
0x0000000003001060: 0x88001c00 0x00000000 0x00000000 0x00000000
0x0000000003001070: 0x00000000 0x00000000 0x89021501 0x00000000
0x0000000003001080: 0x00000000 0x00000000 0x00000000 0x00000000
0x0000000003001090: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000030010a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000030010b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000030010c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000030010d0: 0x00000000 0x00000000 0x00000000 0x00000000
...
```

通过上述方式，可以查看寄存器值，从而发现问题所在。

5 模块使用范例

以 spi 模块的时钟处理部分作为 demo 分析：

dtsi 配置如下：

```
spi0: spi@4025000 {
   #address-cells = <1>;
   #size-cells = <0>;
   compatible = "allwinner,sun8i-spi";
   device_type = "spi0";
   reg = <0x0 0x04025000 0x0 0x1000>;
   interrupts = <GIC_SPI 15 IRQ_TYPE_LEVEL_HIGH>;
   clocks = <&clk_pll_periph0300m>, <&clk_spi0>;
   status = "disabled";
};
```

dts 配置如下：

```
spi0_pins_default: spi0@0 {
   pins = "PC12", "PC2", "PC4"; /* clk, mosi, miso */
   function = "spi0";
   drive-strength = <10>;
};
spi0_pins_cs: spi0@1 {
   pins = "PC3", "PC15", "PC16"; /* cs, hold, wp */
   function = "spi0";
   drive-strength = <10>;
   bias-pull-up;
};
spi0_pins_sleep: spi0@2 {
   pins = "PC12", "PC2", "PC4", "PC3", "PC15", "PC16";
   function = "io_disabled";
};
```

驱动时钟处理代码如下：

```
struct sunxi_spi {
 ...
   struct clk *pclk;  /* PLL clock */
   struct clk *mclk;  /* spi module clock */
   struct clk *bus_clk; /*spi bus clock*/
   struct reset_control *reset; /*reset clock*/
 ...
};

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,即上述设备树节点定义的clk_pll_periph0300m */
   sspi->pclk = of_clk_get(sspi->pdev->dev.of_node, 0);
```

```
    /* 判断clk句柄的有效性 */
    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,即上述设备树节点定义的clk_spi0 */
    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);
}
```

著作权声明

商标声明

免责声明