



Linux CCU 开发指南

**版本号: 2.2
发布日期: 2022.05.16**

版本历史

版本号	日期	制/修订人	内容描述
1.0	2020.06.29	AWA1440	添加初版
2.0	2020.11.19	AWA1527	for linux-5.4
2.1	2021.05.24	XAA0191	增加部分使用描述
2.2	2022.05.16	AWA1538	修正部分使用描述



目 录

1 概述	1
1.1 编写目的	1
1.2 适用范围	1
1.3 相关人员	1
2 模块介绍	2
2.1 模块功能介绍	2
2.2 相关术语介绍	2
2.3 源码结构介绍	2
2.4 模块配置介绍	4
2.4.1 kernel menuconfig 配置	4
2.4.2 device tree 源码结构和路径	5
2.4.2.1 linux4.9	5
2.4.2.2 linux5.4	5
2.4.3 dts 配置	5
2.4.4 CLK 子系统使用说明	6
3 模块接口说明	7
3.1 时钟 API 定义	7
3.2 时钟 API 说明	7
3.2.1 clk_get	7
3.2.2 devm_clk_get(推荐使用)	8
3.2.3 clk_put	8
3.2.4 of_clk_get(推荐使用)	8
3.2.5 clk_set_parent	9
3.2.6 clk_get_parent	9
3.2.7 clk_prepare	9
3.2.8 clk_enable	10
3.2.9 clk_prepare_enable (推荐使用)	10
3.2.10 clk_disable	11
3.2.11 clk_unprepare	11
3.2.12 clk_disable_unprepare (推荐使用)	11
3.2.13 clk_get_rate	12
3.2.14 clk_set_rate	12
3.2.15 devm_reset_control_get	12
3.2.16 reset_control_deassert	13
3.2.17 reset_control_assert	13
3.2.18 reset_control_reset	13
3.2.19 sunxi_periph_reset_assert	14

3.2.20 sunxi_periph_reset_deassert	14
4 模块使用范例	15
5 FAQ	17
5.1 常用 debug 方法说明	17
5.1.1 clk tree	17
5.1.1.1 clk debugfs	18
5.1.1.2 利用 sunxi_dump 读写相应寄存器	20



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

表 1-1: 适用产品列表

内核版本	驱动文件
Linux-4.9 及以上	drivers/clk/sunxi*/*.c

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。

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。

2.2 相关术语介绍

表 2-1: CCU 模块相关术语介绍

术语	解释说明
SUNXI	Allwinner 一系列 SOC 硬件平台
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

2.3 源码结构介绍

CCU 的源码结构如下图所示：

```
linux
|
|-- drivers
|   |-- clk
|       |-- sunxi    //适用于linux-4.9
|           |-- clk-periph.h
|           |-- clk-periph.c
```


2.4 模块配置介绍

2.4.1 kernel menuconfig 配置

- tina 开发环境

首先进入 tina 开发环境根目录执行：

```
source build/envsetup.sh ----配置tina环境变量。
lunch ----选择对应平台。
make kernel_menuconfig ----进入内核配置主界面。
```

- longan 开发环境

首先进入 longan 开发环境根目录执行：

```
source build/envsetup.sh ----配置longan环境变量，选择对应的平台。
./build.sh menuconfig ----进入内核配置主界面。
```

在 sunxi linux-4.9 平台上，目前 ccu 驱动是依赖 CONFIG_ARCH_SUNXI 这个宏的，因此在内核 menuconfig 菜单下，目前没有提供配置菜单。

Linux-5.4 内核版本，进入配置主界面，并进入如下目录勾选对应驱动：

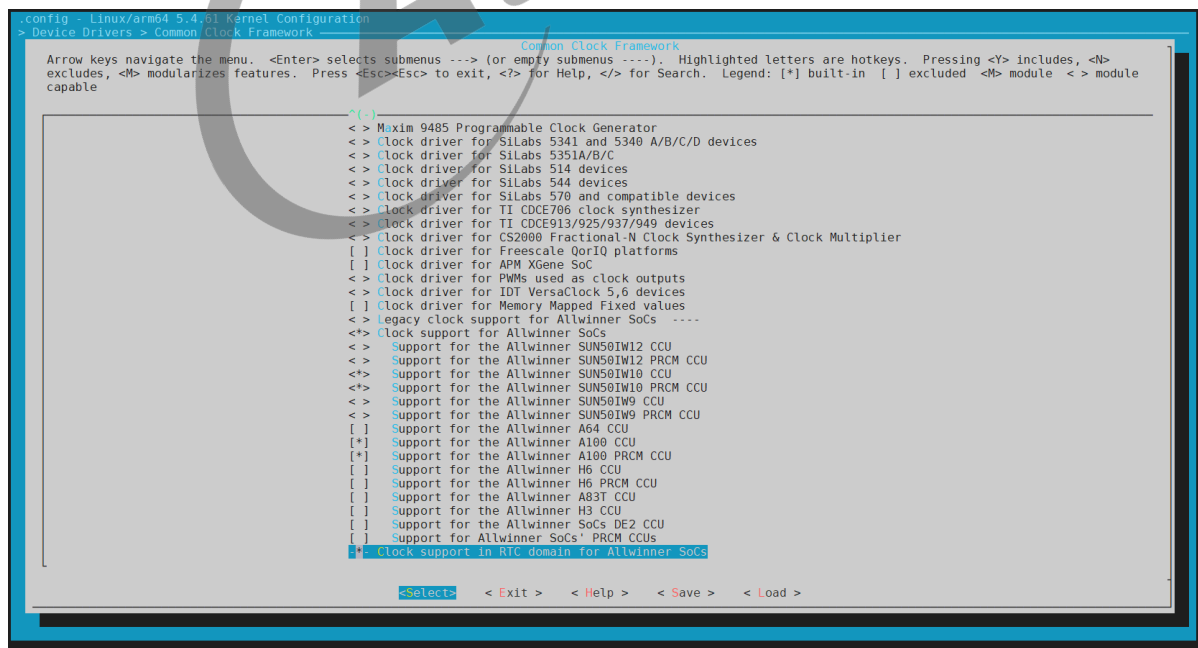


图 2-1: 内核 menuconfig 菜单

2.4.2 device tree 源码结构和路径

2.4.2.1 linux4.9

- 设备树文件的配置是该 SoC 所有方案的通用配置，对于 ARM64 CPU 而言，设备树的路径为：kernel/{KERNEL}/arch/arm64/boot/dts/sunxi/sun*-clk.dtsi。
- 设备树文件的配置是该 SoC 所有方案的通用配置，对于 ARM32 CPU 而言，设备树的路径为：kernel/{KERNEL}/arch/arm/boot/dts/sun*-clk.dtsi。
- 板级设备树 (board.dts) 路径：/device/config/chips/{IC}/configs/{BOARD}/board.dts。

device tree 的源码结构关系如下：

```
board.dts
|-----sun*.dtsi
|         |-----sun*-pinctrl.dtsi
|         |-----sun*-clk.dtsi
```

2.4.2.2 linux5.4

- SoC 级设备树路径：

```
kernel/arch/riscv/boot/dts/sunxi/sun*.dtsi
```

- 板级设备树 (board.dts) 路径：

```
/device/config/chips/{IC}/configs/{BOARD}/board.dts
```

- device tree 的源码包含关系如下：

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

2.4.3 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。

说明如下：

```
对于linux-4.9:
g2d: g2d@01480000 {
    compatible = "allwinner,sunxi-g2d";
    reg = <0x0 0x01480000 0x0 0x3ffff>;
    interrupts = <GIC_SPI 90 IRQ_TYPE_LEVEL_HIGH>;
    /* 配置模块使用的clk */
    clocks = <&clk_g2d>;
    iommus = <&mmu_aw 6 1>;
};

对于linux-5.4:
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CLK_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）。

2.4.4 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（见 3.2）。

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 NULL。

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 NULL。

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` 结点的指针。

- index: 在 dts 中属性的索引值。
- 返回：
 - 成功，返回时钟句柄。
 - 失败，返回 NULL。

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。
- 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 NULL。

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，返回错误码。

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。

3.2.19 sunxi_periph_reset_assert

- 作用：用于 assert 模块。
- 参数：
 - clk: 待操作的时钟句柄。
- 返回：
 - 成功，返回 0。
 - 失败，返回 1。

注意

该接口是 allwinner 私有 API，不建议驱动调用。(linux-5.4 上已被废弃。)
linux5.4 中建议使用内核通用接口代替:reset_control_assert

3.2.20 sunxi_periph_reset_deassert

- 作用：用于 deassert 模块。
- 参数：
 - clk: 待操作的时钟句柄。
- 返回：
 - 成功，返回 0。
 - 失败，返回 1。

注意

该接口是 allwinner 私有 API，不建议驱动调用。(linux-5.4 上已被废弃)。
linux5.4 中建议使用内核通用接口代替:reset_control_deassert。

说明

该接口和 *sunxi_periph_reset_assert* 一起使用，用于模块 *reset* 操作。

4 模块使用范例

以 spi 模块的时钟处理部分作为 demo 分析：

```

1 struct sunxi_spi {
2     ...
3     struct clk *pclk; /* PLL clock */
4     struct clk *mclk; /* spi module clock */
5     struct clk *bus_clk; /*spi bus clock*/
6     struct reset_control *reset; /*reset clock*/
7     ...
8 };
9
10 static int sunxi_spi_clk_init(struct sunxi_spi *sspi, u32 mod_clk)
11 {
12     int ret = 0;
13     long rate = 0;
14
15     /* 获取index = 0的clk句柄 */
16     sspi->pclk = of_clk_get(sspi->pdev->dev.of_node, 0);
17     /* 判断clk句柄的有效性 */
18     if (IS_ERR_OR_NULL(sspi->pclk)) {
19         SPI_ERR("[spi-%d] Unable to acquire module clock '%s', return %x\n",
20             sspi->master->bus_num, sspi->dev_name, PTR_RET(sspi->pclk));
21         return -1;
22     }
23     /* 获取index = 1的clk句柄并判断有效性 */
24     sspi->mclk = of_clk_get(sspi->pdev->dev.of_node, 1);
25     if (IS_ERR_OR_NULL(sspi->mclk)) {
26         SPI_ERR("[spi-%d] Unable to acquire module clock '%s', return %x\n",
27             sspi->master->bus_num, sspi->dev_name, PTR_RET(sspi->mclk));
28         return -1;
29     }
30     /* 设置clk的父时钟并判断有效性 */
31     ret = clk_set_parent(sspi->mclk, sspi->pclk);
32     if (ret != 0) {
33         SPI_ERR("[spi-%d] clk_set_parent() failed! return %d\n",
34             sspi->master->bus_num, ret);
35         return -1;
36     }
37
38     rate = clk_round_rate(sspi->mclk, mod_clk);
39     /* 设置clk的频率并判断有效性 */
40     if (clk_set_rate(sspi->mclk, rate)) {
41         SPI_ERR("[spi-%d] spi clk_set_rate failed\n", sspi->master->bus_num);
42         return -1;
43     }
44
45     SPI_INF("[spi-%d] mclk %u\n", sspi->master->bus_num, (unsigned)clk_get_rate(sspi->mclk));
46     /* 使能clk并判断有效性 */
47     if (clk_prepare_enable(sspi->mclk)) {

```

```
49     SPI_ERR("[spi-%d] Couldn't enable module clock 'spi'\n", sspi->master->bus_num);  
50     return -EBUSY;  
51 }  
52 /* 获取clk的频率值 */  
53 return clk_get_rate(sspi->mclk);  
54 }
```



5 FAQ

5.1 常用 debug 方法说明

5.1.1 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	
hoscddiv32k	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	
usbhci3_12m	0	0	12000000	0 0	
usbhci2_12m	0	0	12000000	0 0	
usbhci1_12m	0	0	12000000	0 0	
usbhci0_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
hdmi_slow	1	1	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
spil	0	0	24000000	0 0
spi0	0	0	24000000	0 0
...				

5.1.1.1 clk debugfs

利用 debugfs 提供的节点测试 clk 接口是否存在问题 (linux-5.4 暂不支持)，按以下步骤操作：

- 1. 在内核菜单项打开 clk debugfs 的配置，如下图所示：

在命令行中进入内核根目录 (kernel/linux-4.9)，执行 `make ARCH=arm64 arm menuconfig` 进入配置主界面，并按以下步骤操作：首先，选择 Device Drivers 选项进入下一级配置，如下图所示：

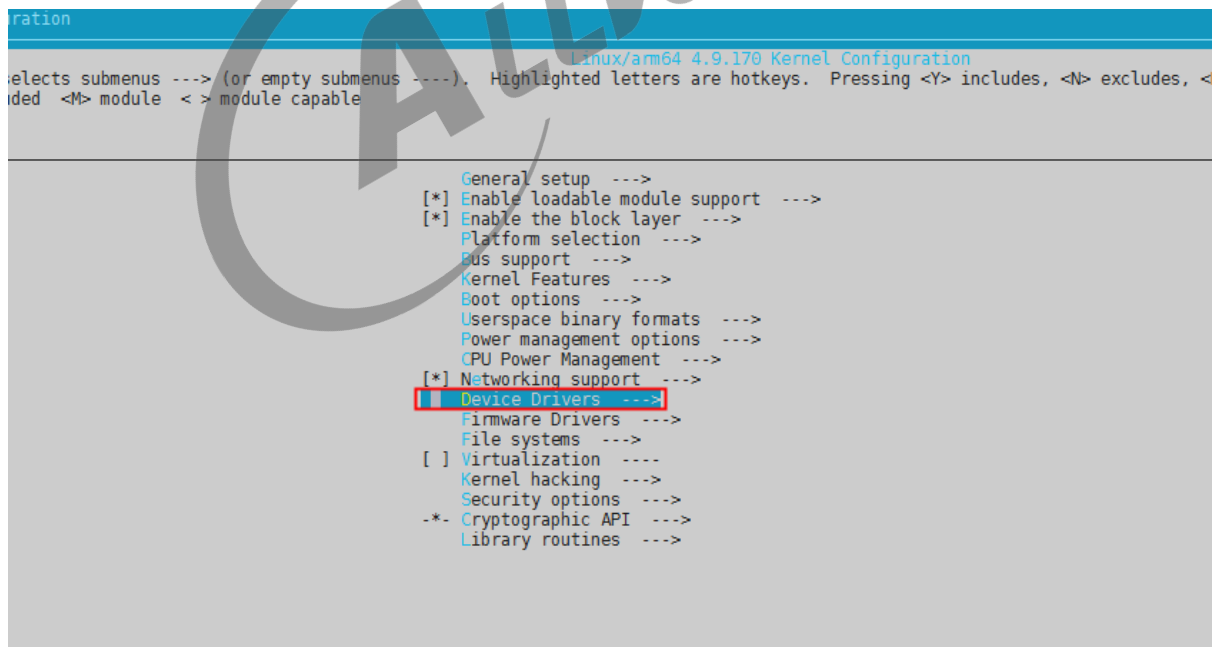


图 5-1: 内核 menuconfig 根菜单

选择 Common Clock Framework，进入下级配置，如下图所示：

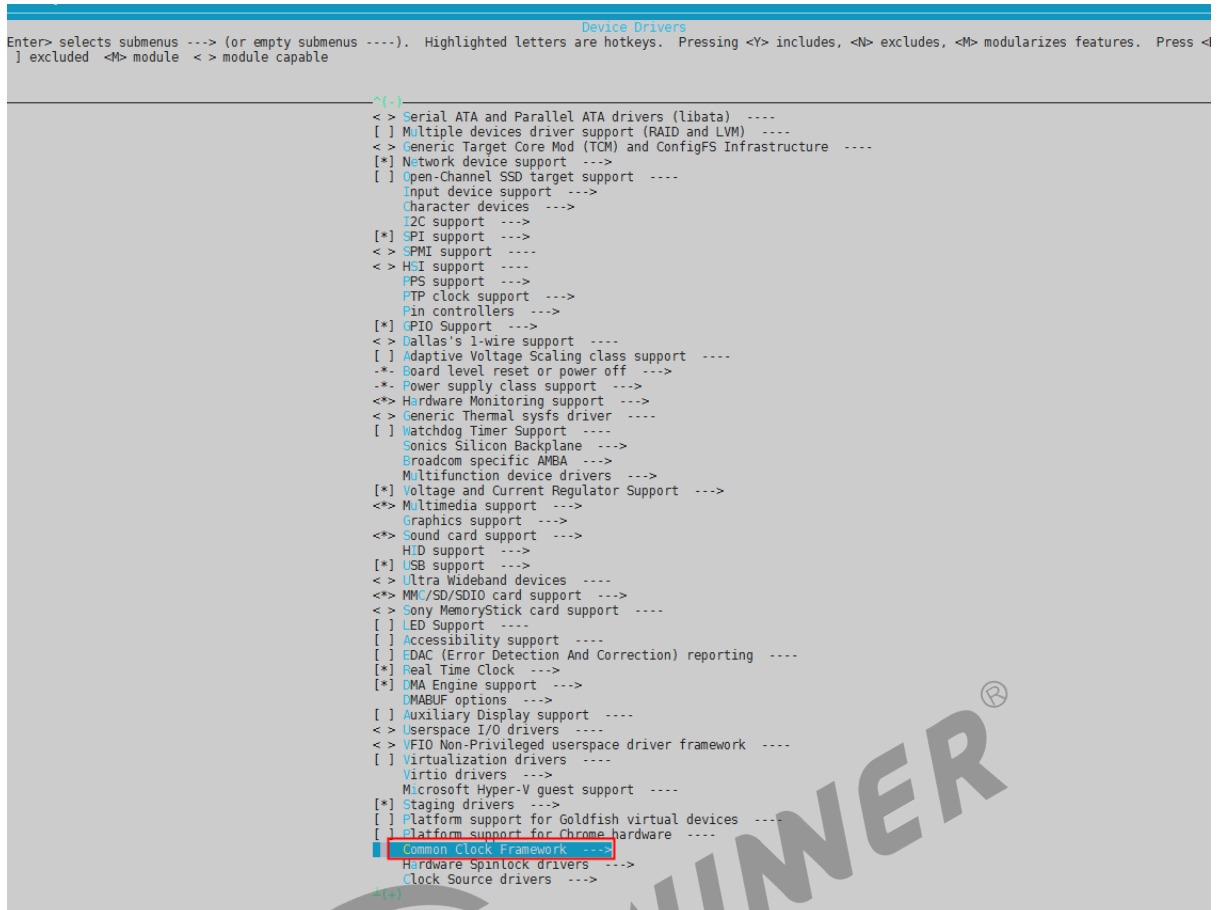


图 5-2: Common clock framework 菜单

选择 DebugFS representation of clock tree, 如下图所示:

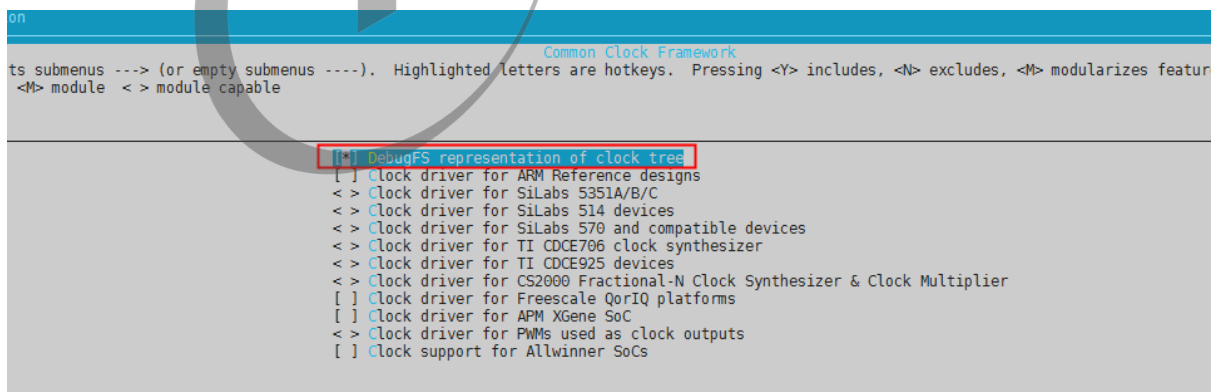


图 5-3: clk DebugFS 菜单

- 2. 利用 clk debugfs 提供的结点测试。

通过步骤 1 中在内核菜单项打开 CONFIG_COMMON_CLK_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从 clk_summary结点获取*/
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从 clk_summary结点获取 */
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。

```

5.1.1.2 利用 sunxi_dump 读写相应寄存器

需要开启 SUNXI_DUMP 模块：

```

make kernel_menuconfig

---> Device Drivers
----> dump reg driver for sunxi platform (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-pl:/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-pl:/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，从而发现问题所在。

著作权声明

版权所有 © 2022 珠海全志科技股份有限公司。保留一切权利。

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明

、、**全志科技**、（不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。