Linux TIMER
开发指南

版本号: 1.3
发布日期: 2025.3.7

版本历史

版本号	日期	制/修订人	内容描述
1.0	2022.08.04	XAA0249	创建初版
1.1	2024.03.20	XAA0312	更新适用范围
1.2	2024.10.23	XAA0309	增加 Linux-5.4 内核版本支持，并修改非 bsp 独立仓库配置。
1.3	2025.3.7	XAA0331	增加 Linux-6.6 内核版本支持。

目　录

1 引言

1.1 编写目的

介绍 Linux 内核中 Timer 驱动的适配和使用方法，为 Timer 设备的使用者和维护者提供参考。

1.2 适用范围

适用于 allwinnertech 全系列平台。

	支持版本	说明
linux	5.4、5.10、5.15、6.1、6.6	
硬件平台	allwinner 全平台	

1.3 相关术语介绍

序号	术语或缩略	解释说明
1	Sunxi	指 Allwinner 的一系列 SOC 硬件平台。
2	Timer	内核定时器，到规定数值后，产生中断标志 Timer 任务的完成

2 模块描述

2.1 模块功能

Timer（定时器/计数器）以一定的频率计数；到规定数值后，产生中断标志 Timer 任务的完成。（如：Timer 为子系统产生内部中断，实现精准有效的系统时间管理。）

2.2 模块配置

2.2.1 dts 配置说明

在不同的 Sunxi 硬件平台中，Timer 控制器的数目也不同，但对于每一个 Timer 控制器来说，在 dtsi 中配置参数相似，如下：

```
soc_timer0: timer@3009000 {
    compatible = "allwinner,hstimer-v100";
    device_type = "soc_timer";
    reg = <0x0 0x03009000 0x0 0x400>;
    interrupt-parent = <&gic>;
    interrupts = <GIC_SPI 89 IRQ_TYPE_LEVEL_HIGH>;
    clock-names = "parent", "bus", "timer0-mod", "timer1-mod";
    clocks = <&sys24M>, <&ccu CLK_BUS_TIMER>, <&ccu CLK_TIMER0>, <&ccu CLK_TIMER1>;
    resets = <&ccu RST_BUS_TIMER0>;
};
```

其中：

1. compatible: 表征具体的设备, 用于驱动和设备的绑定;
2. device_type：表征是 soc 级 timer
3. reg: 设备使用的地址;
4. interrupt-parent: 设备引用的中断节点;
5. interrupts: 设备使用的中断号;
6. clocks: 设备使用的时钟;
7. resets: 设备使用的复位时钟;

2.2.2 menuconfig 配置说明

2.2.2.1 bsp SDK

在命令行中进入 SDK 根目录, 执行./build.sh menuconfig 进入配置主界面, 并按以下步骤操作。首先, 选择 Allwinner BSP 选项进入下一级配置, 如下图所示:

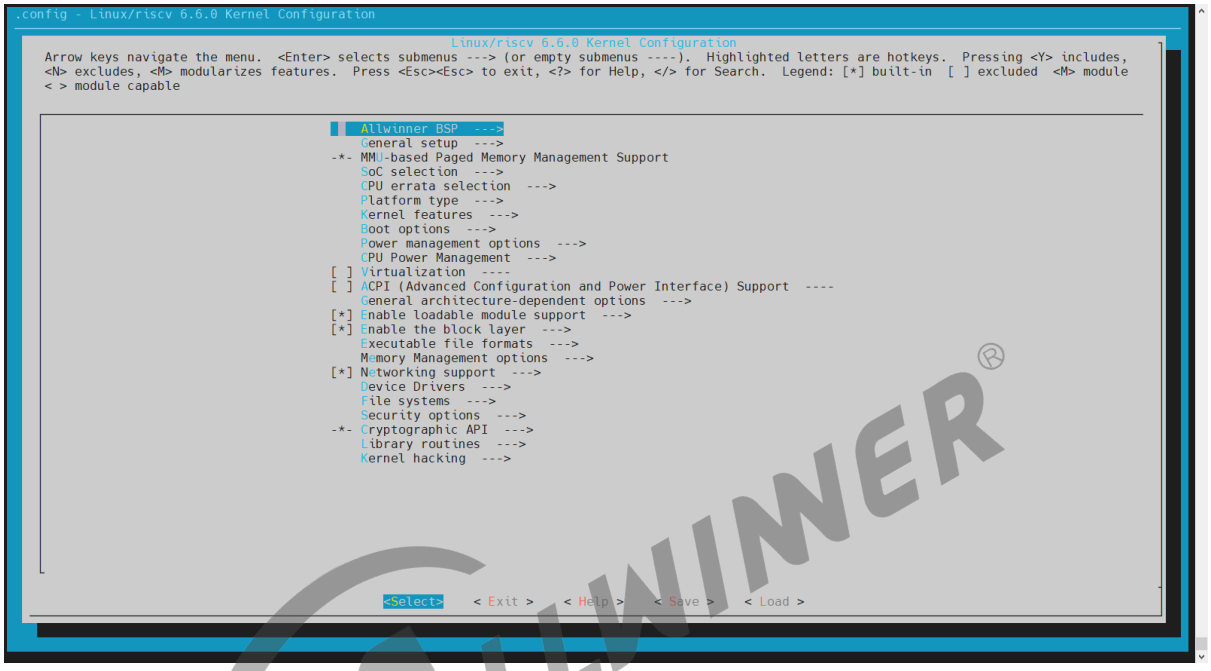

图 2-1: Device-Drivers 配置

然后, 选择 Device Drivers 选项, 进入下一级配置, 如下图所示:

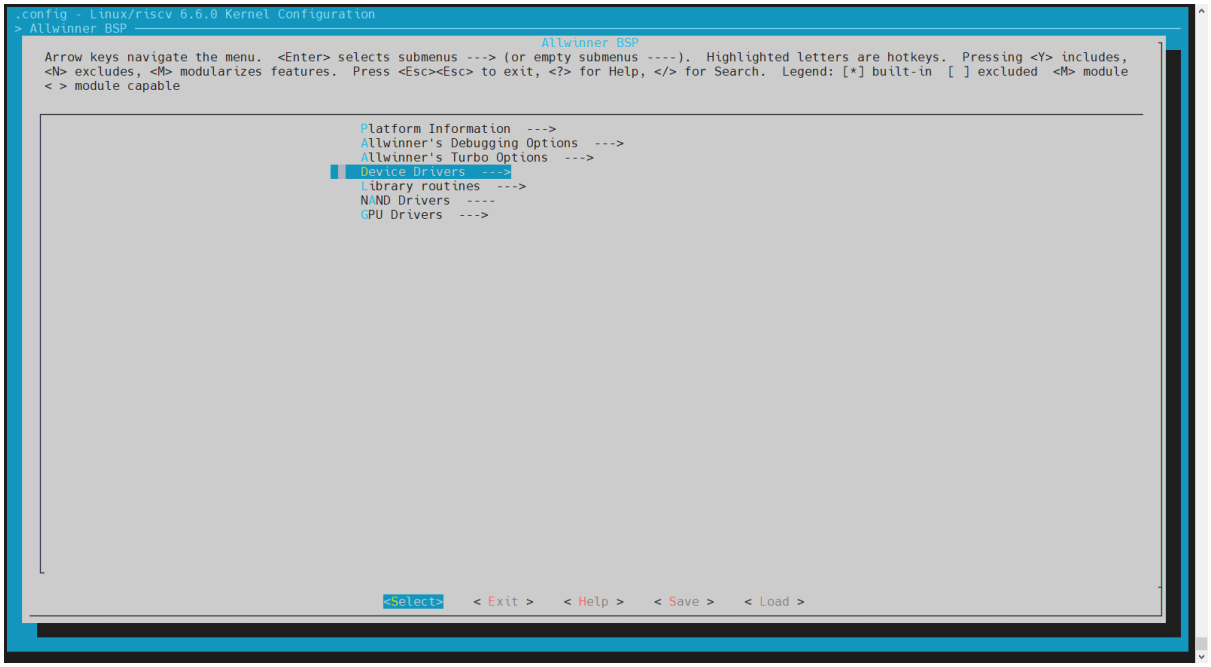

图 2-2: clock-source 配置

接着选择 Timer Drivers 选项, 进入下一级配置, 如图 4 所示:

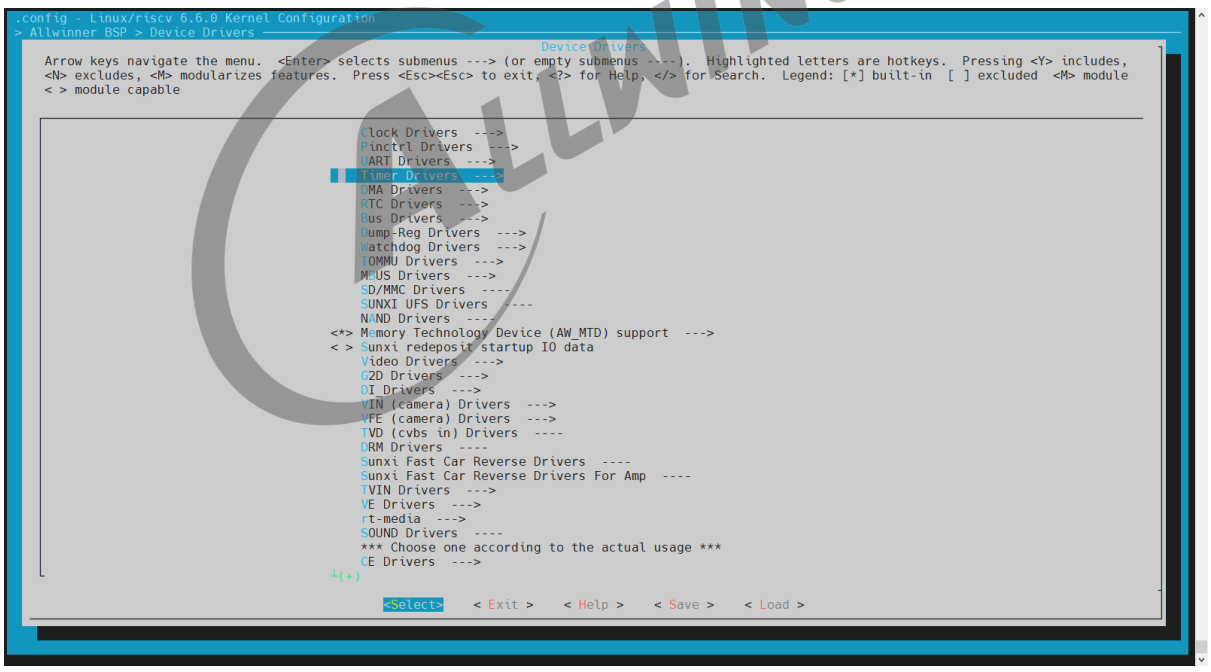

图 2-3: timer 配置选项

具体是选择 sun4i 还是 sun50i 需要结合 ic 设计决定

2.2.2.2 tina SDK

首先运行 source build/envsetup.sh;

第二步运行 lunch 命令选择具体方案

第三步运行 make kernel_menuconfig 命令进入配置页面，参考 BSP SDK 的配置主页面步骤完成配置。

2.3 源码结构介绍

Timer 驱动的源代码位于内核在 SDK/bsp/drivers/timer 目录下：

```
SDK/bsp/drivers/timer

├─── Makefile
├─── Kconfig
├─── timer-sun50i.c    // 使用sun5i 平台的 Timer 控制器驱动代码
├─── timer-sun4i.c    // 使用suni 平台的 Timer 控制器驱动代码
```

2.4 软件框架

Timer 驱动模块内部的功能可划分如下所示：

1. 驱动注册，向 Core 注册 Timer 驱动，驱动注册后会跟 Device Tree 中描述的 Timer 控制器进行匹配，匹配成功后会调用 init 函数，初始化 Timer 控制器；

2.Timer 操作集，提供一套标准的接口集，内核可以通过这些接口控制 Timer 控制器完成计时功能；

3. 中断处理，Timer 控制器在计时结束时会触发中断；

4. 休眠处理，在 timer 被 shutdown 时会保存寄存器；

3 模块设计

3.1 关键数据定义

```
#define TIMER_SYNC_TICKS    3
```

TIMER_SYNC_TICKS 用于同步时钟

```
static u32 regs_backup[ARRAY_SIZE(regs_offset)];
```

用于在 shutdown 时保存寄存器信息

3.2 关键数据结构

regs_offset 结构体

```
/* Registers which needs to be saved and restored before and after sleeping */
static u32 regs_offset[] = {
    TIMER_IRQ_REG,
    TIMER_STA_REG,
    TIMER_CTL_REG(0),
    TIMER_IVL_REG(0),
    TIMER_CVL_REG(0),
    TIMER_IVH_REG(0),
    TIMER_CVH_REG(0),
    TIMER_CTL_REG(1),
    TIMER_IVL_REG(1),
    TIMER_CVL_REG(1),
    TIMER_IVH_REG(1),
    TIMER_CVH_REG(1),
};
```

休眠唤醒时需要保存的寄存器信息

3.3 模块流程设计

3.3.1 初始化流程

```
flowchat
st=>start: 开始
e=>end: 结束
op=>operation: timer of init
op1=>operation: 获取并设置时钟
op2=>operation: 注册timer
op3=>operation: 注册中断

st->op->op1->op2->op3->e
```

3.3.2 中断处理流程

```
flowchat
st=>start: 开始
e=>end: 结束
op=>operation: 清除中断标志位
op1=>operation: 执行中断注册的回调函数


st->op->op1->e
```

3.3.3 休眠/唤醒处理流程

```
flowchat
st=>start: 开始
e=>end: 结束
op=>operation: 获取timer_of
op1=>operation: 保存指定的寄存器信息
op2=>operation: 停止timer计时

st->op->op1->op2->e
```

4 模块接口说明

4.1 内核态接口

4.1.1 sun50i_clkevt_time_setup

- 函数原型：

```
void sun50i_clkevt_time_setup(void __iomem *base, u8 timer, unsigned long delay)
```

- 作用：timer 参数设置
- 参数：
  - 参数 1:Timer 基地址
  - 参数 2:Timer 号
  - 参数 3: 延迟时长
- 返回：无返回值

4.1.2 sun50i_clkevt_time_start

- 函数原型：

```
void sun50i_clkevt_time_start(void __iomem *base, u8 timer, bool periodic)
```

- 作用：开始 timer 计时
- 参数：
  - 参数 1:Timer 基地址
  - 参数 2:Timer 号
  - 参数 3: 是否周期计数
- 返回：无返回值

4.1.3 sun50i_clkevt_time_stop

- 函数原型：

```
static void sun50i_clkevt_time_stop(void __iomem *base, u8 timer)
```

- 作用：停止 timer 计时
- 参数：
  - 参数 1:Timer 基地址
  - 参数 2:Timer 号
- 返回：无返回值

4.2 用户态接口

4.2.1 time

- 函数原型：

```
time_t time(time_t *t);
```

- 作用：返回当前时间点到 Linux 纪元的秒数
- 参数：
- 参数 1:time_t 类型变量
- 返回：
- 大于 0: 成功返回当前时间点到 Linux 纪元的秒数
- -1: 失败

4.2.2 gettimeofday

- 函数原型：

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

- 作用：获取当前时间，包括秒和微秒
- 参数：

- 参数 1：用于存储秒数和微妙
- 参数 2：用于存储时区
- 返回：
- 0: 成功
- -1: 失败

4.2.3 settimeofday

- 函数原型：

```
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

- 作用：设置当前时间戳
- 参数：
- 参数 1：用于存储秒数和微妙
- 参数 2：用于存储时区
- 返回：
- 0: 成功
- -1: 失败

4.2.4 adjtime

- 函数原型：

```
int adjtime(const struct timeval *delta, struct timeval *olddelta);
```

- 作用：调整系统时间
- 参数：
- 参数 1：指定时间调整量
- 参数 2：返回实际调整量
- 返回：
- 0: 成功
- -1: 失败

5 功能开发

5.1 内核态定时器开发

5.1.1 内核态定时器开发流程

步骤 1 创建 hrtimer 定时器结构体

```
static struct hrtimer timer; /* 创建hrtimer定时器 */
```

步骤 2 初始化 hrimter 结构体

```
hrtimer_init(&timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL_HARD);
```

说明

使用 CLOCK_MONOTONIC 时间，HRTIMER_MODE_REL_HARD 表示在硬中断环境下处理

步骤 3 创建超时处理函数

```
timer.function = hrtimer_handler; /* 设置超时处理函数 */
```

说明

到设置的超时时间后，在超时处理函数进行超时后相应的操作

步骤 4 取消定时器

```
hrtimer_cancel(&timer); /* 取消定时器 */
```

5.1.2 内核态定时器开发示例代码

在内核中直接使用 hrtimer 示例代码如下，每 8ms 周期性触发并打印 log。

```
static struct hrtimer timer; /* 创建hrtimer定时器 */

/* 定时器到期处理函数 */
static enum hrtimer_restart hrtimer_handler(struct hrtimer *hrt)
{
    printk("hrtimer_handler");
    hrtimer_forward_now(hrt, 8000000); /* 将超时时间向后移8000000ns=8ms */
    return HRTIMER_RESTART; /* 返回重新启动标志，无需再次调用hrtimer_start */
}

static int __init hrtimer_test_init(void)
{
    /* 初始化hrtimer，使用CLOCK_MONOTONIC时间，HRTIMER_MODE_REL_HARD表示在硬中断环境下处理 */
```

```
    hrtimer_init(&timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL_HARD);
    timer.function = hrtimer_handler; /* 设置超时处理函数 */
    /* 启动定时器 */
    hrtimer_start(&timer, 8000000, HRTIMER_MODE_REL_HARD);
}

static void __exit hrtimer_test_exit(void)
{
    hrtimer_cancel(&timer);  /* 取消定时器 */
}
```

5.2 用户态定时开发

5.2.1 用户态定时器开发流程

步骤 1 创建 timer_t 时间戳变量

```
time_t timestamp;
```

步骤 2 创建 struct timeval 结构体变量用于存储时间

```
struct timeval tv;
```

步骤 3 调用标准接口设置或获取时间

```
settimeofday(&tv, NULL);

gettimeofday(&tv1, NULL);
```

5.2.2 用户态定时器开发示例代码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

int main() {
  /*
  * 假设你有一个时间戳，这里以秒为单位
  * 注意：这里的时间戳应该是自1970年1月1日以来的秒数
  * 你可能需要将时间戳转换为struct timeval需要的微秒精度
  */
  time_t timestamp = 1609459200; /* 示例时间戳，比如2021年1月1日 */
  /* 转换为struct timeval，因为settimeofday需要这个结构体 */
  struct timeval tv;
  struct timeval tv1;
  tv.tv_sec = timestamp;
  tv.tv_usec = 0; /* 微秒部分设置为0 */
  /* 调用settimeofday设置系统时间 */
  if (settimeofday(&tv, NULL) == -1) {
    perror("settimeofday");
```

```
        exit(EXIT_FAILURE);
    }
    printf("System time set successfully\n");  /* 打印成功信息 */

    gettimeofday(&tv1, NULL); /* 获取当前时间 */
    printf("Seconds: %ld\n", tv1.tv_sec); /* 打印秒数 */
    printf("Microseconds: %ld\n", tv1.tv_usec); /* 打印微秒数 */
    printf("System time get successfully\n");  /* 打印成功信息 */

    return 0;
}
```

著作权声明

商标声明

免责声明