Titan-Board SDK

开发文档

Titan Board 开发板是 RT-Thread 推出基于瑞萨 Cortex-M85 与 Cortex-M33 双核架构 R7KA8P1 芯片，为工程师们提供了一个灵活、全面的开发平台，助力开发者在嵌入式物联网领域获得更深层次的体验。

Version 1.0.0 | 中文 | 2025年10月15日

www.rt-thread.org

Copyright (c) 2006-2025, RT-Thread Development Team

目录

- 1. 快速上手
  - 1.1. Titan Board BSP 说明
- 2. 基础篇
  - 2.1. RGB 使用说明
  - 2.2. 蜂鸣器使用说明
  - 2.3. 按键中断使用说明
- 3. 驱动篇
  - 3.1. ADC应用示例说明
  - 3.2. CANFD 应用示例说明
  - 3.3. Ethernet 示例说明
  - 3.4. GPT 使用说明
  - 3.5. HyperRAM 使用说明
  - 3.6. IST8310 示例说明
  - 3.7. RGB LCD 使用说明
  - 3.8. RS485 驱动例程
  - 3.9. RTC 及 alarm 使用例程说明
  - 3.10. SD卡文件系统使用说明
  - 3.11. BMI088 陀螺仪使用说明
  - 3.12. USB-HHID 驱动例程
  - 3.13. USB-PCDC 驱动例程
  - 3.14. WDT 使用说明
  - 3.15. Wi-Fi使用说明
- 4. 组件篇
  - 4.1. Flash 文件系统使用说明
  - 4.2. netutils 示例说明
- 5. 多媒体显示篇
  - 5.1. CEU 摄像头使用说明
  - 5.2. MIPI CSI 摄像头使用说明

- 5.3. LVGL 示例说明
- 6. 多核通信篇
  - 6.1. RPMsg-Lite 使用说明（Core0 Master）
  - 6.2. RPMsg-Lite 使用说明（Core1 Remote）

1. 快速上手

1.1. Titan Board BSP 说明

中文 | English

简介

本文档为 RT-Thread Titan Board 开发板提供 BSP (板级支持包) 说明。通过阅读快速上手章节，开发者可以快速地上手该 BSP，将 RT-Thread 运行在开发板上。

主要内容如下：

- 开发板介绍
- BSP 快速上手指南

开发板介绍

基于瑞萨 RA8P1 开发的 Titan Board 开发板，通过灵活配置软件包和 IDE，对嵌入系统应用程序进行开发。

开发板正面外观如下图：

该开发板常用 **板载资源** 如下：

- MCU：R7KA8P1KFLCAC，高性能 1GHz Arm® Cortex®-M85 与 250MHz Arm® Cortex®-M33 双内核，1MB code MRAM 以及 2MB 带 ECC 的 SRAM。
- 调试接口：板载 DAP-Link 接口
- 板载接口：40Pin 树莓派接口、MIPI DSI/CSI、CEU Camera、RGB LCD接口、2个以太网接口、4个 CAN 电机接口、2个 RS485接口、3路串口、1个 DBUS 接口、1个 USBHS/USBFS 接口、1个 RTC 电池接口。

外设支持

本 BSP 目前对外设的支持情况如下：

片上外设	支持情况	组件	支持情况
UART	支持	LWIP	支持
GPIO	支持	TCP/UDP	支持
HWTIMER	支持	MQTT	支持
IIC	支持	TFTP	支持

片上外设	支持情况	组件	支持情况
WDT	支持	Telnet	支持
RTC	支持	**多核通信**	**支持情况**
ADC	支持	RPmsg-Lite	支持
DAC	支持	**拓展外设**	**支持情况**
SPI	支持	MIPI CSI Camera	支持
RS485	支持	CEU Camera	支持
CANFD	支持	RGB LCD	支持
SDHI	支持	CYW43438 WIFI	支持
USB	支持		
HyperRAM	支持		
HyperFlash	支持		

使用说明

使用说明分为如下两个章节：

- 快速上手

本章节是为刚接触 RT-Thread 的新手准备的使用说明，遵循简单的步骤即可将 RT-Thread 操作系统运行在该开发板上，看到实验效果。

- 进阶使用

本章节是为需要在 RT-Thread 操作系统上使用更多开发板资源的开发者准备的。通过使用 FSP 和 RT-Thread Settings 工具对项目进行配置，可以开启更多板载资源，实现更多高级功能。

快速上手

本 BSP 目前提供 RT-Thread Studio 工程。下面以 RT-Thread Studio 开发环境为例，介绍如何将系统运行起来。

创建工程

- 下载 Titan Board 资源包、GNU_Tools_for_ARM_Embedded_Processors v13.3。

- 新建 RT-Thread 项目

硬件连接

使用 USB 数据线连接开发板到 PC，使用 DAP-Link 接口下载和 DEBUG 程序。

编译下载

查看运行结果

下载程序成功之后，系统会自动运行并打印系统信息。

连接开发板对应串口到 PC , 在终端工具里打开相应的串口（115200-8-1-N），复位设备后，可以看到 RT-Thread 的输出信息。输入 help 命令可查看系统中支持的命令。

```
 \ | /
- RT -     Thread Operating System
```

```
 / | \     5.2.0 build Aug 5 2025 17:24:30
 2006 - 2024 Copyright by RT-Thread team

===================================================
Hello, Titan Board!
===================================================
msh >ps
thread       pri  status      sp     stack size max used left tick
------------ ---  ------- ---------- ----------  ------  ----------
tshell        20  running 0x000000cc 0x00001000    16%   0x0000000a
sys workq     23  suspend 0x000000c4 0x00000800    11%   0x0000000a
tidle0        31  ready   0x000000a4 0x00000100    67%   0x00000011
timer          4  suspend 0x000000a4 0x00000200    38%   0x00000009
main          10  suspend 0x000000dc 0x00000800    18%   0x00000007
msh >
```

应用入口函数

应用层的入口函数在 **src\hal_entry.c** 中 的 `void hal_entry(void)` 。用户编写的源文件可直接放在 src 目录下。

```
void hal_entry(void)
{
    rt_kprintf("\n===================================================
    rt_kprintf("Hello, Titan Board!\n");
    rt_kprintf("===================================================\

    while (1)
    {
        rt_pin_write(LED_PIN_0, PIN_HIGH);
        rt_thread_mdelay(1000);
        rt_pin_write(LED_PIN_0, PIN_LOW);
        rt_thread_mdelay(1000);
    }
}
```

进阶使用

资料及文档

- 开发板官网主页
- 开发板数据手册
- 开发板硬件手册
- 多核开发指南
- Renesas RA8P1 Group

FSP 配置

需要修改瑞萨的 BSP 外设配置或添加新的外设端口，需要用到瑞萨的 RA 可扩展性强的配置软件包 (FSP) 配置工具。请务必按照如下步骤完成配置。配置中有任何问题可到 RT-Thread 社区论坛 中提问。

1. 下载灵活配置软件包 (FSP) | Renesas，请使用 FSP 6.0.0 版本
2. 请参考文档：RA系列使用FSP配置外设驱动。

RT-Thread Settings

在 RT-Thread Settings 中可以对 RT-Thread 的内核、组件、软件包以及 Titan Board 的设备驱动进行配置。

联系人信息

在使用过程中若您有任何的想法和建议，建议您通过以下方式来联系到我们 RT-Thread 社区论坛

贡献代码

如果您对 Titan Board 感兴趣，并且有一些好玩的项目愿意与大家分享的话欢迎给我们贡献代码，您可以参考 如何向 RT-Thread 代码贡献。

2. 基础篇

2.1. RGB 使用说明

中文|English

简介

本例程作为SDK 的第一个例程，也是最简单的例程，类似于程序员接触的第一个程序 Hello World 一样简洁。它的主要功能是让板载的 RGB LED 进行周期性闪烁。

GPIO 简介

GPIO (General Purpose Input/Output) 是 MCU 最常用的外设接口之一，能够在软件控制下配置为 **输入模式** 或 **输出模式**：

- **输入模式**：用于读取外部电平状态，例如按键输入。
- **输出模式**：用于控制外设电平，例如点亮 LED、驱动蜂鸣器。

RA8 系列 GPIO 特点

- **丰富的引脚资源**：每个端口可独立配置为输入/输出/复用功能。
- **多功能复用**：除 GPIO 功能外，同一引脚还可能作为 UART、SPI、I2C 等外设的信号线。
- **电平控制**：输出时可驱动高电平或低电平，部分引脚支持开漏模式。
- **方向控制**：输入/输出可动态切换。

RT-Thread 对 GPIO 的抽象

RT-Thread 提供了 **PIN 设备驱动框架**，通过统一的接口屏蔽底层硬件差异：

- `rt_pin_mode(pin, mode)`：设置引脚工作模式（输入/输出/上拉/下拉等）
- `rt_pin_write(pin, value)`：输出电平（高/低）
- `rt_pin_read(pin)`：读取输入电平

这样开发者不需要直接操作寄存器，而是通过 RT-Thread 的 API 即可完成 GPIO 控制。

在本示例中，LED 引脚被配置为 **输出模式**，软件循环输出高低电平，从而实现 LED 闪烁。

硬件说明

如上图所示，Titan Board 提供三个用户LED，分别为LED_R（RED）、LED_B（BLUE）、LED_G（GREEN），其中 LED_B 对应引脚P012。单片机引脚输出低电平即可点亮LED，输出高电平则会熄灭LED。

LED在开发板中的位置如下图所示：

软件说明

本例程的源码位于 `/project/Titan_basic_blink_led` 。RGB LED 对应的单片机引脚定义及 RGB 变换源码可以通过查阅 `src/hal_entry.c` 中。

```
/* 配置 LED 灯引脚 */
#define LED_PIN_R   BSP_IO_PORT_00_PIN_13
#define LED_PIN_B   BSP_IO_PORT_00_PIN_12
#define LED_PIN_G   BSP_IO_PORT_06_PIN_13

/* 定义 LED 亮灭电平 */
#define LED_ON  (0)
#define LED_OFF (1)

/* 定义 8 组 LED 闪灯表，其顺序为 R B G */
static const rt_uint8_t _blink_tab[][3] =
{
    {LED_OFF, LED_OFF, LED_OFF},
    {LED_ON,  LED_OFF, LED_OFF},
    {LED_OFF, LED_ON,  LED_OFF},
    {LED_OFF, LED_OFF, LED_ON},
    {LED_ON,  LED_OFF, LED_ON},
    {LED_ON,  LED_ON,  LED_OFF},
    {LED_OFF, LED_ON,  LED_ON},
    {LED_ON,  LED_ON,  LED_ON},
};
```

```
void hal_entry(void)
{
    rt_kprintf("\nHello RT-Thread!\n");
    rt_kprintf("==================================================\
    rt_kprintf("This example project is a RGB flicker routine!\n\n"
    rt_kprintf("==================================================\

    unsigned int count = 0;
    unsigned int group_num = sizeof(_blink_tab)/sizeof(_blink_tab[0
    unsigned int group_current;

    /* 设置 RGB 灯引脚为输出模式 */
    rt_pin_mode(LED_PIN_R, PIN_MODE_OUTPUT);
    rt_pin_mode(LED_PIN_G, PIN_MODE_OUTPUT);
    rt_pin_mode(LED_PIN_B, PIN_MODE_OUTPUT);
    rt_pin_write(LED_PIN_R, LED_OFF);
    rt_pin_write(LED_PIN_G, LED_OFF);
    rt_pin_write(LED_PIN_B, LED_OFF);

    do
    {
        /* 获得组编号 */
        group_current = count % group_num;

        /* 控制 RGB 灯 */
        rt_pin_write(LED_PIN_R, _blink_tab[group_current][0]);
        rt_pin_write(LED_PIN_B, _blink_tab[group_current][1]);
        rt_pin_write(LED_PIN_G, _blink_tab[group_current][2]);

        /* 输出 LOG 信息 */
        LOG_D("group: %d | red led [%-3.3s] | | blue led [%-3.3s] |
            group_current,
            _blink_tab[group_current][0] == LED_ON ? "ON" : "OFF",
            _blink_tab[group_current][1] == LED_ON ? "ON" : "OFF",
            _blink_tab[group_current][2] == LED_ON ? "ON" : "OFF");

        count++;

        /* 延时一段时间 */
        rt_thread_mdelay(500);
    }while(count > 0);
}
```

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

按下复位按键重启开发板，观察开发板上 RBG-LED 的实际效果。正常运行后，RGB 会周期性变化。

此时也可以在 PC 端使用终端工具打开开发板的默认配置的串口，设置波特率为 115200N。开发板的运行日志信息即可实时输出出来。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 30 2025 13:29:07
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
====================================================
This example project is an RGB flicker routine!
====================================================
[D/main] group: 0 | red led [OFF] | | blue led [OFF] | | green led
[D/main] group: 1 | red led [ON ] | | blue led [OFF] | | green led
[D/main] group: 2 | red led [OFF] | | blue led [ON ] | | green led
[D/main] group: 3 | red led [OFF] | | blue led [OFF] | | green led
[D/main] group: 4 | red led [ON ] | | blue led [OFF] | | green led
[D/main] group: 5 | red led [ON ] | | blue led [ON ] | | green led
[D/main] group: 6 | red led [OFF] | | blue led [ON ] | | green led
[D/main] group: 7 | red led [ON ] | | blue led [ON ] | | green led
```

2.2. 蜂鸣器使用说明

中文 | English

简介

在我们具体的应用场合中往往都离不开 timer 的使用，本例程主要介绍了如何在 Titan Board 上使用 **GPT (General Purpose Timer)** 产生 PWM 信号，驱动无源蜂鸣器演奏一段旋律。

本示例的旋律选择了经典儿童歌曲 **《两只老虎》**，通过定时器动态调整 PWM 的频率，改变蜂鸣器的发声音调。

通过该示例，用户可以学习：

- RA8 GPT 模块配置与 PWM 输出方法
- RT-Thread 下 PWM 设备驱动框架的使用
- 如何利用 PWM 占空比和频率合成不同音调，进而实现简单音乐播放

PWM / GPT 简介

无源蜂鸣器与PWM

- **无源蜂鸣器** 本质上是一个小型压电/电磁换能器，需要外部提供**一定频率**的交流（方波）以发声。
- **音高**由**PWM 频率**决定；**响度/音色**主要受**占空比**与驱动能力影响。常用 **50% 占空比** 得到基波最强、谐波较低的纯净音。
- 若蜂鸣器额定电流较大（或板载GPIO不直驱），应使用 **NPN/MOSFET** 低边开关 + 二极管（电磁式）做保护；压电式一般无需续流二极管，但仍建议串联 **100–1kΩ** 电阻限流/降噪（依据实际器件与板卡而定）。

PWM 基本概念

- **周期**：一轮高低电平的总时间，记为 `T`。
- **频率**：`f = 1 / T` (单位 Hz)。
- **占空比**：高电平持续时间与周期之比，`D = t_high / T`。
- **电平幅值**：通常为 VCC（如 3.3V）。

用于蜂鸣器播放乐曲时：

- 改变 **频率 f** → 改变音高；
- 改变 **保持时长** → 改变节拍与节奏；
- 微调 **占空比 D**（常在 40–60%）→ 影响响度与音色。

RA8 GPT 概览（用于PWM）

- **GPT（General PWM Timer）**：RA8 系列通用定时器，支持 **PWM 输出**，通常具备 A/B 两路输出（以 FSP/BSP 引脚映射为准）。
- 典型能力：可配置 **计数源/分频**、**周期/比较值**、**边沿对齐/中心对齐** 等；亦支持互补与死区（多用于电机/驱动）。
- 在 **RT-Thread** 下，通过 **PWM 设备驱动** 统一操作 GPT：只需设置 **周期 (period)** 与 **脉宽 (pulse)** 即可产生 PWM。

在 RT-Thread 中设置频率与占空比

`rt_pwm_set(device, channel, period_ns, pulse_ns)` 以纳秒为单位：

- 周期 (ns)：`T_ns = 10^9 / f`
- 高电平时间 (ns)：`t_high = T_ns * D`

例如 `D=50%` 时，`t_high = T_ns / 2`。

> *注意：`period_ns/pulse_ns` 通常为 32 位数值，建议避免过低频率导致周期过大；若需要极低频，可考虑软件定时或分段播放。*

硬件说明

蜂鸣器的引脚为 PA07。

FSP配置

- FSP 配置使能 GPT7 为 PWM 模式：

- 并配置 Pins 使能 GPT7：

RT-Thread Settings 配置

- 在配置中打开 PWM7 使能：

示例工程说明

为了演奏《两只老虎》，我们需要定义简谱对应的频率（单位：Hz）：

音符	频率(Hz)
C4	262
D4	294
E4	330
F4	349
G4	392
A4	440
B4	494
C5	523

> *实际输出频率会受 **GPT时钟/分频/计数步进** 量化影响，驱动会在可达范围内取最近值。若需要更准的音高，可优先挑选**量化误差更小**的频率（或调整系统时钟/分频）。*

本例程的源码位于`/project/Titan_basic_buzzer/src/buzzer.c`：

```
#include <rtthread.h>
#include <rtdevice.h>

#define PWM_DEV_NAME        "pwm7"  /* PWM设备名称 */
#define PWM_DEV_CHANNEL     0       /* PWM通道 */

struct rt_device_pwm *pwm_dev;

typedef struct
{
    uint16_t freq;   // 频率Hz
    uint16_t duration; // 持续时间ms
} note_t;

note_t song[] =
{
    {262,400}, {294,400}, {330,400}, {262,400},  // 1 2 3 1
    {262,400}, {294,400}, {330,400}, {262,400},  // 1 2 3 1
    {330,400}, {349,400}, {392,800},             // 3 4 5
    {330,400}, {349,400}, {392,800},             // 3 4 5
    {392,200}, {440,200}, {392,200}, {349,200}, {330,400}, {262,400
    {392,200}, {440,200}, {392,200}, {349,200}, {330,400}, {262,400
    {262,400}, {196,400}, {262,400}, {0,400},    // 1(低) 7(低) 1 高
    {262,400}, {196,400}, {262,400}, {0,400},    // 1 7 1 休止
};

static int buzzer_test(void)
{
    pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
    if (!pwm_dev)
    {
        rt_kprintf("Cannot find PWM device %s\n", PWM_DEV_NAME);
        return -1;
    }

    for (size_t i = 0; i < sizeof(song)/sizeof(song[0]); i++)
    {
        if (song[i].freq == 0)
        {
            rt_pwm_disable(pwm_dev, PWM_DEV_CHANNEL);
        }
        else
        {
            uint32_t period_ns = 1000000000 / song[i].freq;  // ns
            uint32_t pulse_ns  = period_ns / 2;              // 50%

            rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period_ns, pulse_n
            rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL);
        }
```

```
        rt_thread_mdelay(song[i].duration);
    }

    rt_pwm_disable(pwm_dev, PWM_DEV_CHANNEL);
    return 0;
}
MSH_CMD_EXPORT(buzzer_test, Play song on buzzer);
```

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

在串口终端输入 `buzzer_sample` 即可看到效果。蜂鸣器将播放 **“两只老虎”**。

2.3. 按键中断使用说明

中文 | English

简介

本例程主要功能是通过板载的按键 KEY 实现外部中断，当 USER_KEY 被按下时，改变 LED_G(Green) 的亮灭状态。

RA8 系列 MCU 中断体系概述

具有如下中断特性：

1. **NVIC（嵌套向量中断控制器）**

- 支持 **中断嵌套**：高优先级中断可打断低优先级中断
- 支持 **优先级分组**（Preemption Priority 与 Subpriority）
- IRQn 映射到中断向量表，由 NVIC 调用对应 ISR

2. **GPIO 外部中断 (EXTI)**

- 每个 GPIO 引脚可配置为外部中断输入
- 支持 **上升沿、下降沿或双沿触发**
- 中断通道由 **ICU (Interrupt Controller Unit)** 管理
- 可以通过 Renesas FSP 或底层寄存器配置
- 中断响应延迟低，适合按键、传感器等事件驱动

3. **RT-Thread PIN 驱动模型**

- GPIO 在 RT-Thread 中被封装为 **PIN 设备**
- 提供统一接口：
  - `rt_pin_mode(pin, mode)`：配置输入/输出模式
  - `rt_pin_read(pin)` / `rt_pin_write(pin, value)`：读写 GPIO
  - `rt_pin_attach_irq(pin, mode, callback, args)`：注册中断回调
  - `rt_pin_irq_enable(pin, enable)`：使能/禁用中断

> *使用 RT-Thread PIN 驱动，无需直接操作 NVIC 或 MCU 寄存器即可实现中断响应。*

硬件说明

如下图所示， USER_KEY 引脚连接单片机 P110 引脚, KEY 按键按下为低电平，松开为高电平。

KEY 在开发板中的位置如下图所示：

FSP配置

- 首先下载官方FSP代码生成工具：

https://github.com/renesas/fsp/releases/download/v6.0.0/setup_fsp_v6_0_0_rasc_v2025-04.1.exe

- 安装成功之后我们双击 eclipse 目录下的 `rasc.exe`，并打开 `Titan_basic_key_irq` 工程目录下的配置文件 `configuration.xml`：
- 下面我们新增两个Stack：New Stack->Input->External IRQ(r_icu)：

- 接着我们需要在引脚配置那开启IRQ功能，根据下图选中我们要使能的中断引脚：USER_KEY(IRQ20)：

- 回到Stacks界面，设置 IRQ20，配置对应的中断名称、通道号以及中断回调函数：

示例代码说明

本例程的源码位于 `/project/Titan_basic_key_irq` 。

USER_KEY 对应的单片机引脚定义如下:

```
/* 配置 key irq 引脚 */
#define KEY_PIN     BSP_IO_PORT_01_PIN_10 /* Onboard KEY pins */
```

LED灯的单片机引脚定义如下:

```
/* 配置 LED 灯引脚 */
#define LED_PIN_B   BSP_IO_PORT_00_PIN_12 /* Onboard LED pins */
#define LED_PIN_G   BSP_IO_PORT_06_PIN_13 /* Onboard LED pins */
```

按键中断的源代码位于 `/project/Titan_basic_key_irq/src/hal_entry.c` 中，当按下 USER_KEY 按键，会改变绿色 LED 的亮灭状态。

```
#define LED_PIN_B   BSP_IO_PORT_00_PIN_12 /* Onboard LED pins */
#define LED_PIN_G   BSP_IO_PORT_06_PIN_13 /* Onboard LED pins */
#define KEY_PIN     BSP_IO_PORT_01_PIN_10 /* Onboard KEY pins */

volatile rt_bool_t flag = 0;

void key_callback(void)
{
    if (flag)
        rt_pin_write(LED_PIN_G, PIN_HIGH);
    else
        rt_pin_write(LED_PIN_G, PIN_LOW);

    flag = flag ? RT_FALSE : RT_TRUE;
}

void hal_entry(void)
{
    rt_kprintf("\nHello RT-Thread!\n");
    rt_kprintf("====================================================\
    rt_kprintf("This example project is a key routine!\n");
    rt_kprintf("====================================================\

    LOG_I("Tips:");
    LOG_I("You can press the button to run the test case for button
    LOG_I("Use the buttons to control the on/off state of the green

    rt_pin_mode(KEY_PIN, PIN_MODE_INPUT_PULLUP);
    rt_pin_attach_irq(KEY_PIN, PIN_IRQ_MODE_FALLING, key_callback,
    rt_pin_irq_enable(KEY_PIN, PIN_IRQ_ENABLE);

    while (1)
```

```
    {
        rt_pin_write(LED_PIN_B, PIN_HIGH);
        rt_thread_mdelay(1000);
        rt_pin_write(LED_PIN_B, PIN_LOW);
        rt_thread_mdelay(1000);
    }
}
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

按下复位按键重启开发板，初始状态下的 LED_G 处于灭灯状态，当按下 USER_KEY 时，LED_G(Green) 的亮灭状态会改变。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 28 2025 11:01:50
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
===================================================
This example project is a key routine!
===================================================
[I/key] Tips:
[I/key] You can press the button to run the test case for button interruption.
[I/key] Use the buttons to control the on/off state of the green LED.
msh >
```

3. 驱动篇

3.1. ADC应用示例说明

中文 | **English**

简介

本示例展示了如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 ADC（Analog-to-Digital Converter）**，结合 **RT-Thread ADC 驱动框架**进行模拟信号采集和处理。

主要功能包括：

- 初始化 ADC 硬件模块
- 配置 ADC 通道、采样时间和触发方式
- 通过 RT-Thread ADC 驱动接口读取模拟信号
- 支持单次采样、连续采样及硬件触发

RA8P1 ADC 特性

1. ADC概述

ADC（Analog-to-Digital Converter） 是将连续的模拟信号转换为离散的数字信号的器件或模块，是现代数字控制系统、信号处理和测量系统中的核心部件。

- **功能**：把电压、电流等连续信号转换为数字值，以便微控制器（MCU）、DSP 或 FPGA 进行处理。
- **重要指标**：
- **分辨率（Resolution）**：ADC输出的数字位数，表示可区分的电平数。RA8P1为**16位**，即 2^16 = 65536 个不同电平。
- **采样率（Sampling Rate）**：ADC每秒采样次数，影响可捕捉的信号频率范围。
- **输入范围（Input Range）**：ADC能处理的模拟电压范围。
- **精度（Accuracy）**：表示ADC输出与实际输入信号的接近程度，受噪声、非线性、偏移误差影响。

2. ADC工作原理

ADC通常分为几个阶段：

1. **采样与保持（Sample & Hold, S/H）**

- 将连续变化的模拟信号在采样瞬间固定，保证后续转换过程中信号不变。
- RA8系列ADC支持 **采样保持时间可调**，可优化高阻抗信号采样。

2. **量化（Quantization）**

- 将模拟信号划分为离散电平，每个电平对应一个数字编码。
- 16位ADC将输入电压范围分为 65536 个电平，量化精度可以表示为：ΔV = VREF / 65536。

3. **编码（Encoding）**

- 将量化后的电平转换成二进制代码输出。
- RA8P1 ADC支持 **右对齐/左对齐数据输出**，便于不同应用场景读取。

3. ADC类型与RA8P1特性

该 MCU 包含两组 **噪声整形型 SAR 型 A/D 转换器（ADC16H）**，其为 **SAR 型与 Δ-Σ 调制型特性混合的架构**。A/D 转换器单元 0（ADC0）最多可选择 15 路模拟输入。A/D 转换器单元 1（ADC1）最多可选择 15 路模拟输入。温度传感器、内部参考电压、VBATT 1/6 电压监测输出以及 D/A 转换器的输出均可由 A/D 转换器单元 0 或单元 1 进行 A/D 转换。A/D 转换数据可选择 16 位、14 位、12 位和 10 位的数据格式。

ADC16H 具有以下特性：

● 分辨率：最高 16 位

● 快速转换：最高 6.25 Msps（每通道 0.16 μs）（当 A/D 转换时钟 ADCLK = 50 MHz）

● 输入通道：最多 23 路模拟输入通道

● 支持单端输入或差分输入

● 自校准功能

● 内置通道专用采样保持电路（S&H）

> ***SAR ADC工作流程：***
>
> *1. 采样保持输入电压*
>
> *2. 逐步比较输入与DAC输出*
>
> *3. 根据比较结果调整二进制码*

> *4. 输出数字值*

4. ADC关键参数解释

1. **分辨率（Resolution）**

- RA8P1 16位ADC：理论最小可分辨电压ΔV = VREF / 65536。
- 举例：VREF=3.3V，则 ΔV ≈ 0.00005 V ≈ 50 μV。

2. **采样时间（Sampling Time）**

- 决定输入电压稳定性与ADC误差。
- 高阻抗信号需要较长采样时间，否则可能出现采样误差。

3. **线性度（Linearity）**

- **INL（Integral Nonlinearity）**：累积误差，理想直线偏离实际值
- **DNL（Differential Nonlinearity）**：相邻码间的间隔误差

4. **噪声与精度**

- 系统噪声会影响低电平分辨率，RA8P1 ADC在16位模式下实际有效位通常略低于16位（比如15位有效位）。

5. **输入阻抗**

- 高输入阻抗信号可直接采样，低阻抗或高速信号需要缓冲电路。

5. RA8P1 ADC的典型应用

- **工业测量**：温度、压力、流量传感器采集
- **电机控制**：电流、电压采样，实现闭环控制
- **信号处理**：音频采集、振动监测
- **数据记录**：多通道高速采样，存储或发送到上位机

硬件说明

如下面原理图所示，Titan Board 上留有 4 个 ADC 通道接口，分别为 adc0的通道0、1、2、3。

FSP 配置

- 第一步：打开FSP导入xml配置文件；（或者直接点击 RT-Thread Studio 的 FSP 链接文件）；
- 第二步：新建 r_adc Stack 配置 adc 设备以及所用通道;

Threads
New Thread
Remove
HAL/Common
g_ioport I/O Port (r_ioport)
g_uart8 UART (r_sci_b_uart)
g_adc0 ADC Driver on r_adc_b
Objects
New Object >
Remove
g_adc0 ADC Driver on r_adc_b Stacks
g_adc0 ADC Driver on r_adc_b
Summary
BSP
Clocks
Pins
Interrupts
Event Links
Linker Sections
Stacks
Components
问题
控制台
属性
Smart Browser
Smart Manual
g_adc0 ADC Driver on r_adc_b
Settings
API Info
属性
值
Common
Parameter Checking
Default (BSP)
Module g_adc0 ADC Driver on r_adc_b
General
Operation
ADC 0
Conversion Method
SAR Mode
Scan Mode
Continuous Scan
扫描模式为持续扫描
ADC 1
ADC Successive Approximation Time
Synchronous Operation
Calibration
Sampling State Table
Name
g_adc0
Clock Configuration
Divider
Div /1
Source
ADCCLK
Interrupts
Digital Filter
Sample and Hold

*[Titan_driver_adc] FSP Configuration
Stacks Configuration
Generate Project Content
Threads
New Thread
Remove
g_ioport I/O Port (r_ioport)
g_uart8 UART (r_sci_b_uart)
g_adc0 ADC Driver on r_adc_b
Objects
New Object >
Remove
g_adc0 ADC Driver on r_adc_b Stacks
New Stack >
Extend Stack >
Remove
g_adc0 ADC Driver on r_adc_b
Summary
BSP
Clocks
Pins
Interrupts
Event Links
Stacks
Components
Properties
Problems
Console
g_adc0 ADC Driver on r_adc_b
Settings
API Info
Property
Value
Virtual Channels
Virtual Channel 0
Scan Group
Scan Group 0
Channel Select
ADC Channel 0
配置通道0
Sampling State Table ID
Sampling State Entry 0
Channel Gain Table
Disabled
Channel Offset Table
Disabled
Add/Average Mode
Disabled
Add/Average Count
1-time conversion (Normal Conversion)
Limit Clip Table Id
Disabled
Conversion Data Format Select
16-bit Data Format
数据格式16bit
Digital Filter Selection
Disabled
Input Mode Selection
Single-ended Input
Data Sign Selection
Signed
Virtual Channel 1
Scan Group
Scan Group 0
Channel Select
ADC Channel 1
配置通道1
Sampling State Table ID
Sampling State Entry 0
Channel Gain Table
Disabled
Channel Offset Table
Disabled
Add/Average Mode
Disabled
Add/Average Count
1-time conversion (Normal Conversion)
Limit Clip Table Id
Disabled
Conversion Data Format Select
16-bit Data Format
数据格式16bit
Digital Filter Selection
Disabled
Input Mode Selection
Single-ended Input
Data Sign Selection
Signed
Virtual Channel 2
Virtual Channel 3

- 第三步：保存并点击 Generate Project；生成的代码保存到 hal_data.c 中；

RT-Thread Settings 配置

- 使能 ADC0：

工程实例说明

ADC 的源代码位于 `/project/Titan_driver_adc/src/hal_entry.c` 中，使用的宏定义如下所示：

```
#define DBG_TAG             "adc"
#define DBG_LVL             DBG_LOG
#include <rtdbg.h>
#define ADC_DEV_NAME        "adc0"
#define ADC_DEV_CHANNEL     1
#define REFER_VOLTAGE       330
#define CONVERT_BITS        (1 << 16)

#define LED_PIN             BSP_IO_PORT_00_PIN_12 /* Onboard LED pins */
```

具体功能为每隔 1000ms 对 ADC0 的通道 1 采集一次模拟电压并进行一次转化，代码如下：

```
static int adc_vol_sample()
{
    rt_adc_device_t adc_dev;
    rt_uint32_t value, vol;
    rt_err_t ret = RT_EOK;

    adc_dev = (rt_adc_device_t)rt_device_find(ADC_DEV_NAME);
    if (adc_dev == RT_NULL)
    {
        rt_kprintf("adc sample run failed! can't find %s device!\n"
        return RT_ERROR;
    }

    ret = rt_adc_enable(adc_dev, ADC_DEV_CHANNEL);

    while(1)
    {
        value = rt_adc_read(adc_dev, ADC_DEV_CHANNEL);
        rt_kprintf("the value is :%d \n", value);

        vol = value * REFER_VOLTAGE / CONVERT_BITS;
        rt_kprintf("the voltage is :%d.%02d \n", vol / 100, vol % 1

        rt_thread_mdelay(1000);
    }

    ret = rt_adc_disable(adc_dev, ADC_DEV_CHANNEL);

    return ret;
}
```

示例中 While 循环每隔 1000ms 调用一次 adc_vol_sample;

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

使用 adc0 的 1通道采集 1.8v 电压效果如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 25 2025 14:33:42
 2006 - 2024 Copyright by RT-Thread team
[D/drv.adc] adc0 init success

Hello RT-Thread!
====================================================
This example project is an driver adc routine!
====================================================
[I/adc] Tips:
[I/adc] You can run adc testcase by executing the instruction: 'adc_sample'
[I/adc] And apply a voltage ranging from 0 to 3.3V to the P001 pin.
msh >adc_sample
the value is :0
the voltage is :0.00
msh >the value is :22544
the voltage is :1.13
the value is :22452
the voltage is :1.13
the value is :12320
the voltage is :0.62
the value is :12068
the voltage is :0.60
the value is :35516
the voltage is :1.78
the value is :35504
the voltage is :1.78
the value is :35516
the voltage is :1.78
the value is :35504
the voltage is :1.78
the value is :35480
the voltage is :1.78
the value is :35480
the voltage is :1.78
```

3.2. CANFD 应用示例说明

中文 | English

简介

本示例展示了如何在 **Titan Board** 上使用 **CANFD（CAN with Flexible Data Rate）接口**，结合 **RT-Thread CAN 驱动框架**进行高速车载或工业通信。

主要功能包括：

- 初始化 CANFD 硬件接口
- 配置波特率、数据帧和时序参数
- 发送和接收标准帧/扩展帧
- 使用 RT-Thread CAN API 实现设备统一管理
- 支持中断、FIFO、DMA 高速数据传输

CAN 与 CAN FD 协议详解

1. CAN 协议概述

CAN（Controller Area Network） 是一种多主机串行通信协议，由 Bosch 在 1980 年代为汽车电子设计。其主要特点是高可靠性、实时性强，适用于汽车、工业自动化、智能设备等领域。

- **数据传输方式**：基于差分信号的总线通信，使用 CAN_H 和 CAN_L 两根线。
- **通信模式**：多主机，支持 **非破坏性总线仲裁**，即优先级高的报文可以先发送。
- **物理层标准**：ISO 11898-2（高速 CAN）、ISO 11898-3（低速容错 CAN）。

2. CAN 协议帧格式

CAN 的数据传输以 **帧（Frame）** 为单位，主要分为以下几种帧类型：

1. **数据帧（Data Frame）**：传输数据
2. **远程帧（Remote Frame）**：请求数据
3. **错误帧（Error Frame）**：用于报告总线错误
4. **过载帧（Overload Frame）**：用于间隔时间控制

2.1 标准帧（CAN 2.0A）

- **帧标识符（ID）**：11 位
- **数据长度（DLC）**：0~8 字节
- **帧结构**：

字段	长度	描述
SOF	1 bit	帧起始位
ID	11 bit	报文标识符
RTR	1 bit	数据帧/远程帧标志
IDE	1 bit	标准/扩展帧标志
r0	1 bit	保留位
DLC	4 bit	数据长度码
Data	0~8 bytes	数据域
CRC	15 bit + 1 bit	CRC 校验
ACK	2 bit	应答位
EOF	7 bit	帧结束

2.2 扩展帧（CAN 2.0B）

- **帧标识符（ID）**：29 位
- 其它字段与标准帧类似，只是 ID 扩展了 18 位。

3. CAN 协议特性

1. **非破坏性总线仲裁**

- 利用 ID 的优先级，ID 越小优先级越高。
- 总线冲突时，低优先级报文自动停止发送，不影响高优先级报文。

2. **差分信号传输**

- CAN_H - CAN_L 差分传输，抗干扰能力强。

3. **速度与长度**

- 最高 1 Mbps（高速 CAN），传输距离与速率相关：

- 1 Mbps → 最大 40 m
- 500 kbps → 最大 100 m
- 125 kbps → 最大 500 m

4. **错误检测与容错**

- CRC 校验
- 位填充（Bit Stuffing）
- ACK 检查
- 错误计数器管理，节点可自动离线

4. CAN FD 协议概述

CAN FD（Flexible Data-rate） 是 Bosch 在 2012 年推出的 CAN 协议扩展标准（ISO 11898-1:2015），解决传统 CAN 带宽不足和数据长度受限问题。

- **主要改进**：

1. 数据长度扩展：从 8 字节提升到 64 字节
2. 可变数据速率：仲裁段保持原速率，数据段可以提高速率（通常 2~8 倍）

5. CAN FD 帧格式

5.1 CAN FD 数据帧结构

字段	描述
SOF	帧起始位
ID	11/29 位标识符
RTR	数据帧标志（远程帧不再使用）
IDE	扩展帧标志
FDF	CAN FD 数据帧标志
BRS	位速率切换（Bit Rate Switch）标志
ESI	错误状态指示
DLC	数据长度码（0~64 字节）

字段	描述
Data	数据字段（0~64 字节）
CRC	CRC 校验（17/21 bit）
ACK	应答位
EOF	帧结束位

- **BRS = 1**：数据段以更高速率传输
- **ESI = 1**：发送节点处于错误被动状态

6. CAN FD 特性

1. **更大数据长度**：最大 64 字节/帧
2. **更高数据速率**：数据段可提高至传统 CAN 速率的 2~8 倍
3. **兼容性**：CAN FD 节点可以与传统 CAN 节点共存，但 FD 功能需硬件支持
4. **错误处理增强**：保持 CAN 错误机制，同时增加 CRC 位数以适应更长数据

7. CAN 与 CAN FD 对比

特性	CAN	CAN FD
最大数据长度	8 字节	64 字节
数据速率	最高 1 Mbps	仲裁段 1 Mbps，数据段可 2~8 Mbps
帧格式	标准/扩展	标准/扩展，增加 FDF/BRS/ESI
兼容性	所有节点均支持	与传统 CAN 向下兼容，需要 FD 节点支持
CRC 校验	15-bit	17/21-bit
使用场景	车辆 ECU、工业控制	高速数据采集、摄像头、ADAS 系统

RA8 系列 CANFD 特性

RA8 系列 MCU 集成高性能 **CANFD 控制器**，支持 ISO CAN 2.0A/B 及 CANFD 协议，具备高速通信、灵活帧长度和硬件 FIFO 功能。

1. 基本参数

1. **协议支持**

- ISO CAN 2.0A/B（经典 CAN）
- CAN FD（Flexible Data Rate）
- 支持标准帧（11 位 ID）和扩展帧（29 位 ID）

2. **数据帧长度**

- CAN：0–8 字节
- CAN FD：0–64 字节

3. **通信速率**

- CAN：最高 1 Mbps
- CAN FD：数据段最高可达 8 Mbps（取决 MCU 时钟和 PHY）

4. **硬件 FIFO**

- RX/TX 独立 FIFO
- 支持多帧缓冲，提高数据吞吐量
- FIFO 触发中断，可结合 RT-Thread ISR 处理

2. 硬件特性

1. **控制器架构**

- 包含独立 RX/TX FIFO
- 硬件滤波器（标准 ID / 扩展 ID / 组合滤波）
- 自动重发、错误检测和总线状态监控

2. **时序与波特率**

- 支持独立 **同步段、时间段 1/2、采样点** 配置
- 支持 CAN FD 异步波特率（数据段波特率可不同于仲裁段）
- 提供自动比特率计算工具

3. **中断机制**

- 传输完成中断（TX）
- 接收帧中断（RX FIFO）
- 错误警告/被动中断
- 总线离线/恢复中断

4. **DMA 支持**

- 可与 MCU DMAC 协作，实现高速数据传输
- TX/RX FIFO 自动写入/读取内存
- 减少 CPU 占用

5. **错误处理与总线监控**

- 错误计数器（TEC / REC）
- 总线状态机（Error Active / Error Passive / Bus Off）
- 自动恢复机制

3. 帧格式

1. **经典 CAN 帧**

- 标准帧：11 位 ID，0–8 字节数据
- 扩展帧：29 位 ID，0–8 字节数据

2. **CAN FD 帧**

- 支持 0–64 字节数据长度
- 支持快速数据段（Higher Data Rate）
- 帧结构增加 **BRS（Bit Rate Switch）** 位用于数据段加速

3. **帧标记与控制位**

- RTR（Remote Transmission Request）
- IDE（Identifier Extension）
- FDF（CAN FD Frame）
- ESI（Error State Indicator）

4. 应用场景

- 汽车 ECU 通信
- 工业控制网络（如 CANopen、J1939）
- 实时传感器数据采集
- 高速数据日志记录和分布式控制

硬件说明

FSP配置

- 新建两个 r_canfdlite Stack：

- 配置 CANFD0 stack：

- 配置 CANFD0 引脚：

- 配置 CANFD1 stack：

- 配置 CANFD1 引脚：

RT-Thread Settings 配置

- 使能 CANFD0 和 CANFD1：

工程示例说明

工程通过 canfd0 发送报文 canfd1 接收报文并将其使用串口打印。

发送代码示例：

```
static void can_tx_thread(void *parameter)
{
    struct rt_can_msg msg = {0};
    rt_size_t  size;
    msg.id = 0x78;              /* ID 为 0x78 */
    msg.ide = RT_CAN_STDID;     /* 标准格式 */
    msg.rtr = RT_CAN_DTR;       /* 数据帧 */
    msg.len = 8;                /* 数据长度为 8 */
    /* 待发送的 8 字节数据 */
    rt_memset(msg.data,0,sizeof(msg.data));
    while(1)
    {
        /* 发送一帧 CAN 数据 */
        for(int i = 0; i < 8; i++)
        {
            msg.data[i] = i;
        }
        size = rt_device_write(can0_dev, 0, &msg, sizeof(msg));
        if (size == 0)
        {
            rt_kprintf("can dev write data failed!\n");
        }
        rt_thread_delay(1000);
    }
}

int can_send(int argc, char *argv[])
{
    rt_err_t res;
```

```
    rt_thread_t thread;
    char can_name[RT_NAME_MAX];

    if (argc == 2)
    {
        rt_strncpy(can_name, argv[1], RT_NAME_MAX);
    }
    else
    {
        rt_strncpy(can_name, CAN0_DEV_NAME, RT_NAME_MAX);
    }
    /* 查找 CAN 设备 */
    can0_dev = rt_device_find(can_name);
    if (!can0_dev)
    {
        rt_kprintf("find %s failed!\n", can_name);
        return RT_ERROR;
    }

    /* 以中断接收及发送方式打开 CAN 设备 */
    res = rt_device_open(can0_dev, RT_DEVICE_FLAG_INT_TX | RT_DEVIC
    RT_ASSERT(res == RT_EOK);
    /* 创建数据接收线程 */
    thread = rt_thread_create("can0_tx", can_tx_thread, RT_NULL, 10
    if (thread != RT_NULL)
    {
        rt_thread_startup(thread);
    }
    else
    {
        rt_kprintf("create can_rx thread failed!\n");
    }

    return res;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(can_send, can device sample);
```

接收代码示例;

```
/* 接收数据回调函数 */
static rt_err_t can_rx_call(rt_device_t dev, rt_size_t size)
{
    /* CAN 接收到数据后产生中断，调用此回调函数，然后发送接收信号量 */
    rt_sem_release(&rx_sem);

    return RT_EOK;
}

static void can_rx_thread(void *parameter)
```

```
{
    struct rt_can_msg rxmsg = {0};

    /* 设置接收回调函数 */
    rt_device_set_rx_indicate(can1_dev, can_rx_call);

    while (1)
    {
        /* hdr 值为 - 1，表示直接从 uselist 链表读取数据 */
        rxmsg.hdr_index = -1;
        /* 阻塞等待接收信号量 */
        rt_sem_take(&rx_sem, RT_WAITING_FOREVER);
        /* 从 CAN 读取一帧数据 */
        rt_device_read(can1_dev, 0, &rxmsg, sizeof(rxmsg));
        /* 打印数据 ID 及内容 */
        rt_kprintf("ID:%x\tmessege:", rxmsg.id);
        for(int i = 0; i < 8; i++)
        {
            rt_kprintf("%d",  rxmsg.data[i]);
        }
        rt_kprintf("\n");
    }
}

int can_receive(int argc, char *argv[])
{
    rt_err_t res;
    rt_thread_t thread;
    char can_name[RT_NAME_MAX];

    if (argc == 2)
    {
        rt_strncpy(can_name, argv[1], RT_NAME_MAX);
    }
    else
    {
        rt_strncpy(can_name, CAN1_DEV_NAME, RT_NAME_MAX);
    }
    /* 查找 CAN 设备 */
    can1_dev = rt_device_find(can_name);
    if (!can1_dev)
    {
        rt_kprintf("find %s failed!\n", can_name);
        return RT_ERROR;
    }

    /* 初始化 CAN 接收信号量 */
    rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);

    /* 以中断接收及发送方式打开 CAN 设备 */
    res = rt_device_open(can1_dev, RT_DEVICE_FLAG_INT_TX | RT_DEVIC
    RT_ASSERT(res == RT_EOK);
```

```
    /* 创建数据接收线程 */
    thread = rt_thread_create("can1_rx", can_rx_thread, RT_NULL, 10
    if (thread != RT_NULL)
    {
        rt_thread_startup(thread);
    }
    else
    {
        rt_kprintf("create can_rx thread failed!\n");
    }
    return res;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(can_receive, can device sample);
```

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

将 CAN0 与 CAN1 对接，分别输入 `can_send` 和 `can_receive` 命令进行回环测试。

3.3. Ethernet 示例说明

中文 | English

简介

本示例展示了如何在 **Titan Board** 上使用 **以太网接口**，结合 **RT-Thread Ethernet 驱动框架** 实现网络通信功能。

主要功能包括：

- 初始化 RA8 系列以太网硬件
- 配置 IP 地址、子网掩码和网关
- 发送和接收以太网帧
- 集成 RT-Thread `netdev` 框架，实现统一网络设备管理
- 支持 DMA 和中断，实现高速数据传输

以太网（Ethernet）简介

1. 概述

以太网（Ethernet） 是最广泛使用的局域网（LAN）技术，由 Xerox PARC 在 1970 年代提出，后由 IEEE 802.3 标准化。以太网具有以下特点：

- **数据传输方式**：基于 **帧（Frame）** 的分组交换，物理上可通过双绞线、光纤或无线介质传输。
- **拓扑结构**：传统以太网为总线或星型拓扑，现代以太网主要采用星型和树型。
- **协议层次**：属于 OSI 模型的数据链路层（第 2 层）及物理层（第 1 层）技术。

2. 以太网帧结构

以太网使用 **帧（Frame）** 作为数据传输单位。以太网帧由以下字段组成：

字段	长度	描述
前导码（Preamble）	7 字节	用于帧同步
帧开始定界符（SFD）	1 字节	帧起始标志，值为 10101011

字段	长度	描述
目的 MAC 地址	6 字节	接收方硬件地址
源 MAC 地址	6 字节	发送方硬件地址
类型/长度	2 字节	上层协议类型或帧长度
数据载荷	46~1500 字节	上层数据（如 IP 包）
CRC 校验	4 字节	循环冗余校验，保证帧完整性

> ***最小帧长度：** 64 字节*
> ***最大帧长度：** 1518 字节（不含 VLAN 标记时）*

3. 以太网物理层与速率

以太网有多种速率和物理介质标准，主要分为以下几类：

标准	速率	介质	特点
10BASE-T	10 Mbps	双绞线	最初以太网标准
100BASE-TX	100 Mbps	双绞线	快速以太网
1000BASE-T	1 Gbps	双绞线	千兆以太网
10GBASE-T	10 Gbps	双绞线	10G 以太网
10GBASE-SR	10 Gbps	光纤	短距离光纤传输
40GBASE / 100GBASE	40/100 Gbps	光纤	数据中心高速以太网

- **BASE-T**：基于双绞线传输
- **BASE-SX / LX / SR**：基于光纤传输

4. 以太网访问控制机制

以太网使用 **CSMA/CD（Carrier Sense Multiple Access with Collision Detection）** 机制：

1. **Carrier Sense（载波监听）**：发送前监听信道是否空闲
2. **Multiple Access（多路访问）**：所有节点共享同一信道
3. **Collision Detection（碰撞检测）**：若同时发送发生冲突，节点停止发送并在随机延迟后重传

> *注意：现代交换机以太网大多为全双工（Full Duplex），不再使用 CSMA/CD。*

5. VLAN 与以太网扩展

- **VLAN（Virtual LAN）**：通过 IEEE 802.1Q 标准，实现逻辑网络划分，提高网络安全和管理性。
- **VLAN 标签帧结构**：

字段	长度	描述
TPID	2 字节	以太网类型标识 0x8100
TCI	2 字节	VLAN ID 与优先级
数据载荷	46~1500 字节	上层数据
CRC	4 字节	校验

- **QoS 支持**：VLAN 标签中的 PCP 字段可用于优先级标记。

6. 以太网帧类型

1. **单播（Unicast）**：目标 MAC 地址唯一
2. **广播（Broadcast）**：目标 MAC 地址为 FF:FF:FF:FF:FF:FF
3. **组播（Multicast）**：目标 MAC 地址对应一组节点

7. 以太网发展趋势

1. **高速化**：从 10 Mbps → 100 Mbps → 1 Gbps → 10/40/100 Gbps
2. **工业以太网**：支持实时控制协议（EtherCAT、Profinet、TSN）
3. **以太网供电（PoE）**：通过网线同时传输数据和电力

4. **全双工交换机网络**：消除碰撞，提高带宽利用率
5. **时间敏感网络（TSN）**：为工业自动化和汽车网络提供确定性时延

RA8 系列以太网特性

RA8 系列 MCU（如 RA8P1）集成高性能 **以太网 MAC**，支持 RGMII、RMII 和 MII 接口，具备稳定可靠的高速网络通信能力。MAC 可配合外部 PHY 使用，并兼容 LwIP TCP/IP 协议栈。

1. 网络接口特性

1. **接口类型**

- **RMII（Reduced Media Independent Interface）**：节省引脚，支持 10/100 Mbps
- **MII（Media Independent Interface）**：标准接口，支持 10/100 Mbps
- **RGMII（Reduced Gigabit MII）**：支持 10/100/1000 Mbps，高速接口，适用于千兆以太网

2. **PHY 连接**

- 外部 PHY 通过 MDC/MDIO 接口连接
- 支持自动协商速率和双工模式
- 可读取和写入 PHY 寄存器，用于配置和状态监控

2. MAC 特性

1. **双工与速率支持**

- 全/半双工
- 支持 10/100/1000 Mbps（RGMII）
- 支持自动协商或强制配置

2. **帧处理**

- VLAN 标签支持（IEEE 802.1Q，可选）
- 支持多播和广播帧过滤
- 硬件 CRC 生成和校验

3. **MAC 地址管理**

- 支持单个或多个 MAC 地址

- 可通过 FSP 或软件动态配置

3. DMA 与缓冲特性

1. **独立 TX/RX DMA 引擎**

- 支持同时发送和接收数据
- 减少 CPU 占用，提高吞吐量

2. **描述符队列**

- TX/RX 缓冲区数量可配置
- 支持链式 DMA，提高大数据传输效率

3. **多缓冲区管理**

- 支持环形缓冲区，实现连续传输
- 减少丢包

4. **硬件加速**

- 帧过滤、长度检测和 CRC 校验

4. 中断机制

1. **中断类型**

- 接收完成（RX）
- 发送完成（TX）
- 错误中断（CRC 错误、缓冲溢出）

2. **中断配置**

- 优先级可通过 FSP 配置
- 支持 RT-Thread ISR 集成

3. **优化**

- RX/TX 中断可与 DMA 联动
- 可选择性启用中断，提高性能

5. PHY 管理

1. **MDIO 接口**

- 可读写 PHY 寄存器，实现配置、复位和状态监控

2. **自动协商**

- 支持速率（10/100/1000 Mbps）和双工模式自动协商

3. **链路监控**

- 检测链路状态（Up/Down）
- CRC 错误和冲突检测

6. 协议与栈支持

1. **TCP/IP 协议栈集成**

- 兼容 LwIP
- 支持 TCP/UDP/ICMP、DHCP 客户端/服务器、ARP

2. **应用层支持**

- 支持 Telnet、HTTP、MQTT 等应用
- 多线程安全，支持并发访问

7. 性能与可靠性

1. **吞吐量优化**

- DMA + 中断减少 CPU 占用
- TX/RX 缓冲区大小可调

2. **可靠性特性**

- 硬件 CRC 校验
- VLAN 和多播过滤减少干扰
- 链路检测与自动重连

硬件说明

ETH0：

ETH1：

FSP 配置

- 新建 r_rmac stack：

- 配置 r_mac stack：

- 配置 r_layer3_switch：

- 配置 r_rmac_phy：

- 配置 g_rmac_phy_lsi0：

- ETH0 引脚配置：

- **注意**：ETH 相关的所有引脚需要将驱动能力改为 H。

RT-Thread Settings 配置

- 在 RT-Thread Settings 中使能 Ethernet。

软件说明

以太网的 phy 芯片初始化函数在 `./src/hal_entry.c` 中：

```
void rmac_phy_target_rtl8211_initialize (rmac_phy_instance_ctrl_t *
{
#define RTL_8211F_PAGE_SELECT 0x1F
#define RTL_8211F_EEELCR_ADDR 0x11
#define RTL_8211F_LED_PAGE 0xD04
#define RTL_8211F_LCR_ADDR 0x10
```

```
    uint32_t val1, val2 = 0;

    /* switch to led page */
    R_RMAC_PHY_Write(phydev, RTL_8211F_PAGE_SELECT, RTL_8211F_LED_P

    /* set led1(green) Link 10/100/1000M, and set led2(yellow) Link
    R_RMAC_PHY_Read(phydev, RTL_8211F_LCR_ADDR, &val1);
    val1 |= (1 << 5);
    val1 |= (1 << 8);
    val1 &= (~(1 << 9));
    val1 |= (1 << 10);
    val1 |= (1 << 11);
    R_RMAC_PHY_Write(phydev, RTL_8211F_LCR_ADDR, val1);

    /* set led1(green) EEE LED function disabled so it can keep on
    R_RMAC_PHY_Read(phydev, RTL_8211F_EEELCR_ADDR, &val2);
    val2 &= (~(1 << 2));
    R_RMAC_PHY_Write(phydev, RTL_8211F_EEELCR_ADDR, val2);

    /* switch back to page0 */
    R_RMAC_PHY_Write(phydev, RTL_8211F_PAGE_SELECT, 0xa42);
}

bool rmac_phy_target_rtl8211_is_support_link_partner_ability (rmac_
                                                              uint32
{
    FSP_PARAMETER_NOT_USED(p_instance_ctrl);
    FSP_PARAMETER_NOT_USED(line_speed_duplex);

    /* This PHY-LSI supports half and full duplex mode. */
    return true;
}
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

将网线插入 ETH0 网口，按下复位按键重启开发板，等待 PHY0 link up 之后输入 `ifconfig` 可以查看开发板获取到的 IP 地址，然后输入 `ping baidu.com` 命令进行连通性测试。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Aug 14 2025 15:15:46
 2006 - 2024 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/DBG] Initializing 2 PHY ports...
[I/DBG] PHY0 initialization:
[I/DBG]   Basic Status: 0x7989
[I/DBG]   Control Reg:  0x1000
[I/DBG]   Link Status:  DOWN
[I/DBG]   Auto-Neg:     In Progress
[I/DBG] PHY1 initialization:
[I/DBG]   Basic Status: 0x0000
[I/DBG]   Control Reg:  0x0000
[I/DBG]   Link Status:  DOWN
[I/DBG]   Auto-Neg:     In Progress
[I/DBG] PHY Link Status Summary:
[I/DBG]   PHY0: DOWN
[I/DBG]   PHY1: DOWN
[W/DBG] R_ETHER_Write failed!, res = 4001
[I/sal.skt] Socket Abstraction Layer initialize success.

Hello RT-Thread!
===================================================
This example project is an ethernet routine!
===================================================
msh />[W/DBG] R_ETHER_Write failed!, res = 4001
[I/DBG] Interrupt: Link ON detected, status: 0x01
[I/DBG] PHY0 Manual Link status changed: UP
[I/DBG] PHY0 link up

msh />if
ifconfig
msh />ifconfig
network interface device: e0 (Default)
MTU: 1500
MAC: 00 11 22 33 44 55
FLAGS: UP LINK_UP INTERNET_DOWN DHCP_ENABLE ETHARP BROADCAST IGMP
ip address: 10.23.8.224
gw address: 10.23.8.254
net mask  : 255.255.255.0
dns server #0: 10.23.8.11
dns server #1: 0.0.0.0
msh />ping baidu.com
ping: not found specified netif, using default netdev e0.
60 bytes from 182.61.201.211 icmp_seq=1 ttl=51 time=29 ms
60 bytes from 182.61.201.211 icmp_seq=2 ttl=51 time=28 ms
60 bytes from 182.61.201.211 icmp_seq=3 ttl=51 time=28 ms
60 bytes from 182.61.201.211 icmp_seq=4 ttl=51 time=28 ms

--- 182.61.201.211 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss
minimum = 28ms, maximum = 29ms, average = 28ms
msh />
```

3.4. GPT 使用说明

中文 | English

简介

在我们具体的应用场合中往往都离不开 timer 的使用，本例程主要介绍了如何在 Titan Board 上使用 GPT 设备，包括基本定时器的使用和 PWM 的使用。

PWM 简介

PWM(Pulse Width Modulation , 脉冲宽度调制) 是一种对模拟信号电平进行数字编码的方法，通过不同频率的脉冲使用方波的占空比用来对一个具体模拟信号的电平进行编码，使输出端得到一系列幅值相等的脉冲，用这些脉冲来代替所需要波形的设备。

上图是一个简单的 PWM 原理示意图，假定定时器工作模式为向上计数，当计数值小于阈值时，则输出一种电平状态，比如高电平，当计数值大于阈值时则输出相反的电平状态，比如低电平。当计数值达到最大值是，计数器从0开始重新计数，又回到最初的电平状态。高电平持续时间（脉冲宽度）和周期时间的比值就是占空比，范围为0~100%。上图高电平的持续时间刚好是周期时间的一半，所以占空比为50%。

一个比较常用的pwm控制情景就是用来调节灯或者屏幕的亮度，根据占空比的不同，就可以完成亮度的调节。PWM调节亮度并不是持续发光的，而是在不停地点

亮、熄灭屏幕。当亮、灭交替够快时，肉眼就会认为一直在亮。在亮、灭的过程中，灭的状态持续时间越长，屏幕给肉眼的观感就是亮度越低。亮的时间越长，灭的时间就相应减少，屏幕就会变亮。

定时器（Timer）简介

1. 概述

定时器（Timer） 是微控制器（MCU）或嵌入式系统中用于产生精确时间间隔或计数事件的硬件模块。定时器通常可以：

- 产生固定时间间隔的中断
- 测量时间长度或事件间隔
- 驱动 PWM 信号生成
- 计数外部脉冲

定时器是 MCU 实时控制、PWM 输出、事件计数、定时采样等应用的核心模块。

2. 定时器类型

根据功能和用途，定时器主要可分为以下几类：

1. **基本定时器（Basic Timer）**

- 简单的计数器，可用于产生固定时间间隔中断
- 通常只有向上计数功能

2. **通用定时器（General Purpose Timer）**

- 支持多种模式：单次计数、循环计数、PWM 输出、输入捕获
- 可配置预分频器、计数模式和自动重装载值

3. **高级定时器（Advanced Timer）**

- 支持更复杂功能，如死区时间、同步触发、互补 PWM 输出
- 多用于电机控制和功率电子应用

4. **看门狗定时器（Watchdog Timer）**

- 用于系统可靠性监控
- 如果 MCU 未按时复位定时器，会触发复位或中断

5. **实时时钟（RTC）定时器**

- 提供日历/时钟功能
- 通常低功耗，可在待机模式下继续计时

3. 定时器工作原理

定时器通常由以下几个部分组成：

1. **计数器（Counter）**

- 用于累加或递减计数值
- 计数单位由 **时钟频率（Timer Clock）** 决定

2. **预分频器（Prescaler）**

- 用于将系统时钟分频，降低计数器计数速度
- 公式：Timer Tick Frequency = Timer Clock / (Prescaler + 1)

3. **自动重装载寄存器（ARR / Period Register）**

- 当计数器达到该值时触发中断或产生事件
- 可实现循环计数

4. **中断/事件生成**

- 定时器计数到 ARR 时可触发中断、PWM 更新或外部事件

4. 定时器模式

1. **单次模式（One-Shot Mode）**

- 计数器计数到目标值后停止
- 用于一次性延时或事件触发

2. **循环模式（Auto-Reload / Continuous Mode）**

- 计数器计数到目标值后自动重装载继续计数
- 用于周期性定时器和 PWM 输出

3. **PWM 模式**

- 通过比较寄存器（Compare Register）生成占空比可调的 PWM 信号
- 广泛应用于电机控制、LED 调光

4. **输入捕获模式（Input Capture）**

- 捕获外部信号的到达时间
- 可用于测量脉宽、频率或事件间隔

5. **输出比较模式（Output Compare）**

- 当计数器达到比较值时改变输出状态
- 用于定时事件触发或波形产生

RA8 系列 GPT 模块概述

Renesas RA8 系列微控制器集成了高性能的通用 PWM 定时器（GPT）模块，支持多种定时和控制功能，适用于多种应用场景。

GPT 模块特性

- **支持模式**：周期模式、单次模式和 PWM 模式。
- **计数源**：支持 PCLK、外部触发引脚（GTETRG）、GTIOC 引脚或 ELC 事件作为计数源。
- **PWM 输出**：支持对 GTIOC 引脚输出周期性或 PWM 信号。
- **可配置性**：支持运行时重新配置周期、占空比和比较匹配值。
- **计数方式**：支持计数上升、下降或双向计数。
- **高分辨率**：支持高分辨率 PWM 波形生成，适用于精密控制应用。

PWM 输出实现

PWM（脉宽调制）是一种常用的模拟信号模拟方法，通过调节脉冲的占空比来控制输出信号的平均电压。

1. RT-Thread PWM 驱动框架

RT-Thread 提供了统一的 PWM 驱动框架，应用程序可以通过以下接口访问 PWM 设备：

- `rt_device_find()`：查找 PWM 设备句柄。
- `rt_pwm_set()`：设置 PWM 的周期和脉冲宽度。
- `rt_pwm_enable()`：使能 PWM 输出。
- `rt_pwm_disable()`：禁用 PWM 输出。

2. 运行时调整 PWM 参数

RA8 GPT 模块支持在运行时调整 PWM 的周期和占空比。可以使用以下函数进行调整：

- `rt_pwm_set()`：设置 PWM 的周期和脉冲宽度。

例如，要将 PWM 的周期调整为 500000，占空比调整为 70%，可以使用以下代码：

```
#defien PWM_DEV_NAME    "pwm12"
#define PWM_DEV_CHANNEL 0
struct rt_device_pwm *pwm_dev = RT_NULL;
rt_uint_32_t period = 500000;
rt_uint_32_t pulse = 350000;

pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL)
```

硬件定时器（hwtimer）实现

硬件定时器用于实现高精度的定时控制，适用于周期性任务调度、事件计时等场景。

1. RT-Thread hwtimer 驱动框架

RT-Thread 提供了统一的硬件定时器驱动框架，应用程序可以通过以下接口访问硬件定时器：

- `rt_device_find()`：查找硬件定时器设备句柄。

- `rt_device_open()`：以读写方式打开定时器设备。
- `rt_device_set_rx_indicate()`：设置超时回调函数。
- `rt_device_control()`：控制定时器设备，可以设置定时模式(单次/周期)/计数频率，或者停止定时器。
- `rt_device_write()`：设置定时器超时值，定时器随即启动。
- `rt_device_read()`：获取定时器当前值。
- `rt_device_close()`：关闭定时器设备。

2. RA8 GPT 配置示例

以下是使用 RA8 GPT 模块实现硬件定时器的配置示例：

```
/* 获取定时器时钟频率 */
rt_uint32_t freq = R_FSP_SystemClockHzGet(FSP_PRIV_CLOCK_PCLKD) >>
/* 查找定时器设备 */
rt_device_t hw_dev = rt_device_find("timer1");
/* 打开定时器设备 */
rt_err_t ret = rt_device_open(hw_dev, RT_DEVICE_OFLAG_RDWR);
/* 设置超时回调函数 */
rt_device_set_rx_indicate(hw_dev, timeout_cb);
/* 设置时钟频率 */
rt_device_control(hw_dev, HWTIMER_CTRL_FREQ_SET, &freq);
/* 设置定时模式为周期定时 */
rt_hwtimer_mode_t mode = HWTIMER_MODE_PERIOD;
ret = rt_device_control(hw_dev, HWTIMER_CTRL_MODE_SET, &mode);
/* 设置定时器超时时间 */
timeout_s.sec = 1;  /* secend */
timeout_s.usec = 0; /* microsecend */
if (rt_device_write(hw_dev, 0, &timeout_s, sizeof(timeout_s)) != si
{
    rt_kprintf("set timeout value failed\n");
    return -RT_ERROR;
}
```

在上述示例中，硬件定时器的超时时间被设置为1秒。

3. 定时器中断处理

RA8 GPT 模块支持定时器中断，可以在定时器溢出时触发中断服务程序。以下是定时器中断处理的配置示例：

```
static rt_err_t timeout_cb(rt_device_t dev, rt_size_t size)
{
    rt_kprintf("this is hwtimer timeout callback fucntion!\n");
    rt_kprintf("tick is :%d !\n", rt_tick_get());
```

```
    return RT_EOK;
}
```

硬件说明

Titan Board 的树莓派接口上有 6 个 PWM 接口，本示例使用 P714 输出 PWM 波。

FSP 配置

- FSP 分别配置使能 GPT1 为基本定时器模式，GPT12 为 PWM 模式：

- 并配置 Pins 使能 GPT12：

RT-Thread Settings 配置

- 在配置中打开 timer1 使能与 PWM12 使能：

示例工程说明

本例程的源码位于 `/project/Titan_driver_gpt`：

```
/* This is a hwtimer example */
#define HWTIMER_DEV_NAME "timer1" /* device name */
static rt_err_t timeout_cb(rt_device_t dev, rt_size_t size)
{
    rt_kprintf("this is hwtimer timeout callback fucntion!\n");
    rt_kprintf("tick is :%d !\n", rt_tick_get());
    return RT_EOK;
```

```
}

int hwtimer_sample(void)
{
    rt_err_t ret = RT_EOK;
    rt_hwtimerval_t timeout_s;
    rt_device_t hw_dev = RT_NULL;
    rt_hwtimer_mode_t mode;

    rt_uint32_t freq = R_FSP_SystemClockHzGet(FSP_PRIV_CLOCK_PCLKD)
    rt_kprintf("GPT Timer clock freq is: %d hz\n", freq);

    hw_dev = rt_device_find(HWTIMER_DEV_NAME);
    if (hw_dev == RT_NULL)
    {
        rt_kprintf("hwtimer sample run failed! can't find %s device
        return -RT_ERROR;
    }

    ret = rt_device_open(hw_dev, RT_DEVICE_OFLAG_RDWR);
    if (ret != RT_EOK)
    {
        rt_kprintf("open %s device failed!\n", HWTIMER_DEV_NAME);
        return ret;
    }

    rt_device_set_rx_indicate(hw_dev, timeout_cb);

    rt_device_control(hw_dev, HWTIMER_CTRL_FREQ_SET, &freq);

    mode = HWTIMER_MODE_PERIOD;
    ret = rt_device_control(hw_dev, HWTIMER_CTRL_MODE_SET, &mode);
    if (ret != RT_EOK)
    {
        rt_kprintf("set mode failed! ret is :%d\n", ret);
        return ret;
    }

    /* Example Set the timeout period of the timer */
    timeout_s.sec = 1;  /* secend */
    timeout_s.usec = 0; /* microsecend */
    if (rt_device_write(hw_dev, 0, &timeout_s, sizeof(timeout_s)) !
    {
        rt_kprintf("set timeout value failed\n");
        return -RT_ERROR;
    }

    /* read hwtimer value */
    rt_device_read(hw_dev, 0, &timeout_s, sizeof(timeout_s));
    rt_kprintf("Read: Sec = %d, Usec = %d\n", timeout_s.sec, timeou

    return ret;
```

```
}
MSH_CMD_EXPORT(hwtimer_sample, hwtimer sample);
```

每隔 1s 触发一次中断回调函数打印输出，下面是 PWM 配置使能：

PWM 相关宏定义：

当前版本的 PWM 驱动将每个通道都看做一个单独的 PWM 设备，每个设备都只有一个通道 0。使用 PWM12 设备，注意此处通道选择为 0 通道；

```
#define PWM_DEV_NAME        "pwm12" /* PWM设备名称 */
#define PWM_DEV_CHANNEL      0      /* PWM通道 */
struct rt_device_pwm *pwm_dev;      /* PWM设备句柄 */
```

配置 PWM 周期以及占空比：

```
static int pwm_sample(int argc, char *argv[])
{
    rt_uint32_t period, pulse;

    if (argc != 3)
    {
        LOG_I("Usage: pwm_sample <period> <pulse>");
        LOG_I("Example: pwm_sample 500000 250000");
        return -RT_ERROR;
    }

    period = (rt_uint32_t)atoi(argv[1]);
    pulse  = (rt_uint32_t)atoi(argv[2]);

    if (period == 0 || pulse > period)
    {
        LOG_E("Error: Invalid parameters. Ensure period > 0 and pul
        return -RT_ERROR;
    }

    pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
    if (pwm_dev == RT_NULL)
    {
        LOG_E("Error: Cannot find PWM device named '%s'\n", PWM_DEV
        return -RT_ERROR;
    }

    if (rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse) != RT_E
    {
        LOG_E("Error: Failed to set PWM configuration.");
        return -RT_ERROR;
    }

    if (rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL) != RT_EOK)
    {
```

```
        LOG_E("Error: Failed to enable PWM output.");
        return -RT_ERROR;
    }

    LOG_I("PWM started on device: %s, channel: %d", PWM_DEV_NAME, P
    LOG_I("Period: %u ns, Pulse: %u ns", period, pulse);
    LOG_I("Please connect the \'P714\' to a logic analyzer or oscil

    return RT_EOK;
}
MSH_CMD_EXPORT(pwm_sample, Configure and start PWM output: pwm_samp
```

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

在串口终端分别输入pwm_sample、hwtimer_sample查看具体效果；

每隔 1s 触发回调函数并打印输出：

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 25 2025 15:18:01
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
==================================================
This example project is an pwm(GPT) routine!
==================================================
[I/pwm] Tips
[I/pwm] Please enter one of the above commands to run the corresponding sample.
[I/pwm]   • hwtimer_sample : Demonstration of Hardware Timer usage.
[I/pwm]   • pwm_sample     : Demonstration of PWM (Pulse Width Modulation) usage.
msh >pwm_sample 500000 250000
[I/pwm] PWM started on device: pwm12, channel: 0
[I/pwm] Period: 500000 ns, Pulse: 250000 ns
[I/pwm] Please connect the 'P714' to a logic analyzer or oscilloscope for waveform observation.
msh >hwtimer_sample
GPT Timer clock freq is: 250000000 hz
Read: Sec = 1, Usec = 0
msh >this is hwtimer timeout callback fucntion!
tick is :42566 !
this is hwtimer timeout callback fucntion!
tick is :43561 !
this is hwtimer timeout callback fucntion!
tick is :44556 !
this is hwtimer timeout callback fucntion!
tick is :45551 !
this is hwtimer timeout callback fucntion!
tick is :46546 !
this is hwtimer timeout callback fucntion!
tick is :47541 !
this is hwtimer timeout callback fucntion!
tick is :48536 !
```

使用逻辑分析仪量取 PWM 输出波形如下所示：

3.5. HyperRAM 使用说明

中文 | **English**

简介

本示例演示了如何在 **Titan Board** 上使用 **Octa-SPI (OSPI) 接口**连接 **HyperRAM** 外部存储器，并基于 RT-Thread 驱动框架进行读写测试。通过该示例，用户可以熟悉 RA8 OSPI 控制器的配置方法、HyperRAM 的工作模式，以及在 RT-Thread 下访问外部存储器的应用流程。

HyperRAM 简介

1. 概述

HyperRAM 是一种由 **Cypress（现 Infineon）** 首次推出的 **高性能低引脚数（Low Pin Count, LPC）DRAM**。

它基于 **HyperBus 接口**，主要面向 **嵌入式系统、显示控制、IoT 设备、汽车电子** 等需要 **高带宽、低功耗、简单接口** 的应用。

HyperRAM 属于 **pSRAM（Pseudo-SRAM，伪静态RAM）**，它在外部表现得像 SRAM（简单读写，无需用户刷新），但内部实际上是低功耗 DRAM（自刷新）。

2. 架构与接口

HyperRAM 使用 **HyperBus 接口**，其特点是：

- **引脚数少**：通常仅需 **13 个信号引脚**（8-bit 数据总线 + 控制/时钟），相比传统 SDRAM（几十个引脚）大幅减少 PCB 复杂度。
- **双数据速率（DDR）传输**：在时钟上升沿和下降沿传输数据，提高带宽。
- **串行控制协议**：通过命令-地址-数据序列访问内存，简化设计。

接口结构如下：

- **数据线 DQ[7:0]**：8 位双向数据
- **RWDS（Read-Write Data Strobe）**：数据同步信号
- **CLK**：时钟输入
- **CS#**：片选信号
- **RESET#**：复位
- **CKE**：时钟使能

3. 工作原理

HyperRAM 通过 **命令+地址+数据** 的方式访问：

1. **命令阶段**

- 主机发送读/写命令和目标地址。

2. **延迟阶段**

- HyperRAM 准备内部存储阵列（延迟可配置）。

3. **数据传输阶段**

- 以 DDR 方式在 **DQ[7:0]** 上传输数据，RWDS 提供数据同步。

内部采用 DRAM 技术，支持 **自刷新**，但对外表现为 “像 SRAM 一样” ——用户无需关心刷新操作。

4. 性能特性

- **数据总线宽度**：8 位
- **工作电压**：1.8 V 或 3.0 V 低功耗设计
- **数据速率**：最高可达 **400 MB/s（200 MHz DDR × 8-bit）**
- **容量范围**：32 Mb ~ 512 Mb（4 MB ~ 64 MB）
- **低功耗**：支持深度睡眠模式，待机电流 < 10 μA
- **简单接口**：13 根引脚即可完成高速访问

5. HyperRAM 的优势

1. **低引脚数**

- 与传统 SDRAM/PSRAM（30+ 引脚）相比大幅减少引脚需求，节省 PCB 走线。

2. **高带宽**

- DDR 接口，带宽可达 400 MB/s，足以支持 **图像缓存、显示刷新** 等应用。

3. **低功耗**

- 适合电池供电设备，如 IoT、可穿戴设备。

4. **易用性**

- 对外表现类似 SRAM，简单易用，无需用户刷新。

6. 与其他存储的对比

特性	HyperRAM	SDRAM / DDR	SRAM	NOR Flash
接口	HyperBus (13-pin)	并行 16~32 位	并行/串行	SPI/QSPI
容量范围	32Mb ~ 512Mb	64Mb ~ 1Gb+	小（Kb~Mb）	4Mb ~ 2Gb
带宽	~400 MB/s	~800 MB/s+	~50 MB/s	~100 MB/s
功耗	低	较高	较低	较低
应用场景	缓存/帧缓冲	系统主存	高速小容量	程序存储

RA8 系列 OSPI (Octa-SPI) 特性

OSPI (Octa-SPI) 是 RA8 系列 MCU 中的重要外设接口模块，专门用于连接高速外部存储器（如 **Octa Flash、HyperFlash、HyperRAM**），以满足嵌入式应用在 **大容量存储、高速缓存、XIP 执行**等方面的需求。

1. 协议支持

RA8 OSPI 兼容多种存储接口协议，灵活适配不同外部存储器：

- **SPI 标准模式**：1-1-1 (单线指令、地址、数据)
- **Dual-SPI**：2-2-2 模式（双线）
- **Quad-SPI**：4-4-4 模式（四线）
- **Octa-SPI**：8-8-8 模式（八线，支持 DDR/SDR）
- **HyperBus 协议**：
- 用于连接 **HyperRAM / HyperFlash**
- 双边沿数据采样（DDR）
- 传输速率可达 **333 MB/s**

2. 数据传输能力

- **最高频率**：支持 **166MHz 时钟**
- **DDR 模式**：数据在时钟上升沿与下降沿同时传输（带宽翻倍）
- **有效带宽**：

- Octa-SPI DDR 模式，8-bit 总线，166 MHz → **333 MB/s**
- HyperBus 模式同样可达 **333 MB/s**
- **Burst 访问**：支持可编程突发长度（16/32/64/128 字节），提升连续读写效率

3. 存储器映射与执行

RA8 OSPI 支持将外部存储映射到 **MCU 地址空间**，使外设访问与片上 SRAM 类似，方便应用：

- **XIP (Execute-In-Place)**：可直接在外部 Flash 中执行代码
- **存储映射缓存**：
- 内建预取缓冲器，提高指令读取速度
- 适合频繁读取的代码和数据场景
- **可配置读写访问模式**：
- 直接访问模式（命令驱动）
- 存储映射模式（内存地址空间直接访问）

4. 命令与时序支持

- **可编程命令集**：支持用户配置的指令序列（适配不同厂商 Flash/HyperRAM）
- **DQS (Data Strobe) 支持**：在 DDR 模式下提供时序同步，提高数据完整性
- **片选控制**：多片选引脚，可支持多个外部存储器
- **延迟补偿**：可配置时钟延迟、采样点位置，以适配高速 HyperRAM/HyperFlash

5. 低功耗与休眠管理

- **支持外设低功耗模式**：
- HyperRAM 深度休眠模式 (Deep Power-Down)
- HyperFlash 待机模式
- **快速唤醒**：通过片选控制快速恢复存储访问
- **适合电池供电应用**

硬件说明

FSP 配置

- 新建 r_ospi_b stack：

- 配置 r_ospi_b stack：

- HyperRAM 引脚配置：

- HyperRAM 所有相关引脚的驱动能力配置为 H，OM_1_SIO0~OM_1_SIO7 需要配置为输入上拉。

RT-Thread Settings 配置

- 在 RT-Thread Settings 中使能 HyperRAM。

软件说明

本例程的 HyperRAM 测试源码位于 `./board/ports/hyperram` 中：

```
#include <string.h>
#include <inttypes.h>
```

```
#include "hal_data.h"
#include "hyper_ram_test.h"
#include <rtthread.h>

#define DBG_TAG "hyperram"
#define DBG_LVL DBG_INFO
#include <rtdbg.h>

#define ram_cfg     g_ospi1_cfg
#define ram_ctrl    g_ospi1_ctrl

#define OSPI_OM_RESET               BSP_IO_PORT_12_PIN_07
#define HYPER_RAM_RESET_DELAY()     R_BSP_SoftwareDelay(10UL, BSP_D

#define HYPER_RAM_CFG_REG_0_ADDRESS (0x01000000)
#define HYPER_RAM_CFG_REG_1_ADDRESS (0x01000001)

ospi_b_xspi_command_set_t g_hyper_ram_commands[] =
{
    {
        .protocol = SPI_FLASH_PROTOCOL_8D_8D_8D,
        .frame_format = OSPI_B_FRAME_FORMAT_XSPI_PROFILE_2_EXTENDED
        .latency_mode = OSPI_B_LATENCY_MODE_FIXED,
        .command_bytes = OSPI_B_COMMAND_BYTES_1,
        .address_bytes = SPI_FLASH_ADDRESS_BYTES_4,

        .read_command = 0xA0,
        .read_dummy_cycles = 11,
        .program_command = 0x20,
        .program_dummy_cycles = 11,

        .address_msb_mask = 0xF0,
        .status_needs_address = false,

        .p_erase_commands = NULL,
    }
};

uint16_t swap16(uint16_t value)
{
    uint16_t ret;
    ret  = value << 8;
    ret |= value >> 8;
    return ret;
}

static fsp_err_t hyper_ram_config_get(uint32_t address, uint16_t *
{
    spi_flash_direct_transfer_t xfer = {
            .address = address,
            .address_length = 4,
            .command_length = 2,
```

```
            .command = 0xE000,
            .data_length = 2,
            .dummy_cycles = 1,
    };

    fsp_err_t err = R_OSPI_B_DirectTransfer(&ram_ctrl, &xfer, SPI_F
    if (err != FSP_SUCCESS)
    {
        LOG_E("HyperRAM config get failed!");
        return err;
    }

    *p_value_out = (uint16_t) xfer.data;

    return FSP_SUCCESS;
}

static fsp_err_t hyper_ram_config_set(uint32_t address, uint16_t va
{
    spi_flash_direct_transfer_t xfer = {
            .address = address,
            .address_length = 4,
            .command = 0x6000,
            .command_length = 2,
            .data = (uint16_t) value,
            .data_length = 2,
            .dummy_cycles = 0,
    };

    fsp_err_t err = R_OSPI_B_DirectTransfer(&ram_ctrl, &xfer, SPI_F
    if (err != FSP_SUCCESS)
    {
        LOG_E("HyperRAM config set failed!");
        return err;
    }

    return FSP_SUCCESS;
}


/* Define the static array at address 0x70000000 using section attr
#define TEST_SIZE 8388608 // 32MB (8M x 4 bytes)
static uint32_t test_array[TEST_SIZE] __attribute__((section(".ospi

void hyper_ram_test(void)
{
    const uint32_t test_bytes = TEST_SIZE * 4; // Total bytes: 32MB
    uint32_t errors = 0;
    rt_tick_t start_time, end_time;
    uint32_t write_speed_kbs, read_speed_kbs;
    uint32_t write_time_ms, read_time_ms;
```

```
    /* Write pattern to RAM and measure time */
    start_time = rt_tick_get();
    for (uint32_t i = 0; i < TEST_SIZE; i++)
    {
        test_array[i] = i ^ 0xA5A5A5A5; // Use XOR pattern for bett
    }
    end_time = rt_tick_get();

    /* Calculate write speed in KB/s and time in ms */
    write_time_ms = (end_time - start_time) * 1000 / RT_TICK_PER_SE
    if (write_time_ms > 0)
    {
        write_speed_kbs = (test_bytes / 1024) / write_time_ms * 100
    }
    else
    {
        write_speed_kbs = 0; // Avoid division by zero
    }

    /* Verify written pattern while measuring read time */
    start_time = rt_tick_get();
    for (uint32_t i = 0; i < TEST_SIZE; i++)
    {
        uint32_t expected = i ^ 0xA5A5A5A5;
        uint32_t actual = test_array[i];
        if (actual != expected)
        {
            errors++;
//            rt_kprintf("errors:%d actual:%d\n", errors, actual);
        }
    }
    end_time = rt_tick_get();

    /* Calculate read speed in KB/s and time in ms */
    read_time_ms = (end_time - start_time) * 1000 / RT_TICK_PER_SEC
    if (read_time_ms > 0)
    {
        read_speed_kbs = (test_bytes / 1024) / read_time_ms * 1000;
    }
    else
    {
        read_speed_kbs = 0; // Avoid division by zero
    }

    /* Print test results */
    if (errors == 0)
    {
        LOG_I("Hyper RAM test passed successfully!");
    }
    else
    {
        LOG_E("Hyper RAM test failed with %u errors", errors);
```

```
    }

    /* Print read and write speeds as integers */
    LOG_I("Write speed: %u KB/s (%u ms)", write_speed_kbs, write_ti
    LOG_I("Read speed: %u KB/s (%u ms)", read_speed_kbs, read_time_
}
MSH_CMD_EXPORT(hyper_ram_test, hyper_ram_test);

int hyper_ram_init(void)
{
    /* Change OM_RESET back to normal IO mode. */
    R_IOPORT_PinCfg(&g_ioport_ctrl,
                    OSPI_OM_RESET,
                    IOPORT_CFG_PORT_DIRECTION_OUTPUT
                        | IOPORT_CFG_DRIVE_HIGH
                        | IOPORT_CFG_PORT_DIRECTION_OUTPUT
                        | IOPORT_CFG_PORT_OUTPUT_HIGH);

    /* Pin reset the OctaFlash */
    R_BSP_PinWrite(OSPI_OM_RESET, BSP_IO_LEVEL_LOW);
    HYPER_RAM_RESET_DELAY();
    R_BSP_PinWrite(OSPI_OM_RESET, BSP_IO_LEVEL_HIGH);
    HYPER_RAM_RESET_DELAY();

    /* Open the interface and immediately transition to 8D-8D-8D mo
    R_OSPI_B_Open((spi_flash_ctrl_t *)&ram_ctrl, &ram_cfg);
    R_OSPI_B_SpiProtocolSet(&ram_ctrl, SPI_FLASH_PROTOCOL_8D_8D_8D)

    uint16_t cfg_reg0 = 0;
    hyper_ram_config_get(HYPER_RAM_CFG_REG_0_ADDRESS, &cfg_reg0);
    LOG_D("Read CR0 value: 0x%x", swap16(cfg_reg0));

    uint16_t value0 = 0x8f1d;
    hyper_ram_config_set(HYPER_RAM_CFG_REG_0_ADDRESS, swap16(value0
    LOG_D("Set CR0 to 0x%x", value0);

    cfg_reg0 = 0;
    hyper_ram_config_get(HYPER_RAM_CFG_REG_0_ADDRESS, &cfg_reg0);
    LOG_D("Read CR0 value: 0x%x", swap16(cfg_reg0));

    return RT_EOK;
}
INIT_BOARD_EXPORT(hyper_ram_init);
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

按下复位按键重启开发板，输入 `hyper_ram_test` 命令开始 HyperRAM 测试。

```
[D/hyperram] Set CR0 to 0x8f1d
[D/hyperram] Read CR0 value: 0x8f1d

 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Sep 25 2025 10:04:12
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
===================================================
This example project is an hyperram routine!
===================================================
msh >hyper_ram_test
[I/hyperram] Hyper RAM test passed successfully!
[I/hyperram] Write speed: 183000 KB/s (179 ms)
[I/hyperram] Read speed: 98000 KB/s (334 ms)
msh >
```

3.6. IST8310 示例说明

中文 | **English**

简介

本示例演示了如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 I2C 接口**连接 **IST8310 三轴磁力计传感器**，并通过 RT-Thread 提供的 **I2C 驱动框架**进行数据读取。
通过该示例，用户可以熟悉 **RA8 I2C Master 外设的配置方法**，以及在 RT-Thread 下访问传感器的应用流程。

IST8310 磁力计简介

1. 概述

IST8310 是由 **艾迈斯半导体（iSentek）** 出品的一款 **三轴地磁传感器（Magnetometer）**，主要用于测量地磁场强度，从而实现 **电子罗盘（E-Compass）功能**。

它常见于 **无人机、智能手机、可穿戴设备、导航系统** 中，用于提供航向信息或磁场检测。

2. 主要特性

- **三轴磁场测量**：X、Y、Z 三个方向同时测量
- **通信接口**：I^2C（最高 400 kHz）
- **测量范围**：±1600 μT
- **分辨率**：0.3 μT
- **输出速率**：可配置，最高支持 **200 Hz**
- **低功耗**：工作电流约 85 μA，待机电流 < 1 μA
- **供电电压**：1.8 V ~ 3.6 V（兼容常见 MCU）
- **封装**：3 × 3 × 1 mm LGA 封装，体积小，适合嵌入式应用

3. 工作原理

IST8310 基于 **霍尔效应（Hall Effect）** 或 **磁阻效应（Magneto-Resistive Effect）**实现磁场检测，核心流程如下：

1. **磁场检测**

- 通过传感单元感知地磁场（~50 μT）或其他磁场变化。

2. **信号调理**

- 内部模拟电路将感应信号转换为电压信号，并进行放大、滤波。

3. **A/D 转换**

- 内置 ADC 将模拟信号转为数字值。

4. **数据输出**

- 通过 I^2C 总线输出 X、Y、Z 三轴的磁场强度数值，供 MCU 计算航向角或姿态。

4. 性能指标

- **灵敏度**：0.3 μT / LSB
- **偏移误差**：典型值 ±1 μT
- **零点漂移**：温度补偿机制可减小漂移
- **采样率**：0.5 Hz ~ 200 Hz 可配置
- **工作温度范围**：-40 °C ~ +85 °C

5. 优势与限制

优势

- 体积小，功耗低，适合电池供电系统
- 输出为数字信号，MCU 接入简单
- 宽工作电压范围，适配性强

限制

- 需要外部 **校准**（硬铁、软铁效应校正）才能保证精度
- 受温度、环境磁场干扰较大
- 单独使用时易受噪声影响，常与 IMU（加速度计 + 陀螺仪）融合

RA8 系列 I2C Master 特性

RA8 系列 MCU 内置的 **I2C 控制器**是一款高性能、多功能的主从兼容模块，可用于与各种 I2C 设备（如传感器、EEPROM、时钟芯片）高速可靠地通信。

1. 总体架构

RA8 I2C 控制器主要由以下模块组成：

- **主/从控制逻辑 (Master/Slave Control)**
- 支持主模式、从模式和多主模式
- 自动处理总线仲裁与冲突
- 支持 7-bit 和 10-bit 地址格式
- **时钟生成与分频模块 (Clock Generator & Prescaler)**
- 内置可编程分频器，支持标准、快速、高速模式
- 支持波特率精确设置
- **FIFO 缓冲区 (Transmit/Receive FIFO)**
- 发送 FIFO 和接收 FIFO 独立
- 减少 CPU 干预，提高数据吞吐率
- 支持批量读写，适合高速传感器数据采集
- **中断控制模块 (Interrupt Controller)**
- 支持多种事件中断，包括：
  - 传输完成
  - 仲裁丢失
  - 总线错误（Bus Error）
  - FIFO 空/满检测
- 可选择中断或轮询模式
- **DMA 支持 (Direct Memory Access)**
- 支持发送/接收数据通过 DMA 传输
- 减少 CPU 占用，降低功耗
- 适合高频率、大数据量的 I2C 读写

2. 支持的 I2C 模式

模式	速率	特性
Standard Mode	100 kbps	兼容传统 I2C 设备
Fast Mode	400 kbps	高速传感器和 EEPROM 支持
Fast Mode Plus	1 Mbps	提升总线带宽，减少通信延迟
High-Speed Mode	3.4 Mbps	适合高速数据采集设备

模式	速率	特性
Address Mode	7-bit / 10-bit	支持标准与扩展从机地址格式

3. 总线管理与多主功能

- **总线仲裁 (Arbitration)**
- 当多主同时发送时，自动检测冲突
- 按 I2C 标准规则处理仲裁，防止总线占用冲突
- **起始/停止条件生成**
- 硬件自动生成 START / STOP 信号
- 支持重复起始条件（Repeated Start）
- **ACK/NACK 支持**
- 自动检测从机应答
- 支持 NACK 检测以判断通信异常
- **时钟同步 (Clock Stretching)**
- 支持从机延长时钟拉低的功能
- 确保慢速从机数据传输可靠

4. 时序与数据传输特性

- **双向数据线 (SDA) 与时钟线 (SCL)**
- SDA/SCL 通过开漏输出，需要外部上拉电阻（4.7kΩ~10kΩ）
- 支持 1.8V / 3.3V IO 电平
- **高速数据传输**
- 支持标准模式/快速模式/高速模式
- Fast-Mode Plus 可达 1 Mbps
- 支持多主/多从、全双向同步通信
- **FIFO & Burst 支持**
- 支持 8~32 字节 FIFO，可连续发送/接收数据
- 降低 CPU 中断频率

5. DMA 与中断机制

- **DMA 支持**
- TX/RX FIFO 可连接 DMA 控制器
- 自动触发 DMA 传输，减轻 CPU 负载
- 适合高频传感器数据采集或连续数据流
- **中断类型**
- **传输完成中断**：TX FIFO 空或 RX FIFO 满
- **仲裁丢失中断**：多主冲突时触发
- **总线错误中断**：起始/停止/ACK/NACK 错误
- **FIFO 阀值中断**：FIFO 高/低水位报警

6. 异常检测与容错机制

- **仲裁丢失检测**：保证多主总线安全
- **总线错误检测**：检测意外停止条件或非法操作
- **从机未响应处理**：可配置重试次数或中断响应
- **SCL 超时保护**：防止总线被长时间拉低

7. 电气与功耗特性

- **IO 电压**：1.8V / 3.3V 可配置
- **低功耗模式**：支持总线空闲模式进入低功耗
- **SDA/SCL 空闲功耗极低**
- **支持深度睡眠唤醒**：I2C 外设可用于唤醒 MCU

硬件说明

Titan Board 使用 IIC2 与 IST8310 通信。

FSP 配置

- 新建 stacks 选择 r_iic_master 并配置 IIC2 配置信息如下：

RT-Thread Settings 配置

- 在配置中打开 RT-Thread 的 IIC 驱动框架与 IST8310 的驱动软件包；

示例工程说明

基于 IST8310 的驱动软件包实现对磁力计的数据通信。

```
/*
* Copyright (c) 2006-2025, RT-Thread Development Team
*
* SPDX-License-Identifier: Apache-2.0
*
* Change Logs:
* Date           Author        Notes
* 2025-06-13     kurisaW       first version
*/

#include <rtthread.h>
#include "ist8310.h"

static void ist8310_entry()
{
```

```
    ist8310_device_t dev = ist8310_init(IST8310_SAMPLE_I2C_DEV_NAME
    if (dev == RT_NULL) {
        rt_kprintf("IST8310 init failed\n");
        return;
    }

    /* 设置磁偏角（根据实际位置设置） */
    ist8310_set_declination(dev, 0.15f);  /* 例如: 0.15弧度 */

    while (1)
    {
        ist8310_data_t data;
        if (ist8310_read_magnetometer(dev, &data) == RT_EOK)
        {
            rt_kprintf("Magnetic: X=%.2f μT, Y=%.2f μT, Z=%.2f μT\n
        }

        float heading = ist8310_read_heading(dev);
        rt_kprintf("Heading: %.2f°\n", heading);

        rt_thread_mdelay(1000);
    }
}

void ist8310_app()
{
    rt_thread_t ist8310 = rt_thread_create("ist8310", ist8310_entry
    if(ist8310 != RT_NULL)
    {
        rt_thread_startup(ist8310);
    }

    return;
}
MSH_CMD_EXPORT(ist8310_app, IST8310 app);
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

串口终端输入 ist8310_app 指令：

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 28 2025 09:17:31
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
===================================================
This example project is an sensor routine for IST8310!
===================================================
[I/iic] Tips:
[I/iic] You can run iic testcase by executing the instruction: 'ist8310_app'
msh >ist8310_app
msh >Magnetic: X=0.00 μT, Y=0.00 μT, Z=0.00 μT
Heading: 193.36°
Magnetic: X=1792.00 μT, Y=-21760.00 μT, Z=16128.00 μT
Heading: 193.30°
Magnetic: X=1792.00 μT, Y=-21760.00 μT, Z=16128.00 μT
Heading: 193.36°
Magnetic: X=1792.00 μT, Y=-21760.00 μT, Z=16128.00 μT
Heading: 193.36°
Magnetic: X=1792.00 μT, Y=-21760.00 μT, Z=15872.00 μT
Heading: 193.30°
Magnetic: X=1792.00 μT, Y=-21248.00 μT, Z=16128.00 μT
Heading: 193.30°
Magnetic: X=1792.00 μT, Y=-21760.00 μT, Z=16384.00 μT
Heading: 193.30°
Magnetic: X=1792.00 μT, Y=-21504.00 μT, Z=15872.00 μT
Heading: 193.36°
```

3.7. RGB LCD 使用说明

中文 | **English**

简介

本示例演示了如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 GLCDC 模块**结合 **RT-Thread 的 LCD 驱动框架**驱动 RGB LCD 屏幕，实现图像显示与界面更新。文档将详细介绍 **RA8 系列 GLCDC 外设特性**及 **RT-Thread LCD 驱动框架**，并提供示例配置和操作方法。

RA8 系列 GLCDC 模块

1. 简介

GLCDC（Graphics LCD Controller）是 RA8 系列 MCU 内置的高性能图形控制器模块，专门用于驱动 TFT/RGB LCD 屏幕，支持各种分辨率、色彩格式和图像处理功能。结合 RT-Thread 的 **LCD 驱动框架**，可以实现统一接口下的屏幕初始化、刷新、图像绘制及 DMA 加速等功能。

RA8 系列 **GLCDC (Graphics LCD Controller)** 提供了从 MCU 内部存储器或外部图像缓存向 RGB/LCD 显示屏输出图像的能力。它集成了：

- **帧缓冲控制**：可配置多帧缓冲，实现页面切换或双缓冲显示
- **颜色格式支持**：RGB565, RGB888, ARGB8888 等
- **图形处理功能**：背景图层、文字/图形合成、透明度混合、调色板等
- **同步信号生成**：HSYNC, VSYNC, DE（Data Enable）等
- **DMA 支持**：高速数据传输，减少 CPU 占用
- **中断功能**：帧结束中断、行结束中断等

2. 模块架构

RA8 GLCDC 模块主要包含以下子模块：

1. **图层合成单元（Layer Composition Unit）**

- 支持多图层叠加
- 提供 alpha blending、透明度控制、颜色键控
- 可以对图层进行旋转、翻转处理

2. **帧缓冲接口（Frame Buffer Interface）**

- 支持访问 MCU 内部 SRAM 或外部存储器
- 提供单/双缓冲模式，保证连续显示
- 配合 DMA 自动读取图像数据

3. **DMA 控制器（DMA Controller）**

- 自动传输像素数据到 RGB 输出端口
- 可配置突发长度，提高带宽利用率
- 支持循环传输，适合视频或动画场景

4. **同步信号生成器（Timing Generator）**

- 自动生成 HSYNC/VSYNC/DE 信号
- 支持 TTL 接口 RGB 时序
- 可配置极性、同步宽度、前后肩时间等

5. **中断与事件处理单元（Interrupt/Event Controller）**

- 提供帧结束中断、行结束中断
- 可用于页面切换、动态绘制或滚动显示
- 支持 DMA 传输完成触发中断

3. GLCDC 工作原理

1. **帧缓冲读取**

- GLCDC 通过 DMA 从内存读取图像数据，支持单/双缓冲模式，保证连续显示。

2. **图层合成**

- 支持多图层叠加，例如背景图层 + 前景图层 + 图标/文字图层
- 提供透明度控制和调色板映射

3. **像素时序输出**

- 根据 LCD 的接口要求生成 HSYNC/VSYNC/DE 信号
- 支持 RGB 并行接口、TTL 接口或 LVDS（视具体板级实现）

4. **中断与事件**

- **帧结束中断（VBlank）**：可用于更新下一帧数据
- **行结束中断**：可用于滚动显示或动态绘制

4. GLCDC 支持的功能与特性

功能类别	描述
分辨率	最高可达 800×480 (具体取决于 MCU 型号及时钟配置)
色彩模式	RGB565、RGB888、ARGB8888 等
多图层	背景 + 前景 + 符号图层，可叠加混合
帧缓冲	支持单帧/双帧缓冲模式，DMA 传输提高性能
调色板	支持 8/16 位调色板映射，实现色彩转换
同步信号	HSYNC, VSYNC, DE，可配置极性和时序
DMA 支持	自动从内存传输图像数据，无需 CPU 干预
中断	帧结束、行结束中断，可用于屏幕刷新同步
旋转/翻转	支持 90°/180°/270°旋转及 X/Y 翻转

RT-Thread LCD 驱动框架

RT-Thread 提供了 **统一的 LCD 驱动框架**，通过封装底层控制器，支持不同屏幕类型与 MCU 外设。框架主要接口如下：

主要接口

函数/宏	功能
`rt_device_find("lcd")`	查找 LCD 设备句柄
`rt_device_open(dev, flags)`	打开设备，初始化 LCD 硬件
`rt_device_control(dev, cmd, args)`	控制 LCD，例如更新lcd图像、设置背光、读取屏幕信息
`rt_device_write(dev, pos, buffer, size)`	向 LCD 写入像素数据

函数/宏	功能
`rt_device_read(dev, pos, buffer, size)`	从 LCD 读取像素数据（部分屏幕支持）
`rt_device_close(dev)`	关闭设备

硬件说明

+5V
C110
100nF
C112
4.7uF
GND
+3V3
R96
R97
10KΩ
10KΩ
SDA0
SCL0
+5V
J8
AFC01-S40FCA-00
1 +5V
2 +5V
3 R0
4 R1
5 R2
6 R3
7 R4
8 R5
9 R6
10 R7
11 GND
12 G0
13 G1
14 G2
15 G3
16 G4
17 G5
18 G6
19 G7
20 GND
21 B0
22 B1
23 B2
24 B3
25 B4
26 B5
27 B6
28 B7
29 GND
30 PCLK
31 HSYNC
32 VSYNC
33 DE
34 LCD_BL
35 CTP_RST
36 CTP_SDA
37 #RESERVED
38 CTP_SCL
39 CTP_IRQ_N
40 LCD_RST
41 SHELL
42 SHELL
LCD_DATA0
LCD_DATA1
LCD_DATA2
LCD_DATA3
LCD_DATA4
LCD_DATA5
LCD_DATA6
LCD_DATA7
LCD_DATA8
LCD_DATA9
LCD_DATA10
LCD_DATA11
LCD_DATA12
LCD_DATA13
LCD_DATA14
LCD_DATA15
LCD_CLK
LCD_TCON0
LCD_TCON1
LCD_TCON3
LCD_BL
CTP_RST
SDA0
SCL0
CTP_IRQ_N
LCD_RST
GND
Titar
Sheet: /CAM
File: CAMERA
Title: POI

FSP 配置

HyperRAM 配置

- 新建 r_ospi_b stack：

- 配置 r_ospi_b stack：

- HyperRAM 引脚配置：

RGB LCD 配置

- 新建 `r_glcdc` stack：

- 配置中断回调和图形层1：

- 配置输出参数、CLUT、TCON和抖动。

- 配置 GLCDC 的引脚：

LCD 背光配置

- 新建 `r_gpt` stack：

- 配置背光 PWM 输出：

D/AVE 2D 配置

- 新建 `r_drw` stack：

RT-Thread Settings 配置

- 在 RT-Thread Settings 中使能 RGB565 LCD，使用 pwm7 输出屏幕背光。

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

复位开发板后在终端输入 `lcd_test` 命令运行刷屏程序。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Aug 14 2025 16:31:18
 2006 - 2024 Copyright by RT-Thread team
[D/drv_lcd]
lcd framebuffer address:0x70000000

Hello RT-Thread!
==================================================
This example project is an rgb lcd routine!
==================================================
msh >lcd_test
[D/drv_lcd] red buffer...
[D/drv_lcd] green buffer...
[D/drv_lcd] blue buffer...
[D/drv_lcd] red buffer...
[D/drv_lcd] green buffer...
[D/drv_lcd] blue buffer...
msh >
```

ALIENTEK 4.3' RGBLCD

3.8. RS485 驱动例程

中文 | English

简介

本示例演示了如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 UART 模块**实现 **RS485 半双工通信**，并基于 **RT-Thread 串口驱动框架**进行数据收发。通过该示例，用户可以熟悉 **RA8 UART 外设的配置方法**、RS485 通信模式的设置，以及在 RT-Thread 下的应用流程。

RS-485 简介

1. 概述

RS-485（也称 **TIA/EIA-485**）是一种**差分信号串行通信标准**，用于工业控制、楼宇自动化和远距离数据传输。它是 **RS-232 的增强版**，可以支持更长距离和多节点通信。

主要特点：

- **差分传输**：使用 A/B 两条线传输信号，提高抗干扰能力
- **多点通信**：单总线可挂载 **多达 32 个驱动器和 32 个接收器**（扩展器件可更多）
- **长距离传输**：标准可达 **1200 米**，速率与距离成反比
- **半双工或全双工**：可灵活配置，常见工业总线协议（Modbus RTU）采用半双工

2. 物理层特性

特性	描述
信号类型	差分信号（A/B 两线）
驱动电压	±1.5 V 至 ±5 V
接收器阈值	⩾ ±200 mV
总线最大节点数	32 个驱动器 + 32 个接收器（标准）
总线长度	最高可达 1200 m（速率低可更长）

特性	描述
终端电阻	120 Ω，匹配总线阻抗，减少反射

差分信号原理：

- A/B 两条线同时传输电流方向相反的信号
- 接收器测量 **V_AB = V_A - V_B**
- 即使有共模干扰，差分电压仍可保持正确逻辑

3. 通信模式

1. **半双工（Half-Duplex）**

- 总线单方向通信
- 通过控制发送/接收使能实现
- 常用于 Modbus RTU 等工业协议

2. **全双工（Full-Duplex）**

- 使用两对差分线（A/B + A’ /B’）同时收发
- 提高通信效率
- 对 PCB 布线要求更高

3. **点对点和多点通信**

- 点对点：两节点通信，简单可靠
- 多点（多驱动器/多接收器）：需总线仲裁，避免冲突

4. 信号特性

- **逻辑电平**：
- 逻辑 “1”（标记电平）：A < B
- 逻辑 “0”（空闲电平）：A > B
- **抗干扰能力**
- 差分信号可抑制共模干扰
- 常用于工业环境中的长距离传输
- **波特率与距离**：

波特率	最大传输距离
9600 bps	1200 m
115200 bps	100 m

- 波特率越高，传输距离越短

5. 优势与限制

优势：

- 抗干扰能力强
- 支持长距离和多节点
- 总线成本低，布线简单

限制：

- 半双工总线通信需软件或硬件管理冲突
- 高速通信距离受限
- 需要终端电阻匹配

RA8 系列 UART 模块概述

RA8 系列 MCU 内置高性能 UART 外设，支持多种通信模式和波特率，能够满足 RS485 半双工通信需求。

1. UART 总体特性

- **通信模式**：标准 UART 异步串行通信
- **数据位长度**：5~9 位可选
- **停止位**：1、1.5、2 位可选
- **奇偶校验**：支持无、偶、奇校验
- **波特率**：支持 300 bps ~ 12 Mbps，部分型号可更高
- **FIFO 支持**：发送/接收 FIFO 缓冲，减轻 CPU 负担
- **DMA 支持**：TX/RX 可通过 DMA 提高数据吞吐
- **中断事件**：传输完成、中断触发、FIFO 阀值、错误检测（帧错误、溢出、奇偶校验错误）

2. RS485 相关功能

RA8 UART 模块对 RS485 支持如下：

- **DE（Driver Enable）自动控制**
- 当发送数据时自动拉高 DE
- 发送完成后自动拉低 DE
- **半双工模式**
- 单线收发器，通过方向控制实现收发切换
- **地址检测功能（可选）**
- 支持多机通讯的地址匹配

3. UART 架构与工作原理

1. **发送/接收 FIFO**

- 提供独立的 TX/RX 缓冲区
- 支持 FIFO 阈值中断，提高连续收发效率

2. **波特率生成器**

- 根据 PCLK 和分频寄存器生成所需波特率
- 支持标准波特率和非标准波特率

3. **中断与事件处理**

- TX 空中断：发送缓冲区空
- RX 满中断：接收缓冲区满
- 错误中断：帧错误、溢出、奇偶校验错误
- 发送完成中断：可用于 RS485 DE 自动控制

RT-Thread UART 驱动框架

RT-Thread 提供统一的 **串口驱动框架**，支持标准 UART 和 RS485。

主要接口

函数/宏	功能
`rt_device_find("uartX")`	查找 UART 设备句柄

函数/宏	功能
`rt_device_open(dev, flags)`	打开设备，初始化硬件
`rt_device_control(dev, cmd, args)`	控制 UART，例如设置波特率、模式、RS485 方向
`rt_device_write(dev, pos, buffer, size)`	发送数据
`rt_device_read(dev, pos, buffer, size)`	接收数据
`rt_device_close(dev)`	关闭 UART

硬件说明

FSP 配置

- 打开 FSP 工具，新建 `r_sci_b_uart` stack：

- 配置 `r_sci_b_uart` stack：

- 配置 `r_sci_b_uart` 引脚：

RT-Thread Settings 配置

- 使能并配置 RS485。

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与PC 机连接，然后将固件下载至开发板。

运行效果

在终端中输入 `rs485_cmd -t RT-Thread` 命令发送 “RT-Thread” 字符串。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Aug 15 2025 11:54:32
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
===================================================
This example project is an rs485 routine!
===================================================
msh >rs485_cmd -t RT-Thread
[I/drv.rs485] send==>>
[I/drv.rs485]    0:   R
[I/drv.rs485]    1:   T
[I/drv.rs485]    2:   -
[I/drv.rs485]    3:   T
[I/drv.rs485]    4:   h
[I/drv.rs485]    5:   r
[I/drv.rs485]    6:   e
[I/drv.rs485]    7:   a
[I/drv.rs485]    8:   d
msh >
```

将 Titan Board 的 RS485 接口与另一块开发板的 RS485 接口连接，使用另一块开发板不断发送数据，终端会输出 Titan Board 接收到的数据。

```
===================================================
This example project is an rs485 routine!
===================================================
msh >[I/drv.rs485] recv data:R
[I/drv.rs485] recv data:T
[I/drv.rs485] recv data:-
[I/drv.rs485] recv data:T
[I/drv.rs485] recv data:h
[I/drv.rs485] recv data:r
[I/drv.rs485] recv data:e
[I/drv.rs485] recv data:a
[I/drv.rs485] recv data:d
[I/drv.rs485] recv data:R
[I/drv.rs485] recv data:T
[I/drv.rs485] recv data:-
[I/drv.rs485] recv data:T
[I/drv.rs485] recv data:h
[I/drv.rs485] recv data:r
[I/drv.rs485] recv data:e
[I/drv.rs485] recv data:a
[I/drv.rs485] recv data:d
```

3.9. RTC 及 alarm 使用例程说明

中文|English

简介

本例程主要介绍了如何在 Titan Board 上使用 **RA8 系列 MCU 的 RTC 模块**，结合 **RT-Thread RTC 驱动框架**，实现基本的实时时钟功能及闹钟功能。RTC 可以提供精确的实时时间，它可以用于产生年、月、日、时、分、秒等信息。目前实时时钟芯片大多采用精度较高的晶体振荡器作为时钟源。有些时钟芯片为了在主电源掉电时还可以工作，会外加电池供电，使时间信息一直保持有效。

RT-Thread 的 RTC 设备为操作系统的时间系统提供了基础服务。面对越来越多的 IoT 场景，RTC 已经成为产品的标配，甚至在诸如 SSL 的安全传输过程中，RTC 已经成为不可或缺的部分。

RTC（实时时钟）简介

1. 概述

RTC（Real-Time Clock） 是一种用于 **计量和跟踪实际时间（年、月、日、时、分、秒）** 的电子模块或芯片。它通常内置在 **微控制器（MCU）** 或作为独立芯片存在，用于提供 **系统时间**、**定时唤醒**、**定时事件触发** 等功能。

RTC 的核心特点是 **低功耗、长时间稳定运行**，即使系统主电源断电，也可通过备用电源（如电池或超级电容）继续计时。

2. 工作原理

RTC 基本上是一个 **低功耗振荡器 + 计数器** 系统：

1. **时钟源**

- RTC 通常使用 **32.768 kHz 石英晶振**（1 Hz = 2^15 周期）
- 提供稳定时基

2. **分频计数**

- 将晶振信号分频得到 1 Hz 的秒脉冲
- 累计计数生成分钟、小时、天、月、年

3. **寄存器存储**

- RTC 内部寄存器保存当前时间、日期、闹钟设置等

4. **电源冗余**

- 内部电池或超级电容供电，使 RTC 在主电源断电时继续运行

3. RTC 类型

1. **内部 RTC**

- 集成在 MCU 内部
- 优点：节省芯片数量，成本低
- 缺点：晶振精度受 PCB 布线、温度影响

2. **外部 RTC 芯片**

- 独立芯片，如 DS3231、PCF8563
- 优点：高精度、可用 I^2C/SPI 接口连接 MCU
- 缺点：增加 PCB 封装面积和成本

4. 关键参数

参数	描述
振荡器频率	通常 32.768 kHz，低功耗稳定
时间精度	ppm / 秒/天，决定时钟漂移
电源电压	1.8~5V，支持备用电源
功耗	1~5 μA（低功耗模式）
接口类型	I^2C、SPI 或 MCU 内部总线
功能扩展	闹钟输出、方波输出、温度补偿、定时唤醒

5. RTC 功能

1. **实时时钟**

- 提供当前时间和日期

2. **闹钟功能（Alarm）**

- 可设置指定时间触发中断或唤醒 MCU

3. **定时唤醒（Wake-up Timer）**

- MCU 进入低功耗模式时，可由 RTC 定时唤醒

4. **温度补偿**（部分高精度芯片，如 DS3231）

- 自动校正晶振频率，减少漂移

5. **方波输出（Square Wave Output）**

- 提供固定频率脉冲，用于驱动 MCU 定时器或其他模块

6. RTC 应用场景

- **嵌入式设备**
- MCU 实时时间追踪，事件记录
- **低功耗物联网设备**
- RTC 唤醒 MCU 进行周期性数据采集
- **手表与智能穿戴设备**
- 提供精准计时功能
- **数据记录仪与工业控制**
- 时间戳、日志记录
- **汽车电子**
- 行车记录仪、车载娱乐系统计时

RA8 系列 RTC 模块概述

RA8 系列 MCU 内置高度灵活的 **实时时钟（RTC）** 外设，适用于低功耗、高精度的嵌入式时间管理应用。

1. 总体特性

- **时间管理**：支持秒、分、时、日、月、年及星期。
- **时钟源**：
- **外部低速晶振（通常为 32.768 kHz）**，高精度。
- **内部低速 RC 振荡器**，低功耗，精度稍低。
- **闹钟功能**：支持单次和重复闹钟，可生成中断。
- **定时器**：子秒定时器、周期性唤醒定时器。
- **防篡改检测（可选）**：可在配置的引脚检测外部篡改事件。
- **低功耗操作**：RTC 可在待机或睡眠模式下运行，确保 MCU 断电时仍保持时间。

2. RTC 模块架构

RA8 RTC 模块包含以下功能模块：

1. **时间计数器（Calendar Counter）**

- 维护当前时间和日期。
- 支持 24 小时或 12 小时制。
- 自动处理闰年调整。

2. **闹钟与比较单元**

- 将当前时间与预设闹钟值比较。
- 匹配时生成中断。
- 支持重复闹钟（每日、每小时等）。

3. **时钟源与分频器**

- 将外部 32.768 kHz 或内部 RC 振荡器分频生成 1 Hz 时基。
- 可选择时钟精度与功耗的平衡。

4. **中断控制器**

- 支持闹钟中断、周期中断、溢出中断。
- 可配置中断屏蔽、使能及状态标志。

5. **防篡改及备份寄存器（可选）**

- 非易失性备份寄存器，可存储断电保留数据。
- 防篡改引脚可触发中断，实现安全应用。

3. 关键特性

特性	描述
时间分辨率	秒级，可选子秒计数
时间范围	年、月、日、时、分、秒
闹钟	可配置闹钟并触发中断
时钟源	外部 32.768 kHz 晶振，内部低速 RC 振荡器
功耗	超低功耗，支持待机/睡眠模式

特性	描述
中断	闹钟、周期性、溢出
备份寄存器	非易失性寄存器，可在断电期间保存数据
防篡改检测（可选）	外部引脚事件可触发中断

4. 工作原理

1. **选择时钟源**：外部晶振或内部 RC 振荡器。
2. **设置时间**：将初始时间和日期写入日历计数器。
3. **设置闹钟**：配置闹钟时间并使能闹钟中断。
4. **中断处理**：MCU 可通过 RTC 中断标志响应闹钟或周期性事件。
5. **电源管理**：RTC 在低功耗模式下继续计时，确保时间准确。

RT-Thread RTC 驱动框架

RT-Thread 提供统一的 **RTC 驱动框架**，抽象底层硬件接口，提供标准化 API 以支持时间管理和闹钟功能。

1. 主要接口

函数 / 宏	功能
`rt_device_find("rtc")`	获取 RTC 设备句柄
`rt_device_open(dev, flags)`	打开 RTC 设备并初始化硬件
`rt_device_control(dev, cmd, args)`	配置 RTC：设置/获取时间，设置闹钟等
`rt_device_read(dev, pos, buffer, size)`	读取当前时间或闹钟状态
`rt_device_write(dev, pos, buffer, size)`	写入时间或闹钟值
`rt_device_close(dev)`	关闭 RTC 设备

---

2. `rt_device_control` 常用命令

命令	描述
`RT_DEVICE_CTRL_RTC_SET_TIME`	设置当前 RTC 时间
`RT_DEVICE_CTRL_RTC_GET_TIME`	获取当前 RTC 时间
`RT_DEVICE_CTRL_RTC_SET_ALARM`	设置闹钟时间并使能闹钟中断
`RT_DEVICE_CTRL_RTC_GET_ALARM`	获取已配置的闹钟时间
`RT_DEVICE_CTRL_RTC_ENABLE`	使能 RTC
`RT_DEVICE_CTRL_RTC_DISABLE`	禁用 RTC

---

3. 闹钟处理

- 通过 `RT_DEVICE_CTRL_RTC_SET_ALARM` 配置闹钟。
- 使能中断，当闹钟触发时 MCU 收到通知。
- 中断服务例程可更新标志、触发事件或唤醒低功耗 MCU。

硬件说明

本例程使用的RTC 设备依赖于LOCO 时钟，此外无需过多连接。

FSP 配置

- 打开 FSP，选择对应的工程文件下的 configuration.xml，新增 RTC Stack;

- 下面进行 RTC 参数的配置，设置 rtc stack name 为 g_rtc，设置 RTC 中断回调函数为 rtc_callback，并配置中断回调优先级；

RT-Thread Settings 配置

- 打开 RT-Thread Settings，找到硬件选项，使能 RTC；

- 接下来我们配置 RTC，首先需要使能 RT-Thread 的 RTC 设备框架，同时使能 alarm 功能；

示例代码说明

本例程的源码位于 `/src/hal_entry.c`。

```
    rt_err_t ret = RT_EOK;
    time_t now;
    rt_device_t device = RT_NULL;

    device = rt_device_find(RTC_NAME);
    if (!device)
    {
      rt_kprintf("find %s failed!\n", RTC_NAME);
    }

    if(rt_device_open(device, 0) != RT_EOK)
    {
      rt_kprintf("open %s failed!\n", RTC_NAME);
```

```
    }

    /* 设置日期 */
    ret = set_date(2025, 8, 1);
    rt_kprintf("set RTC date to 2025-8-1\n");
    if (ret != RT_EOK)
    {
        rt_kprintf("set RTC date failed\n");
    }

    /* 设置时间 */
    ret = set_time(15, 00, 00);
    if (ret != RT_EOK)
    {
        rt_kprintf("set RTC time failed\n");
    }

    /* 延时3秒 */
    rt_thread_mdelay(3000);

    /* 获取时间 */
    get_timestamp(&now);
    rt_kprintf("now: %.*s", 25, ctime(&now));
```

下面代码可创建一个 RTC 闹钟，然后设置 1 秒后唤醒，最后把该函数导入 msh 命令行中。

```
void user_alarm_callback(rt_alarm_t alarm, time_t timestamp)
{
    LOG_I("user alarm callback function.");
}

void alarm_sample(void)
{
    rt_device_t dev = rt_device_find("rtc");
    struct rt_alarm_setup setup;
    struct rt_alarm * alarm = RT_NULL;
    static time_t now;
    struct tm p_tm;

    if (alarm != RT_NULL)
        return;

    /* 获取当前时间戳，并把下一秒时间设置为闹钟时间 */
    now = get_timestamp(NULL) + 1;
    gmtime_r(&now,&p_tm);

    setup.flag = RT_ALARM_SECOND;
    setup.wktime.tm_year = p_tm.tm_year;
    setup.wktime.tm_mon = p_tm.tm_mon;
    setup.wktime.tm_mday = p_tm.tm_mday;
```

```
    setup.wktime.tm_wday = p_tm.tm_wday;
    setup.wktime.tm_hour = p_tm.tm_hour;
    setup.wktime.tm_min = p_tm.tm_min;
    setup.wktime.tm_sec = p_tm.tm_sec;

    alarm = rt_alarm_create(user_alarm_callback, &setup);
    if(RT_NULL != alarm)
    {
        rt_alarm_start(alarm);
    }
}
/* export msh cmd */
MSH_CMD_EXPORT(alarm_sample,alarm sample);
```

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

按下复位按键重启开发板，输入 `alarm_sample` 命令运行闹钟示例，可以看到板子上会打印如下信息：

拓展：RTC 掉电运行

当板卡主电源断开后，如需让 RTC 掉电继续运行，需要通过 VBAT 引脚接上外部供电，需要注意的是，当 VBAT 给 RTC 供电时，只会继续 Calendar 计数器的计时，其

他 RTC 的功能无法使用，下面是一个参考代码：

ps: rtc 如果打算掉电运行，那么应该确保 rtc_init 只初始化一次，后续可通过 rtc_get 获取时间

```
#define RTC_YEAR_SET 2025
#define RTC_MON_SET  8
#define RTC_MDAY_SET 5

#define RTC_WDAY_SET (RTC_YEAR_SET-2000 \
                  + ((RTC_YEAR_SET-2000)/4) \
                  - 35 + (26*(RTC_MON_SET+1))/10 \
                  + RTC_MDAY_SET -1 )%7

#define RTC_HOUR_SET 12
#define RTC_SEC_SET 0
#define RTC_MIN_SET 0

void rtc_get()
{
    rtc_time_t get_time;
    R_RTC_CalendarTimeGet(g_rtc.p_ctrl, &get_time);
    rt_kprintf ("%d-%d-%d-%d:%d:%d\r\n", get_time.tm_year + 1900, g
             get_time.tm_hour, get_time.tm_min, get_time.tm_sec);
}
MSH_CMD_EXPORT(rtc_get, rtc_get);

void rtc_init(void)
{
   rtc_time_t set_time =
   {
     .tm_min = RTC_MIN_SET,
     .tm_hour = RTC_HOUR_SET,
     .tm_mday = RTC_MDAY_SET,
     .tm_wday = RTC_WDAY_SET,
     .tm_mon = RTC_MON_SET - 1,
     .tm_year = RTC_YEAR_SET-1900,
   };

   R_RTC_Open (g_rtc.p_ctrl, g_rtc.p_cfg);

   R_RTC_CalendarTimeSet (g_rtc.p_ctrl, &set_time);
}
MSH_CMD_EXPORT(rtc_init, rtc_init);
```

FSP配置：

Threads
New Thread
Remove
HAL/Common
g_ioport I/O Port (r_ioport)
g_uart8 UART (r_sci_b_uart)
g_rtc Realtime Clock (r_rtc)
1
Objects
New Object >
Remove
g_rtc Realtime Clock (r_rtc) Stacks
g_rtc Realtime Clock (r_rtc)
Summary
BSP
Clocks
Pins
Interrupts
Event Links
Stacks
Components
Properties
Problems
g_rtc Realtime Clock (r_rtc)
Settings
API Info
Property
Value
Common
Parameter Checking
Default (BSP)
Set Source Clock in Open
Enabled
Module g_rtc Realtime Clock (r_rtc)
Name
g_rtc
Clock Source
Sub-Clock
2
Frequency Comparison Value (LOCO)
255
Automatic Adjustment Mode
Enabled
Automatic Adjustment Period
10 Seconds
Adjustment Type (Plus-Minus)
NONE
Error Adjustment Value
0
Callback
NULL
Alarm Interrupt Priority
Disabled
Period Interrupt Priority
Disabled
Carry Interrupt Priority
Priority 12
Pins
RTCIC0
None
RTCIC1
None
RTCIC2
None
RTCOUT
None

Pin Selection
Type filter text
HMI:GLCDC
HMI:MIPI
Interrupt:IRQ
System:CGC
TRG:ADC(Digital)
TRG:CAC
Timers:AGT
Timers:GPT
Timers:GPT_OPS
Timers:GPT_POEG
Timers:RTC
RTC
Timers:ULPT
Pin Configuration
Name
Value
Lock
Link
Pin Group Selection
Mixed
Operation Mode
Custom
Input/Output
RTCIC0
None
RTCIC1
None
RTCIC2
None
RTCOUT
None
Module name: RTC
Usage: When using RTC on P402, P403, P404, Please refer to t

3.10. SD卡文件系统使用说明

中文|English

简介

本例程使用 SD 卡作为文件系统的存储设备，展示如何在 SD 卡上创建文件系统（格式化卡），并挂载文件系统到 RT-Thread 操作系统中。

文件系统挂载成功后，展示如何使用文件系统提供的功能对目录和文件进行操作。

SD 卡简介

1. 概述

SD 卡（Secure Digital Card） 是一种小型、便携的非易失性存储设备，广泛用于 **嵌入式系统、相机、手机、数据记录仪** 等场景。

SD 卡由 **控制器 + NAND Flash 存储芯片** 组成，外部通过标准接口与主机通信。

主要特点：

- 小巧轻便，体积通常为 **32 × 24 × 2.1 mm**（标准卡）
- 采用 **非易失性闪存（NAND Flash）** 存储数据
- 支持热插拔和掉电保护

2. SD 卡类型

1. **按尺寸分类**

- **标准 SD 卡（Standard SD）**：32 × 24 mm
- **Mini SD**：21.5 × 20 mm
- **Micro SD（TF 卡）**：15 × 11 mm，最常用

2. **按容量分类**

类型	容量范围
SDSC	1 MB ~ 2 GB
SDHC	4 GB ~ 32 GB

类型	容量范围
SDXC	32 GB ~ 2 TB
SDUC	2 TB ~ 128 TB

3. **按速度等级**

- **Class 2/4/6/10**：最低写入速度分别为 2、4、6、10 MB/s
- **UHS（Ultra High Speed）**：UHS-I/UHS-II/UHS-III，速率可达 **312 MB/s**
- **Video Speed Class（V6/V10/V30/V60/V90）**：适用于高清视频录制

3. SD 卡接口

1. **SPI 模式**

- 使用 SPI 总线（MISO, MOSI, SCK, CS）
- 简单易用，适合 MCU
- 数据传输速率较低

2. **SD 模式（1-bit / 4-bit）**

- 使用专用 SD 总线
- 支持 1-bit 或 4-bit 数据线
- 速率高于 SPI 模式

3. **UHS 模式**

- 支持高速数据传输，常用于摄像机和高性能嵌入式应用

4. 工作原理

1. **命令/数据传输**

- 主机通过 SD 卡协议发送命令（CMD）
- 卡片返回响应（R1, R2 等）
- 读写数据块（Block），每块通常为 **512 Byte**

2. **控制器管理**

- 内部控制器负责 **坏块管理、错误纠正（ECC）、逻辑到物理地址映射**
- 外部主机无需直接管理 NAND Flash 特性

3. **数据存储**

- 数据存储在 NAND Flash 中
- 支持多次擦写和擦写寿命管理（典型寿命 10 万次擦写）

5. SD 卡性能指标

参数	描述
容量	1 GB ~ 128 TB
数据块大小	512 Byte（标准）
接口速率	SPI/SD 1-bit/4-bit/UHS
最大传输速度	25 MB/s（标准），312 MB/s（UHS-III）
工作电压	3.3 V（部分 Micro SD 支持 1.8 V）
工作温度	-25 °C ~ 85 °C（工业级）
耐用性	擦写次数 10^4 ~ 10^5

6. SD 卡应用场景

1. **消费电子**

- 手机、平板、数码相机、摄像机存储

2. **嵌入式系统**

- MCU/FPGA 数据存储
- 日志记录、配置文件保存

3. **工业应用**

- 工业控制器、数据采集系统
- 高温环境工业级 SD 卡可使用

4. **音视频应用**

- 高速视频录制（UHS/V Class）

5. **汽车电子**

- 行车记录仪、导航系统数据存储

RA8 系列 SDHI 模块概述

RA8 系列 MCU 内置高性能 SDHI 模块，专门用于与 SD/SDHC/SDXC 卡高速通信，支持 SPI 模式和 SD/SDIO 模式。

1. 总体特性

- **支持 SD 标准**
- SD v1.x / v2.x / SDHC / SDXC
- 支持 SPI 模式和 SD/MMC 模式
- **高速数据传输**
- 最高可达 50 MHz SDCLK（具体取决于 MCU 时钟配置）
- 支持 1-bit/4-bit 数据总线
- **自动命令和数据传输**
- 支持 DMA 传输模式，减少 CPU 占用
- 自动命令序列生成（CMD0~CMD59）
- **错误检测**
- CRC7 校验命令，CRC16 校验数据
- 超时检测，响应错误识别
- **中断支持**
- 卡插拔检测中断
- 命令完成中断
- 数据传输完成中断
- 错误中断

2. SDHI 模块架构

RA8 SDHI 模块主要包含以下子模块：

1. **命令控制单元（Command Control Unit）**

- 负责发送 SD 命令（CMD0~CMD59）
- 处理命令响应（R1、R2、R3、R7 等）
- 支持命令超时检测和 CRC 校验

2. **数据传输单元（Data Transfer Unit）**

- 通过内部 FIFO 或 DMA 实现数据收发

- 支持块读/写，最大 512 字节块
- 支持单块/多块传输模式

3. **时钟与总线控制**

- SDCLK 生成和分频
- 1-bit 或 4-bit 总线切换
- 可配置高/低电平保持时间

4. **卡检测与电源控制**

- 检测 SD 卡插入/拔出状态
- 可控制卡片电源开关（如支持）

5. **中断与事件控制单元**

- 命令完成中断
- 数据传输完成中断
- 错误中断
- 卡插拔中断

3. SDHI 工作原理

1. **初始化阶段**

- 检测 SD 卡插入
- 发送 CMD0、CMD8 初始化卡
- 查询卡容量与版本信息

2. **命令发送**

- Host 向卡发送命令
- Card 返回响应，SDHI 模块验证 CRC 并触发中断

3. **数据传输**

- 读/写数据块时，通过 FIFO 或 DMA 进行高速传输
- 支持单块或多块操作

4. **错误处理**

- 超时、CRC 错误、响应错误等
- SDHI 模块可触发错误中断，由驱动进行重试或异常处理

硬件说明

本次示例和存储器连接通过 SDIO 接口，使用的是硬件的 SD0，原理图如下所示：

FSP 配置

- 新建 stacks 选择 r_sdhi 并配置 sdhi0 配置信息如下：

RT-Thread Settings 配置

- 在配置中使能 SD 卡文件系统。

- 使能 SDHI0 并将 SDHI0 的 Bus Width 设置为 1。

示例工程说明

本例程的文件系统初始化源码位于 `./board/ports/drv_filesystem.c` 中：

```
#include <rtthread.h>

#if defined(BSP_USING_FILESYSTEM)
#include <dfs_romfs.h>
#include <dfs_fs.h>
#include <dfs_file.h>

#if DFS_FILESYSTEMS_MAX < 4
#error "Please define DFS_FILESYSTEMS_MAX more than 4"
#endif
#if DFS_FILESYSTEM_TYPES_MAX < 4
#error "Please define DFS_FILESYSTEM_TYPES_MAX more than 4"
#endif

#define DBG_TAG "app.filesystem"
#define DBG_LVL DBG_INFO
#include <rtdbg.h>

#ifdef BSP_USING_FS_AUTO_MOUNT
#ifdef BSP_USING_SDCARD_FATFS
static int onboard_sdcard_mount(void)
{
    if (dfs_mount("sd", "/sdcard", "elm", 0, 0) == RT_EOK)
    {
        LOG_I("SD card mount to '/sdcard'");
    }
    else
    {
        LOG_E("SD card mount to '/sdcard' failed!");
        rt_pin_write(0x000D, PIN_LOW);
    }
```

```
    return RT_EOK;
}
#endif /* BSP_USING_SDCARD_FATFS */
#endif /* BSP_USING_FS_AUTO_MOUNT */

#ifdef BSP_USING_FLASH_FS_AUTO_MOUNT
#ifdef BSP_USING_FLASH_FATFS
#define FS_PARTITION_NAME "filesystem"

static int onboard_fal_mount(void)
{
    /* 初始化 fal 功能 */
    extern int fal_init(void);
    extern struct rt_device* fal_mtd_nor_device_create(const char *
    fal_init ();
    /* 在 ospi flash 中名为 "filesystem" 的分区上创建一个块设备 */
    struct rt_device *mtd_dev = fal_mtd_nor_device_create (FS_PARTI
    if (mtd_dev == NULL)
    {
        LOG_E("Can't create a mtd device on '%s' partition.", FS_PA
        return -RT_ERROR;
    }
    else
    {
        LOG_D("Create a mtd device on the %s partition of flash suc
    }

    /* 挂载 ospi flash 中名为 "filesystem" 的分区上的文件系统 */
    if (dfs_mount (FS_PARTITION_NAME, "/fal", "lfs", 0, 0) == 0)
    {
        LOG_I("Filesystem initialized!");
    }
    else
    {
        dfs_mkfs ("lfs", FS_PARTITION_NAME);
        if (dfs_mount ("filesystem", "/fal", "lfs", 0, 0) == 0)
        {
            LOG_I("Filesystem initialized!");
        }
        else
        {
            LOG_E("Failed to initialize filesystem!");
            rt_pin_write(0x000D, PIN_LOW);
        }
    }

    return RT_EOK;
}
#endif /*BSP_USING_FLASH_FATFS*/
#endif /*BSP_USING_FLASH_FS_AUTO_MOUNT*/
```

```
const struct romfs_dirent _romfs_root[] =
{
#ifdef BSP_USING_SDCARD_FATFS
    {ROMFS_DIRENT_DIR, "sdcard", RT_NULL, 0},
#endif

#ifdef BSP_USING_FLASH_FATFS
  { ROMFS_DIRENT_DIR, "fal", RT_NULL, 0 },
#endif
        };

const struct romfs_dirent romfs_root =
{
ROMFS_DIRENT_DIR, "/", (rt_uint8_t*) _romfs_root, sizeof(_romfs_roo
};

static int filesystem_mount(void)
{

#ifdef RT_USING_DFS_ROMFS
    if (dfs_mount(RT_NULL, "/", "rom", 0, &(romfs_root)) != 0)
    {
        LOG_E("rom mount to '/' failed!");
    }

    /* 确保块设备注册成功之后再挂载文件系统 */
    rt_thread_delay(500);
#endif
#ifdef BSP_USING_FS_AUTO_MOUNT
    onboard_sdcard_mount();
#endif /* BSP_USING_FS_AUTO_MOUNT */

#ifdef BSP_USING_FLASH_FS_AUTO_MOUNT
    onboard_fal_mount ();
#endif

    return RT_EOK;
}
INIT_COMPONENT_EXPORT(filesystem_mount);
#endif /* defined(BSP_USING_FILESYSTEM)*/
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与PC 机连接，然后将固件下载至开发板。

运行效果

按下复位按键重启开发板，等待 SD 挂载后进入 SD 卡文件系统目录查看 SD 卡上的文件。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 28 2025 18:20:14
 2006 - 2024 Copyright by RT-Thread team
msh />[I/SDIO] SD card capacity 3932160 KB.
[I/SDIO] sd: switch to High Speed / SDR25 mode

found part[0], begin: 65536, size: 3.766GB
[I/app.filesystem] SD card mount to '/sdcard'

Hello RT-Thread!
==================================================
This example project is the sdio routine!
==================================================

msh />cd sdcard
msh /sdcard>ls
Directory /sdcard:
System Volume Inform<DIR>
TEST                <DIR>
C.TXT               5
hello.txt           <DIR>
test.txt            8
Song.wav            44982044
1.wav               0
tt.wav              50732
111.wav             0
2.wav               20268
SeeYouAgain.wav     40488498
msh /sdcard>
```

3.11. BMI088 陀螺仪使用说明

中文 | English

简介

本示例展示了如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 SPI 模块**，结合 **RT-Thread SPI 驱动框架** 与 **BMI088 软件包（六轴 IMU：三轴加速度计 + 三轴陀螺仪）** 完成传感器初始化与数据读取。

BMI088 简介

1. 概述

BMI088 是 Bosch Sensortec 推出的 **高性能 6 轴 IMU（Inertial Measurement Unit，惯性测量单元）**，集成了 **3 轴加速度计** 和 **3 轴陀螺仪**。

它专为 **无人机、机器人、工业自动化、导航系统** 等需要高稳定性和低噪声的场景设计。

BMI088 以 **高抗振性** 和 **低噪声性能** 著称，相比常见消费级 IMU（如 MPU6050、BMI160 等），在工业和无人机领域应用更广泛。

2. 核心特性

(1) 加速度计（Accelerometer）

- 测量范围：±3 g / ±6 g / ±12 g / ±24 g
- 噪声密度：**约 120 μg/√Hz**
- 采样率：最高 **1.6 kHz**
- 零点偏移（Offset）：出厂已校准
- 特点：高稳定性、低噪声、宽动态范围

(2) 陀螺仪（Gyroscope）

- 测量范围：±125 °/s、±250 °/s、±500 °/s、±1000 °/s、±2000 °/s
- 噪声密度：**约 0.014 °/s/√Hz**
- 采样率：最高 **2 kHz**
- 内置 **抗震设计**，减少高频振动影响

- 零偏稳定性好，适合长期运行

(3) 接口与电源

- **通信接口**：SPI（最高 10 MHz）、I^2C（最高 400 kHz）
- 工作电压：1.71V ~ 3.6V
- 封装：3 mm × 4.5 mm × 0.95 mm（LGA-16）

3. 工作原理

- **加速度计**：通过检测 MEMS 结构在三轴方向的加速度，输出对应的数字信号。
- **陀螺仪**：基于科里奥利效应测量角速度，提供三轴旋转运动信息。
- **数据融合**：结合加速度和陀螺仪数据，可实现姿态解算（Pitch、Roll、Yaw）。

4. 主要优势

1. **抗振动能力强**

- 陀螺仪内部有特殊抗震结构，适合无人机等强振动环境。

2. **低噪声性能**

- 加速度计和陀螺仪均具备低噪声输出，保证姿态解算的精度。

3. **工业级可靠性**

- 比消费级 IMU（如 BMI160、MPU6050）更稳定，温漂小。

4. **高带宽**

- 陀螺仪带宽最高可达 2 kHz，适合高速控制场景。

5. 应用场景

- **无人机**：飞行姿态检测、导航、稳定控制
- **机器人**：运动控制、SLAM、平衡控制
- **工业测量**：平台稳定、机械臂运动检测
- **车载系统**：惯性导航、辅助定位
- **运动追踪**：VR/AR、智能穿戴设备

RA8 系列 SPI 模块（r_spi_b）概述

RA8 系列 MCU 内置 **增强型 SPI 模块（r_spi_b）**，可用于与各类外设进行高速串行通信，如 Flash、传感器、显示器、音频编解码器等。该模块支持主机模式和从机模式，提供全双工通信能力，并在硬件层面集成了丰富的缓冲、时钟和 DMA 控制机制，能够有效降低 CPU 负担。

1. 基本特性

- **工作模式**
- 支持 **主机模式（Master）** 和 **从机模式（Slave）**
- 支持 **全双工通信**（同时发送与接收）
- 支持 **半双工模式**，适合某些单向传输的外设
- **时钟特性**
- 支持 **高达 166 MHz** 的串行时钟（SPCLK，取决于系统时钟和芯片规格）
- 可配置的 **极性（CPOL）** 和 **相位（CPHA）**，支持标准 **SPI Mode 0/1/2/3**
- **数据格式**
- 支持 8 位、16 位、32 位帧格式
- 支持 **MSB-first** 和 **LSB-first** 传输
- **片选管理**
- 支持 **硬件片选（SSL0~SSL3）**
- 也可使用 **GPIO 软件片选**
- **缓冲与 DMA**
- 内部具备 **FIFO 缓冲区**，可减少中断频率
- 支持 **DTC/DMA** 自动传输，适合大数据量通信
- **中断与事件**
- 发送完成中断
- 接收完成中断
- 传输错误中断（如溢出、欠载）

2. SPI 模块架构

RA8 系列 SPI（r_spi_b）内部主要由以下子模块组成：

1. **时钟控制单元（Clock Control Unit）**

- 控制串行时钟 SPCLK 的生成
- 可选择分频因子，调整通信速率
- 提供极性（CPOL）和相位（CPHA）配置

2. **传输控制单元（Transfer Control Unit）**

- 控制数据帧的长度（8/16/32 bit）
- 配置数据顺序（MSB/LSB）
- 控制发送/接收方向

3. **FIFO 缓冲区**

- 发送缓冲 FIFO
- 接收缓冲 FIFO
- 有效降低中断处理频率，提高系统性能

4. **片选控制单元（SSL Control Unit）**

- 提供 4 个独立片选信号（SSL0~SSL3）
- 自动拉低/释放片选信号
- 支持多从设备应用场景

5. **DMA/DTC 支持**

- SPI 模块可直接与 DMA/DTC 配合
- 实现 **无 CPU 干预的数据流式传输**
- 特别适用于传感器数据采集、外部存储器访问等

6. **中断控制**

- 发送完成中断（TXI）
- 接收完成中断（RXI）
- 传输结束中断（TEI）
- 错误中断（ERI）

3. 工作原理

1. **主机模式下通信**

- 配置 SPCLK、CPOL、CPHA
- 配置片选信号 SSLx
- 写入数据到发送 FIFO
- SPI 硬件自动移位输出，同时接收数据

2. **从机模式下通信**

- 等待主机片选信号有效
- 接收来自主机的数据，同时可发送响应数据
- 支持 DMA，将接收数据写入内存

3. **FIFO 与 DMA 协同**

- 短数据传输：由中断驱动
- 大数据传输：由 DMA/DTC 自动完成，CPU 仅需配置起始参数

硬件说明

Titan Board 使用 SPI0 与 BMI088 陀螺仪通信。

FSP 配置

- 打开 FSP 工具，新建 Stacks 并选择 r_spi_b：

- 配置 SPI0 引脚：

RT-Thread Settings 配置

- 使能 bmi088：

- 在 bmi088 软件包配置中使能 sensor_v1 和 sample：

软件说明

在 Titan Board 上使用 BMI088 软件包需要做如下修改进行适配：

- 修改 `./packages/bmi088-latest/sample/bmi088_sample.c`：

将 SPI BUS 修改为 spi0，修改片选引脚：

RT-Thread Settings | drv_filesystem.c | bmi088_sample.c

```c
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date           Author       Notes
 * 2020-03-10     MyGuo        the first version
 */

#include <rtthread.h>
#include <rtdbg.h>
#include "bmi088.h"
#include "sensor_intf_bmi088.h"
#include "drv_spi.h"
#include "bmi088_sample.h"

#define BMI088_BUS_NAME "spi0"
#define BMI088A_SPI_NAME "spi00"
#define BMI088G_SPI_NAME "spi01"

static rt_device_t acce_device_t;
static rt_device_t gyro_device_t;
struct rt_sensor_data acc_test;
struct rt_sensor_data gyr_test;

void imu_thread_entry(void* parameter)
{
    // 配置spi,配置片选引脚(要在acc、gyr初始化之前配置,因为器件初始化中涉及到引脚操作)
    rt_hw_spi_device_attach(BMI088_BUS_NAME, BMI088A_SPI_NAME, BSP_IO_PORT_06_PIN_12);
    rt_hw_spi_device_attach(BMI088_BUS_NAME, BMI088G_SPI_NAME, BSP_IO_PORT_03_PIN_00);

    // 注册传感器
    struct rt_sensor_config acc_cfg = {0};
    struct rt_sensor_config gyr_cfg = {0};

    acc_cfg.intf.dev_name = BMI088A_SPI_NAME;
    gyr_cfg.intf.dev_name = BMI088G_SPI_NAME;
```

添加 imu 数据打印并创建数据采集线程：

```c
    gyro_device_t = rt_device_find("gyro_bmi");
    if (gyro_device_t == RT_NULL)
    {
        LOG_E("Can't find gyro device\r\n");
    }
    else
    {
        rt_device_open(gyro_device_t, RT_DEVICE_OFLAG_RDWR);
    }

    while (1)
    {
        _bmi088_get_data((rt_sensor_t) acce_device_t, &acc_test);
        _bmi088_get_data((rt_sensor_t) gyro_device_t, &gyr_test);

        rt_kprintf("acce value:[X]:%d [Y]:%d [Z]:%d\n", acc_test.data.acce.x, acc_test.data.acce.y, acc_test.data.acce.z);
        rt_kprintf("gyro value:[X]:%d [Y]:%d [Z]:%d\n\n", gyr_test.data.gyro.x, gyr_test.data.gyro.y, gyr_test.data.gyro.z);

        rt_thread_mdelay(500);
    }
}

void bmi088_app(void)
{
    rt_thread_t bmi088 = rt_thread_create("bmi088", imu_thread_entry, RT_NULL, 2048, 25, 10);
    if(bmi088 != RT_NULL)
    {
        rt_thread_startup(bmi088);
    }

    return;
}
MSH_CMD_EXPORT(bmi088_app, bmi088_app);
```

- 修改 `./packages/bmi088-latest/src/bmi088.c`

修改 CS 引脚适配瑞萨开发板：

RT-Thread Settings | drv_filesystem.c | bmi088_sample.c | bmi088.c

```c
/*
 * Copyright (c) 2006-2020, RT-Thread Development Team
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date           Author       Notes
 * 2020-02-28     MyGuo        the first version
 */

#include "bmi088.h"
#include <rtdbg.h>
#include <rtdevice.h>
#include <board.h>
#include "drv_spi.h"
#include <drv_gpio.h>

#define BMI088_SPI_MAX_SPEED (10 * 1000 * 1000) // M

#define CSB1_Pin      BSP_IO_PORT_06_PIN_12
#define CSB2_Pin      BSP_IO_PORT_07_PIN_04

static rt_err_t _bmi088_spi_read(struct rt_spi_device *dev, rt_uint8_t reg_addr, const rt_uint8_t len, rt_uint8_t *buf)
{
    reg_addr |= 0x80;

    dev->bus->owner = dev;
    rt_spi_send_then_recv(dev, &reg_addr, 1, buf, len);

    return RT_EOK;
}
```

BMI088 驱动示例程序位于 `./packages/bmi088-latest/samples/bmi088_sample.c`：

```c
#include <rtthread.h>
#include <rtdbg.h>
#include "bmi088.h"
#include "sensor_intf_bmi088.h"
#include "drv_spi.h"
#include "bmi088_sample.h"

#define BMI088_BUS_NAME "spi0"
#define BMI088A_SPI_NAME "spi00"
#define BMI088G_SPI_NAME "spi01"

static rt_device_t acce_device_t;
static rt_device_t gyro_device_t;
struct rt_sensor_data acc_test;
struct rt_sensor_data gyr_test;

#define BMI088A_PIN     BSP_IO_PORT_12_PIN_10
#define BMI088G_PIN     BSP_IO_PORT_12_PIN_14

void imu_thread_entry(void* parameter)
{
    // 配置spi,配置片选引脚(要在acc、gyr初始化之前配置,因为器件初始化中涉及到
    rt_hw_spi_device_attach(BMI088_BUS_NAME, BMI088A_SPI_NAME, 0x06
    rt_hw_spi_device_attach(BMI088_BUS_NAME, BMI088G_SPI_NAME, 0x07

    // 注册传感器
    struct rt_sensor_config acc_cfg = {0};
    struct rt_sensor_config gyr_cfg = {0};
```

```
    acc_cfg.intf.dev_name = BMI088A_SPI_NAME;
    gyr_cfg.intf.dev_name = BMI088G_SPI_NAME;

    rt_hw_bmi088_init("bmi", &acc_cfg, &gyr_cfg);

    acce_device_t = rt_device_find("acce_bmi");
    if (acce_device_t == RT_NULL)
    {
        LOG_E("Can't find acce device\r\n");
    }
    else
    {
        rt_device_open(acce_device_t, RT_DEVICE_OFLAG_RDWR);
    }

    gyro_device_t = rt_device_find("gyro_bmi");
    if (gyro_device_t == RT_NULL)
    {
        LOG_E("Can't find gyro device\r\n");
    }
    else
    {
        rt_device_open(gyro_device_t, RT_DEVICE_OFLAG_RDWR);
    }

    while (1)
    {
        _bmi088_get_data((rt_sensor_t) acce_device_t, &acc_test);
        _bmi088_get_data((rt_sensor_t) gyro_device_t, &gyr_test);

        rt_kprintf("acce value:[X]:%d [Y]:%d [Z]:%d\n", acc_test.da
        rt_kprintf("gyro value:[X]:%d [Y]:%d [Z]:%d\n\n", gyr_test.

        rt_thread_mdelay(500);
    }
}

void bmi088_app(void)
{
    rt_thread_t bmi088 = rt_thread_create("bmi088", imu_thread_entr
    if(bmi088 != RT_NULL)
    {
        rt_thread_startup(bmi088);
    }

    return;
}
MSH_CMD_EXPORT(bmi088_app, bmi088_app);
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

打开串口工具，在终端里输入 bmi088_app 指令获取陀螺仪数据：

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 28 2025 10:08:14
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
===================================================
This example project is an driver spi routine!
===================================================
[I/spi] Tips:
[I/spi] You can run spi testcase by executing the instruction: 'bmi088_app'
msh >bmi088_app
msh >[I/sensor] rt_sensor[acce_bmi] init success
[I/sensor] rt_sensor[gyro_bmi] init success
acce value:[X]:21 [Y]:52 [Z]:9625
gyro value:[X]:13988 [Y]:-24480 [Z]:13988

acce value:[X]:-32 [Y]:93 [Z]:9569
gyro value:[X]:-3497 [Y]:24480 [Z]:-3497

acce value:[X]:35 [Y]:78 [Z]:9670
gyro value:[X]:-20982 [Y]:-13988 [Z]:-20982

acce value:[X]:46 [Y]:68 [Z]:9655
gyro value:[X]:55954 [Y]:0 [Z]:55954

acce value:[X]:-32 [Y]:93 [Z]:9585
gyro value:[X]:-13988 [Y]:-10491 [Z]:-13988

acce value:[X]:43 [Y]:80 [Z]:9679
gyro value:[X]:-3497 [Y]:27977 [Z]:-3497

acce value:[X]:21 [Y]:130 [Z]:9605
gyro value:[X]:-20982 [Y]:-17485 [Z]:-20982

acce value:[X]:-17 [Y]:100 [Z]:9603
gyro value:[X]:-20982 [Y]:-48960 [Z]:-20982
```

3.12. USB-HHID 驱动例程

中文 | **English**

简介

本文档介绍在 **Titan Board** 上使用 **RA8 系列 MCU 的 USB Host HID（HHID）模块（r_usb_hhid）** 实现 USB 键盘识别与按键输入采集的完整示例。USB HID 是 Universal Serial Bus Human Interface Device 的缩写，由其名称可以了解 HID 设备是直接与人交互的设备，例如键盘、鼠标与游戏杆等。

USB HHID 简介

1. 概述

USB HID（Human Interface Device） 是 USB 协议中定义的一类设备，用于实现人与计算机之间的交互。

常见的 **HID 设备** 包括：

- 键盘（Keyboard）
- 鼠标（Mouse）
- 游戏手柄（Gamepad）
- 触摸屏（Touch Screen）
- 消费类控制器（音量调节键、媒体键）

USB HHID（Host HID） 通常指 **主机（Host）端对 HID 类设备的支持**，即 MCU 或嵌入式系统作为 USB 主机，能够识别、管理和通信 HID 设备。

2. HID 协议特点

- **即插即用（Plug and Play）**：无需额外驱动，主机操作系统（Windows、Linux、RTOS 等）一般自带 HID 类驱动。
- **基于报告描述符（Report Descriptor）**：设备通过 HID 报告描述符（HID Report Descriptor）告诉主机其数据格式。
- **低带宽占用**：典型使用 USB Interrupt Transfer（中断传输），数据量小、延迟低，适合人机交互。
- **跨平台支持**：标准化协议，广泛适用于 PC、嵌入式系统、移动设备。

3. 数据通信方式

HID 通信基于 **端点（Endpoint）**：

- **控制端点（Endpoint 0）**：用于设备枚举、描述符传输。
- **中断端点（Interrupt Endpoint）**：主要用于 HID 数据传输，典型的 Polling 方式。

HID 数据流：

1. **设备连接** → 主机通过 USB 控制传输读取设备描述符。
2. **解析 HID 报告描述符** → 获取数据格式定义。
3. **周期性轮询（Polling）** → 主机定期从 HID 设备获取输入报告（Input Report）。
4. **可选输出（Output Report）** → 主机向 HID 设备发送控制数据，例如 LED 灯状态。

4. HID 设备分类

- **输入设备**：键盘、鼠标、触摸板、手柄。
- **输出设备**：LED 灯、力反馈设备。
- **复合设备**：例如带按键的鼠标，同时有指针输入和按钮输入。

5. 应用场景

- **嵌入式系统调试**：MCU 作为 USB Host，支持接入键盘/鼠标，用于人机交互。
- **工业控制**：接入条码扫描枪、专用控制手柄，提高设备操作便利性。
- **医疗设备**：通过标准化接口接入输入装置。
- **消费电子**：智能电视、机顶盒通过 USB Host 接口支持遥控器或游戏手柄。

RA8 系列 USB HHID 模块简介

1. 模块定位与功能

- **USB Host Controller 支持**：支持 USB Host 模式（Full-Speed / High-Speed，取决 MCU 硬件），负责总线层枚举、USB 事务调度、端点管理、USB 供电控制等。
- **HHID 类驱动职责**：实现 USB HID class（0x03）中键盘（SubClass Boot、Protocol Keyboard）解析逻辑，完成：枚举后查找 IN 中断端点、提交周期性中

断传输、解析 HID 报告描述符或使用 Boot Protocol、解码按键码并上报事件。

- **传输类型**：采用 **Interrupt IN**（周期性）从键盘读取输入报告；也支持 control transfer（获取 descriptor、切换协议等）。
- **事件分发**：将解析出的按键事件（make/break，修饰键状态）通过回调、消息队列或 RT-Thread 设备/传感器框架上报给上层应用。

2. 支持能力

- USB 标准枚举（Get Descriptor / Set Configuration / Set Interface）
- 识别 HID 报告描述符并按 Report Descriptor 解析复杂报告
- 支持 **HID Boot Protocol（键盘）** 的快速解析（如果设备支持）
- 管理中断轮询周期（poll interval）与传输超时、重试策略
- 支持多设备（通过 Hub 支持）与动态插拔检测（hot-plug）
- 错误检测：CRC/NAK/STALL/timeout 处理与恢复
- 支持与 DMA 协同（若 USB Host 控制器/驱动支持）

3. 工作流程

1. **Host 初始化**：host controller 驱动初始化，VBUS 打开，USB interrupt/sof 启动。
2. **设备连接检测**：检测到设备插入（GPIO / ID / VBUS sense），触发枚举流程。
3. **枚举与类判断**：读取设备描述符、配置描述符、接口描述符，判断类为 HID（bDeviceClass/bInterfaceClass=0x03）。
4. **找端点并获取报告描述符**：获取 HID Descriptor 与 Report Descriptor，或判断是否支持 Boot Protocol。
5. **配置端点**：打开 Interrupt IN 端点，根据端点描述（最大包长、poll interval）提交中断接收请求（循环）。
6. **接收并解析报告**：每次中断到达后，解析 report（按 report descriptor 或 boot protocol），生成按键按下/释放事件。
7. **事件上报**：通过回调/消息队列/RT-Thread 设备上报给上层应用。
8. **拔出处理**：设备拔出时，停止轮询、释放资源、上报拔出事件。

硬件说明

Titan Board 提供一个 USB-Device 外设，位于开发板的位置如下所示：

FSP 配置

- 新建 `r_usb_hhid` stack。

- 配置 `r_usb_basic` stack。

- 配置 USB FS 引脚。

RT-Thread Settings 配置

- 使能 USB HHID。

- USB-HHID 示例目前使用的是 freertos 接口驱动，因此我们还需要使能 Freertos 兼容层软件包。

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与PC 机连接，然后将固件下载至开发板。

运行效果

将键盘通过转接线连接到 Titan Board 的 USB-DEV 接口上，按下键盘上的按键后，终端输出按下的是哪个按键。

3.13. USB-PCDC 驱动例程

中文 | **English**

简介

本示例演示如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 USB PCDC 模块（r_usb_pcdc）** 将设备枚举为 **USB 虚拟串口（CDC）**，从而在 PC 上以 “COM 口/ttyACM” 的形式与板卡进行串行数据通信。

USB PCDC 简介

1. 概述

USB PCDC（USB Peripheral Communications Device Class） 属于 USB 协议中定义的 **CDC（Communications Device Class，通信设备类）** 的一种实现。

它主要用于 **将嵌入式设备通过 USB 接口虚拟成串口（Virtual COM Port, VCOM）**，使主机（PC、嵌入式 Host）能够像访问传统 UART 串口一样访问 USB 设备。

常见应用场景：

- MCU 作为 USB 设备，通过 PCDC 连接到 PC，PC 识别为虚拟串口（COMx）。
- 嵌入式设备与上位机进行数据通信或调试。
- 代替物理 RS-232/RS-485 接口，降低硬件成本。

2. PCDC 的协议特性

1. **CDC 类协议**

- 属于 USB 定义的通信类设备，使用标准类驱动（无需额外安装驱动，在 Windows/Linux/MacOS 中大多数情况自动识别）。

2. **数据传输方式**

- **控制端点（Endpoint 0）**：用于设备枚举、描述符传输。
- **中断端点（Interrupt Endpoint）**：用于状态和控制信号（如 DTR、RTS）。
- **批量传输端点（Bulk Endpoint）**：用于大数据量的串口数据传输（高效可靠）。

3. **虚拟串口功能**

- 在主机端表现为标准串口（COM 口或 ttyUSB 设备）。
- 支持波特率、数据位、校验位、停止位等 UART 常见配置。

3. 典型应用架构

- **USB Device（MCU 端）**：实现 PCDC 协议栈，将 USB 数据与 MCU 内部 UART/应用层对接。
- **USB Host（PC/嵌入式系统）**：通过操作系统自带 CDC 驱动，映射出虚拟串口接口。
- **应用层**：上位机可直接使用串口工具（如 PuTTY、Tera Term、串口助手）进行交互。

4. 优点

- **无需额外驱动**：大多数 OS 自带 CDC 驱动。
- **替代传统串口**：即使 MCU 无物理串口，也能通过 USB 实现调试/通信。
- **速率更高**：USB 全速（12 Mbps）、高速（480 Mbps）带宽远高于传统 UART。
- **通用性强**：支持各种上位机应用（LabVIEW、Python pySerial、C# 串口库等）。

RA8 系列 USB PCDC 模块（r_usb_pcdc）简介

RA8 系列 MCU 内置 **USB PCDC（Peripheral CDC）模块**，专用于将 MCU 枚举为 **USB 虚拟串口（CDC 设备）**，从而实现与 PC 或其他 USB 主机的数据双向通信。该模块与 RA8 的基础 USB 模块（r_usb_basic）结合使用，通过类驱动 r_usb_pcdc 提供完整的 CDC 功能。

1. 模块定位与功能

- **设备模式支持**：实现 MCU 作为 USB 设备连接到主机（Full-Speed / High-Speed，根据具体 MCU 规格）
- **虚拟串口（CDC）功能**：
- 处理类请求：`SET_LINE_CODING`、`GET_LINE_CODING`、`SET_CONTROL_LINE_STATE`、`SEND_BREAK`
- 支持标准 Bulk IN/OUT 端点和 Interrupt IN 端点
- 生成串口状态通知（如 DTR/RTS、线路状态）
- **数据收发管理**：可通过 DMA/DTC 或 FIFO 缓冲实现大数据量高效传输
- **事件回调机制**：连接、断开、配置完成、数据收发完成、类请求处理、错误事件等
- **热插拔与低功耗**：支持 VBUS 侦测、挂起/唤醒、远程唤醒功能

- **应用层接口**：通过类驱动 API 提供读写缓冲、发送完成通知、事件回调机制

2. 支持能力

- **标准 USB 枚举**：支持 Device Descriptor、Configuration Descriptor、Interface Descriptor 等
- **端点管理**：配置 Bulk IN/OUT 和 Interrupt IN 端点
- **类请求解析**：按 CDC 协议解析行编码、控制线状态
- **传输可靠性**：支持 CRC、NAK、STALL 及超时处理
- **多设备与复合设备支持**：可在 Hub 环境下动态识别和管理多个 CDC 设备
- **与 DMA 协同**：优化数据吞吐，减少 CPU 占用

3. 模块架构

```
[ 应用层任务/线程 ]
          ↑            回调/事件通知（连接、配置、收发完成）
[ r_usb_pcdc 类驱动 ]
          ↕            类请求解析、数据收发管理、端点配置
[ r_usb_basic 基础驱动 ]
          ↕            USB 设备枚举、端点控制、SOF 中断、DMA 支持
[ USB 设备控制器/PHY ]
          ↕            USB D+/D- 信号（FS）或 ULPI/HS PHY（HS）
[ USB 线缆 ]  <–> 主机（PC）
```

关键子模块说明：

1. **类请求处理器**：解析 CDC 类请求，回调上层应用
2. **端点控制与缓冲**：管理 Bulk/Interrupt 端点传输，支持 DMA/FIFO
3. **事件管理**：连接/断开、配置完成、挂起/恢复、传输完成、错误处理
4. **电源与时钟控制**：USB PHY、VBUS 侦测、挂起唤醒管理

4. 工作流程

1. **USB 初始化**：USB 控制器、PHY 及类驱动初始化
2. **设备连接检测**：检测 VBUS/ID 信号触发枚举
3. **枚举与配置**：上传描述符、设置接口和端点
4. **端点数据传输**：应用通过 Bulk IN/OUT 与主机进行数据收发
5. **类请求处理**：处理主机发来的 CDC 控制请求
6. **事件回调上报**：将连接、配置、收发完成等事件通知应用

7. **断开处理**：停止端点传输，释放缓冲区，处理拔出事件

硬件说明s

Titan Board 提供一个 USB-Device 外设，位于开发板的位置如下所示：

FSP 配置

- 新建 r_usb_pcdc stack：

- 配置 r_usb_basic：

Settings / API Info	Property	Value
	Dedicated Charging Port (DCP) Mode	Disabled
	Notifications for SET_INTERFACE/SET_FEATURE/CLEAR_FEATURE	Enabled
	Double Buffering	Enabled
	Continuous Transfer Mode	Disabled
	LDO Regulator	Not Supported
	DMA/DTC Support	DMA Disabled
	DMA/DTC Source Address	DMA Disabled
	DMA/DTC Destination Address	DMA Disabled
	USB Compliance Test mode	Disabled
	USB TPL table name	NULL
	Module g_basic0 USB (r_usb_basic)	
	Name	g_basic0
	USB Mode	Peri mode
	USB Speed	Hi Speed
	USB Module Number	USB_IP1 Port
	USB Device Class	Peripheral Communications Device Class
	USB Descriptor	g_usb_descriptor
	USB Compliance Callback	NULL
	USBFS Interrupt Priority	Priority 12

USB Speed选择Hi Speed

USB Module 选择USB_IP1_Port

- 配置 USB FS 引脚：

- 配置 USB HS 引脚：

PS：注意，切换HS模式需要拉高P413引脚，如下图所示：

RT-Thread Settings 配置

使能 USB PCDC。

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

使用 USB 线连接 Titan Board 的 USB-DEV 接口与电脑，使用串口助手打开串口终端，向串口终端中输入 1，能在开发板的调试终端看到输出。

向 USB-PCDC 的串口终端中输入 2，能在 USB-PCDC 的串口终端中看到如下输出。

```
 2. NEXT STEPS

Visit the following URLs to learn about the kit and the RA family of MCUs, download tools and documentation, and get support:

 a) Titan Board resources:      renesas.com/ra/ra8p1
 b) RA product information:     renesas.com/ra
 c) RA product support forum:   renesas.com/ra/forum
 d) Renesas support:            renesas.com/support

 Press 1 for Kit Information or 2 for Next Steps.
```

3.14. WDT 使用说明

中文 | English

简介

看门狗设备可以保证我们的代码在我们的预期中进行，可以有效防止我们的程序因为一些其它不可控因素导致代码”跑飞“。本示例展示了如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 WDT 模块**，结合 **RT-Thread 的 WDT 驱动框架** 实现看门狗功能，包括定时器配置、喂狗操作及超时响应。

WDT（Watchdog Timer）简介

1. 概述

WDT（看门狗定时器） 是嵌入式系统中用于 **监控系统运行状态、防止程序卡死或异常挂起的硬件定时器**。

当 MCU 或处理器在规定时间内未对 WDT 进行复位（Kick/Reload）时，WDT 会 **自动复位系统**，确保系统能够恢复正常运行。

WDT 常用于：

- 防止软件死循环或任务卡死
- 提高系统可靠性和稳定性
- 工业控制、汽车电子、无人机、通信设备等关键应用

2. 工作原理

1. **计数器模式**

- WDT 内部有一个递减计数器，从设定初值开始计数。
- MCU 必须在计数器到达零前 **清零计数器（Reload/Kick）**。
- 如果计数器归零，触发 **系统复位** 或 **中断**。

2. **喂狗机制**

- MCU 定期向 WDT 写入特定值，重置计数器
- 俗称 “喂狗”，保证系统正常运行时 WDT 不复位

3. **复位/中断模式**

- **复位模式**：WDT 到期直接复位 MCU

- **中断模式**：WDT 到期产生中断，可先尝试软件自我恢复

3. WDT 类型

类型	描述
独立看门狗（Independent WDT, IWDT）	独立时钟源，系统重启或挂起时仍能运行
窗口看门狗（Window WDT）	MCU 必须在规定窗口时间内喂狗，否则触发复位
超级看门狗（Super WDT）	提供更高精度或可配置更长时间的保护
软看门狗（Software WDT）	通过软件定时器实现，看门狗功能由操作系统或应用维护

4. 关键参数

- **计数器位数**：决定 WDT 最大时间间隔
- **超时时间（Timeout）**：从上次喂狗到 WDT 触发复位的时间
- **时钟源**：独立低速晶振或系统时钟
- **复位方式**：硬件复位或中断通知

示例：

- MCU WDT 超时时间 500 ms
- MCU 任务每 100 ms 喂狗一次
- 若任务卡死 500 ms 未喂狗 → WDT 自动复位系统

5. WDT 使用原则

1. **定期喂狗**：确保系统在正常运行时 WDT 不触发
2. **关键任务喂狗**：通常在主循环或关键任务中喂狗
3. **中断安全**：喂狗操作应避免被中断阻塞
4. **合理超时设置**：超时时间既要大于任务执行时间，又不能太长，防止死机长时间不被复位

RA8 系列 WDT 模块（r_wdt）简介

RA8 系列 MCU 内置高性能 **WDT 模块（r_wdt）**，提供可靠的系统监控能力，可通过硬件复位或中断模式实现故障保护。

1. 基本特性

- **定时器模式**：支持单次计时或周期计时
- **时钟源**：独立低功耗时钟（LPO）或系统时钟
- **复位与中断输出**：
- 超时可触发 MCU 系统复位
- 可选择中断模式，产生 WDT 中断供软件处理
- **计数范围**：支持 8~32 位计数器，可灵活配置超时时间
- **安全性特性**：
- 禁止误操作保护（有些寄存器需要解锁序列）
- 多级复位模式（WDT Reset, Non-maskable Reset）
- **独立操作**：在 MCU 低功耗或睡眠模式下依然可运行

2. 模块架构

RA8 WDT 模块主要包含以下子模块：

1. **计数器单元（Counter Unit）**

- 基于时钟源递减计数
- 支持周期性溢出检测

2. **控制寄存器（Control Register）**

- 启动/停止 WDT
- 配置超时时间和中断/复位模式
- 配置喂狗序列

3. **状态寄存器（Status Register）**

- 反映 WDT 当前状态
- 超时标志
- 中断/复位触发标志

4. **复位与中断控制单元**

- 超时触发 MCU 复位
- 可选择中断模式通知 CPU
- 支持 NMI 或系统复位输出

3. 工作原理

1. **启动 WDT**：配置计数器、时钟源、超时时间后启动 WDT。
2. **喂狗操作**：软件周期性写入特定寄存器序列，重置计数器防止超时。
3. **超时响应**：

- 若未喂狗，计数器溢出触发超时
- 根据配置选择：
  - 硬件复位 MCU
  - 触发 WDT 中断供程序处理

4. **安全保护**：WDT 控制寄存器通常需解锁后才能修改，防止误操作。

RT-Thread WDT 框架简介

RT-Thread WDT（Watchdog Timer）框架 是 RT-Thread 设备驱动层提供的统一接口，用于管理各类 MCU 的硬件看门狗模块。该框架将硬件 WDT 功能抽象为标准设备接口，使应用层能够跨平台使用统一的 API 来实现系统自恢复机制。

1. 设备模型

在 RT-Thread 中，WDT 被作为 **设备对象**（`struct rt_device` 的子类，类型为 `RT_Device_Class_WDT`）进行管理，开发者无需关心底层硬件寄存器操作，只需调用标准接口即可完成看门狗功能。

2. 操作接口

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问看门狗硬件，相关接口如下所示：

函数	描述
rt_device_find()	根据看门狗设备设备名称查找设备获取设备句柄
rt_device_init()	初始化看门狗设备

函数	描述
rt_device_control()	控制看门狗设备
rt_device_close()	关闭看门狗设备

3. 框架特点

- **设备抽象统一**：所有硬件 WDT 通过相同的设备接口暴露给上层应用。
- **跨平台支持**：应用程序可在不同 MCU 平台间移植而无需修改 WDT 相关代码。
- **超时保护**：支持配置超时时间，系统未及时喂狗则触发中断或复位。
- **运行模式可选**：支持 **中断模式** 和 **复位模式**，满足不同场景需求。
- **安全可靠**：结合硬件 WDT 提供强制复位机制，防止系统异常卡死。

4. 使用流程

- **查找 wdt 设备**

```
#define WDT_DEVICE_NAME    "wdt"    /* 看门狗设备名称 */

static rt_device_t wdg_dev;         /* 看门狗设备句柄 */
/* 根据设备名称查找看门狗设备，获取设备句柄 */
wdg_dev = rt_device_find(WDT_DEVICE_NAME);
```

- **初始化看门狗**

```
/* 初始化设备 */
rt_device_init(wdg_dev);
```

- **控制看门狗**

命令控制字可取如下宏定义值：

```
#define RT_DEVICE_CTRL_WDT_GET_TIMEOUT    (1) /* 获取溢出时间 */
#define RT_DEVICE_CTRL_WDT_SET_TIMEOUT    (2) /* 设置溢出时间 */
#define RT_DEVICE_CTRL_WDT_GET_TIMELEFT   (3) /* 获取剩余时间 */
#define RT_DEVICE_CTRL_WDT_KEEPALIVE      (4) /* 喂狗 */
#define RT_DEVICE_CTRL_WDT_START          (5) /* 启动看门狗 */
#define RT_DEVICE_CTRL_WDT_STOP           (6) /* 停止看门狗 */
```

设置看门狗溢出时间使用示例如下所示：

```
#define WDT_DEVICE_NAME    "wdt"    /* 看门狗设备名称 */
```

```
rt_uint32_t timeout = 1;        /* 溢出时间，单位：秒*/
static rt_device_t wdg_dev;     /* 看门狗设备句柄 */
/* 根据设备名称查找看门狗设备，获取设备句柄 */
wdg_dev = rt_device_find(WDT_DEVICE_NAME);
/* 初始化设备 */
rt_device_init(wdg_dev);

/* 设置看门狗溢出时间 */
rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_SET_TIMEOUT, &timeout
/* 设置空闲线程回调函数 */
rt_thread_idle_sethook(idle_hook);
```

在空闲线程钩子函数里喂狗使用示例如下所示：

```
static void idle_hook(void)
{
    /* 在空闲线程的回调函数里喂狗 */
    rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_KEEPALIVE, NULL);
}
```

- **关闭看门狗**

```
rt_device_close(wdg_dev);
```

硬件说明

无

FSP 配置

- 打开 FSP 工具 新建 Stacks 选择 r_wdt：

- **注意**：使用 WDT 需要使能 OFS0 寄存器配置。

RT-Thread Settings 配置

- 使能 WDT。

示例工程说明

示例程序位于 `project/Titan_driver_wdt/src/hal_entry.c` 。

```
/*
 * Copyright (c) 2006-2024, RT-Thread Development Team
 *
 * SPDX-License-Identifier: Apache-2.0
 *
 * Change Logs:
 * Date           Author        Notes
 * 2024-03-11     kurisaw       first version
 */

#include <rtthread.h>
#include "hal_data.h"
#include <rtdevice.h>
#include <board.h>

#define DBG_TAG "wdt"
#define DBG_LVL DBG_LOG
#include <rtdbg.h>

#define WDT_DEVICE_NAME "wdt"     // 默认看门狗设备名，视 BSP 而定
#define WDT_FEED_INTERVAL 1000    // 喂狗间隔（单位 ms）
#define WDT_TIMEOUT 3             // 看门狗超时时间（单位 s）

static rt_device_t wdt_dev = RT_NULL;
static rt_thread_t feed_thread = RT_NULL;

#define LED_PIN_0    BSP_IO_PORT_06_PIN_00 /* Onboard LED pins */

void hal_entry(void)
{
```

```
    rt_kprintf("\nHello RT-Thread!\n");
    rt_kprintf("==================================================\
    rt_kprintf("This example project is an driver wdt routine!\n");
    rt_kprintf("==================================================\

    LOG_I("Tips:");
    LOG_I("You can run wdt testcase by executing the instruction: \

    while (1)
    {
        rt_pin_write(LED_PIN_0, PIN_HIGH);
        rt_thread_mdelay(1000);
        rt_pin_write(LED_PIN_0, PIN_LOW);
        rt_thread_mdelay(1000);
    }
}

static void feed_dog_entry(void *parameter)
{
    int count = 0;

    while (1)
    {
        if (count < 10)
        {
            rt_device_control(wdt_dev, RT_DEVICE_CTRL_WDT_KEEPALIVE
            LOG_I("[FeedDog] Feeding watchdog... %d", count);
        }
        else
        {
            LOG_E("[FeedDog] Simulate exception! Stop feeding.");
        }

        count++;
        rt_thread_mdelay(WDT_FEED_INTERVAL);
    }
}

static int wdt_sample(void)
{
    rt_err_t ret;

    wdt_dev = rt_device_find(WDT_DEVICE_NAME);
    if (wdt_dev == RT_NULL)
    {
        LOG_E("Cannot find %s device!", WDT_DEVICE_NAME);
        return -1;
    }

    ret = rt_device_control(wdt_dev, RT_DEVICE_CTRL_WDT_START, RT_N
    if (ret != RT_EOK)
    {
```

```
        LOG_E("Start watchdog failed!");
        return -1;
    }

    LOG_I("Watchdog started...", WDT_TIMEOUT);

    feed_thread = rt_thread_create("feed_dog", feed_dog_entry, RT_N
    if (feed_thread != RT_NULL)
        rt_thread_startup(feed_thread);

    return 0;
}
MSH_CMD_EXPORT(wdt_sample, wdt_sample);
```

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

在终端输入 wdt_sample 指令运行 WDT 测试程序，在喂狗 10 次后停止喂狗，模拟了程序异常情况。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 28 2025 10:09:53
 2006 - 2024 Copyright by RT-Thread team

Hello RT-Thread!
===================================================
This example project is an driver wdt routine!
===================================================
[I/wdt] Tips:
[I/wdt] You can run wdt testcase by executing the instruction: 'wdt_sample'
msh >wdt_sample
[I/wdt] Watchdog started...
[I/wdt] [FeedDog] Feeding watchdog... 0
msh >[I/wdt] [FeedDog] Feeding watchdog... 1
[I/wdt] [FeedDog] Feeding watchdog... 2
[I/wdt] [FeedDog] Feeding watchdog... 3
[I/wdt] [FeedDog] Feeding watchdog... 4
[I/wdt] [FeedDog] Feeding watchdog... 5
[I/wdt] [FeedDog] Feeding watchdog... 6
[I/wdt] [FeedDog] Feeding watchdog... 7
[I/wdt] [FeedDog] Feeding watchdog... 8
[I/wdt] [FeedDog] Feeding watchdog... 9
[E/wdt] [FeedDog] Simulate exception! Stop feeding.
[E/wdt] [FeedDog] Simulate exception! Stop feeding.
```

引用参考

设备与驱动：WDT 设备

3.15. Wi-Fi使用说明

中文|English

简介

本示例演示了如何在 **Titan Board** 上使用 **RA8 系列 MCU 的 SDHI 模块（r_sdhi）** 驱动 **CYWL6208-GS WiFi 模块**，从而实现基于 SDIO 接口的无线网络连接。
在该示例中，MCU 通过 **SDIO 总线** 与 CYWL6208-GS 模块（芯片 CYW43438）通信，结合 RT-Thread 的网络协议栈（lwIP）实现 TCP/IP 网络功能。

WiFi 简介

1. 概述

WiFi（Wireless Fidelity，无线保真） 是一种基于 **IEEE 802.11 标准** 的无线局域网（WLAN）通信技术。它通过无线电波在终端设备（如 MCU、手机、电脑、IoT 设备）与接入点（AP）之间传输数据，实现 **短距离高速无线通信**。

WiFi 目前是最常用的无线连接技术之一，广泛应用于家庭网络、办公网络、工业控制、智能家居和物联网设备。

2. 工作原理

1. **通信方式**

- WiFi 使用 **ISM 频段**（2.4 GHz、5 GHz、6 GHz），通过 **OFDM、DSSS、QAM 等调制方式**实现数据传输。
- WiFi 网络由 **接入点（AP）** 和 **终端设备（Station, STA）** 构成。

2. **数据传输流程**

- 终端 STA 通过 **扫描（Scan）** 找到可用 WiFi 网络
- STA 发送 **认证与关联请求**
- 接入点 AP 完成 **握手与密钥协商**
- STA 与 AP 建立安全通信通道后进行数据传输

3. WiFi 协议标准

WiFi 的不同版本对应 IEEE 802.11 标准：

标准	频段	最大速率	特点
802.11b	2.4 GHz	11 Mbps	早期版本，抗干扰差
802.11g	2.4 GHz	54 Mbps	向下兼容 b
802.11n (WiFi 4)	2.4/5 GHz	600 Mbps	MIMO 技术
802.11ac (WiFi 5)	5 GHz	3.5 Gbps	MU-MIMO、80/160 MHz 信道
802.11ax (WiFi 6)	2.4/5/6 GHz	9.6 Gbps	OFDMA，更低延迟，更高并发
802.11be (WiFi 7)	2.4/5/6 GHz	>30 Gbps	多链路操作（MLO），超高吞吐量

4. WiFi 模式

- **STA 模式（Station）**：设备作为终端接入 AP
- **AP 模式（Access Point）**：设备作为热点提供网络
- **P2P 模式（Peer-to-Peer/Ad-Hoc）**：设备间直接通信
- **混合模式**：STA 与 AP 同时存在

5. WiFi 安全机制

- **WEP**（已废弃，不安全）
- **WPA/WPA2**：基于 TKIP、AES 加密，主流安全方式
- **WPA3**：增强安全性，支持 SAE 握手，防止暴力破解

6. WiFi 特点

优点

- 高速传输（Gbps 级别）
- 广泛支持（手机、PC、MCU 等）
- 成本低，生态成熟
- 可同时接入多设备

缺点

- 功耗较高，不适合超低功耗应用
- 传输距离有限（一般 10~100 米）

- 在 2.4 GHz 下易受干扰（蓝牙、微波炉等）

RA8 系列 SDHI 模块概述

RA8 系列 MCU 内置的 **SDHI（SD Host Interface）** 模块是一个符合 **SD/SDIO/eMMC 标准** 的主机控制器，能够通过 **1-bit 或 4-bit 数据线**与存储设备（SD 卡、eMMC）或支持 SDIO 协议的外设（如 WiFi、蓝牙模块）进行通信。

在**存储应用**中，它可作为 SD/eMMC 控制器实现高速读写；在 **外设扩展应用**中，它可作为 SDIO 主机与 WiFi、蓝牙、GNSS 等模块建立数据通道。

在本示例中，SDHI 主要用于驱动 **CYWL6208-GS WiFi 模块（基于 SDIO 接口）**，从而实现 MCU 的无线联网功能。

1. 总体特性

- **支持 SD 标准**
- SD v1.x / v2.x / SDHC / SDXC
- 支持 SPI 模式和 SD/MMC 模式
- **高速数据传输**
- 最高可达 50 MHz SDCLK（具体取决于 MCU 时钟配置）
- 支持 1-bit/4-bit 数据总线
- **自动命令和数据传输**
- 支持 DMA 传输模式，减少 CPU 占用
- 自动命令序列生成（CMD0~CMD63）
- **错误检测**
- CRC7 校验命令，CRC16 校验数据
- 超时检测，响应错误识别
- **中断支持**
- 卡插拔检测中断
- 命令完成中断
- 数据传输完成中断
- 错误中断

2. SDHI 模块架构

RA8 SDHI 模块主要包含以下子模块：

1. **命令控制单元（Command Control Unit）**

- 负责发送命令（CMD0~CMD63）
- 处理命令响应（R1、R2、R3、R7 等）
- 支持命令超时检测和 CRC 校验

2. **数据传输单元（Data Transfer Unit）**

- 通过内部 FIFO 或 DMA 实现数据收发
- 支持块读/写，最大 512 字节块
- 支持单块/多块传输模式

3. **时钟与总线控制**

- SDCLK 生成和分频
- 1-bit 或 4-bit 总线切换
- 可配置高/低电平保持时间

4. **卡检测与电源控制**

- 检测 SD 卡插入/拔出状态
- 可控制卡片电源开关（如支持）

5. **中断与事件控制单元**

- 命令完成中断
- 数据传输完成中断
- 错误中断
- 卡插拔中断

3. 工作流程（以 SDIO WiFi 为例）

1. **硬件初始化**

- 配置 SDHI 引脚：CMD、CLK、DAT0~3
- 使能时钟，复位 SDHI 控制器

2. **总线初始化**

- 设置时钟频率（初始低速 → 高速模式）
- 发送初始化命令（CMD0、CMD5、CMD3 等）
- 枚举设备，分配 RCA

3. **功能配置**

- 识别为 SDIO WiFi 设备
- 配置块大小、总线宽度、传输速率
- 打开中断机制

4. **正常运行**

- WiFi 驱动通过 SDIO 寄存器访问与数据传输接口控制模块
- MCU 通过 **DMA + 中断** 与 WiFi 芯片交换大数据包
- 上层协议栈（lwIP）完成 TCP/IP 通信

硬件说明

FSP 配置

- 首先需要配置 Flash，Flash 配置参考 `Titan_component_flash_fs` 工程下的 README.md。
- 接下来配置 SDHI1，新建 r_sdhi stack：

- 配置 SDHI1 stack：

- 配置 SDHI1 引脚：

RT-Thread Settings 配置

- 使能 ospi Flash。

- SDHI1 的 Bus Width 设置为 4。

- 配置 WHD 软件包。

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

按下复位按键重启开发板，如果终端输出固件读取错误的信息，那么需要使用 **ymodem** 向 Flash 中下载 WiFi 固件。使用 `whd_res_download whd_firmware` 命令下载 43438A1.bin 到 Flash 的 `whd_firmware` 分区中；使用 `whd_res_download whd_clm` 命令下载 43438A1.clm_blob 到 Flash 的 `whd_clm` 分区中。

WiFi 固件可以在工程根目录的 `/firmware` 文件夹中找到。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Aug 14 2025 18:08:15
 2006 - 2024 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.

Hello RT-Thread core0!
[I/drv.ospi_flash] Get flash id: 0x021a5bef
[I/drv.ospi_flash] OSPI flash initialized successfully
[D/FAL] Flash device |              ospi_flash | addr: 0x90000000 | len: 0x04000000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name         | flash_dev  |   offset   |    length  |
[I/FAL] ------------------------------------------------------------
[I/FAL] | whd_firmware | ospi_flash | 0x00000000 | 0x00080000 |
[I/FAL] | whd_clm      | ospi_flash | 0x00080000 | 0x00080000 |
[I/FAL] | download     | ospi_flash | 0x00100000 | 0x00200000 |
[I/FAL] | easyflash    | ospi_flash | 0x00300000 | 0x00100000 |
[I/FAL] | filesystem   | ospi_flash | 0x00400000 | 0x00c00000 |
[I/FAL] ============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL MTD NOR device (filesystem) created successfully
[I/wifi] Filesystem initialized!
msh />[E/whd.resources] Read resource head error for partition[whd_firmware]
Fatal error: download_resource doesn't exist, whd_bus_write_wifi_firmware_image failed at line 253
Failed to load wifi firmware
[E/whd.resources] Read resource head error for partition[whd_firmware]
Function whd_bus_sdio_init failed at line 542 checkres = 4002
[E/whd.wlan] Unable to start the WiFi module!
whd_res_download whd_firmware
Please select the whd_firmware file and use Ymodem to send.CCCCC
Download whd_firmware to flash success. file size: 419799
msh />whd_res_download whd_clm
Please select the whd_clm file and use Ymodem to send.CC
Download whd_clm to flash success. file size: 7222
msh />
```

向 whd_firmware 下载 43438A1.bin

向 whd_clm 下载 43438A1.clm_blob

下载 WiFi 固件后复位开发板，能看到 WiFi 固件正常加载，WiFi 模块初始化成功。接着输入 `wifi join TitanBoard 12345678` 连接 WiFi，然后输入 `ping baidu.com` 进行 ping 测试。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Aug 14 2025 18:08:15
 2006 - 2024 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.

Hello RT-Thread core0!
[I/drv.ospi_flash] Get flash id: 0x021a5bef
[I/drv.ospi_flash] OSPI flash initialized successfully
[D/FAL] Flash device |              ospi_flash | addr: 0x90000000 | len: 0x04000000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name         | flash_dev  |   offset   |    length  |
[I/FAL] ------------------------------------------------------------
[I/FAL] | whd_firmware | ospi_flash | 0x00000000 | 0x00080000 |
[I/FAL] | whd_clm      | ospi_flash | 0x00080000 | 0x00080000 |
[I/FAL] | download     | ospi_flash | 0x00100000 | 0x00200000 |
[I/FAL] | easyflash    | ospi_flash | 0x00300000 | 0x00100000 |
[I/FAL] | filesystem   | ospi_flash | 0x00400000 | 0x00c00000 |
[I/FAL] ============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL MTD NOR device (filesystem) created successfully
[I/wifi] Filesystem initialized!
msh />WLAN MAC Address : 2C:B0:FD:A2:2D:60
WLAN Firmware    : wl0: Mar 28 2021 22:55:55 version 7.45.98.117 (dc5d9c4 CY) FWID 01-d36e8386
WLAN CLM         : API: 12.2 Data: 9.10.39 Compiler: 1.29.4 ClmImport: 1.36.3 Creation: 2021-03-28 22:47:33
WHD VERSION      : 3.1.0.23284 : v3.1.0 : GCC 13.2 : 2024-03-21 22:57:11 +0800
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w1

msh />wifi join TitanBoard 12345678
[I/WLAN.mgnt] wifi connect success ssid:TitanBoard
msh />[I/WLAN.lwip] Got IP address : 192.168.173.8

msh />ping baidu.com
ping: not found specified netif, using default netdev w0.
60 bytes from 182.61.201.211 icmp_seq=1 ttl=50 time=60 ms
60 bytes from 182.61.201.211 icmp_seq=2 ttl=50 time=55 ms
60 bytes from 182.61.201.211 icmp_seq=3 ttl=50 time=55 ms
60 bytes from 182.61.201.211 icmp_seq=4 ttl=50 time=245 ms

--- 182.61.201.211 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss
minimum = 55ms, maximum = 245ms, average = 103ms
msh />
```

4. 组件篇

4.1. Flash 文件系统使用说明

中文 | **English**

简介

本例程展示了如何在 **Titan Board** 上使用 **OSPI Flash** 挂载 **LittleFS 文件系统**。通过此示例，你可以在外部闪存中读写文件，实现数据持久化存储。程序基于 **RT-Thread 文件系统组件** 和 **OSPI 驱动**。

RA8 系列 OSPI 特性

RA8 系列 MCU 提供 **Octal SPI / Quad SPI 接口（OSPI）**，专为高速外部闪存访问设计，主要特性如下：

1. 多线高速访问

- 支持 **1-8 数据线模式**：
- 1线 SPI（标准 SPI）
- 4线 QSPI（Quad SPI）
- 8线 OSPI（Octal SPI）
- 最高时钟可达 **100 MHz**（取决于 MCU 和 Flash 器件），适合大容量高速读写。
- 可配置 **连续读取模式**，减少每次读取命令开销。

2. 灵活的 Flash 操作模式

- 支持 **页写入（Page Program）**，单页通常为 256 字节，可分块管理跨页写入。
- 支持 **块擦除（Block Erase）**，典型块大小 4KB、64KB，可按需擦除提高写入效率。
- 支持 **随机读取** 与 **连续读取**。

- 可配置 **读写指令、Dummy Cycle、Data Line Width**，满足不同 Flash 型号要求。
- 支持 **高速模式（DDR / Quad / Octal I/O）**，提高传输效率。

3. 中断与 DMA 支持

- OSPI 外设可通过 **中断** 通知传输完成。
- 支持 **DMA 模式**，减少 CPU 占用。
- 与 RT-Thread 结合，可封装为 **块设备（blkdev）**，方便挂载文件系统或缓存数据。

4. RT-Thread 驱动集成

- RA8 OSPI 驱动可注册为 RT-Thread **块设备**。
- LittleFS 或 FatFS 可挂载在该块设备上，实现文件系统操作。
- 支持 **线程安全 API**，方便多线程访问 Flash。
- 可配置 **页缓存大小** 和 **块擦写策略**，优化 LittleFS 性能。

5. 高可靠性特性

- 支持 **写保护（Write Protect）** 和 **软件锁定（Software Lock）**。
- 支持 **错误检测**，如命令错误或数据 CRC 校验。
- 可在断电或系统重启后安全挂载文件系统。

FAL 抽象层

FAL (Flash Abstraction Layer) Flash 抽象层，是对 Flash 及基于 Flash 的分区进行管理、操作的抽象层，对上层统一了 Flash 及 分区操作的 API (框架图如下所示)，并具有以下特性：

- 支持静态可配置的分区表，并可关联多个 Flash 设备；
- 分区表支持 **自动装载** 。避免在多固件项目，分区表被多次定义的问题；
- 代码精简，对操作系统 **无依赖** ，可运行于裸机平台，比如对资源有一定要求的 Bootloader；
- 统一的操作接口。保证了文件系统、OTA、NVM（例如：EasyFlash） 等对 Flash 有一定依赖的组件，底层 Flash 驱动的可重用性；

- 自带基于 Finsh/MSH 的测试命令，可以通过 Shell 按字节寻址的方式操作（读写擦） Flash 或分区，方便开发者进行调试、测试；

硬件说明

FSP 配置

- 新建 r_ospi_b stack：

- 配置 r_ospi_b stack：

*[Titan_component_flash_fs] FSP Configuration

Stacks Configuration

Generate Project Content

Summary | BSP | Clocks | Pins | Interrupts | Event Links | Stacks | Components

Properties | Problems | Console

g_ospi_b OSPI (r_ospi_b)

Settings | API Info

Property	Value
Initial Mode	
Read	
Command Code	0x0B
Dummy Cycles	8
Program	
Command Code	0x02
Dummy Cycles	0
Row Load	
Row Store	
Write Enable	
Command Code	0x06
Status Read	
Command Code	0x05
Dummy Cycles	0
Sector Erase	
Command Code	0x20
Block Erase	
Command Code	0xD8
Chip Erase	
Command Code	0x60
Protocol Mode	SPI (1S-1S-1S)
Frame Format	Standard
Latency Mode	Fixed
Address Length	3 bytes
Address MSB Mask	0xF0
Command Code Length	1 byte
Status Register Address Length	No address
Status Register Address	0x00
High-speed Mode	
Timing Settings	
XiP Mode	
DOTF	
Row Addressing	
Pins	
OM_0_CS0	None
OM_0_CS1	P104
OM_0_DQS	P801
OM_0_ECSINT1	P105

初始化模式参数

*[Titan_component_flash_fs] FSP Configuration

Stacks Configuration

Generate Project Content

Summary | BSP | Clocks | Pins | Interrupts | Event Links | Stacks | Components

Properties | Problems | Console

g_ospi_b OSPI (r_ospi_b)

Settings | API Info

Property	Value
High-speed Mode	
Read	
Command Code	0xFDFD
Dummy Cycles	16
Program	
Command Code	0x8484
Dummy Cycles	0
Row Load	
Row Store	
Write Enable	
Command Code	0x0606
Status Read	
Command Code	0x0505
Dummy Cycles	8
Sector Erase	
Command Code	0x2121
Block Erase	
Command Code	0xDCDC
Chip Erase	
Command Code	0x6060
Protocol Mode	Dual data rate OPI (8D-8D-8D)
Frame Format	xSPI Profile 1.0
Latency Mode	Fixed
Address Length	4 bytes
Address MSB Mask	0xF0
Command Code Length	2 bytes
Status Register Address Length	4 bytes
Status Register Address	0x00
Timing Settings	
Chip Select	
Command Interval	2 cycles
Pull-up Timing	No Extension
Pull-down Timing	No Extension
SDR	
Drive Timing	Drive 1/2 cycle before CK
CK Sampling Edge	Falling edge
Sampling Delay	None
DDR	

高速模式参数

- Flash 引脚配置：

RT-Thread Settings 配置

- 在 RT-Thread Settings 中使能 Flash。

软件说明

本例程的文件系统初始化源码位于 `./board/ports/drv_filesystem.c` 中：

```
#include <rtthread.h>

#if defined(BSP_USING_FILESYSTEM)
#include <dfs_romfs.h>
#include <dfs_fs.h>
#include <dfs_file.h>

#if DFS_FILESYSTEMS_MAX < 4
#error "Please define DFS_FILESYSTEMS_MAX more than 4"
#endif
#if DFS_FILESYSTEM_TYPES_MAX < 4
#error "Please define DFS_FILESYSTEM_TYPES_MAX more than 4"
#endif

#define DBG_TAG "app.filesystem"
#define DBG_LVL DBG_INFO
#include <rtdbg.h>

#ifdef BSP_USING_FS_AUTO_MOUNT
#ifdef BSP_USING_SDCARD_FATFS
static int onboard_sdcard_mount(void)
{
    if (dfs_mount("sd", "/sdcard", "elm", 0, 0) == RT_EOK)
    {
        LOG_I("SD card mount to '/sdcard'");
    }
    else
    {
        LOG_E("SD card mount to '/sdcard' failed!");
        rt_pin_write(0x000D, PIN_LOW);
    }
```

```
    return RT_EOK;
}
#endif /* BSP_USING_SDCARD_FATFS */
#endif /* BSP_USING_FS_AUTO_MOUNT */

#ifdef BSP_USING_FLASH_FS_AUTO_MOUNT
#ifdef BSP_USING_FLASH_FATFS
#define FS_PARTITION_NAME "filesystem"

static int onboard_fal_mount(void)
{
    /* 初始化 fal 功能 */
    extern int fal_init(void);
    extern struct rt_device* fal_mtd_nor_device_create(const char *
    fal_init ();
    /* 在 ospi flash 中名为 "filesystem" 的分区上创建一个块设备 */
    struct rt_device *mtd_dev = fal_mtd_nor_device_create (FS_PARTI
    if (mtd_dev == NULL)
    {
        LOG_E("Can't create a mtd device on '%s' partition.", FS_PA
        return -RT_ERROR;
    }
    else
    {
        LOG_D("Create a mtd device on the %s partition of flash suc
    }

    /* 挂载 ospi flash 中名为 "filesystem" 的分区上的文件系统 */
    if (dfs_mount (FS_PARTITION_NAME, "/fal", "lfs", 0, 0) == 0)
    {
        LOG_I("Filesystem initialized!");
    }
    else
    {
        dfs_mkfs ("lfs", FS_PARTITION_NAME);
        if (dfs_mount ("filesystem", "/fal", "lfs", 0, 0) == 0)
        {
            LOG_I("Filesystem initialized!");
        }
        else
        {
            LOG_E("Failed to initialize filesystem!");
            rt_pin_write(0x000D, PIN_LOW);
        }
    }

    return RT_EOK;
}
#endif /*BSP_USING_FLASH_FATFS*/
#endif /*BSP_USING_FLASH_FS_AUTO_MOUNT*/
```

```
const struct romfs_dirent _romfs_root[] =
{
#ifdef BSP_USING_SDCARD_FATFS
    {ROMFS_DIRENT_DIR, "sdcard", RT_NULL, 0},
#endif

#ifdef BSP_USING_FLASH_FATFS
  { ROMFS_DIRENT_DIR, "fal", RT_NULL, 0 },
#endif
        };

const struct romfs_dirent romfs_root =
{
ROMFS_DIRENT_DIR, "/", (rt_uint8_t*) _romfs_root, sizeof(_romfs_roo
};

static int filesystem_mount(void)
{

#ifdef RT_USING_DFS_ROMFS
    if (dfs_mount(RT_NULL, "/", "rom", 0, &(romfs_root)) != 0)
    {
        LOG_E("rom mount to '/' failed!");
    }

    /* 确保块设备注册成功之后再挂载文件系统 */
    rt_thread_delay(500);
#endif
#ifdef BSP_USING_FS_AUTO_MOUNT
    onboard_sdcard_mount();
#endif /* BSP_USING_FS_AUTO_MOUNT */

#ifdef BSP_USING_FLASH_FS_AUTO_MOUNT
    onboard_fal_mount ();
#endif

    return RT_EOK;
}
INIT_COMPONENT_EXPORT(filesystem_mount);
#endif /* defined(BSP_USING_FILESYSTEM)*/
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

按下复位按键重启开发板，观察开发板终端日志。

```
initialize rti_board_start
:0 done

 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Jul 25 2025 15:14:51
 2006 - 2024 Copyright by RT-Thread team
do components initialization.
initialize rti_board_end
:0 done
initialize rt_work_sys_workqueue_init
:0 done
initialize dfs_init
:0 done
initialize dfs_lfs_init
:0 done
initialize finsh_system_init
:0 done

Hello RT-Thread!
================================================
This example project is an ospi flash routine!
================================================
[I/drv.ospi_flash] Get flash id: 0x021a5bef
[I/drv.ospi_flash] OSPI flash initialized successfully
[D/FAL] Flash device |               ospi_flash | addr: 0x90000000 | len: 0x04000000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev  |   offset   |    length  |
[I/FAL] ------------------------------------------------------------
[I/FAL] | wifi_image | ospi_flash | 0x00000000 | 0x00080000 |
[I/FAL] | bt_image   | ospi_flash | 0x00080000 | 0x00080000 |
[I/FAL] | download   | ospi_flash | 0x00100000 | 0x00200000 |
[I/FAL] | easyflash  | ospi_flash | 0x00300000 | 0x00100000 |
[I/FAL] | filesystem | ospi_flash | 0x00400000 | 0x00c00000 |
[I/FAL] ============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL MTD NOR device (filesystem) created successfully
[I/main] Filesystem initialized!
msh />
```

4.2. netutils 示例说明

中文 | English

简介

netutils 是一个包含众多简洁好用网络工具的软件包，利用该软件包，可以给开发者在调试网络功能时带来很多便利。当需要使用一些调试网络的小工具时，只需要拥有 netutils 软件包就够了，堪称网络功能调试界的瑞士军刀。

本例程展示如何在 Titan Board 开发板上使用 netutils 软件包的各种功能。

硬件说明

本例程需要依赖 Titan Board 上的 WiFi 模块完成网络通信，因此请确保硬件平台上的 WiFi 模组可以正常工作。

FSP配置

- 首先需要配置 Flash，Flash 配置参考 `Titan_component_flash_fs` 工程下的 README_zh.md。
- 接下来配置 SDHI1，新建 r_sdhi stack：

- 配置 SDHI1 stack：

- 配置 SDHI1 引脚：

RT-Thread Settings 配置

- 添加 netutils 软件包，使能要使用的网络工具。

软件说明

本例程的源码位于 `/project/Titan_component_netutils`。

netutils 软件包文件结构说明

下面是 RT-Thread netutils 软件包功能的分类和简介：

名称	分类	功能简介
Ping	调试测试	利用 “ping” 命令可以检查网络是否连通，可以很好地帮助我们分析和判定网络故障
NTP	时间同步	网络时间协议
TFTP	文件传输	TFTP 是一个传输文件的简单协议，比 FTP 还要轻量级
Iperf	性能测试	测试最大 TCP 和 UDP 带宽性能，可以报告带宽、延迟抖动和数据包丢失
NetIO	性能测试	测试网络的吞吐量的工具
Telnet	远程访问	可以远程登录到 RT-Thread 的 Finsh/MSH Shell
tcpdump	网络调试	tcpdump 是 RT-Thread 基于 lwIP 的网络抓包工具

netutils 软件包文件结构如下所示：

```
netutils        // netutils 文 件 夹
├── iperf       // iperf 网 络 性 能 测 试
├── netio       // netio 网 络 吞 吐 量 测 试
├── ntp         // ntp 时 间 同 步 功 能
├── ping        // ping 功 能
├── tcpdump     // 网 络 抓 包 工 具
├── telnet      // telnet 服 务 器
├── tftp        // TFTP 功 能
├── tools       // 网 络 测 试 工 具
```

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

挂载文件系统

由于在使用 TFTP 功能向系统内传输文件时需要文件系统的支持，所以系统在初始化时会进行文件系统相关功能的初始化。文件系统正常初始化提示信息如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Sep  1 2025 14:02:13
 2006 - 2024 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
Hello RT-Thread!
============================================================
This example project is an driver wifi routine(hyperflash)!
============================================================
[I/drv.ospi_flash] Get flash id: 0x021a5bef
[I/drv.ospi_flash] OSPI flash initialized successfully
[D/FAL] Flash device |               ospi_flash | addr: 0x90000000
[I/FAL] ==================== FAL partition table ==================
[I/FAL] | name         | flash_dev  |   offset   |    length  |
[I/FAL] ------------------------------------------------------------
[I/FAL] | whd_firmware | ospi_flash | 0x00000000 | 0x00080000 |
[I/FAL] | whd_clm      | ospi_flash | 0x00080000 | 0x00080000 |
[I/FAL] | download     | ospi_flash | 0x00100000 | 0x00200000 |
[I/FAL] | easyflash    | ospi_flash | 0x00300000 | 0x00100000 |
[I/FAL] | filesystem   | ospi_flash | 0x00400000 | 0x00c00000 |
[I/FAL] ============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL MTD NOR device (filesystem) created successfully
[I/wifi] Filesystem initialized!
msh />WLAN MAC Address : 2C:B0:FD:A2:2D:60
WLAN Firmware    : wl0: Mar 28 2021 22:55:55 version 7.45.98.117 (d
WLAN CLM         : API: 12.2 Data: 9.10.39 Compiler: 1.29.4 ClmImpo
WHD VERSION      : 3.1.0.23284 : v3.1.0 : GCC 13.2 : 2024-03-21 22:
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w1
```

连接 Wi-Fi

程序运行后会进入 MSH 命令行，等待用户配置设备接入网络。使用 MSH 命令 wifi join ssid password 配置网络，如下所示：

```
msh />wifi join ssid_test password
join ssid:ssid_test
```

```
[I/WLAN.mgnt] wifi connect success ssid:ssid_test
msh />[I/WLAN.lwip] Got IP address : 192.168.173.8
```

Ping 工具

Ping 是一种网络诊断工具，用来测试数据包能否通过 IP 协议到达特定主机。估算与主机间的丢失数据包率（丢包率）和数据包往返时间。

- Ping 域名

```
msh />ping www.rt-thread.org
ping: not found specified netif, using default netdev w0.
60 bytes from 120.222.223.251 icmp_seq=1 ttl=48 time=76 ms
60 bytes from 120.222.223.251 icmp_seq=2 ttl=48 time=54 ms
60 bytes from 120.222.223.251 icmp_seq=3 ttl=48 time=74 ms
60 bytes from 120.222.223.251 icmp_seq=4 ttl=48 time=74 ms

--- 120.222.223.251 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss
minimum = 54ms, maximum = 76ms, average = 69ms
msh />
```

- Ping IP

```
msh />ping 192.168.173.247
ping: not found specified netif, using default netdev w0.
60 bytes from 192.168.173.247 icmp_seq=1 ttl=64 time=42 ms
60 bytes from 192.168.173.247 icmp_seq=2 ttl=64 time=44 ms
60 bytes from 192.168.173.247 icmp_seq=3 ttl=64 time=24 ms
60 bytes from 192.168.173.247 icmp_seq=4 ttl=64 time=24 ms

--- 192.168.173.247 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss
minimum = 24ms, maximum = 44ms, average = 33ms
msh />
```

NTP 工具

NTP 是网络时间协议 (Network Time Protocol)，它是用来同步网络中各个计算机时间的协议。在 netutils 软件包实现了 NTP 客户端，连接上网络后，可以获取当前 UTC 时间，并更新至 RTC 中。

MSH 命令效果如下：

```
msh />ntp_sync
[I/ntp] Get local time from NTP server: Mon Sep  1 15:15:35 2025
msh />
```

TFTP 工具

TFTP （**Trivial File Transfer Protocol**，简单文件传输协议）是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务，端口号为 69 ，比传统的FTP 协议要轻量级很多，适用于小型的嵌入式产品上。

TFTP 工具的准备工作需要下面两个步骤：

- 安装 TFTP 客户端安装文件位于 **packages/netutils/tools/Tftpd64-4.60-setup.exe** ，使用 TFTP 前，请先安装该软件。
- 启动 TFTP 服务器在传输文件前，需要在 RT-Thread 上使用 Finsh/MSH 命令来启动 TFTP 服务器，大致效果如下：

```
msh />tftp -s
tftp server start!
msh />
```

- 连接 RT-Thread 操作系统并发送文件

打开刚安装的 Tftpd64 软件，按如下操作进行配置：

1. 选择 Tftp Client ；
2. 在 Server interfaces 下拉框中，务必选择与 RT-Thread 处于同一网段的网卡；
3. 填写 TFTP 服务器的 IP 地址。可以在 RT-Thread 的 MSH 下使用 ifconfig 命令查看；
4. 填写 TFTP 服务器端口号，默认：69；
5. 选择好要发送文件，Remote File 是服务器端保存文件的路径（包括文件名），选项支持相对路径和绝对路径。由于 RT-Thread 默认开启 DFS_USING_WORKDIR 选项，此时相对路径是基于 MSH 当前进入的目录。所以，使用相对路径时，务必提前切换好目录；
6. 点击 Put 按钮即可。

注意：如果 DFS_USING_WORKDIR 未开启，同时 Remote File 为空，文件会将保存至根路径下。

在 MSH 中查看 PC 发送的文件：

```
msh />ls
Directory /:
test.txt             16
msh />cat test.txt
Hello RT-Thread!
msh />
```

- 从 RT-Thread 获取文件

首先在根目录创建一个文本文件，输入内容 “Hello!”。

```
msh />echo "Hello!" test1.txt
msh />cat test1.txt
Hello!
msh />
```

1. 在 Tftpd64 软件中，填写好要接收保存的文件路径（包含文件名）；
2. Remote File 是服务器端待接收回来的文件路径（包括文件名），选项支持相对路径和绝对路径。由于 RT-Thread 默认开启 DFS_USING_WORKDIR 选项，此时相对路径是基于 Finsh/MSH 当前进入的目录。所以，使用相对路径时，务必提前切换好目录；
3. 点击 Get 按钮即可。如下所示，将 /test1.txt 保存到本地，这里使用的是绝对路径：

下图是从 RT-Thread 获取到的文件：

Telnet 工具

Telnet 是一种基于 TCP 的远程终端协议，用于在网络中远程登录设备的命令行界面。通过 Telnet，可以在 PC 上远程访问开发板的命令行，实现调试、监控和控制。要在 RT-Thread 上使用 Finsh/MSH 命令来启动 telnet 服务，大致效果如下：

```
msh />telnet_server
Telnet server start successfully
msh />telnet: waiting for connection
```

接下来使用远程连接工具连接开发板。

连接成功后就可以远程控制开发板了。

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Sep  1 2025 14:02:13
 2006 - 2024 Copyright by RT-Thread team
msh />
msh />
msh />list device
device               type         ref count
------------ -------------------- ----------
telnet       Character Device     2
w1           Network Interface    1
w0           Network Interface    1
wlan0        Network Interface    1
wlan1        Network Interface    1
filesystem   MTD Device           1
rtc          RTC                  0
uart8        Character Device     0
pin          Pin Device           0
msh />
```

更多网络工具

除了上述常用的网络工具，netutils 软件包也提供一些开发调试中比较实用的网络工具，如 NetIO 工具、Iperf 工具和 tcpdump 工具。这些工具的使用方法可以参考软件包功能目录下的说明文件。

5. 多媒体显示篇

5.1. CEU 摄像头使用说明

中文|English

简介

本示例展示了如何在 **Titan Board** 上使用 **CEU（Camera Engine Unit）** 接口连接 **OV5640 摄像头**，并通过 **RT-Thread LCD 框架** 将捕获的图像显示到 **RGB565 LCD 屏幕**。

主要功能包括：

- 初始化 CEU 摄像头接口，捕获实时视频流
- 配置 OV5640 摄像头参数（分辨率、帧率、输出格式）
- 使用 RT-Thread LCD 驱动显示摄像头捕获的图像
- 支持图像格式转换（YUV422 → RGB565）

RA8 系列 CEU（Camera Engine Unit）特性

RA8 系列 MCU 内置 **CEU 硬件模块**，用于高效捕获摄像头图像数据，并支持多种图像格式和分辨率，直接将数据传输到内存或显示接口。

1. CEU 硬件接口特性

1. **接口类型**

- **DVP（Digital Video Port）** 接口，用于连接 CMOS 摄像头
- 支持 8/10/12 位数据总线
- 同步信号：
  - **PCLK**：像素时钟
  - **HSYNC**：行同步信号
  - **VSYNC**：帧同步信号

2. **输入分辨率与帧率**

- 支持 VGA、QVGA、SXGA、UXGA 等常用分辨率
- 帧率支持 1–60 fps，可配置以适应不同应用

3. **摄像头兼容性**

- 与 OV5640、OV7670 等常用 CMOS 摄像头兼容
- 支持自动初始化和寄存器配置

2. 图像格式与处理能力

1. **图像格式支持**

- **YUV422**（常用用于视频传输）
- **RGB565**（适合 LCD 显示）
- **RAW10/RAW12**（用于图像处理和算法开发）

2. **图像处理功能**

- **颜色空间转换**：YUV ↔ RGB
- **图像裁剪**：可只采集部分 ROI（Region of Interest）
- **图像缩放**：支持按比例缩小或放大
- **镜像与翻转**：可水平或垂直镜像

3. **硬件加速**

- CEU 内置硬件处理单元，可减轻 CPU 负担
- 提供快速图像格式转换和缩放功能

3. DMA 支持与缓冲机制

1. **高速 DMA 传输**

- 与 MCU 的 DMAC 配合，实现内存高速写入
- 支持直接将图像写入 Frame Buffer 或 LCD 缓冲区

2. **多缓冲区机制**

- 支持双缓冲或环形缓冲，保证连续视频采集
- 减少帧丢失和显示延迟

3. **DMA 配置灵活**

- 可配置缓冲区起始地址、大小
- 支持中断触发与回调

4. 中断机制

1. **中断类型**

- **帧完成中断（Frame End）**：每帧采集完成时触发
- **行中断（Line End，可选）**：每行采集完成时触发
- **错误中断**：包括缓冲区溢出、同步信号异常

2. **中断特点**

- 支持 RT-Thread ISR 回调注册
- 可与 DMA 联动，实现实时处理和显示

5. 时序与同步特性

1. **行/帧同步**

- HSYNC 对齐每行数据
- VSYNC 对齐每帧图像

2. **像素时钟**

- CEU 支持外部 PCLK 或内部分频
- 确保与摄像头输出同步，避免采样错误

3. **数据对齐**

- 支持字节对齐或像素对齐
- 可根据图像格式自动调整

6. 性能与优化

1. **高吞吐量**

- DMA + 双缓冲机制实现连续视频采集
- CPU 占用低，适合实时应用

2. **可靠性**

- 同步信号异常可触发中断
- 缓冲区溢出检测
- 支持帧丢失自动恢复

3. **灵活性**

- 支持多分辨率和多格式切换

- 可配置裁剪和缩放区域，提高显示效率

7. 应用场景

- 实时视频显示到 LCD
- 视频采集与处理算法测试
- 嵌入式视觉应用，如监控、手势识别、机器人视觉

RA8 系列 MCU GLCDC（Graphics LCD Controller）特性

RA8 系列 MCU（如 RA8P1）集成 **GLCDC 硬件模块**，用于驱动 TFT/LCD 显示屏，实现高速图形渲染和视频显示，支持多种分辨率、色彩格式和显示模式。

1. 硬件特性

1. **分辨率支持**

- 可驱动从 QVGA (320×240) 到 WQVGA/XGA 等常用分辨率
- 受片上 RAM 和显示接口带宽限制

2. **色彩支持**

- 支持 1/4/8/16/24/32 位色深
- 常用 RGB565、RGB888 格式
- 支持调色板模式（Palette mode）
- 可硬件进行色彩转换

3. **接口类型**

- 并行 RGB（TFT LCD 接口）
- 支持 16/18/24 位数据总线
- 可与外部 LCD 面板直接连接
- 时序可编程：HSYNC、VSYNC、DE、PCLK、RGB 输出

2. 图层与显示模式

1. **图层支持**

- 单图层模式（单画面显示）
- 多图层模式（通过调色板或硬件 alpha 混合叠加）

- 支持透明/半透明叠加

2. **显示模式**

- RGB 模式（直显颜色输出）
- CLUT/Palette 模式（索引颜色，通过查表转换）
- 可配置扫描方向（水平/垂直扫描）

3. DMA 与帧缓冲

1. **帧缓冲访问**

- GLCDC 可直接访问片上/外部 SRAM 中的帧缓冲
- 支持单缓冲、双缓冲模式
- 支持环形缓冲，用于连续刷新

2. **DMA 支持**

- 与 MCU DMAC 协同工作，减少 CPU 占用
- 可将图像直接从内存传输到 LCD
- 支持行、块或全帧传输

4. 硬件图形功能

1. **窗口裁剪与缩放**

- 可指定显示窗口区域
- 可支持简单缩放（水平/垂直）

2. **硬件图形加速**

- 支持矩形填充、颜色替换
- 支持图像透明处理
- 可结合 CEU 或 DMA 进行视频显示

3. **颜色格式转换**

- YCbCr → RGB
- RGB888 → RGB565
- 硬件加速，减少 CPU 负担

5. 中断机制

1. **中断类型**

- 帧完成中断（Frame End）
- 行中断（Line End，可选）
- 访问错误/溢出中断

2. **中断应用**

- 与 RT-Thread ISR 集成
- 可在帧完成时触发更新缓冲区或切换双缓冲
- 方便实现动画和视频显示

6. 性能优化

1. **双缓冲机制**

- 支持双缓冲减少闪烁
- CPU 可在后台绘制下一帧
- GLCDC 硬件自动切换显示缓冲

2. **帧率控制**

- 可编程时钟和行/帧同步
- 支持 30fps、60fps 等常见刷新率

3. **CPU 卸载**

- 大量图形操作由硬件完成
- DMA + GLCDC 结合，实现高效图像显示

硬件说明

CEU 摄像头接口和 RGB LCD 接口如下图所示：

FSP 配置

HyperRAM 配置

- 新建 r_ospi_b stack：

- 配置 r_ospi_b stack：

- HyperRAM 引脚配置：

- HyperRAM 所有相关引脚的驱动能力配置为 H，OM_1_SIO0~OM_1_SIO7 需要配置为输入上拉。

CEU 配置

- 新建 `r_ceu` stack：

- 配置 CEU：

- 配置 CEU 引脚：

CEU 时钟配置

- 新建 `r_gpt` stack：

- 配置摄像头时钟 PWM 输出：

D/AVE 2D 配置

- 新建 `r_drw` stack：

RGB LCD 配置

- 新建 `r_glcdc` stack：

- 配置中断回调和图形层1：

- 配置输出参数、CLUT、TCON和抖动。

[Titan_driver_rgb_lcd] FSP Configuration
Stacks Configuration
Generate Project Content
Summary BSP Clocks Pins Interrupts Event Links Stacks Components
Properties Problems Console
g_display0 Graphics LCD (r_glcdc)
Settings
API Info
Property
Value
Timing
Horizontal total cycles 1024 在水平线中指定总循环数，设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal active video cycles 800 在水平线中指定活动视频周期数。设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal back porch cycles 46
Horizontal sync signal cycles 1 指定 Hsync 信号断言周期数。设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal sync signal polarity Low active 水平同步信号线极性为 Low active
Vertical total lines 525 指定一个帧的总行数
Vertical active video lines 480 指定一帧中活动视频线的数量
Vertical back porch lines 23
Vertical sync signal lines 1 垂直同步信号线为1
Vertical sync signal polarity Low active 垂直同步信号的极性为 Low active
Data Enable Signal Polarity High active 数据使能信号的极性为 High active
Sync edge Rising edge 设置同步信号的边沿为下降沿
Format
Color format 16bits RGB565 颜色格式
Color order BGR 按照 BGR 的颜色顺序将信号输出到 LCD 面板
Endian Little endian 将信号输出到 LCD 面板的数据字节顺序设置为小端
Background
Alpha 255
Red 0
Green 0
Blue 0
CLUT
Enabled No
Size 256
TCON
Hsync pin select LCD_TCON0 Hsync 信号引脚
Vsync pin select LCD_TCON1 Vsync 信号引脚
Data enable (DE) pin select LCD_TCON3 数据使能引脚
Panel clock source Internal clock (GLCDCLK)
Panel clock division ratio 1/8
Dithering
Enabled No
Mode Truncate
Pattern A Pattern 11
Pattern B Pattern 11
Pattern C Pattern 11
Pattern D Pattern 11

- 配置 GLCDC 的引脚：

[Titan_driver_rgb_lcd] FSP Configuration
Pin Configuration
Generate Project Content
Select Pin Configuration
Export to CSV file
Configure Pin Driver Warnings
Titan_Board
Manage configurations...
Generate data: g_bsp_pin_cfg
Pin Selection
Type filter text
Connectivity:ESWM_RMII
Connectivity:I3C
Connectivity:IIC
Connectivity:OSPI
Connectivity:PDM
Connectivity:SCI
Connectivity:SDHI
Connectivity:SPI
Connectivity:SSIE
Connectivity:USB FS
Connectivity:USB HS
Debug:JTAG/SWD
Debug:TRACE
ExBus:BUS
ExBus:SDRAM
HMI:CEU
HMI:GLCDC
GLCDC
HMI:MIPI
Interrupt:IRQ
System:CGC
TRG:ADC(Digital)
Pin Configuration
Cycle Pin Group
Name Value Lock Link
Pin Group Selection Mixed
Operation Mode Custom
Input/Output
LCD_CLK P515
LCD_DATA0 P914
LCD_DATA1 P915
LCD_DATA2 P903
LCD_DATA3 P902
LCD_DATA4 P910
LCD_DATA5 P911
LCD_DATA6 P912
LCD_DATA7 P913
LCD_DATA8 P904
LCD_DATA9 P207
LCD_DATA10 P807
LCD_DATA11 P806
LCD_DATA12 P805
LCD_DATA13 P801
LCD_DATA14 P804
LCD_DATA15 P803
LCD_DATA16 None
Module name: GLCDC
Pin Function
Pin Number
Summary BSP Clocks Pins Interrupts Event Links Stacks Components
Properties Problems Console

LCD 背光配置

- 新建 `r_gpt` stack：

- 配置背光 PWM 输出：

RT-Thread Settings 配置

- 使能 CUE 摄像头，使用 i2c1 和 ov5640 摄像头；使能 RGB565 LCD，使用 pwm7 输出背光。

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与PC 机连接，然后将固件下载至开发板。

运行效果

复位 Titan Board 后终端会输出如下信息：

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Aug 15 2025 14:55:33
 2006 - 2024 Copyright by RT-Thread team
[D/drv_lcd]
lcd framebuffer address:0x22097200

Hello RT-Thread!
============================================================
This example project is an display with ceu camera routine!
============================================================

msh >i2c probe camera id:0x78
[I/sensor] set framesize:640X480
[I/sensor] Open CEU device
```

下面是 LCD 屏幕上显示的图像：

5.2. MIPI CSI 摄像头使用说明

中文|English

简介

本示例展示了如何在 **Titan Board** 上使用 **MIPI CSI（Camera Serial Interface）** 接口连接 **OV5640 摄像头**，并通过 **RT-Thread LCD 框架**将捕获的图像显示到 **RGB565 LCD 屏幕**。

主要功能包括：

- 初始化 MIPI CSI 摄像头接口，捕获实时视频流
- 配置 OV5640 摄像头参数（分辨率、帧率、输出格式）
- 使用 RT-Thread LCD 驱动显示摄像头捕获的图像
- 支持图像格式转换（YUV422 → RGB565）

RA8 系列 MIPI CSI 特性

RA8 系列 MCU 集成 **MIPI CSI 硬件模块**，用于高速、低功耗摄像头数据采集，适合高清视频和实时图像处理。

1. 硬件接口特性

1. **接口类型**

- **MIPI CSI-2** D-PHY 接口，支持高带宽串行传输
- 支持 1–4 数据通道 (Lanes)
- 同步信号由 MIPI D-PHY 控制，无需额外 HSYNC/VSYNC

2. **数据速率与分辨率**

- 支持每通道最高 1.5–2.5 Gbps（取决 MCU 系列和 PHY 配置）
- 可驱动常见摄像头分辨率：VGA、QVGA、SXGA、UXGA、1080p 等

3. **摄像头兼容性**

- 兼容 OV5640、IMX219 等常见 CMOS 摄像头
- 支持摄像头寄存器配置和自动初始化

2. 图像格式与处理能力

1. **图像格式支持**

- **RAW10 / RAW12**（用于算法开发和图像处理）
- **YUV422**（用于视频显示）
- **RGB565**（适合 LCD 显示）

2. **图像处理功能**

- **颜色空间转换**（YUV → RGB）
- **图像裁剪**（ROI capture）
- **镜像与翻转**
- **硬件缩放加速**

3. **硬件加速**

- MIPI CSI 内置 DMA，可减轻 CPU 负担
- 高速格式转换与缩放

3. DMA 支持与缓冲机制

1. **高速 DMA 传输**

- 与 MCU DMAC 协作，将图像直接写入内存或 LCD 缓冲区
- 减少 CPU 干预，提高帧率

2. **多缓冲区机制**

- 支持双缓冲或环形缓冲，实现连续视频采集
- 避免帧丢失和显示延迟

3. **DMA 配置灵活**

- 可配置缓冲区起始地址和大小
- 支持中断回调处理

4. 中断机制

1. **中断类型**

- **帧完成中断**：每帧采集完成时触发
- **行完成中断（可选）**
- **错误中断**：缓冲区溢出、PHY 错误等

2. **中断特点**

- 支持 RT-Thread ISR 回调
- 可与 DMA 联动，实现实时显示

5. 时序与同步特性

1. **同步方式**

- CSI 使用 MIPI D-PHY 提供时钟和数据同步
- 无需额外 HSYNC/VSYNC

2. **像素时钟与数据对齐**

- 支持按像素对齐或按字节对齐
- 自动处理 RAW/YUV/RGB 数据对齐

6. 性能优化

1. **高吞吐量**

- DMA + 双缓冲实现无间断视频采集
- CPU 占用低，适合实时应用

2. **可靠性**

- PHY 错误、数据丢失中断可触发异常处理
- 支持缓冲区溢出检测

3. **灵活性**

- 多分辨率、多格式支持
- 支持 ROI 捕获和硬件缩放，提高显示效率

7. 应用场景

- 实时视频显示到 LCD
- 视频采集与处理算法验证
- 嵌入式视觉应用，如监控、手势识别、机器人视觉

RA8 系列 MCU GLCDC（Graphics LCD Controller）特性

RA8 系列 MCU（如 RA8P1）集成 **GLCDC 硬件模块**，用于驱动 TFT/LCD 显示屏，实现高速图形渲染和视频显示，支持多种分辨率、色彩格式和显示模式。

1. 硬件特性

1. **分辨率支持**

- 可驱动从 QVGA (320×240) 到 WQVGA/XGA 等常用分辨率
- 受片上 RAM 和显示接口带宽限制

2. **色彩支持**

- 支持 1/4/8/16/24/32 位色深
- 常用 RGB565、RGB888 格式
- 支持调色板模式（Palette mode）
- 可硬件进行色彩转换

3. **接口类型**

- 并行 RGB（TFT LCD 接口）
- 支持 16/18/24 位数据总线
- 可与外部 LCD 面板直接连接
- 时序可编程：HSYNC、VSYNC、DE、PCLK、RGB 输出

2. 图层与显示模式

1. **图层支持**

- 单图层模式（单画面显示）
- 多图层模式（通过调色板或硬件 alpha 混合叠加）
- 支持透明/半透明叠加

2. **显示模式**

- RGB 模式（直显颜色输出）
- CLUT/Palette 模式（索引颜色，通过查表转换）
- 可配置扫描方向（水平/垂直扫描）

3. DMA 与帧缓冲

1. **帧缓冲访问**

- GLCDC 可直接访问片上/外部 SRAM 中的帧缓冲
- 支持单缓冲、双缓冲模式
- 支持环形缓冲，用于连续刷新

2. **DMA 支持**

- 与 MCU DMAC 协同工作，减少 CPU 占用
- 可将图像直接从内存传输到 LCD
- 支持行、块或全帧传输

4. 硬件图形功能

1. **窗口裁剪与缩放**

- 可指定显示窗口区域
- 可支持简单缩放（水平/垂直）

2. **硬件图形加速**

- 支持矩形填充、颜色替换
- 支持图像透明处理
- 可结合 CEU 或 DMA 进行视频显示

3. **颜色格式转换**

- YCbCr → RGB
- RGB888 → RGB565
- 硬件加速，减少 CPU 负担

5. 中断机制

1. **中断类型**

- 帧完成中断（Frame End）
- 行中断（Line End，可选）
- 访问错误/溢出中断

2. **中断应用**

- 与 RT-Thread ISR 集成

- 可在帧完成时触发更新缓冲区或切换双缓冲
- 方便实现动画和视频显示

6. 性能优化

1. **双缓冲机制**

- 支持双缓冲减少闪烁
- CPU 可在后台绘制下一帧
- GLCDC 硬件自动切换显示缓冲

2. **帧率控制**

- 可编程时钟和行/帧同步
- 支持 30fps、60fps 等常见刷新率

3. **CPU 卸载**

- 大量图形操作由硬件完成
- DMA + GLCDC 结合，实现高效图像显示

硬件说明

MIPI DSI/CSI接口和 RGB LCD 接口如下图所示：

FSP 配置

HyperRAM 配置

- 新建 r_ospi_b stack：

- 配置 r_ospi_b stack：

- HyperRAM 引脚配置：

- HyperRAM 所有相关引脚的驱动能力配置为 H，OM_1_SIO0~OM_1_SIO7 需要配置为输入上拉。

I2C0 配置

- 新建 `r_iic_master` stack：

- 配置 I2C0：

- 配置 I2C0 引脚：

VIN 配置

- 新建 `r_vin` stack：

- 配置 VIN：

MIPI CSI 配置

- 配置 MIPI CSI：

[Titan_display_camera_csi] FSP Configuration
Stacks Configuration
Generate Project Content
Threads
New Thread
Remove
HAL/Common
g_ioport I/O Port (r_ioport)
g_uart8 UART (r_sci_b_uart)
Objects
New Object >
Remove
g_cam_vin Video Input (r_vin) Stacks
New Stack >
Extend Stack >
Remove
g_cam_mipi_csi MIPI Camera Interface (r_mipi_csi)
Summary
BSP
Clocks
Pins
Interrupts
Event Links
Stacks
Components
Properties
Problems
Console
g_cam_mipi_csi MIPI Camera Interface (r_mipi_csi)
Settings
API Info
Property
Value
Module g_cam_mipi_csi MIPI Camera Interface (r_mipi_csi)
General
Name
g_cam_mipi_csi
模块名称设置为 g_cam_mipi_csi
Receive
Data Enable (User Options - Configured VIN input data format is always enabled)
End of Transmission Code
Generic Short Packet Code 1
Generic Short Packet Code 2
Generic Short Packet Code 3
Generic Short Packet Code 4
Generic Short Packet Code 5
Generic Short Packet Code 6
Generic Short Packet Code 7
Generic Short Packet Code 8
YUV422 8-bit
YUV422 10-bit
RAW8
RGB888
Interrupts
Interrupt Priorities
Receive Interrupt
Priority 12
Data Lane Interrupt
Priority 12
Virtual Channel 0..15 Interrupt
Priority 12
Power Management
Priority 12
Short Packet FIFO
Priority 12
设置中断优先级
Receive Events
RX (Reception) Active Detect Interrupt Enable
Data Lane 0 Events
Start of Transmission Error Detect
Start of Transmission Synchronization Error Detect

[Titan_display_camera_csi] FSP Configuration
Stacks Configuration
Generate Project Content
Summary
BSP
Clocks
Pins
Interrupts
Event Links
Stacks
Components
Properties
Problems
Console
g_cam_mipi_csi MIPI Camera Interface (r_mipi_csi)
Settings
API Info
Property
Value
RAW8
RGB888
Interrupts
Interrupt Priorities
Receive Interrupt
Priority 12
Data Lane Interrupt
Priority 12
Virtual Channel 0..15 Interrupt
Priority 12
Power Management
Priority 12
Short Packet FIFO
Priority 12
设置中断优先级
Receive Events
RX (Reception) Active Detect Interrupt Enable
Data Lane 0 Events
Start of Transmission Error Detect
Start of Transmission Synchronization Error Detect
Control Error Detect
Escape Sequence Error Detect
ULPS Exit Detect
ULPS Enter Detect
Data Lane 1 Events
Start of Transmission Error Detect
Start of Transmission Synchronization Error Detect
Control Error Detect
Escape Sequence Error Detect
ULPS Exit Detect
ULPS Enter Detect
设置 Data Lane 事件
Virtual Channels
Power Management
Short Packet FIFO
Callback
cam_mipi_csi_callback
设置中断回调函数
Callback Context
NULL
Number of Data Lanes
2
Zero Length Long Packet Output Mode
Disabled
Error Frame Notification Mode
Enabled
Reserved Packet Reception
Enabled
ECC Check 24-bit mode (CSI-2 Ver 1.3 Mode)
Disabled
LFSR (Descramble) Enable Mode
Disabled
Frequency Clock Rate Adjust
0
Frequency Clock Rate Adjust
0

- 配置 MIPI 引脚：

MIPI PHY 配置

- 设置 MIPI PHY 模块名称：

D/AVE 2D 配置

- 新建 `r_drw` stack：

RGB LCD 配置

- 新建 `r_glcdc` stack：

- 配置中断回调和图形层1：

- 配置输出参数、CLUT、TCON和抖动。

[Titan_driver_rgb_lcd] FSP Configuration
Stacks Configuration
Generate Project Content
Summary BSP Clocks Pins Interrupts Event Links Stacks Components
Properties Problems Console
g_display0 Graphics LCD (r_glcdc)
Settings
API Info
Property
Value
Timing
Horizontal total cycles 1024 在水平线中指定总循环数，设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal active video cycles 800 在水平线中指定活动视频周期数。设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal back porch cycles 46
Horizontal sync signal cycles 1 指定 Hsync 信号断言周期数。设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal sync signal polarity Low active 水平同步信号线极性为 Low active
Vertical total lines 525 指定一个帧的总行数
Vertical active video lines 480 指定一帧中活动视频线的数量
Vertical back porch lines 23
Vertical sync signal lines 1 垂直同步信号线为1
Vertical sync signal polarity Low active 垂直同步信号的极性为 Low active
Data Enable Signal Polarity High active 数据使能信号的极性为 High active
Sync edge Rising edge 设置同步信号的边沿为下降沿
Format
Color format 16bits RGB565 颜色格式
Color order BGR 按照 BGR 的颜色顺序将信号输出到 LCD 面板
Endian Little endian 将信号输出到 LCD 面板的数据字节顺序设置为小端
Background
Alpha 255
Red 0
Green 0
Blue 0
CLUT
Enabled No
Size 256
TCON
Hsync pin select LCD_TCON0 Hsync 信号引脚
Vsync pin select LCD_TCON1 Vsync 信号引脚
Data enable (DE) pin select LCD_TCON3 数据使能引脚
Panel clock source Internal clock (GLCDCLK)
Panel clock division ratio 1/8
Dithering
Enabled No
Mode Truncate
Pattern A Pattern 11
Pattern B Pattern 11
Pattern C Pattern 11
Pattern D Pattern 11

- 配置 GLCDC 的引脚：

[Titan_driver_rgb_lcd] FSP Configuration
Pin Configuration
Generate Project Content
Select Pin Configuration
Export to CSV file
Configure Pin Driver Warnings
Titan_Board
Manage configurations...
Generate data: g_bsp_pin_cfg
Pin Selection
Type filter text
Connectivity:ESWM_RMII
Connectivity:I3C
Connectivity:IIC
Connectivity:OSPI
Connectivity:PDM
Connectivity:SCI
Connectivity:SDHI
Connectivity:SPI
Connectivity:SSIE
Connectivity:USB FS
Connectivity:USB HS
Debug:JTAG/SWD
Debug:TRACE
ExBus:BUS
ExBus:SDRAM
HMI:CEU
HMI:GLCDC
GLCDC
HMI:MIPI
Interrupt:IRQ
System:CGC
TRG:ADC(Digital)
Pin Function
Pin Number
Pin Configuration
Cycle Pin Group
Name
Value
Lock
Link
Pin Group Selection Mixed
Operation Mode Custom
Input/Output
LCD_CLK P515
LCD_DATA0 P914
LCD_DATA1 P915
LCD_DATA2 P903
LCD_DATA3 P902
LCD_DATA4 P910
LCD_DATA5 P911
LCD_DATA6 P912
LCD_DATA7 P913
LCD_DATA8 P904
LCD_DATA9 P207
LCD_DATA10 P807
LCD_DATA11 P806
LCD_DATA12 P805
LCD_DATA13 P801
LCD_DATA14 P804
LCD_DATA15 P803
LCD_DATA16 None
Module name: GLCDC
Summary BSP Clocks Pins Interrupts Event Links Stacks Components
Properties Problems Console

LCD 背光配置

- 新建 `r_gpt` stack：

- 配置背光 PWM 输出：

RT-Thread Settings 配置

- 使能 MIPI CSI 摄像头，使用 i2c0；使能 RGB565 LCD，使用 pwm7 输出背光。

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与PC 机连接，然后将固件下载至开发板。

运行效果

复位 Titan Board 后终端会输出如下信息：

```
 \ | /
- RT -     Thread Operating System
 / | \     5.2.0 build Aug 15 2025 15:02:50
 2006 - 2024 Copyright by RT-Thread team
[D/drv_lcd]
lcd framebuffer address:0x22020d40
msh >
Hello Titan Board!
==================================================
This example project is an mipi-csi camera display routine!
==================================================
[I/camera.i2c] I2C bus initialized: i2c0
The vin driver uses hardware mailboxes for the buffer.
```

下面是 LCD 屏幕上显示的图像：

5.3. LVGL 示例说明

中文 | English

简介

本示例基于 **LVGL** 图形库，构建了一个适用于嵌入式设备的图形用户界面（GUI）示例项目，结合实际的显示驱动和输入驱动，展示了从底层驱动适配到上层界面构建的完整流程。通过该示例，开发者不仅可以学习如何在嵌入式环境中移植和配置 LVGL，还能掌握界面布局、事件响应、主题定制等关键技术点，从而为后续构建实际产品界面打下基础。

LVGL 简介

LVGL（Light and Versatile Graphics Library，轻量且多功能的图形库）是一个开源的嵌入式 GUI（图形用户界面）开发框架。它的设计初衷是为资源受限的嵌入式设备提供流畅、现代化的图形界面，因此在运行效率、内存占用和可移植性方面都做了大量优化。无论是在简单的低端 MCU 上，还是在功能更强大的 MPU 平台上，LVGL 都能够高效运行，并提供丰富的图形控件和交互体验。

主要特点

1. **轻量级**
   LVGL 最大的优势在于轻量化，它对资源的需求非常低。在一个内存只有几十 KB 的微控制器上，LVGL 依然可以流畅运行。框架本身的内存开销较小，渲染算法经过精心优化，既可以保证较低的 CPU 使用率，又能在有限的硬件条件下呈现出较好的视觉效果。这使得 LVGL 特别适合那些对硬件资源敏感的应用场景，比如低功耗设备、可穿戴设备、家用电器控制面板等。

2. **跨平台**
   LVGL 的跨平台特性非常突出，它能够运行在多种操作系统之上，包括 FreeRTOS、RT-Thread、Zephyr、Linux 等，也可以直接运行在裸机环境中。开发者只需要为 LVGL 提供底层的显示驱动和输入驱动接口，便可以快速完成移植，从而在不同的硬件平台之间复用同一套 UI 代码。这种灵活性大大降低了开发成本，使得 LVGL 成为嵌入式 GUI 的通用解决方案。

3. **丰富的控件（Widgets）**
   LVGL 内置了种类丰富的 GUI 控件，例如按钮、标签、滑条、进度条、复选框、列表、表格、图表等。这些控件几乎涵盖了常见人机交互界面的需求，开发者无需从零开始设计和实现组件，大大缩短了开发周期。同时，LVGL 还允许用户基

于现有控件扩展新的组件，从而构建符合特定需求的界面。无论是简单的数字显示，还是复杂的图形控制界面，LVGL 都能胜任。

4. **多样的渲染能力**
   在视觉呈现方面，LVGL 提供了丰富的渲染功能。它支持抗锯齿、透明度、渐变、阴影、边框、圆角等效果，可以让界面看起来更加美观和现代。除此之外，LVGL 还内置动画系统，支持多种缓动函数，能够实现平滑的控件移动、渐变过渡和动态效果。这些特性使得界面不仅功能完善，还能带来良好的用户体验。

5. **输入设备支持**
   LVGL 支持多种输入设备类型，包括触摸屏、电容屏、鼠标、键盘和编码器，甚至可以实现多点触控交互。它提供了统一的输入接口层，开发者只需实现底层的驱动适配，就能轻松把输入事件传递到 LVGL 的事件系统中。这样一来，开发者可以更加专注于上层的界面逻辑，而不必花费过多时间在输入事件处理上。

6. **国际化与多语言**
   LVGL 在国际化方面也有很好的支持。它使用 UTF-8 编码，可以处理几乎所有语言的字符集。同时，它支持双向文本渲染，可以正确显示如阿拉伯语、希伯来语等从右到左书写的语言。这使得 LVGL 能够应用在面向全球用户的产品中，不同国家和地区的用户都能通过本地化界面获得良好的体验。

7. **可扩展性**
   LVGL 提供了灵活的主题和样式系统，开发者可以方便地定制控件的外观样式，以实现不同风格的 UI。通过更换主题，可以快速切换界面的整体视觉风格。此外，LVGL 还可以与第三方图形库、文件系统和图像解码器结合使用，从而扩展其功能。例如，可以使用文件系统来加载外部字体和图片，或者集成 JPEG/PNG 解码库来显示复杂图像。

应用场景

LVGL 在实际应用中有着非常广泛的覆盖范围。在消费电子领域，它常用于智能家居控制面板、家电显示屏、智能手表和健身设备等，这些产品往往需要在有限硬件资源下提供友好的用户交互界面。在工业控制领域，LVGL 被用于 HMI（人机交互界面）和各种仪器仪表，帮助用户直观地监控和操作设备。在汽车电子中，LVGL 可以驱动中控屏、副驾娱乐屏，甚至车载仪表盘。而在医疗设备方面，它则适合做小型显示界面，例如手持检测仪、便携式监护设备等。

生态与社区

LVGL 作为一个开源项目，采用 **MIT License** 开源协议，既适合个人学习，也可以在商业产品中免费使用。围绕 LVGL，已经形成了一个活跃的生态系统。官方提供了 **SquareLine Studio** 这类可视化设计工具，支持拖拽式设计界面并导出 LVGL 代码，极大地提升了开发效率。此外，**LVGL Simulator** 可以让开发者直接在 PC 上进行界

面调试，不需要每次都烧录到目标硬件上。全球开发者社区十分活跃，贡献了大量开源控件、主题和移植案例，为新手和企业开发者提供了丰富的资源和支持。

硬件说明

运行本示例需要外接一块 RGB565 LCD 屏幕，开发板上 40pin RGB565 LCD 接口如下：

FSP 配置

RGB LCD 配置

- 新建 `r_glcdc` stack：

- 配置中断回调和图形层1：

- 配置输出参数、CLUT、TCON和抖动。

[Titan_driver_rgb_lcd] FSP Configuration
Stacks Configuration
Generate Project Content
Summary BSP Clocks Pins Interrupts Event Links Stacks Components
Properties Problems Console
g_display0 Graphics LCD (r_glcdc)
Settings
API Info
Property
Value
Timing
Horizontal total cycles 1024 在水平线中指定总循环数，设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal active video cycles 800 在水平线中指定活动视频周期数。设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal back porch cycles 46
Horizontal sync signal cycles 1 指定 Hsync 信号断言周期数。设置为系统中 LCD 面板数据表中定义的循环次数。
Horizontal sync signal polarity Low active 水平同步信号线极性为 Low active
Vertical total lines 525 指定一个帧的总行数
Vertical active video lines 480 指定一帧中活动视频线的数量
Vertical back porch lines 23
Vertical sync signal lines 1 垂直同步信号线为1
Vertical sync signal polarity Low active 垂直同步信号的极性为 Low active
Data Enable Signal Polarity High active 数据使能信号的极性为 High active
Sync edge Rising edge 设置同步信号的边沿为下降沿
Format
Color format 16bits RGB565 颜色格式
Color order BGR 按照 BGR 的颜色顺序将信号输出到 LCD 面板
Endian Little endian 将信号输出到 LCD 面板的数据字节顺序设置为小端
Background
Alpha 255
Red 0
Green 0
Blue 0
CLUT
Enabled No
Size 256
TCON
Hsync pin select LCD_TCON0 Hsync 信号引脚
Vsync pin select LCD_TCON1 Vsync 信号引脚
Data enable (DE) pin select LCD_TCON3 数据使能引脚
Panel clock source Internal clock (GLCDCLK)
Panel clock division ratio 1/8
Dithering
Enabled No
Mode Truncate
Pattern A Pattern 11
Pattern B Pattern 11
Pattern C Pattern 11
Pattern D Pattern 11

- 配置 GLCDC 的引脚：

[Titan_driver_rgb_lcd] FSP Configuration
Pin Configuration
Generate Project Content
Select Pin Configuration
Export to CSV file
Configure Pin Driver Warnings
Titan_Board
Manage configurations...
Generate data: g_bsp_pin_cfg
Pin Selection
Type filter text
Connectivity:ESWM_RMII
Connectivity:I3C
Connectivity:IIC
Connectivity:OSPI
Connectivity:PDM
Connectivity:SCI
Connectivity:SDHI
Connectivity:SPI
Connectivity:SSIE
Connectivity:USB FS
Connectivity:USB HS
Debug:JTAG/SWD
Debug:TRACE
ExBus:BUS
ExBus:SDRAM
HMI:CEU
HMI:GLCDC
GLCDC
HMI:MIPI
Interrupt:IRQ
System:CGC
TRG:ADC(Digital)
Pin Configuration
Cycle Pin Group
Name
Value
Lock
Link
Pin Group Selection Mixed
Operation Mode Custom
Input/Output
LCD_CLK P515
LCD_DATA0 P914
LCD_DATA1 P915
LCD_DATA2 P903
LCD_DATA3 P902
LCD_DATA4 P910
LCD_DATA5 P911
LCD_DATA6 P912
LCD_DATA7 P913
LCD_DATA8 P904
LCD_DATA9 P207
LCD_DATA10 PB07
LCD_DATA11 PB06
LCD_DATA12 PB05
LCD_DATA13 PB01
LCD_DATA14 PB04
LCD_DATA15 PB03
LCD_DATA16 None
Module name: GLCDC
Pin Function
Pin Number
Summary BSP Clocks Pins Interrupts Event Links Stacks Components
Properties Problems Console

LCD 背光配置

- 新建 `r_gpt` stack：

- 配置背光 PWM 输出：

D/AVE 2D 配置

- 新建 `r_drw` stack：

I2C0 配置

- 新建 `r_iic_master` stack：

- 配置 `r_iic_master` stack：

- 配置 I2C0 引脚：

触摸中断配置

- 新建 `r_icu` stack：

- 配置 IRQ28：

- 配置 IRQ28 引脚：

RT-Thread Settings 配置

- 使能 RGB565 LCD、GT9147触摸驱动、使能 LVGL。

软件说明

LVGL 相关代码在 `./board/lvgl` 中。

- lv_port_disp.c

```
#include <lvgl.h>
#include <rtthread.h>
#include "ra8/lcd_config.h"
#include "hal_data.h"

static rt_sem_t _SemaphoreVsync = RT_NULL;
static uint8_t lvgl_init_flag = 0;

void DisplayVsyncCallback(display_callback_args_t *p_args)
{
    rt_interrupt_enter();
    if (DISPLAY_EVENT_LINE_DETECTION == p_args->event)
    {
        if (lvgl_init_flag != 0)
            rt_sem_release(_SemaphoreVsync);
    }
    rt_interrupt_leave();
}
```

```
static void vsync_wait_cb(lv_display_t *display)
{
    if (!lv_display_flush_is_last(display)) return;

    // If Vsync semaphore has already been set, clear it then wait
    rt_sem_take(_SemaphoreVsync, RT_WAITING_FOREVER);
}

static void disp_flush(lv_display_t *display, const lv_area_t *area
{
    if (!lv_display_flush_is_last(display)) return;

#if (BSP_CFG_DCACHE_ENABLED)
    int32_t size;
    /* Invalidate cache - so the HW can access any data written by
    size = sizeof(fb_background[0]);

    SCB_CleanInvalidateDCache_by_Addr(px_map, size);
#endif

    R_GLCDC_BufferChange(&g_display0_ctrl,
                         (uint8_t *) px_map,
                         (display_frame_layer_t) 0);
}

void lv_port_disp_init(void)
{
    static rt_device_t device;
    /* LCD Device Init */
    device = rt_device_find("lcd");
    RT_ASSERT(device != RT_NULL);

    _SemaphoreVsync = rt_sem_create("lvgl_sem", 1, RT_IPC_FLAG_PRIO

    if (RT_NULL == _SemaphoreVsync)
    {
        rt_kprintf("lvgl semaphore create failed\r\n");
        RT_ASSERT(0);
    }

    /*------------------------------------
     * Create a display and set a flush_cb
     * ------------------------------------*/
    lv_display_t *disp = lv_display_create(LV_HOR_RES_MAX, LV_VER_R
    lv_display_set_flush_cb(disp, disp_flush);
    lv_display_set_flush_wait_cb(disp, vsync_wait_cb);
    lv_display_set_buffers(disp, &fb_background[0][0], &fb_backgrou
```

```
    lvgl_init_flag = 1;
}
```

- lv_port_indev.c

```
#include <lvgl.h>
#include <rtdevice.h>

#include "gt9147.h"

#define DBG_TAG "lv_port_indev"
#define DBG_LVL DBG_LOG
#include <rtdbg.h>

#include "hal_data.h"

#define GT9147_IRQ_PIN   BSP_IO_PORT_05_PIN_02
#define GT9147_RST_PIN   BSP_IO_PORT_04_PIN_12

static rt_device_t touch_dev;
struct rt_touch_data *read_data;
static lv_indev_t *touch_indev;

volatile static rt_uint8_t touch_detect_flag = 0;

#if LVGL_VERSION_MAJOR < 9
static void touchpad_read(lv_indev_drv_t *indev, lv_indev_data_t *d
#else
static void touchpad_read(lv_indev_t *indev, lv_indev_data_t *data)
#endif
{
    if (touch_detect_flag != 1)
        return;

    rt_device_read(touch_dev, 0, read_data, 1);

    if (read_data->event == RT_TOUCH_EVENT_NONE)
        return;

    data->point.x = read_data->x_coordinate;
    data->point.y = read_data->y_coordinate;

    if (read_data->event == RT_TOUCH_EVENT_DOWN)
        data->state = LV_INDEV_STATE_PR;
    if (read_data->event == RT_TOUCH_EVENT_MOVE)
        data->state = LV_INDEV_STATE_PR;
    if (read_data->event == RT_TOUCH_EVENT_UP)
        data->state = LV_INDEV_STATE_REL;

    touch_detect_flag = 0;
```

```
    rt_device_control(touch_dev, RT_TOUCH_CTRL_ENABLE_INT, RT_NULL)
}

static rt_err_t rx_callback(rt_device_t dev, rt_size_t size)
{
    touch_detect_flag = 1;
    rt_device_control(dev, RT_TOUCH_CTRL_DISABLE_INT, RT_NULL);
    return 0;
}

rt_err_t gt9147_probe(rt_uint16_t x, rt_uint16_t y)
{
    void *id;

    touch_dev = rt_device_find("gt9147");
    if (touch_dev == RT_NULL)
    {
        rt_kprintf("can't find device gt9147\n");
        return -1;
    }

    if (rt_device_open(touch_dev, RT_DEVICE_FLAG_INT_RX) != RT_EOK)
    {
        rt_kprintf("open device failed!");
        return -1;
    }

    id = rt_malloc(sizeof(rt_uint8_t) * 8);
    rt_device_control(touch_dev, RT_TOUCH_CTRL_GET_ID, id);
    rt_uint8_t *read_id = (rt_uint8_t *)id;
    rt_kprintf("id = GT%d%d%d \n", read_id[0] - '0', read_id[1] - '

    rt_device_control(touch_dev, RT_TOUCH_CTRL_SET_X_RANGE, &x);  /
    rt_device_control(touch_dev, RT_TOUCH_CTRL_SET_Y_RANGE, &y);
    rt_device_control(touch_dev, RT_TOUCH_CTRL_GET_INFO, id);
    rt_kprintf("range_x = %d \n", (*(struct rt_touch_info *)id).ran
    rt_kprintf("range_y = %d \n", (*(struct rt_touch_info *)id).ran
    rt_kprintf("point_num = %d \n", (*(struct rt_touch_info *)id).p
    rt_free(id);

    rt_device_set_rx_indicate(touch_dev, rx_callback);

    read_data = (struct rt_touch_data *)rt_calloc(1, sizeof(struct
    if (!read_data)
    {
        return -RT_ENOMEM;
    }

    return RT_EOK;
}

#define RST_PIN   "p412"
```

```
#define INT_PIN   "p502"

rt_err_t rt_hw_gt9147_register(void)
{
    struct rt_touch_config cfg;
    rt_base_t int_pin = rt_pin_get(INT_PIN);
    rt_base_t rst_pin = rt_pin_get(RST_PIN);

    cfg.dev_name = "i2c0";
    cfg.irq_pin.pin = int_pin;
    cfg.irq_pin.mode = PIN_MODE_INPUT_PULLDOWN;
    cfg.irq_pin.mode = PIN_MODE_INPUT;
    cfg.user_data = &rst_pin;

    rt_hw_gt9147_init("gt9147", &cfg);

    gt9147_probe(800, 480);

    return RT_EOK;
}

void lv_port_indev_init(void)
{
#if LVGL_VERSION_MAJOR < 9
    static lv_indev_drv_t indev_drv;         /* Descriptor of a inp
    lv_indev_drv_init(&indev_drv);           /* Basic initializatio
    indev_drv.type = LV_INDEV_TYPE_POINTER;  /* Touch pad is a poin
    indev_drv.read_cb = touchpad_read;       /* Set your driver fun

    /* Register the driver in LVGL and save the created input devic
    touch_indev = lv_indev_drv_register(&indev_drv);
#else
    /*Register a touchpad input device*/
    touch_indev = lv_indev_create();
    lv_indev_set_type(touch_indev, LV_INDEV_TYPE_POINTER);
    lv_indev_set_read_cb(touch_indev, touchpad_read);
#endif
    /* Register touch device */
    rt_err_t res = rt_hw_gt9147_register();
    RT_ASSERT(res == RT_EOK);
}
```

编译&下载

- RT-Thread Studio：在RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

运行效果

复位开发板后就能在 LCD 屏幕上看到 LVGL Demo 正在运行。

6. 多核通信篇

6.1. RPMsg-Lite 使用说明（Core0 Master）

中文 | **English**

简介

本工程运行于 **RA8P1 的 Core0（Cortex-M85，标称 1 GHz）**，作为双核系统的主控工程：

- 负责 **系统初始化、外设资源统筹** 与 **Core1 启动/管理**；
- 通过 **RPMsg-Lite** 在共享内存上建立与 Core1 的 **消息通道**（virtqueue/vring），用于命令下发、数据交换、事件通知等。

主要特性

- **多核通信**：RPMsg-Lite master 侧初始化、端点创建、消息收发、阻塞/非阻塞队列。
- **RT-Thread支持**：独立通信线程、安全队列、事件同步。
- **共享内存与缓存安全**：预留 **非缓存/强制一致性** 的共享内存段；提供内存屏障与 Cache 操作要点。
- **启动次序清晰**：Core0 负责装载/拉起 Core1，并进行 **Link-Up**/握手。
- **可裁剪的协议层**：提供简易 **Ping/Pong** 与 **帧格式** 示例，便于扩展业务指令集。

RPMsg-Lite 简介

RPMsg-Lite 是一个轻量级远程处理器间通信（Remote Processor Messaging, RPMsg）实现，主要用于多核处理器系统中，实现不同核心之间的数据交换和消息传递。它基于 **OpenAMP** 框架，利用共享内存（Shared Memory）和虚拟通道（Virtio Channels）机制，在资源受限的嵌入式系统中实现高效、可靠的跨核通信。

一、设计背景

在多核 MCU 或 SoC 系统中，不同核心可能运行不同类型的任务，例如：

- Cortex-M 核心运行实时控制任务

- Cortex-A 核心运行操作系统或高层应用
  这时核心之间需要高效的通信机制来交换数据或触发事件。RPMsg-Lite 提供了一个轻量、易用、可靠的消息传递解决方案。

二、核心特性

1. **轻量级**

- 相比完整的 RPMsg 实现，RPMsg-Lite 占用的内存和 CPU 资源更少，非常适合资源受限的嵌入式系统。
- 核心数据结构和队列机制经过优化，减少内存拷贝。

2. **跨核心通信**

- 支持不同核心之间的通信。
- 利用共享内存作为数据缓冲区，消息通过虚拟通道进行传递。

3. **可靠性**

- 提供可靠的消息发送机制，保证消息不会丢失或重复。
- 可配置消息队列长度，实现高可靠性数据传输。

4. **灵活性**

- 支持静态或动态创建通信通道，适应不同系统架构。

5. **简单接口**

- 提供易用的 API，包括消息发送、接收和通道管理接口。
- 开发者无需了解底层共享内存或中断机制即可快速集成。

三、工作原理

RPMsg-Lite 基于 **virtio** 虚拟设备机制：

1. 共享内存（Shared Memory）用于存放消息数据和队列信息。
2. 发送核心将消息写入共享内存队列，并通过中断通知接收核心。
3. 接收核心读取消息，并根据应用需求进行处理。
4. 消息处理完成后，接收核心可回馈确认信息，保证通信可靠性。

四、应用场景

- **双核或多核 MCU 系统**：例如 RA8P1 双核 MCU 中 M85 处理高性能任务，M33 处理实时任务，两核通过 RPMsg-Lite 通信。
- **实时任务分离**：将时间敏感任务和高层应用任务分开运行，提高系统性能。

- **资源受限的嵌入式系统**：占用内存小、通信开销低，适合工业控制、机器人、汽车电子等领域。

硬件说明

无

FSP 配置

- 新建两个 `r_ipc` stack：

- 配置 IPC0：

- 配置 IPC1：

RT-Thread Settings 配置

- 使能 sub-nuclear startup 和 rpmsg-lite。

示例工程说明

示例程序位于 `/board/ports/rpmsg-adapt`。

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

编译完成后，将开发板的 USB-DBG 接口与 PC 机连接，然后将固件下载至开发板。

下载方式

目前 DAP-Link 还不支持烧录 Core1 固件，所以需要**使用 Renesas Flash Programmer 工具来烧录固件**，具体烧录步骤如下：

1. 下载并安装 Renesas Flash Programmer 工具：Renesas Flash Programmer (Programming GUI) | Renesas。
2. 将开发板的 USB-DEV 接口与 PC 机连接。
3. 创建工程：开发板需要进入 BOOT 模式（按住 BOOT 按钮，点按一下 RESET 按钮）

4. 选择要烧录的固件（Core0 和 Core1 的固件）：

5. 烧录固件:（此时开发板需要处于 BOOT 模式）

运行效果

在 Core0 终端中输入 `rp_master` 命令启动 rpmsg master。

6.2. RPMsg-Lite 使用说明（Core1 Remote）

中文 | **English**

简介

本工程运行于 **RA8P1 的 Core1（Cortex-M33，250 MHz）**，作为双核系统的从核（Remote）工程：

- 通过 **RPMsg-Lite** 接收来自 Core0（Master）的指令和数据；
- 处理特定任务（如通信处理、数据采集、安全控制、低功耗管理），并将结果回传 Core0；
- 支持独立应用逻辑和消息驱动机制。

主要特性

- **多核通信**：RPMsg-Lite Remote 端初始化、端点创建、消息收发。
- **RT-Thread支持**：独立线程处理收发消息，支持事件/队列同步。
- **共享内存与缓存安全**：使用 Core0 预留的共享内存，保证缓存一致性。
- **握手机制**：与 Core0 Master 进行 HELLO/READY/ACK 通信。
- **可扩展协议层**：轻量协议示例（Ping/Pong、命令帧、数据流）。

RPMsg-Lite 简介

RPMsg-Lite 是一个轻量级远程处理器间通信（Remote Processor Messaging, RPMsg）实现，主要用于多核处理器系统中，实现不同核心之间的数据交换和消息传递。它基于 **OpenAMP** 框架，利用共享内存（Shared Memory）和虚拟通道（Virtio Channels）机制，在资源受限的嵌入式系统中实现高效、可靠的跨核通信。

一、设计背景

在多核 MCU 或 SoC 系统中，不同核心可能运行不同类型的任务，例如：

- Cortex-M 核心运行实时控制任务
- Cortex-A 核心运行操作系统或高层应用
  这时核心之间需要高效的通信机制来交换数据或触发事件。RPMsg-Lite 提供了一个轻量、易用、可靠的消息传递解决方案。

二、核心特性

1. **轻量级**

- 相比完整的 RPMsg 实现，RPMsg-Lite 占用的内存和 CPU 资源更少，非常适合资源受限的嵌入式系统。
- 核心数据结构和队列机制经过优化，减少内存拷贝。

2. **跨核心通信**

- 支持多核心之间的通信。
- 利用共享内存作为数据缓冲区，消息通过虚拟通道进行传递。

3. **可靠性**

- 提供可靠的消息发送机制，保证消息不会丢失或重复。
- 可配置消息队列长度，实现高可靠性数据传输。

4. **灵活性**

- 支持静态或动态创建通信通道，适应不同系统架构。

5. **简单接口**

- 提供易用的 API，包括消息发送、接收和通道管理接口。
- 开发者无需了解底层共享内存或中断机制即可快速集成。

三、工作原理

RPMsg-Lite 基于 **virtio** 虚拟设备机制：

1. 共享内存（Shared Memory）用于存放消息数据和队列信息。
2. 发送核心将消息写入共享内存队列，并通过中断通知接收核心。
3. 接收核心读取消息，并根据应用需求进行处理。
4. 消息处理完成后，接收核心可回馈确认信息，保证通信可靠性。

四、应用场景

- **双核或多核 MCU 系统**：例如 RA8P1 双核 MCU 中 M85 处理高性能任务，M33 处理实时任务，两核通过 RPMsg-Lite 通信。
- **实时任务分离**：将时间敏感任务和高层应用任务分开运行，提高系统性能。
- **资源受限的嵌入式系统**：占用内存小、通信开销低，适合工业控制、机器人、汽车电子等领域。

硬件说明

无

FSP 配置

- 新建两个 `r_ipc` stack：

- 配置 IPC0：

- 配置 IPC1：

RT-Thread Settings 配置

- 使能 sub-nuclear startup 和 rpmsg-lite。

示例工程说明

示例程序位于 `/board/ports/rpmsg-adapt`。

编译&下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 Titan Board 资源包，然后创建新工程，执行编译。

烧录方式

目前 DAP-Link 还不支持烧录 Core1 固件，所以需要**使用 Renesas Flash Programmer 工具来烧录固件**，具体烧录步骤如下：

1. 下载并安装 Renesas Flash Programmer 工具：Renesas Flash Programmer (Programming GUI) | Renesas。
2. 将开发板的 USB-DEV 接口与 PC 机连接。
3. 创建工程：开发板需要进入 BOOT 模式（按住 BOOT 按钮，点按一下 RESET 按钮）

4. 选择要烧录的固件：

5. 烧录固件（此时开发板需要处于 BOOT 模式）：

运行效果

Core1 会自动启动 rp_remote，等 Core0 启动 rp_master 后就能进行核间通信了。