RT-THREAD 星火 1 号用户手册

RT-THREAD 文档中心

上海睿赛德电子科技有限公司版权 **@2023**

WWW.RT-THREAD.ORG

Wednesday 20th September, 2023

版本和修订

Date	Version	Author	Note
2023-06-30	V0.1.0	RT-Thread	初始版本
2023-07-06	V0.2.0	RT-Thread	发布版本
2023-08-11	V1.0.0	RT-Thread	SDK V1.0.0 版本对应文档
2023-09-20	V1.1.0	RT-Thread	SDK V1.1.0 版本对应文档

目录

第 1 章

简介

“星火 1 号”，一款专为工程师和高校学生设计的嵌入式 RTOS 开发学习板。在这个科技飞速发展的时代，嵌入式系统已经成为了现代工业、交通、通信等众多领域的核心驱动力。而 RTOS 实时操作系统作为嵌入式领域的基石，更是工程师们必须熟练掌握的核心技术。作为业界主流的 RTOS 实时操作系统 RT-Thread，我们有义务帮助更多开发者掌握这项技术。为此，我们精心打造了一款专为工程师和高校学生设计的嵌入式开发学习板。

星火 1 号主控选用了目前行业中比较常用且学习门槛较低的 STM32F407，性能强劲、功能丰富，完全能够满足嵌入式入门的需求。此开发板不仅具有众多的板载资源（Flash 存储、WIFI 通信、多个传感器），还支持丰富的扩展接口，让您轻松实现各种复杂的应用场景。通过使用这款开发学习板，您将能够深入了解 RTOS/RT-Thread 的工作原理，提升自己的技能水平，为当前以及未来的职业生涯做好充分准备。

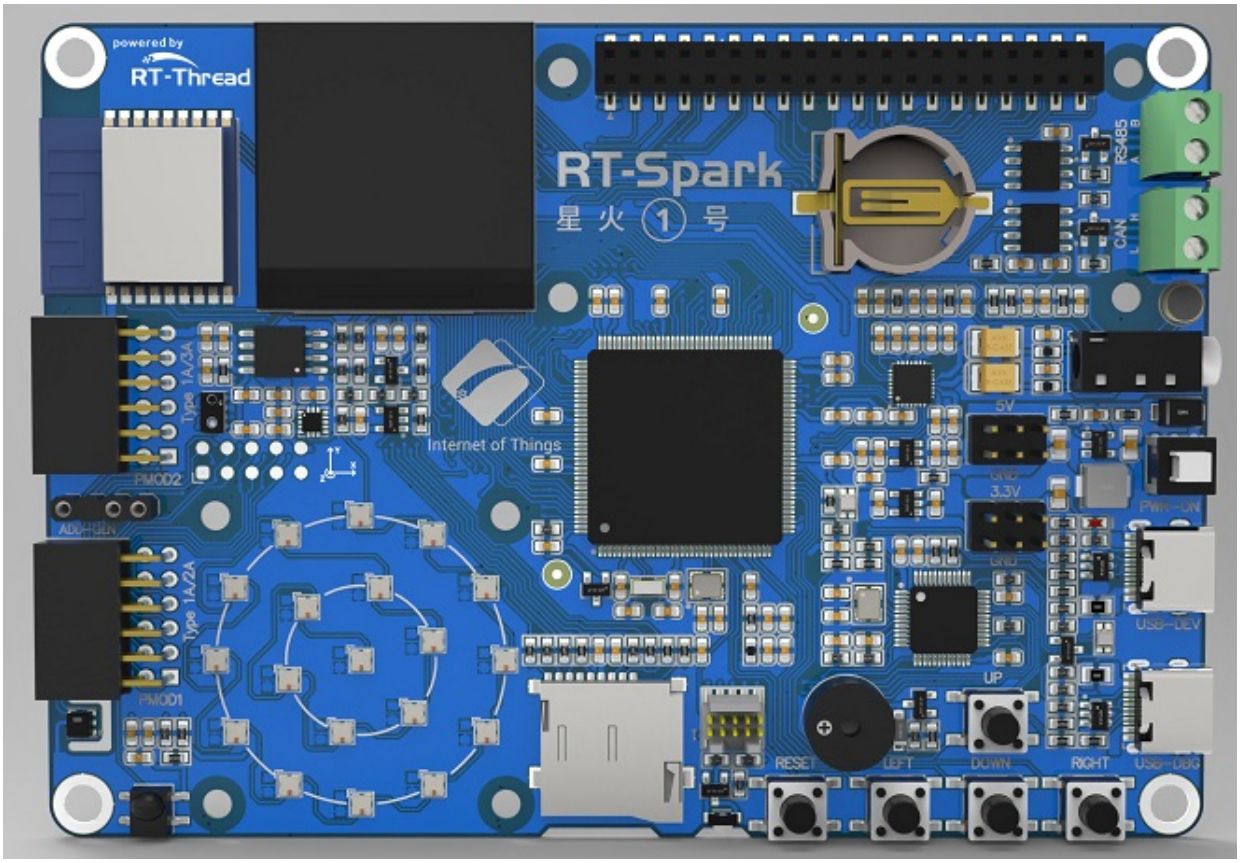

图 1.1: *board-small*

1.1 目录结构

```
|-- README.md
|-- docs
|-- libraries
|   |-- Board_Drivers
|   |-- HAL_Drivers
|   `-- STM32F4xx_HAL
|-- projects
|-- rt-thread
`-- sdk-bsp-rt-spark.yaml
```

- docs：星火 1 号原理图、用户手册等
- libraries：STM32F4 固件库、通用外接驱动程序
- projects：示例项目文件夹，包括各种示例代码
- rt-thread：rt-thread 源代码
- sdk-bsp-rt-spark.yaml：描述星火 1 号的硬件信息

1.2 学习路线

项目文件夹按照不同的学习阶段添加了数字编号，如 01_kernel，02_basic_led_blink，分别表示两个不同的学习阶段，编号小则内容相对简单，建议初学者按照数字编号大小顺序进行学习，循序渐进掌握 RT-Thread。

示例项目文件夹和相应学习阶段如下所示：

阶段	序号	项目实现功能	项目文件夹名称
01，入门 RT-Thread 内核	01	RT-Thread 内核学习	01_kernel
02，简单外设的使用	02	LED 闪烁例程	02_basic_led_blink
	03	RGB LED 例程	02_basic_rgb_led
	04	按键输入例程	02_basic_key
	05	蜂鸣器和 LED 控制例程	02_basic_irq_beep
	06	红外遥控例程	02_basic_ir
	07	RTC 和 RTC 闹钟的使用例程	02_basic_rtc
03，板载外设模块的使用	08	LCD 显示例程	03_driver_lcd
	09	AHT10 温湿度传感器例程	03_driver_temp_humi
	10	AP3216C 接近与光强传感器例程	03_driver_als_ps
	11	ICM20608 六轴传感器例程	03_driver_axis
	12	CAN 通信例程	03_driver_can
	13	LED MATRIX 闪烁例程	03_driver_led_matrix
04，RT-Thread 组件的使用	14	USB 鼠标例程	04_component_usb_mouse
	15	TF 卡文件系统例程	04_component_fs_tf_card
	16	低功耗例程	04_component_pm
	17	Flash 分区管理例程	04_component_fal
	18	KV 参数存储例程	04_component_kv
	19	SPI Flash 文件系统例程	04_component_fs_flash
05，IoT 相关组件的使用	20	WiFi 管理例程	05_iot_wifi_manager
	21	MQTT 协议通信例程	05_iot_mqtt
	22	HTTP Client 功能实现例程	05_iot_http_client
	23	MEMBEDTLS 例程	05_iot_mbedtls
	24	Ymodem 协议固件升级例程	05_iot_ota_ymodem
	25	HTTP 协议固件升级例程	05_iot_ota_http
	26	网络小工具集使用例程	05_iot_netutils

阶段	序号	项目实现功能	项目文件夹名称
	27	中国移动 OneNET 云平台接入例程	05_iot_cloud_onenet
	28	阿里云物联网平台接入例程	05_iot_cloud_ali_iotkit
	29	使用 Web 服务器组件：WebNet	05_iot_web_server
06，综合 Demo 学习	30	LVGL 例程	06_demo_lvgl
	31	MicroPython 例程	06_demo_micropython
	32	板载 LED matrix 和 RS485 驱动例程	06_demo_rs485_led_matrix
	33	nes 模拟器实验	06_demo_nes_simulator
	34	开发板综合 Demo（出厂 Demo）	06_demo_factory
07，教育套件对应模块例程	35	矩阵键盘模块例程	07_module_key_matrix
	36	ENC28J60 网络模块例程	07_module_spi_eth_enc28j60
	37	超声波测距模块例程	07_module_ultrasonic_sr04
	38	ws2812 led 灯带例程	07_module_ws2812_led

1.3 使用

sdk-bsp-stm32f407-spark 支持 RT-Thread Studio 和 MDK 开发。

1.3.1 RT-Thread Studio 开发

1. 打开 RT-Thread Studio 的包管理器，安装 星火1 号开发板 资源包
2. 安装完成后，选择基于 BSP 创建工程即可

1.3.2 MDK 开发

为了避免 SDK 在持续更新中，每一个 `projects` 都创建一份 `rt-thread` 文件夹和 `libraries` 文件夹导致的 SDK 越来越臃肿，所以这些通用文件夹被单独提取了出来。这样就会导致直接打开 `MDK` 的工程编译会提示缺少上述两个文件夹的文件，我们使用如下步骤解决这个问题：

1. 双击某个 `project` 目录下的 `mklinks.bat` 文件，或者使用 Env 工具执行 mklinks.bat 命令，分别为 `rt-thread` 及 `libraries` 文件创建符号链接。
2. 查看目录下是否有 `rt-thread` 和 `libraries` 的文件夹图标。
3. 使用 Env 工具执行 scons –target=mdk5 更新 MDK5 工程文件。

1.4 交流平台

对星火 1 号感兴趣的小伙伴可以加入 QQ 群 - RT-Thread 星火学习板群号: 839583041、852752783。

第 2 章

LED 闪烁例程

2.1 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 中的红色 LED 不间断闪烁。

2.2 硬件说明

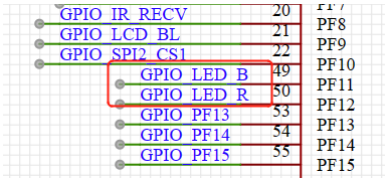

图 2.1: *LED 连接单片机引脚*

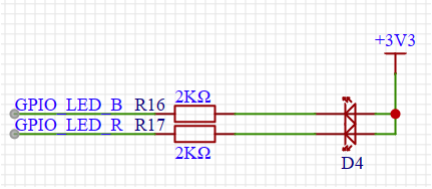

图 2.2: *LED 电路原理图*

如上图所示，RBG-LED 属于共阳 LED，阴极分别与单片机的引脚连接，其中红色 LED 对应 PF12 引脚。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：

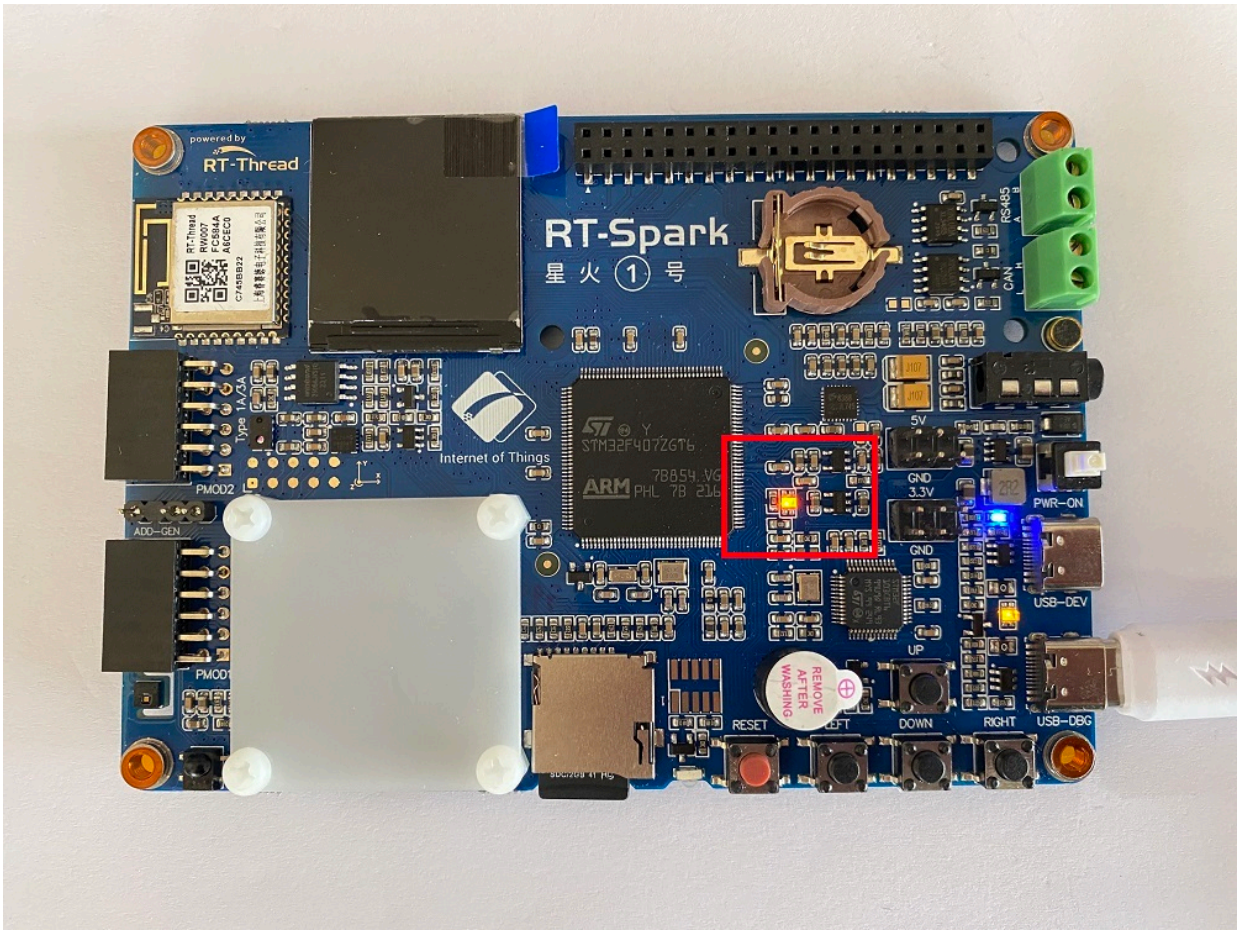

图 2.3: *LED 位置*

2.3 软件说明

本例程的源码位于 /projects/02_basic_led_blink。

闪灯的源代码位于 applications/main.c 中。首先定义了一个宏 LED_PIN ，代表闪灯的 LED 引脚编号，然后与 PIN_LED_R（PF12）对应：

```
/* 配 置 LED 灯 引 脚 */
#define LED_PIN PIN_LED_R
```

在 main 函数中，将该引脚配置为输出模式，并在下面的 while 循环中，周期性（500 毫秒）开关 LED，同时输出一些日志信息。

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

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

/* 配置 LED 灯引脚 */
```

```
#define PIN_LED_B               GET_PIN(F, 11)      // PF11 :  LED_B        --> LED
#define PIN_LED_R               GET_PIN(F, 12)      // PF12 :  LED_R        --> LED

int main(void)
{
    unsigned int count = 1;

    /* 设置 LED 引脚为输出模式 */
    rt_pin_mode(PIN_LED_R, PIN_MODE_OUTPUT);

    while (count > 0)
    {
        /* LED 灯亮 */
        rt_pin_write(PIN_LED_R, PIN_LOW);
        LOG_D("led on, count: %d", count);
        rt_thread_mdelay(500);

        /* LED 灯灭 */
        rt_pin_write(PIN_LED_R, PIN_HIGH);
        LOG_D("led off");
        rt_thread_mdelay(500);

        count++;
    }

    return 0;
}
```

2.4 运行

2.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

2.4.2 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，红色 LED 会周期性闪烁，如下图所示：

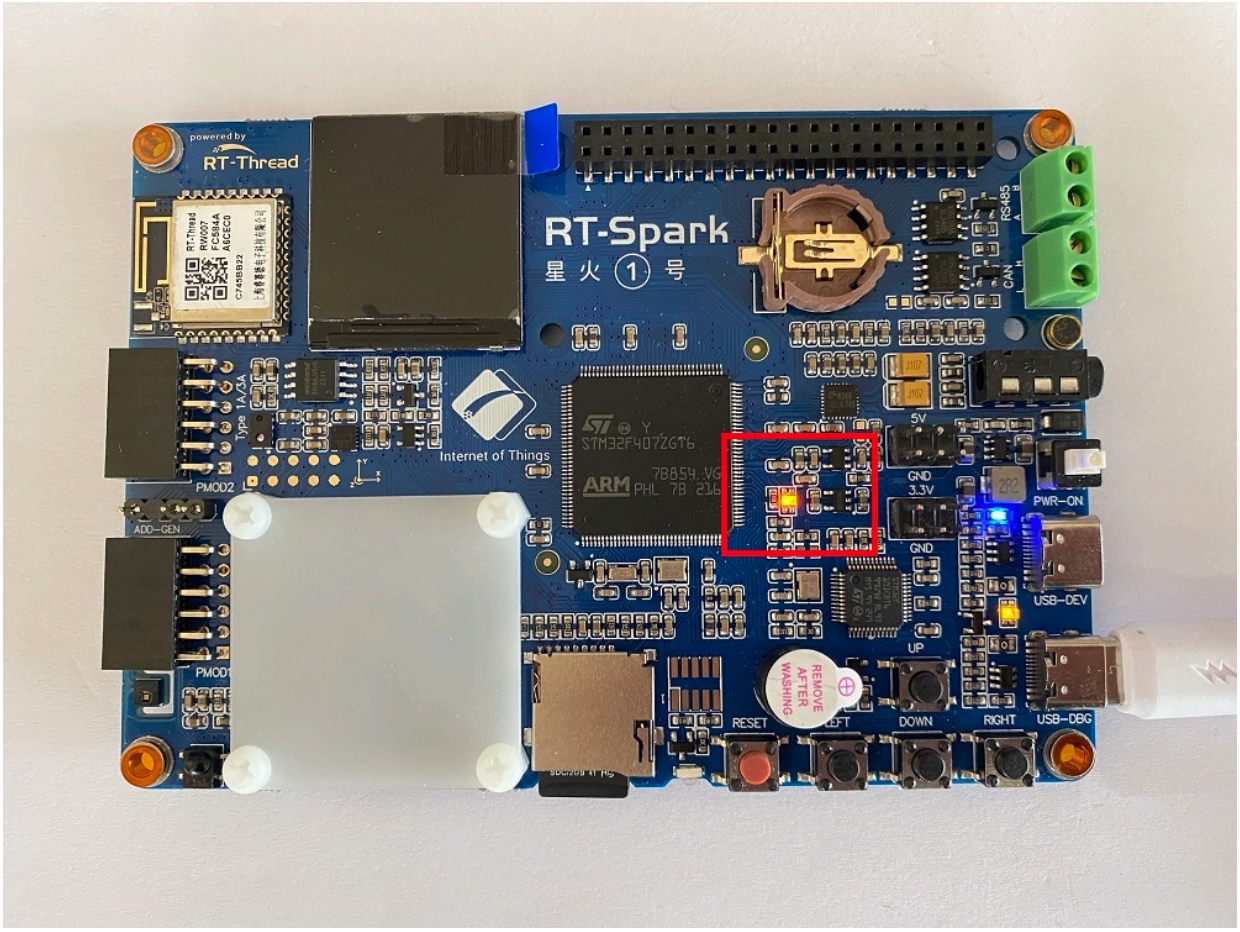

图 2.4: *红灯亮起*

此时也可以在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置 115200 8 1 N 。开发板的运行日志信息即可实时输出出来。

```
[D/main] led on, count: 1
[D/main] led off
[D/main] led on, count: 2
[D/main] led off
[D/main] led on, count: 3
[D/main] led off
[D/main] led on, count: 4
[D/main] led off
[D/main] led on, count: 5
[D/main] led off
[D/main] led on, count: 6
[D/main] led off
[D/main] led on, count: 7
[D/main] led off
[D/main] led on, count: 8
```

2.5 注意事项

如果想要修改 LED_PIN 宏定义，可以参考 /drivers/drv_gpio.h 文件，该文件中里有定义单片机的其它引脚编号。

2.6 引用参考

- 设备与驱动：PIN 设备

2.7 小实验

思考一下如何点亮蓝色 LED 呢？

第 3 章

RGB LED 例程

3.1 简介

本例程主要功能是让板载的 RGB-LED 灯周期性地变换颜色。

3.2 硬件说明

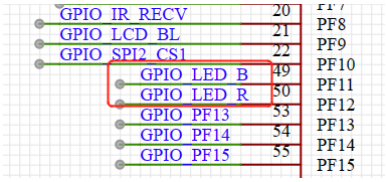

图 3.1: *RGB 连接单片机引脚*

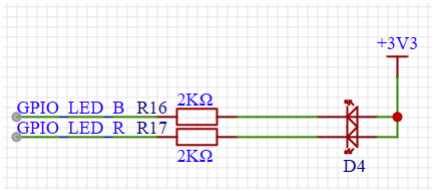

图 3.2: *RGB 电路原理图*

如上图所示，RGB-LED 属于共阳 LED，阴极分别与单片机的引脚连接，其中红色 LED 对应 PF12 号引脚，蓝色 LED 对应 PF11 号引脚。单片机对应的引脚输出低电平即可点亮对应的 LED，输出高电平则会熄灭对应的 LED。

注意：由于生产批次不同，具体 LED 灯的颜色可能会不一样，以实际开发板上颜色为准。

RGB-LED 在开发板中的位置如下图所示：

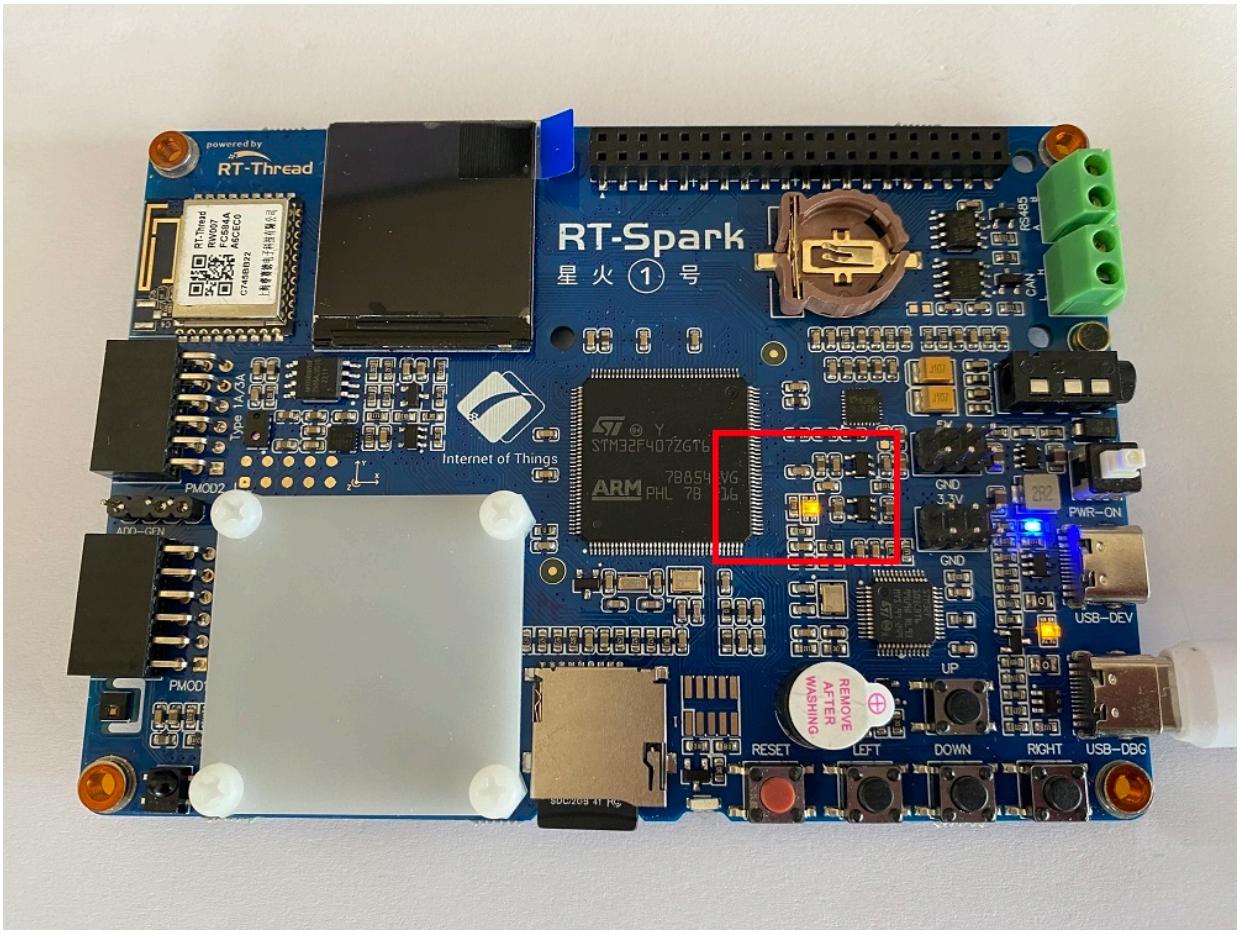

图 3.3: *RGB LED 位置*

3.3 软件说明

本例程的源码位于 /projects/02_basic_rgb_led。

RGB-LED 对应的单片机引脚定义可以通过查阅头文件 /drivers/drv_gpio.h 获知。

```
/* 配置 LED 灯引脚 */
#define PIN_LED_B               GET_PIN(F, 11)      // PF11 :  LED_B        --> LED
#define PIN_LED_R               GET_PIN(F, 12)      // PF12 :  LED_R        --> LED
```

RGB-LED 灯变换的源代码位于 /projects/02_basic_rgb_led/applications/main.c 中。

```
int main(void)
{
    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(PIN_LED_R, PIN_MODE_OUTPUT);
    rt_pin_mode(PIN_LED_B, PIN_MODE_OUTPUT);
    rt_pin_write(PIN_LED_R, LED_OFF);
```

```
    rt_pin_write(PIN_LED_B, LED_OFF);

    do
    {

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

        /* 控制 RGB 灯 */
        rt_pin_write(PIN_LED_R, _blink_tab[group_current][0]);
        rt_pin_write(PIN_LED_B, _blink_tab[group_current][1]);

        /* 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");

        count++;

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

3.4 运行

3.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

3.4.2 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，如下图所示：

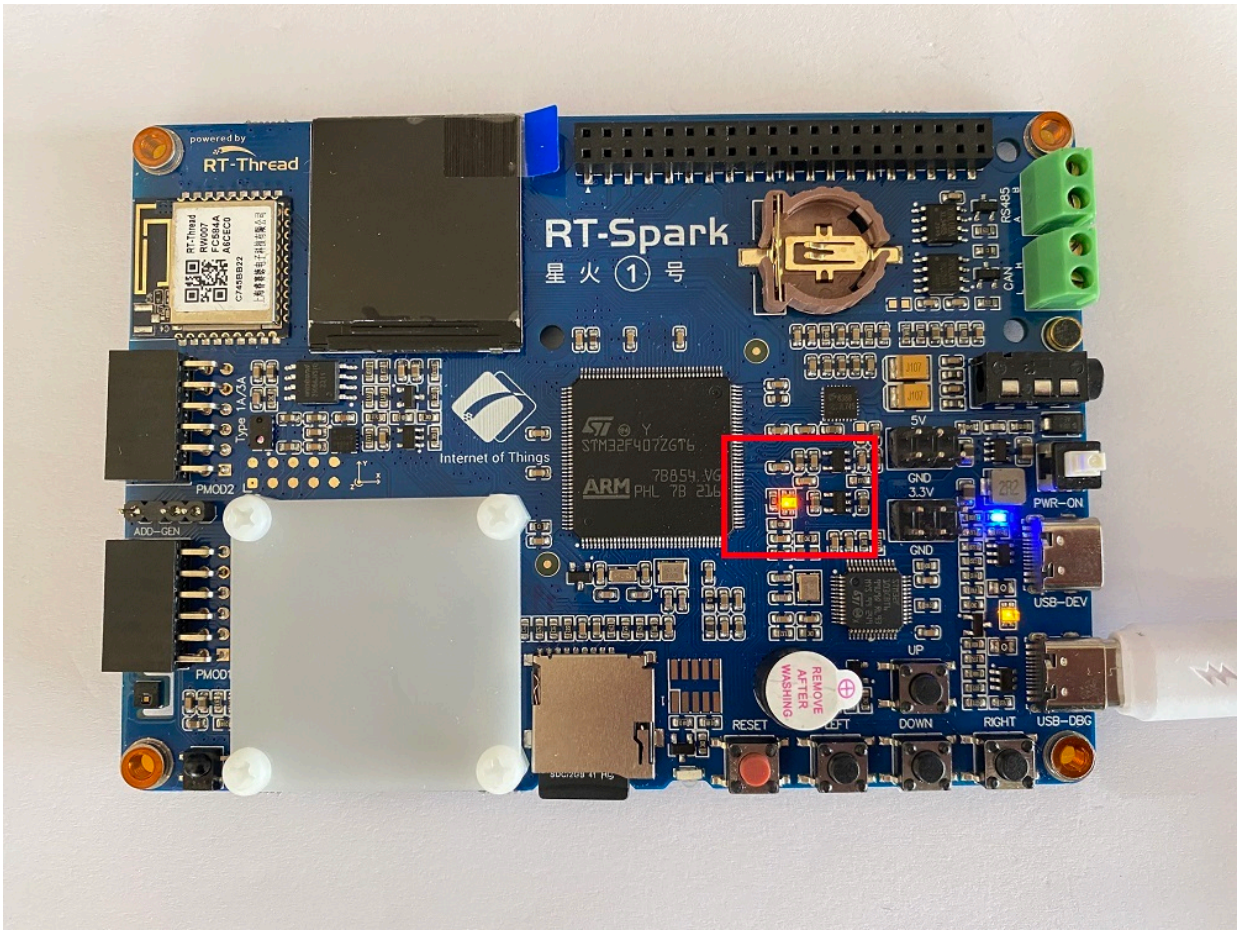

图 3.4: 红色（或其他颜色）灯运行

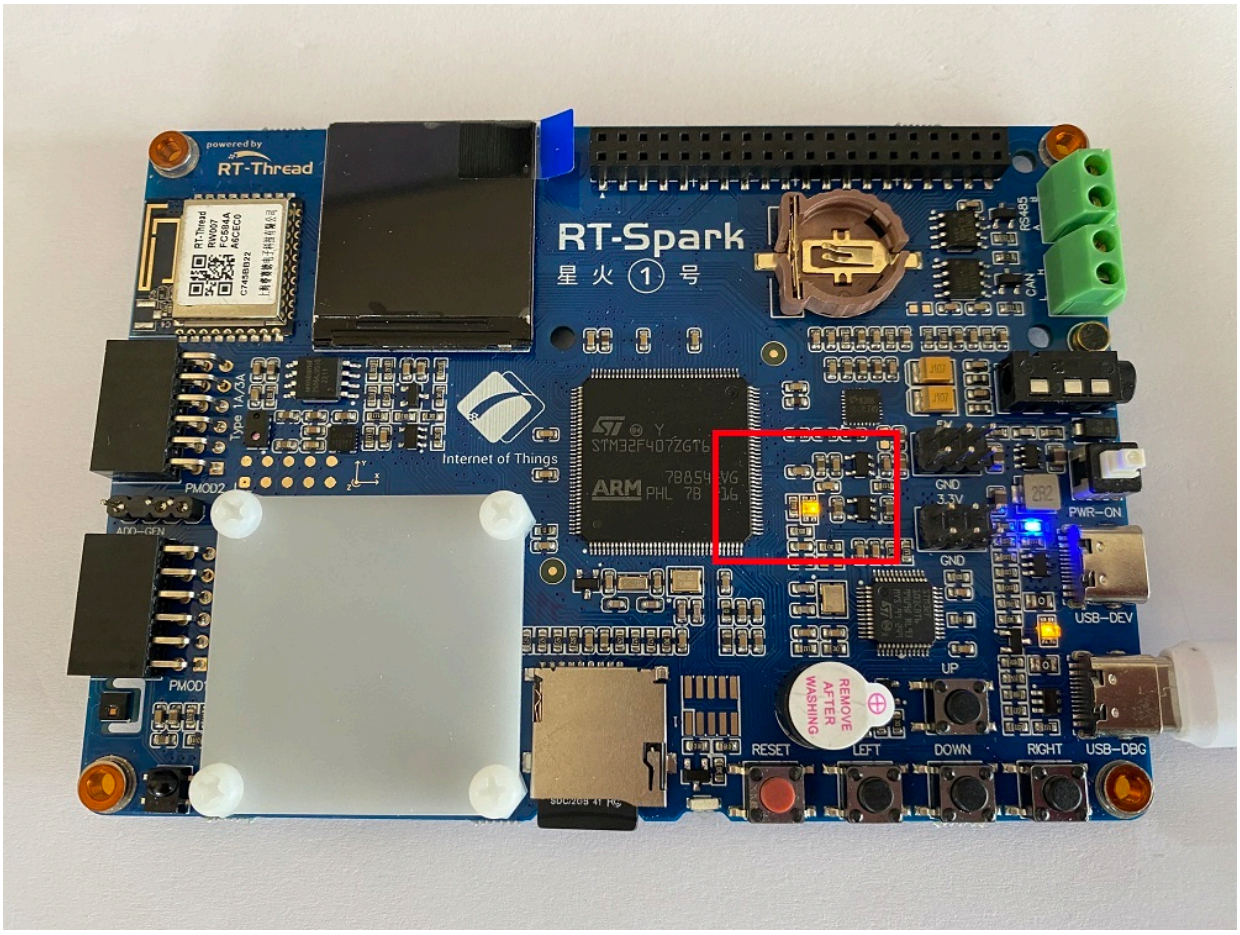

图 3.5: *蓝色（或其他颜色）灯运行*

此时也可以在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置 115200 8 1 N 。开发板的运行日志信息即可实时输出出来。

```
[D/main] group: 0 | red led [ON ] | | blue led [ON ]
[D/main] group: 1 | red led [OFF] | | blue led [ON ]
[D/main] group: 2 | red led [ON ] | | blue led [OFF]
[D/main] group: 3 | red led [ON ] | | blue led [ON ]
[D/main] group: 4 | red led [OFF] | | blue led [OFF]
[D/main] group: 5 | red led [ON ] | | blue led [OFF]
[D/main] group: 6 | red led [OFF] | | blue led [ON ]
[D/main] group: 7 | red led [OFF] | | blue led [OFF]
[D/main] group: 0 | red led [ON ] | | blue led [ON ]
[D/main] group: 1 | red led [OFF] | | blue led [ON ]
[D/main] group: 2 | red led [ON ] | | blue led [OFF]
[D/main] group: 3 | red led [ON ] | | blue led [ON ]
[D/main] group: 4 | red led [OFF] | | blue led [OFF]
[D/main] group: 5 | red led [ON ] | | blue led [OFF]
[D/main] group: 6 | red led [OFF] | | blue led [ON ]
[D/main] group: 7 | red led [OFF] | | blue led [OFF]
```

3.5 注意事项

暂无

3.6 引用参考

- 设备与驱动：PIN 设备

第 4 章

按键输入例程

4.1 简介

本例程主要功能是通过板载的按键 KEY0(LEFT) 控制 RGB-LED 中的红色 LED 的亮灭。

4.2 硬件说明

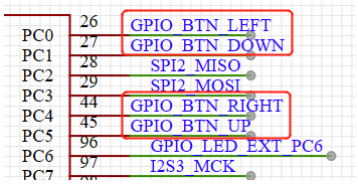

图 4.1: *连接单片机引脚*

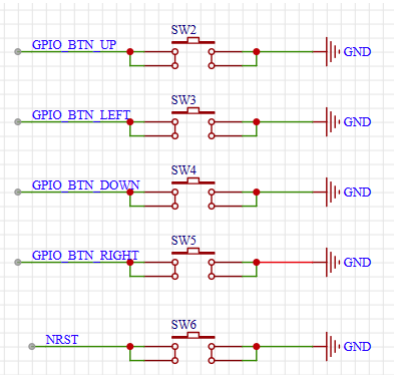

图 4.2: *电路原理图*

如上图所示，KEY0(LEFT) 引脚连接单片机 PC0(LEFT) 引脚, KEY0(LEFT) 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：

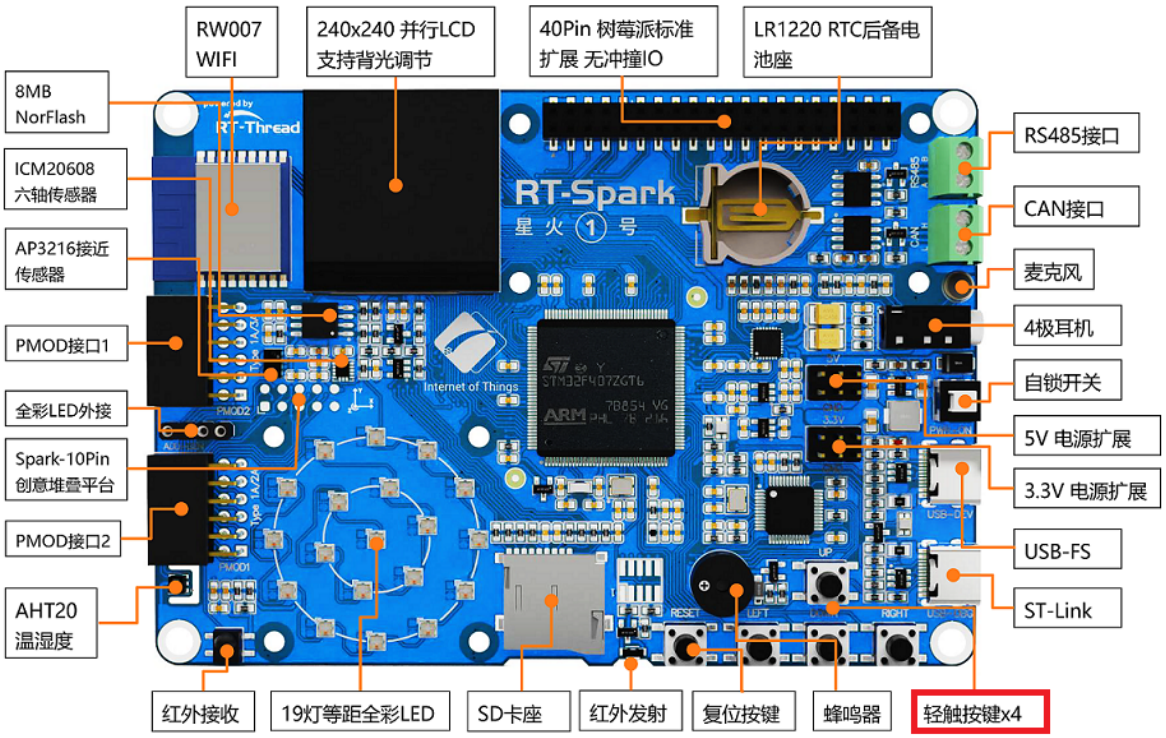

图 4.3: *按键位置*

4.3 软件说明

本例程的源码位于 /projects/02_basic_key。

KEY0(LEFT) 对应的单片机引脚定义。

```
#define PIN_KEY0        GET_PIN(C, 0)     // PC0:  KEY0         --> KEY
```

按键输入的源代码位于 /projects/02_basic_key/applications/main.c 中。在 main 函数中，首先为了实验效果清晰可见，板载 RGB 红色 LED 作为 KEY0(LEFT) 的状态指示灯，设置 RGB 红灯引脚的模式为输出模式，然后设置 PIN_KEY0 引脚为输入模式，最后在 while 循环中通过 rt_pin_read(PIN_KEY0) 判断 KEY0(LEFT) 的电平状态，并作 50ms 的消抖处理，如果成功判断 KEY0(LEFT) 为低电平状态（即按键按下）则打印输出 “KEY0 pressed!” 并且指示灯亮，否则指示灯熄灭。

```
int main(void)
{
    unsigned int count = 1;

    /* 设置 RGB 红灯引脚的模式为输出模式 */
    rt_pin_mode(PIN_LED_R, PIN_MODE_OUTPUT);
    /* 设置 KEY0 引脚的模式为输入上拉模式 */
    rt_pin_mode(PIN_KEY0, PIN_MODE_INPUT_PULLUP);

    while (count > 0)
```

```
    {
        /* 读取按键 KEY0 的引脚状态 */
        if (rt_pin_read(PIN_KEY0) == PIN_LOW)
        {
            rt_thread_mdelay(50);
            if (rt_pin_read(PIN_KEY0) == PIN_LOW)
            {
                /* 按键已被按下，输出 log，点亮 LED 灯 */
                LOG_D("KEY0 pressed!");
                rt_pin_write(PIN_LED_R, PIN_LOW);
            }
        }
        else
        {
            /* 按键没被按下，熄灭 LED 灯 */
            rt_pin_write(PIN_LED_R, PIN_HIGH);
        }
        rt_thread_mdelay(10);
        count++;
    }
    return 0;
}
```

4.4 运行

4.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

4.4.2 运行效果

按下复位按键重启开发板，按住 KEY0(LEFT) 可以观察到开发板上 RBG 红色 LED 指示灯的亮起，松开 KEY0(LEFT) 可以观察到开发板上的 RBG 红色 LED 指示灯熄灭。按住 KEY0(LEFT) 按键后如下图所示：

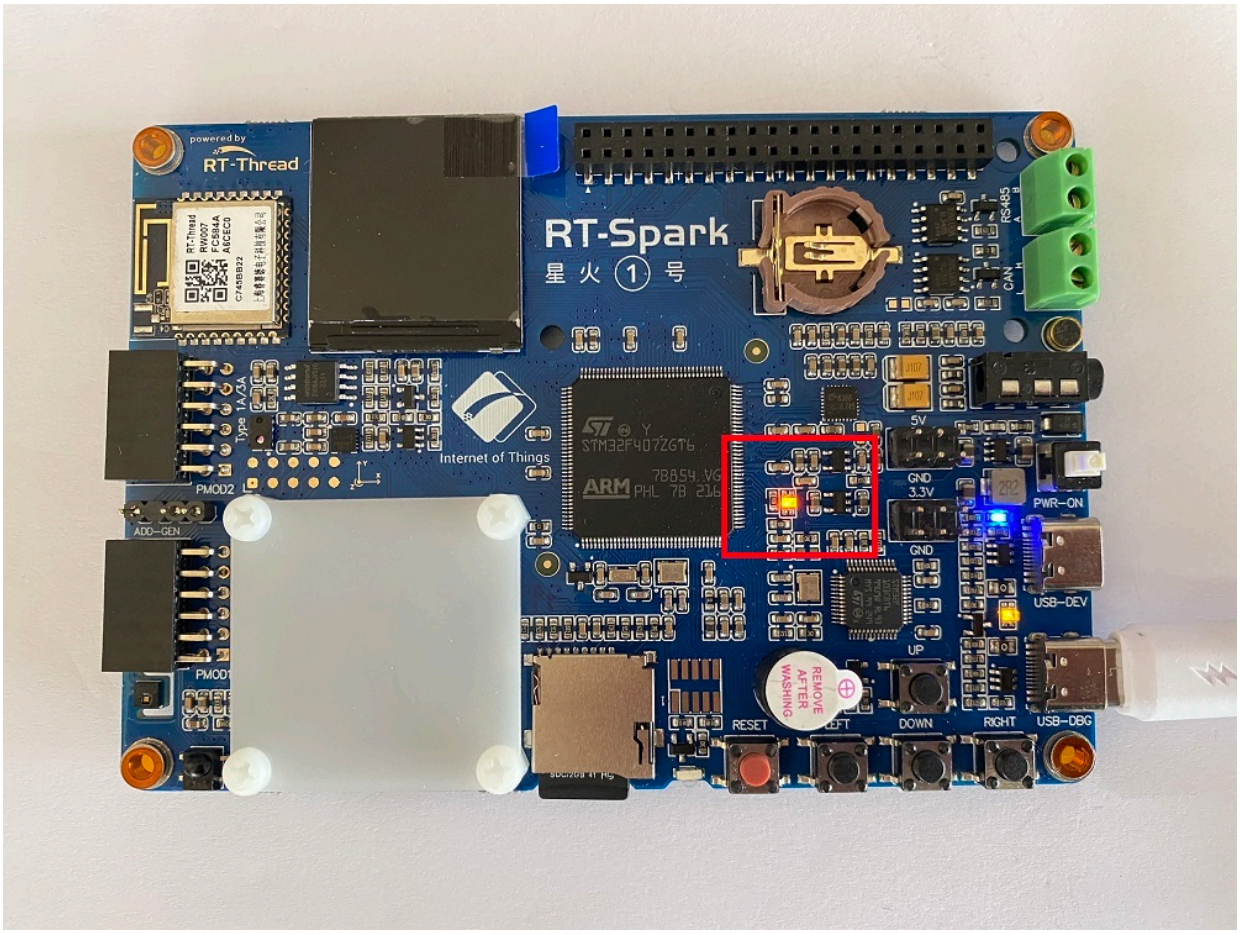

图 4.4: *按住 KEY0(LEFT) 红灯亮起*

此时也可以在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置 115200 8 1 N 。开发板的运行日志信息即可实时输出出来。

```
[D/main] KEY0 pressed!
[D/main] KEY0 pressed!
[D/main] KEY0 pressed!
```

4.5 注意事项

暂无。

4.6 引用参考

- 设备与驱动：PIN 设备

第 5 章

外部中断控制蜂鸣器例程

5.1 简介

本例程主要功能为使用外部中断控制蜂鸣器，当按下 WK_UP 蜂鸣器响起，松开 WK_UP 后蜂鸣器关闭。

中断是计算机系统中的一个重要概念，用于处理来自外部设备或软件的事件或信号。当一个事件发生时，例如用户按下键盘上的一个键或者硬盘传输数据完成，系统会发出一个中断信号，以通知 CPU 停止当前执行的任务并处理该事件。中断的目的是实现多任务处理和异步事件处理。它允许计算机在执行某个任务时，能够立即响应外部设备的输入或其他需要处理的事件。中断可以被看作是一种特殊的信号，它打断了正常的程序执行流程，使得处理器能够优先处理一些紧急的任务。当一个中断事件发生时，处理器会保存当前的执行状态，包括程序计数器和寄存器的值，并转而执行一个预先定义的中断处理程序（中断服务程序）。中断处理程序会根据不同的中断类型进行相应的处理，例如读取键盘输入、发送数据到打印机等。完成中断处理后，处理器会恢复之前保存的执行状态，继续执行被中断的任务。

5.2 硬件说明

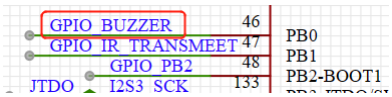

图 5.1: *蜂鸣器连接单片机引脚*

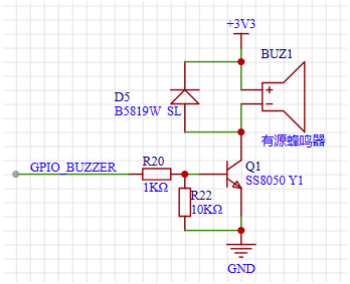

图 5.2: *蜂鸣器电路原理图*

蜂鸣器在开发板位置如下图所示：

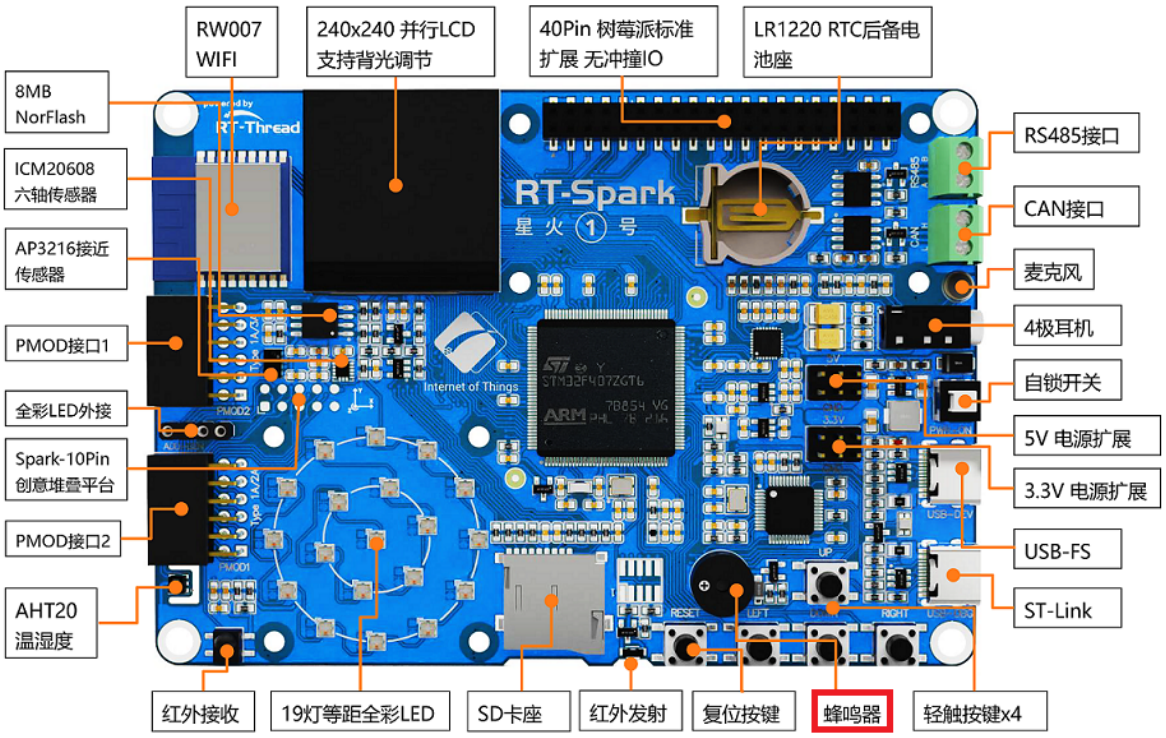

图 5.3: *蜂鸣器位置*

5.3 软件说明

按键控制蜂鸣器的源代码位于 ‘/projects/02_basic_irq_beep 下的 applications/main.c 中。

1. 查对应原理图可知，WK_UP、KEY1 按下为低电平，松开为高电平。所以在 main 函数中，WK_UP 和 KEY1 按键设置为上拉输入模式，PIN_BEEP 引脚设置为输出模式。
2. 然后使用 rt_pin_attach_irq 函数分别设置 KEY1 和 WK_UP 按键中断为下降沿触发中断并且绑定回调函数、设置回调函数相应的入参, 使用 rt_pin_irq_enable 函数使能这两个按键中断。

```
int main(void)
{
    /* 设置按键引脚为输入模式 */
    rt_pin_mode(PIN_KEY1, PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(PIN_WK_UP, PIN_MODE_INPUT_PULLUP);

    /* 设置蜂鸣器引脚为输出模式 */
    rt_pin_mode(PIN_BEEP, PIN_MODE_OUTPUT);

    /* 设置按键中断模式与中断回调函数 */
    rt_pin_attach_irq(PIN_KEY1, PIN_IRQ_MODE_FALLING, irq_callback, (void *)
        PIN_KEY1);
```

```
    rt_pin_attach_irq(PIN_WK_UP, PIN_IRQ_MODE_FALLING, irq_callback, (void *)
        PIN_WK_UP);

    /* 使能中断 */
    rt_pin_irq_enable(PIN_KEY1, PIN_IRQ_ENABLE);
    rt_pin_irq_enable(PIN_WK_UP, PIN_IRQ_ENABLE);

    while (1)
    {

    }
    return 0;
}
```

在中断回调函数中判断入参，根据不同入参进行相应的操作。

```
/* 中断回调 */
void irq_callback(void *args)
{
    rt_uint32_t sign = (rt_uint32_t) args;
    switch (sign)
    {
    case PIN_WK_UP :
        rt_pin_write(PIN_BEEP,PIN_HIGH);
        LOG_D("WK_UP interrupt. beep on.");
        break;
    case PIN_KEY1 :
        rt_pin_write(PIN_BEEP,PIN_LOW);
        LOG_D("KEY1 interrupt. beep off.");
        break;
    default:
        LOG_E("error sign= %d !", sign);
        break;
    }
}
```

5.4 运行

5.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 `STM32F407-RT-SPARK` 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

5.4.2 运行效果

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jul 13 2023 17:46:59
 2006 - 2022 Copyright by RT-Thread team
[D/main] WK_UP interrupt. beep on.
[D/main] KEY1 interrupt. beep off.
[D/main] WK_UP interrupt. beep on.
[D/main] KEY1 interrupt. beep off.
[D/main] WK_UP interrupt. beep on.
[D/main] WK_UP interrupt. beep on.
[D/main] KEY1 interrupt. beep off.
[D/main] WK_UP interrupt. beep on.
[D/main] KEY1 interrupt. beep off.
[D/main] WK_UP interrupt. beep on.
[D/main] KEY1 interrupt. beep off.
[D/main] WK_UP interrupt. beep on.
[D/main] WK_UP interrupt. beep on.
[D/main] WK_UP interrupt. beep on.
[D/main] WK_UP interrupt. beep on.
[D/main] WK_UP interrupt. beep on.
[D/main] WK_UP interrupt. beep on.
[D/main] KEY1 interrupt. beep off.
```

5.5 注意事项

暂无

5.6 引用参考

- 设备与驱动：PIN 设备

第 6 章

红外遥控例程

6.1 简介

本例程主要功能是通过板载的红外接收头接收红外遥控器信号以及通过板载的红外发射头发送红外信号。

6.2 硬件说明

图 6.1: *红外接收引脚*

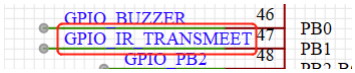

图 6.2: *红外发射连接单片机引脚*

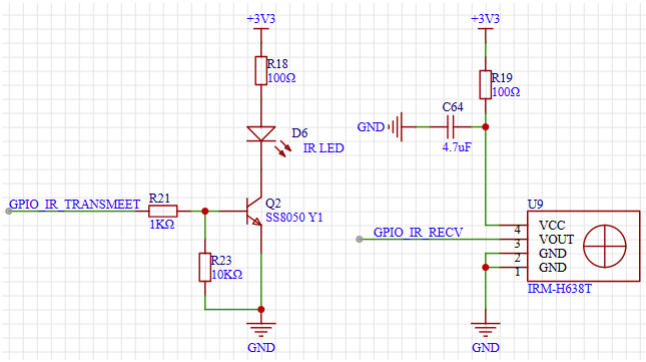

图 6.3: *红外发射接收电路原理图*

如上图所示，红外发射脚 GPIO_IR_TRANSMEET 接单片机引脚 PB1，红外接收头引脚 GPIO_IR_RECV 接单片机引脚 PF8（TIM3_CH4 通道）。红外传感器在开发板中的位置如下图所示：

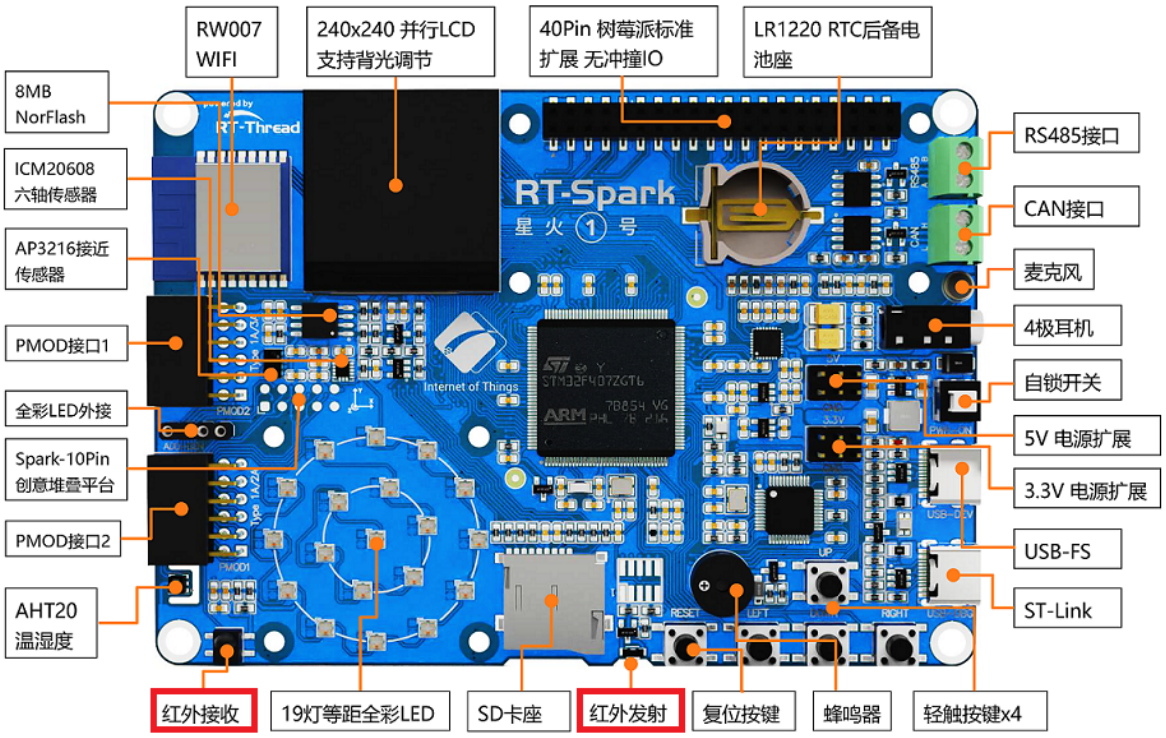

图 6.4: *红外位置*

6.3 软件说明

红外例程的示例代码位于 `/projects/02_basic_ir` 下的 applications/main.c 中，主要流程：选择 NEC 解码器，初始化 GPIO 引脚。然后在 while 循环中扫描按键、打印输出接收到的红外数据，当 KEY0 按下后将会把最近一次接收到的红外数据通过红外发射头发送出去。

```
int main(void)
{
    unsigned int count = 1;
    rt_int16_t key;
    struct infrared_decoder_data infrared_data;

    /* 选择 NEC 解码器 */
    ir_select_decoder("nec");

    /* 设置按键引脚为输入模式 */
    rt_pin_mode(PIN_KEY0, PIN_MODE_INPUT_PULLUP);

    /* 设置 RGB 引脚为输出模式 */
    rt_pin_mode(PIN_LED_R, PIN_MODE_OUTPUT);
    rt_pin_mode(PIN_LED_B, PIN_MODE_OUTPUT);

    rt_pin_write(PIN_LED_R, PIN_HIGH);
```

```
    rt_pin_write(PIN_LED_B, PIN_HIGH);

    while (count > 0)
    {
        /* 按键扫描 */
        key = key_scan();
        if(key == PIN_KEY0)
        {
            /* 有按键按下，蓝灯亮起 */
            rt_pin_write(PIN_LED_B, PIN_LOW);
            infrared_data.data.nec.repeat = 0;
            /* 发送红外数据 */
            infrared_write("nec",&infrared_data);
            rt_thread_mdelay(200);
            LOG_I("SEND    OK: addr:0x%02X key:0x%02X repeat:%d",
                infrared_data.data.nec.addr, infrared_data.data.nec.key,
                    infrared_data.data.nec.repeat);
        }
        else if(infrared_read("nec",&infrared_data) == RT_EOK)
        {
            /* 读取到红外数据，红灯亮起 */
            rt_pin_write(PIN_LED_R, PIN_LOW);
            LOG_I("RECEIVE OK: addr:0x%02X key:0x%02X repeat:%d",
                infrared_data.data.nec.addr, infrared_data.data.nec.key,
                    infrared_data.data.nec.repeat);
        }
        rt_thread_mdelay(10);

        /* 熄灭蓝灯 */
        rt_pin_write(PIN_LED_B, PIN_HIGH);
        /* 熄灭红灯 */
        rt_pin_write(PIN_LED_R, PIN_HIGH);
        count++;
    }
    return 0;
}
```

本例程的实现主要使用了红外软件包的以下三个函数：

```
/* 选 择 解 码 器 */
rt_err_t ir_select_decoder(const char* name);
/* 读 取 红 外 数 据。 */
rt_err_t infrared_read(const char* decoder_name,struct infrared_decoder_data* data);
/* 接 收 红 外 数 据。 */
rt_err_t infrared_write(const char* decoder_name, struct infrared_decoder_data* data
    );
```

6.4 运行

6.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

6.4.2 运行效果

按下复位按键重启开发板，观察板载 RGB 灯，用户可以使用红外遥控器对准板载红外接收头发送红外信号。在接收红外信号的时候，RGB 红灯闪烁。按下 KEY0 键，开发板将会通过红外发射头发送最近一次接收到的红外数据，发送红外数据时 RGB 蓝灯亮起。

此时也可以在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置波率：115200 ，数据位：8 ，停止位：1 N 。开发板的运行日志信息即可实时输出来。

```
[I/main] RECEIVE OK: addr:0x00 key:0x18 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x18 repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0x18 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x18 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x98 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x98 repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0x5A repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x5A repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0x7A repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0xB0 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0xA8 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0xA8 repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0xE0 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x68 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x30 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x02 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x02 repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0x02 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
[I/main] SEND    OK: addr:0x00 key:0x42 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
[I/main] SEND    OK: addr:0x00 key:0x42 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
[I/main] SEND    OK: addr:0x00 key:0x42 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
[I/main] SEND    OK: addr:0x00 key:0x42 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
```

```
[I/main] SEND    OK: addr:0x00 key:0x42 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
[I/main] SEND    OK: addr:0x00 key:0x42 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x42 repeat:0
```

6.5 注意事项

请使用 38KHZ 载波的红外遥控器实验。

6.6 引用参考

- 设备与驱动：PIN 设备
- 红外框架：https://github.com/RT-Thread-packages/infrared_framework

第 7 章

RTC 和 RTC 闹钟的使用例程

7.1 简介

本例程主要介绍了如何在星火 1 号上使用 RTC（Real-Time Clock）实时时钟，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。

7.2 硬件说明

本例程使用的 RTC 设备依赖于 LSE 时钟，此外无需过多连接。

7.3 软件说明

本例程的源码位于 `/projects/02_basic_rtc`。

在 main 函数中，获取到了 RTC 设备，然后设置一次系统时间，随后获取一次系统时间以便检测时间是否设置成功，最后延时 1s 后再次获取系统时间。

```
int main()
{
    rt_err_t ret = RT_EOK;
    time_t now;
    rt_device_t device = RT_NULL;

    /* 寻找设备 */
    device = rt_device_find(RTC_NAME);
    if (!device)
    {
        rt_kprintf("find %s failed!", RTC_NAME);
```

```
        return RT_ERROR;
    }

    /* 初始化 RTC 设备 */
    if(rt_device_open(device, 0) != RT_EOK)
    {
        rt_kprintf("open %s failed!", RTC_NAME);
        return RT_ERROR;
    }

    /* 设置日期 */
    ret = set_date(2018, 12, 3);
    if (ret != RT_EOK)
    {
        rt_kprintf("set RTC date failed\n");
        return ret;
    }

    /* 设置时间 */
    ret = set_time(11, 15, 50);
    if (ret != RT_EOK)
    {
        rt_kprintf("set RTC time failed\n");
        return ret;
    }

    /* 获取时间 */
    now = time(RT_NULL);
    rt_kprintf("%s\n", ctime(&now));

    /* 延时 1 秒 */
    rt_thread_mdelay(1000);

    /* 获取时间 */
    now = time(RT_NULL);
    rt_kprintf("%s\n", ctime(&now));

    return ret;
}
```

下面代码创建可一个 RTC 闹钟，然后设置 5 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)
{
    rt_kprintf("user alarm callback function.\n");
}

void alarm_sample(void)
{
```

```
    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 5 秒时间设置为闹钟时间 */
    now = time(NULL) + 5;
    gmtime_r(&now,&p_tm);

    setup.flag = RT_ALARM_ONESHOT;
    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);
```

7.4 运行

7.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

7.4.2 运行效果

按下复位按键重启开发板，可以看到板子上会打印如下信息：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jul 10 2023 09:37:14
```

```
 2006 - 2022 Copyright by RT-Thread team
Mon Dec  3 11:15:50 2018

msh >Mon Dec  3 11:15:51 2018
```

如上所示，两次系统时间相差 1s , 和设置的延时时间相对应。之后我们启动闹钟功能测试：

```
msh >alarm_sample
msh >user alarm callback function.
```

在启动 alarm_sample 函数后，5s 后，闹钟被叫醒并且进入提前设置好的回调函数。

7.5 注意事项

暂无。

7.6 引用参考

- 设备与驱动：RTC 设备

第 8 章

LCD 显示例程

8.1 简介

本例程主要介绍了如何在 LCD 上显示文字和图片。

8.2 硬件说明

星火 1 号开发板板载的是一块 1.3 寸，分辨率为 240x240 的 LCD 显示屏，显示效果十分细腻。显示屏的驱动芯片是 ST7789 v3, 通信接口使用的是 8080 并口，通过 fsmc 模拟出驱动时序和单片机进行通讯。使用了 8 根数据线传输数据，一根地址选择线作为芯片的使能信号。

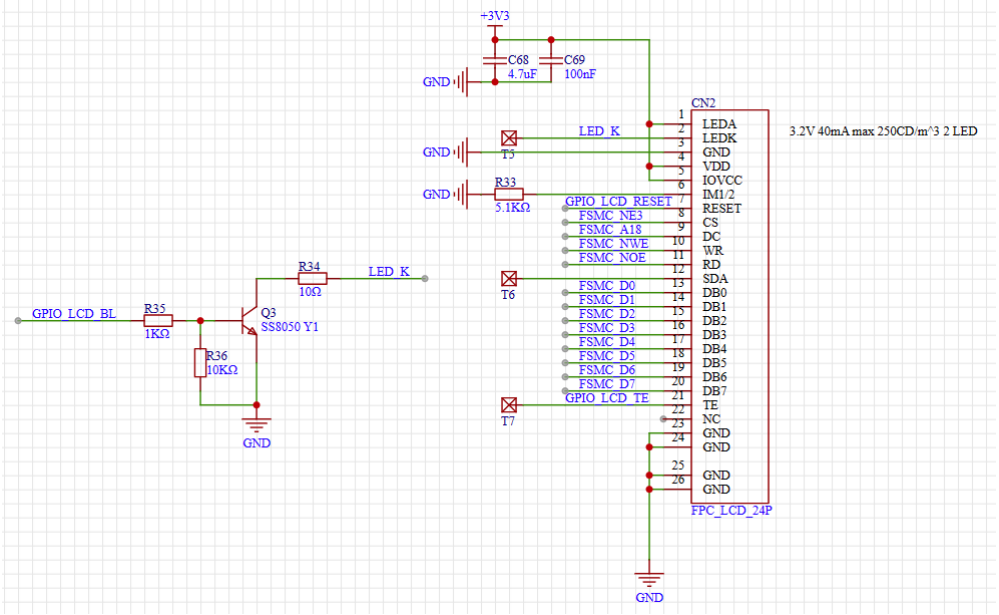

图 8.1: *LCD 原理图*

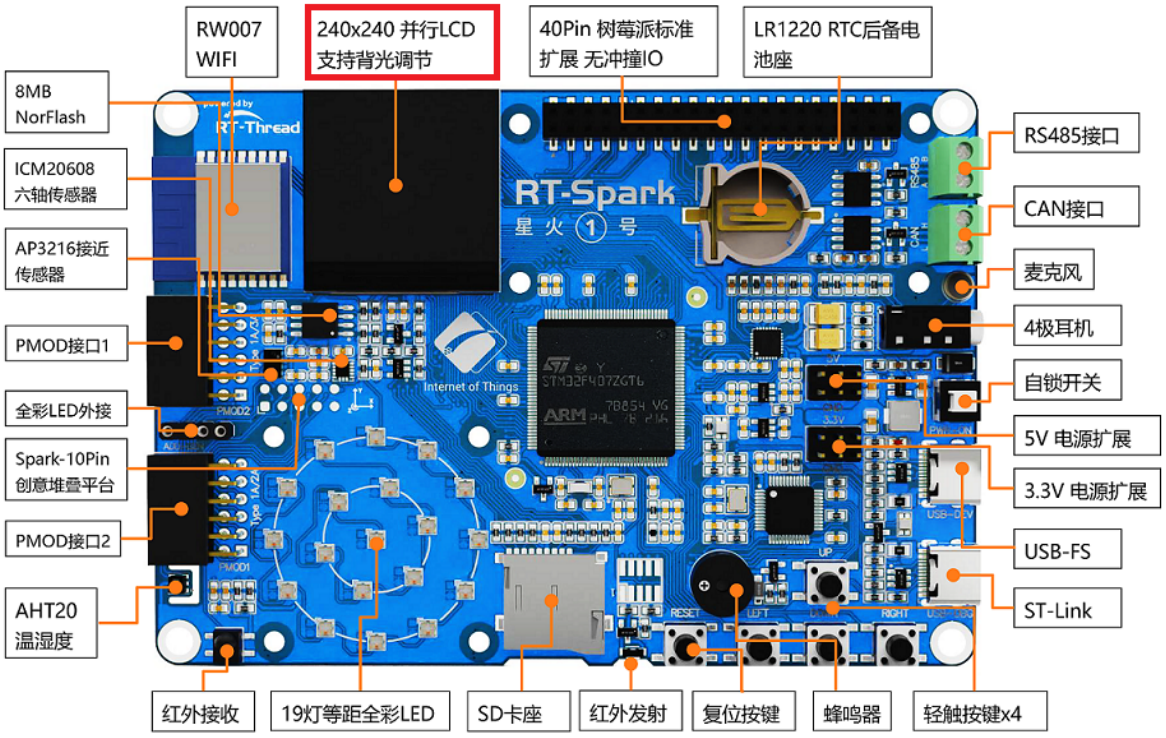

图 8.2: *LCD 位置图*

8.3 软件说明

本例程的源码位于 `/projects/03_driver_lcd`。

显示图片和文字的源代码位于 libraries/Board_Drivers/lcd/drv_lcd.c 中。

在 main 函数中，通过调用已经封装好的 LCD API 函数，首先执行的是清屏操作，将 LCD 全部刷成白色。然后设置画笔的颜色为黑色，背景色为白色。接着显示 RT-Thread 的 LOGO。最后会显示一些信息，包括 16x16 像素，24x24 像素和 32x32 像素的三行英文字符，一条横线和一个同心圆。

```
int main(void)
{
    lcd_clear(WHITE);

    /* show RT-Thread logo */
    lcd_show_image(0, 0, 240, 69, image_rttlogo);

    /* set the background color and foreground color */
    lcd_set_color(WHITE, BLACK);

    /* show some string on lcd */
    lcd_show_string(10, 69, 16, "Hello, RT-Thread!");
    lcd_show_string(10, 69 + 16, 24, "RT-Thread");
    lcd_show_string(10, 69 + 16 + 24, 32, "RT-Thread");
```

```
    /* draw a line on lcd */
    lcd_draw_line(0, 69 + 16 + 24 + 32, 240, 69 + 16 + 24 + 32);

    /* draw a concentric circles */
    lcd_draw_point(120, 194);
    for (int i = 0; i < 46; i += 4)
    {
        lcd_draw_circle(120, 194, i);
    }
    return 0;
}
```

8.4 运行

8.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

8.4.2 运行效果

按下复位按键重启开发板，观察开发板上 LCD 的实际效果。正常运行后，LCD 上会显示 RT-Thread LOGO，下面会显示 3 行大小为 16、24、32 像素的文字，文字下面是一行直线，直线的下方是一个同心圆。如下图所示：

图 8.3: *LCD 显示图案*

8.5 注意事项

屏幕的分辨率是 240x240，输入位置参数时要注意小于 240，不然会出现无法显示的现象。图像的取模方式为自上而下，自左向右，高位在前，16 位色（RGB-565）。本例程未添加中文字库，不支持显示中文。

8.6 引用参考

暂无。

第 9 章

AHT10 温湿度传感器例程

9.1 简介

本例程主要功能是利用 RT-Thread 的 AHT10 软件包的读取传感器 aht10(aht21) 所测量的温度（temperature）与湿度（humidity）。因为两款传感器的驱动互相兼容，所以下文所说的 aht10 指的是我们板载的 aht21。

9.2 AHT10 软件包简介

AHT10 软件包提供了使用温度与湿度传感器 aht10 基本功能，并且提供了软件平均数滤波器可选功能，如需详细了解该软件包，请参考 AHT10 软件包中的 README。

9.3 硬件说明

aht10 硬件原理图如下所示：

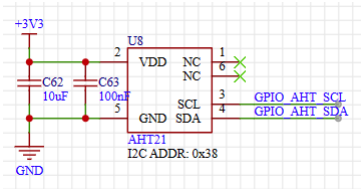

图 9.1: *温湿度传感器模块电路*

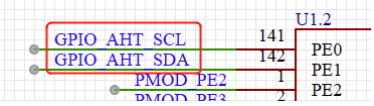

图 9.2: *温湿度传感器与芯片连接图*

如上图所示，单片机通过软件 iic I2C3(soft) scl(PE0)、I2C3(soft) sda(PE1) 对传感器 aht10 发送命令、读取数据等。温度与湿度传感器在开发板中的位置如下图所示：

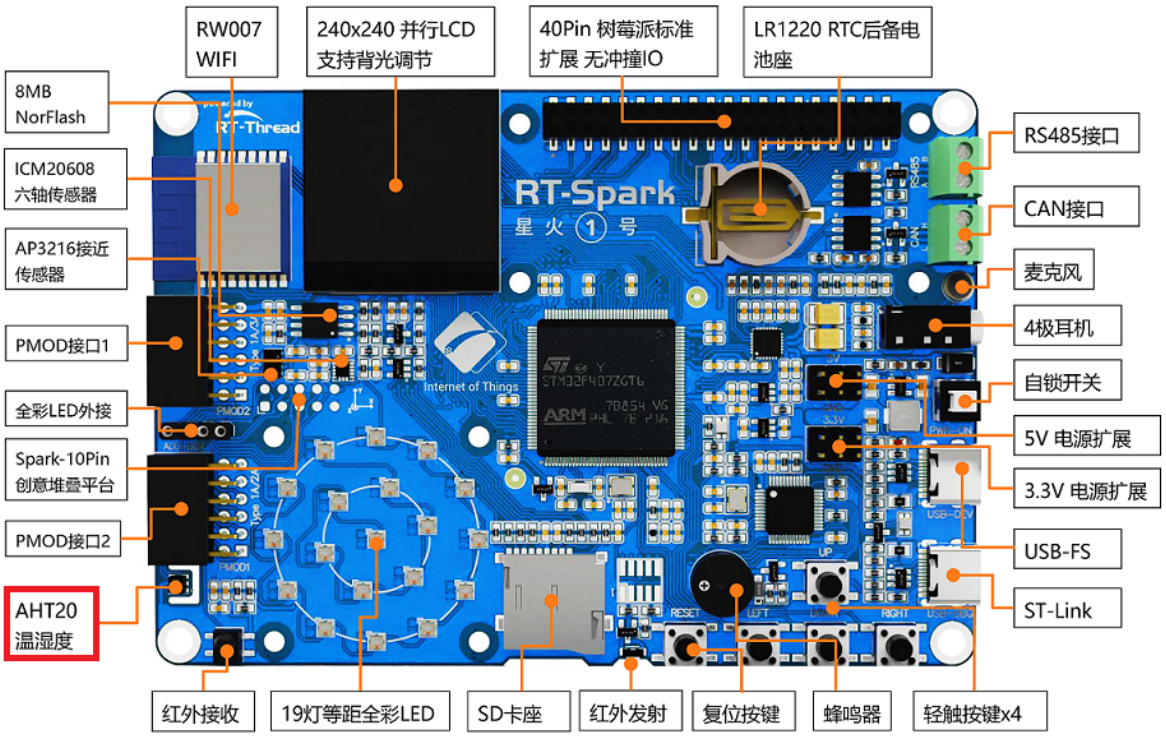

图 9.3: *温湿度传感器位置*

该传感器输入电压范围为 1.8v - 3.3v，测量温度与湿度的量程、精度如下表所示：

功能	量程	精度	单位
温度	-40 - 85	±0.5	摄氏度
相对湿度	0 - 100	±3	%

9.4 软件说明

本例程的源码位于 `/projects/03_driver_temp_humi`。

温度与湿度传感器的示例代码位于 applications/main.c 中，主要流程：初始化传感器 aht10，传入参数 i2c2 为该传感器挂载的 i2c 总线的名称；初始化若失败，则返回空，程序不会被执行，若成功，则返回传感器设备对象；然后将返回的设备对象分别传入读取湿度与温度的函数，获取测量的湿度与温度值（详细的 API 介绍参考 aht10 软件包读取温度与湿度章节，源码参考 aht10.c）。示例代码如下：

```
int main(void)
{
    float humidity, temperature;
    aht10_device_t dev;

    /* 总线名称 */
```

```
    const char *i2c_bus_name = "i2c3";
    int count = 0;

    /* 等待传感器正常工作 */
    rt_thread_mdelay(2000);

    /* 初始化 aht10 */
    dev = aht10_init(i2c_bus_name);
    if (dev == RT_NULL)
    {
        LOG_E("The sensor initializes failure");
        return 0;
    }

    while (count++ < 100)
    {
        /* 读取湿度 */
        humidity = aht10_read_humidity(dev);
        LOG_D("humidity   : %d.%d %%", (int)humidity, (int)(humidity * 10) % 10);

        /* 读取温度 */
        temperature = aht10_read_temperature(dev);
        LOG_D("temperature: %d.%d", (int)temperature, (int)(temperature * 10) % 10);

        rt_thread_mdelay(1000);
    }
    return 0;
}
```

9.5 运行

9.5.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

9.5.2 运行效果

烧录完成后，此时可以在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置串口波特率为 115200，数据位 8 位，停止位 1 位，无流控，开发板的运行日志信息即可实时输出出来，显示如下所示：

```
 \ | /
```

```
 - RT -     Thread Operating System
  / | \     4.1.1 build Jun  9 2023 13:18:28
  2006 - 2022 Copyright by RT-Thread team
msh >[D/main] humidity   : 0.0 %
[D/main] temperature: -50.0
[D/main] humidity   : 58.8 %
[D/main] temperature: 26.2
[D/main] humidity   : 58.7 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.6 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.5 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.5 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.5 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.4 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.3 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.4 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.2 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.1 %
[D/main] temperature: 26.3
[D/main] humidity   : 58.1 %
[D/main] temperature: 26.3
[D/main] humidity   : 57.9 %
[D/main] temperature: 26.4
[D/main] humidity   : 57.7 %
[D/main] temperature: 26.4
[D/main] humidity   : 57.4 %
[D/main] temperature: 26.3
[D/main] humidity   : 57.3 %
[D/main] temperature: 26.4
[D/main] humidity   : 57.1 %
[D/main] temperature: 26.3
[D/main] humidity   : 57.0 %
[D/main] temperature: 26.4
[D/main] humidity   : 57.0 %
[D/main] temperature: 26.4
```

9.6 注意事项

暂无。

9.7 引用参考

- 设备与驱动：I2C 设备
- aht10 软件包：https://github.com/RT-Thread-packages/aht10

第 10 章

AP3216C 接近与光强传感器例程

10.1 简介

本例程主要功能是利用 RT-Thread 的 AP3216C 软件包读取传感器 ap3216c 测量的接近感应（ps，proximity sensor）与光照强度（als，ambient light sensor）。

10.2 AP3216C 软件包简介

AP3216C 软件包提供了使用接近感应（ps）与光照强度（als）传感器 ap3216c 基本功能，并且提供了硬件中断的可选功能，如需详细了解该软件包，请参考 AP3216C 软件包中的 README。

10.3 硬件说明

ap3216c 硬件原理图如下所示：

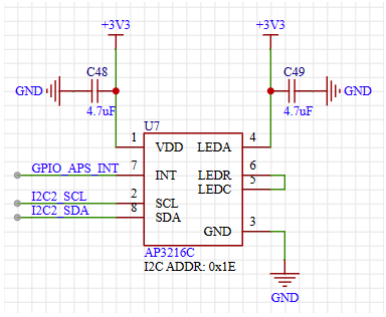

图 10.1: *接近与光强传感器模块原理图*

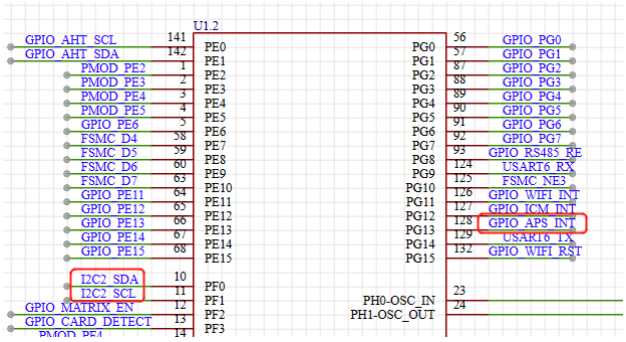

图 **10.2:** *接近与光强传感器模块与芯片连接图*

如上图所示，单片机通过 I2C2(soft) scl(PF1)、I2C2(soft) sda(PF0) 对传感器 ap3216c 发送命令、读取数据等，AP_INT(PG13) 为硬件中断引脚。

接近感应与光照强度传感器在开发板中的位置如下图所示：

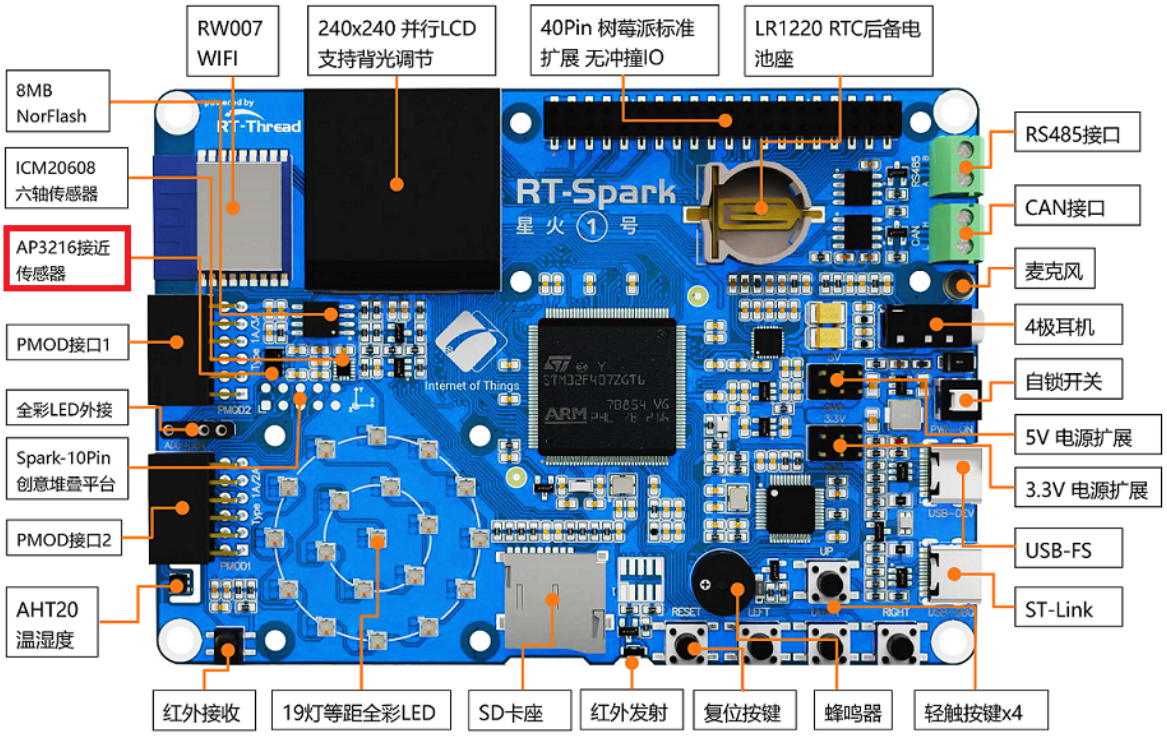

图 **10.3:** *接近与光强传感器位置*

该传感器能够实现如下功能：

- 光照强度：支持 4 个量程
- 接近感应：支持 4 种增益
- 中断触发：光照强度及接近感应同时支持高于阈值或低于阈值的两种硬件中断触发方式

10.4 软件说明

本例程的源码位于 /projects/03_driver_als_ps。

接近感应与光照强度传感器 ap3216c 的示例代码位于 applications/main.c 中，主要流程：初始化传感器 ap3216c，传入参数 i2c1 为该传感器挂载的 i2c 总线的名称；初始化若失败，则返回空，程序不会被执行，若成功，则返回传感器设备对象；然后将返回设备对象分别传入获取 als 与 ps 函数，获取测量的 als 与 ps 值（详细的 API 介绍参考 ap3216c 软件包读取接近感应与光照强度章节，源码参考 ap3216c.c）。示例代码如下：

```
int main(void)
{
    ap3216c_device_t dev;
    const char *i2c_bus_name = "i2c2";
    int count = 0;

    /* 初始化 ap3216c */
    dev = ap3216c_init(i2c_bus_name);
    if (dev == RT_NULL)
    {
        LOG_E("The sensor initializes failure.");
        return 0;
    }

    while (count++ < 100)
    {
        rt_uint16_t ps_data;
        float brightness;

        /* 读接近感应值 */
        ps_data = ap3216c_read_ps_data(dev);
        if (ps_data == 0)
        {
            LOG_D("object is not proximity of sensor.");
        }
        else
        {
            LOG_D("current ps data   : %d.", ps_data);
        }

        /* 读光照强度值 */
        brightness = ap3216c_read_ambient_light(dev);
        LOG_D("current brightness: %d.%d(lux).", (int)brightness, ((int)(10 *
            brightness) % 10));

        rt_thread_mdelay(1000);
    }
    return 0;
}
```

10.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

10.4.2 运行效果

烧录完成后，此时可以在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置串口波特率为 115200，数据位 8 位，停止位 1 位，无流控，开发板的运行日志信息即可实时输出出来，显示如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun  9 2023 13:18:36
 2006 - 2022 Copyright by RT-Thread team
msh >[D/main] object is not proximity of sensor.
[D/main] current brightness: 32.9(lux).
[D/main] object is not proximity of sensor.
[D/main] current brightness: 33.6(lux).
[D/main] current ps data   : 7.
[D/main] current brightness: 33.9(lux).
[D/main] object is not proximity of sensor.
[D/main] current brightness: 33.9(lux).
[D/main] current ps data   : 7.
[D/main] current brightness: 33.9(lux).
[D/main] current ps data   : 11.
[D/main] current brightness: 33.9(lux).
[D/main] current ps data   : 1.
[D/main] current brightness: 33.9(lux).
[D/main] object is not proximity of sensor.
[D/main] current brightness: 33.9(lux).
[D/main] object is not proximity of sensor.
[D/main] current brightness: 33.9(lux).
[D/main] current ps data   : 1023.
[D/main] current brightness: 0.0(lux).
[D/main] current ps data   : 1023.
[D/main] current brightness: 0.0(lux).
[D/main] object is not proximity of sensor.
[D/main] current brightness: 32.9(lux).
[D/main] object is not proximity of sensor.
[D/main] current brightness: 33.6(lux).
[D/main] current ps data   : 1.
[D/main] current brightness: 33.6(lux).
[D/main] current ps data   : 8.
[D/main] current brightness: 33.6(lux).
```

```
[D/main] object is not proximity of sensor.
[D/main] current brightness: 33.6(lux).
```

10.5 注意事项

暂无。

10.6 引用参考

- 设备与驱动：I2C 设备
- ap3216c 软件包：https://github.com/RT-Thread-packages/ap3216c

第 11 章

ICM20608 六轴传感器例程

11.1 简介

本例程主要功能是利用 RT-Thread 的 ICM20608 软件包读取传感器 icm20608 所测量的三轴加速度（three accelerate）、三轴陀螺仪（three gyroscope）。

11.2 ICM20608 软件包简介

ICM20608 软件包是 RT-Thread 针对六轴传感器 icm20608 功能使用的实现，使用这个软件包，可以让该传感器在 RT-Thread 上非常方便使用 icm20608 的基本功能，包括读取三轴加速度（3-axisaccelerometer）、三轴陀螺仪（3-axis gyroscope）、零值校准等功能，如需详细了解该软件包，请参考 ICM20608 软件包中的 README。

11.3 硬件说明

icm20608 硬件原理图如下所示：

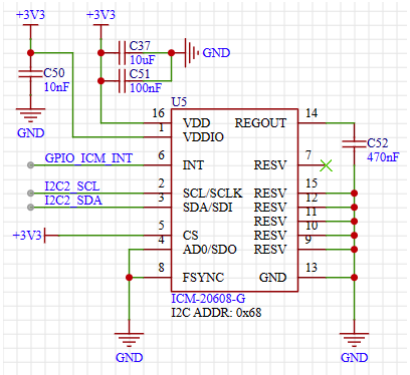

图 11.1: *六轴传感器原理图*

图 11.2: *六轴传感器与芯片连接图*

如上图所示，单片机通过 I2C2(soft) scl(PF1)、I2C2(soft) sda(PF0) 对传感器 icm20608 发送命令、读取数据等，ICM_INT(PG12) 为硬件中断引脚。

六轴传感器在开发板中的位置如下图所示：

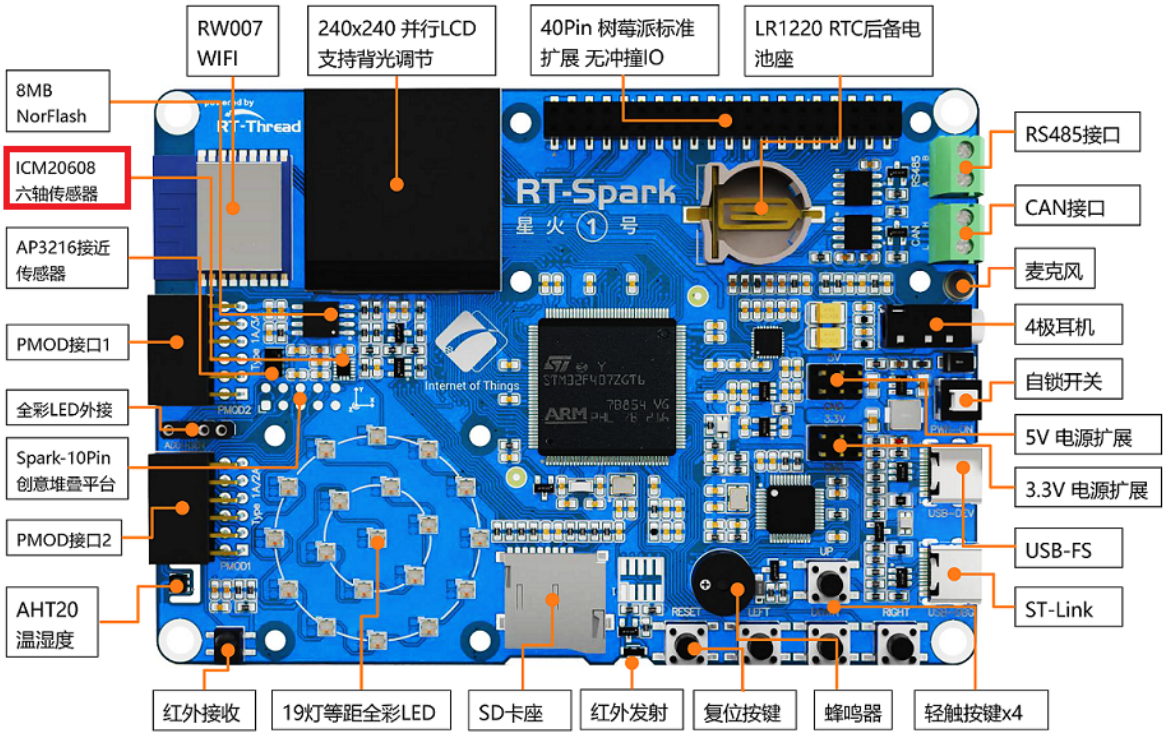

图 11.3: *六轴传感器位置*

该传感器能够实现如下功能：

- 支持 4 种三轴加速度量程
- 支持 4 种三轴陀螺仪量程
- 支持零值校准

11.4 软件说明

本例程的源码位于 `/projects/03_driver_axis`。

六轴传感器 icm20608 的示例代码位于 applications/main.c 中，主要流程：初始化传感器 -> 零值校准 -> 读取三轴加速度与三轴陀螺仪，分别展开如下所述：

1. 初始化传感器

初始化函数 icm20608_init 传入的参数 i2c_bus_name 为该传感器挂载的 i2c 总线名称，进行初始化；初始化若失败，则返回空，若成功，则返回六轴传感器的设备对象 dev。

2. 零值校准

首先在进行零值校准时，x 轴、y 轴应处于水平状态，且传感器处于静态；其次使用零值校准函数 icm20608_calib_level 进行零值校准时，传入设备对象 dev 与读取零值次数（此处为 10 次，可以改动），若失败，释放资源，提示失败，释放资源，若成功，返回 RT_EOK，零值数据存放在设备对象 dev 中，详细零值存放参考 icm20608 软件包中零值校准章节。

3. 读取三轴加速度与三轴陀螺仪

成功校准后，进行数据读取。如果失败，提示传感器不正常工作；如果成功，打印读取的三轴加速度与三轴陀螺仪的测量值（详细的 API 介绍参考 icm20608 软件包读取三轴加速度与三轴陀螺仪章节，源码参考 icm20608.c）。

示例代码如下：

```
int main(void)
{
    icm20608_device_t dev = RT_NULL;
    const char *i2c_bus_name = "i2c2";
    int count = 0;
    rt_err_t result;

    /* 初始化 icm20608 传感器 */
    dev = icm20608_init(i2c_bus_name);
    if (dev == RT_NULL)
    {
        LOG_E("The sensor initializes failure");

        return 0;
    }
    else
    {
        LOG_D("The sensor initializes success");
    }

    /* 对 icm20608 进行零值校准：采样 10 次，求取平均值作为零值 */
    result = icm20608_calib_level(dev, 10);
    if (result == RT_EOK)
    {
        LOG_D("The sensor calibrates success");
        LOG_D("accel_offset: X%6d  Y%6d  Z%6d", dev->accel_offset.x, dev->
            accel_offset.y, dev->accel_offset.z);
        LOG_D("gyro_offset : X%6d  Y%6d  Z%6d", dev->gyro_offset.x, dev->gyro_offset
            .y, dev->gyro_offset.z);
```

```
    }
    else
    {
        LOG_E("The sensor calibrates failure");
        icm20608_deinit(dev);

        return 0;
    }

    while (count++ < 100)
    {
        rt_int16_t accel_x, accel_y, accel_z;
        rt_int16_t gyros_x, gyros_y, gyros_z;

        /* 读取三轴加速度 */
        result = icm20608_get_accel(dev, &accel_x, &accel_y, &accel_z);
        if (result == RT_EOK)
        {
            LOG_D("current accelerometer: accel_x%6d, accel_y%6d, accel_z%6d",
                accel_x, accel_y, accel_z);
        }
        else
        {
            LOG_E("The sensor does not work");
            break;
        }

        /* 读取三轴陀螺仪 */
        result = icm20608_get_gyro(dev, &gyros_x, &gyros_y, &gyros_z);
        if (result == RT_EOK)
        {
            LOG_D("current gyroscope    : gyros_x%6d, gyros_y%6d, gyros_z%6d",
                gyros_x, gyros_y, gyros_z);
        }
        else
        {
            LOG_E("The sensor does not work");
            break;
        }
        rt_thread_mdelay(1000);
    }

    return 0;
}
```

11.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

11.4.2 运行效果

烧录完成后，此时可以在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置串口波特率为 115200，数据位 8 位，停止位 1 位，无流控，开发板的运行日志信息即可实时输出出来，显示如下所示：

```
[D/main] The sensor initializes success
[D/main] The sensor calibrates success
[D/main] accel_offset: X   718  Y  1524  Z  1199
[D/main] gyro_offset : X   -89  Y    71  Z   -54
[D/main] current accelerometer: accel_x   122, accel_y   -68, accel_z 16441
[D/main] current gyroscope    : gyros_x    45, gyros_y    -9, gyros_z   -19
[D/main] current accelerometer: accel_x    94, accel_y   -36, accel_z 16561
[D/main] current gyroscope    : gyros_x    10, gyros_y   -15, gyros_z    -7
[D/main] current accelerometer: accel_x   -18, accel_y   -40, accel_z 16421
[D/main] current gyroscope    : gyros_x     5, gyros_y    -3, gyros_z    13
[D/main] current accelerometer: accel_x    42, accel_y     4, accel_z 16549
[D/main] current gyroscope    : gyros_x    10, gyros_y    -6, gyros_z    -5
[D/main] current accelerometer: accel_x    14, accel_y    32, accel_z 16489
[D/main] current gyroscope    : gyros_x    13, gyros_y    -4, gyros_z    -5
[D/main] current accelerometer: accel_x    38, accel_y    40, accel_z 16381
[D/main] current gyroscope    : gyros_x    12, gyros_y    11, gyros_z    13
[D/main] current accelerometer: accel_x    70, accel_y   -20, accel_z 16369
[D/main] current gyroscope    : gyros_x    -5, gyros_y     9, gyros_z     0
[D/main] current accelerometer: accel_x    30, accel_y   -52, accel_z 16457
[D/main] current gyroscope    : gyros_x    11, gyros_y     5, gyros_z    -3
[D/main] current accelerometer: accel_x   -94, accel_y   -24, accel_z 16473
[D/main] current gyroscope    : gyros_x    42, gyros_y     0, gyros_z    13
[D/main] current accelerometer: accel_x   -38, accel_y   -64, accel_z 16421
[D/main] current gyroscope    : gyros_x    27, gyros_y    -7, gyros_z    -3
[D/main] current accelerometer: accel_x    78, accel_y    52, accel_z 16513
[D/main] current gyroscope    : gyros_x    11, gyros_y    16, gyros_z   -23
[D/main] current accelerometer: accel_x    26, accel_y   -16, accel_z 16409
[D/main] current gyroscope    : gyros_x     5, gyros_y    -8, gyros_z    -2
```

11.5 注意事项

暂无。

11.6 引用参考

- 设备与驱动：PIN 设备
- icm20608 软件包：https://github.com/RT-Thread-packages/icm20608

第 12 章

CAN 通信例程

12.1 简介

本例程主要完成的工作是完成 CAN 总线的接收和发送功能。

CAN 是控制器局域网络 (Controller Area Network, CAN) 的简称，是由以研发和生产汽车电子产品著称的德国 BOSCH 公司开发的，并最终成为国际标准（ISO 11898），是国际上应用最广泛的现场总线之一。

12.2 硬件说明

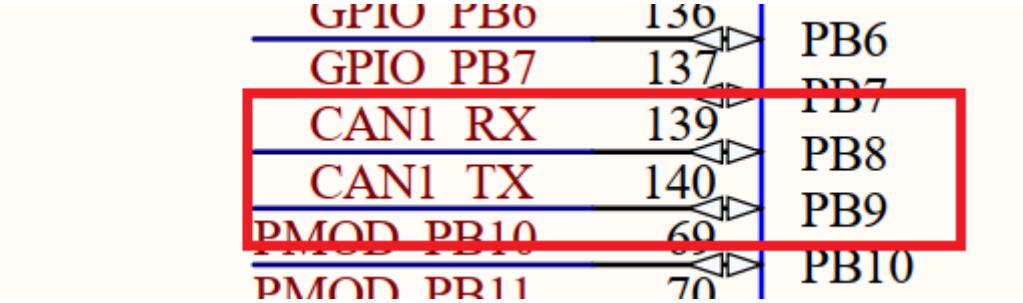

图 12.1: *CAN 连接单片机引脚*

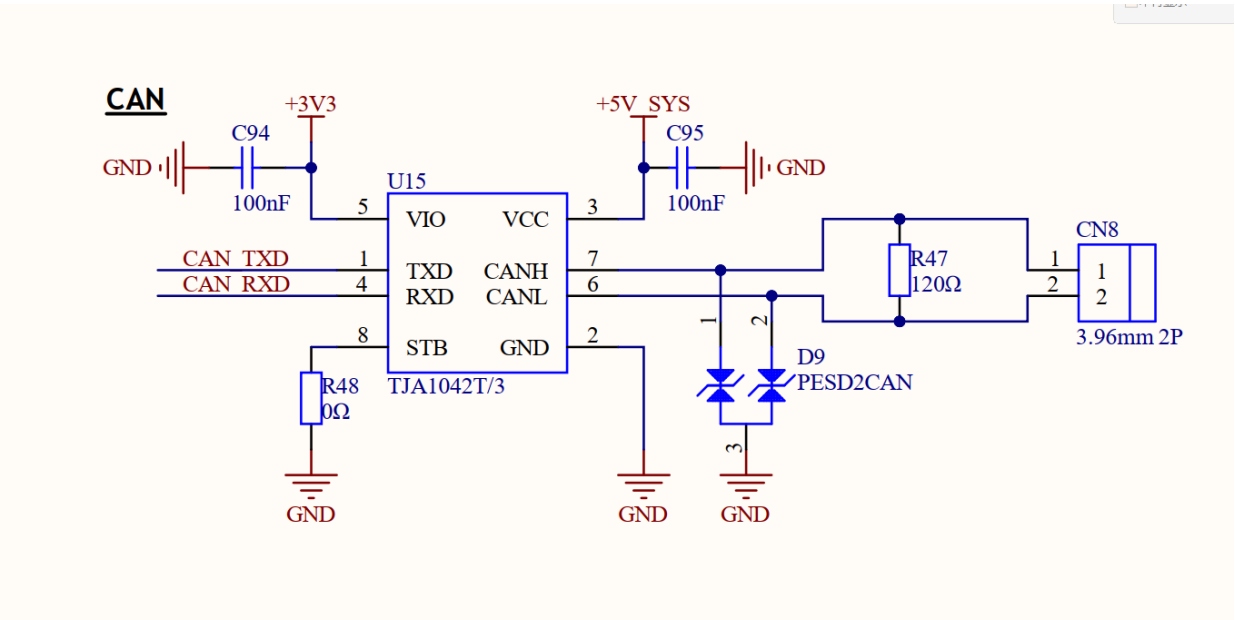

图 12.2: *CAN 电路原理图*

如上图所示，CAN_RX 连接单片机的 PB8 引脚，CAN_TX 连接单片机的 PB9 引脚。

此例程需要两张星火 1 号开发板来完成 CAN 的接收和发送功能，连接示意图如下：

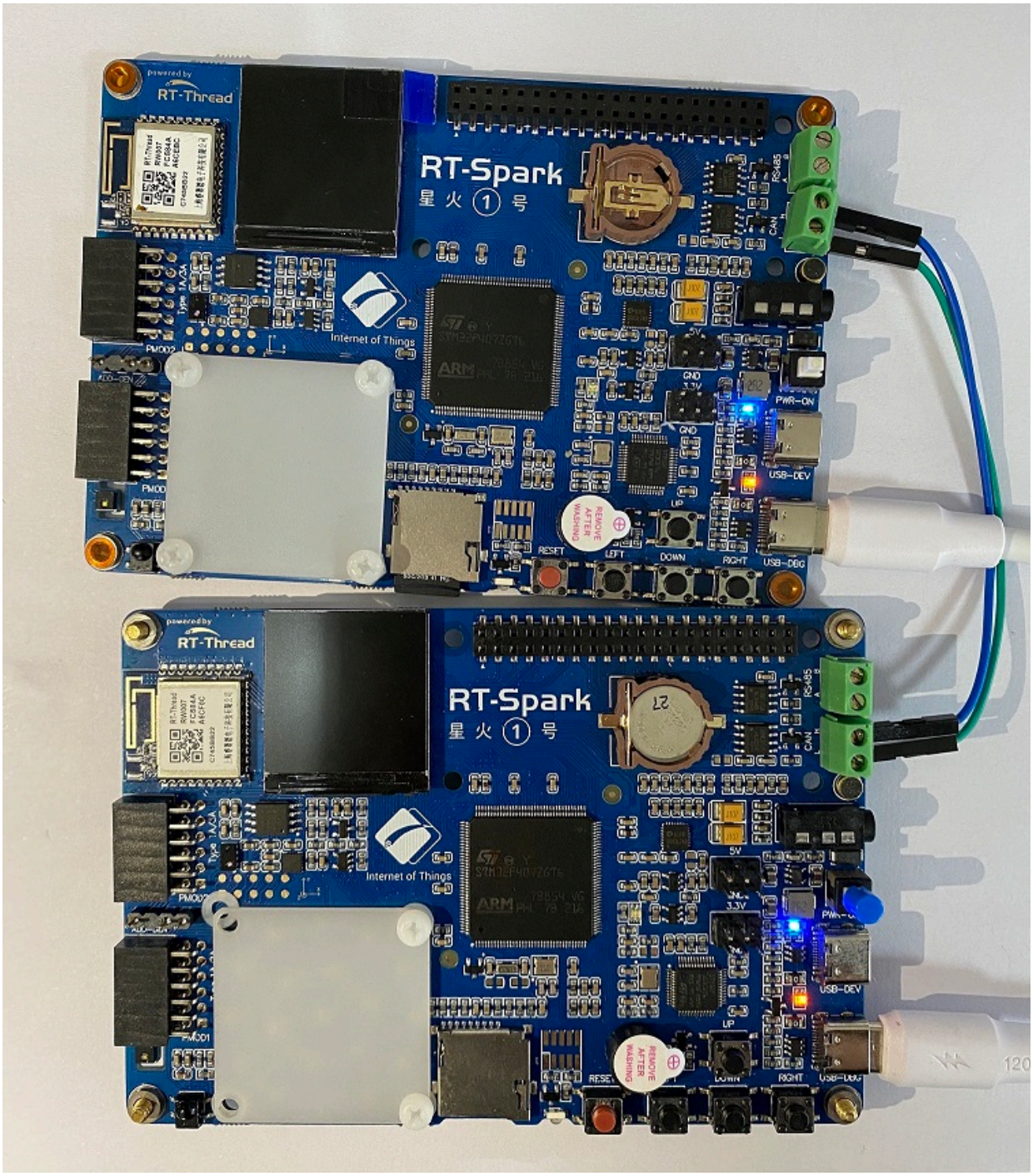

图 12.3: *CAN 连接*

12.3 软件说明

本例程的源码位于 `/projects/03_driver_can`。

CAN 通信的源代码位于 applications/main.c 中。

```
int can_sample_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, CAN_DEV_NAME, RT_NAME_MAX);
    }
    /* 查找 CAN 设备 */
    can_dev = rt_device_find(can_name);
    if (!can_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(can_dev, RT_DEVICE_FLAG_INT_TX | RT_DEVICE_FLAG_INT_RX);
    RT_ASSERT(res == RT_EOK);
    /* 创建数据接收线程 */
    thread = rt_thread_create("can_tx", can_tx_thread, RT_NULL, 1024, 25, 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_sample_send, can device sample);
```

上述代码打开了一个 CAN 设备 can1, 随后创建了一个 CAN 发送线程。最后将测试函数导入到 MSH 命令列表中。

```
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));
    int number = 0;
    while(1)
    {
        /* 发送一帧 CAN 数据 */
        rt_sprintf((char *)msg.data,"rtt:%d",number++);
        size = rt_device_write(can_dev, 0, &msg, sizeof(msg));
        if (size == 0)
        {
            rt_kprintf("can dev write data failed!\n");
        }
        rt_thread_delay(1000);
    }
}
```

在发送线程中，首先设置 CAN 发送参数，id=0x78 使用标准帧，帧类型为数据帧，数据长度为 8。随后每隔 1s 调用一次发送函数，将 number 通过 CAN 总线发送出去。

```
int can_sample_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, CAN_DEV_NAME, RT_NAME_MAX);
    }
    /* 查找 CAN 设备 */
    can_dev = rt_device_find(can_name);
    if (!can_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(can_dev, RT_DEVICE_FLAG_INT_TX | RT_DEVICE_FLAG_INT_RX);
    RT_ASSERT(res == RT_EOK);
    /* 创建数据接收线程 */
    thread = rt_thread_create("can_rx", can_rx_thread, RT_NULL, 1024, 25, 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_sample_receive, can device sample);
```

上述函数为 CAN 接收的实现，首先初始化接收信号量，随后打开了 CAN 设备 can1，然后创建了一个接收线程。

```
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(can_dev, can_rx_call);

    while (1)
    {
        /* hdr 值为 - 1，表示直接从 uselist 链表读取数据 */
        rxmsg.hdr = -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(can_dev, 0, &rxmsg, sizeof(rxmsg));
        /* 打印数据 ID 及内容 */
        rt_kprintf("ID:%x\tmessege:%s\n", rxmsg.id, rxmsg.data);
    }
}
```

在接收线程中首先绑定接收回调函数，在回调函数中释放信号量。随后接收线程阻塞，直到获得到了信号量。最后获得信号量之后将读取到的信息打印出来。

12.4 运行

12.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

12.4.2 运行效果

按下复位按键重启开发板，分别在两个板子上启动发送例程和接收例程。

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jul 10 2023 15:13:34
 2006 - 2022 Copyright by RT-Thread team
msh >can_sample_send
msh >
```

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jul 10 2023 15:13:34
 2006 - 2022 Copyright by RT-Thread team
msh >can_sample_receive
msh >ID:78      messege:rtt:0
ID:78   messege:rtt:1
ID:78   messege:rtt:2
ID:78   messege:rtt:3
ID:78   messege:rtt:4
ID:78   messege:rtt:5
ID:78   messege:rtt:6
ID:78   messege:rtt:7
ID:78   messege:rtt:8
ID:78   messege:rtt:9
ID:78   messege:rtt:10
ID:78   messege:rtt:11
ID:78   messege:rtt:12
ID:78   messege:rtt:13
ID:78   messege:rtt:14
ID:78   messege:rtt:15
```

如上所示，在启动发送例程后，发送例程每隔 1s 发送一次数据。而接收例程，收到了 ID 为 78 的设备发送过来的消息并且打印了出来。## 注意事项

在连接 CAN 总线的时候，请将 CAN L 连接到 CAN L,CAN H 连接到 CAN H。

12.5 引用参考

- 设备与驱动：CAN 设备

第 13 章

LED MATRIX 闪烁例程

13.1 简介

本例程主要介绍了如何驱动板载 LED MATRIX

13.2 硬件说明

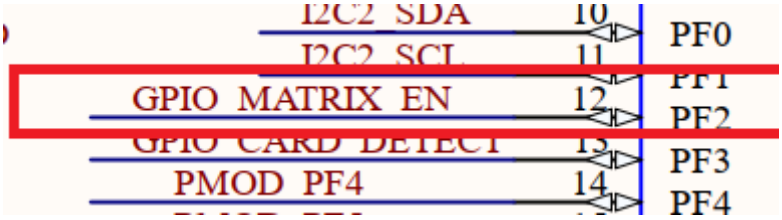

图 13.1: *LED MATRIX 连接单片机使能引脚*

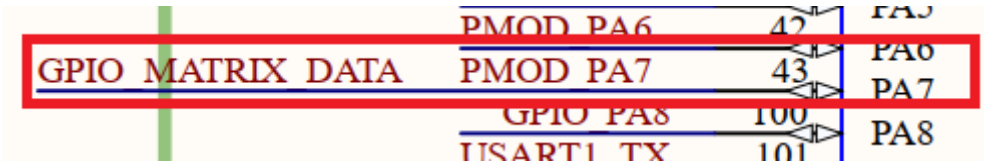

图 13.2: *LED MATRIX 连接单片机数据引脚*

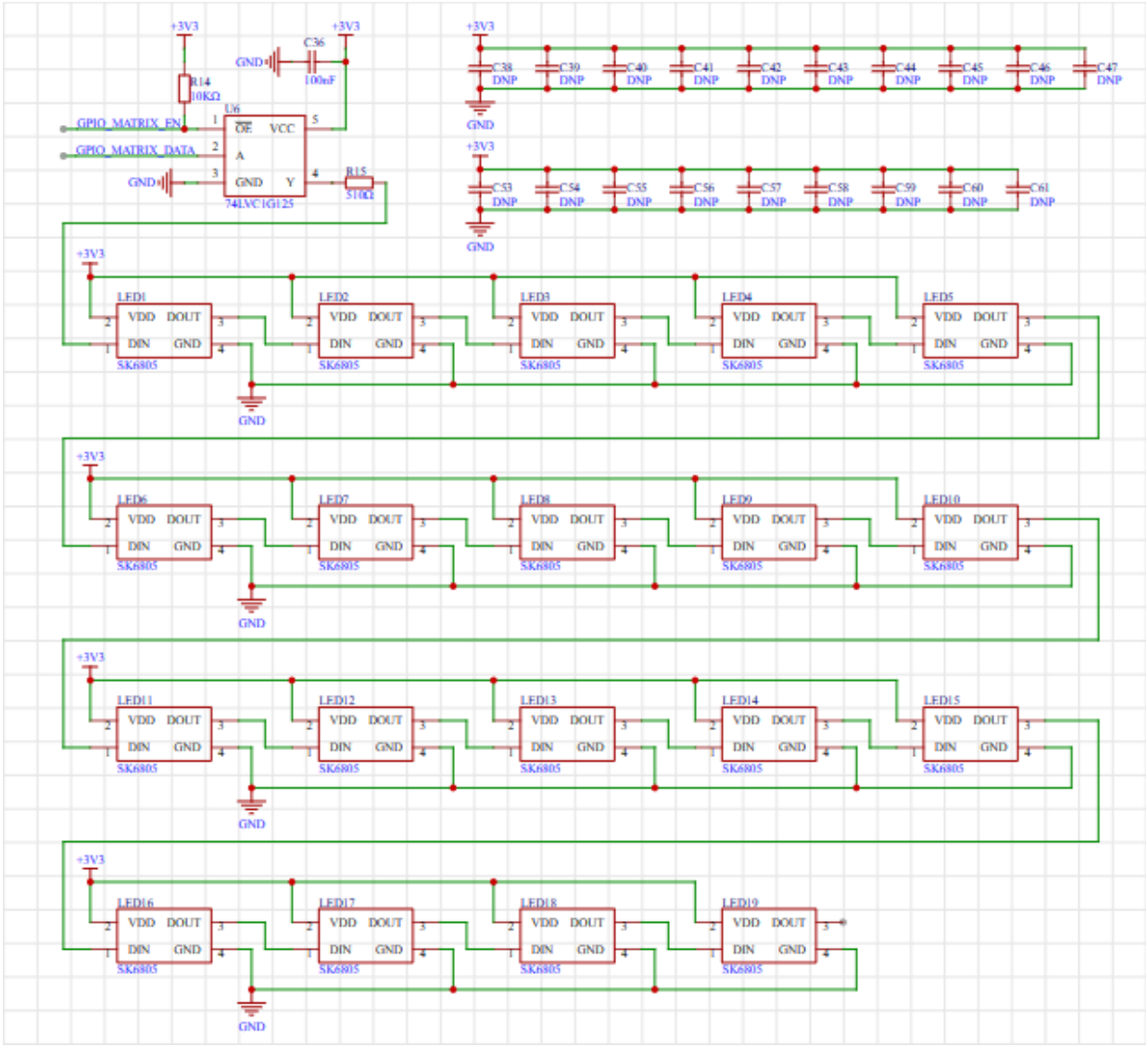

图 13.3: *LED MATRIX 电路原理图*

如上图所示，LED MATRIX 使用的是 SK6805 RGB 三色 LED, 驱动时使用串行信号。

LED MATRIX 在开发板中的位置如下图所示：

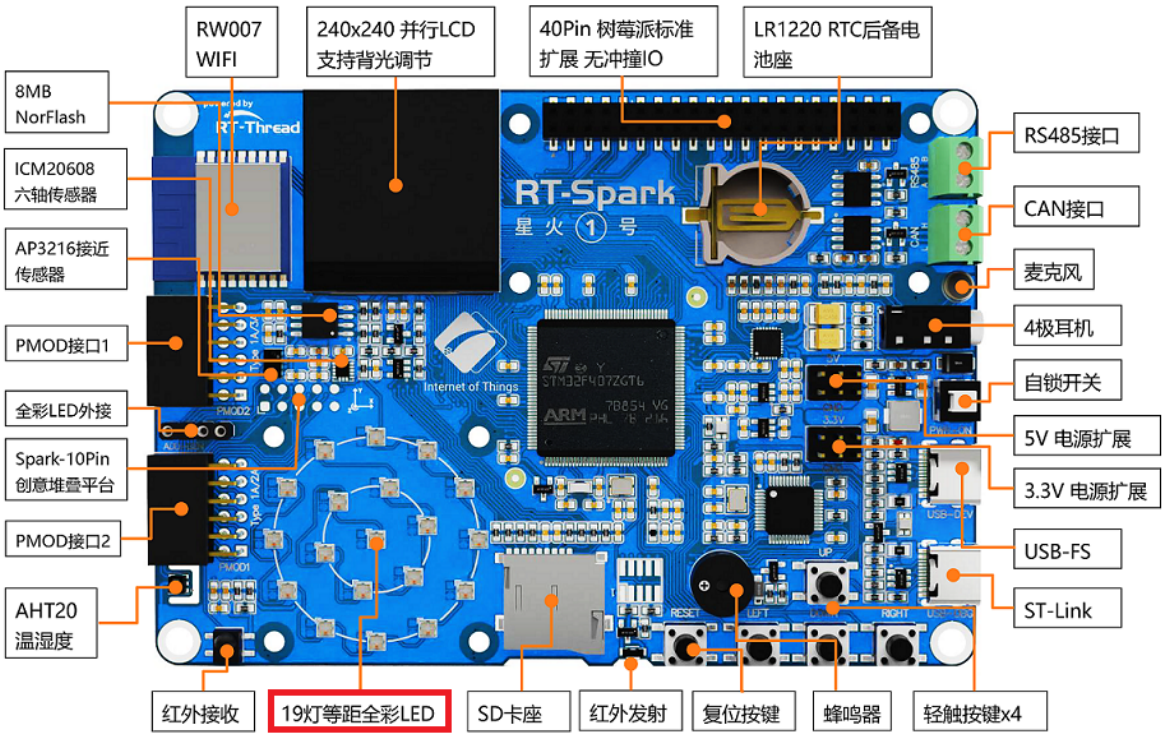

图 13.4: *LED 位置*

13.3 软件说明

本例程的源码位于 `/projects/03_driver_led_matrix`。

在 main 函数中，创建了一个 led matrix demo 线程并且启动该线程。

```
int main(void)
{
    led_matrix_thread = rt_thread_create("led matrix demo", led_matrix_example_entry
        , RT_NULL, 1024, 20, 20);
    if(led_matrix_thread == RT_NULL)
    {
        rt_kprintf("led matrix demo thread creat failed!\n");
        return 0;
    }
    rt_thread_mdelay(200); // avoid multi-thread on LED matrix transmit.
    rt_thread_startup(led_matrix_thread);

    return 0;
}
```

下面代码先定义了 LED MATRIX 中所有 LED 的序号，然后在 `led_matrix_example_entry()` 中循环的点亮 LED 。

```
/* define LED  */
enum{
    EXTERN_LED_0,
    EXTERN_LED_1,
    EXTERN_LED_2,
    EXTERN_LED_3,
    EXTERN_LED_4,
    EXTERN_LED_5,
    EXTERN_LED_6,
    EXTERN_LED_7,
    EXTERN_LED_8,
    EXTERN_LED_9,
    EXTERN_LED_10,
    EXTERN_LED_11,
    EXTERN_LED_12,
    EXTERN_LED_13,
    EXTERN_LED_14,
    EXTERN_LED_15,
    EXTERN_LED_16,
    EXTERN_LED_17,
    EXTERN_LED_18,
};

rt_thread_t led_matrix_thread;

static void led_matrix_example_entry()
{
    int count = 0;
    while (1)
    {
        for (int i = EXTERN_LED_0; i <= EXTERN_LED_18; i++)
        {
            switch (count)
            {
            case 0:
                led_matrix_set_color(i, RED);
                break;
            case 1:
                led_matrix_set_color(i, GREEN);
                break;
            case 2:
                led_matrix_set_color(i, BLUE);
                break;
            default:
                return;
                break;
            }
            led_matrix_reflash();
```

```
            rt_thread_delay(20);
        }
        count = (count + 1) % 3;
    }
}
```

13.4 运行

13.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

13.4.2 运行效果

按下复位按键重启开发板，观察开发板上 LED MATRIX 的实际效果。正常运行后，所有 LED 会按照红、绿、蓝的顺序逐个点亮如下图所示：

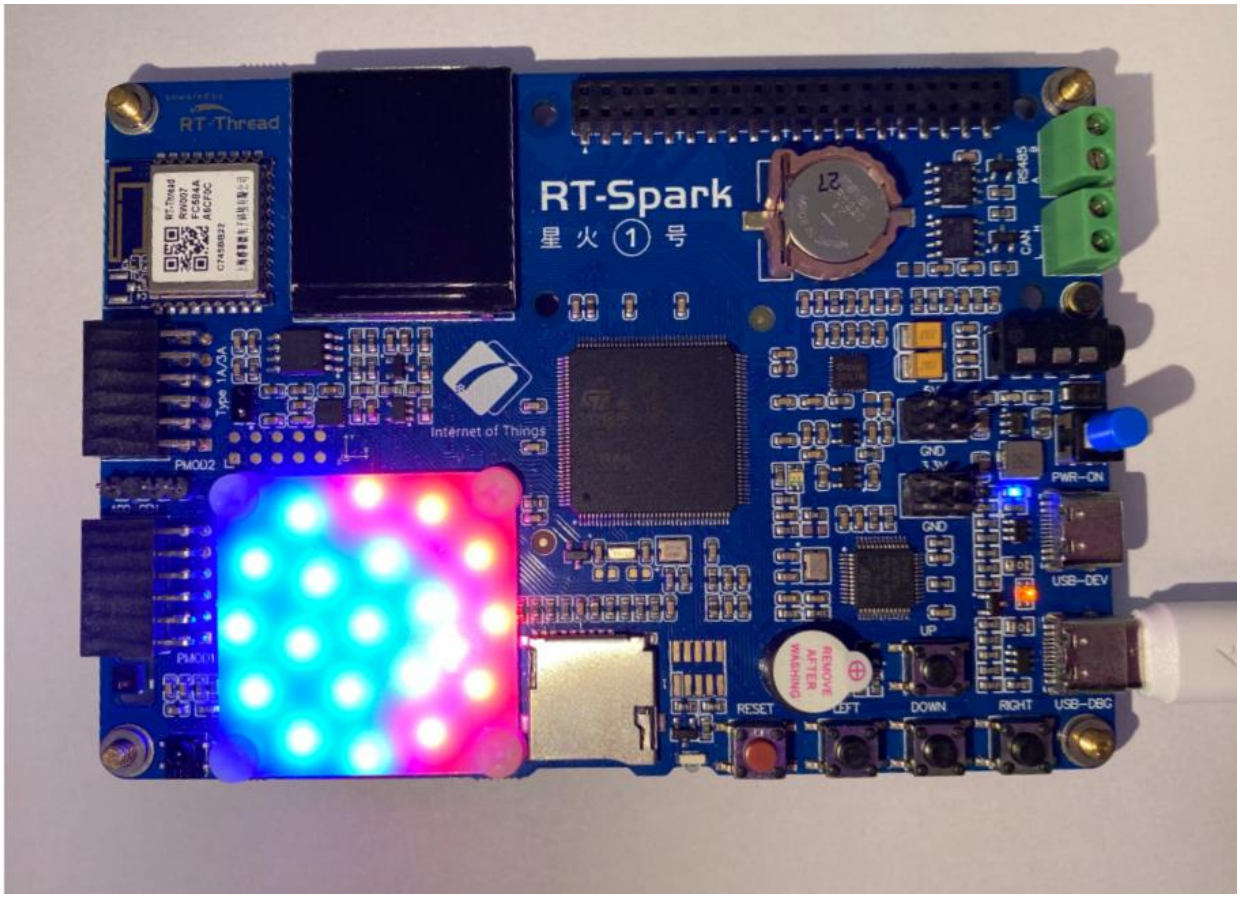

图 13.5: *LED 循环点亮*

13.5 注意事项

暂无。

13.6 引用参考

暂无

第 14 章

USB 鼠标例程

14.1 例程简介

本例程使用板载的六轴传感器 icm20608 获取开发板的旋转方向及角度，然后转换为鼠标的位移信息。同时使用板载按键实现鼠标的左右键，最后将这些鼠标信息通过 RT-Thread 的 USB 组件发送至电脑，从而实现开发板模拟 USB 鼠标的功能。

14.2 相关组件与软件包简介

14.2.1 RT-Thread USB 组件

该组件位于 /rt-thread/components/drivers/usb，是 RT-Thread 依据 USB2.0 协议规范将 USB 协议栈逻辑高度抽象，支持 host（主机）模式、device（从机）模式。

该组件允许用户通过宏 RT_USB_DEVICE_COMPOSITE 开启复合功能，无需额外的代码即可对多个设备类型进行复合，虚拟串口、以太网卡、人体学输入设备、大容量存储设备、微软通用 USB 等。

该组件在驱动移植方面提供了非常友好的移植接口，用户可将厂商 SDK（软件开发工具包）中 PCD（端口连接检测）驱动直接接入到 RT-Thread，实现 0 代码使用 USB。

14.2.2 ICM20608 软件包

该软件包位于 /projects/04_component_usb_mouse/packages/icm20608-v1.0.0 中，是 RT-Thread 针对六轴传感器 icm20608 功能使用的实现，使用这个软件包，可以让该传感器在 RT-Thread 上非常方便使用 icm20608 的基本功能，包括读取三轴加速度（3-axis accelerometer）、三轴陀螺仪（3-axis gyroscope）、零值校准等功能，如需详细了解该软件包，请参考 ICM20608 软件包中的 README。

14.3 硬件说明

如 USB 原理图所示：

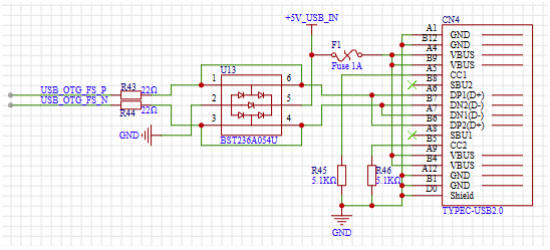

图 14.1: *USB 原理图*

USB-DEV 是本例程使用的 USB OTG 接口，其中 USB_D-(PA11) 、USB_D+(PA12) 为 USB 主机与从机的数据交互接口，本例程中开发板作为从机，电脑作为主机。

USB 接口、鼠标按键与传感器 icm20608 在开发板中的位置如下图所示:

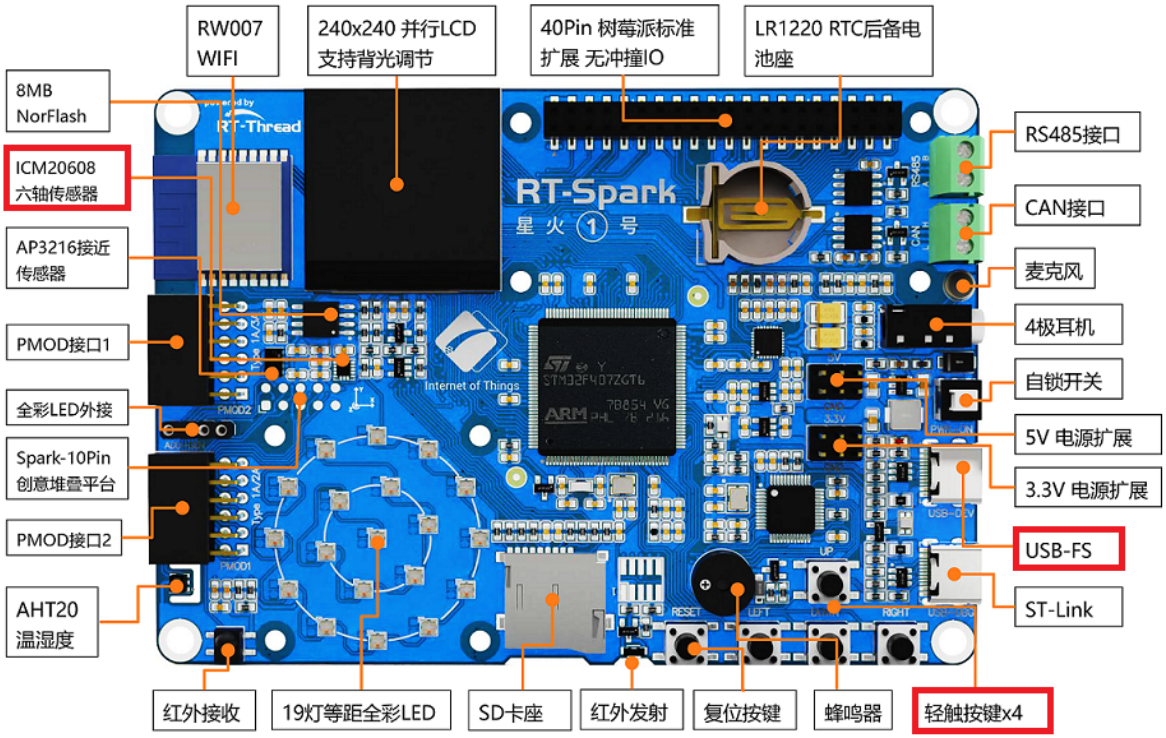

图 14.2: *硬件位置图*

14.4 软件说明

USB 鼠标例程位于 `/projects/04_component_usb_mouse` 目录下，重要文件摘要说明如下所示：

文件	说明
applications	应用
applications/main.c	app 入口
applications/SD_mouse.c	USB 鼠标应用主要文件

文件	说明
packages	内含简易好用的官方软件包
packages/icm20608-v1.0.0	六轴传感器

例程主要流程如下：

1. 初始化设备或者查找设备

   查找设备名称为 hidd 的 hid 设备，初始化 icm20608 传感器，初始化鼠标按键。

2. 打开设备

   打开查找到的 hid 设备。

3. 创建线程

   分别创建鼠标数据发送处理线程、传感器数据读取与处理线程、按键检测线程，并且启动这些创建的线程。

14.4.1 USB 鼠标功能指标定义

```
const static float mouse_rang_scope = 6.0f; /* 变 动 识 别 值 */
```

这个值设定开发板上下、左右移动变动识别值，可以自定义，值越小，鼠标越灵敏，越大鼠标越迟钝，但是应该大于 0，小于 45。

```
const static float mouse_angle_range = 80.0f; /* 读 取 角 度 有 效 角 度 */
```

这个值决定了六轴传感器 icm20608 读取角度控制值，范围为 0 - 90 度。

```
const static float mouse_move_range = 127.0f; /* 移 动 值 的 最 大 值 */
```

这个值决定了开发板倾斜角度转换成鼠标移动值的最大值，默认为最大值。

```
#define mouse_ratio (mouse_move_range / mouse_angle_range) /* 角 度 移 动 比 */
```

这个值由上面两个值决定，是将传感器读到的角度值转换成鼠标移动距离的比率。在角度一定的情况下，值越大，鼠标移动的距离越大，值越小，鼠标移动距离就小。

```
const static rt_uint8_t mouse_pixel_len = 5; /* 移 动 步 长 */
```

这个值决定了每次鼠标移动的步长，值越小，鼠标单次移动的就小，鼠标指针精度就越高，取值范围 0 - 127。

```
const static rt_uint32_t mouse_sample_times = 0; /* 鼠 标 响 应 时 间 */
```

这个值决定了鼠标响应时间，默认立即响应程序调度。初次使用，可以调大，便于观察日志。

14.4.2 原理性介绍

USB HID 鼠标实现流程图如下所示：

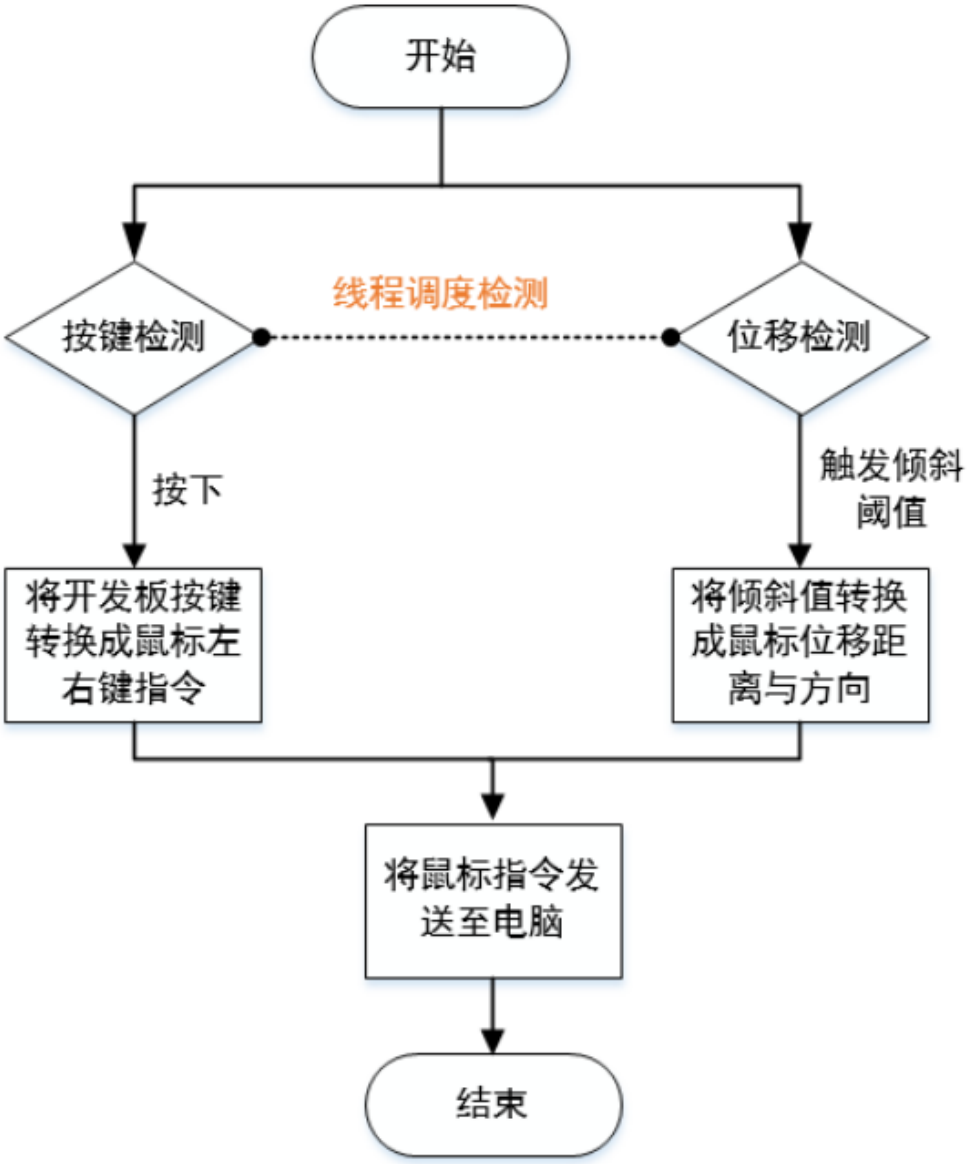

图 14.3: *原理流程图*

开发板开始正常运行后，RT-Thread 操作系统的线程调度函数，管理程序的运行，调度检测过程如下：

传感器数据读取与处理线程负责读取三轴加速度与三轴陀螺仪的测量值，在将测量值超过设置的上下、左右移动变动识别值，将读取到测量值，也就是倾斜值，转换成鼠标位移信息，其中倾斜值的正负号对应鼠标位移的方向，倾斜值的大小对应鼠标位移的大小，处理完成后将数据发送至电脑，完成鼠标位移操作。

按键检测线程负责检测开发板上按键是否被按下，若被按下，将响应的按键转换成鼠标的左键或者右键指令，处理完成后将数据发送至电脑，完成鼠标按键操作。

14.4.3 USB 数据例程程序入口

```
static int application_usb_init(void)
{
    /* 查找名称为 hidd 的设备 */
    rt_device_t device = rt_device_find("hidd");
    /* 初始化六轴传感器设备 */
    icm_device = mouse_init_icm();
    /* 初始化按键 */
    mouse_init_key();

    RT_ASSERT(device != RT_NULL);
    RT_ASSERT(icm_device != RT_NULL);

    /* 打开查找到的 hid 设备 */
    rt_device_open(device, RT_DEVICE_FLAG_WRONLY);

    /* 初始化 USB 线程 */
    rt_thread_init(&usb_thread,
                   "hidd",
                   usb_thread_entry, device,
                   usb_thread_stack, sizeof(usb_thread_stack),
                   10, 20);
    rt_thread_startup(&usb_thread);

    /* 初始化六轴传感器线程 */
    rt_thread_init(&icm_thread,
                   "icm20608",
                   icm_thread_entry, RT_NULL,
                   icm_thread_stack, sizeof(icm_thread_stack),
                   10, 20);
    rt_thread_startup(&icm_thread);

    /* 初始化按键线程 */
    rt_thread_init(&key_thread,
                   "key",
                   key_thread_entry, device,
                   key_thread_stack, sizeof(key_thread_stack),
                   10,20);
    rt_thread_startup(&key_thread);

    return 0;
}
```

14.4.4 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。

- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

14.4.5 运行效果

烧写程序之后，并将开发板的 USB OTG 接口与电脑连接，电脑通过设备管理器查询，可以发现多了一个鼠标设备，如下图所示：

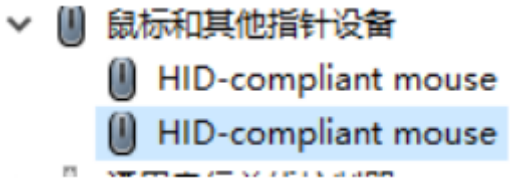

图 14.4: *运行日志*

打开串口，重启后，分别进行相关操作即可显示响应的日志，具体解释如下：

```
\ | /
- RT - Thread Operating System
/ | \ 4.0.1 build Mar 28 2019
2006 - 2019 Copyright by rt-thread team
[D/3D_mouse] The 3D mouse initializes success                    #鼠标初始化成
    功
msh >[D/3D_mouse] left down                                      #左键按下
[D/3D_mouse] left down
[D/3D_mouse] left down
[D/3D_mouse] right down                                          #右键按下
[D/3D_mouse] right down
[D/3D_mouse] right down
[D/3D_mouse] right down
[D/3D_mouse] move_max :   5, x:    0, y :   5                    #鼠标移动
[D/3D_mouse] move_max :   7, x:    3, y :   7
[D/3D_mouse] move_max :   7, x:    0, y :   7
[D/3D_mouse] move_max :  14, x:    0, y :  14
[D/3D_mouse] move_max :  14, x:    0, y :  14
[D/3D_mouse] move_max :  21, x:    0, y :  21
```

14.5 注意事项

暂无。

第 15 章

TF 卡文件系统例程

15.1 简介

本例程使用开发板上 TF 卡槽中的 TF 卡作为文件系统的存储设备，展示如何在 TF 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。

15.2 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 SDIO，原理图如下所示：

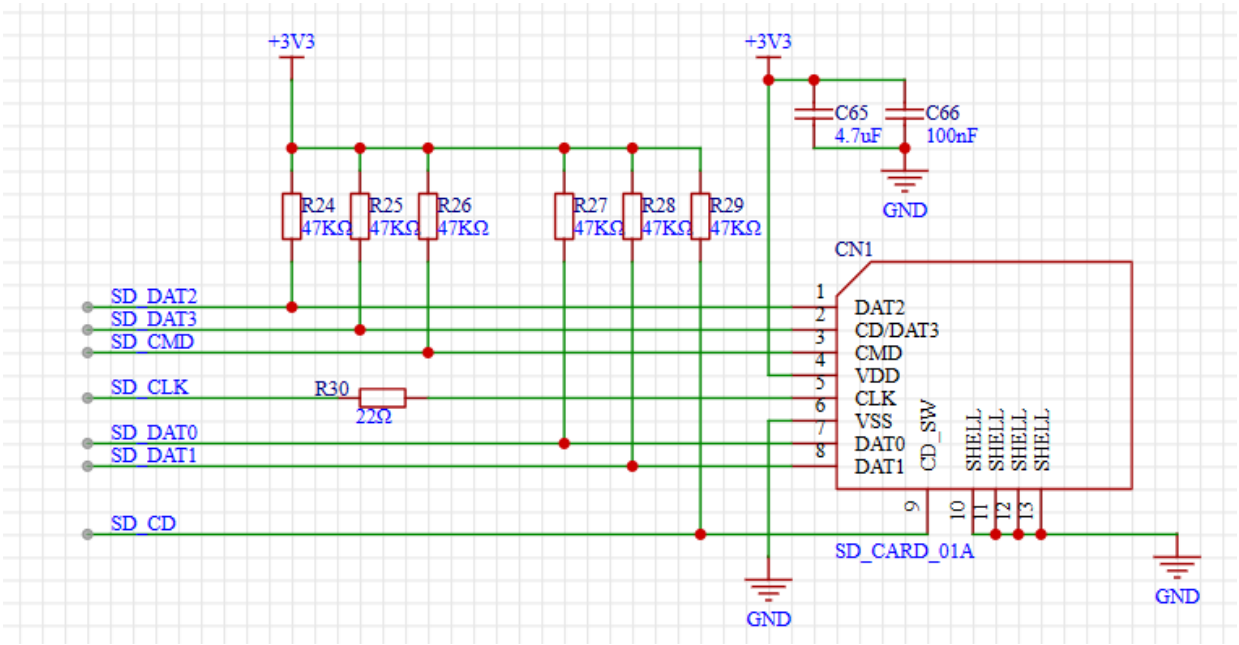

图 15.1: *TF 卡连接原理图*

TF 卡的卡槽在开发板中的位置如下图所示：

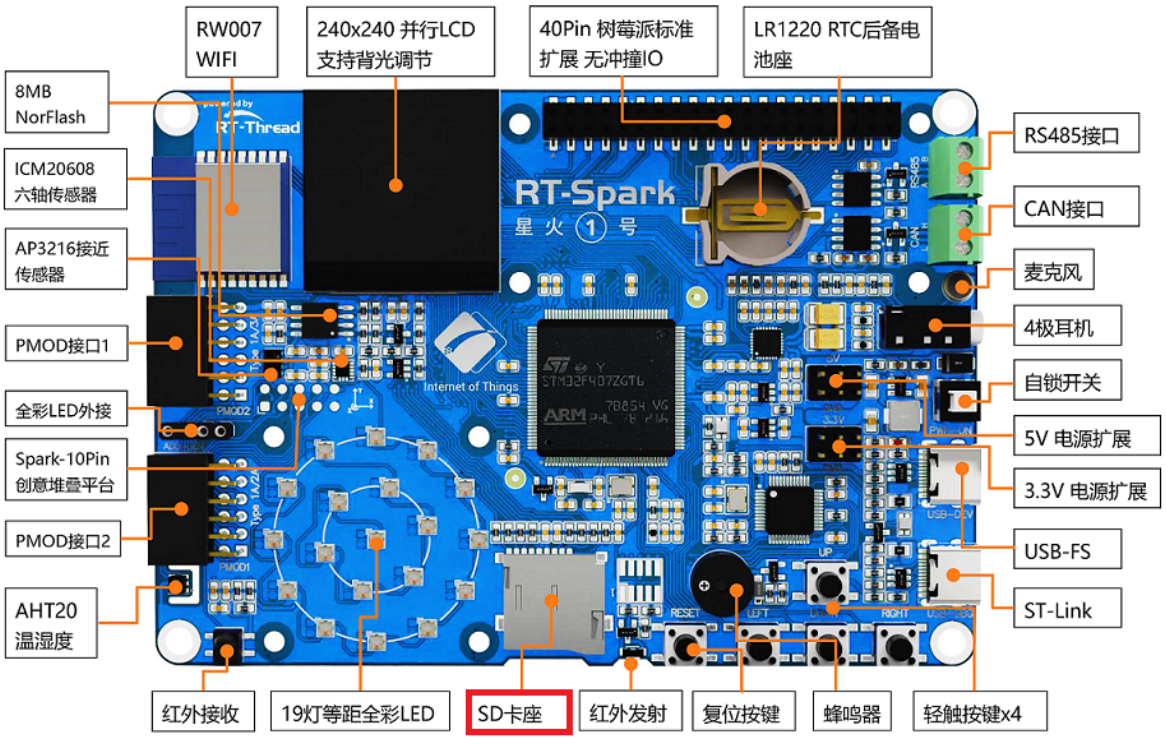

图 15.2: *TF 卡槽位置*

15.3 软件说明

本例程的源码位于 /projects/04_component_fs_tf_card。

15.3.1 挂载操作代码说明

挂载文件系统的源代码位于 applications/main.c 中。在示例代码中会将块设备 sd0 中的文件系统以 fatfs 文件系统格式挂载到根目录 / 上。

```
int main(void)
{
    /* 确保块设备注册成功之后再挂载文件系统 */
    rt_thread_delay(1000);

#ifdef BSP_USING_SDCARD_FATFS
    /* 挂载 TF 卡中的文件系统，参数 elm 表示挂载的文件系统类型为 elm-fat 文件系统 */
    if (dfs_mount("sd0", "/", "elm", 0, 0) == 0)
    {
        LOG_I("Filesystem initialized!");
    }
    else
    {
        LOG_E("Failed to initialize filesystem!");
```

```
    }
#endif /*BSP_USING_TF_CARD*/
    return 0;
}
```

15.4 运行

15.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

15.4.2 运行效果

1. 在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置 115200 8 1 N 。
2. 向开发板的 TF 卡槽里插入 TF 卡。
3. 按下复位按键重启开发板，如果看到提示 “Failed to initialize filesystem!”，这是因为 TF 卡中还没有创建文件系统。
4. 该步骤可选，如果确定自己的卡是 fat 格式，可以忽略。在 msh 中使用命令 mkfs -t elm sd0 可以在块设备 sd0 中创建 elm-fat 类型的文件系统，即对 TF 卡执行格式化。注意：mkfs 操作会清空存储设备中的数据，请谨慎操作。
5. 此时按下复位按键重启开发板，可以看到提示 “Filesystem initialized!”，表明文件系统挂载成功。打印信息如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 13 2023 09:54:41
 2006 - 2022 Copyright by RT-Thread team
msh />[I/SDIO] SD card capacity 1967104 KB.
[I/main] Filesystem initialized!
msh />ls                          # 使用 ls 命令查看文件系统目录信息
Directory /:                      # 可以看到已经存在根目录 /


msh />mkdir rt-thread             # 创建 rt-thread 文件夹
msh />ls                          # 查看目录信息如下
Directory /:
rt-thread <DIR>


msh />echo "hello rt-thread!!!"            # 将字符串输出到标准输出
```

```
hello rt-thread!!!
msh />echo "hello rt-thread!!!" hello.txt  # 将字符串输出到 hello.txt
msh />ls
Directory /:
rt-thread <DIR>
hello.txt 18
msh />

msh />cat hello.txt                       # 查看 hello.txt 文件的内容并输出
hello rt-thread!!!

msh />ls                                  # 查看当前目录信息
Directory /:
rt-thread <DIR>
hello.txt 18
msh />rm rt-thread                        # 删除 rt-thread 文件夹
msh />ls
Directory /:
hello.txt 18
msh />rm hello.txt                        # 删除 hello.txt 文件
msh />ls
Directory /:
msh />
```

15.5 注意事项

挂载文件系统之前一定要确认 TF 卡被格式化为 Fat 文件系统（使用 `mkfs` 格式化），否则会挂载失败。

15.6 引用参考

- 组件：虚拟文件系统组件

第 16 章

低功耗例程

16.1 简介

随着物联网 (IoT) 的兴起，产品对功耗的需求越来越强烈。作为数据采集的传感器节点通常需要在电池供电时长期工作，而作为联网的 SoC 也需要有快速的响应功能和较低的功耗。

在产品开发的起始阶段，首先考虑是尽快完成产品的功能开发。在产品功能逐步完善之后，就需要加入电源管理功能。为了适应 IoT 的这种需求，RT-Thread 提供了电源管理框架。电源管理框架的理念是尽量透明，使得产品加入低功耗功能更加轻松。

PM 组件有以下特点：

- PM 组件是基于模式来管理功耗
- PM 组件可以根据模式自动更新设备的频率配置，确保在不同的运行模式都可以正常工作
- PM 组件可以根据模式自动管理设备的挂起和恢复，确保在不同的休眠模式下可以正确的挂起和恢复
- PM 组件支持可选的休眠时间补偿，让依赖 OS Tick 的应用可以透明使用
- PM 组件向上层提供设备接口，如果使用了设备文件系统组件，那么也可以用文件系统接口来访问

本例程演示 RT-Thread 的电源管理组件 (Power Management，以下简称 PM 组件) 的使用。基于 PM 组件，用户可以很轻松地完成低功耗的开发。

16.2 硬件说明

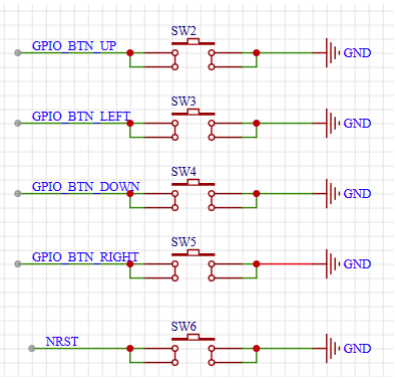

图 16.1: *按键电路原理图*

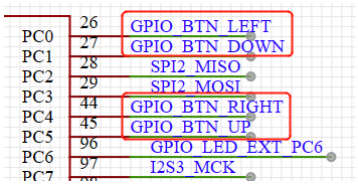

图 16.2: *按键连接单片机引脚*

如上图所示，例程里将使用 BTN_UP 按键唤醒处于休眠状态的 MCU。BTN_UP 按键被连接到单片机的 45 引脚 (PC5)，该引脚连接外部中断，通过中断唤醒 MCU。

16.3 软件说明

PM 例程的源代码位于 `/projects/04_component_pm/applications/main.c` 中。

我们使用 BTN_UP 按键来唤醒处于休眠模式的 MCU。一般情况下，在 MCU 处于比较深度的休眠模式，只能通过特定的方式唤醒。MCU 被唤醒之后，会触发相应的中断。以下例程是用 BTN_UP 按键从 `PM_SLEEP_MODE_DEEP` 唤醒 MCU, 唤醒之后红色 LED 持续被点亮 2 秒，期间蓝色 LED 持续闪烁。

例程的入口在 `main()` 函数里。

```
int main(void)
{
    /* 创建一个线程检测CPU是否还在继续运行 */
    rt_thread_t led_cycle_tid = rt_thread_create("led_cycle",led_cycle,RT_NULL
        ,512,21,10);
    if(led_cycle_tid == RT_NULL)
    {
        rt_kprintf("led_cycle_tid is null\n");
    }
    rt_thread_startup(led_cycle_tid);
    /* 唤醒回调函数初始化 */
    wakeup_init();
```

```
    /* 电源管理初始化 */
    pm_mode_init();
    rt_kprintf("pm_mode_init!\n");
    while (1)
    {
        rt_kprintf("wait sem!\n");
        /* 等待唤醒事件 */
        if (rt_event_recv(wakeup_event,
                          WAKEUP_EVENT_BUTTON,
                          RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                          RT_WAITING_FOREVER, RT_NULL) == RT_EOK)
        {
            led_app();
        }
    }
    return 0;
}
```

main() 函数里首先完成初始化工作：包括唤醒事件的初始化、PM 模式的配置、线程的创建；然后在循环里一直等待中断里发来的唤醒事件，如果接收到一次唤醒事件，就执行一次 led_app()。

wakeup_init() 包括了唤醒事件的初始化和唤醒按键中断回调函数的初始化：

```
static void wakeup_init(void)
{
    wakeup_event = rt_event_create("wakup", RT_IPC_FLAG_FIFO);
    RT_ASSERT(wakeup_event != RT_NULL);

    bsp_register_wakeup(wakeup_callback);
}
```

PM_SLEEP_MODE_DEEP 是 STM32F407 的 SLEEP 模式。我们希望停留在 PM_SLEEP_MODE_DEEP 模式，我们首先需要调用一次 rt_pm_request() 来请求该模式。

```
static void pm_mode_init(void)
{
    /* 上电之后默认高功耗模式，在这里退出高功耗模式 */
    rt_pm_release_all(PM_SLEEP_MODE_NONE);
    rt_pm_request(PM_SLEEP_MODE_DEEP);
}
```

led_app() 里我们希望点亮 LED 灯，并延时足够的时间以便我们观察到现象。在延时的时候，CPU 可能处于空闲状态，如果没有任何运行模式被请求，将会进入休眠。我们请求了PM_SLEEP_MODE_NONE 模式，即该模式下 CPU 不进行休眠，并在完成 LED 闪烁完成之后释放它，这样就可以让确保请求和释放中间的代码运行在 PM_SLEEP_MODE_NONE 模式。

```
static void led_app(void)
{
    rt_pm_request(PM_SLEEP_MODE_NONE);
```

```
    rt_pin_mode(PIN_LED_R, PIN_MODE_OUTPUT);
    rt_pin_write(PIN_LED_R, 0);
    rt_thread_mdelay(2000);
    rt_pin_write(PIN_LED_R, 1);
    rt_pm_release(PM_SLEEP_MODE_NONE);
}
```

led_cycle() 里我们希望循环的点亮蓝色 LED，目的是为了确认在低功耗模式下 CPU 是否停止了运行。

```
static void led_cycle()
{
    rt_pin_mode(PIN_LED_B, PIN_MODE_OUTPUT);
    while(1)
    {
        rt_pin_write(PIN_LED_B,1);
        rt_thread_mdelay(500);
        rt_pin_write(PIN_LED_B,0);
        rt_thread_mdelay(500);
    }
}
```

16.4 运行

16.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

16.4.2 运行效果

按下复位按键重启开发板，很快就进入休眠模式。我们三次按下 BTN_UP 按键。每次按下按键，MCU 都会被唤醒点亮红色 LED 2 秒并且期间蓝色 LED 持续闪烁，随后再次进入休眠。

16.5 注意事项

在芯片处于休眠模式下，一般是无法连接到仿真接口，所以就无法使用 ST-Link 下载程序。我们可以在芯片处于非休眠模式时下载，具体方式如下：

- 按住复位键，点击下载后，1 秒左右再松开复位键
- 按下 BTN_UP 键，马上点击下载

16.6 引用参考

- 设备与驱动：pin 设备
- 组件：电源管理组件

第 17 章

Flash 分区管理例程

本例程演示如何通过 RT-Thread 提供的 FAL 组件对 Flash 进行分区管理操作。例程中，通过调用 FAL 接口完成了对指定分区的测试工作，完成了对 Flash 读、写、擦的测试，同时也通过该例程完成了对 Flash 驱动的基本测试。

17.1 FAL 简介

FAL (Flash Abstraction Layer) Flash 抽象层，是 RT-Thread 的一个组件，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 ，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 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；

本例程旨在演示如何使用 fal 管理多个 Flash 设备，指导用户通过 fal 分区表操作 Flash 设备。该例程将是后续 OTA、easyflash 等例程的基础。通过本例程指导用户学习使用 fal 处理以下问题：

- 使用 fal 管理多个 Flash 设备
- 创建分区表
- 使用 fal 操作分区表

17.2 硬件说明

本例程使用到的硬件资源如下所示：

- UART1
- 片内 FLASH (1MBytes)
- 片外 Nor Flash (8MBytes)

17.3 软件说明

本例程的源码位于 /projects/04_component_fal。

文件路径	文件描述
applications/main.c	app 入口（fal 例程程序）
libraries/Board_Drivers/fal/fal_cfg.h	fal 配置文件（Flash 设备配置和分区表配置）
libraries/Board_Drivers/fal/fal_spi_flash_sfud_port.c	fal 操作片外 Nor Flash 的移植文件（将 Flash 读写擦除接口注册到 fal）
libraries/HAL_Drivers/drv_flash/drv_flash_f4.c	fal 操作片内 Flash 的移植文件（将 Flash 读写擦除接口注册到 fal）
rt-thread/components/fal	FAL 组件（fal 源码实现）
rt-thread/components/fal/inc/fal.h	FAL 组件对外提供的操作接口

从上表中可以看到，如果要使用 FAL 组件需要进行必要的移植工作，移植文件存放在 ports 目录下。本 stm32F4 平台已经完成相关的移植工作。

17.3.1 fal 配置说明

fal 配置存放在 libraries/Board_Drivers/fal/fal_cfg.h 文件中，主要包括 Flash 设备的配置、分区表的配置。

Flash 设备配置

fal 中使用 Flash 设备列表管理多个 Flash 设备。本例程中涉及到两个 Flash 设备，STM32F4 芯片内的 Flash 和片外的 QSPI Flash（Nor Flash）。

```
/* flash device table */
#define FAL_FLASH_DEV_TABLE                                          \
{                                                                    \
    &stm32_onchip_flash_128k,                                        \
    &w25q64,                                                        \
}
```

- stm32_onchip_flash_128k 是 STM32F4 片内 Flash 设备，定义在 libraries/Board_Drivers/fal/fal_spi_flash_sfud_port.c 文件中，参考 Flash 设备对接说明 章节。
- w25q64 是外部扩展的 Nor FLASH 设备，定义在 libraries/HAL_Drivers/drv_flash/drv_flash_f4.c 文件中，参考 Flash 设备对接说明 章节。

17.3.2 分区表配置

分区表存放在 libraries/Board_Drivers/fal/fal_cfg.h

```
#define FAL_PART_TABLE

    \
{

    \
    {FAL_PART_MAGIC_WROD,        "app", "onchip_flash_128k",
                                  0,       384 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD,      "param", "onchip_flash_128k",                    384 *
         1024,       640 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD,  "easyflash",    "W25Q64",
                                       0,       512 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD,   "download",    "W25Q64",                          512 *
         1024,      1024 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD, "wifi_image",    "W25Q64",                 (512 + 1024) *
         1024,       512 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD,       "font",    "W25Q64",            (512 + 1024 + 512) *
         1024,  3 * 1024 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD, "filesystem",    "W25Q64", (512 + 1024 + 512 + 3 * 1024) *
         1024,  3 * 1024 * 1024, 0}, \
}
```

这里有一个宏定义 FAL_PART_HAS_TABLE_CFG，如果定义，则表示应用程序使用 fal_cfg.h 文件中定义的分区表。是否使用 fal_cfg.h 文件中定义的分区表，有这样一个准则:

- 如果使用 bootloader 则不定义 FAL_PART_HAS_TABLE_CFG 宏，而使用 bootloader 中定义的分区表
- 如果不使用 bootloader 则需要用户定义 FAL_PART_HAS_TABLE_CFG 宏，从而使用 fal_cfg.h 文件中定义的分区表 fal_cfg.h 文件中定义的分区表最终会注册到 struct fal_partition 结构体数组中。

fal_partition 结构体定义如下所示：

```
struct fal_partition
{
    uint32_t magic_word;
    /* FLASH 分 区 名 称 */
    char name[FAL_DEV_NAME_MAX];
    /* FLASH 分 区 所 在 的 FLASH 设 备 名 称 */
    char flash_name[FAL_DEV_NAME_MAX];
    /* FLASH 分 区 在 FLASH 设 备 的 偏 移 地 址 */
    long offset;
    size_t len;
    uint8_t reserved;`
};
```

fal_partition 结构体成员简要介绍如下所示：

成员变量	说明
magic_word	魔法数，系统使用，用户无需关心
name	分区名字，最大 23 个 ASCII 字符
flash_name	分区所属的 Flash 设备名字，最大 23 个 ASCII 字符
offset	分区起始地址相对 Flash 设备起始地址的偏移量
len	分区大小，单位字节
reserved	保留项

17.3.3 Flash 设备对接说明

fal 是 Flash 抽象层，要操作 Flash 设备必然要将 Flash 的读、写、擦接口对接到 fal 抽象层中。在 fal 中，使用 `struct fal_flash_dev` 结构体来让用户注册该 Flash 设备的操作接口。

`fal_flash_dev` 结构体定义如下所示：

```
struct fal_flash_dev
{
    char name[FAL_DEV_NAME_MAX];
    /* FLASH 设备的起始地址 */
    uint32_t addr;
    size_t len;
    /* FLASH 设备最小擦除的块大小 */
    size_t blk_size;

    struct
    {
        int (*init)(void);
        int (*read)(long offset, uint8_t *buf, size_t size);
        int (*write)(long offset, const uint8_t *buf, size_t size);
        int (*erase)(long offset, size_t size);
    } ops;

    /* write minimum granularity, unit: bit.
    1(nor flash)/ 8(stm32f2/f4)/ 32(stm32f1)/ 64(stm32l4)
    0 will not take effect. */
    size_t write_gran;
};
```

`fal_flash_dev` 结构体成员简要介绍如下所示：

成员变量	说明
name	Flash 设备名字，最大 23 个 ASCII 字符
addr	Flash 设备的起始地址（片内 Flash 为 0x08000000，片外 Flash 为 0x00）

成员变量	说明
len	Flash 设备容量，单位字节
blk_size	Flash 设备最小擦除单元的大小，单位字节
ops.init	Flash 设备的初始化函数，会在 fal_init 接口中调用
ops.read	Flash 设备数据读取接口
ops.write	Flash 设备数据写入接口
ops.erase	Flash 设备数据擦除接口
write_gran	Flash 设备写入最小粒度，单位为位。

17.3.3.1 片内 Flash 对接说明

片内 Flash 设备实例定义在 libraries/HAL_Drivers/drv_flash/drv_flash_f4.c 文件中，如下所示：

```
const struct fal_flash_dev stm32_onchip_flash_128k =
{
    "onchip_flash_128k",
    STM32_FLASH_START_ADRESS_128K,
    FLASH_SIZE_GRANULARITY_128K,
    (128 * 1024),
    {
        NULL,
        fal_flash_read_128k,
        fal_flash_write_128k,
        fal_flash_erase_128k,
    },
    8,
};
```

Flash 设备名称为 onchip_flash_128k，最小擦除单元为 128K，无初始化接口。

17.3.3.2 片外 Nor Flash 对接说明

片外 Nor Flash 设备实例定义在 libraries/Board_Drivers/fal/fal_spi_flash_sfud_port.c 文件中，使用了 RT-Thread 内置的 SFUD 框架。

> SFUD 是一款开源的串行 SPI Flash 通用驱动库，覆盖了市面上绝大多数串行 Flash 型号，无需软件开发就能驱动。

Nor Flash 设备实例如下所示：

```
struct fal_flash_dev w25q64 =
{
    .name       = "W25Q64",
    .addr       = 0,
```

```
    .len        = 8 * 1024 * 1024,
    .blk_size   = 4096,
    .ops        = {init, read, write, erase},
    .write_gran = 1
};
```

Flash 设备名称为 W25Q64，设备容量为 8M，最小擦除单元为 4K。这里使用的 read、write、erase 接口最终调用 SFUD 框架中的接口，无需用户进行驱动开发。

17.3.4 例程使用说明

fal 例程代码位于 application/main.c 文件中。例程中封装了一个分区测试函数 fal_test，如下所示：

```
static int fal_test(const char *partiton_name);
```

fal_test 函数输入参数为 Flash 分区的名字，功能是对输入分区进行完整的擦、读、写测试，覆盖整个分区。

注意，fal 在使用前，务必使用 fal_init 函数完成 fal 功能组件的初始化。

以擦除为例，对代码进行简要说明：

17.3.4.1 1. 擦除整个分区

```
/* 擦除 `partition` 分区上的全部数据 */
ret = fal_partition_erase_all(partition);
```

使用 fal_partition_erase_all API 接口将 download 分区完整擦除，擦除后 download 分区内的数据全为 **0xFF**。

17.3.4.2 2. 校验擦除操作是否成功

```
/* 循环读取整个分区的数据，并对内容进行检验 */
for (i = 0; i < partition->len;)
{
    rt_memset(buf, 0x00, BUF_SIZE);
    len = (partition->len - i) > BUF_SIZE ? BUF_SIZE : (partition->len - i);

    /* 从 Flash 读取 len 长度的数据到 buf 缓冲区 */
    ret = fal_partition_read(partition, i, buf, len);
    if (ret < 0)
    {
        LOG_E("Partition (%s) read failed!", partition->name);
        ret = -1;
        return ret;
    }
    for(j = 0; j < len; j++)
    {
```

```
        /* 校验数据内容是否为 0xFF */
        if (buf[j] != 0xFF)
        {
            LOG_E("The erase operation did not really succeed!");
            ret = -1;
            return ret;
        }
    }
    i += len;
}
```

通过上面的代码，循环读取整个分区的数据，并对数据内容进行校验，判断是否为 0xFF，校验通过则说明擦除操作正常。

17.3.4.3 3. 写整个分区

```
    /* 把 0 写入指定分区 */
    for (i = 0; i < partition->len;)
    {
        /* 设置写入的数据 0x00 */
        rt_memset(buf, 0x00, BUF_SIZE);
        len = (partition->len - i) > BUF_SIZE ? BUF_SIZE : (partition->len - i);

        /* 写入数据 */
        ret = fal_partition_write(partition, i, buf, len);
        if (ret < 0)
        {
            LOG_E("Partition (%s) write failed!", partition->name);
            ret = -1;
            return ret;
        }
        i += len;
    }
    LOG_I("Write (%s) partition finish! Write size %d(%dK).", partiton_name, i, i /
        1024);
```

通过上面的代码，循环写入数据 0x00 到整个分区。

17.3.4.4 4. 校验写操作是否成功

```
    /* 从指定的分区读取数据并校验数据 */
    for (i = 0; i < partition->len;)
    {
        /* 清空读缓冲区，以 0xFF 填充 */
        rt_memset(buf, 0xFF, BUF_SIZE);
        len = (partition->len - i) > BUF_SIZE ? BUF_SIZE : (partition->len - i);

        /* 读取数据到 buf 缓冲区 */
```

```
        ret = fal_partition_read(partition, i, buf, len);
        if (ret < 0)
        {
            LOG_E("Partition (%s) read failed!", partition->name);
            ret = -1;
            return ret;
        }
        for(j = 0; j < len; j++)
        {
            /* 校验读取的数据是否为步骤 3 中写入的数据 0x00 */
            if (buf[j] != 0x00)
            {
                LOG_E("The write operation did not really succeed!");
                ret = -1;
                return ret;
            }
        }
        i += len;
    }
```

通过上面的代码，循环读取整个分区的数据，并对数据内容进行校验，判断是否为步骤 3 写入的数据 0x00，校验通过则说明写操作正常。

17.4 运行

17.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

17.4.2 运行效果

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 13 2023 11:29:32
 2006 - 2022 Copyright by RT-Thread team
[I/SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[I/SFUD] W25Q64 flash device is initialize success.
[I/SFUD] Probe SPI flash W25Q64 by SPI device spi20 success.
[D/FAL] (fal_flash_init:49) Flash device |        onchip_flash_128k | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00020000 |initialized finish.
```

```
[D/FAL] (fal_flash_init:49) Flash device |                  W25Q64 | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev         |   offset    |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | app        | onchip_flash_128k | 0x00000000 | 0x00060000 |
[I/FAL] | param      | onchip_flash_128k | 0x00060000 | 0x000a0000 |
[I/FAL] | easyflash  | W25Q64            | 0x00000000 | 0x00080000 |
[I/FAL] | download   | W25Q64            | 0x00080000 | 0x00100000 |
[I/FAL] | wifi_image | W25Q64            | 0x00180000 | 0x00080000 |
[I/FAL] | font       | W25Q64            | 0x00200000 | 0x00300000 |
[I/FAL] | filesystem | W25Q64            | 0x00500000 | 0x00300000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/main] Flash device : onchip_flash_128k   Flash size : 1024K   Partition : param
     Partition size: 640K
[I/main] Erase (param) partition finish!
[I/main] Write (param) partition finish! Write size 655360(640K).
[I/main] Fal partition (param) test success!
[I/main] Flash device : W25Q64   Flash size : 8192K   Partition : download
    Partition size: 1024K
```

17.5 SHELL 命令

为了方便用户验证 fal 功能是否正常，以及 Flash 驱动是否正确工作，分区表配置是否合理，RT-Thread 为 fal 提供了一套测试命令。

fal 测试命令如下所示：

```
msh >fal
Usage:
fal probe [dev_name|part_name] - probe flash device or partition by given name
fal read addr size - read 'size' bytes starting at 'addr'
fal write addr data1 ... dataN - write some bytes 'data' starting at 'addr'
fal erase addr size - erase 'size' bytes starting at 'addr'
fal bench <blk_size> - benchmark test with per block size
```

- 使用 `fal probe [dev_name|part_name]` 命令探测指定的 Flash 设备或者 Flash 分区

  当探测到指定的 Flash 设备或分区后，会显示其属性信息，如下所示：

```
msh >fal probe W25Q64
Probed a flash device | W25Q64 | addr: 0 | len: 8388608 |.
msh >fal probe download
Probed a flash partition | download | flash_dev: W25Q64 | offset: 524288 | len:
    1048576 |.
```

- 擦除数据

首先选择要擦除数据的分区，演示使用的是 download 分区，然后使用 fal erase 命令擦除，如下所示：

```
msh >fal probe download
Probed a flash partition | download | flash_dev: W25Q64 | offset: 524288 | len:
msh >fal erase 0 4096
Erase data success. Start from 0x00000000, size is 4096.
msh >
```

其中，使用擦除命令时，addr 为相应探测 Flash 分区的偏移地址，size 为不超过该分区的值，以下写入数据、读取数据与此类似。

- 写入数据

在完成擦除操作后，才能在已擦除区域写入数据，先输入 fal write ，后面跟着 N 个待写入的数据，并以空格隔开（能写入的数据数量取决与 MSH 命令行的配置）。演示从地址 0x00000008 的位置开始写入数据 1 2 3 4 5 6 7，共 7 个数据，如下所示：

```
msh >fal write 8 1 2 3 4 5 6 7
Write data success. Start from 0x00000008, size is 7.
Write data: 1 2 3 4 5 6 7 .
```

- 读取数据

先输入 fal read ，后面跟着待读取数据的起始地址以及长度。演示从 0 地址开始读取 64 字节数据，读取前面写入的数据，如下所示：

```
msh >fal read 0 64
Read data success. Start from 0x00000000, size is 64. The data is:
Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
[00000000] FF FF FF FF FF FF FF FF 01 02 03 04 05 06 07 FF
[00000010] FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[00000020] FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[00000030] FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

从日志上可以看到，在 0x00000008 地址处开始就是演示所写入的 7 个数据。– 注：Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 作为读取数据的行号标记。

- 性能测试

  性能测试将会测试 Flash 的擦除、写入及读取速度，同时将会测试写入及读取数据的准确性，保证整个 Flash 或整个分区的写入与读取数据的一致性。

  先输入 `fal bench`，后面跟着待测试 Flash 的扇区大小（请查看对应的 Flash 手册，SPI Nor Flash 一般为 4096）。由于性能测试将会让整个 Flash 或者整个分区的数据丢失，所以命令最后必须跟 yes 。

```
msh >fal bench 4096 yes
Erasing 1048576 bytes data, waiting...
Erase benchmark success, total time: 3.744S.
Writing 1048576 bytes data, waiting...
Write benchmark success, total time: 4.096S.
Reading 1048576 bytes data, waiting...
Read benchmark success, total time: 1.961S.
```

17.6 注意事项

- 如果要修改分区表，请正确配置起始地址和分区大小，不要有分区重叠
- 在使用 fal 测试命令的时候，请先使用 `fal probe` 命令选择一个 Flash 分区

17.7 引用参考

- 文档中心：RT-Thread 文档中心
- FAL 组件

第 18 章

KV 参数存储例程

18.1 简介

本例程将演示使用 EasyFlash 存储 KV 参数，记录开机次数。

18.2 背景知识

KV 是 key-value（键值对）的缩写，保存数据时，都是 key 和 value 一起保存，读取数据时，只要输入 key 值，就可读取到 value 值，十分方便。本例程中的环境变量就是利用 KV 值存储的。

EasyFlash 是一款开源的轻量级嵌入式 Flash 存储器库，主要为 MCU 提供便捷、通用的上层应用接口，使得开发者更加高效实现基于的 Flash 存储器常见应用开发。

该库目前提供三大实用功能：

- Env 快速保存产品参数，支持写平衡（磨损平衡）及掉电保护模式。

  EasyFlash 不仅能够实现对产品的设定参数或运行日志等信息的掉电保存功能，还封装了简洁的增加、删除、修改及查询方法，降低了开发者对产品参数的处理难度，也保证了产品在后期升级时拥有更好的扩展性。让 Flash 变为 NoSQL（非关系型数据库）模型的小型键值（Key-Value）存储数据库。

- IAP 在线升级再也不是难事儿

  该库封装了 IAP（In-Application Programming）功能常用的接口，支持 CRC32 校验，同时支持 Bootloader 及 Application 的升级。

- Log 无需文件系统，日志可直接存储在 Flash 上

  非常适合应用在小型的不带文件系统的产品中，方便开发人员快速定位、查找系统发生崩溃或死机的原因。同时配合 EasyLogger（开源的超轻量级、高性能 C 日志库，它提供与 EasyFlash 的无缝接口）一起使用，轻松实现 C 日志的 Flash 存储功能。

18.3 硬件说明

本例程使用到的硬件资源如下所示：

- UART1
- 片内 FLASH (1024KBytes)
- 片外 Nor Flash (8MBytes)

18.4 软件说明

本例程的源码位于 `/projects/04_component_kv`。

18.4.1 EasyFlash 配置说明

EasyFlash 配置存放在 packages/EasyFlash-v3.2.1/inc/ef_cfg.h 文件中，主要包括环境变量功能的配置、在线升级功能的配置和日志功能的配置等。

本例程只演示环境变量功能，配置的是掉电保护模式（在写入环境变量时出现掉电现象，写入前的数据并不会丢失）。同时还开启了环境变量自动更新功能。当版本号变动时，会自动追加新添加的环境变量。设定 ENV 缓冲区的大小为 2K，擦写的最小粒度为 4K。详细的配置说明见 官方主页。

18.4.2 EasyFlash 移植说明

EasyFlash 在使用前需要进项移植，不同的底层驱动，移植方法也不一样。本例程的底层驱动是 FAL，对 FAL 还不熟悉的用户可以去学习上一个例程。

基于 FAL 的移植十分方便，因为官方已经做好了基于 FAL 的移植文件。

移植主要分为 3 步：

1. 复制移植文件 `ef_fal_port.c`

   从 packages/EasyFlash-v3.2.1/ports 复制到 libraries/Board_Drivers/ef_fal_port.c。

2. 修改 `FAL_EF_PART_NAME` 宏定义（存储环境变量的分区名）的值

   本例程里存储环境变量的分区名为 easyflash。

3. 修改 `static const ef_env default_env_set[]` 数组里的环境变量

   本例程只记录开机次数，所以数组里只有 {“boot_times”, “0”} 一个环境变量。

详细的移植说明见移植参考示例。

18.4.3 例程使用说明

在 main 函数中，首先执行的是 FAL 的初始化，完成分区表的加载。接着执行 EasyFlash 的初始化，初始化成功后，会执行读取和写入 KV 值的 demo。EasyFlash 会将 KV 参数存储在 easyflash 分区。

例程启动后，会从 easyflash 分区读取 boot_times（开机次数）参数，读取成功后，将字符串转换成数字，累加后再次存储到 easyflash 分区，并将开机次数通过串口打印出来。

```
int main(void)
{
    fal_init();

    if (easyflash_init() == EF_NO_ERR)
    {
        /* 演示环境变量功能 */
        test_env();
    }

    return 0;
}

static void test_env(void)
{
    uint32_t i_boot_times = 0;
    char *c_old_boot_times, c_new_boot_times[11] = {0};

    /* 从环境变量中获取启动次数 */
    c_old_boot_times = ef_get_env("boot_times");
    /* 获取启动次数是否失败 */
    if (c_old_boot_times == RT_NULL)
        c_old_boot_times[0] = '0';

    i_boot_times = atol(c_old_boot_times);
    /* 启动次数加 1 */
    i_boot_times++;
    LOG_D("================================================");
    LOG_D("The system now boot %ld times", i_boot_times);
    LOG_D("================================================");
    /* 数字转字符串 */
    sprintf(c_new_boot_times, "%ld", i_boot_times);
    /* 保存开机次数的值 */
    ef_set_env("boot_times", c_new_boot_times);
    ef_save_env();
}
```

18.5 运行

18.5.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

18.5.2 运行效果

开机后开发板会通过串口自动打印开机次数，示例如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun  9 2023 14:36:52
 2006 - 2022 Copyright by RT-Thread team
[I/SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[I/SFUD] W25Q64 flash device is initialize success.
[I/SFUD] Probe SPI flash W25Q64 by SPI device spi20 success.
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[Flash] (../packages/EasyFlash-v4.1.0/src/ef_env.c:1818) ENV start address is 0
    x00000000, size is 8192 bytes.
[Flash] EasyFlash V4.1.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[D/main] ============================================
[D/main] The system now boot 6 times
[D/main] ============================================
```

按下复位按键，可以看到开机次数加一。

EasyFlash 还提供了 5 个测试命令，可以在 FinSH 里非常方便的查看，修改环境变量。

- **setenv**：设置环境变量
- **printenv**：打印环境变量
- **saveenv**：存储环境变量
- **getvalue**：获取环境变量值
- **resetenv**：复位环境变量

下面是这些命令的使用示例。

```
msh >printenv                 # 打印所有的环境变量
boot_times=13

mode: power fail safeguard
size: 32/2048 bytes, write bytes 64/8192.
saved count: 15
ver num: 0
msh >getvalue boot_times      # 获取 boot_times 的值
The boot_times value is 13.
msh >setenv boot_times 5      # 设置 boot_times 的值为5
msh >saveenv                  # 保存环境变量
[Flash] Erased ENV OK.
[Flash] Saved ENV OK.
msh >getvalue boot_times      # 获取 boot_times 的值
The boot_times value is 5.
```

```
msh >resetenv                # 复位环境变量的值
[Flash] Erased ENV OK.
[Flash] Saved ENV OK.
[Flash] Erased ENV OK.
[Flash] Saved ENV OK.
msh >printenv                # 打印所有的环境变量
boot_times=0

mode: power fail safeguard
size: 32/2048 bytes, write bytes 64/8192.
saved count: 2
ver num: 0
msh >setenv boot_times       # 删除 boot_times 环境变量
msh >saveenv                 # 保存环境变量
msh >printenv                # 打印所有的环境变量

mode: power fail safeguard
size: 16/2048 bytes, write bytes 32/8192.
saved count: 9
ver num: 0
```

18.6 引用参考

- 文档中心：RT-Thread 文档中心
- EasyFlash 软件包

第 19 章

SPI Flash 文件系统例程

19.1 简介

本例程使用板载的 SPI Flash 作为文件系统的存储设备，展示如何在 Flash 的指定分区上创建文件系统，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。

本例程中使用的是 FAT 文件系统，也支持 Littlefs 文件系统。Littlefs 文件系统的使用可以参考 RT-Thread 文档中心。

由于本例程需要使用 fal 组件对存储设备进行分区等操作，所以在进行本例程的实验前，需要先进行 fal 例程的实验，对 fal 组件的使用有一定的了解。

19.2 硬件说明

本次示例和存储器连接通过 SPI 接口，使用的硬件接口是 SPI2，原理图如下所示：

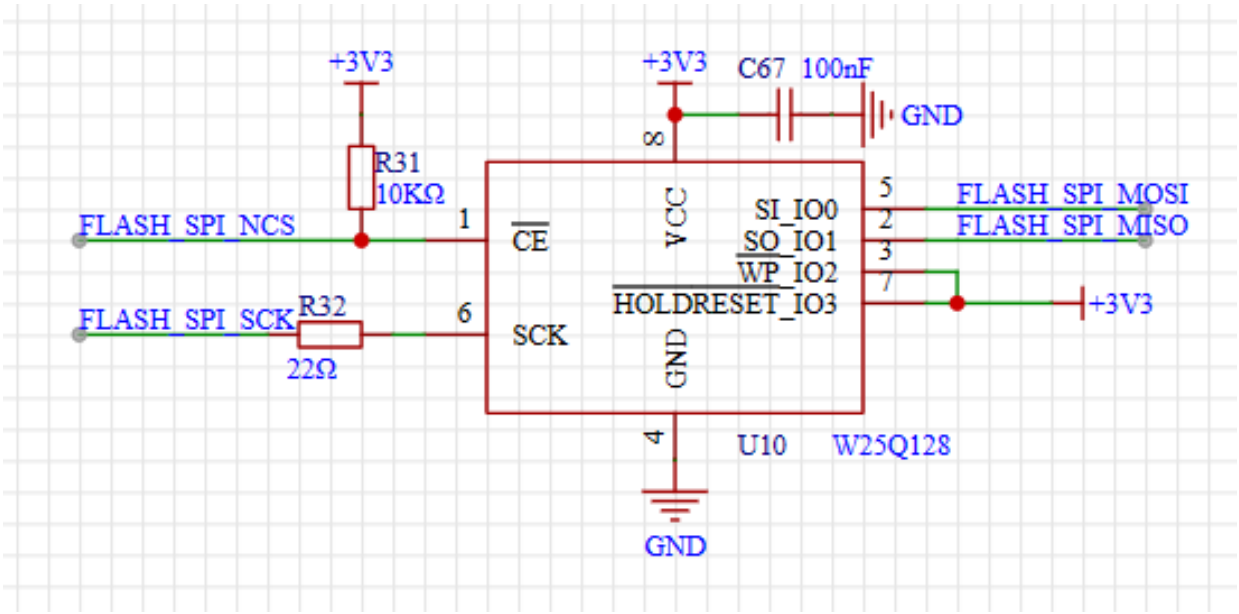

图 19.1: *SPI FLASH 原理图*

SPI FLASH 在开发板中的位置如下图所示：

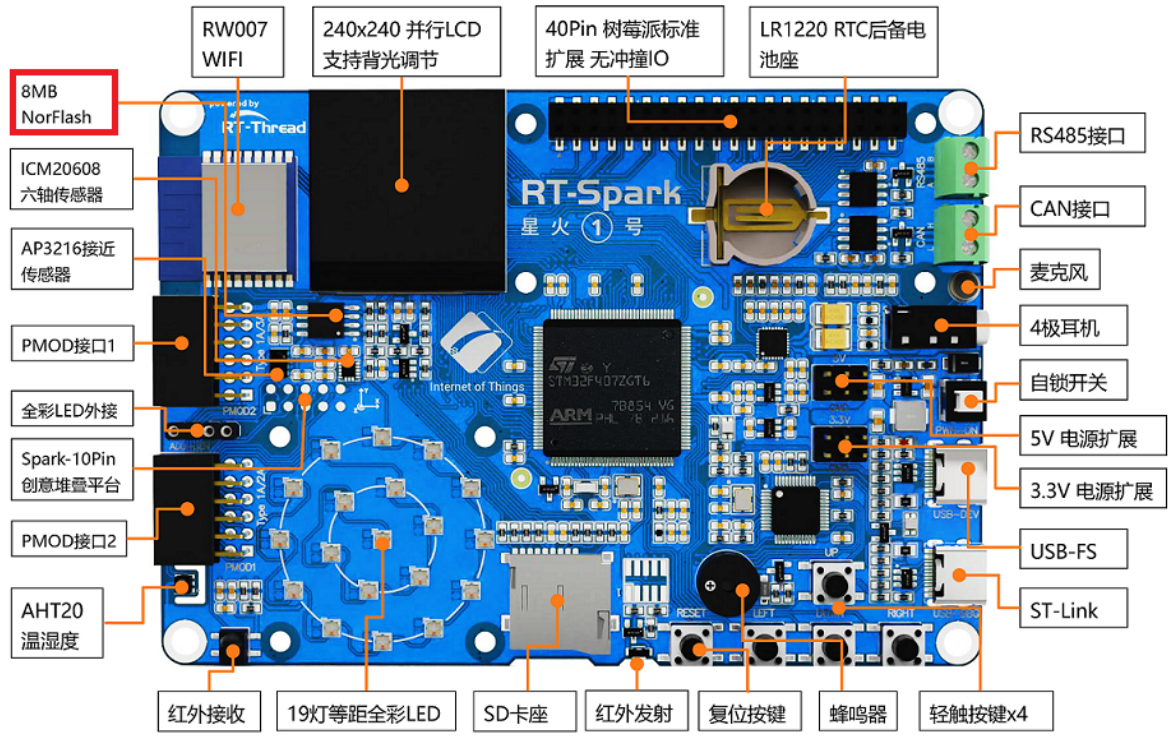

图 19.2: *SPI FLASH 位置*

19.3 软件说明

本例程的源码位于 `/projects/04_component_fs_flash`。

19.3.1 挂载操作代码说明

挂载文件系统的源代码位于 main.c 中。

在示例代码中会执行如下操作：

1. 使用 `fal_blk_device_create()` 函数在 spi flash 中名为 “filesystem” 的分区上创建一个块设备，作为文件系统的存储设备。
2. 使用 `dfs_mount()` 函数将该块设备中的文件系统挂载到根目录下的 `/fal` 目录上。

```
#define FS_PARTITION_NAME  "filesystem"

int main(void)
{
    /* 初始化 fal 功能 */
    fal_init();
```

```
    /* 在 spi flash 中名为 "filesystem" 的分区上创建一个块设备 */
    struct rt_device *flash_dev = fal_blk_device_create(FS_PARTITION_NAME);
    if (flash_dev == NULL)
    {
        LOG_E("Can't create a block device on '%s' partition.", FS_PARTITION_NAME);
    }
    else
    {
        LOG_D("Create a block device on the %s partition of flash successful.",
            FS_PARTITION_NAME);
    }

    /* 挂载 spi flash 中名为 "filesystem" 的分区上的文件系统 */
    if (dfs_mount(flash_dev->parent.name, "/fal", "elm", 0, 0) == 0)
    {
        LOG_I("Filesystem initialized!");
    }
    else
    {
        LOG_E("Failed to initialize filesystem!");
        LOG_D("You should create a filesystem on the block device first!");
    }

    return 0;
}
```

19.4 运行

19.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

19.4.2 运行效果

1. 在 PC 端使用终端工具打开开发板的 ST-Link 提供的虚拟串口，设置 115200 8 1 N 。
2. 按下复位按键重启开发板，如果看到提示”Failed to initialize filesystem!，这是因为指定的挂载设备中还没有创建文件系统。
3. 在 msh 中使用命令 mkfs -t elm filesystem 可以在名为 “filesystem” 的块设备上创建 elm-fat 类型的文件系统。
4. 此时按下复位按键重启开发板，可以看到提示 “FileSystem initialized!”，表明文件系统挂载成功。

打印信息如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun  9 2023 18:14:07
 2006 - 2022 Copyright by RT-Thread team
[I/SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[I/SFUD] W25Q64 flash device is initialize success.
[I/SFUD] Probe SPI flash W25Q64 by SPI device spi20 success.
[D/FAL] (fal_flash_init:49) Flash device |        onchip_flash_128k | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00020000 |initialized finish.
[D/FAL] (fal_flash_init:49) Flash device |                  W25Q64 | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev         |   offset   |    length  |
[I/FAL] -------------------------------------------------------------
[I/FAL] | app        | onchip_flash_128k | 0x00000000 | 0x00060000 |
[I/FAL] | param      | onchip_flash_128k | 0x00060000 | 0x000a0000 |
[I/FAL] | easyflash  | W25Q64            | 0x00000000 | 0x00080000 |
[I/FAL] | download   | W25Q64            | 0x00080000 | 0x00100000 |
[I/FAL] | wifi_image | W25Q64            | 0x00180000 | 0x00080000 |
[I/FAL] | font       | W25Q64            | 0x00200000 | 0x00300000 |
[I/FAL] | filesystem | W25Q64            | 0x00500000 | 0x00300000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL block device (filesystem) created successfully
[D/main] Create a block device on the filesystem partition of flash successful.
    # 在 flash 的文件系统分区上创建块设备成功
[I/main] Filesystem initialized!
    # 文件系统初始化成功
msh />
```

19.4.3 常用功能展示

19.4.4 ls: 查看当前目录信息

```
msh />ls                          # 使用 ls 命令查看文件系统目录信息
Directory /:                      # 可以看到已经存在根目录 /
fal                 <DIR>         # ROMFS 内置的fal目录
```

19.4.5 cd：切换目录到 FAL

```
msh />cd fal                      # 使用 cd 命令切换当前目录
msh /fal>
```

19.4.6 mkdir: 创建文件夹

```
msh /fal>mkdir rt-thread              # 创建 rt-thread 文件夹
msh /fal>ls                           # 查看目录信息如下
Directory /fal:
rt-thread           <DIR>
```

19.4.7 echo: 将输入的字符串输出到指定输出位置

```
msh /fal>echo "hello rt-thread!!!"              # 将字符串输出到标准输出
hello rt-thread!!!
msh /fal>echo "hello rt-thread!!!" hello.txt    # 将字符串出输出到 hello.txt
msh /fal>ls
Directory /fal:
rt-thread           <DIR>
hello.txt           18
msh /fal>

### cat: 查看文件内容
```shell
msh /fal>cat hello.txt                    # 查看 hello.txt 文件的内容并输出
hello rt-thread!!!
```

19.4.8 rm: 删除文件夹或文件

```
msh /fal>ls                               # 查看当前目录信息
Directory /fal:
rt-thread           <DIR>
hello.txt           18
msh /fal>rm rt-thread                     # 删除 rt-thread 文件夹
msh /fal>ls
Directory /fal:
hello.txt           18
msh /fal>rm hello.txt                     # 删除 hello.txt 文件
msh /fal>ls
Directory /:
msh /fal>
```

19.5 注意事项

挂载文件系统之前一定要先在存储设备中创建相应类型的文件系统，否则会挂载失败。
```

19.6 引用参考

- 组件：虚拟文件系统
- 设备与驱动：SPI 设备
- 文档中心：RT-Thread 文档中心

第 20 章

WiFi 管理例程

20.1 简介

本例程使用 RT-Thread Wlan Manager 对 WiFi 网络管理，展示 WiFi 热点扫描，Join 网络，WiFi 自动连接以及 WiFi Event 处理等功能。

20.2 硬件说明

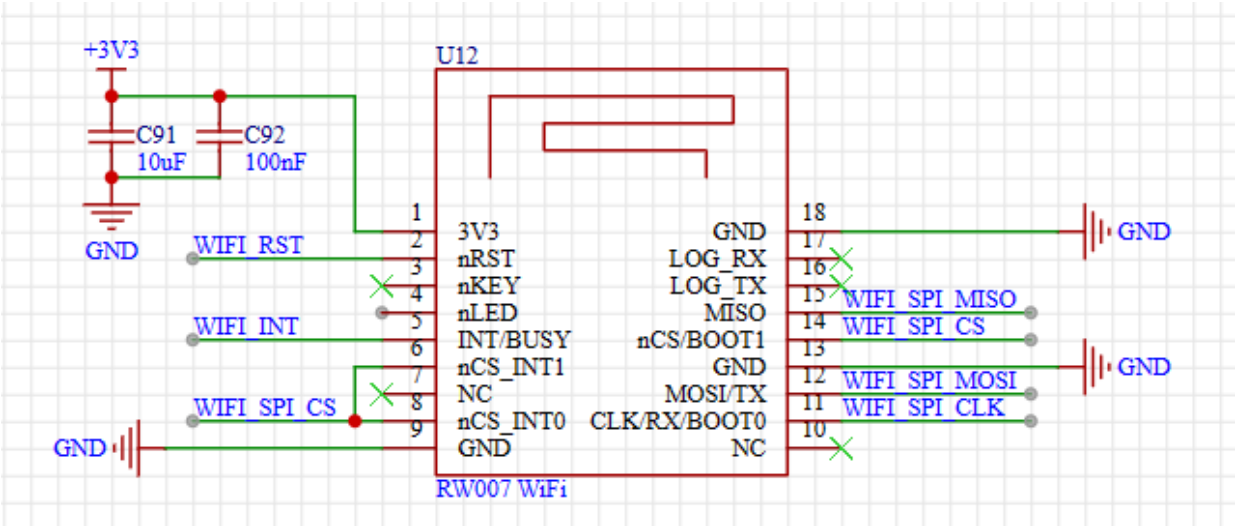

图 20.1: *WIFI 原理图*

RW007 是由上海睿赛德电子科技有限公司开发的高速 WiFi 模块，使用 SPI 与主机通信，支持 IEEE 802.11b/g/n 网络、WEP/WPA/WPA2 加密方式和 STA 和 AP 模式。

主要特性

- 高性能 MCU
- 使用 SPI 通信方式
- SPI 时钟高达 30Mbps

- SPI 模式下有效以太网带宽高达上传 1MBytes/s，下载 1MBytes/s
- 支持 WiFi+BLE 主从机功能
- BLE 主机功能可同时连接多个 BLE 设备
- WiFi 支持 STA+AP 模式
- 支持微信小程序 BLE 快速配网
- 支持断网快速回连
- 内置 Bootloader，支持 OTA 固件升级、安全固件功能
- 支持批量远程升级，方便运维管理

更多相关信息资料见 RW007 介绍页面 睿赛德科技推出高速 Wi-Fi 模块 RW007：内置 RT-Thread 物联网操作系统

20.3 软件说明

本例程的源码位于 `/projects/05_iot_wifi_manager`。

该用例的主要流程如下图所示，首先调用 Scan 接口扫描周围环境中的 AP(Access Point ， 即无线访问热点) ，并打印扫描结果；然后连接一个测试用的 AP (名字: test_ssid , 密码: 12345678)，并等待联网成功，打印网络信息；接着在等待 5 秒之后，断开和 AP 的连接；最后，调用接口初始化自动连接的相关配置，开启自动连接功能。在开启自动连接之后，Wlan Manager 会根据存储介质中的历史记录进行 AP 连接。

注意：请在 main.c 根据实际情况修改 WLAN_SSID 和 WLAN_PASSWORD 两个宏（分别定义了 AP 的名字和密码），否则将无法联网成功。

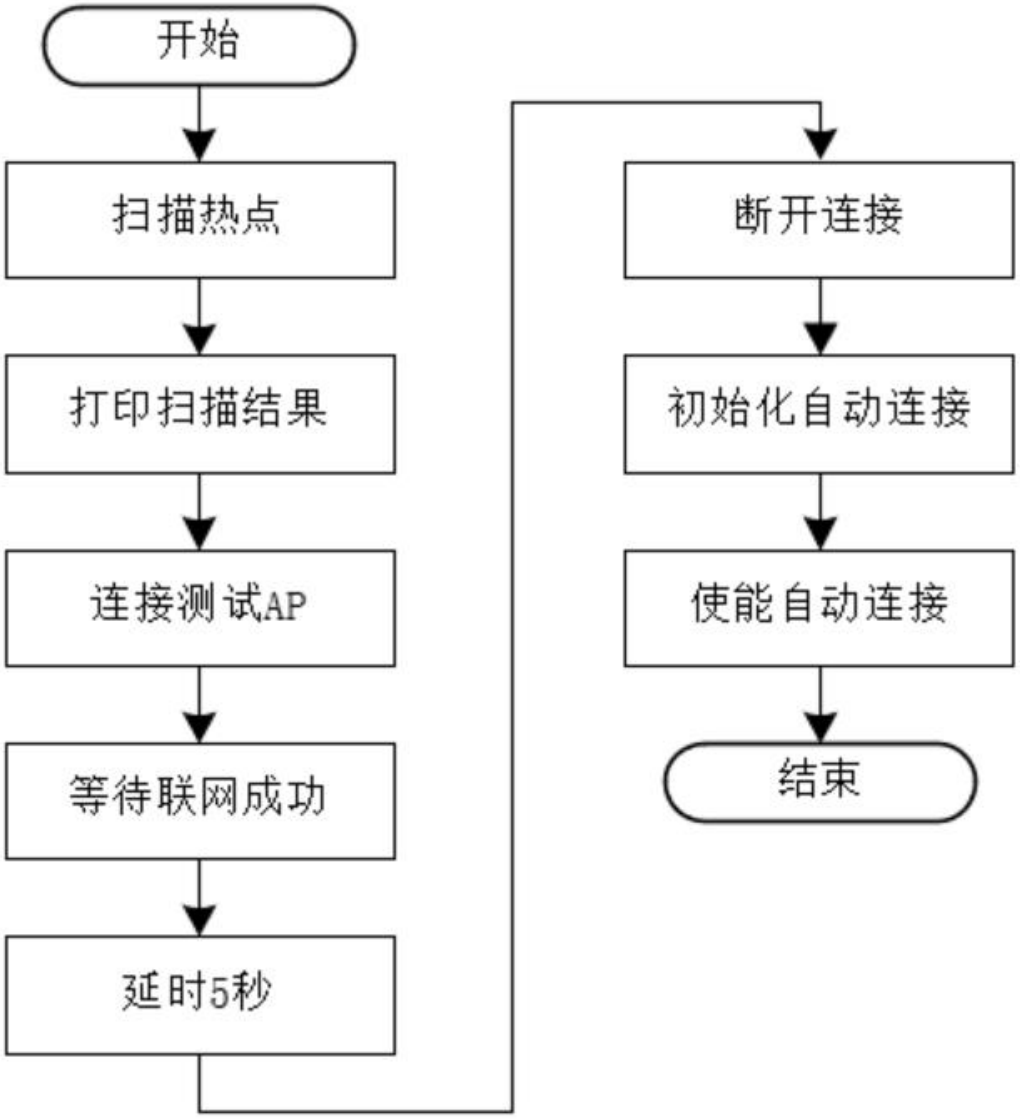

图 20.2: *wlan manager 流程*

20.3.1 热点扫描

初始化信号量，注册回调函数

```
    /* 执行扫描 */
    rt_sem_init(&scan_done,"scan_done",0,RT_IPC_FLAG_FIFO);
    rt_wlan_register_event_handler(RT_WLAN_EVT_SCAN_REPORT, wlan_scan_report_hander
        ,&i);
    rt_wlan_register_event_handler(RT_WLAN_EVT_SCAN_DONE, wlan_scan_done_hander,
        RT_NULL);

    if(rt_wlan_scan() == RT_EOK)
    {
        LOG_D("the scan is started...");
```

```
    }else
    {
        LOG_E("scan failed");
    }
    /* 等待扫描完毕 */
    rt_sem_take(&scan_done,RT_WAITING_FOREVER);
```

扫描完成回调函数和扫描报道的回调函数

```
void wlan_scan_report_hander(int event,struct rt_wlan_buff *buff,void *parameter)
{
    struct rt_wlan_info *info = RT_NULL;
    int index = 0;
    RT_ASSERT(event == RT_WLAN_EVT_SCAN_REPORT);
    RT_ASSERT(buff != RT_NULL);
    RT_ASSERT(parameter != RT_NULL);

    info = (struct rt_wlan_info *)buff->data;
    index = *((int *)(parameter));
    print_wlan_information(info,index);
    ++ *((int *)(parameter));
}

void wlan_scan_done_hander(int event,struct rt_wlan_buff *buff,void *parameter)
{
    RT_ASSERT(event == RT_WLAN_EVT_SCAN_DONE);
    rt_sem_release(&scan_done);
}
```

20.3.2 Join 网络

RT-Thread Wlan Manager 提供极简的接口进行 WiFi 联网操作，仅需要输入 ssid 和 password 即可。另外，Wlan Manager 也提供事件通知机制，RT_WLAN_EVT_READY 事件标志着 WiFi 联网成功，可以使用 Network 进行通信；RT_WLAN_EVT_STA_DISCONNECTED 用于 Network 断开的事件。下面代码片段展示了 WiFi 联网的操作。

```
    /* 热点连接 */
    LOG_D("start to connect ap ...");
    rt_sem_init(&net_ready, "net_ready", 0, RT_IPC_FLAG_FIFO);

    /* 注册 wlan ready 回调函数 */
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY, wlan_ready_handler, RT_NULL);
    /* 注册 wlan 断开回调函数 */
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_DISCONNECTED,
        wlan_station_disconnect_handler, RT_NULL);
    /* 同步连接热点 */
    result = rt_wlan_connect(WLAN_SSID, WLAN_PASSWORD);
    if (result == RT_EOK)
```

```
    {
        rt_memset(&info, 0, sizeof(struct rt_wlan_info));
        /* 获取当前连接热点信息 */
        rt_wlan_get_info(&info);
        LOG_D("station information:");
        print_wlan_information(&info,0);
        /* 等待成功获取 IP */
        result = rt_sem_take(&net_ready, NET_READY_TIME_OUT);
        if (result == RT_EOK)
        {
            LOG_D("networking ready!");
            msh_exec("ifconfig", rt_strlen("ifconfig"));
        }
        else
        {
            LOG_D("wait ip got timeout!");
        }
        /* 回收资源 */
        rt_wlan_unregister_event_handler(RT_WLAN_EVT_READY);
        rt_sem_detach(&net_ready);
    }
    else
    {
        LOG_E("The AP(%s) is connect failed!", WLAN_SSID);
    }
```

此处通过 rt_wlan_register_event_handler 接口注册 RT_WLAN_EVT_READY （网络准备就绪）事件，当 WiFi Join AP 成功，且 IP 分配成功，会触发该事件的回调，标志着可以正常使用网络接口进行通信。

20.3.3 自动连接

打开自动连接功能后，Wlan Manager 会在 WiFi Join 网络成功后，保存该 AP 的信息至存储介质（默认保存最近 3 次的连接信息）。当系统重启或者网络异常断开后，自动读取介质中的信息，进行 WiFi 联网。下面代码片段展示自动连接功能的使用。

```
    /* 自动连接 */
    LOG_D("start to autoconnect ...");
    wifi_autoconnect();
```

20.4 Shell 操作 WiFi

wifi 相关的 shell 命令如下：

```
wifi                                    : 打印帮助
wifi help                               : 查看帮助
wifi join SSID [PASSWORD]               : 连接wifi，SSDI为空，使用配置自动连接
wifi ap   SSID [PASSWORD]               : 建立热点
```

```
wifi scan                                : 扫描全部热点
wifi disc                                : 断开连接
wifi ap_stop                             : 停止热点
wifi status                              : 打印wifi状态 sta + ap
wifi smartconfig                         : 启动配网功能
```

20.4.1 WiFi 扫描

- wifi 扫描命令格式如下

```
wifi scan
```

在调试工具中输入该命令，即可进行 wifi 命令扫描，扫描结果如下所示：

```
[D/main] start to scan ap ...
[D/main] the scan is started...
             SSID                     MAC            security    rssi chn Mbps
------------------------------ -----------------  -------------- ---- --- ----
demo                           5c:de:34:9b:85:08  WPA2_AES_PSK   -35   11 4294
```

20.4.2 WiFi 连接

```
msh />
msh />wifi join test_ssid 12345678
join ssid:test_ssid
[I/WLAN.mgnt] wifi connect success ssid:test_ssid
msh />[I/WLAN.lwip] Got IP address : 192.168.43.6
```

小技巧：如果已经存储 Join 成功的 AP 历史记录，可直接输入 `wifi join` 进行网络连接，忽略 'SSID' 和 'PASSWORD'。

20.4.3 WiFi 断开

```
msh />wifi disc
wifi link down
[I/main] disconnect from the network!
[I/WLAN.mgnt] disconnect success!
```

20.5 运行

20.5.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 `STM32F407-RT-SPARK` 资源包，然后创建新工程，执行编译。

- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

20.5.2 运行效果

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun  9 2023 18:14:15
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
msh >[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

[D/main] start to scan ap ...
[D/main] the scan is started...
             SSID                     MAC            security    rssi chn Mbps
------------------------------ -----------------  -------------- ---- --- ----
test_ssid                      5c:de:34:9b:85:08  WPA2_AES_PSK   -37   11 4294

[D/main] start to connect ap ...
[I/main] disconnect from the network!
[I/WLAN.mgnt] wifi connect success ssid:test_ssid
[D/main] station information:
             SSID                     MAC            security    rssi chn Mbps
------------------------------ -----------------  -------------- ---- --- ----
test_ssid                      00:00:00:00:00:00  UNKNOWN        0     -1    0
[I/WLAN.lwip] Got IP address : 192.168.137.187
[D/main] networking ready!
network interface device: w0 (Default)
MTU: 1500
MAC: fc 58 4a a6 ce cc
FLAGS: UP LINK_UP INTERNET_DOWN DHCP_ENABLE ETHARP BROADCAST IGMP
ip address: 192.168.137.187
gw address: 192.168.137.1
net mask  : 255.255.255.0
dns server #0: 192.168.137.1
dns server #1: 0.0.0.0

network interface device: w1
```

```
MTU: 1500
MAC: fc 58 4a a6 ce cd
FLAGS: UP LINK_DOWN INTERNET_DOWN DHCP_ENABLE ETHARP BROADCAST IGMP
ip address: 0.0.0.0
gw address: 0.0.0.0
net mask  : 0.0.0.0
dns server #0: 192.168.137.1
dns server #1: 0.0.0.0
[D/main] ready to disconect from ap ...
[I/main] disconnect from the network!
[I/WLAN.mgnt] disconnect success!
[D/main] start to autoconnect ...
[I/main] disconnect from the network!
[I/WLAN.mgnt] wifi connect success ssid:test_ssid
wlan_connect_handler
ssid : test_ssid
[I/WLAN.lwip] Got IP address : 192.168.137.187
```

20.6 注意事项

执行例程之前，需先设置一个名字为 ‘test_ssid’，密码为 ‘12345678’ 的 WiFi 热点。

20.7 引用参考

- 文档中心：RT-Thread 文档中心
- 设备与驱动：WLAN 设备
- rw007 软件包：https://github.com/RT-Thread-packages/rw007

第 21 章

MQTT 协议通信例程

21.1 简介

本例程基于 Paho-MQTT 软件包，展示了向服务器订阅主题和向指定主题发布消息的功能。

21.2 硬件说明

本例程需要依赖星火 1 号板卡上的 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。

21.3 软件说明

本例程的源码位于 `/projects/05_iot_mqtt`。

21.3.1 MQTT

MQTT（Message Queuing Telemetry Transport，消息队列遥测传输协议），是一种基于发布 / 订阅（publish/subscribe）模式的 “轻量级” 通讯协议，该协议构建于 TCP/IP 协议上，由 IBM 在 1999 年发布。MQTT 最大优点在于，可以以极少的代码和有限的带宽，为连接远程设备提供实时可靠的消息服务。作为一种低开销、低带宽占用的即时通讯协议，使其在物联网、小型设备、移动应用等方面有较广泛的应用。

MQTT 是一个基于客户端 - 服务器的消息发布 / 订阅传输协议。MQTT 协议是轻量、简单、开放和易于实现的，这些特点使它适用范围非常广泛。在很多情况下，包括受限的环境中，如：机器与机器（M2M）通信和物联网（IoT）。其在，通过卫星链路通信传感器、偶尔拨号的医疗设备、智能家居、及一些小型化设备中已广泛使用。

21.3.2 Paho MQTT 包

Paho MQTT 是 Eclipse 实现的基于 MQTT 协议的客户端，本软件包是在 Eclipse paho-mqtt 源码包的基础上设计的一套 MQTT 客户端程序。

RT-Thread MQTT 客户端功能特点如下：

- 断线自动重连

  RT-Thread MQTT 软件包实现了断线重连机制，在断网或网络不稳定导致连接断开时，会维护登陆状态，重新连接，并自动重新订阅 Topic。提高连接的可靠性，增加了软件包的易用性。

- pipe 模型，非阻塞 API

  降低编程难度，提高代码运行效率，适用于高并发数据量小的情况。

- 事件回调机制

  在建立连接、收到消息或者断开连接等事件时，可以执行自定义的回调函数。

- TLS 加密传输

  MQTT 可以采用 TLS 加密方式传输，保证数据的安全性和完整性。

21.3.3 例程使用说明

本示例的源代码位于 main.c 中。

MQTT 软件包已经实现了 MQTT 客户端的完整功能，开发者只需要设定好 MQTT 客户端的配置即可使用。本例程使用的测试服务器是 Eclipse 的测试服务器，服务器网址、用户名和密码已在 main 文件的开头定义。

在 main 函数中，首先将 MQTT 客户端启动函数（mq_start()）注册为网络连接成功的回调函数，然后执行 WiFi 自动连接初始化。当开发板连接上 WiFi 后，MQTT 客户端函数（mq_start()）会被自动调用。mq_start() 函数主要是配置 MQTT 客户端的连接参数（客户端 ID、保持连接时间、用户名和密码等），设置事件回调函数（连接成功、在线和离线回调函数），设置订阅的主题，并为每个主题设置不同的回调函数去处理发生的事件。设置完成后，函数会启动一个 MQTT 客户端。客户端会自动连接服务器，并订阅相应的主题。

mq_publish() 函数用来向指定的主题发布消息。例程里的主题就是我们 MQTT 客户端启动时订阅的主题，这样，我们会接收到自己发布的消息，实现自发自收的功能。本例程在在线回调函数里调用了 mq_publish() 函数，发布了 Hello,RT-Thread! 消息，所以我们在 MQTT 客户端成功连上服务器，处于在线状态后，会收到 Hello,RT-Thread! 的消息。

本例程的部分示例代码如下所示：

```
/**
 * MQTT URI farmat:
 * domain mode
 * tcp://broker.emqx.io:1883
 *
 * ipv4 mode
 * tcp://192.168.10.1:1883
 * ssl://192.168.10.1:1884
 *
 * ipv6 mode
 * tcp://[fe80::20c:29ff:fe9a:a07e]:1883
 * ssl://[fe80::20c:29ff:fe9a:a07e]:1884
```

```
 */
#define MQTT_URI "tcp://broker.emqx.io:1883"
#define MQTT_SUBTOPIC "/mqtt/test/"
#define MQTT_PUBTOPIC "/mqtt/test/"

/* define MQTT client context */
static MQTTClient client;
static void mq_start(void);
static void mq_publish(const char *send_str);

char sup_pub_topic[48] = {0};

/* 连接成功的回调函数 */
static void wlan_connect_handler(int event, struct rt_wlan_buff *buff, void *
    parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

/* 连接失败的回调函数 */
static void wlan_connect_fail_handler(int event, struct rt_wlan_buff *buff, void *
    parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

int main(void)
{
    /* 注册 wlan 回调函数 */
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY, (void (*)(int, struct
        rt_wlan_buff *, void *))mq_start, RT_NULL);
    /* 初始化自动连接功能 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    rt_wlan_config_autoreconnect(RT_TRUE);
    /*  */
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED, wlan_connect_handler,
        RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED_FAIL,
        wlan_connect_fail_handler, RT_NULL);
}
```

```
static void mqtt_sub_callback(MQTTClient *c, MessageData *msg_data)
{
    *((char *)msg_data->message->payload + msg_data->message->payloadlen) = '\0';
    LOG_D("Topic: %.*s receive a message: %.*s",
          msg_data->topicName->lenstring.len,
          msg_data->topicName->lenstring.data,
          msg_data->message->payloadlen,
          (char *)msg_data->message->payload);

    return;
}

static void mqtt_sub_default_callback(MQTTClient *c, MessageData *msg_data)
{
    *((char *)msg_data->message->payload + msg_data->message->payloadlen) = '\0';
    LOG_D("mqtt sub default callback: %.*s %.*s",
          msg_data->topicName->lenstring.len,
          msg_data->topicName->lenstring.data,
          msg_data->message->payloadlen,
          (char *)msg_data->message->payload);
    return;
}

static void mqtt_connect_callback(MQTTClient *c)
{
    LOG_I("Start to connect mqtt server");
}

static void mqtt_online_callback(MQTTClient *c)
{
    LOG_D("Connect mqtt server success");
    LOG_D("Publish message: Hello,RT-Thread! to topic: %s", sup_pub_topic);
    mq_publish("Hello,RT-Thread!");
}

static void mqtt_offline_callback(MQTTClient *c)
{
    LOG_I("Disconnect from mqtt server");
}

/* 创建与配置 mqtt 客户端 */
static void mq_start(void)
{
    /* 初始 condata 参数 */
    MQTTPacket_connectData condata = MQTTPacket_connectData_initializer;
    static char cid[20] = {0};

    static int is_started = 0;
    if (is_started)
```

```
    {
        return;
    }
    /* 配置 MQTT 文本参数 */
    {
        client.isconnected = 0;
        client.uri = MQTT_URI;

        /* 生成随机客户端 ID */
        rt_snprintf(cid, sizeof(cid), "rtthread%d", rt_tick_get());
        rt_snprintf(sup_pub_topic, sizeof(sup_pub_topic), "%s%s", MQTT_PUBTOPIC, cid
            );
        /* 配置连接参数 */
        memcpy(&client.condata, &condata, sizeof(condata));
        client.condata.clientID.cstring = cid;
        client.condata.keepAliveInterval = 60;
        client.condata.cleansession = 1;
        client.condata.username.cstring = "";
        client.condata.password.cstring = "";

        /* 配置 mqtt 参数 */
        client.condata.willFlag = 0;
        client.condata.will.qos = 1;
        client.condata.will.retained = 0;
        client.condata.will.topicName.cstring = sup_pub_topic;

        client.buf_size = client.readbuf_size = 1024;
        client.buf = malloc(client.buf_size);
        client.readbuf = malloc(client.readbuf_size);
        if (!(client.buf && client.readbuf))
        {
            LOG_E("no memory for MQTT client buffer!");
            goto _exit;
        }

        /* 设置事件回调 */
        client.connect_callback = mqtt_connect_callback;
        client.online_callback = mqtt_online_callback;
        client.offline_callback = mqtt_offline_callback;
        /* 设置要订阅的 topic 和 topic 对应的回调函数 */
        client.messageHandlers[0].topicFilter = sup_pub_topic;
        client.messageHandlers[0].callback = mqtt_sub_callback;
        client.messageHandlers[0].qos = QOS1;

        /* 设置默认订阅回调函数 */
        client.defaultMessageHandler = mqtt_sub_default_callback;
    }

    /* 启动 MQTT 客户端 */
```

```
    LOG_D("Start mqtt client and subscribe topic:%s", sup_pub_topic);
    paho_mqtt_start(&client);
    is_started = 1;

_exit:
    return;
}

/* MQTT 消息发布函数 */
static void mq_publish(const char *send_str)
{
    MQTTMessage message;
    const char *msg_str = send_str;
    const char *topic = sup_pub_topic;
    message.qos = QOS1;
    message.retained = 0;
    message.payload = (void *)msg_str;
    message.payloadlen = strlen(message.payload);

    MQTTPublish(&client, topic, &message);

    return;
}


static void msh_mq_publish(int argc, char *argv[])
{
    char send_buff[100] = {'\0'};
    for(int i=1;i<argc;i++)
    {
        if(i> 1)
        {
            strcat(send_buff," ");
        }
        strcat(send_buff,argv[i]);
    }
    mq_publish(send_buff);  // 发布消息给主题 "test"
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(msh_mq_publish, publish messege by msh);
```

21.4 运行

21.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。

- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

21.4.2 运行效果

按下复位按键重启开发板，开发板会自动连上 WiFi（首次使用需要输入 wifi join wifi 名称 wifi 密码进行连接），可以看到板子会打印出如下信息：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 13 2023 16:40:30
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
msh />[E/[RW007]] The wifi Stage 1 status 67452301 efcdab89 1 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

wifi join test_ssid 12345678
[I/WLAN.mgnt] wifi connect success ssid:test_ssid
wlan_connect_handler
ssid : test_ssid
msh />[I/WLAN.lwip] Got IP address : 192.168.137.187
[D/main] Start mqtt client and subscribe topic:/mqtt/test/rtthread47037
[I/main] Start to connect mqtt server
[D/mqtt] ipv4 address port: 1883
[D/mqtt] HOST = 'broker.emqx.io'
[I/mqtt] MQTT server connect success.
[I/mqtt] Subscribe #0 /mqtt/test/rtthread47037 OK!
[D/main] Connect mqtt server success
[D/main] Publish message: Hello,RT-Thread! to topic: /mqtt/test/rtthread47037
[D/main] Topic: /mqtt/test/rtthread47037 receive a message: Hello,RT-Thread!
```

我们可以看到，WiFi 连接成功后，MQTT 客户端就自动连接了服务器，并订阅了我们指定的主题。连接服务器成功，处于在线状态后，发布了一条 Hello,RT-Thread! 的消息，我们很快接收到了服务器推送过来的这条消息。

21.5 注意事项

使用本例程前需要先连接 WiFi。

21.6 引用参考

- mqtt 软件包：https://github.com/RT-Thread-packages/paho-mqtt
- 文档中心：RT-Thread 文档中心

第 22 章

HTTP Client 功能实现例程

22.1 简介

本例程介绍如何使用 WebClient 软件包发送 HTTP 协议 GET 和 POST 请求，并且接收响应的数据。

22.1.1 HTTP 协议

HTTP （Hypertext Transfer Protocol）协议，即超文本传输协议，是互联网上应用最为广泛的一种网络协议，由于其简捷、快速的使用方式，适用于分布式和合作式超媒体信息系统。HTTP 协议是基于 TCP/IP 协议的网络应用层协议。默认端口为 80 端口。协议最新版本是 HTTP 2.0，目前是用最广泛的是 HTTP 1.1。

HTTP 协议是一种请求 / 响应式的协议。一个客户端与服务器建立连接之后，发送一个请求给服务器。服务器接收到请求之后，通过接收到的信息判断响应方式，并且给予客户端相应的响应，完成整个 HTTP 数据交互流程。

22.1.2 WebClient 软件包

WebClient 软件包是 RT-Thread 自主研发的，基于 HTTP 协议的客户端实现，它提供设备与 HTTP 服务器的通讯的基本功能。

WebClient 软件包功能特点：

- **支持 IPv4/IPv6 地址**

  WebClient 软件包会自动根据传入的 URI 地址的格式判断是 IPv4 地址或 IPv6 地址，并且从中解析出连接服务器需要的信息，提高代码兼容性。

- **支持 GET/POST 请求方法**

  目前 WebClient 软件包支持 HTTP 协议 GET 和 POST 请求方法，这也是嵌入式设备最常用到的两个命令类型，满足设备开发需求。

- **支持文件的上传和下载功能**

  WebClient 软件包提供文件上传和下载的接口函数，方便用户直接通过 GET/POST 请求方法上传本地文件到服务器或者下载服务器文件到本地。

- **支持 HTTPS 加密传输**

  WebClient 软件包可以采用 TLS 加密方式传输数据，保证数据的安全性和完整性。

- **完善的头部数据添加和处理方式**

  WebClient 软件包中提供简单的添加发送请求头部信息的方式，方便用于快速准确的拼接头部信息。

22.2 硬件说明

本例程需要依赖星火 1 号板卡上的 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。

22.3 软件说明

本例程的源码位于 `/projects/05_iot_http_client`。

HTTP Client 例程重要文件说明：

文件路径	说明
applications/main.c	app 入口（WebClient 例程程序）
packages/webclient-v2.2.0	webclient 软件包
packages/webclient-v2.2.0/inc	webclient 软件包头文件
packages/webclient-v2.2.0/src	webclient 软件包源码文件

本例程主要实现设备通过 GET 请求方式从指定服务器中获取数据，之后通过 POST 请求方式上传一段数据到指定服务器，并接收服务器响应数据。例程的源代码位于 applications/main.c 中。

其中 main 函数主要完成 wlan 网络初始化配置，并等待设备联网成功，程序如下所示：

```
int main(void)
{
    int result = RT_EOK;

    result = rt_sem_init(&net_ready, "net_ready", 0, RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        return -RT_ERROR;
    }

    /* 注册 wlan 连接网络成功的回调，wlan 连接网络成功后释放'net_ready' 信号量 */
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY, wlan_ready_handler, RT_NULL);
    /* 注册 wlan 网络断开连接的回调 */
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_DISCONNECTED,
        wlan_station_disconnect_handler, RT_NULL);
```

```
    /* 等待 wlan 连接网络成功 */
    result = rt_sem_take(&net_ready, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        LOG_E("Wait net ready failed!");
        rt_sem_delete(&net_ready);
        return -RT_ERROR;
    }

    /* HTTP GET 请求发送 */
    webclient_get_smpl(HTTP_GET_URL);
    /* HTTP POST 请求发送 */
    webclient_post_smpl(HTTP_POST_URL,post_data,rt_strlen(post_data));
}
```

设备成功接入网络之后，会自动顺序的执行 webclient_get_smpl() 和 webclient_post_smpl() 函数，通过 HTTP 协议发送 GET 和 POST 请求。

1. GET 请求发送, 会根据传入的 URL 发送 get 请求然后返回结果存储在 response 里面

```
int webclient_get_smpl(const char *uri)
{
    char *response = RT_NULL;
    size_t resp_len = 0;
    int index;

    if (webclient_request(uri, RT_NULL, RT_NULL, 0, (void **)&response, &resp_len) <
        0)
    {
        rt_kprintf("webclient send get request failed.");
        return -RT_ERROR;
    }

    rt_kprintf("webclient send get request by simplify request interface.\n");
    rt_kprintf("webclient get response data: \n");
    for (index = 0; index < rt_strlen(response); index++)
    {
        rt_kprintf("%c", response[index]);
    }
    rt_kprintf("\n");

    if (response)
    {
        web_free(response);
    }

    return 0;
}
```

2. POST 请求发送

```
int webclient_post_smpl(const char *uri, const char *post_data, size_t data_len)
{
    char *response = RT_NULL;
    char *header = RT_NULL;
    size_t resp_len = 0;
    int index = 0;

    webclient_request_header_add(&header, "Content-Length: %d\r\n", strlen(post_data
        ));
    webclient_request_header_add(&header, "Content-Type: application/octet-stream\r\
        n");

    if (webclient_request(uri, header, post_data, data_len, (void **)&response, &
        resp_len) < 0)
    {
        rt_kprintf("webclient send post request failed.");
        web_free(header);
        return -RT_ERROR;
    }

    rt_kprintf("webclient send post request by simplify request interface.\n");
    rt_kprintf("webclient post response data: \n");
    for (index = 0; index < resp_len; index++)
    {
        rt_kprintf("%c", response[index]);
    }
    rt_kprintf("\n");

    if (header)
    {
        web_free(header);
    }

    if (response)
    {
        web_free(response);
    }

    return 0;
}
```

本例程中使用 POST 方式请求发送一段数据到 HTTP 服务器 www.rt-thread.com ，HTTP 服务器响应并下发同样的数据到设备上。

例程中调用封装的 `webclient_post_smpl()` 函数发送 POST 请求，该函数内部直接使用 `webclient_request()` 完成整个 POST 请求发送头部信息和获取响应数据的流程，这里客户端发送默认 POST 请求头部信息，响应的数据存储到 response 缓冲区并打印到 FinSH 控制台。

22.4 运行

22.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

按下复位按键重启开发板，程序运行日志如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun  9 2023 15:23:02
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
msh >[E/[RW007]] The wifi Stage 1 status 67452301 efc1ab09 1 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

wifi join test_ssid 12345678
[I/WLAN.mgnt] wifi connect success ssid:test_ssid
msh >[I/WLAN.lwip] Got IP address : 192.168.137.187
webclient send get request by simplify request interface.
webclient get response data:
RT-Thread is an open source IoT operating system from China, which has strong
    scalability: from a tiny kernel running on a tiny core, for example ARM Cortex-M0
    , or Cortex-M3/4/7, to a rich feature system running on MIPS32, ARM Cortex-A8,
    ARM Cortex-A9 DualCore etc.

webclient send post request by simplify request interface.
webclient post response data:
RT-Thread is an open source IoT operating system from China!
```

重启开发板之后，使用 wifi 命令连接 wifi，在连接成功的回调函数里面启动 http 客户端发送 get 请求和 post 请求。并且打印返回的结果。

22.5 注意事项

使用本例程前需要先连接 WiFi。

22.6 引用参考

- webclient 软件包：https://github.com/RT-Thread-packages/webclient
- 文档中心：RT-Thread 文档中心

第 23 章

TLS 安全连接例程

本例程介绍如何使用 mbedTLS 软件包与 HTTPS 服务器建立安全的通讯连接。

23.1 简介

mbedTLS（前身 PolarSSL）是一个由 ARM 公司开源和维护的 SSL/TLS 算法库。其使用 C 编程语言以最小的编码占用空间实现了 SSL/TLS 功能及各种加密算法，易于理解、使用、集成和扩展，方便开发人员轻松地在嵌入式产品中使用 SSL/TLS 功能。该软件包已经被移植到了 RT-Thread 操作系统上，开发者可以方便地在任何使用 RT-Thread OS 的平台上直接使用 mbedTLS 软件包建立 TLS 安全连接。

23.2 硬件说明

mbedTLS 例程需要依赖星火 1 号板卡上的 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。

23.3 软件说明

tls 例程位于 `/projects/05_iot_mbedtls` 目录下，重要文件摘要说明如下所示：

文件	说明
applications/main.c	app 入口
packages/mbedtls-v2.7.10.2/samples/tls_app_test.c	mbedTLS 例程程序
packages/mbedtls-v2.7.10.2/ports	移植文件
packages/mbedtls-v2.7.10.2/mbedtls	mbedTLS 软件包（tls 源码实现）
packages/mbedtls-v2.7.10.2/certs	TLS 证书存放目录

23.3.1 例程使用说明

1. HTTPS 服务器设置

本例程中使用 HTTP **GET** 方法请求 TLS 服务器 www.rt-thread.org 中的 rt-thread.txt 文本文件。文件路径为 /download/rt-thread.txt，默认端口为 443，程序配置如下所示：

```
// https://www.rt-thread.org/download/rt-thread.txt
#define MBEDTLS_WEB_SERVER  "www.rt-thread.org"
#define MBEDTLS_WEB_PORT    "443"

static const char *REQUEST = "GET /download/rt-thread.txt HTTP/1.1\r\n"
    "Host: www.rt-thread.org\r\n"
    "User-Agent: rtthread/3.1 rtt\r\n"
    "\r\n";
```

2. 启动 mbedTLS 客户端

```
int mbedtls_client_start(void)
{
    rt_thread_t tid;

    tid = rt_thread_create("tls_c", mbedtls_client_entry, RT_NULL, 6 * 1024,
        RT_THREAD_PRIORITY_MAX / 3 - 1, 5);
    if (tid)
    {
        rt_thread_startup(tid);
    }

    return RT_EOK;
}
```

通过 mbedtls_client_start API 创建 tls_c 线程，启动 mbedTLS 客户端。

3. 创建 mbedTLS 上下文

```
MbedTLSSession *tls_session = RT_NULL;
tls_session = (MbedTLSSession *) tls_malloc(sizeof(MbedTLSSession));
if (tls_session == RT_NULL)
{
    rt_kprintf("No memory for MbedTLS session object.\n");
    return;
}

tls_session->host = tls_strdup(MBEDTLS_WEB_SERVER);
tls_session->port = tls_strdup(MBEDTLS_WEB_PORT);
tls_session->buffer_len = 1024;
tls_session->buffer = tls_malloc(tls_session->buffer_len);
if (tls_session->buffer == RT_NULL)
{
    rt_kprintf("No memory for MbedTLS buffer\n");
```

```
    tls_free(tls_session);
    return;
}
```

MbedTLSSession 数据结构存在于整个 TLS 连接的生命周期内，存储着 TLS 连接所必要的属性信息，需要用户在进行 TLS 客户端上下文初始化前完成配置。

4. 初始化 mbedTLS 客户端

应用程序使用 mbedtls_client_init 函数初始化 TLS 客户端。

初始化阶段按照 API 参数定义传入相关参数即可，主要用来初始化网络接口、证书、SSL 会话配置等 SSL 交互必须的一些配置，以及设置相关的回调函数。

示例代码如下所示：

```
char *pers = "hello_world"; // 设置随机字符串种子
if((ret = mbedtls_client_init(tls_session, (void *)pers, strlen(pers))) != 0)
{
    rt_kprintf("MbedTLSClientInit err return : -0x%x\n", -ret);
    goto __exit;
}
```

5. 初始化 mbedTLS 客户端上下文

应用程序使用 mbedtls_client_context 函数配置客户端上下文信息，包括证书解析、设置主机名、设置默认 SSL 配置、设置认证模式（默认 MBEDTLS_SSL_VERIFY_OPTIONAL）等。

```
if ((ret = mbedtls_client_context(tls_session)) < 0)
{
    rt_kprintf("MbedTLSCLientContext err return : -0x%x\n", -ret);
    goto __exit;
}
```

6. 建立 SSL/TLS 连接

使用 mbedtls_client_connect 函数为 SSL/TLS 连接建立通道。这里包含整个的握手连接过程，以及证书校验结果。

示例代码如下所示：

```
if((ret = mbedtls_client_connect(tls_session)) != 0)
{
    rt_kprintf("MbedTLSCLientConnect err return : -0x%x\n", -ret);
    goto __exit;
}
```

7. 读写数据

通过前面的操作，TLS 客户端已经与服务器成功建立了 TLS 握手连接，然后就可以通过加密的连接进行 socket 读写。

向 SSL/TLS 中写入数据

示例代码如下所示：

```
static const char *REQUEST = "GET /download/rt-thread.txt HTTP/1.1\r\n"
    "Host: www.rt-thread.org\r\n"
    "User-Agent: rtthread/3.1 rtt\r\n"
    "\r\n";

while((ret = mbedtls_client_write(tls_session, (const unsigned char *)REQUEST,strlen
    (REQUEST))) <= 0)
{
    if(ret != MBEDTLS_ERR_SSL_WANT_READ && ret != MBEDTLS_ERR_SSL_WANT_WRITE)
    {
        rt_kprintf("mbedtls_ssl_write returned -0x%x\n", -ret);
        goto __exit;
    }
}
```

从 SSL/TLS 中读取数据

示例代码如下所示：

```
rt_memset(tls_session->buffer, 0x00, MBEDTLS_READ_BUFFER);
ret = mbedtls_client_read(tls_session, (unsigned char *) tls_session->buffer,
    MBEDTLS_READ_BUFFER);
if (ret == MBEDTLS_ERR_SSL_WANT_READ || ret == MBEDTLS_ERR_SSL_WANT_WRITE
        || ret == MBEDTLS_ERR_SSL_PEER_CLOSE_NOTIFY)
    goto __exit;
if (ret < 0)
{
    rt_kprintf("Mbedtls_ssl_read returned -0x%x\n", -ret);
    goto __exit;
}
if (ret == 0)
{
    rt_kprintf("TCP server connection closed.\n");
    goto __exit;
}
```

注意，如果读写接口返回了一个错误，必须关闭连接。

23.4 运行

23.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。编译完成后，将开发板的 ST-Link USB 口与 PC 机连接，然后将固件下载至开发板。

程序运行日志如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jul 14 2023 11:26:47
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/SFUD] Warning: Read SFDP parameter header information failed. The W25Q64 is not
    support JEDEC SFDP.
[I/SFUD] Warning: This flash device is not found or not support.
[I/SFUD] Error: W25Q64 flash device is initialize fail.
[E/SFUD] ERROR: SPI flash probe failed by SPI device spi20.
[I/sal.skt] Socket Abstraction Layer initialize success.
msh />[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cf0c]
rw007 ver: [RW007_2.1.0-a7a0d089-57]
```

23.4.2 连接无线网络并运行

程序运行后会进行 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（ssid 和 password 分别为设备连接的 WIFI 用户名和密码），使用 tls_test 启动 tls 客户端，如下所示：

```
wifi join demo 12345678
[I/WLAN.mgnt] wifi connect success ssid:demo
msh />[I/WLAN.lwip] Got IP address : 192.168.137.180
msh />tls_test
MbedTLS test sample!
Memory usage before the handshake connection is established:
total    : 65936
used     : 23112
maximum  : 23112
available: 42824
Start handshake tick:21806
msh />Finish handshake tick:24159
MbedTLS connect success...
Memory usage after the handshake connection is established:
total    : 65936
used     : 57380
maximum  : 62164
available: 8556
Writing HTTP request success...
Getting HTTP response...
```

```
HTTP/1.1 200 OK
Server: Tengine
Content-Type: text/plain
Content-Length: 267
Connection: keep-alive
Strict-Transport-Security: max-age=604800
Date: Thu, 06 Jul 2023 06:07:23 GMT
Last-Modified: Sat, 04 Aug 2018 02:14:51 GMT
Vary: Accept-Encoding
ETag: "5b650c1b-10b"
Strict-Transport-Security: max-age=1800; includeSubdomains; preload
Accept-Ranges: bytes
Strict-Transport-Security: max-age=1800; includeSubdomains; preload
Ali-Swift-Global-Savetime: 1688623643
Via: cache14.l2cn1805[46,46,200-0,M], cache50.l2cn1805[47,0], kunlun9.cn1446
    [0,0,200-0,H], kunlun1.cn1446[2,0]
Age: 681632
X-Cache: HIT TCP_HIT dirn:11:212624630
X-Swift-SaveTime: Thu, 06 Jul 2023 06:07:23 GMT
X-Swift-CacheTime: 1296000
Timing-Allow-Origin: *
EagleId: 6f034eb316893052759227449e

RT-Thread is an open source IoT operating system from China, which has strong
    scalability: from a tiny kernel running on a tiny core, for example ARM Cortex-M0
    , or Cortex-M3/4/7, to a rich feature system running on MIPS32, ARM Cortex-A8,
    ARM Co
MbedTLS connection close success.
```

23.5 注意事项

- **如需修改例程 TLS 服务器**

  如果用户修改例程中指定的 HTTPS 测试服务器，请使用 menuconfig 配置证书，详细请参考《mbedTLS 用户手册》。配置新证书后，请使用 scons 命令重新生成工程，scons 会自动将证书文件添加到程序中。

- **如需优化 RAM&ROM 资源占用**

  本例程使用标准的 mbedTLS 配置文件（config.h 文件），该配置文件未对 RAM 和 ROM 资源占用做深度优化，如果用户想要进行资源优化，请参考《mbedTLS 用户手册》。

- **如需修改 TLS 配置文件**

  mbedTLS 默认使用的配置文件为 `mbedtls/config.h`，文件位置 `packages/mbedtls-v2.7.10.2/mbedtls/include/mbedtls/config.h`。

  如果用户不使用 scons 重新生成 MDK 工程，用户可以直接修改该文件进行自定义配置。

  如果用户需要使用 scons 编译或者生成工程，请修改 `mbedtls-v2.7.10.2/ports/inc/tls_config.h` 文件，scons 会自动将该文件内容拷贝到 `mbedtls/config.h` 文件。

23.6 引用参考

- 文档中心：RT-Thread 文档中心

第 24 章

Ymodem 协议固件升级例程

24.1 背景知识

24.1.1 固件升级简述

固件升级，通常称为 OTA（Over the Air）升级或者 FOTA（Firmware Over-The-Air）升级，即固件通过空中下载进行升级的技术。

24.1.2 Ymodem 简述

Ymodem 是一种文本传输协议，在 OTA 应用中为空中下载技术提供文件传输的支持。基于 Ymodem 协议的固件升级即为 OTA 固件升级的一个具体应用实例。

24.1.3 Flash 分区简述

通常嵌入式系统程序是没有文件系统的，而是将 Flash 分成不同的功能区块，从而形成不同的功能分区。

要具备 OTA 固件升级能力，通常需要至少有两个程序运行在设备上。其中负责固件校验升级的程序称之为 bootloader，另一个负责业务逻辑的程序称之为 app。它们负责不同的功能，存储在 Flash 的不同地址范围，从而形成了 bootloader 分区和 app 分区。

但多数情况下嵌入式系统程序是运行在 Flash 中的，下载升级固件的时候不会直接向 app 分区写入新的固件，而是先下载到另外的一个分区暂存，这个分区就是 download 分区，也有称之为 app2 分区，这取决于 bootloader 的升级模式。bootloader 分区、app 分区、download 分区及其他分区一起构成了分区表。分区表标识了该分区的特有属性，通常包含分区名、分区大小、分区的起止地址等。

24.1.4 bootloader 升级模式

bootloader 的升级模式常见有以下两种：

1. bootloader 分区 + app1 分区 + app2 分区模式

该模式下，bootloader 启动后，检查 app1 和 app2 分区，哪个固件版本最新就运行哪个分区的固件。当有新版本的升级固件时，固件下载程序会将新的固件下载到另外的一个没有运行的 app 分区，下次启动的时候重新选择执行新版本的固件。

优点：无需固件搬运，启动速度快。

缺点：app1 分区和 app2 分区通常要大小相等，占用 Flash 资源；且 app1 和 app2 分区都只能存放 app 固件，不能存放其他固件（如 WiFi 固件）。

2. bootloader 分区 + app 分区 + download 分区模式

   该模式下，bootloader 启动后，检查 download 分区是否有新版本的固件，如果 download 分区内有新版本固件，则将新版本固件从 download 分区搬运到 app 分区，完成后执行 app 分区内的固件；如果 download 分区内没有新版本的固件，则直接执行 app 分区内的固件。

   当有新版本的升级固件时，固件下载程序会将新的固件下载到 download 分区内，重启后进行升级。

   优点：download 分区可以比 app 分区小很多（使用压缩固件），节省 Flash 资源，节省下载流量；download 分区也可以下载其他固件，从而升级其他的固件，如 WiFi 固件、RomFs。

   缺点：需要搬运固件，首次升级启动速度略慢。

RT-Thread OTA 使用的是 bootloader 升级模式 2，bootloader 分区 + app 分区 + download 分区的组合。

24.1.5 固件下载器

在嵌入式设备程序中实现的，具有从远端下载升级固件功能的代码块称之为固件下载器（OTADownloader）。固件下载器的工作是把新版本的升级固件下载设备的 Flash 中，不同的协议以及不同的 OTA 固件托管平台会有不同的固件下载器，即不同的代码实现。

目前 RT-Thread 已经支持了多种固件下载器，如下所示：

- 基于 Ymodem 协议的 Ymodem OTA 固件下载器（可通过串口等方式升级固件）
- 基于 HTTP 协议的 HTTP OTA 固件下载器（可通过网络方式升级固件）
- 基于特定云平台的 OTA 固件下载器
  1. ali-iotkit 固件下载器（适配阿里云 LinkDevelop 平台和 LinkPlatform 平台的 OTA 功能）
  2. Ayla 云 OTA 固件下载器（适配 Ayla 云平台的 OTA 功能）
  3. 其他第三方 OTA 托管平台（其他平台请联系 RT-Thread 团队）

24.1.6 RT-Thread OTA 介绍

RT-Thread OTA 是 RT-Thread 开发的跨 OS、跨芯片平台的固件升级技术，轻松实现对设备端固件的管理、升级与维护。

RT-Thread 提供的 OTA 固件升级技术具有以下优势：

- 固件防篡改：自动检测固件签名，保证固件安全可靠
- 固件加密：支持 AES-256 加密算法，提高固件下载、存储安全性

- 固件压缩：高效压缩算法，降低固件大小，减少 Flash 空间占用，节省传输流量，降低下载时间
- 差分升级：根据版本差异生成差分包，进一步节省 Flash 空间，节省传输流量，加快升级速度
- 断电保护：断电后保护，重启后继续升级
- 智能还原：固件损坏时，自动还原至出厂固件，提升可靠性
- 高度可移植：可跨 OS、跨芯片平台、跨 Flash 型号使用
- 多可用的固件下载器：支持多种协议的 OTA 固件托管平台和物联网云平台

RT-Thread OTA 框架图如下所示：

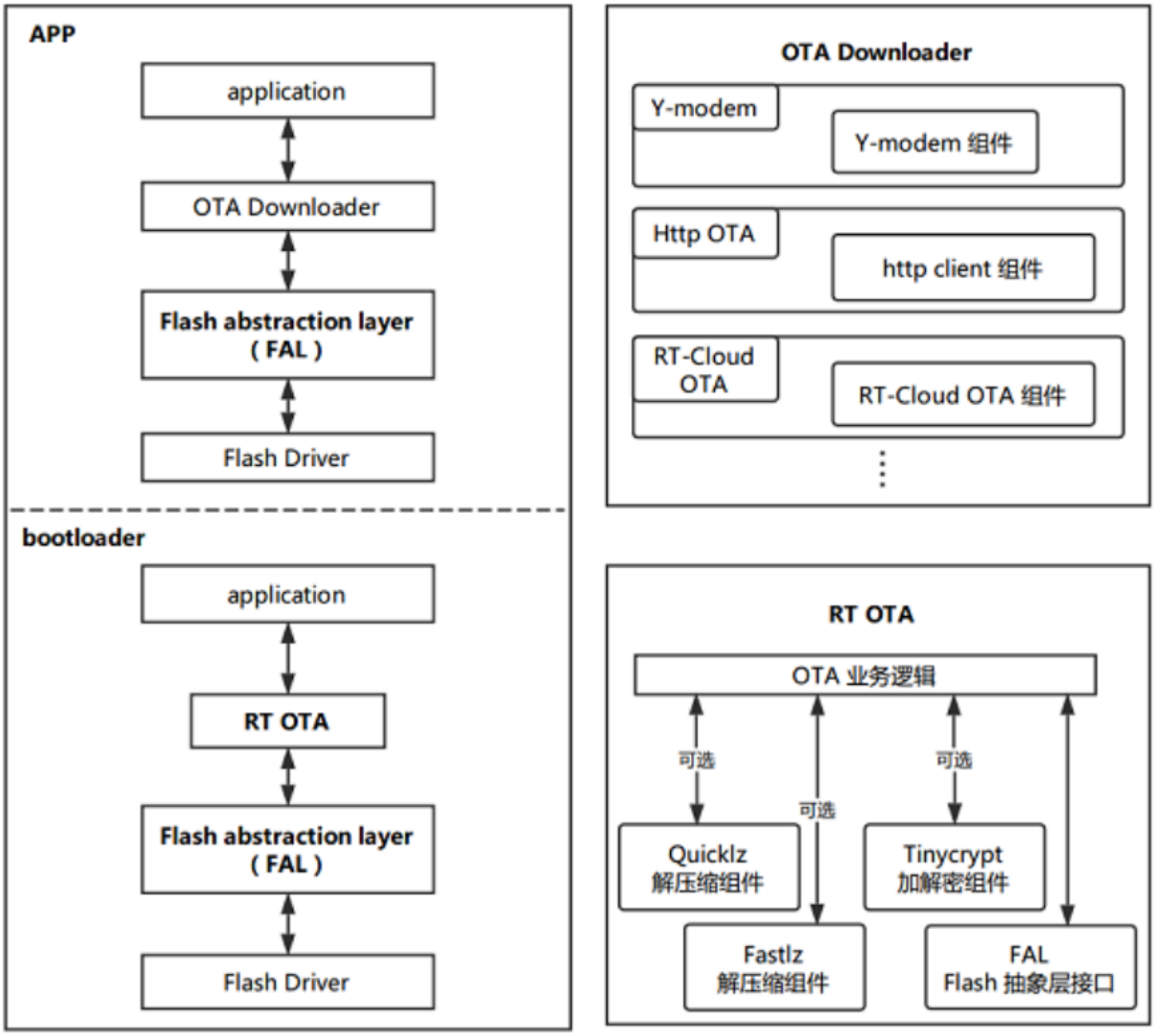

图 24.1: *OTA 框架图*

从上面的 OTA 框架图可以发现，Ymodem 在 OTA 流程中充当的是 OTA Downloader（固件下载器）的角色，核心的 OTA 业务逻辑在 RT OTA 中，也就是封装到了 bootloader 固件里。OTA 业务逻辑与应用程序解耦，极大简化了 OTA 功能增加的难度。

24.1.7 OTA 升级流程

在嵌入式系统方案里，要完成一次 OTA 固件远端升级，通常需要以下阶段：

1. 准备升级固件（RT-Thread OTA 使用特定的 rbl 格式固件），并上传 OTA 固件到固件托管服务器
2. 设备端使用固件托管服务器对应的**固件下载器**下载 OTA 升级固件
3. 新版本固件下载完成后，在适当的时候重启进入 bootloader
4. bootloader 对 OTA 固件进行校验、解密和搬运（搬运到 app 分区）
5. 升级成功，执行新版本 app 固件

OTA 升级流程如下图所示：

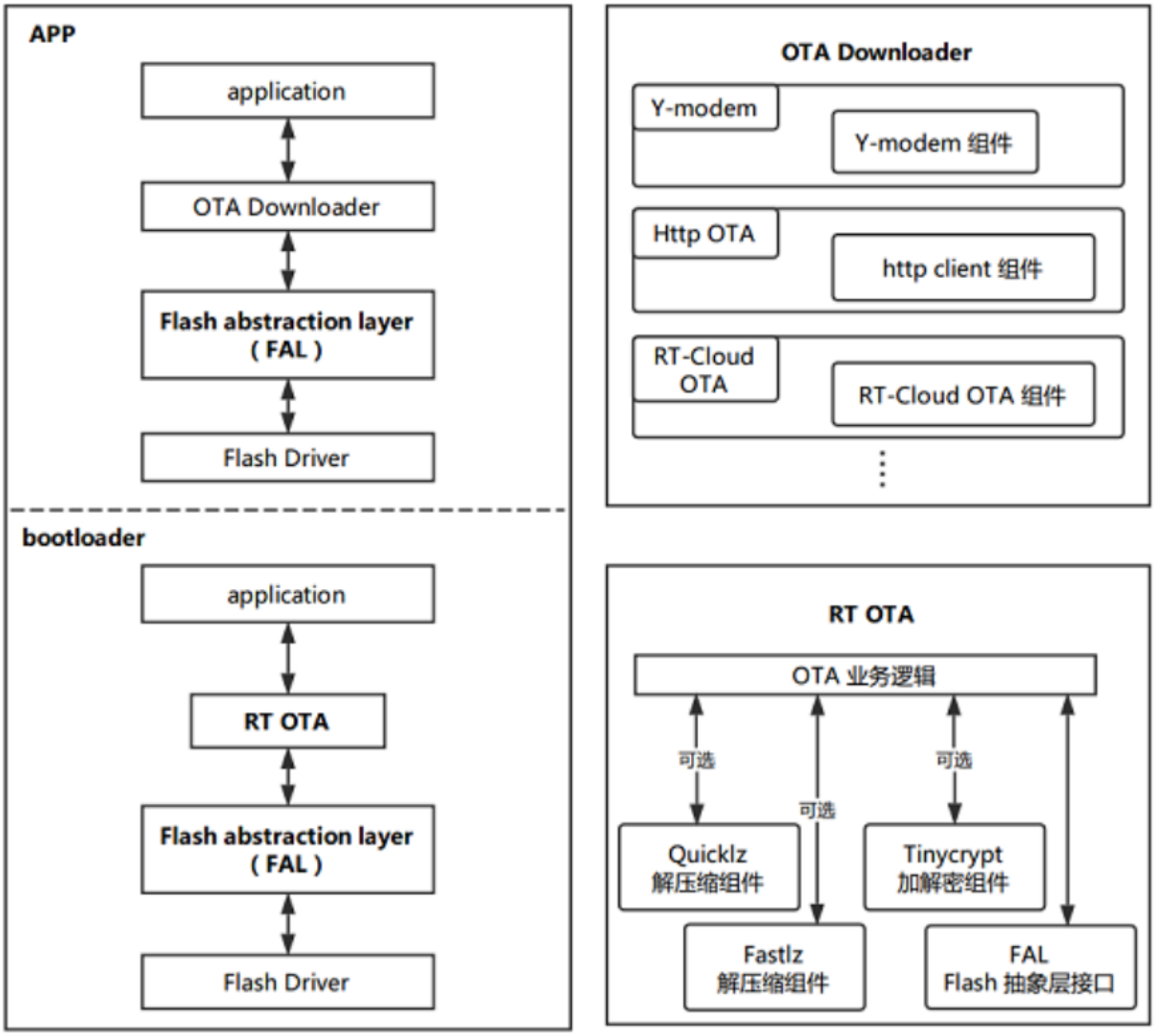

图 24.2: *OTA 升级流程*

24.2 Ymodem OTA 例程说明

Ymodem OTA 升级是 RT-Thread OTA 支持的固件下载器中的一种。在嵌入式设备中通常用于通过串口（UART）进行文件传输及 IAP 在线升级，是常用固件升级方式。

本例程介绍如何使用 RT-Thread Ymodem OTA 固件下载器下载固件，完成 OTA 升级。

在该例程中用到的 bootloader 程序以 bin 文件的形式提供，只能用在该 STM32F4 设备平台，文件位于 bin/bootloader.bin。

24.3 硬件说明

本例程使用到的硬件资源如下所示：

- UART1
- 片内 FLASH (1024KBytes)
- 片外 Nor Flash (8MBytes)

本例程中，bootloader 程序和 app 程序存放在 STM32F4 MCU 的内部 FLASH 中，download 下载区域存放在外部扩展的 Nor FLASH 中。

24.4 分区表

名称	设备	起始地址	大小	结束地址	说明
bootloader	onchip_flash_1	0	128 * 1024	0x20000	引导加载程序分区
app	onchip_flash_128	128 * 1024	384 * 1024	0x80000	应用程序分区
easyflash	W25Q64	0	512 * 1024	0x80000	Easyflash 分区
download	W25Q64	512 * 1024	1024 * 1024	0x180000	下载分区
wifi_image	W25Q64	(512 + 1024) * 1024	512 * 1024	0x180000	WiFi 镜像分区
font	W25Q64	(512 + 1024 + 512) * 1024	3 * 1024 * 1024	0x480000	字体分区
filesystem	W25Q64	(512 + 1024 + 512 + 3 * 1024) * 1024	3 * 1024 * 1024	0x780000	文件系统分区

bootloader 程序的分区表定义在 bootloader 程序中，如果需要修改 bootloader 的分区表，则需要修改 ootloader 程序。目前不支持用户自定义 bootloader，如果有商用需求，请联系 RT-Thread 获取支持。

24.5 软件说明

本例程的源码位于 `/projects/05_iot_ota_ymodem`。

文件路径	说明
application/main.c	app 程序入口
bin/bootloader.bin	需要烧录到 0x08000000 地址的二进制文件完成

文件路径	说明
bin/v1.0.0.bin	第一个版本的应用程序，代码段被链接到了 0x08020000 处
bin/v2.0.0.bin	第二个版本的应用程序，代码段被链接到了 0x08020000 处
libraries/Board_Drivers/fal	Flash 抽象层组件（fal）的移植文件
rt-thread/components/fal	FAL 组件
packages/ota_downloader	包含 ymodem 和 http 下载功能的 otadownloader 的软件包

Ymodem 固件升级流程如下所示：

1. Ymodem 串口终端使用 ymodem 协议发送升级固件
2. APP 使用 Ymodem 协议下载固件到 download 分区
3. bootloader 对 OTA 升级固件进行校验、解密和搬运（搬运到 app 分区）
4. 程序从 bootloader 跳转到 app 分区执行新的固件

24.6 运行

24.6.1 bootloader.bin 烧录

bootloader.bin 的烧录比较方便直接直接将 bootloader.bin 拖入 RT-SPARK 驱动器中即可完成烧录。

图 24.3: *RT-SPARK 驱动器*

24.6.2 app 版本

此例程默认状态下为 app 版本 1 的状态，可以将此流程直接编译下载。

运行结果如下：

```
 __   ___      __   __   __  ___
|__)  |  __ |__) /  \ /  \  |
|  \  |     |__) \__/ \__/  |
2006 - 2019 Copyright by rt-thread team
               0.9.2 build May 23 2023
[D/FAL] (fal_flash_init:63) Flash device |    onchip_flash_16k_part | addr: 0
    x08000000 | len: 0x00020000 | blk_size: 0x00004000 |initialized finish.
[D/FAL] (fal_flash_init:63) Flash device |             onchip_flash | addr: 0
    x08020000 | len: 0x000e0000 | blk_size: 0x00020000 |initialized finish.
[D/FAL] (fal_flash_init:63) Flash device |                nor_flash | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name     | flash_dev    |   offset   |    length  |
[I/FAL] -------------------------------------------------------------
[I/FAL] | app      | onchip_flash | 0x00000000 | 0x00080000 |
[I/FAL] | download | nor_flash    | 0x00080000 | 0x00080000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.4.0) initialize success.
[I/FAL] System initialization successful.
```

```
[I]RT-Thread OTA package(V0.2.1) initialize success.
[E]Get firmware header occur CRC32(calc.crc: 7b93c5c8 != hdr.info_crc32: ffffffff)
    error on 'download' partition!
[E]Get OTA download partition firmware header failed!
[E]Verify firmware hash(calc.hash: a31f0d2c != hdr.hash: 0730fc68) failed on
    partition 'app'.
[I]Begin to execute the program on app partition.
[I/FAL] Find user firmware at app partition 0x08020000 successfully.
[I/FAL] Bootloader jumps to user firmware now.

 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 14 2023 10:55:02
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[I/SFUD] W25Q64 flash device is initialize success.
[I/SFUD] Probe SPI flash W25Q64 by SPI device spi20 success.
msh />[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

[D/main] The current version of APP firmware is 1.0.0
[D/FAL] (fal_flash_init:49) Flash device |        onchip_flash_128k | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00020000 |initialized finish.
[D/FAL] (fal_flash_init:49) Flash device |                  W25Q64 | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev         |   offset   |    length  |
[I/FAL] -------------------------------------------------------------
[I/FAL] | bootloader | onchip_flash_128k | 0x00000000 | 0x00020000 |
[I/FAL] | app        | onchip_flash_128k | 0x00020000 | 0x00060000 |
[I/FAL] | easyflash  | W25Q64            | 0x00000000 | 0x00080000 |
[I/FAL] | download   | W25Q64            | 0x00080000 | 0x00100000 |
[I/FAL] | wifi_image | W25Q64            | 0x00180000 | 0x00080000 |
[I/FAL] | font       | W25Q64            | 0x00200000 | 0x00300000 |
[I/FAL] | filesystem | W25Q64            | 0x00500000 | 0x00300000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL block device (filesystem) created successfully
[D/main] Create a block device on the filesystem partition of flash successful.
[I/main] Filesystem initialized!
```

可以看出首先程序会进入 reboot 中，然后加载 app 分区的程序执行。并且 app 版本号为 1.0.0

24.6.3 升级 app 版本

将 application/main.c 中的代码升级为以下内容：

```
// #define APP_VERSION "1.0.0"
#define APP_VERSION "2.0.0"
```

编译得到的二进制文件即为升级后的程序。

首先将程序运行在 ota 升级模式使用如下命令：

```
msh />ymodem_ota
Default save firmware on download partition.
Warning: Ymodem has started! This operator will not recovery.
Please select the ota firmware file and use Ymodem to send.
CCCCCCCCCC
```

之后将打包好的固件使用 xsheel 的 ymodem 功能发送给应用程序。

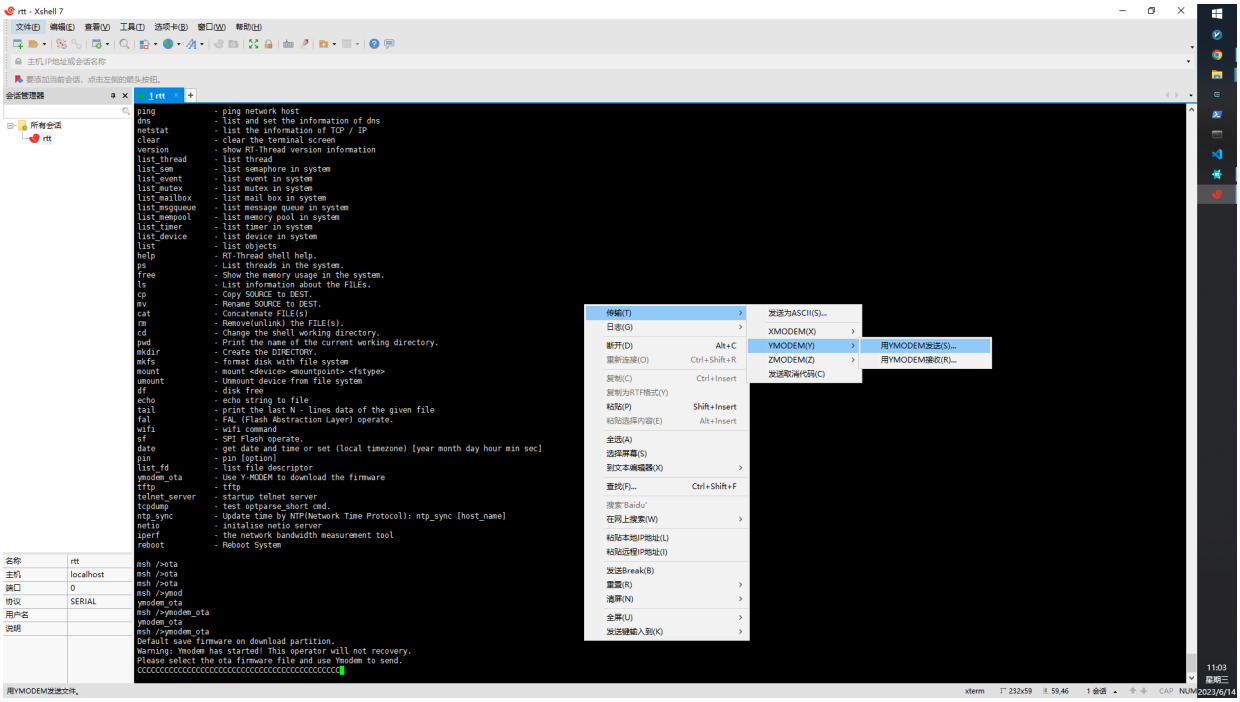

图 24.4: *Xshell ymodem 传输*

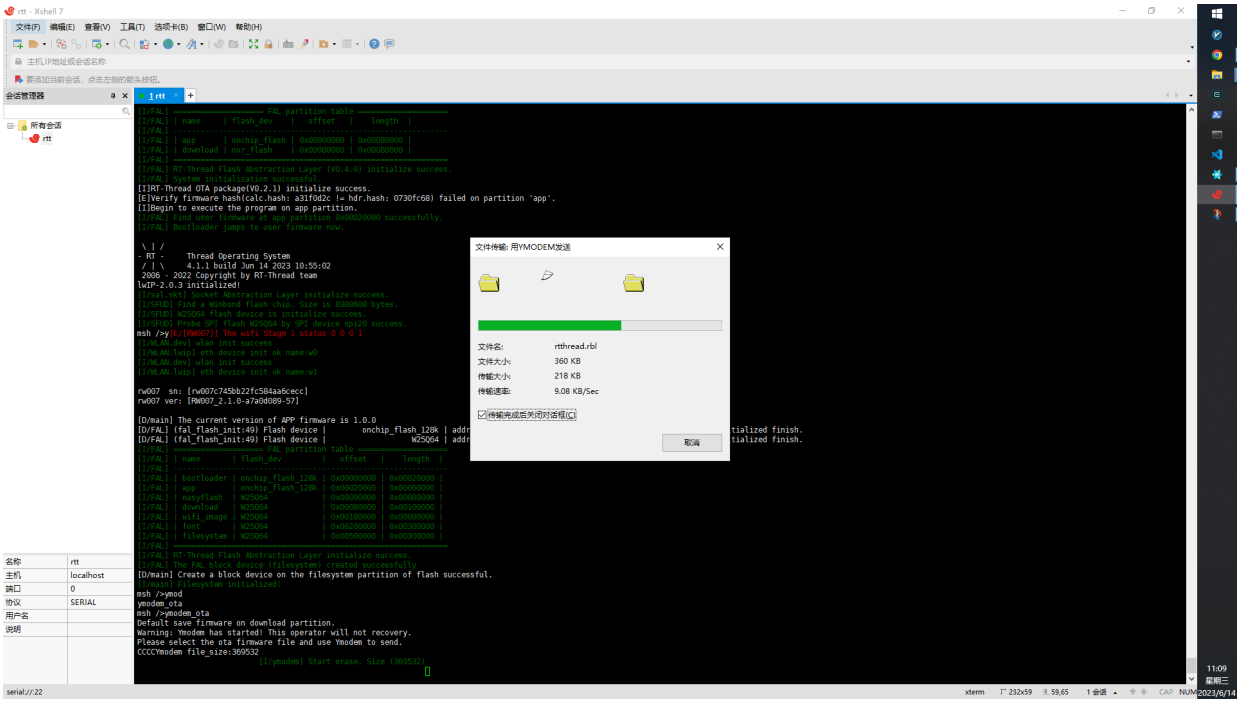

图 24.5: *正在上传*

上传成功之后重启进入 bootloader，bootloader 会自动将 download 分区下的 app 复制到 app 分区然后跳转运行

```
 __   ____      __    __    __  ____
|__)  |   __ |__) /  \ /  \   |
|  \  |      |__) \__/ \__/   |
2006 - 2019 Copyright by rt-thread team
                0.9.2 build May 23 2023
[D/FAL] (fal_flash_init:63) Flash device |    onchip_flash_16k_part | addr: 0
    x08000000 | len: 0x00020000 | blk_size: 0x00004000 |initialized finish.
[D/FAL] (fal_flash_init:63) Flash device |             onchip_flash | addr: 0
    x08020000 | len: 0x000e0000 | blk_size: 0x00020000 |initialized finish.
[D/FAL] (fal_flash_init:63) Flash device |                nor_flash | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name     | flash_dev    |   offset   |    length  |
[I/FAL] -------------------------------------------------------------
[I/FAL] | app      | onchip_flash | 0x00000000 | 0x00080000 |
[I/FAL] | download | nor_flash    | 0x00080000 | 0x00080000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.4.0) initialize success.
[I/FAL] System initialization successful.
[I]RT-Thread OTA package(V0.2.1) initialize success.
[I]Verify 'download' partition(fw ver: 2.0.0, timestamp: 1686712307) success.
[I]OTA firmware(app) upgrade(1.0.0->2.0.0) startup.
[I]The partition 'app' is erasing.
[I]The partition 'app' erase success.
[I]OTA Write:
```

```
    [================================================================================================================================
     100%
[I]Verify 'app' partition(fw ver: 2.0.0, timestamp: 1686712307) success.
[I/FAL] Find user firmware at app partition 0x08020000 successfully.
[I/FAL] Bootloader jumps to user firmware now.

 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 14 2023 11:11:39
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[I/SFUD] W25Q64 flash device is initialize success.
[I/SFUD] Probe SPI flash W25Q64 by SPI device spi20 success.
msh />[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

[D/main] The current version of APP firmware is 2.0.0
[D/FAL] (fal_flash_init:49) Flash device |        onchip_flash_128k | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00020000 |initialized finish.
[D/FAL] (fal_flash_init:49) Flash device |                  W25Q64 | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev         |   offset   |    length  |
[I/FAL] -------------------------------------------------------------
[I/FAL] | bootloader | onchip_flash_128k | 0x00000000 | 0x00020000 |
[I/FAL] | app        | onchip_flash_128k | 0x00020000 | 0x00060000 |
[I/FAL] | easyflash  | W25Q64            | 0x00000000 | 0x00080000 |
[I/FAL] | download   | W25Q64            | 0x00080000 | 0x00100000 |
[I/FAL] | wifi_image | W25Q64            | 0x00180000 | 0x00080000 |
[I/FAL] | font       | W25Q64            | 0x00200000 | 0x00300000 |
[I/FAL] | filesystem | W25Q64            | 0x00500000 | 0x00300000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL block device (filesystem) created successfully
[D/main] Create a block device on the filesystem partition of flash successful.
[I/main] Filesystem initialized!
```

可见程序已经从 1.0.0 升级至 2.0.0

24.6.4 制作升级固件

以此例程为例子，制作用于 Ymodem 升级演示所用到的 app 固件。

制作工具已经放在 `/tools/ota_packager` 文件夹里里面。制作的目的是将 bin 文件打包成 rbl 文件。

RT-Thread OTA 固件打包器如下图所示：

图 24.6: *RT-Thread OTA 固件打包器*

用户可以根据需要，选择是否对固件进行加密和压缩，提供多种压缩算法和加密算法支持，基本操作步骤如下：

- 选择待打包的固件（projects/05_iot_ota_ymodem/Debug/rtthread.bin）
- 选择生成固件的位置
- 选择压缩算法（不压缩则留空）
- 选择加密算法（不加密则留空）
- 配置加密密钥（不加密则留空）
- 配置加密 IV （不加密则留空）
- 填写固件名称（对应分区名称，这里为 app）
- 填写固件版本（填写 application/main.c 中的版本号 2.0.0）

- 开始打包

通过以上步骤制作完成的 `rt-thread.rbl` 文件即可用于后续的升级文件。

Note：

- **加密密钥**和 **加密 IV** 必须与 bootloader 程序中的一致，否则无法正确加解密固件
  默认提供的 bootloader.bin 选择不压缩不加密即可。
- 固件打包过程中有 固件名称 的填写，这里注意需要填入 Flash 分区表中对应分区的名称，不能有误
  如果要升级 **app** 程序，则填写 `app`；如果升级 **WiFi 固件**，则填写 `wifi_image`。
- 使用 **OTA 打包工具**制作升级固件 `rt-thread.rbl`
  正确填写固件名称为 **app**，版本号填写 `main.c` 中定义的版本号 `2.0.0`。

24.7 注意事项

- 在运行该例程前，请务必先将 bootloader.bin 固件烧录到设备
- 必须使用.rbl 格式的升级固件
- 打包 OTA 升级固件时，分区名字必须与分区表中的名字相同（升级 app 固件对应 app 分区），参考分区表章节
- 串口终端工具需要支持 Ymodem 协议，并使用确认使用 Ymodem 协议发送固件
- 串口波特率 115200，无奇偶校验，无流控
- app 固件必须从 0x08020000 地址开始链接，否则应用 bootloader 会跳转到 app 失败

app 固件存储在 app 分区内，起始地址为 0x08020000，如果用户需要升级其他 app 程序，请确保编译器从 0x08020000 地址链接 app 固件。

RT-Thread Studio 工程设置如下:

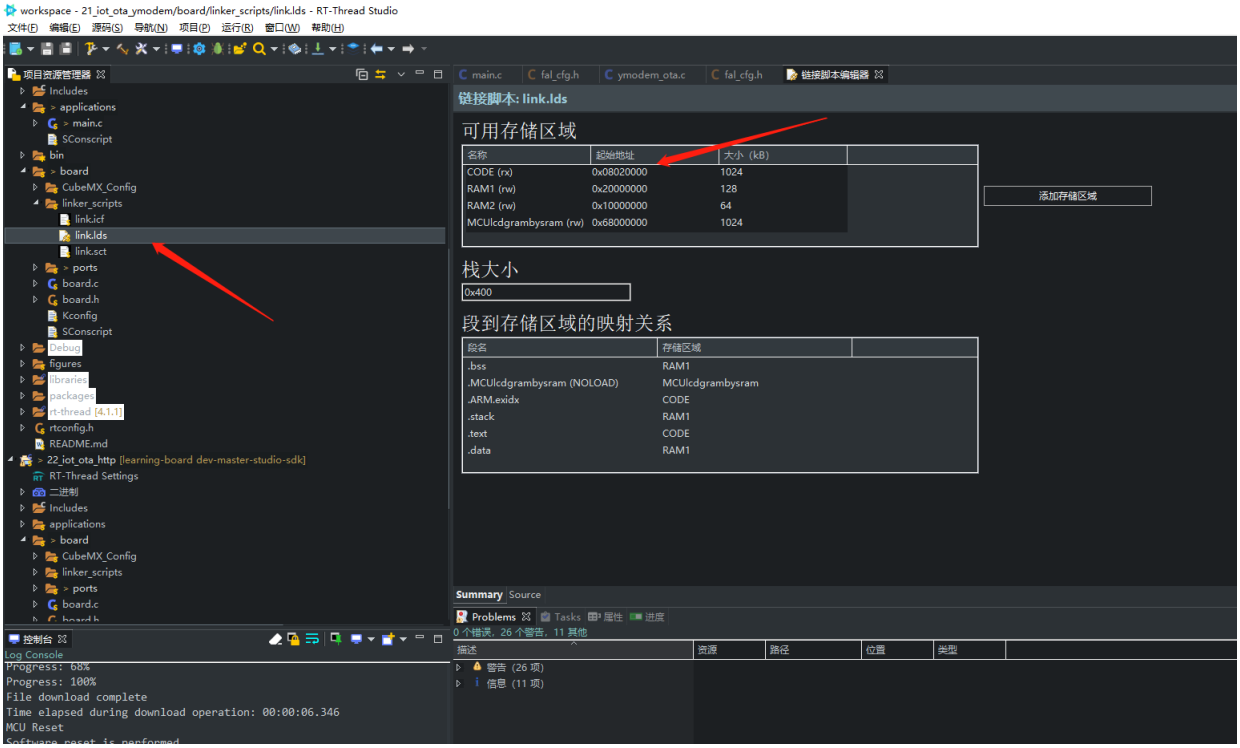

图 24.7: *设置代码段链接地址*

- app 应用重新设置中断向量（使用 bootloader 的时候需要）使用 bootloader 的时候，app 固件从 0x08020000 地址开始链接，因此需要将中断向量重新设置到 0x08020000 地址，程序如下所示：

```
static int ota_app_vtor_reconfig(void)
{
    #define NVIC_VTOR_MASK 0x3FFFFF80
    #define RT_APP_PART_ADDR 0x08020000
    /* 根据应用设置向量表 */
    SCB->VTOR = RT_APP_PART_ADDR & NVIC_VTOR_MASK;

    return 0;
}
INIT_BOARD_EXPORT(ota_app_vtor_reconfig);
```

24.8 引用参考

- 文档中心：RT-Thread 文档中心
- ota downloader 软件包：https://github.com/RT-Thread-packages/ota_downloader

第 25 章

HTTP 协议固件升级例程

25.1 例程说明

HTTP 是一种超文本传输协议，采用请求应答的通讯模式，可以基于通用 socket 实现极简客户端程序，广泛应用在互联网上。HTTP 请求应答的方式，及其很小的客户端代码量，也使其很方便地应用在物联网设备中，用于与服务器进行数据交互，以及 OTA 固件下载。

本例程基于 HTTP 客户端实现 HTTP OTA 固件下载器，通过 HTTP 协议从 HTTP 服务器下载升级固件到设备。HTTP 客户端代码参考 RT-Thread WebClient 软件包。

在该例程中用到的 bootloader 程序以 bin 文件的形式提供，文件位于 bin/bootloader.bin。

25.2 背景知识

参考 Ymodem 固件升级例程。

25.3 硬件说明

本例程使用到的硬件资源如下所示：

- UART1
- 片内 FLASH (1024KBytes)
- 片外 Nor Flash (8MBytes)

本例程中，bootloader 程序和 app 程序存放在 STM32F4 MCU 的内部 FLASH 中，download 下载区域存放在外部扩展的 Nor FLASH 中。

25.4 分区表

名称	设备	起始地址	大小	结束地址	说明
bootloader	onchip_flash_1	0	128 * 1024	0x20000	引导加载程序分区
app	onchip_flash_128	128 * 1024	384 * 1024	0x80000	应用程序分区
easyflash	W25Q64	0	512 * 1024	0x80000	Easyflash 分区
download	W25Q64	512 * 1024	1024 * 1024	0x180000	下载分区
wifi_image	W25Q64	(512 + 1024) * 1024	512 * 1024	0x180000	WiFi 镜像分区
font	W25Q64	(512 + 1024 + 512) * 1024	3 * 1024 * 1024	0x480000	字体分区
filesystem	W25Q64	(512 + 1024 + 512 + 3 * 1024) * 1024	3 * 1024 * 1024	0x780000	文件系统分区

bootloader 程序的分区表定义在 bootloader 程序中，如果需要修改 bootloader 的分区表，则需要修改 bootloader 程序。目前不支持用户自定义 bootloader，如果有商用需求，请联系 RT-Thread 获取支持。

25.5 软件说明

本例程的源码位于 `/projects/05_iot_ota_http`。

文件路径	说明
application/main.c	app 程序入口
bin/bootloader.bin	需要烧录到 0x08000000 地址的二进制文件完成
bin/v1.0.0.bin	第一个版本的应用程序，代码段被链接到了 0x08020000 处
bin/v2.0.0.bin	第二个版本的应用程序，代码段被链接到了 0x08020000 处
libraries/Board_Drivers/fal	Flash 抽象层组件（fal）的移植文件
rt-thread/components/fal	FAL 组件
packages/ota_downloader	包含 ymodem 和 http 下载功能的 otadownloader 的软件包

HTTP 固件升级流程如下所示：

1. 打开 tools/MyWebServer 软件，并配置本机 IP 地址和端口号，选择存放升级固件的目录
2. 在 MSH 中使用 http_ota 命令下载固件到 download 分区

3. bootloader 对 OTA 升级固件进行校验、解密和搬运（搬运到 app 分区）
4. 程序从 bootloader 跳转到 app 分区执行新的固件

25.5.1 程序说明

HTTP OTA 固件下载器程序在 packages/ota_downloader/src 目录下：

```
void http_ota(uint8_t argc, char **argv)
{
    if (argc < 2)
    {
        rt_kprintf("using uri:" HTTP_OTA_URL "\n");
        http_ota_fw_download(HTTP_OTA_URL);
    }
    else
    {
        http_ota_fw_download(argv[1]);
    }
}
/**
 * msh />http_ota [url]
*/
MSH_CMD_EXPORT(http_ota, Use HTTP to download the firmware);
```

```
static int http_ota_fw_download(const char* uri)
{
    int ret = RT_EOK;
    struct webclient_session* session = RT_NULL;

    /* create webclient session and set header response size */
    session = webclient_session_create(GET_HEADER_BUFSZ);
    if (!session)
    {
        LOG_E("open uri failed.");
        ret = -RT_ERROR;
        goto __exit;
    }

    /* get the real data length */
    webclient_shard_head_function(session, uri, &file_size);

    if (file_size == 0)
    {
        LOG_E("Request file size is 0!");
        ret = -RT_ERROR;
        goto __exit;
    }
    else if (file_size < 0)
    {
```

```
        LOG_E("webclient GET request type is chunked.");
        ret = -RT_ERROR;
        goto __exit;
    }
    LOG_I("OTA file size is (%d)", file_size);
    LOG_I("\033[1A");

    /* Get download partition information and erase download partition data */
    if ((dl_part = fal_partition_find("download")) == RT_NULL)
    {
        LOG_E("Firmware download failed! Partition (%s) find error!", "download");
        ret = -RT_ERROR;
        goto __exit;
    }

    LOG_I("Start erase flash (%s) partition!", dl_part->name);

    if (fal_partition_erase(dl_part, 0, file_size) < 0)
    {
        LOG_E("Firmware download failed! Partition (%s) erase error!", dl_part->name
            );
        ret = -RT_ERROR;
        goto __exit;
    }
    LOG_I("Erase flash (%s) partition success!", dl_part->name);

    /* register the handle function, you can handle data in the function */
    webclient_register_shard_position_function(session,
        http_ota_shard_download_handle);

    /* the "memory size" that you can provide in the project and uri */
    ret = webclient_shard_position_function(session, uri, begin_offset, file_size,
        HTTP_OTA_BUFF_LEN);

    /* clear the handle function */
    webclient_register_shard_position_function(session, RT_NULL);

    if (ret == RT_EOK)
    {
        if (session != RT_NULL)
        {
            webclient_close(session);
            session = RT_NULL;
        }

        LOG_I("\033[0B");
        LOG_I("Download firmware to flash success.");
        LOG_I("System now will restart...");
```

```
        rt_thread_delay(rt_tick_from_millisecond(5));

        /* Reset the device, Start new firmware */
        extern void rt_hw_cpu_reset(void);
        rt_hw_cpu_reset();
    }
    else
    {
        LOG_E("Download firmware failed.");
    }

__exit:
    if (session != RT_NULL)
        webclient_close(session);
    begin_offset = 0;

    return ret;
}
```

使用 MSH_CMD_EXPORT 命令将 http_ota 命令导入 msh，使用此命令即可调用 http 下载功能。

25.6 运行

25.6.1 烧录旧版本程序

首先将 bootloader.bin 和 v1.0.0.bin 程序烧录进单片机，烧录过程参考 Ymodem 。

25.6.2 烧录新版本程序

1. 打包新版本程序, 参考 Ymodem 固件升级例程。
2. 启动 HTTP OTA 升级
    1. 解压 /tools/MyWebServer.zip 到当前目录（解压后有 /tools/MyWebServer 目录）
    2. 打开 /tools/MyWebServer 目录下的 MyWebServer.exe 软件配置 MyWebServer 软件，选择 OTA 固件（rbl 文件）的路径，设置本机 IP 和端口号，并启动服务器，如下图所示：

3. 连接开发板串口，复位开发板，进入 MSH 命令行
4. 在设备的命令行里输入 http_ota http://192.168.1.10:80/rtthread.rbl 命令启动 HTTP OTA 升级根据您的 MyWebServer 软件的 IP 和端口号配置修改 http_ota 命令。
5. 设备升级过程

输入命令后，会擦除 download 分区，下载升级固件。下载过程中会打印下载进度条。

```
http_ota http://192.168.181.150:80/rtthread.rbl
[I/http_ota] OTA file size is (379976)
[I/http_ota] Start erase flash (download) partition!
[I/http_ota] Erase flash (download) partition success!
[I/http_ota] Download:
    [=========================================================================================================
     100%
[I/http_ota] Download firmware to flash success.
[I/http_ota] System now will restart...
```

设备重启后，bootloader 会对升级固件进行合法性和完整性校验，验证成功后将升级固件从 download 分区搬运到目标分区（这里是 app 分区）。

升级成功后设备状态如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 14 2023 13:30:35
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[I/SFUD] W25Q64 flash device is initialize success.
[I/SFUD] Probe SPI flash W25Q64 by SPI device spi20 success.
msh />[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

[D/main] The current version of APP firmware is 2.0.0
[D/FAL] (fal_flash_init:49) Flash device |        onchip_flash_128k | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00020000 |initialized finish.
[D/FAL] (fal_flash_init:49) Flash device |                  W25Q64 | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev         |   offset    |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | bootloader | onchip_flash_128k | 0x00000000 | 0x00020000 |
[I/FAL] | app        | onchip_flash_128k | 0x00020000 | 0x00060000 |
```

```
[I/FAL] | easyflash  | W25Q64            | 0x00000000 | 0x00080000 |
[I/FAL] | download   | W25Q64            | 0x00080000 | 0x00100000 |
[I/FAL] | wifi_image | W25Q64            | 0x00180000 | 0x00080000 |
[I/FAL] | font       | W25Q64            | 0x00200000 | 0x00300000 |
[I/FAL] | filesystem | W25Q64            | 0x00500000 | 0x00300000 |
[I/FAL] ==========================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL block device (filesystem) created successfully
[D/main] Create a block device on the filesystem partition of flash successful.
[I/main] Filesystem initialized!
```

设备升级完成后会自动运行新的固件，从上图中的日志上可以看到，app 固件已经从 1.0.0 版本升级到了 2.0.0 版本。

2.0.0 版本的固件同样是支持 HTTP OTA 下载功能的，因此可以一直使用 HTTP 进行 OTA 升级。用户如何需要增加自己的业务代码，可以基于该例程进行修改。

25.7 注意事项

在运行该例程前，请务必先将 all.bin 固件烧录到设备

- 必须使用.rbl 格式的升级固件
- 打包 OTA 升级固件时，分区名字必须与分区表中的名字相同（升级 app 固件对应 app 分区），参考分区表章节
- MyWebServer 软件可能会被您的防火墙限制功能，使用前请检查 Windows 防火墙配置
- 串口波特率 115200，无奇偶校验，无流控
- app 固件必须从 0x08020000 地址开始链接，否则应用 bootloader 会跳转到 app 失败 app 固件存储在 app 分区内，起始地址为 0x08020000，如果用户需要升级其他 app 程序，请确保编译器从 0x08020000 地址链接 app 固件。

25.8 引用参考

- 文档中心：RT-Thread 文档中心
- OTA 说明请参考 **Ymodem 固件升级**章节
- WiFi 使用说明请参考 **使用 WiFi Manager 管理、操作 WiFi 网络**章节

第 26 章

网络小工具集使用例程

26.1 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。

本例程展示如何在星火 1 号开发板上使用 netutils 软件包的各种功能。

26.2 硬件说明

本例程需要依赖星火 1 号板卡上的 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。

26.3 软件说明

本例程的源码位于 `/projects/05_iot_netutils`。

26.3.1 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

名称	分类	功能简介
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
```

26.4 运行

26.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 `STM32F407-RT-SPARK` 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

26.4.2 运行效果

26.4.2.1 准备工作

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。如果在指定的存储器分区上没有可挂载文件系统，可能会出现文件系统挂载失败的情况。此时需要在 msh 中执行 `mkfs -t elm filesystem` 命令，该命令会在存储设备中名为 “filesystem” 的分区上创建 elm 类型的文件系统。

文件系统正常初始化提示信息如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 14 2023 15:04:23
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
```

```
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[I/SFUD] W25Q64 flash device is initialize success.
[I/SFUD] Probe SPI flash W25Q64 by SPI device spi20 success.
msh />[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

[D/FAL] (fal_flash_init:49) Flash device |        onchip_flash_128k | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00020000 |initialized finish.
[D/FAL] (fal_flash_init:49) Flash device |                  W25Q64 | addr: 0
    x00000000 | len: 0x00800000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev         |   offset   |    length  |
[I/FAL] -------------------------------------------------------------
[I/FAL] | app        | onchip_flash_128k | 0x00000000 | 0x00060000 |
[I/FAL] | param      | onchip_flash_128k | 0x00060000 | 0x000a0000 |
[I/FAL] | easyflash  | W25Q64            | 0x00000000 | 0x00080000 |
[I/FAL] | download   | W25Q64            | 0x00080000 | 0x00100000 |
[I/FAL] | wifi_image | W25Q64            | 0x00180000 | 0x00080000 |
[I/FAL] | font       | W25Q64            | 0x00200000 | 0x00300000 |
[I/FAL] | filesystem | W25Q64            | 0x00500000 | 0x00300000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer initialize success.
[I/FAL] The FAL block device (filesystem) created successfully
[D/main] Create a block device on the filesystem partition of flash successful.
[I/main] Filesystem initialized!
```

26.4.2.2 连接无线网络

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 key 配置网络，如下所示：

```
msh />wifi join ssid_test router_key_xxx
join ssid:ssid_test
[I/WLAN.mgnt] wifi connect success ssid:ssid_test
msh />[I/WLAN.lwip] Got IP address : 152.10.200.224
```

26.4.2.3 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 支持访问 IP 地址或域名，使用 Finsh/MSH 命令进行测试，大致使用效果如下：

- Ping 域名

```
msh />ping rt-thread.org
60 bytes from 116.62.244.242 icmp_seq=0 ttl=49 time=11 ticks
60 bytes from 116.62.244.242 icmp_seq=1 ttl=49 time=10 ticks
60 bytes from 116.62.244.242 icmp_seq=2 ttl=49 time=12 ticks
60 bytes from 116.62.244.242 icmp_seq=3 ttl=49 time=10 ticks
msh />
```

- Ping IP

```
msh />ping 192.168.10.12
60 bytes from 192.168.10.12 icmp_seq=0 ttl=64 time=5 ticks
60 bytes from 192.168.10.12 icmp_seq=1 ttl=64 time=1 ticks
60 bytes from 192.168.10.12 icmp_seq=2 ttl=64 time=2 ticks
60 bytes from 192.168.10.12 icmp_seq=3 ttl=64 time=3 ticks
msh />
```

26.4.2.4 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。

开启 RTC 设备后，可以使用下面的命令同步 NTP 的本地时间至 RTC 设备。

MSH 命令效果如下：

```
msh />ntp_sync   # 同步 NTP 网络时间到 RTC 设备
Get local time from NTP server: Fri Sep 21 09:39:15 2023
The system time is updated. Timezone is 8.
msh />date       # 打印当前时间
Fri Sep 21 09:39:47 2023
```

26.4.2.5 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

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

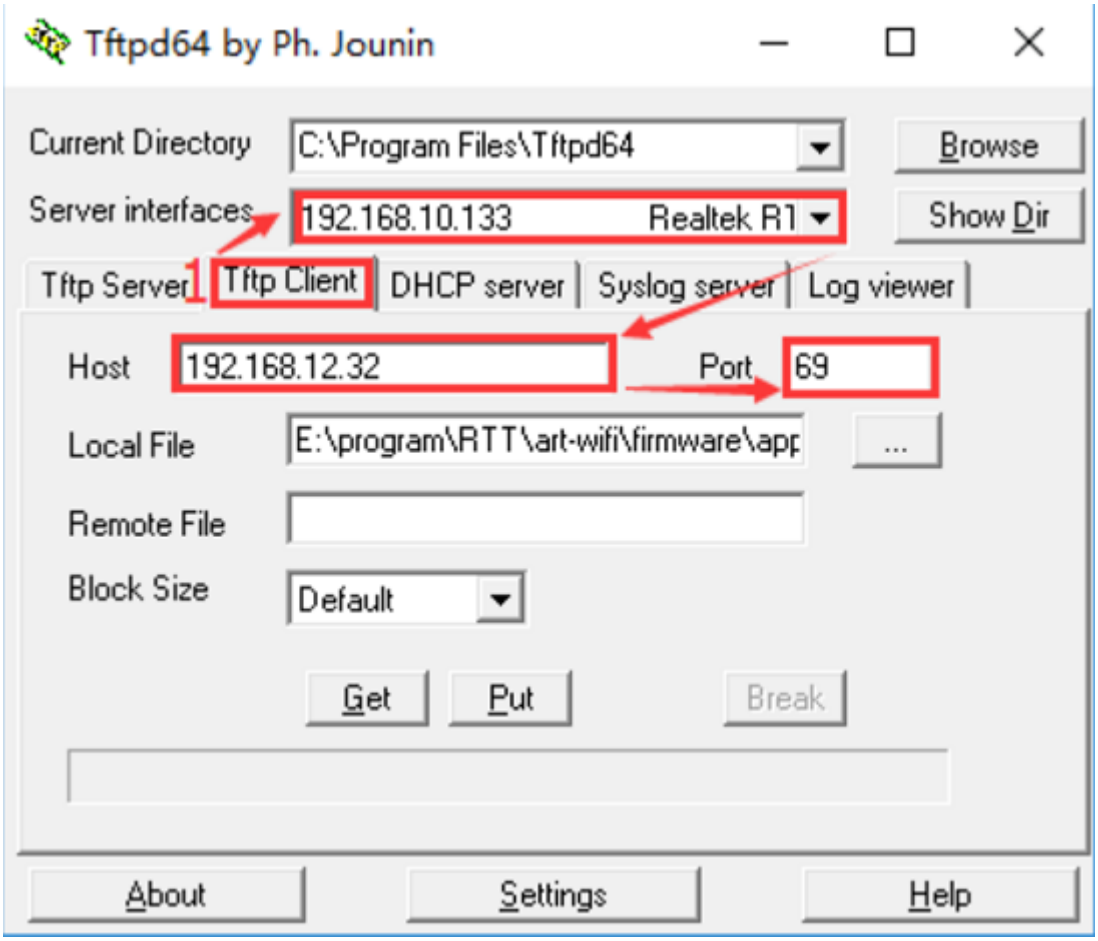

图 26.1: *tftpd config*

向 RT-Thread 发送文件

1. 在 Tftpd64 软件中，选择好要发送文件；
2. 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；

3. 点击 Put 按钮即可。

如下图所示，将文件发送至 Finsh/MSH 当前进入的目录下，这里使用的是相对路径：

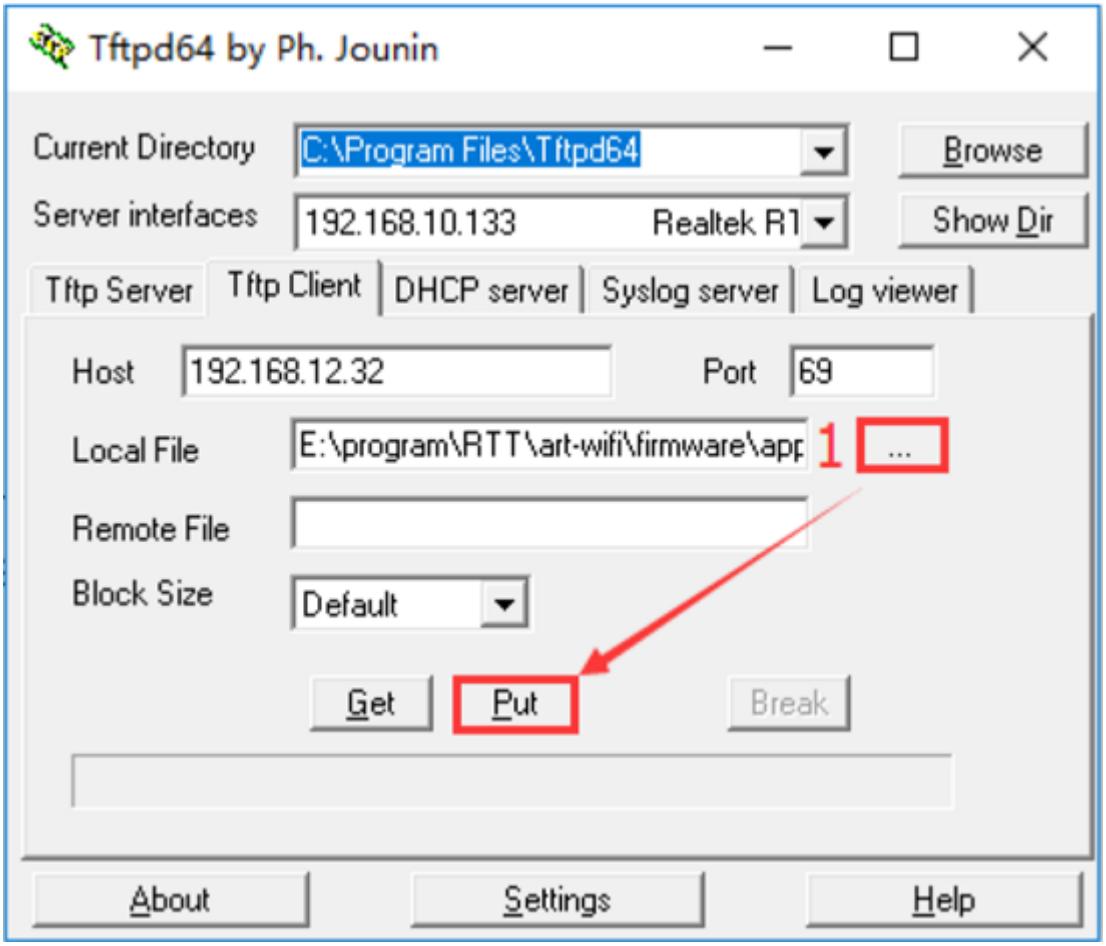

图 26.2: *tftpd put*

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。

从 RT-Thread 获取文件

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 /web_root/image.jpg 保存到本地，这里使用的是绝对路径：

```
msh /web_root>ls                # 查看文件是否存在
Directory /web_root:
image.jpg 10559
msh /web_root>
```

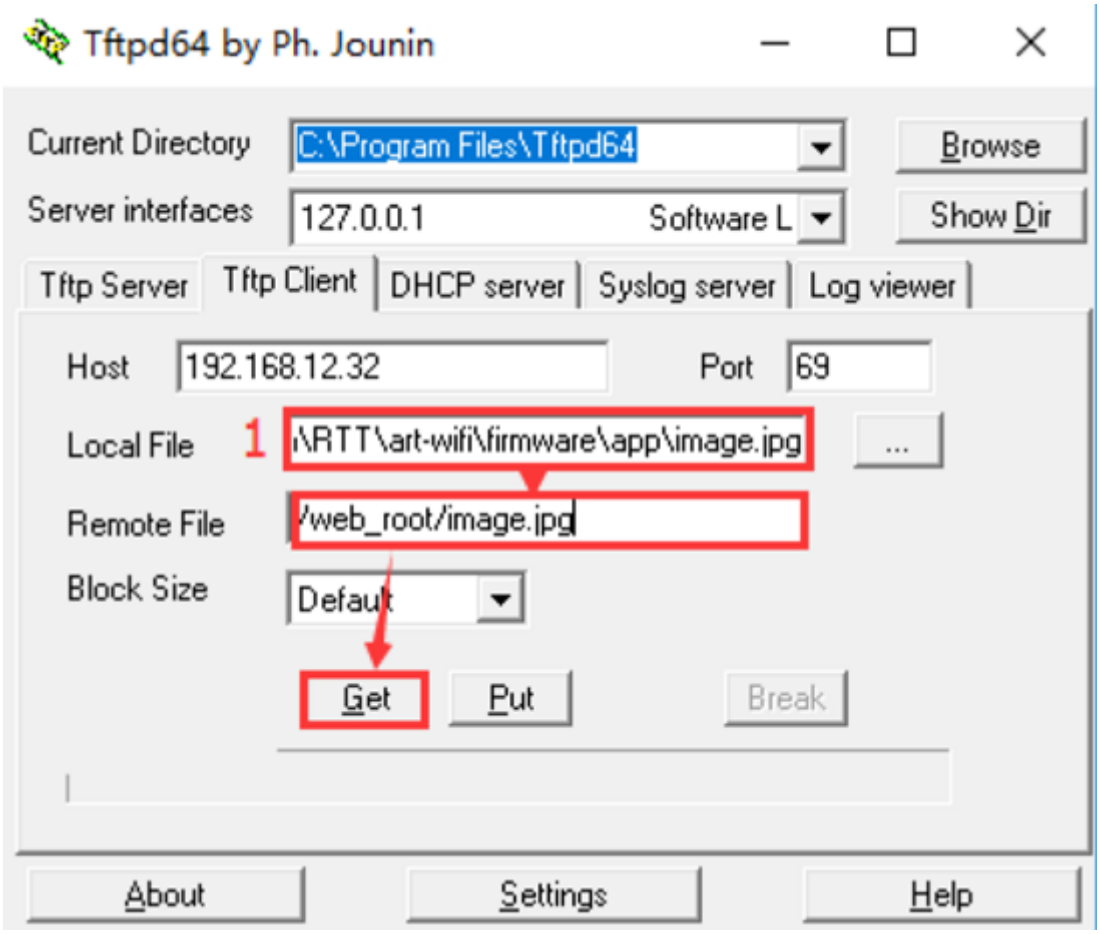

图 26.3: *tftpd get*

26.4.2.6 Iperf 工具

Iperf 是一个网络性能测试工具。Iperf 可以测试最大 TCP 和 UDP 带宽性能，具有多种参数和 UDP 特性，可以根据需要调整，可以报告带宽、延迟抖动和数据包丢失。Iperf 使用的是主从式架构，即一端是服务器，另一端是客户端，在软件包中 Iperf 实现了 TCP 服务器模式和客户端模式，暂不支持 UDP 测试，下面将具体讲解这两种模式的使用方法。

Iperf 服务器模式

1. 获取 IP 地址

   在 RT-Thread 上获取 IP 地址需要执行如下命令：

```
msh />ifconfig
network interface: e0 (Default)
MTU: 1500
MAC: 00 04 9f 05 44 e5
FLAGS: UP LINK_UP ETHARP
ip address: 192.168.12.71
gw address: 192.168.10.1
net mask : 255.255.0.0
dns server #0: 192.168.10.1
```

```
dns server #1: 223.5.5.5
```

```
记下获得的 IP 地址 192.168.12.71（按实际情况记录）
```

2. 启动 Iperf 服务器

   在 RT-Thread 上启动 Iperf 服务器需要执行如下命令：

```
msh />iperf -s -p 5001
```

```
参数 -s 的意思是表示作为服务器启动，参数 -p 表示监听 5001 端口。
```

3. 安装 JPerf 测试软件

   安装文件位于 packages/netutils/tools/jperf.rar ，这个是绿色软件，安装实际上是解压的过程，解压到新文件夹即可。

4. 进行 jperf 测试

   打开 jperf.bat 软件，按如下操作进行配置：

   1. 选择 Client 模式；
   2. 输入刚刚获得的 IP 地址 192.168.12.71（按实际地址填写），修改端口号为 5001；
   3. 点击 run Lperf! 开始测试；
   4. 等待测试结束。测试时，测试数据会在 shell 界面和 JPerf 软件上显示。

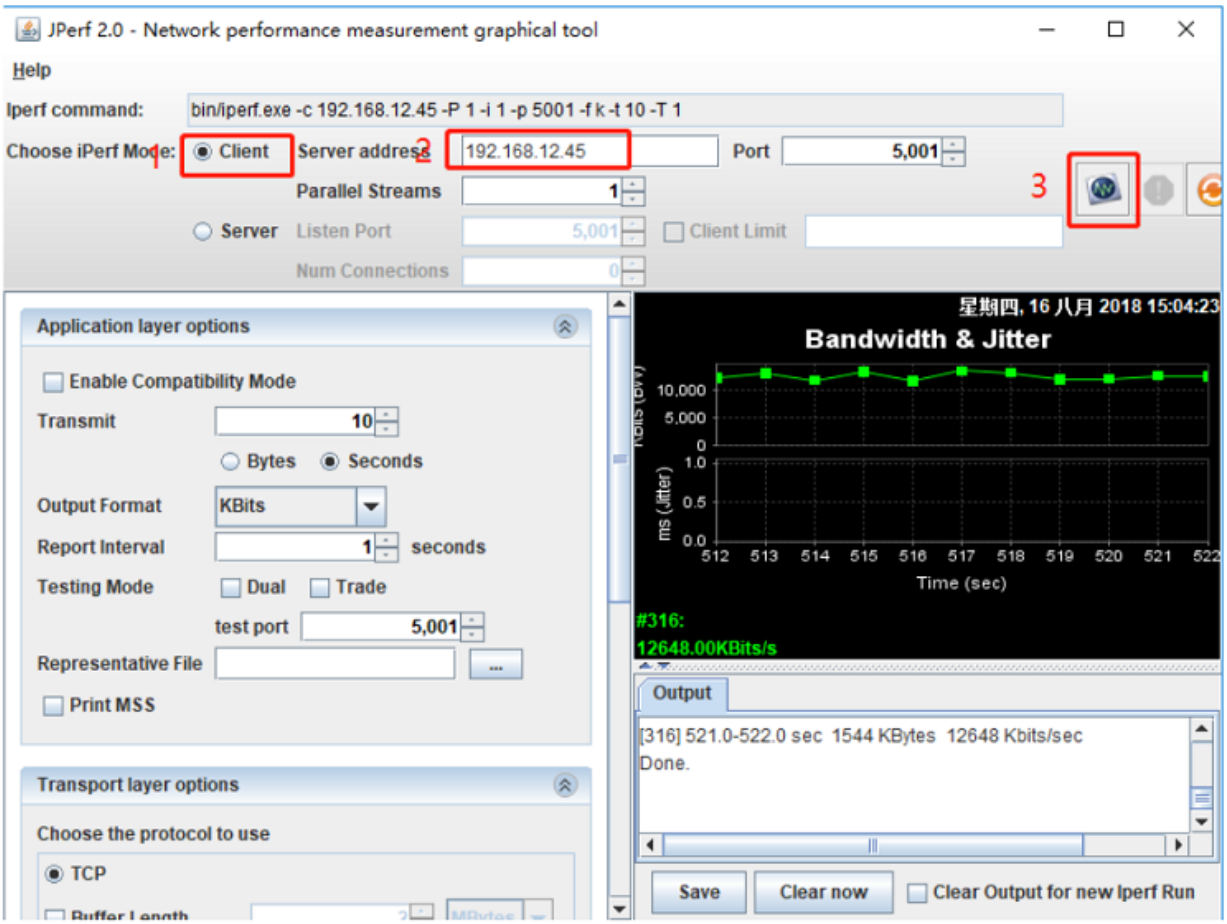

图 26.4: *iperf client*

Iperf 客户端模式

1. 获取 PC 的 IP 地址

   在 PC 的命令提示符窗口上使用 ipconfig 命令获取 PC 的 IP 地址，记下获得的 PC IP 地址为 192.168.12.45（按实际情况记录）。

2. 安装 JPerf 测试软件

   安装文件位于 netutils/tools/jperf.rar ，这个是绿色软件，安装实际上是解压的过程，解压到新文件夹即可。

3. 开启 jperf 服务器

   打开 jperf.bat 软件，按如下操作进行配置：

   1. 选择 Server 模式；
   2. 修改端口号为 5001；
   3. 点击 run Lperf! 开启服务器；

4. 启动 Iperf 客户端

   在 RT-Thread 上启动 Iperf 客户端需要执行如下命令：

```
msh />iperf -c 192.168.12.45 -p 5001
```

```
参数 -c 表示作为客户端启动，后面需要加运行服务器端的 pc 的 IP 地址，参数 -p 表示连接
    5001 端口。等待测试结束，数据会在 shell 界面和 JPerf 软件上显示。

![iperf server](/builds/document/document-control-center/docs/spark-pdf/projects
    -1.0/05_iot_netutils/./figures/jprf2.png)
```

26.4.2.7 更多网络调试工具

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、Telnet 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。

26.5 注意事项

准备工作中，执行挂载文件系统操作之前要确保存储设备中有相应类型的文件系统，否则会挂载失败。

26.6 引用参考

- netutils 软件包：https://github.com/RT-Thread-packages/netutils

第 27 章

中国移动 OneNET 云平台接入例程

本例程演示如何使用 RT-Thread 提供的 onenet 软件包接入中国移动物联网开放平台，介绍如何通过 MQTT 协议接入 OneNET 平台，初次使用 OneNET 平台的用户请先阅读 《OneNET 文档中心》。

27.1 简介

OneNET 平台是中国移动基于物联网产业打造的生态平台，具有高并发可用、多协议接入、丰富 API 支持、数据安全存储、快速应用孵化等特点。OneNET 平台可以适配各种网络环境和协议类型，现在支持的协议有 LWM2M（NB-IOT）、EDP、MQTT、HTTP、MODBUS、JT808、TCP 透传、RGMP 等。用户可以根据不同的应用场景选择不同的接入协议。

onenet 软件包是 RT-Thread 针对 OneNET 平台做的适配，通过这个软件包可以让设备在 RT-Thread 上非常方便的连接 OneNet 平台，完成数据的发送、接收、设备的注册和控制等功能。

27.2 硬件说明

onenet 例程需要依赖星火 1 号板卡上的 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。

27.3 准备工作

27.3.1 创建产品

onenet 用来管理设备的网址为 产品管理 点击添加产品来添加一个产品

产品配置如图

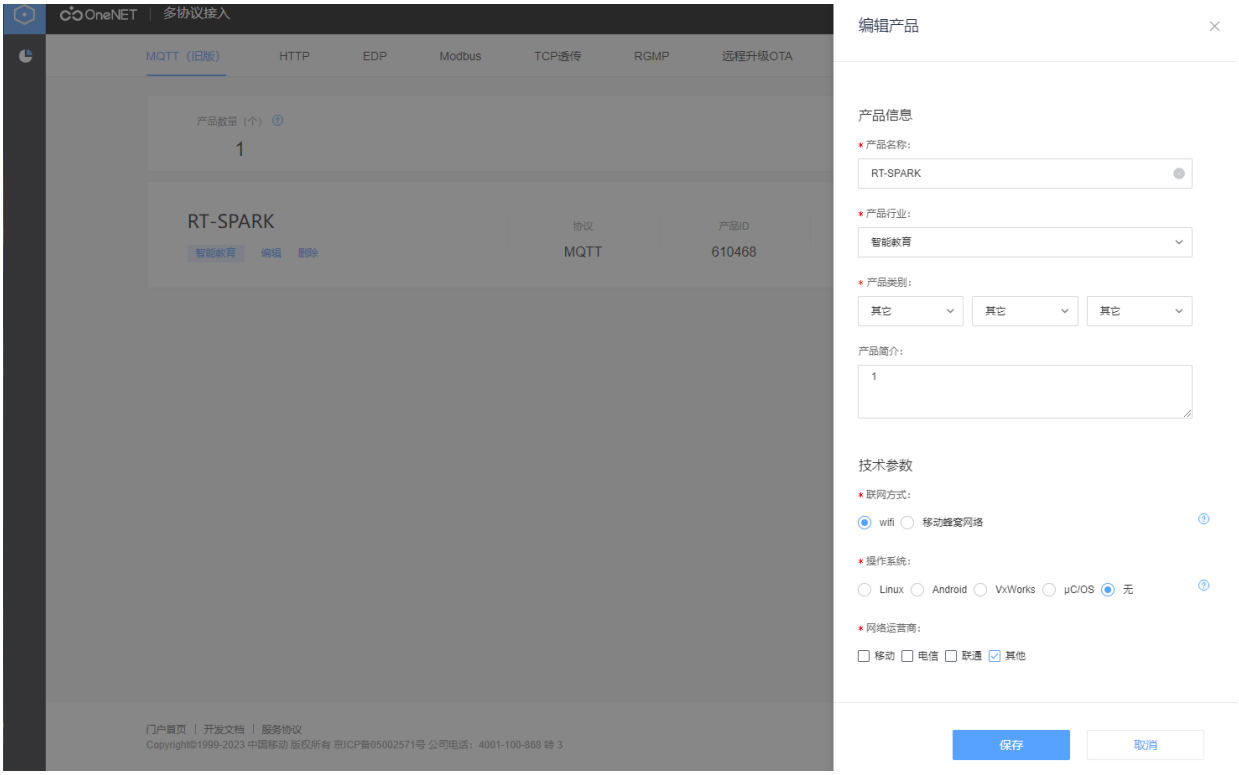

图 27.1: *创建产品*

27.3.2 创建设备

点击进入产品，点击设备列表，点击添加设备。设备配置信息如下：

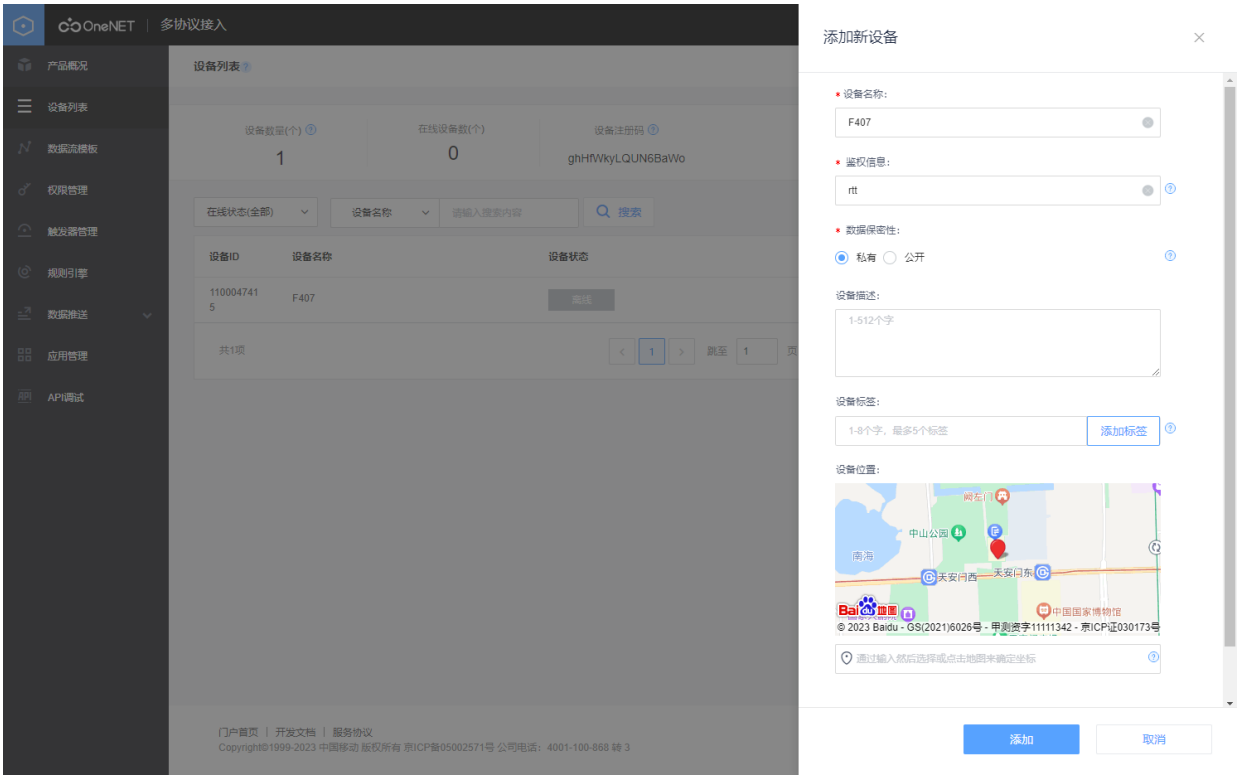

图 27.2: *创建设备*

27.3.3 代码移植

代码移植主要是为了在设备端配置云端账户信息。

双击 RT-Thread Settings，点击 OneNet 软件包进行配置

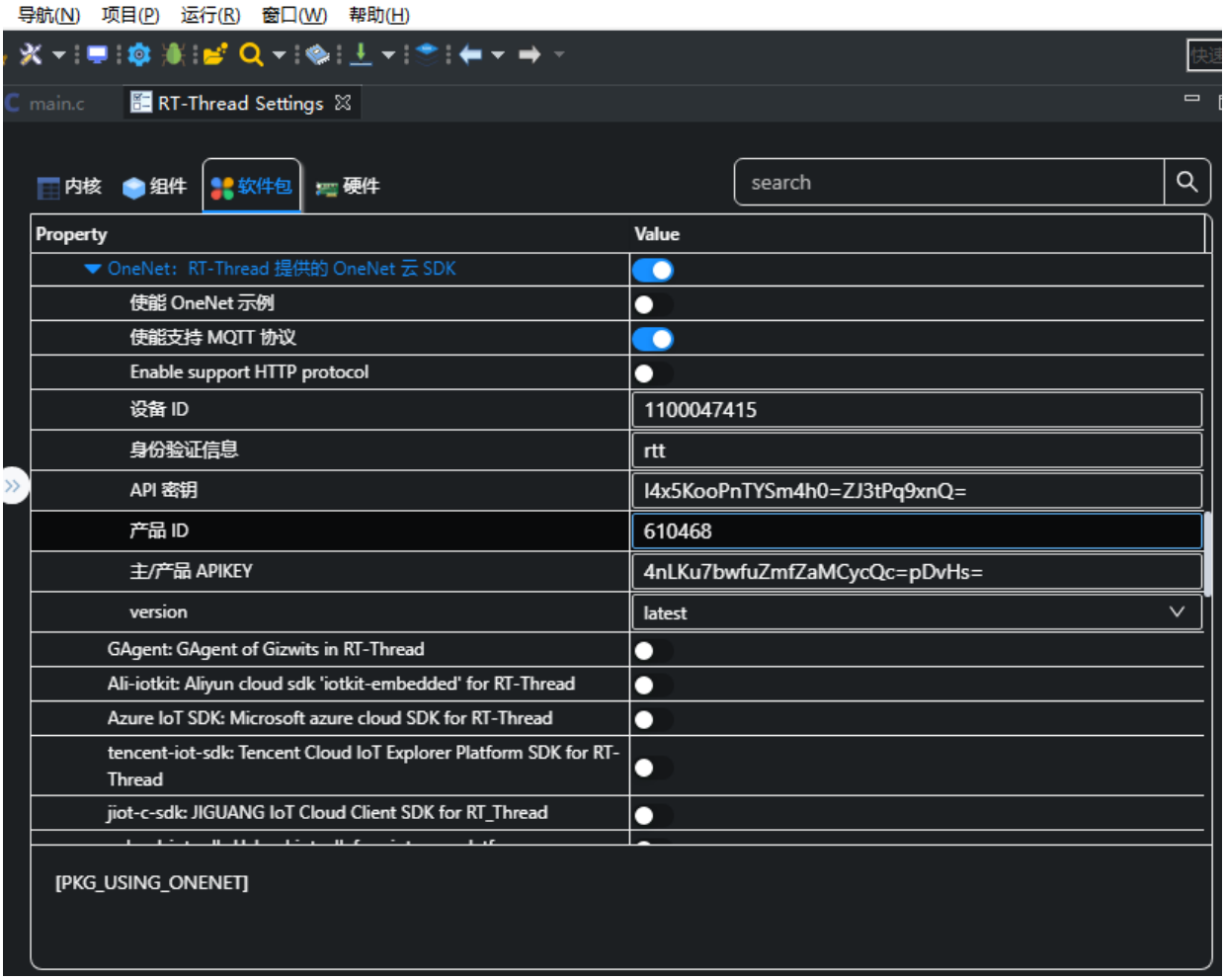

图 27.3: *OneNet 配置*

设备信息和软件包配置映射如下表

软件包配置	onenet 对应信息
设备 ID	设备 ID
身份验证信息	鉴权信息
API 秘钥	APIKey
产品 ID	产品 ID
主/产品 APIKEY	Master-APIkey

对应信息在 OneNet 中查找

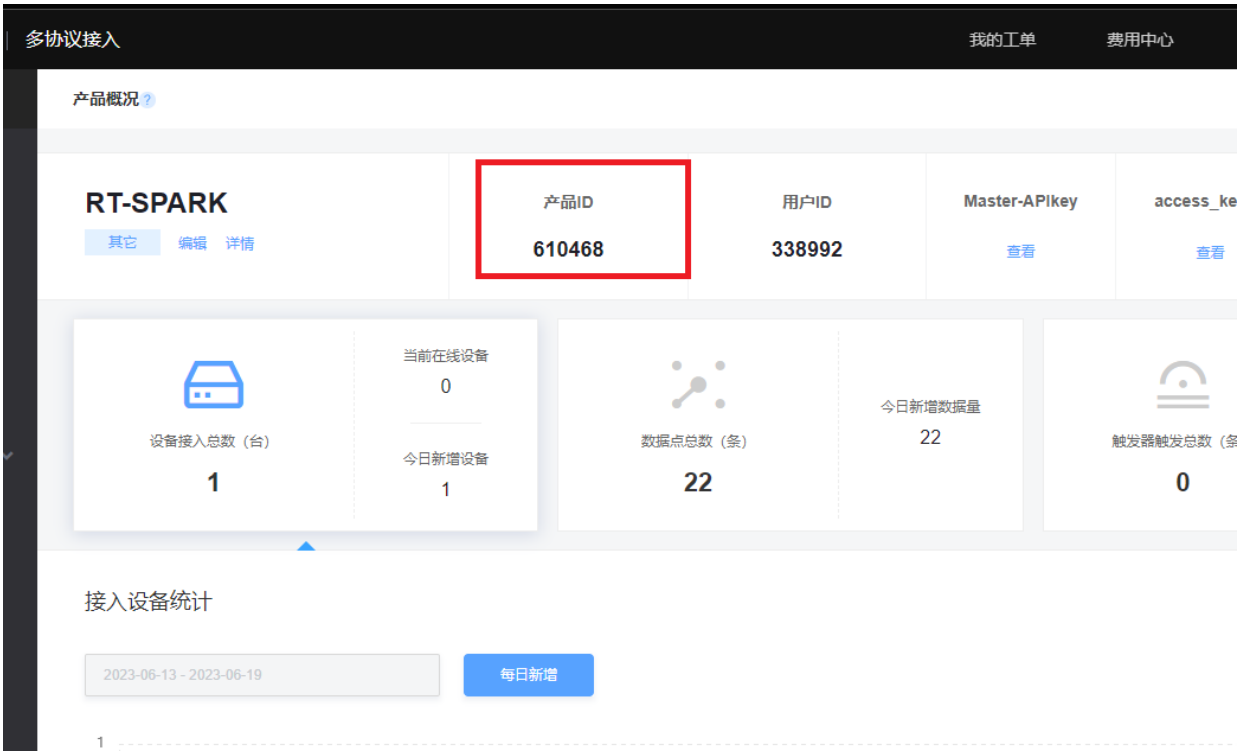

图 27.4: *产品 ID 位置*

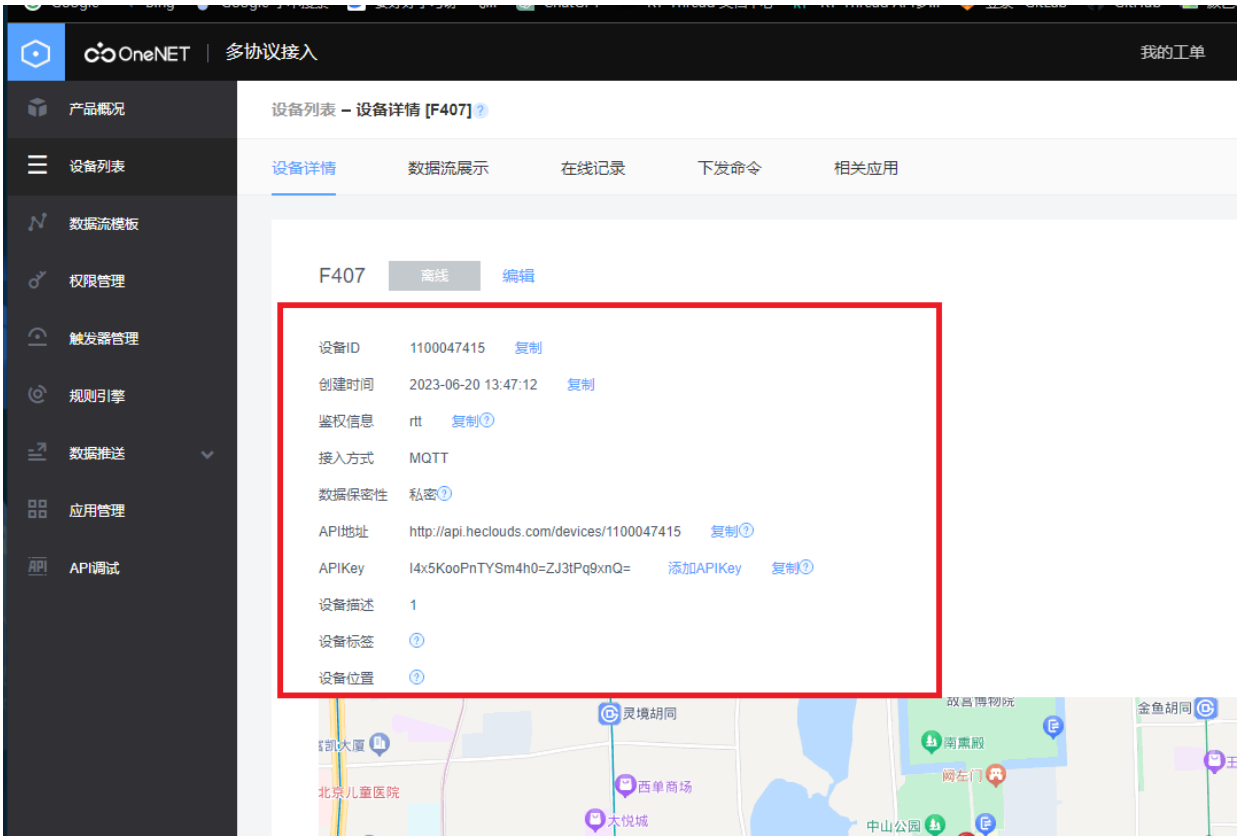

图 27.5: *设备信息位置*

27.4 软件说明

本例程的源码位于 /projects/05_iot_cloud_onenet。本例程主要实现了使用 MSH 验证上传和下发功能，程序代码位于applications/onenet_sample.c‘ 文件中。

在 onenet_sample.c 中，主要完成了以下几个任务：

注册周期上传任务到 MSH

```
static void onenet_upload_entry(void *parameter)
{
    int value = 0;

    while (1)
    {
        value = rand() % 100;

        if (onenet_mqtt_upload_digit("temperature", value) < 0)
        {
            LOG_E("upload has an error, stop uploading");
            break;
        }
        else
        {
            LOG_D("buffer : {\"temperature\":%d}", value);
        }

        rt_thread_delay(rt_tick_from_millisecond(5 * 1000));
    }
}

int onenet_upload_cycle(void)
{
    rt_thread_t tid;

    tid = rt_thread_create("onenet_send",
                           onenet_upload_entry,
                           RT_NULL,
                           2 * 1024,
                           RT_THREAD_PRIORITY_MAX / 3 - 1,
                           5);
    if (tid)
    {
        rt_thread_startup(tid);
    }

    return 0;
}
MSH_CMD_EXPORT(onenet_upload_cycle, send data to OneNET cloud cycle);
```

注册上传数字量函数、上传字符串函数到 MSH

```
int onenet_publish_digit(int argc, char **argv)
{
    if (argc != 3)
    {
        LOG_E("onenet_publish [datastream_id]  [value]  - mqtt pulish digit data to
            OneNET.");
        return -1;
    }

    if (onenet_mqtt_upload_digit(argv[1], atoi(argv[2])) < 0)
    {
        LOG_E("upload digit data has an error!\n");
    }

    return 0;
}
MSH_CMD_EXPORT_ALIAS(onenet_publish_digit, onenet_mqtt_publish_digit, send digit
    data to onenet cloud);

int onenet_publish_string(int argc, char **argv)
{
    if (argc != 3)
    {
        LOG_E("onenet_publish [datastream_id]  [string]  - mqtt pulish string data
            to OneNET.");
        return -1;
    }

    if (onenet_mqtt_upload_string(argv[1], argv[2]) < 0)
    {
        LOG_E("upload string has an error!\n");
    }

    return 0;
}
MSH_CMD_EXPORT_ALIAS(onenet_publish_string, onenet_mqtt_publish_string, send string
    data to onenet cloud);
```

以及接受命令控制 LED 函数

```
/* onenet mqtt command response callback function */
static void onenet_cmd_rsp_cb(uint8_t *recv_data, size_t recv_size, uint8_t **
    resp_data, size_t *resp_size)
{
    char res_buf[] = { "cmd is received!\n" };

    LOG_D("recv data is %x:%x\n", recv_data[0], recv_data[1]);
```

```
    if(recv_data[0] == 0x00)
    {
        rt_pin_write(PIN_LED_B, recv_data[1] > 0 ? PIN_HIGH : PIN_LOW);
        LOG_D("blue light %d", recv_data[1]);
    }else if(recv_data[0] == 0x01)
    {
        rt_pin_write(PIN_LED_R, recv_data[1] > 0 ? PIN_HIGH : PIN_LOW);
        LOG_D("red light %d", recv_data[1]);
    }

    /* user have to malloc memory for response data */
    *resp_data = (uint8_t *) ONENET_MALLOC(strlen(res_buf));

    strncpy((char *)*resp_data, res_buf, strlen(res_buf));

    *resp_size = strlen(res_buf);
}

/* set the onenet mqtt command response callback function */
int onenet_set_cmd_rsp(int argc, char **argv)
{
    rt_pin_mode(PIN_LED_B,PIN_MODE_OUTPUT);
    rt_pin_mode(PIN_LED_R,PIN_MODE_OUTPUT);
    onenet_set_cmd_rsp_cb(onenet_cmd_rsp_cb);
    return 0;
}
MSH_CMD_EXPORT(onenet_set_cmd_rsp, set cmd response function);
```

27.5 运行

27.5.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

程序运行日志如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 20 2023 17:23:42
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
```

```
msh />[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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
```

27.5.2 连接 wifi

输入自己的 wifi 和密码连接 wifi

```
msh />wifi join demo 12345678
[I/WLAN.mgnt] wifi connect success ssid:demo
msh />[I/WLAN.lwip] Got IP address : 192.168.137.118
```

可以看到，wifi 已经连接成功，并且输出了 IP 地址。

27.5.3 连接设备

```
msh />onenet_mqtt_init
[D/onenet.mqtt] Enter mqtt_connect_callback!
[D/mqtt] ipv4 address port: 6002
[D/mqtt] HOST = '183.230.40.39'
[I/onenet.mqtt] RT-Thread OneNET package(V1.0.0) initialize success.
msh />[I/mqtt] MQTT server connect success.
[D/onenet.mqtt] Enter mqtt_online_callback!
```

此时会显示网站会显示设备处于在线状态：

图 27.6: 设备在线

27.5.4 数据上传

成功建立连接之后，输入命令 onenet_upload_cycle，完成数据发送

```
msh />onenet_upload_cycle
[D/onenet.sample] buffer : {"temperature":52}
msh />[D/onenet.sample] buffer : {"temperature":56}
```

```
[D/onenet.sample] buffer : {"temperature":56}
[D/onenet.sample] buffer : {"temperature":19}
[D/onenet.sample] buffer : {"temperature":11}
```

打开 OneNET 平台，在**设备列表**页面，选择**数据流展示**，点击展开 temperature 数据流，可以看到刚刚上传的数据信息。

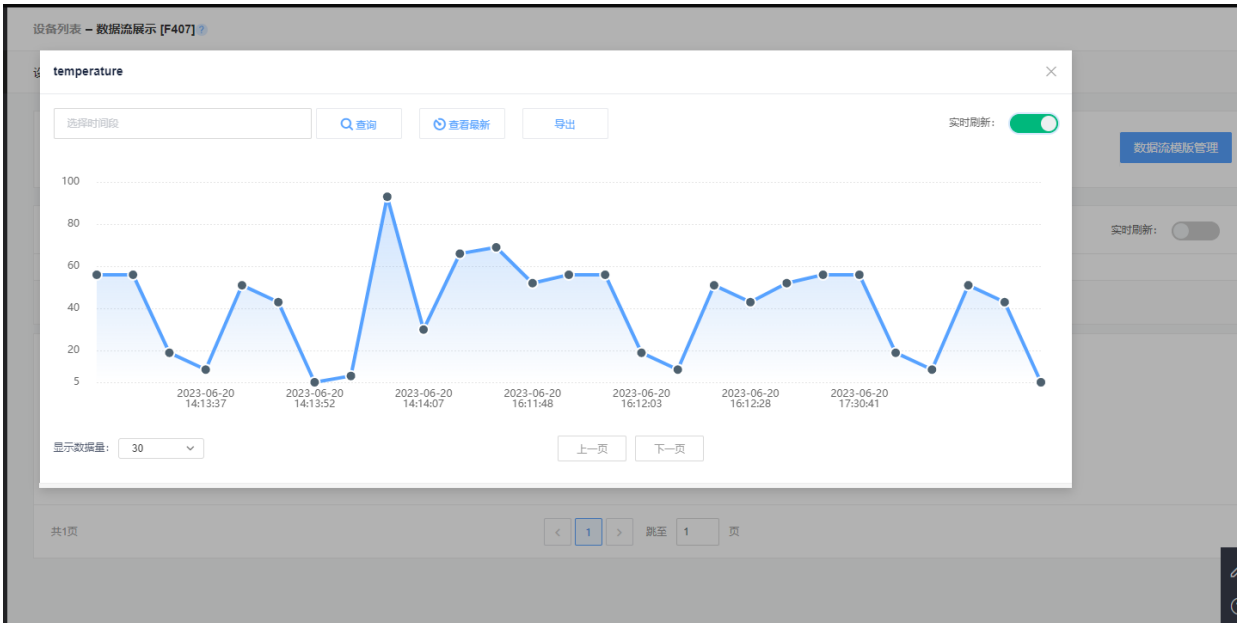

图 27.7: *temperature 数据流*

27.5.5 命令控制

在**设备列表**界面，选择**下发命令**，点击蓝色的**下发命令**按钮，会弹出一个命令窗口，我们可以下发命令给开发板。我们可以定义一个两个字段的命令 xx xx, 第一个字段代表 led 的索引，第二个字段代表值。比如 00 01 代表红灯灭，00 00 代表红灯亮。我们在 OneNet 界面下发命令，控制 LED。

图 27.8: *下发命令*

```
[D/onenet.mqtt] topic $creq/7ede0f8b-1f78-58ca-86d3-cc2314915a4e receive a message
[D/onenet.mqtt] message length is 2
```

并且观察到 LED 状态也发生了变化。

27.6 注意事项

暂无

27.7 引用参考

- 设备与驱动：PIN 设备
- onenet 软件包：https://github.com/RT-Thread-packages/onenet
- 文档中心：RT-Thread 文档中心

第 28 章

阿里云物联网平台接入例程

本例程演示如何使用 RT-Thread 提供的 ali-iotkit 软件包接入阿里云物联网平台，介绍如何通过 MQTT 协议接入阿里云平台。演示使用 mqtt 协议上传和下发数据。

28.1 简介

ali-iotkit 是 RT-Thread 移植的用于连接阿里云 IoT 平台的软件包。基础 SDK 是阿里提供的 **iotkit-embedded C-SDK**。

ali-iotkit 为了方便设备上云封装了丰富的连接协议，如 MQTT、CoAP、HTTP、TLS，并且对硬件平台进行了抽象，使其不受具体的硬件平台限制而更加灵活。

相对传统的云端接入 SDK，RT-Thread 提供的 **ali-iotkit** 具有如下优势：

- 快速接入能力
- 嵌入式设备调优
- 多编译器支持（GCC、IAR、MDK）
- 多连接协议支持（HTTP、MQTT、CoAP）
- 强大而丰富的 RT-Thread 生态支持
- 跨硬件、跨 OS 平台支持
- 设备固件 OTA 升级
- TLS/DTLS 安全连接
- 高可移植的应用端程序
- 高可复用的功能组件

28.2 硬件说明

ali-iotkit 例程需要依赖星火 1 号板卡上的 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。

28.3 软件说明

ali-iotkit 例程位于 projects/05_iot_cloud_ali_iotkit 目录下，重要文件摘要说明如下所示：

文件	说明
applications/main.c	app 入口
packages/ports	移植文件
packages/ali-iotkit-v3.0.1	阿里云物联网平台接入软件包
packages/ali-iotkit-v3.0.1/samples	阿里云物联网平台接入示例

28.3.1 例程使用说明

该 ali-iotkit 演示例程程序代码位于 /projects/05_iot_cloud_ali_iotkit/packages/ali-iotkit-v3.0.1/samples/mqtt/mqtt-example.c 文件中，核心代码说明如下：

1. 设置激活凭证

使用 Rt-Thread Settings 设置设备激活凭证（从阿里云—— IoT Studio 平台获取），如下图所示：

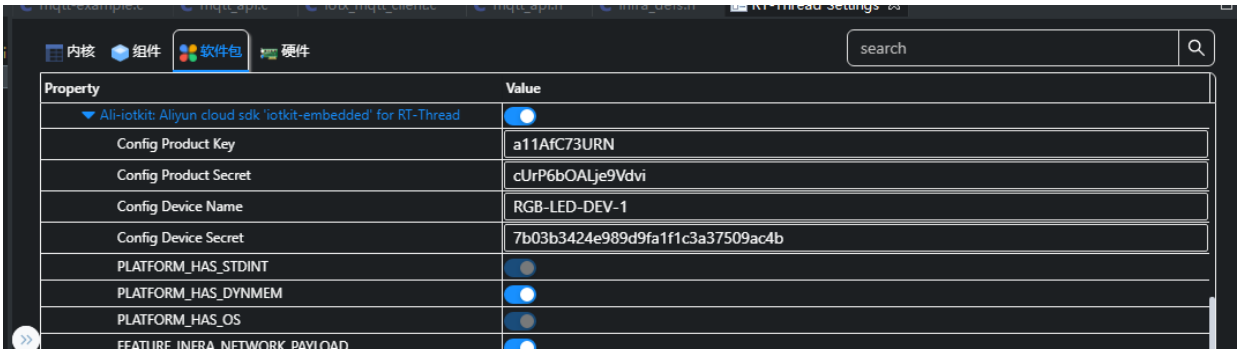

图 28.1: *设置激活凭证*

通过 Rt-Thread Settings 工具完成配置的保存后，工具会将相关的配置项写入到 05_iot_cloud_ali_iotkit/rtconfig.h 文件中，用户也可以直接修改 rtconfig.h 文件，如下所示：

```
/* 使用阿里 LinkDevelop 平台 */
#define PKG_USING_ALI_IOTKIT_IS_LINKDEVELOP
/* 配置产品 ID */
#define PKG_USING_ALI_IOTKIT_PRODUCT_KEY "a1HETlEuvri"
/* 配置设备名字 */
#define PKG_USING_ALI_IOTKIT_DEVICE_NAME "RGB-LED-DEV-1"
/* 配置设备密钥 */
#define PKG_USING_ALI_IOTKIT_DEVICE_SECRET "ZwQwQNXfXVeXF39km0fw6SirQgRIq8aO"
```

2. 注册 MQTT 事件回调

注册 mqtt 的事件回调函数，代码如下所示：

```
mqtt_params.handle_event.h_fp = example_event_handle;
```

3. MQTT 事件回调

每当 MQTT 客户端收到服务器发来的事件就会调用 example_event_handle 函数，将收到的事件信息交由用户自定义处理，代码如下所示：

```
static void example_event_handle(void *pcontext, void *pclient,
    iotx_mqtt_event_msg_pt msg)
{
    EXAMPLE_TRACE("msg->event_type : %d", msg->event_type);
}
```

4. 启动 MQTT 客户端

4.1 初始化 iotkit 连接信息

使用用户配置的设备激活凭证（PRODUCT_KEY、DEVICE_NAME 和 DEVICE_SECRET）初始化 iotkit 连接信息，代码如下所示：

```
    HAL_GetProductKey(DEMO_PRODUCT_KEY);
    HAL_GetDeviceName(DEMO_DEVICE_NAME);
    HAL_GetDeviceSecret(DEMO_DEVICE_SECRET);
```

4.2 配置 MQTT 客户端参数

初始化 MQTT 客户端所必须的用户名、密码、服务器主机名及端口号等配置，代码如下所示：

```
    iotx_mqtt_param_t mqtt_params;

    /* 初始化 MQTT 数据结构 */
    iotx_mqtt_param_t       mqtt_params;

    memset(&mqtt_params, 0x0, sizeof(mqtt_params));
    /* Initialize MQTT parameter */
    /*
     * Note:
     *
     * If you did NOT set value for members of mqtt_params, SDK will use their
        default values
     * If you wish to customize some parameter, just un-comment value assigning
        expressions below
     *
     **/
    mqtt_params.handle_event.h_fp = example_event_handle;
```

4.3 创建 MQTT 客户端

使用 IOT_MQTT_Construct 函数构建一个 MQTT 客户端，代码如下所示：

```
/* 使用指定的 MQTT 参数构建 MQTT 客户端数据结构 */
pclient = IOT_MQTT_Construct(&mqtt_params);
```

```
if (NULL == pclient) {
    EXAMPLE_TRACE("MQTT construct failed");
    return -1;
}
```

4.4 订阅指定的 Topic

使用 IOT_MQTT_Subscribe 函数订阅指定的 Topic，代码如下所示：

```
res = IOT_MQTT_Subscribe(handle, topic, IOTX_MQTT_QOS0, example_message_arrive, NULL
    );
if (res < 0) {
    EXAMPLE_TRACE("subscribe failed");
    HAL_Free(topic);
    return -1;
}
```

消息主题为：/PRODUCT_KEY/DEVICE_NAME/user/get，用于接受和发布日志消息。

4.5 循环发布消息并且等待 MQTT 消息通知

使用 IOT_MQTT_Yield 函数接收来自云端的消息。

```
while (1) {
    if (0 == loop_cnt % 20) {
        example_publish(pclient);
    }

    IOT_MQTT_Yield(pclient, 200);

    loop_cnt += 1;
}
```

5. 关闭 MQTT 客户端

关闭 MQTT 客户端需要取消已经存在的订阅，并销毁 MQTT 客户端连接，以释放资源。

5.1 取消订阅指定的 Topic

使用 IOT_MQTT_Unsubscribe 取消已经存在的订阅。

```
int IOT_MQTT_Unsubscribe(void *handle, const char *topic_filter);
```

5.2 销毁 MQTT 客户端

使用 IOT_MQTT_Destroy 销毁已经存在的 MQTT 客户端，以释放资源占用。

```
int IOT_MQTT_Destroy(void **phandle);
```

6. 发布测试消息

6.1 构建 Alink 协议格式数据

本例程中构建一个日志消息需要的 Alink 协议格式的数据包，如下所示：

```
/* 初始化要发送给 Topic 的消息内容 */
const char      *fmt = "/%s/%s/user/get";
char            *topic = NULL;
int              topic_len = 0;
char            *payload = "{\"message\":\"hello!\"}";

topic_len = strlen(fmt) + strlen(DEMO_PRODUCT_KEY) + strlen(DEMO_DEVICE_NAME) + 1;
topic = HAL_Malloc(topic_len);
if (topic == NULL) {
    EXAMPLE_TRACE("memory not enough");
    return -1;
}
memset(topic, 0, topic_len);
HAL_Snprintf(topic, topic_len, fmt, DEMO_PRODUCT_KEY, DEMO_DEVICE_NAME);
```

6.2 发布消息

使用 IOT_MQTT_Publish_Simple 接口向 MQTT 通道发送 Alink 协议格式的消息。

```
int res = 0;
res = IOT_MQTT_Publish_Simple(0, topic, IOTX_MQTT_QOS0, payload, strlen(payload));
if (res < 0) {
    EXAMPLE_TRACE("publish failed, res = %d", res);
    HAL_Free(topic);
    return -1;
}
HAL_Free(topic);
return 0;
```

28.4 运行

28.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

程序运行日志如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 27 2023 15:45:16
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
```

```
[I/sal.skt] Socket Abstraction Layer initialize success.
/>[E/[RW007]] The wifi Stage 1 status 67452301 e0cdab09 1 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]
```

28.4.2 连接无线网络

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（ssid 和 password 分别为设备连接的 WIFI 用户名和密码），如下所示：

```
wifi join demo 12345678
[I/WLAN.mgnt] wifi connect success ssid:demo
msh />[I/WLAN.lwip] Got IP address : 192.168.137.45
```

28.4.3 运行 mqtt demo

```
msh />ali_mqtt_sample
mqtt_example_main|134 :: mqtt example
host name:a11afc73urn.iot-as-mqtt.cn-shanghai.aliyuncs.com

> {
>     "id": "0",
>     "version": "1.0",
>     "params": [
>         {
>             "attrKey": "SYS_LP_SDK_VERSION",
>             "attrValue": "3.0.1",
>             "domain": "SYSTEM"
>         },
>         {
>             "attrKey": "SYS_SDK_LANGUAGE",
>             "attrValue": "C",
>             "domain": "SYSTEM"
>         }
>     ],
>     "method": "thing.deviceinfo.update"
> }


> {
>     "id": "1",
>     "params": {
```

```
>         "version": "app-1.0.0-20180101.1000"
>     }
> }


> {
>     "message": "hello!"
> }

example_event_handle|107 :: msg->event_type : 9
example_event_handle|107 :: msg->event_type : 9
example_event_handle|107 :: msg->event_type : 3

< {
<     "message": "hello!"
< }

example_message_arrive|041 :: Message Arrived:
example_message_arrive|042 :: Topic   : /a11AfC73URN/RGB-LED-DEV-1/user/get
example_message_arrive|043 :: Payload: {"message":"hello!"}
example_message_arrive|044 ::
```

如上所示，例程启动，程序周期的向订阅的话题发布消息，由于发布的话题也被自己订阅了，所以发布完消息后自己也接收到了自己发布的消息。

28.5 注意事项

- 使用前请先连接 wifi。
- 使用前请在 Rt-Thread Settings 里配置自己的设备激活凭证（PRODUCT_KEY、DEVICE_NAME 和 DEVICE_SECRET）

28.6 引用参考

- 文档中心：RT-Thread 文档中心
- ali-iotkit 软件包：https://github.com/RT-Thread-packages/ali-iotkit

第 29 章

使用 Web 服务器组件：WebNet

29.1 简介

本例程使用 WebNet 软件包创建一个 Web 服务器，并展示 Web 服务器静态页面、CGI（事件处理）、AUTH（基本认证）、Upload（文件上传）等部分功能。

29.2 硬件说明

本例程需要依赖星火 1 号板卡上的 WiFi 模块和 TF 卡模块，因此请确保硬件平台上的 WiFi 模块和 TF 卡模块可以正常工作。

29.3 准备工作

1. 使能文件系统（可以使用 SD 卡文件系统或者 Flash 文件系统）
2. 在 TF 卡根目录下创建 webnet 文件夹。(本示例在 Rt-Thread Settings 中已经开启了 SD 卡文件系统，并且使能了自动挂载。所以路径为 /sdcard/webnet , 并且已经在 Rt-Thread Settings 中配置了 webnet 根目录为 “/sdcard/webnet”)
3. 在 webnet 文件夹里创建 admin 和 upload 两个文件夹。
4. 将 `/projects/05_iot_web_server/packages/webnet-v2.0.3/samples/` 目录下 index.html、index.shtml、version.asp 复制到 TF 卡的 webnet 文件夹里。
    - 方案 A 可以使用 netutils 软件包中的 tftp 工具配合 pc 端 tftpd 软件。
    - 方案 B 可以使用读卡器直接将文件复制进 SD 卡对应目录。

完成之后的目录结构如图所示:

```
msh /sdcard/webnet>ls
Directory /sdcard/webnet:
index.html          2737
index.shtml         322
```

```
version.asp         205
admin               <DIR>
upload              <DIR>
```

index.html 文件是访问 web 服务器时默认展示的页面，我们提供的 html 文件是用来测试本例程的相关功能的。熟悉 HTML 语言的开发者可以自己编写 index.html 页面，并放入 webnet 文件夹里。

29.4 软件说明

WebNet 例程位于 /projects/05_iot_web_server 目录下，重要文件摘要说明如下表所示：

文件	说明
applications/main.c	app 入口
packages/webnet-v2.0.3	示例网页

本例程主要展示了 WebNet 的几个常用功能，程序代码位于 /projects/05_iot_web_server/packages/webnet-v2.0.3/samples/wn_sample.c 文件中。

```
void webnet_test(void)
{
    webnet_cgi_register("hello", cgi_hello_handler);
    webnet_cgi_register("calc", cgi_calc_handler);
    webnet_asp_add_var("version", asp_var_version);
    webnet_alias_set("/test", "/admin");
    webnet_auth_set("/admin", "admin:admin");
    extern const struct webnet_module_upload_entry upload_entry_upload;
    webnet_upload_add(&upload_entry_upload);

    webnet_init();
}
```

29.5 运行

29.5.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

29.5.2 运行效果

按下复位按键重启开发板，正常运行后，终端输出信息如下：

```
msh /sdcard/webnet/upload>
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 28 2023 10:19:44
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
msh />[I/SDIO] SD card capacity 1967104 KB.
m[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cf0c]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

[I/app.filesystem] SD card mount to '/sdcard'
```

连接 wifi 并且启动测试程序

```
wifi join demoo 19981111
[I/WLAN.mgnt] wifi connect success ssid:demoo
msh />[I/WLAN.lwip] Got IP address : 192.168.181.31
webnet_test
msh />webnet_test
```

29.5.3 静态页面展示

访问 web 服务器根目录时，web 服务器会默认展示根目录下的 **index.html** 或者 **index.htm** 文件。如果没找到文件，就会列出根目录下的所有文件。根目录可以通过修改 Rt-Thread Settings 中的 Server root directory 来修改，默认为 `/sdcard/webnet`。

在浏览器（这里使用谷歌浏览器）中输入刚刚打印的设备 IP 地址，将访问我们之前放入根目录（/sdcard/webnet）的 **index.html** 文件，如下图所示，页面文件正常显示：

图 29.1: *root*

该页面上显示了 WebNet 软件包的基本功能介绍，并在下方给出相应的测试示例。

开发者如果想要访问别的页面，可以将要展示的网页放在 TF 卡根目录下任意路径，然后在浏览器里输入服务器的 IP 地址加网页相对于根目录的相对路径，即可访问对应的网页。例如，根目录为 `/sdcard/webnet`，想要访问在 TF 卡里的 `/sdcard/webnet/product/a.html` 网页，在浏览器里输入：开发板的 IP 地址 `/product/a.html`，即可打开 a.html 网页。

29.5.4 AUTH 基本认证例程

在例程主页（/index.html）AUTH Test 模块下点击 `基本认证功能测试：用户名及密码为 admin:admin` 按键，弹出基本认证对话框，输入用户名 **admin**，密码 **admin**。成功输入用户名和密码后，会进入根目录下 /admin 目录，如下图所示流程：

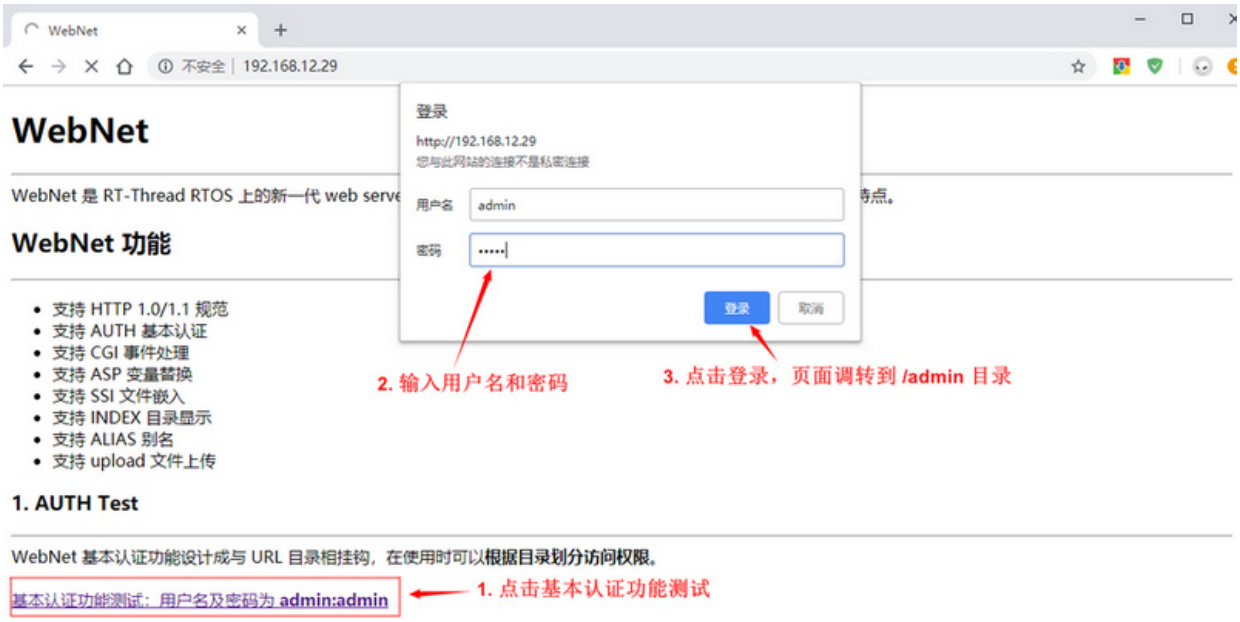

图 29.2: *auth*

29.5.5 Upload 文件上传例程

Upload 例程实现上传文件到 WebNet 服务器固定目录功能。在例程主页上 Upload File Test 模块下点击 选择文件 按键，选取需要上传的文件（本例程是用 upload.txt 文件），点击 上传，可以将文件上传到根目录下的 /upload 目录，如下图所示：

图 29.3: *upload_file*

29.5.6 INDEX 目录显示例程

INDEX 例程演示页面文件列表功能。

然后在例程主页 INDEX Test 模块下点击 INDEX 功能测试：访问 /upload 目录 按键，会跳转到根目录下 /upload 目录，并且列出该目录下所有文件名和文件长度，如下图所示：

图 29.4: *index*

29.5.7 CGI 事件处理例程

本例程提供两个 CGI 示例：**hello world 例程**和 **calc 例程**，用于演示 CGI 事件处理的基本功能。

- **hello world 例程**

hello world 例程演示了在页面演示文本内容功能，在例程主页 CGI Test 模块下点击 > `hello world` 按键，会跳转到新页面显示日志信息，新页面显示了 hello world 说明 CGI 事件处理成功，然后在新页面点击 `Go back to root` 按键回到例程主页面，如下图所示：

图 29.5: *hello_world*

- **calc 例程**

calc 例程中使用 CGI 功能在页面上展示了简单的加法计算器功能，在例程主页 CGI Test 模块下点击 > calc 按键，会跳转到新的页面，输入两个数字点击 计算 按键，页面会显示两数字相加的结果。在新页面点击 Go back to root 按键可以回到例程主页面，如下图所示：

图 29.6: *calc*

29.6 注意事项

- 首次运行例程，需要使用 `wifi join ssid password` 来连接 WIFI。
- 如果初始化文件系统失败，需要使用 `mkfs -t elm sd0` 命令来在 TF 卡上建立一个文件系统，然后重启系统。
- TF 卡里需要有 webnet，webnet/admin 和 webnet/upload 三个文件夹，否则例程访问会出错。
- 想要了解更多的 WebNet 软件包的功能，可以查阅 WEBNET 用户手册。

29.7 引用参考

- 组件：文件系统
- 设备与驱动：WLAN 设备
- WebNet 软件包：`https://github.com/RT-Thread-packages/webnet`
- 文档中心：RT-Thread 文档中心

第 30 章

LVGL 例程

30.1 LVGL 简介

LVGL 全称 Light and Versatile Graphics Library, 是一个自由的，开源的 GUI 库，界面精美，资源消耗小，可移植度高，响应式布局等特点，全库采用纯 c 语言开发。主要特性如下：

- 具有非常丰富的内置控件，像 buttons, charts, lists, sliders, images 等高级图形效果：动画，反锯齿，透明度，平滑滚动
- 支持多种输入设备，像 touchpad, mouse, keyboard, encoder 等
- 支持多语言的 UTF-8 编码
- 支持多个和多种显示设备，例如同步显示在多个彩色屏或单色屏上
- 完全自定制的图形元素
- 硬件独立于任何微控制器或显示器
- 可以缩小到最小内存 (64 kB Flash, 16 kB RAM)
- 支持操作系统、外部储存和 GPU（非必须）
- 仅仅单个帧缓冲设备就可以呈现高级视觉特效
- 使用 C 编写以获得最大兼容性（兼容 C++）
- 支持 PC 模拟器
- 为加速 GUI 设计，提供教程，案例和主题，支持响应式布局
- 提供了在线和离线文档
- 基于自由和开源的 MIT 协议
- 支持 MicroPython

30.2 硬件说明

本例程将使用 LCD 显示屏显示 LVGL 的图像。

30.3 软件说明

本例程的源码位于 `/projects/06_demo_lvgl`。

lvgl 相关代码在 packages/LVGL-v8.3.5 目录，关于更多请参考 lvgl 开发文档。lvgl 的移植代码在 application/lvgl 目录下面。

首先是 lvgl 显示驱动的移植位于 application/lvgl/lv_port_disp.c

修改屏幕刷新函数：

```
static void disp_flush(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t
    * color_p)
{
    lcd_fill_array(area->x1, area->y1, area->x2, area->y2, color_p);

    lv_disp_flush_ready(disp_drv);
}
```

修改 framebuffer, 大小设置为屏幕尺寸的一半，并且放在 ccm 段。

```
    /* Example for 1) */
    static lv_disp_draw_buf_t draw_buf_dsc_1;

    /*GCC*/
#if defined ( __GNUC__ )
    static lv_color_t buf_1[MY_DISP_HOR_RES * MY_DISP_HOR_RES / 2] __attribute__((
        section(".LVGLccm")));                          /*A buffer for 10 rows*/
    /*MDK*/
#elif defined ( __CC_ARM )
    __attribute__((at(0x10000000))) lv_color_t buf_1[LCD_H * LCD_W / 2];
#endif

    lv_disp_draw_buf_init(&draw_buf_dsc_1, buf_1, NULL, MY_DISP_HOR_RES *
        MY_DISP_HOR_RES / 2);   /*Initialize the display buffer*/
```

其次是触摸函数，位于 application/lvgl/lv_port_indev.c。由于开发板没有触摸组件，所以触摸函数设置为空函数。

```
void lv_port_indev_init(void)
{

}
```

之后移植一个配置文件 libraries/Board_Drivers/lvgl/lv_conf.h。主要配置了屏幕的尺寸和颜色

```
#ifndef LV_CONF_H
#define LV_CONF_H

#include <rtconfig.h>

#define LV_COLOR_DEPTH          16
#define LV_USE_PERF_MONITOR     1
#define MY_DISP_HOR_RES          240
#define MY_DISP_VER_RES          240
```

使能 demo，位于 libraries/Board_Drivers/lvgl/demo/lv_demo.c。这里也是 lvgl 用户程序的入口。

```
void lv_user_gui_init(void)
{
    /* display demo; you may replace with your LVGL application at here */

    extern lv_demo_benchmark();
    lv_demo_benchmark();

}
```

30.4 运行

30.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

30.4.2 运行效果

按下复位键重启开发板，观察 lcd 屏幕上已经开始运行 lvgl 的 demo。

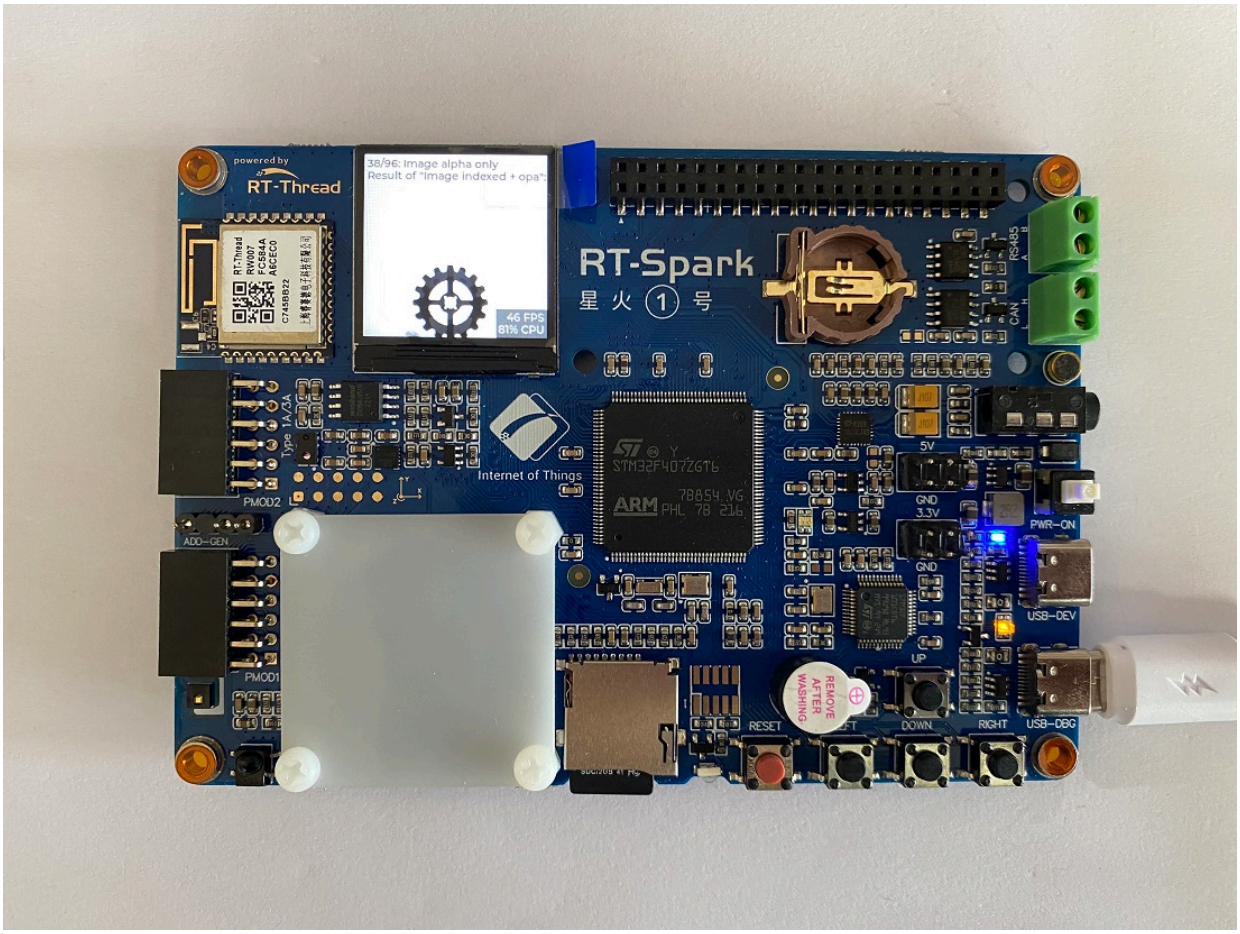

图 **30.1:** *lvgl 运行图*

30.5 注意事项

暂无

30.6 引用参考

- 文档中心：RT-Thread 文档中心
- lvgl 软件包：https://github.com/lvgl/lvgl

第 31 章

MicroPython 例程

31.1 简介

MicroPython 是 Python 3 编程语言的一种精简而高效的实现，它包含 Python 标准库的一个子集，并被优化为在微控制器和受限环境中运行。它具有交互式提示、任意精度整数、闭包函数、列表解析、生成器、异常处理等高级特性，具有很强的可移植性。它能够帮助开发者快速控制各种硬件外设，不用再去研究底层硬件模块的使用方法，翻看寄存器手册。降低了开发难度，而且减少了重复开发工作，可以加快开发速度，提高开发效率。

本例程作将介绍如何在 rt-thread 上使用 MicroPython ，并通过命令行展示 MicroPython 代码的输入与运行，最后使用一个示例，演示使用 MicroPython 控制硬件。

31.2 硬件说明

本例程将使用 MicroPython 控制星火 1 号上的各种硬件，请确保开发板上的相关硬件可以正常工作。

31.3 软件说明

本例程的源码位于 `/projects/06_demo_micropython`。main 程序代码位于 applications/main.c 中，主要是为 MicroPython 的提供必要的运行环境。如下：

```
#define FS_PARTITION_NAME     "filesystem"

int main(void)
{

    /* 打开 MicroPython 命令交互界面 */
    extern void mpy_main(const char *filename);
    mpy_main(NULL);
}
```

主函数内容主要是完成了 python 环境的启动。

31.4 运行

31.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

31.4.2 运行效果

在 PC 端使用终端工具打开开发板的 uart0 串口，设置 115200 8 1 N 。正常运行后，终端输出信息如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jun 12 2023 10:21:47
 2006 - 2022 Copyright by RT-Thread team
lwIP-2.0.3 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
msh >[E/[RW007]] The wifi Stage 1 status 0 0 0 1
[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

rw007  sn: [rw007c745bb22fc584aa6cecc]
rw007 ver: [RW007_2.1.0-a7a0d089-57]

The stack (main) size for executing MicroPython must be >= 4096

MicroPython v1.13-148-ged7ddd4 on 2020-11-03; Universal python platform with RT-
    Thread
Type "help()" for more information.
>>>
>>> print("hello RT-Thread")
hello RT-Thread
```

此时 MicroPython 命令交互界面就已经启动，可以通过命令行与 MicroPython 进行交互。下面将使用一个示例展示如何使用 MicroPython 控制硬件。

31.5 更多使用

MicroPython 的更多使用方法请参考 文档中心

31.6 注意事项

无

31.7 引用参考

- 文档中心：RT-Thread 文档中心
- MicroPython 软件包: https://github.com/RT-Thread-packages/micropython

第 32 章

板载 LED matrix 和 RS485 驱动例程

32.1 简介

本例程主要介绍了如何驱动板载 LED 以及如何使用 rs485 同步两块板子的 LED Matrix

32.2 硬件说明

星火 1 号开发板板载了一块 LED 阵列，共计 19 颗 LED RGB 灯珠。驱动使用的是 PWM 模拟信号来实现的。LED 矩阵使能端口连接在 PF12，LED PWM 信号端口连接在 PA7。星火 1 号开发板还板载了一块 RS-485 电平转换模块，接在芯片的 uart6 上面。

LED 矩阵和 RS485 原理图如下：

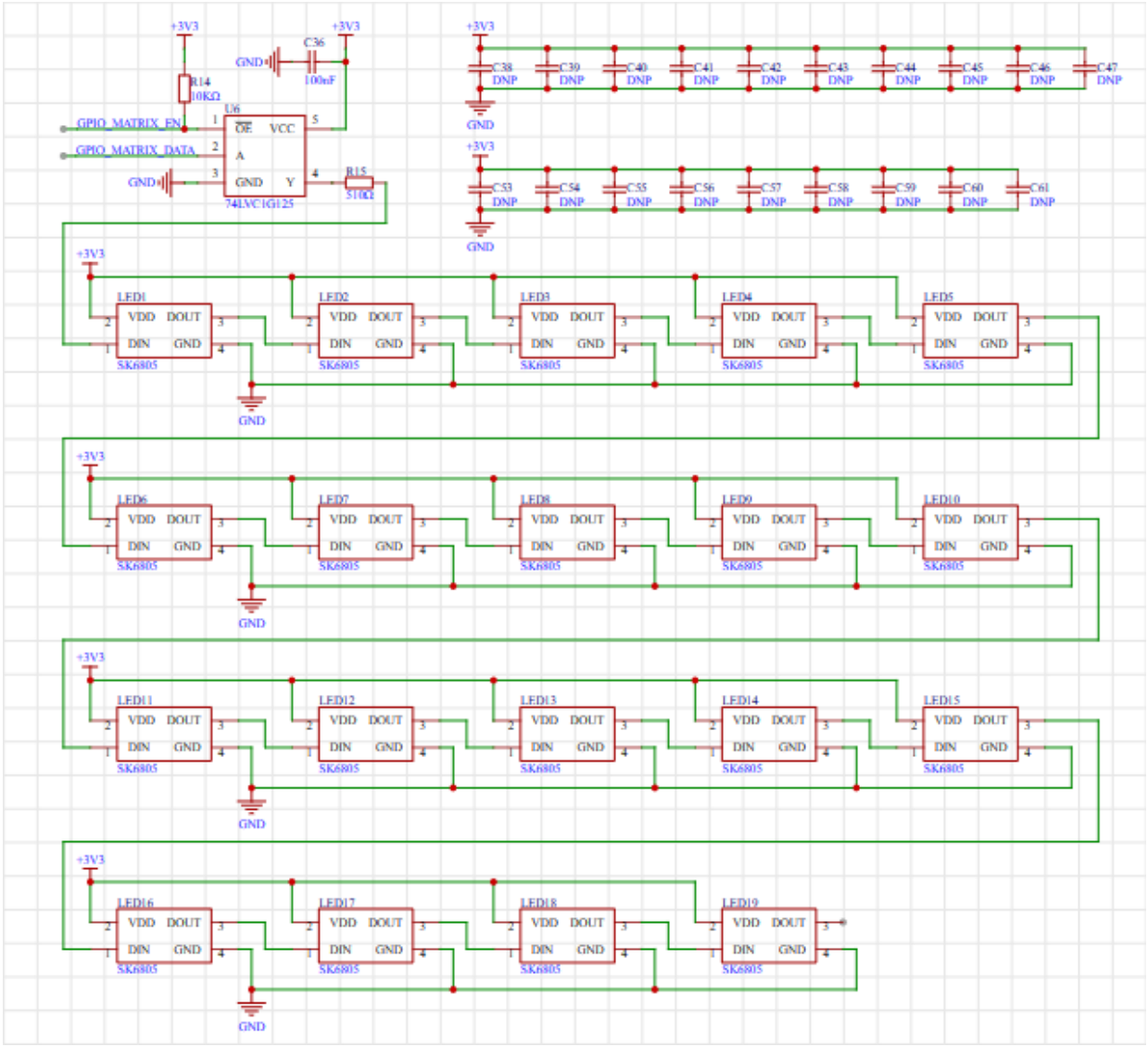

图 **32.1:** *LED 矩阵引脚图*

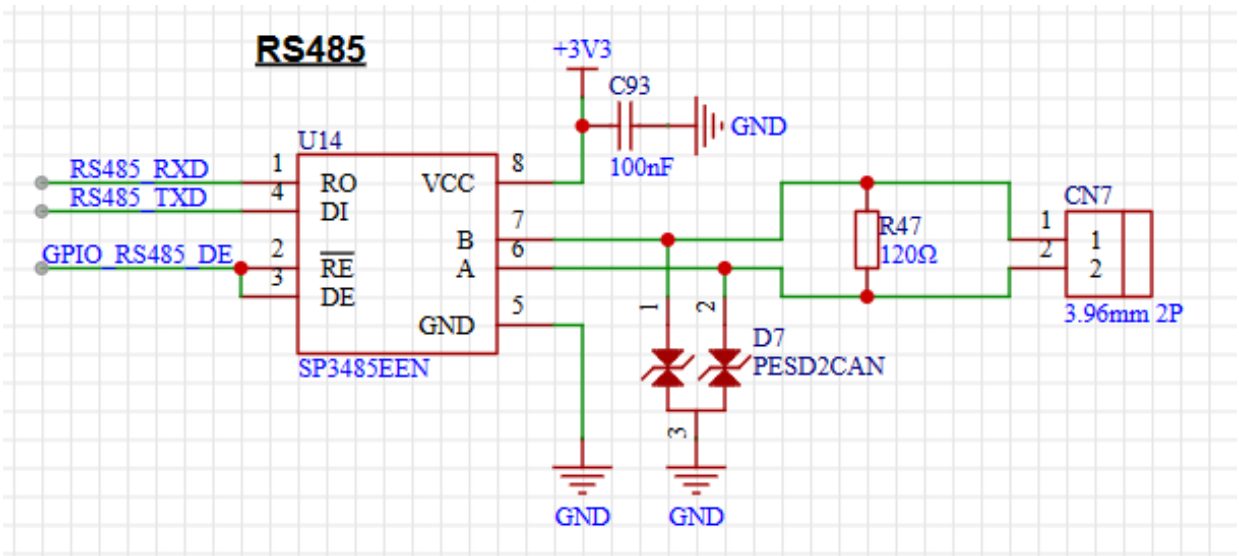

图 **32.2:** *RS485 电路原理图*

LED 矩阵、RS485 接口在开发板上的位置如下：

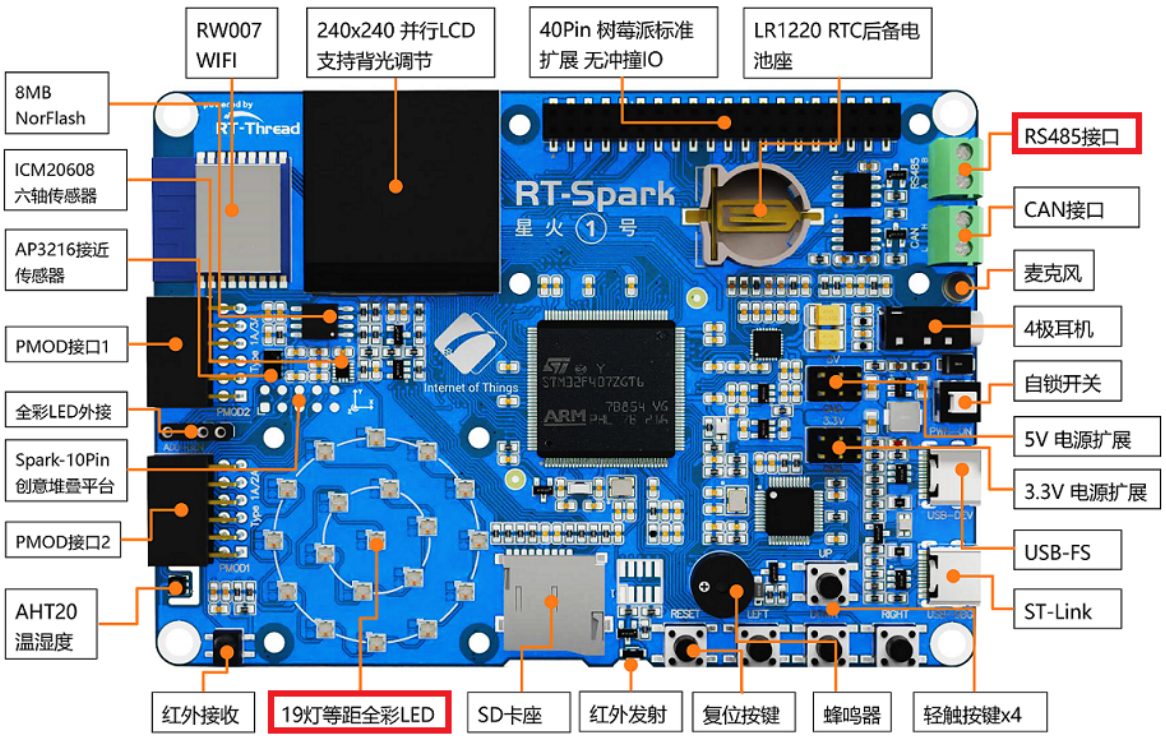

图 32.3: *LED 矩阵在开发板上的位置*

32.3 软件说明

本例程的源码位于 `/projects/06_demo_rs485_led_matrix`。

32.3.1 rs485 驱动代码

rs485 驱动代码在 `libraries/Board_Drivers/rs485/drv_rs485.c` 中：

```
int rs485_init(void)
{
    /* find uart device */
    rs485_serial = rt_device_find(RS485_UART_DEVICE_NAME);
    if (!rs485_serial)
    {
        rt_kprintf("find %s failed!\n", RS485_UART_DEVICE_NAME);
        return -RT_ERROR;
    }

    rt_device_open(rs485_serial, RT_DEVICE_FLAG_INT_RX);

    /* set receive data callback function */
```

```
    rt_device_set_rx_indicate(rs485_serial, rs485_input);

    /* set the send completion callback function */
    rt_device_set_tx_complete(rs485_serial, rs485_output);

    rt_pin_mode(BSP_RS485_RTS_PIN, PIN_MODE_OUTPUT);

    RS485_IN;

    rt_sem_init(&rs485_rx_sem, "rs485_rx_sem", 0, RT_IPC_FLAG_FIFO);
    /* create rs485 receive thread */
    // rt_thread_t thread = rt_thread_create("rs485", rs485_thread_entry, RT_NULL,
        1024, 25, 10);
    extern led_matrix_receieve_task(void *parameter);
    rt_thread_t thread = rt_thread_create("rs485", led_matrix_receieve_task, RT_NULL
        , 1024, 20, 10);

    if (thread != RT_NULL)
    {
        rt_thread_startup(thread);
    }
    else
    {
        return -RT_ERROR;
    }

    return RT_EOK;
}
```

使用串口 6 作为发送设备完成 485 通信。

led_matrix_receieve_task() 为接受的回调函数，收到数据之后会自动进入该回调函数。

32.3.2 led matrix 驱动程序

led matrix 驱动程序位于 libraries/Board_Drivers/led_matrix/drv_matrix_led.c 中

```
/**
 * @brief Set a pixel's color to PWM reload value.
 *
 * @param n Pixel index, starting from 0.
 * @param c pixel_rgb_t value, RGB888 format.
 */
void led_matrix_set_color(uint16_t n, pixel_rgb_t c)
```

使用以上接口即可设置某一个灯珠的颜色。然后调用 led_matrix_reflash() 刷新灯珠颜色。

32.3.3 4.3.3 主机驱动 LED Matrix

```
static void led_matrix_flowing_water_entry()
{
    while(1)
    {
        int base_time = 150;
        /* 分别变速转三圈 */
        for(int time = 1; time<=3; time++)
        {
            for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]);i++)
            {
                led_matrix[i].io_ctl(&led_matrix[i],RED);
                rt_thread_delay(base_time / time);
            }
            for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]);i++)
            {
                led_matrix[i].io_ctl(&led_matrix[i],GREEN);
                rt_thread_delay(base_time / time);
            }
            for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]);i++)
            {
                led_matrix[i].io_ctl(&led_matrix[i],BLUE);
                rt_thread_delay(base_time / time);
            }
            for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]);i++)
            {
                led_matrix[i].io_ctl(&led_matrix[i],WHITE);
                rt_thread_delay(base_time / time);
            }
            for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]);i++)
            {
                led_matrix[i].io_ctl(&led_matrix[i],DARK);
                rt_thread_delay(base_time / time);
            }
        }

        /* 同步转五圈 */
        int extern_led_index_start;
        for(extern_led_index_start = 0; (extern_led_index_start < sizeof(led_matrix)
            )\
        && (led_matrix[extern_led_index_start].location == INTERNAL);
            extern_led_index_start++);
        if(extern_led_index_start < sizeof(led_matrix))
        {
            for(int time = 1; time<=3; time++)
            {
                for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]); i++)
                {
                    if(i < extern_led_index_start)
```

```
                    {
                        led_matrix[i].io_ctl(&led_matrix[i],RED);
                    }
                    if(sizeof(led_matrix)-1-i >= extern_led_index_start)
                    {
                        led_matrix[sizeof(led_matrix)/sizeof(led_matrix[0])-1-i].
                            io_ctl(&led_matrix[sizeof(led_matrix)/sizeof(led_matrix
                            [0])-1-i],RED);
                    }
                    rt_thread_delay(base_time / time);
                }
                for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]); i++)
                {
                    if(i < extern_led_index_start)
                    {
                        led_matrix[i].io_ctl(&led_matrix[i],GREEN);
                    }
                    if(sizeof(led_matrix)-1-i >= extern_led_index_start)
                    {
                        led_matrix[sizeof(led_matrix)/sizeof(led_matrix[0])-1-i].
                            io_ctl(&led_matrix[sizeof(led_matrix)/sizeof(led_matrix
                            [0])-1-i],GREEN);
                    }
                    rt_thread_delay(base_time / time);
                }
                for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]); i++)
                {
                    if(i < extern_led_index_start)
                    {
                        led_matrix[i].io_ctl(&led_matrix[i],BLUE);
                    }
                    if(sizeof(led_matrix)-1-i >= extern_led_index_start)
                    {
                        led_matrix[sizeof(led_matrix)/sizeof(led_matrix[0])-1-i].
                            io_ctl(&led_matrix[sizeof(led_matrix)/sizeof(led_matrix
                            [0])-1-i],BLUE);
                    }
                    rt_thread_delay(base_time / time);
                }
                for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]); i++)
                {
                    if(i < extern_led_index_start)
                    {
                        led_matrix[i].io_ctl(&led_matrix[i],WHITE);
                    }
                    if(sizeof(led_matrix)-1-i >= extern_led_index_start)
                    {
                        led_matrix[sizeof(led_matrix)/sizeof(led_matrix[0])-1-i].
                            io_ctl(&led_matrix[sizeof(led_matrix)/sizeof(led_matrix
```

```
                            [0])-1-i],WHITE);
                    }
                    rt_thread_delay(base_time / time);
                }
                for(int i=0; i<sizeof(led_matrix)/sizeof(led_matrix[0]); i++)
                {
                    if(i < extern_led_index_start)
                    {
                        led_matrix[i].io_ctl(&led_matrix[i],DARK);
                    }
                    if(sizeof(led_matrix)-1-i >= extern_led_index_start)
                    {
                        led_matrix[sizeof(led_matrix)/sizeof(led_matrix[0])-1-i].
                            io_ctl(&led_matrix[sizeof(led_matrix)/sizeof(led_matrix
                            [0])-1-i],DARK);
                    }
                    rt_thread_delay(base_time / time);
                }
            }
        }

    }
}
```

以上为循环控制 LED 的程序，这里需要说明一下，代码采用了函数指针的写法，将外部 LED 矩阵和内部 LED 矩阵抽象成同一种操作方法，使用起来比较方便。

```
static void intern_led_control(rt_led_node_t *node, pixel_rgb_t color)
{
    led_matrix_set_color(node->pin, color);
    led_matrix_reflash();
    node->status = color;
}

static void extern_led_control(rt_led_node_t *node, pixel_rgb_t color)
{
    rs485_send_buf[0] = 0xA5;
    rs485_send_buf[1] = (node->pin)&0xff;
    *(pixel_rgb_t*)(&(rs485_send_buf[2])) = color;
    rs485_send_buf[5] = 0;
    rs485_send_buf[6] = 0;
    rs485_send_buf[7] = 0xA6;
    rs485_send_data((char *)rs485_send_buf, 8);

    node->status = color;
}
```

可以看出，点亮外部 led 矩阵使用 485 发送函数，点亮内部 led 矩阵使用 led 驱动接口。

32.3.4 4.3.3 从机驱动 LED Matrix

```
static rt_led_node_t led_matrix[] = {
    {EXTERN_LED_0, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_1, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_2, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_3, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_4, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_5, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_6, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_7, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_8, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_9, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_10, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_11, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_12, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_13, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_14, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_15, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_16, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_17, {0, 0, 0}, INTERNAL, RT_NULL},\
    {EXTERN_LED_18, {0, 0, 0}, INTERNAL, RT_NULL},
};

/* 接受缓冲区 */
static rt_uint8_t rs485_receive_buf1[8];
static rt_uint8_t rs485_receive_buf2[8];
static rt_uint8_t *buff = rs485_receive_buf2;
static rt_uint8_t *last_buff = rs485_receive_buf1;

static rt_sem_t rs_485_receive_ready = RT_NULL;

static void intern_led_control(rt_led_node_t *node, pixel_rgb_t color)
{
    led_matrix_set_color(node->pin, color);
    led_matrix_reflash();
    node->status = color;
}

void led_matrix_receieve_task(void *parameter) /* 接收 RS485 发送过来的数据包的任务
    */
{
    /* buff 指针，接收采用双 buff */
    static uint8_t index;
    char ch = 0;
    while (1)
    {
        /* A byte of data is read from a rs485_serial port, and if it is not read,
            it waits for the received semaphore */
```

```
        while (rt_device_read(rs485_serial, -1, &ch, 1) != 1)
        {
            rt_sem_take(&rs485_rx_sem, RT_WAITING_FOREVER);
        }
        if(ch == 0xA6)
        {
            buff[index] = ch;
            index = 0;
            if(buff == rs485_receive_buf1)
            {
                buff = rs485_receive_buf2; /* switch buffer pointers */
                last_buff = rs485_receive_buf1;
            }else if(buff == rs485_receive_buf2)
            {
                buff = rs485_receive_buf1; /* switch buffer pointers */
                last_buff = rs485_receive_buf2;
            }
        rt_kprintf("\n receive:");
        if(buff == rs485_receive_buf1)
        {
            rt_kprintf("rs485_receive_buf1:");
        }else if(buff == rs485_receive_buf2)
        {
            rt_kprintf("rs485_receive_buf2:");
        }
        for(int i =0;i<8;i++)
        {
            rt_kprintf("%d:%x",i,buff[i]);
        }
        rt_uint8_t led_no = buff[1];
        rt_kprintf("\nindex:%d",led_no);
        rt_sem_release(rs_485_receive_ready);

        }else
        {
            buff[index++] = ch;
        }
    }
}

void led_matrix_slave_task_entry()
{
    while(1)
    {
        rt_sem_take(rs_485_receive_ready, RT_WAITING_FOREVER);
        rt_uint8_t led_no = last_buff[1];
        if(led_no<0 || led_no>= sizeof(led_matrix)/sizeof(led_matrix[0]))
        {
            continue;
```

```
        }
        pixel_rgb_t color = *(pixel_rgb_t *)(&(last_buff[2]));
        led_matrix[led_no].io_ctl(&led_matrix[led_no],color);
    }
}
```

接受完一帧数据之后会释放信号量通知控制线程控制 led 矩阵。

32.4 运行

32.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

32.4.2 运行效果

使用双绞线连接两个开发板，一个运行主机程序一个运行从机程序会看到两个板子上的 LED 矩阵按照预先设定的逻辑闪烁。主机使用下面的命令启动

```
led_matrix_flowing_water_example
```

从机使用下面的命令启动

```
led_matrix_slave_example
```

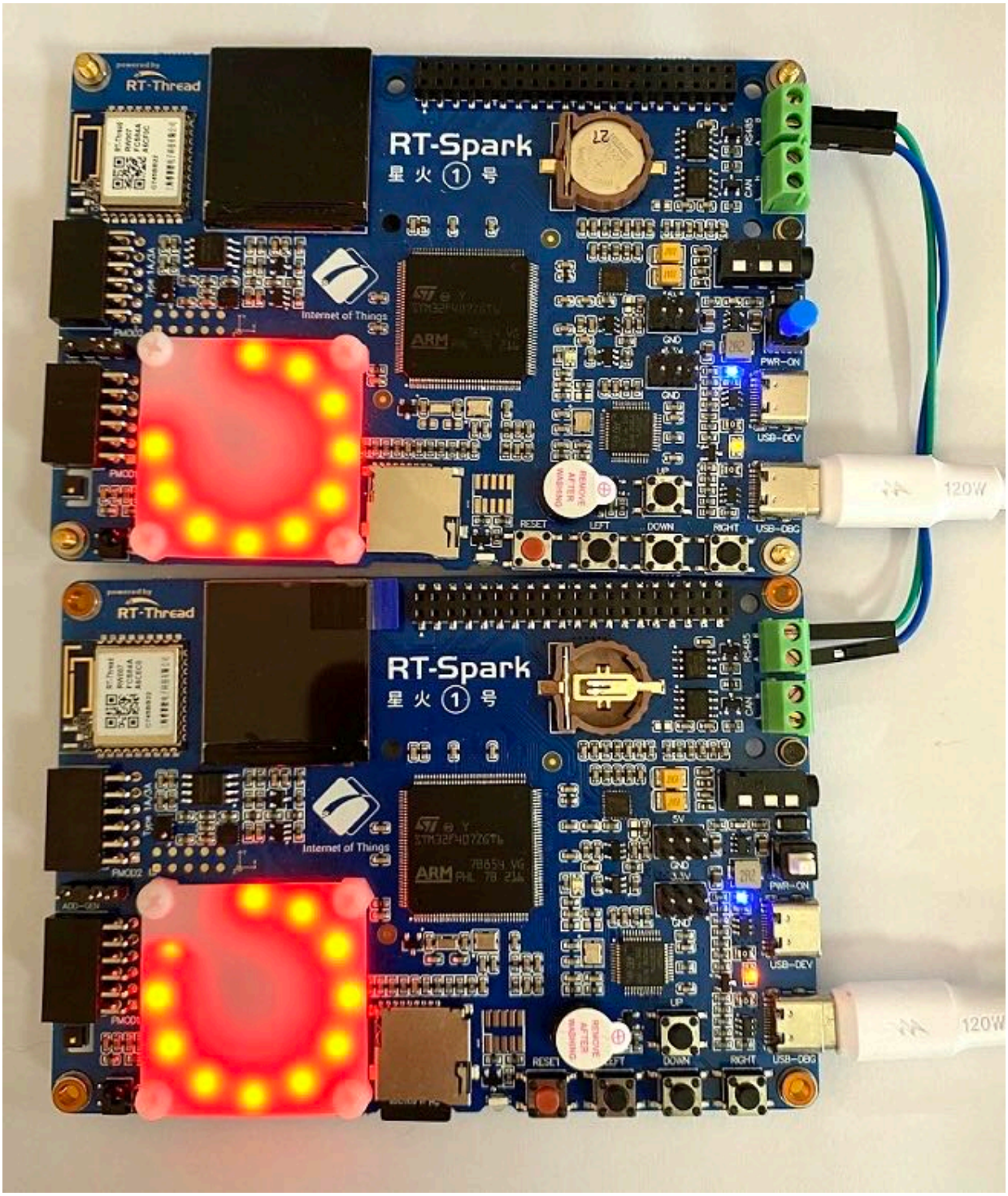

图 32.4: *两块开发板上的 LED Matrix 同时闪烁*

32.5 注意事项

在实施中，请先使用双绞线连接开发板。

32.6 引用参考

- 文档中心：RT-Thread 文档中心

第 33 章

nes 模拟器实验

33.1 简介

本例程主要介绍了在星火 1 号开发板上移植 nes 模拟器的方法。

33.2 硬件说明

本例程使用到了 LCD 显示 nes 模拟器。使用了按键作为 nes 模拟器的输入设备。使用 sd 卡来存储游戏。

33.3 软件说明

本例程的源码位于 `/projects/06_demo_nes_simulator`。nes 的移植代码主要在 application/nes 文件夹下。

其中 application/nes/port/nes_port.c 为移植的内容

```
int nes_draw(size_t x1, size_t y1, size_t x2, size_t y2, nes_color_t* color_data){

    extern void lcd_fill_array(rt_uint16_t x_start, rt_uint16_t y_start, rt_uint16_t
        x_end, rt_uint16_t y_end, void *pcolor);
    lcd_fill_array(x1, y1, x2, y2, color_data);
    return 0;
}
```

上面的函数实现了屏幕显示。

```
static void update_joypad()
{
    rt_uint32_t key_data = 0;
    static_nes.nes_cpu.joypad.joypad = 0;
    if(rt_event_recv(key_event,TP_UP | TP_DOWN | TP_LEFT | TP_RIGHT | TP_SELECT \
    | TP_START | TP_A | TP_B,RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,RT_WAITING_NO,\
```

```
    &key_data) == RT_EOK)
    {
        if(key_data&TP_UP)      static_nes.nes_cpu.joypad.U1 = 1;
        else                    static_nes.nes_cpu.joypad.U1 = 0;
        if(key_data&TP_DOWN)    static_nes.nes_cpu.joypad.D1 = 1;
        else                    static_nes.nes_cpu.joypad.D1 = 0;
        if(key_data&TP_LEFT)    static_nes.nes_cpu.joypad.L1 = 1;
        else                    static_nes.nes_cpu.joypad.L1 = 0;
        if(key_data&TP_RIGHT)   static_nes.nes_cpu.joypad.R1 = 1;
        else                    static_nes.nes_cpu.joypad.R1 = 0;
        if(key_data&TP_SELECT)  static_nes.nes_cpu.joypad.SE1 = 1;
        else                    static_nes.nes_cpu.joypad.SE1 = 0;
        if(key_data&TP_START)   static_nes.nes_cpu.joypad.ST1 = 1;
        else                    static_nes.nes_cpu.joypad.ST1 = 0;
        if(key_data&TP_A)       static_nes.nes_cpu.joypad.A1 = 1;
        else                    static_nes.nes_cpu.joypad.A1 = 0;
        if(key_data&TP_B)       static_nes.nes_cpu.joypad.B1 = 1;
        else                    static_nes.nes_cpu.joypad.B1 = 0;
    }
}
```

上面函数实现了接收从按键线程发送过来的按键事件。

application/nes/port/rt_key_scan.c 为按键捕获函数

```
/* 配置 KEY 输入引脚  */
#define PIN_KEY0        GET_PIN(C, 0)     // PC0:  KEY0         --> KEY
#define PIN_KEY1        GET_PIN(C, 1)      // PC1 :  KEY1         --> KEY
#define PIN_KEY2        GET_PIN(C, 4)      // PC4 :  KEY2         --> KEY
#define PIN_WK_UP       GET_PIN(C, 5)     // PC5:  WK_UP        --> KEY

rt_event_t key_event;

static void key_scan()
{
    uint32_t key_data;
    while(1)
    {
        rt_thread_mdelay(15); //10ms tick wait for input change.
        key_data = 0;
        if (rt_pin_read(PIN_KEY0) == PIN_LOW)
        {

            rt_kprintf("KEY0 pressed!\n");

            key_data |= TP_LEFT;
        }
        if (rt_pin_read(PIN_KEY1) == PIN_LOW)
        {
```

```
            rt_kprintf("KEY1 pressed!\n");

            key_data |= TP_START;
        }
        if (rt_pin_read(PIN_KEY2) == PIN_LOW)
        {

            rt_kprintf("KEY2 pressed!\n");

            key_data |= TP_RIGHT;
        }
        if (rt_pin_read(PIN_WK_UP) == PIN_LOW)
        {

            rt_kprintf("WK_UP pressed!\n");

            key_data |= TP_A;
        }
        if(key_data)
        {
            rt_event_send(key_event,key_data);
        }
    }

}
void key_scan_init()
{

    rt_pin_mode(PIN_KEY0, PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(PIN_KEY1, PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(PIN_KEY2, PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(PIN_WK_UP, PIN_MODE_INPUT_PULLUP);

    key_event = rt_event_create("key", RT_IPC_FLAG_FIFO);

    rt_thread_t thread = rt_thread_create("key scan",key_scan,RT_NULL,1024,20,1);
    if(thread == RT_NULL) rt_kprintf("key scan thread create failed!\n");
    rt_thread_startup(thread);
}
```

此文件初始化了一个按键扫描线程，一旦捕获到按键按下就会发送按键事件。

```
#define PIN_LED_R              GET_PIN(F, 12)      // PE7 :  LED_R        --> LED

#include "nes.h"

static void nes_thread_entry(void *parameter)
{
    nes_t* nes = (nes_t*)parameter;
```

```
    nes_run(nes);
    nes_unload_file(nes);
}
static int nes_start(int argc, char *argv[]){

    if (argc == 2)
    {
        const char *nes_file_path = argv[1];
        size_t nes_file_path_len = strlen(nes_file_path);
        if (memcmp(nes_file_path+nes_file_path_len-4,".nes",4)==0 || memcmp(
            nes_file_path+nes_file_path_len-4,".NES",4)==0)
        {
            nes_printf("nes_file_path:%s\n",nes_file_path);
            nes_t* nes = nes_load_file(nes_file_path);
            if (!nes)
            {
                return -1;
            }
            rt_thread_t thread = rt_thread_create("nes", nes_thread_entry, nes,
                2048, 20, 10);
            if(thread == RT_NULL)
            {
                rt_kprintf("Can't create nes thread!\n");
                return -1;
            }
            rt_thread_startup(thread);
            return 0;
        }else
        {
            nes_printf("Please enter xxx.nes\n");
            return -1;
        }
    }else
    {
        nes_printf("Please enter the nes file path\n");
        return -1;
    }
}

int main()
{
    rt_pin_mode(PIN_LED_R,PIN_MODE_OUTPUT);
    while (1)
    {
        rt_pin_write(PIN_LED_R, PIN_LOW);
        rt_thread_mdelay(500);

        rt_pin_write(PIN_LED_R, PIN_HIGH);
        rt_thread_mdelay(500);
```

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

主函数完成的主要功能为创建游戏机线程函数，并且将函数导入到 msh 命令行。然后周期点亮 led 提示系统运行状态。

33.4 运行

33.4.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 STM32F407-RT-SPARK 资源包，然后创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

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

33.4.2 运行效果

首先将 game 文件夹下的以 .nes 为后缀的游戏拷贝到内存卡。按下复位按键重启开发板，在 msh 中输入启动命令可以观察到游戏已经启动。按下按键，模拟器均有相应反馈。

```
 \ | /
- RT -     Thread Operating System
 / | \     4.1.1 build Jul 19 2023 10:08:16
 2006 - 2022 Copyright by RT-Thread team
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/SDIO] SD card capacity 1967104 KB.
[I/drv.lcd]  LCD ID:81b3
[D/drv.lcd] backlight 80 percent
msh />[I/app.filesystem] SD card mount to '/sdcard'
msh />nes_start sdcard/SuperMary.nes
```

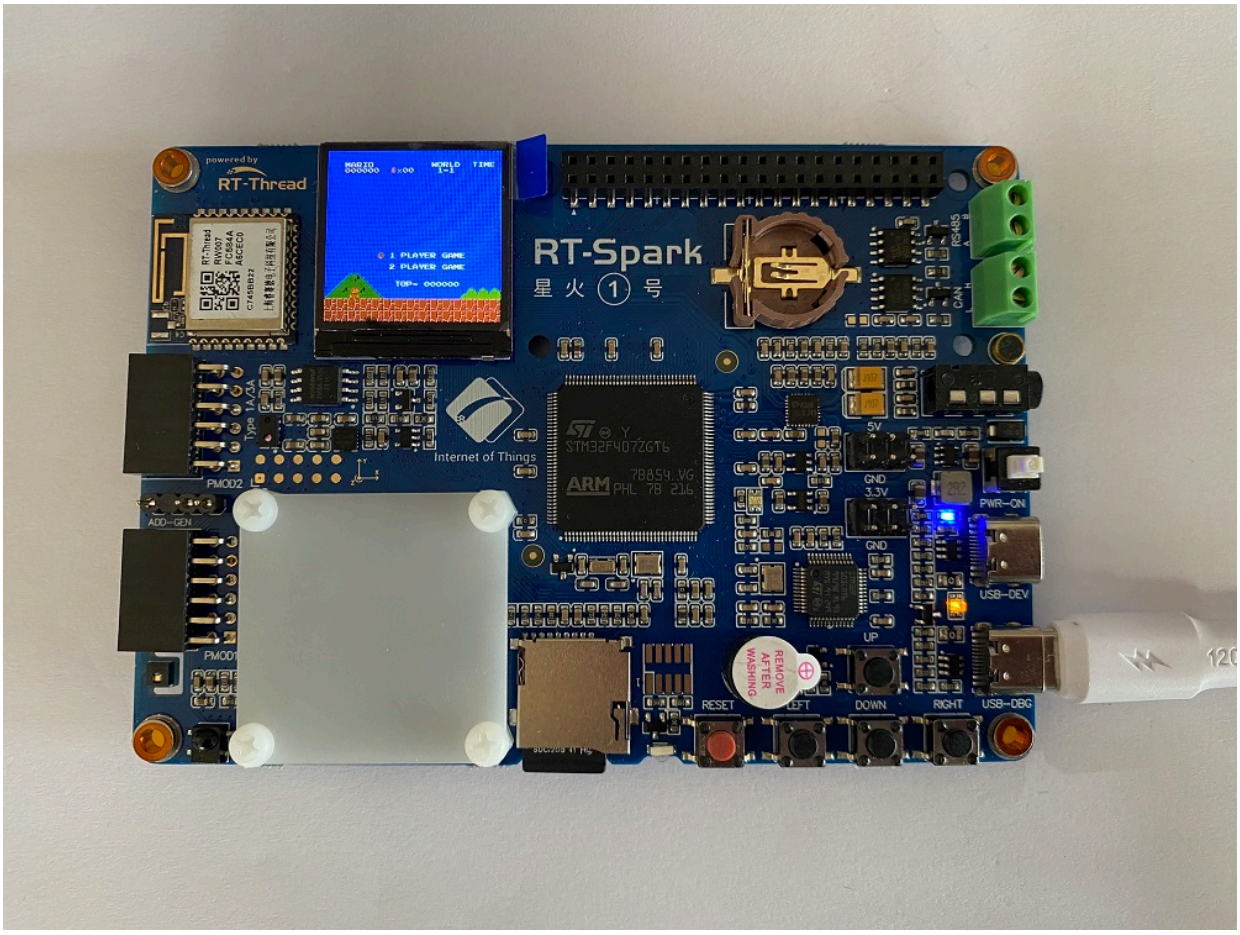

图 33.1: *nes 模拟器*

33.5 注意事项

按键映射关系为 up 为跳跃，down 为开始/暂停，左右为方向。

33.6 引用参考

- 文档中心：RT-Thread 文档中心

第 34 章

出厂综合例程

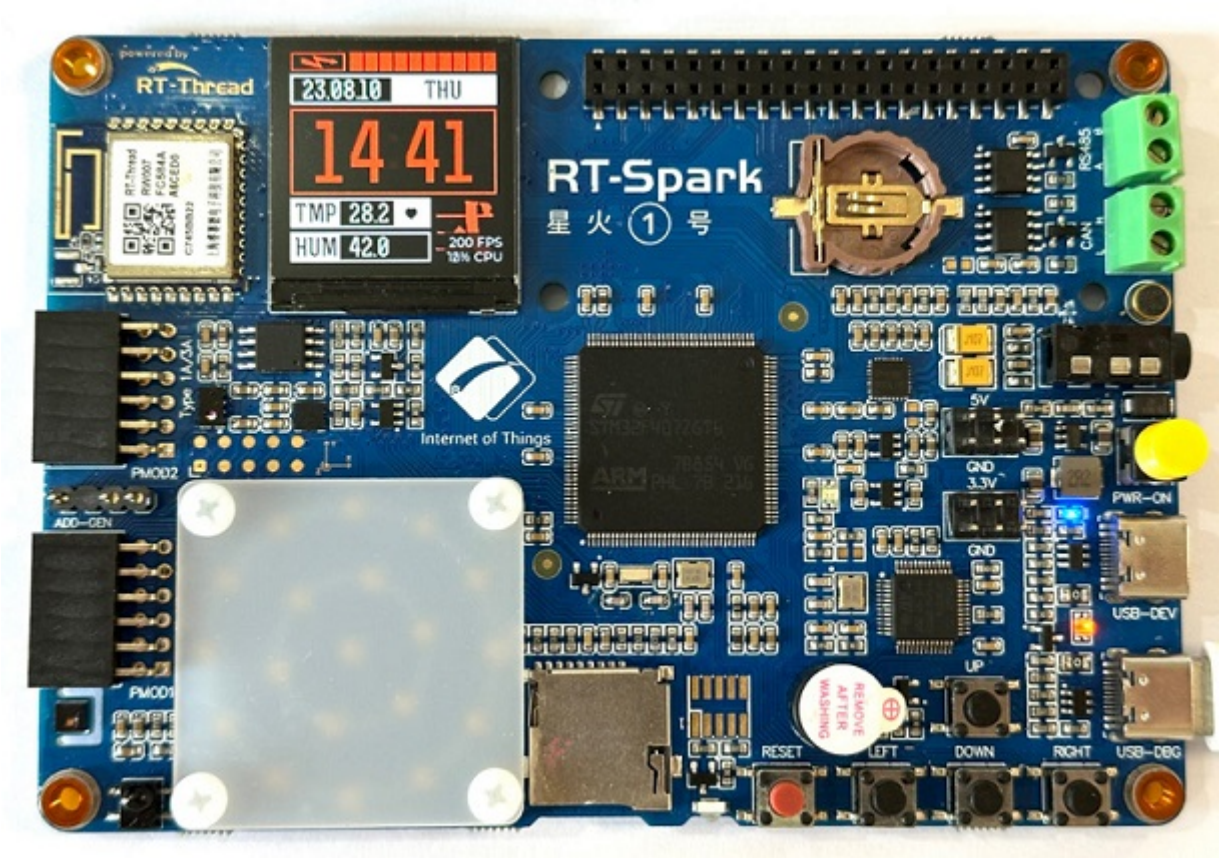

图 34.1: *demo*

34.1 硬件说明

本例程将使用 LCD 显示屏显示 LVGL 的图像。

34.2 软件说明

本例程的源码位于 `/projects/06_demo_factory`。

本例程是基于 LVGL 的个性表盘显示例程，附带了对整板传感器的自动测试界面。

lvgl 相关代码在 packages/LVGL-v8.3.5 目录，关于更多请参考 lvgl 开发文档。lvgl 的移植代码在 application/lvgl 目录下面。

34.3 运行

34.3.1 编译 & 下载

- RT-Thread Studio：在 RT-Thread Studio 的包管理器中下载 `STM32F407-RT-SPARK` 资源包，创建新工程，执行编译。
- MDK：首先双击 mklinks.bat，生成 rt-thread 与 libraries 文件夹链接；再使用 Env 生成 MDK5 工程；最后双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，将开发板的 ST-Link USB 口与 PC 机连接，将固件下载至开发板。

34.3.2 运行效果

按下复位键重启开发板，观察 lcd 屏幕上已经开始运行 lvgl 的 demo。

上电运行，屏幕首先会显示表盘主界面，可以按 `LEFT` 按键执行自动测试。

在除自动测试界面之外，其他界面都是由 `UP` 和 `DOWN` 按键执行增减，`RIGHT` 按键执行下一步，`LEFT` 按键执行返回上一级。

本 demo 自带如下应用展示：

- 系统文件夹展示
- 板载 LED 矩阵手动调光
- 环境检测（环境亮度、温湿度）
- 6 轴传感器数值反馈
- 表盘时间设置

在进入应用后，如果需要退出到菜单页面，部分应用需要按两次 `LEFT` 以退出当前页面。

34.4 注意事项

暂无

34.5 引用参考

- 文档中心：RT-Thread 文档中心
- lvgl 软件包：`https://github.com/lvgl/lvgl`