RT-THREAD W60X SDK 开发手册

RT-THREAD 文档中心

WWW.RT-THREAD.ORG

Thursday 12th September, 2019

目录

第 1 章

前言

开发手册包含两部分内容，包括实验平台硬件 W601 IoT Board 的介绍和基于该平台的丰富示例。

第 2 章

硬件篇

本篇将详细介绍 RT-Thread 联合联盛德与正点原子共同制作的 W601 IoT Board 开发板。

2.1 开发板整体介绍

2.1.1 W601 芯片介绍

W601 芯片架构如下图所示：

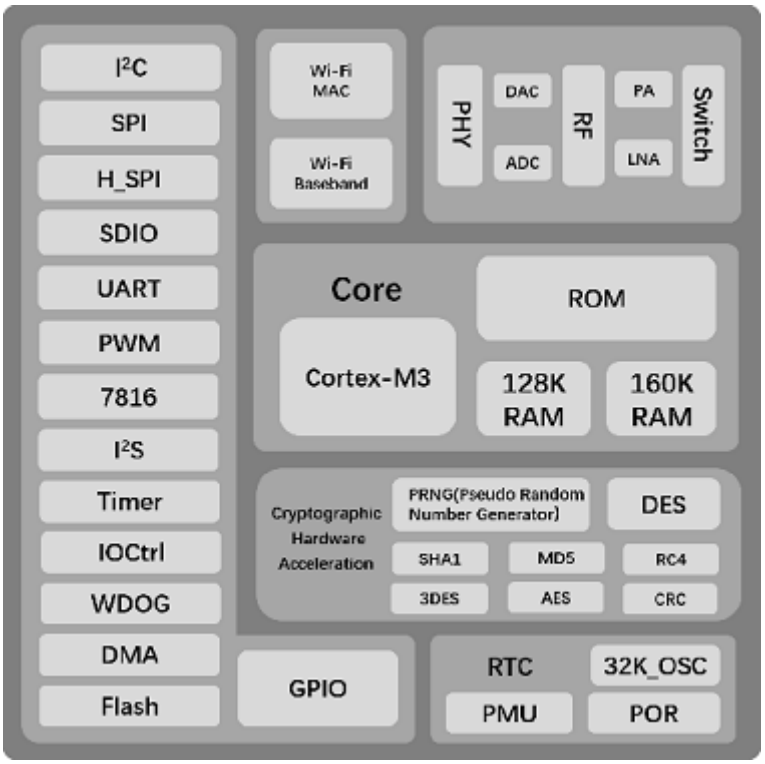


图 2.1: *W601 芯片架构*

W601 芯片资源如下：

- 芯片集成度
  - 集成 32 位嵌入式 Cortex-M3 处理器
  - 集成 288KByte 数据存储器
  - 集成 1MByte Flash
  - 集成 2.4G 射频收发器，满足 IEEE802.11 规范
  - 集成 PA/LNA/TR-Switch
  - 集成 32.768KHz 时钟振荡器
  - 集成电源管理电路
  - 集成通用加密硬件加速器，支持 PRNG(Pseudo random Number Generator)/ SHA1/ MD5/ RC4/ DES/ 3DES/ AES/ CRC 等多种加解密协议
  - 支持 3.3V 单电源供电
  - 集成 PS-Poll、U-APSD 低功耗管理
- 芯片接口
  - 集成 1 个 SDIO2.0 Device 控制器，支持 SDIO1 位/4 位/SPI 三种操作模式；工作时钟范围 0~50MHz
  - 集成 2 个 UART 接口，支持 RTS/CTS，波特率范围 38Kbps~2Mbps
  - 集成 1 个高速 SPI 设备控制器，工作时钟范围 0~50MHz
  - 集成 1 个 I²C 控制器，支持 100/400Kbps 速率
  - 集成 GPIO 控制器
  - 集成 PWM 控制器，支持 5 路 PWM 单独输出或者 2 路 PWM 输入
  - 集成双工 I²S 控制器，支持 32KHz 到 192KHz I²S 接口编解码
  - 集成 7816 接口，支持 ISO-7816-3 T=0/1 模式，支持 EVM2000 规范，并兼容串口功能
- 协议与功能
  - 支持 GB15629.11-2006、IEEE802.11 b/g/n/e/i/d/k/r/s/w
  - 支持 WAPI2.0
  - 支持 Wi-Fi WMM/WMM-PS/WPA/WPA2/WPS
  - 支持 Wi-Fi Direct
  - 支持 EDCA 信道接入方式
  - 支持 20/40M 带宽工作模式
  - 支持 AMPDU、AMSDU
  - 支持 IEEE802.11n MCS 0~7、MCS32 物理层传输速率档位，传输速率最高到 150Mbps
  - 支持 STA/AP/AP+STA 功能
  - 支持监听功能

2.1.2 W601 IoT Board 介绍

W601 IoT Board 主要资源如下图所示：

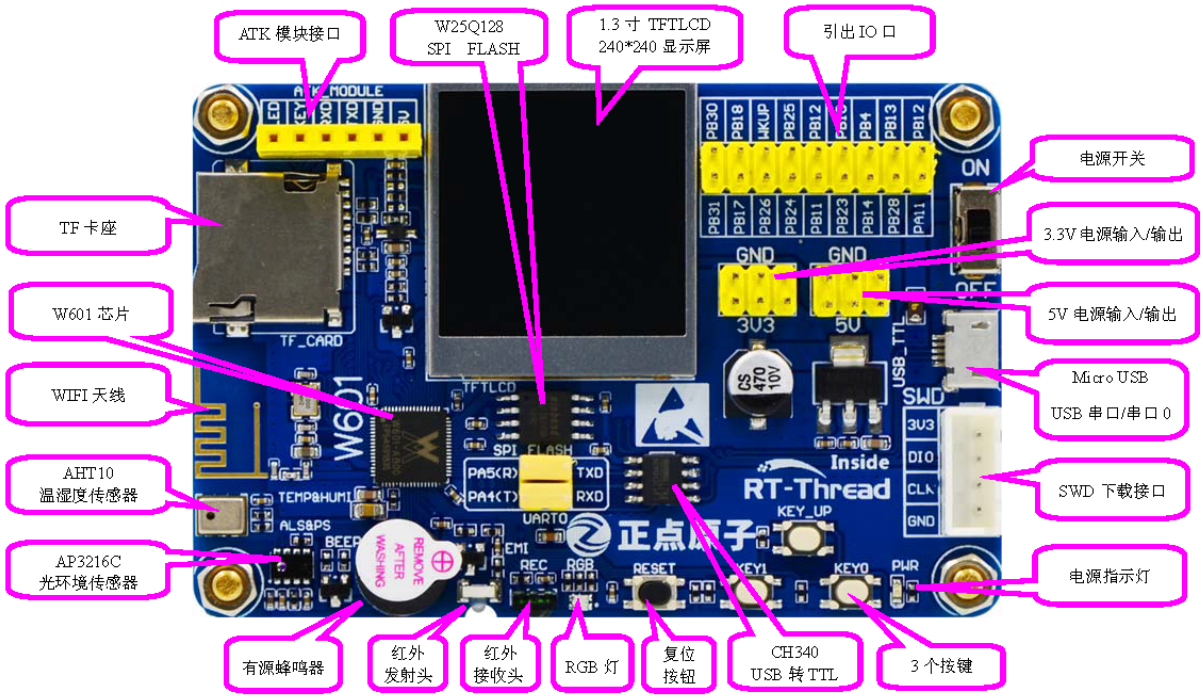


图 2.2: *W601 IoT Board*

从图中可以看出，W601 IoT Board 资源丰富，包括温湿度传感器、接近感应传感器与 LCD 等外设，非常适合初学者学习以及物联网开发者使用。

开发板的特点包括：

- 小巧精致。开发板布局紧凑，丝印清晰美观，方便携带，可随时享受开发的乐趣。
- 接口丰富。可进行大量实验，DIY 喜欢的产品。
- 简单易用。开发板仅一根 micro USB 线即可实现供电、日志输出等功能。并且板载 WiFi 天线，方便接入网络。

2.2 开发板资源说明

对照板载资源图，按逆时针的顺序依次介绍。

2.2.1 TF 卡接口

这是开发板板载的一个标准 TF 卡接口（TF_CARD），采用 SPI 方式驱动，有了这个 TF 卡接口，就可以满足大量数据存储的需求。

2.2.2 W601

W601 是一款支持多接口、多协议的无线局域网 802.11n（1T1R）的 SOC 芯片。该 SOC 芯片集成射频收发前端 RF Transceiver，CMOS PA 功率放大器，基带处理器/媒体访问控制，SDIO、SPI、UART、GPIO 等接口的低功耗 WLAN 芯片。

2.2.3 WIFI 天线

这是开发板板载的一个 WIFI 天线（ANT），可以直接作为 WIFI 的天线使用。

2.2.4 AHT10 温湿度传感器

这是开发板板载的一个温湿度传感器（U7），它集成了温度传感器和湿度传感器的功能，可以用于环境监测等场景，该芯片使用 IIC 通讯。

2.2.5 AP3216 光环境传感器

这是开发板板载的一个光环境三合一传感器（U8），它可以作为：环境光传感器、近距离（接近）传感器和红外传感器。通过该传感器，开发板可以感知周围环境光线的变化，接近距离等，从而可以实现类似手机的自动背光控制。

2.2.6 有源蜂鸣器

这是开发板的板载蜂鸣器（BEEP），可以实现简单的报警/闹铃，让开发板可以发声。

2.2.7 红外发射头

这是开发板板载的红外发射头（IR1），可以实现红外发射功能，使用这个发射头，我们可以模拟红外遥控器的功能。

2.2.8 红外接收头

这是开发板的红外接收头（IR2），可以实现红外接收功能，通过这个接收头，可以接收市面常见的各种遥控器的红外信号，大家甚至可以自己实现万能红外解码。当然，如果应用得当，该接收头也可以用来传输数据。

配备了一个小巧的红外遥控器，外观如红外遥控器图所示：

图 2.3: *红外遥控器*

2.2.9 RGB 灯

这是开发板板载的一个 RGB 灯，通过 R（红）、G（绿）和 B（蓝）三种颜色的组合我们可以实现各种不同的颜色。

2.2.10 复位按钮

这是开发板板载的复位按键（RESET），用于复位开发板上的主芯片。

2.2.11 CH340 USB 转 TTL

串口转 USB 芯片，连接串口 UART0。可以通过跳帽与芯片引脚断开。

2.2.12 3 个按键

这是开发板板载的 3 个轻触按键（KEY0、KEY1 和 WK_UP），可以用于人机交互的输入。其中 WK_UP 是高电平有效，而 KEY0、KEY1 是低电平有效。

2.2.13 电源指示灯

这是开发板板载的一颗蓝色的 LED 灯（PWR），用于指示电源状态。在电源开启的时候（通过板上的电源开关控制），该灯会亮，否则不亮。通过这个 LED，可以判断开发板的上电情况。

2.2.14 SWD 接口

使用 J-Link 工具连接后，可以用于程序下载与调试。

2.2.15 USB 串口/串口 0

连接 CH340 芯片，跳帽连接之后，可通过该 USB 口输出日志。

2.2.16 5V 电源输入/输出

这是开发板板载的一组 5V 电源输入输出排针（2*3）（VOUT1），该排针用于给外部提供 5V 的电源，也可以用于从外部接 5V 的电源给板子供电。

2.2.17 3.3V 电源输入/输出

这是开发板板载的一组 3.3V 电源输入输出排针（2*3）（VOUT2），用于给外部提供 3.3V 的电源，也可以用于从外部接 3.3V 的电源给板子供电（最大电流不能超过 500mA）。

2.2.18 电源开关

这是开发板板载的电源开关（K1）。该开关用于控制整个开发板的供电，如果切断，则整个开发板都将断电，电源指示灯（PWR）会随着此开关的状态而亮灭。

2.2.19 引出 IO 口

这是开发板 IO 引出端口（J1），总共引出 20 个 IO 口，供大家外接使用。

2.2.20 TFTLCD 显示屏

这是开发板板载的一个 TFTLCD 显示屏（LCD_TFT），它是一个 1.3 寸 240*240 超高分辨率的显示屏，支持 16 位真彩色显示。

2.2.21 SPI FLASH

这是开发板外扩的 SPI FLASH 芯片（U9），型号为：W25Q128，容量为 128Mbit，即 16M 字节，可用于存储字库和其他用户数据，满足大容量数据存储要求。

第 3 章

软件篇

在阅读过上面的硬件篇后，相信大家已经对开发板的硬件资源有了较为深入的了解，接下来我们将介绍 RT-Thread W60X SDK 的软件资源。

当前 RT-Thread W60X SDK 例程数量多达 30 个以上的应用例程，并且有一定的代表性，每个例程都有非常详细的注释，代码风格统一，按照基本例程到高级例程的方式编排，方便初学者由浅入深逐步学习。这些例程分为四个类别：基本类、驱动类、组件类和物联网类。不仅包括了硬件资源的使用，更是提供了丰富的物联网领域的应用示例，帮助物联网开发者更好更快的进行开发。

最新版的 SDK 可以通过 GitHub 仓库：https://github.com/RT-Thread/W601_IoT_Board 来获取，后续将会在该仓库中添加更多丰富的例程，大家可以关注该仓库随时获取最新内容。

通过对这些例程的学习，你将了解到：

- 开发板硬件资源的使用方法
- 如何使用 RT-Thread 设备驱动框架
- 如何利用 RT-Thread 组件进行 Flash 管理
- 如何使用丰富的 IoT 软件包进行物联网应用开发

如果想要深入学习例程的实现原理，可以参考每个例程**引用参考**章节提供的阅读材料。通过阅读这些参考资料，可以让你对例程的使用更加得心应手。

第 4 章

LED 闪烁例程

4.1 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。

4.2 硬件说明

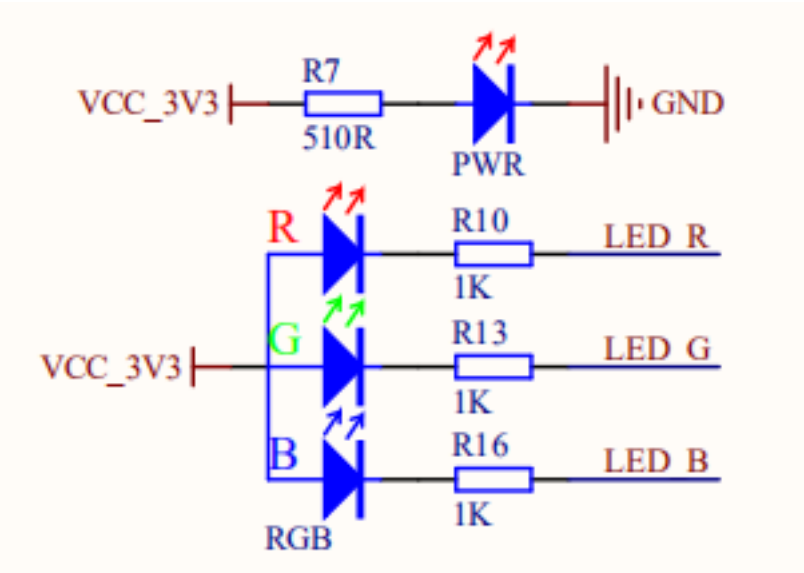


图 4.1: *LED 电路原理图*

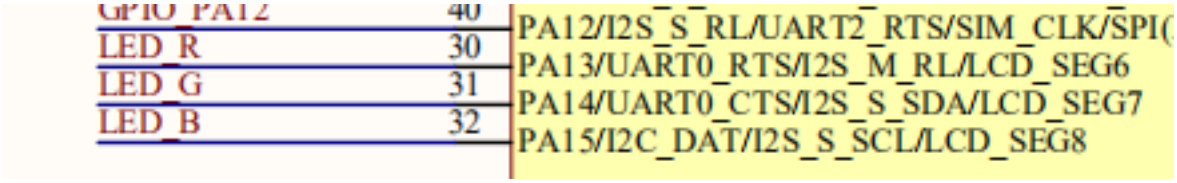


图 4.2: *LED 电路原理图*

如上图所示，RBG-LED 属于共阳 LED，阴极连接单片机的 30,31,32 号引脚上，其中红色 LED 对应 30 号引脚。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：

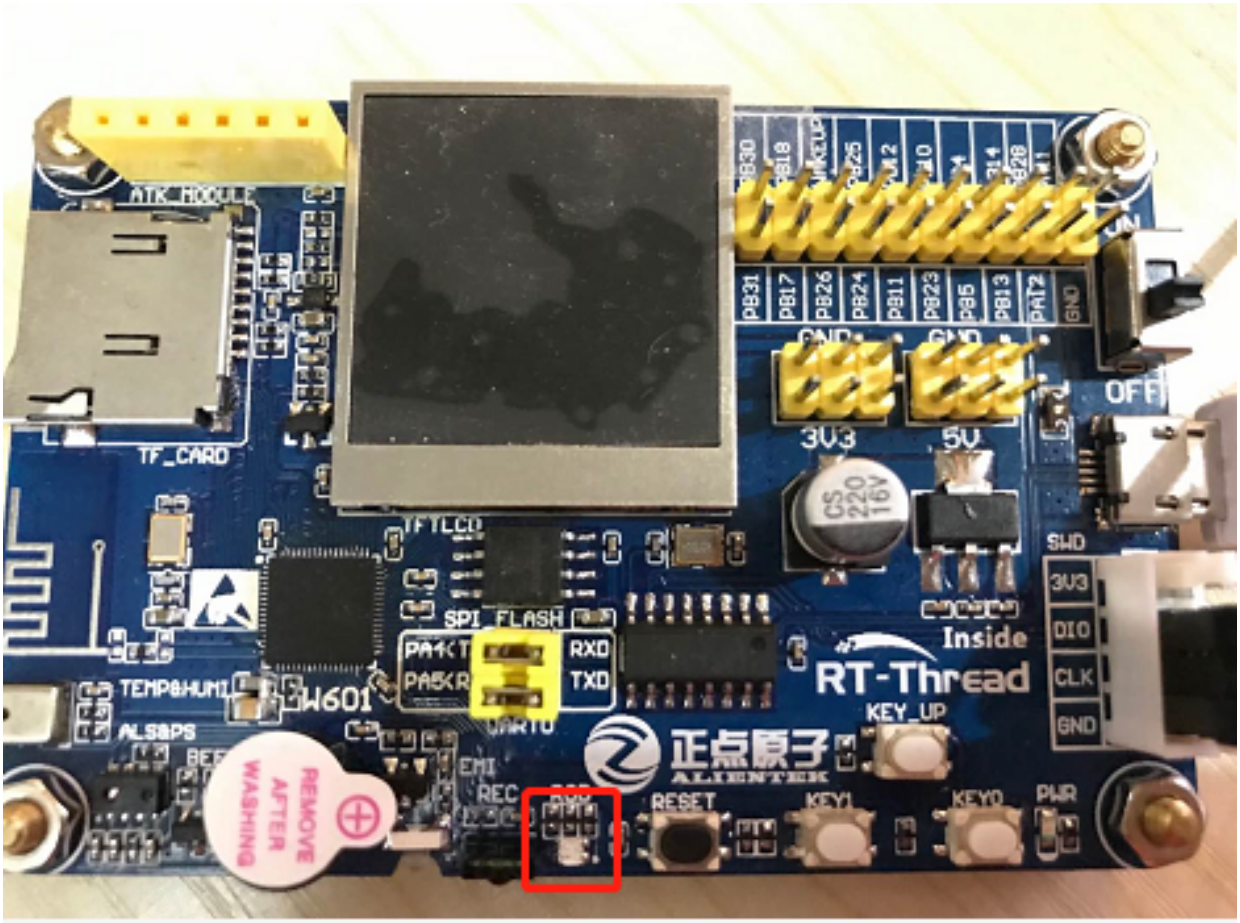


图 4.3: *LED 位置*

4.3 软件说明

闪灯的源代码位于 `/examples/01_basic_led_blink/applications/main.c` 中。首先定义了一个宏 `LED_PIN`，与 LED 红色引脚 `30` 号相对应。

```
/* using RED LED in RGB */
#define LED_PIN     (30)
```

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。

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

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

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

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

        count++;
    }

    return 0;
}
```

4.4 运行

4.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，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，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：

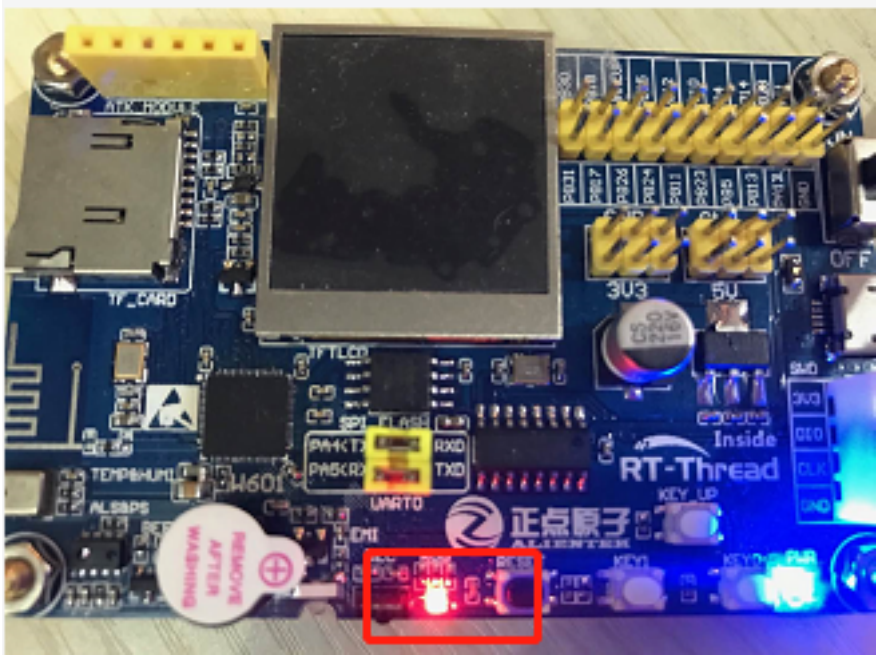


图 4.4: *RGB 红灯亮起*

此时也可以在 PC 端使用终端工具打开开发板的 `uart0` 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
[D/main] led off
[D/main] led on, count: 9
[D/main] led off
[D/main] led on, count: 10
```

4.5 注意事项

如果想要修改 LED_PIN 宏定义，可以参考 /drivers/pin_map.c 文件，该文件中里有定义单片机的其他引脚编号。

4.6 引用参考

- 《通用 GPIO 设备应用笔记》：docs/AN0002-RT-Thread-通用 GPIO 设备应用笔记.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 5 章

按键输入例程

5.1 简介

本例程的主要功能是让板载的 KEY0 控制 RGB-LED 中红色 LED 的亮灭。

5.2 硬件说明

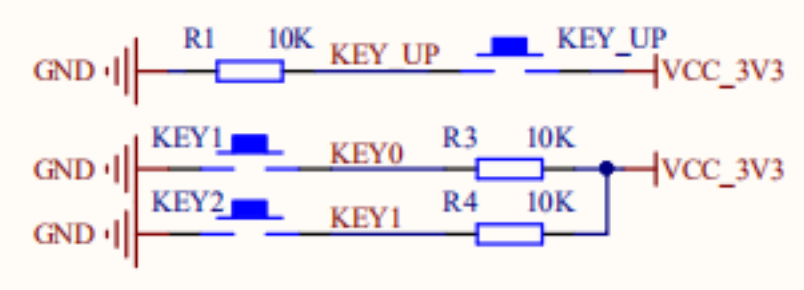


图 5.1: *KEY 电路图*

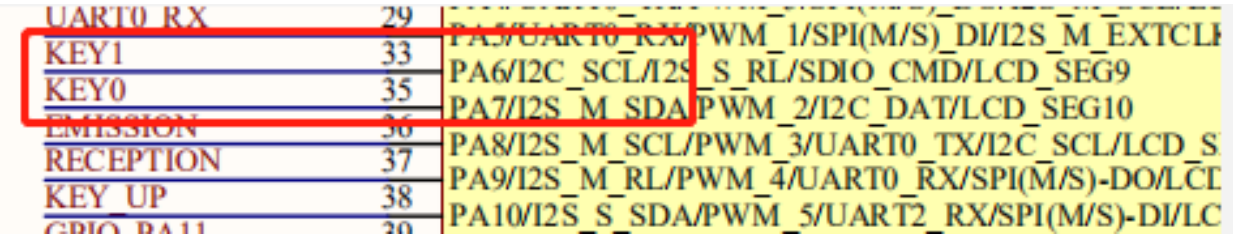


图 5.2: *KEY 电路图*

如上图所示，KEY0 引脚连接单片机 PA7（35）引脚，且外部接 10k 上拉电阻。KEY0 按键按下为低电平，松开为高电平。KEY0 在开发板中的位置如下图所示：

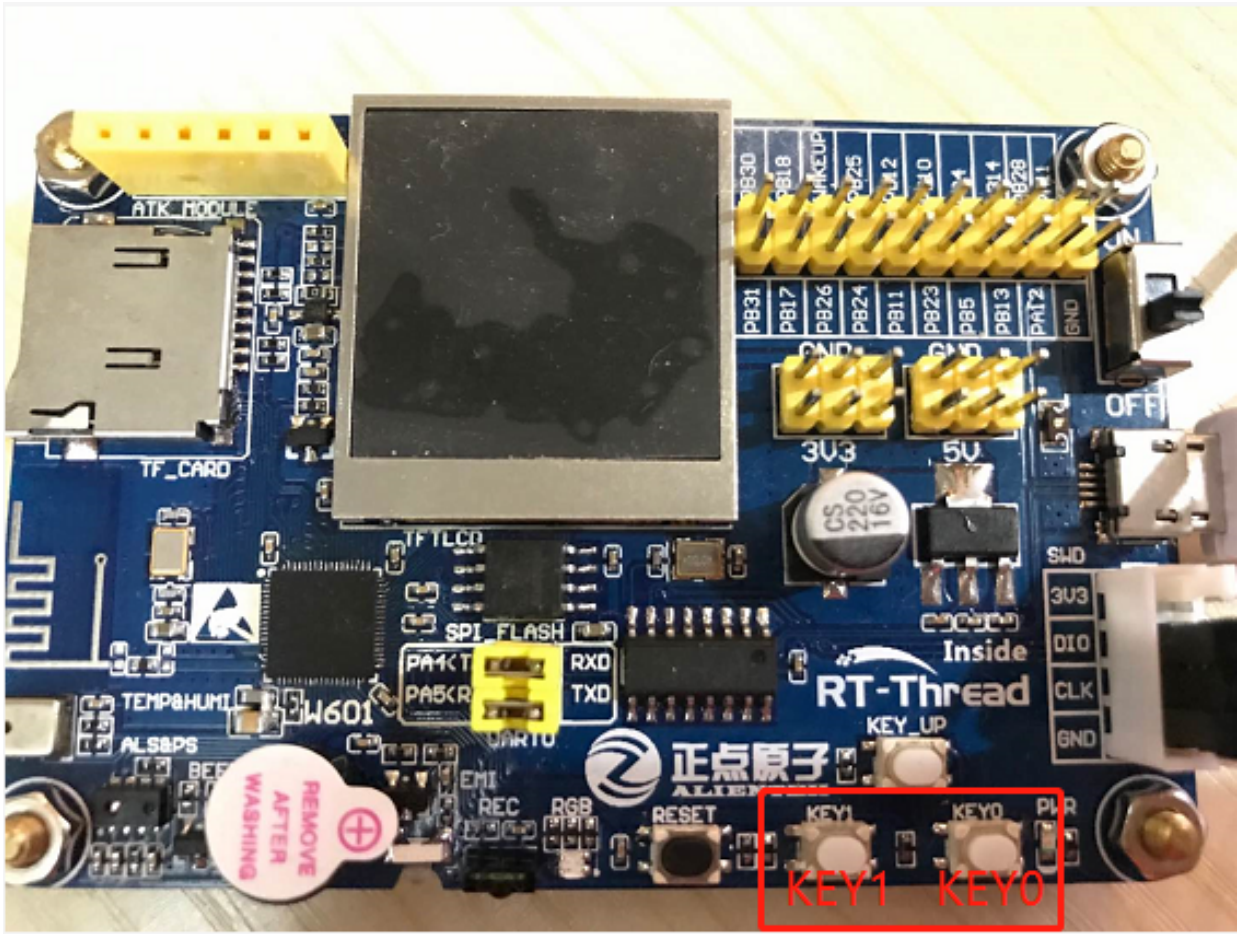


图 5.3: *KEY 位置*

5.3 软件说明

闪灯的源代码位于 /examples/02_basic_key/applications/main.c 中。定义了宏 PIN_KEY0 , KEY0 与单片机 35 号引脚相对应。

```
/* using KEY0  */
#define PIN_KEY0      (35)
```

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 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 的电平状态，并作 50ms 的消抖处理，如果成功判断 KEY0 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 PIN_LED_R 引脚为输出模式 */
    rt_pin_mode(PIN_LED_R, PIN_MODE_OUTPUT);
    /* 设置 PIN_KEY0 引脚为输入模式 */
    rt_pin_mode(PIN_KEY0, PIN_MODE_INPUT);
```

```
    while (count > 0)
    {
        /* 读取 PIN_KEY0 引脚的状态 */
        if (rt_pin_read(PIN_KEY0) == PIN_LOW)
        {
            rt_thread_mdelay(50);
            if (rt_pin_read(PIN_KEY0) == PIN_LOW)
            {
                LOG_D("KEY0 pressed!");
                rt_pin_write(PIN_LED_R, PIN_LOW);
            }
        }
        else
        {
            rt_pin_write(PIN_LED_R, PIN_HIGH);
        }
        rt_thread_mdelay(10);
        count++;
    }
    return 0;
}
```

5.4 运行

5.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

5.4.2 运行效果

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

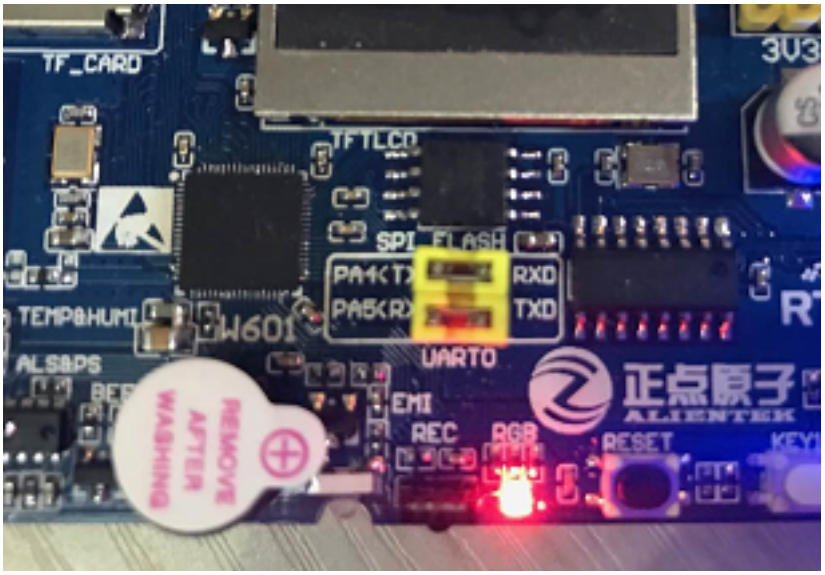


图 5.4: *RGB 红灯亮起*

此时也可以在 PC 端使用终端工具打开开发板的 `uart0` 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!
```

5.5 注意事项

如果想要修改 PIN_LED_R 或者 PIN_KEY0 宏定义，可以参考 /drivers/pin_map.c 文件，该文件中里有定义单片机的其他引脚编号。

5.6 引用参考

- 《通用 GPIO 设备应用笔记》：docs/AN0002-RT-Thread-通用 GPIO 设备应用笔记.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 6 章

RGB LED 例程

6.1 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。

6.2 硬件说明

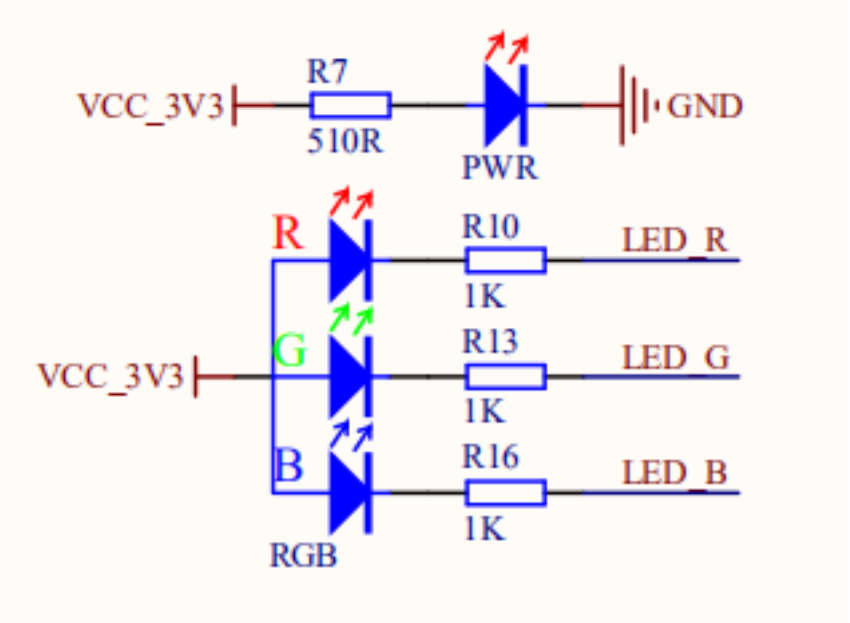


图 6.1: *RGB 灯电路图*

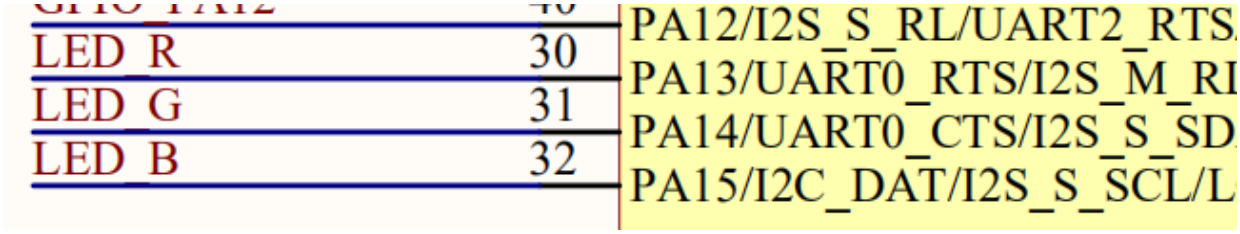


图 6.2: *RGB 灯电路图*

如上图所示，RGB-LED 属于共阳 LED，**阴极**分别与单片机的 30，31，32 号引脚连接，其中红色 LED 对应 **30** 号引脚，蓝色 LED 对应 **31** 号引脚，绿色 LED 对应 **32** 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。

RGB-LED 在开发板中的位置如下图所示：

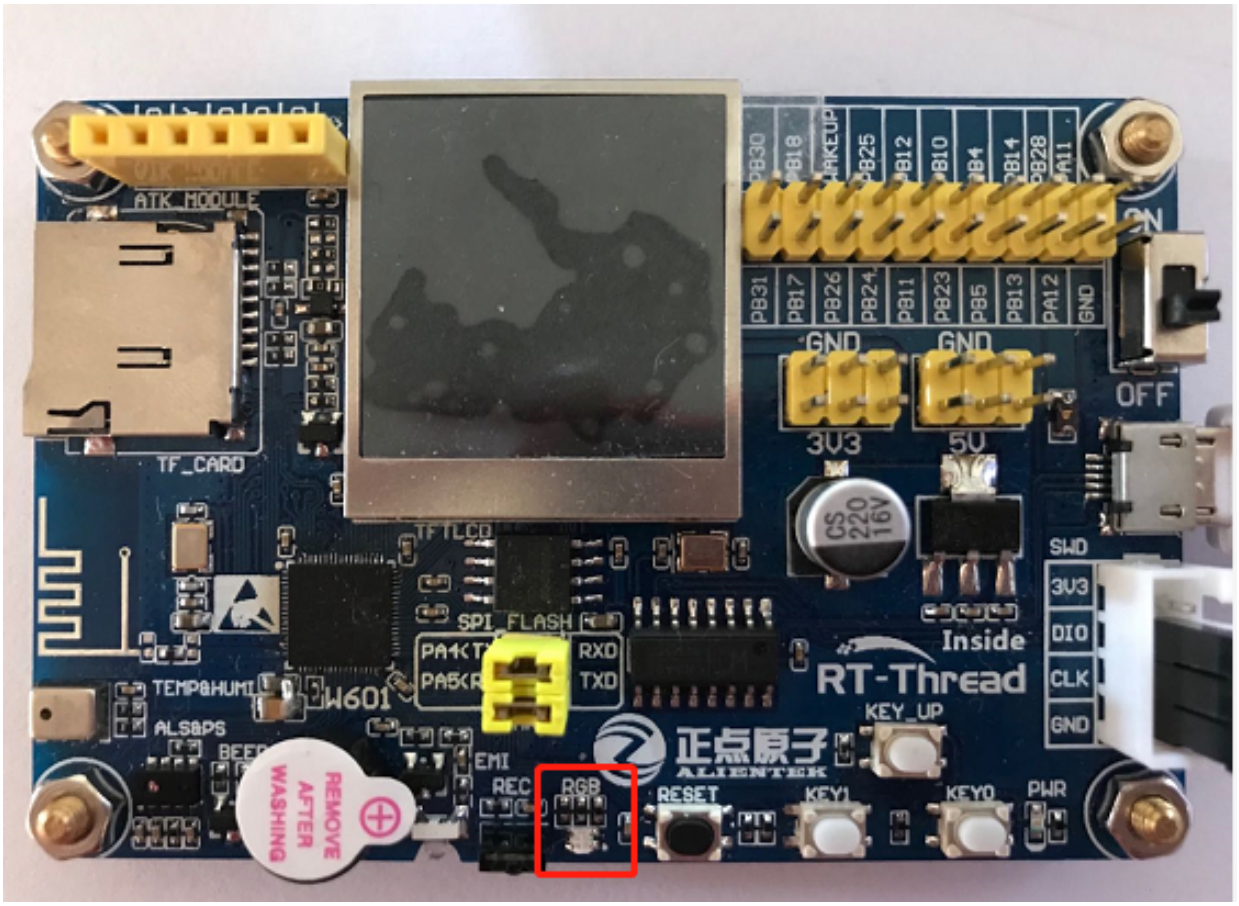


图 6.3: *RGB 灯的位置*

6.3 软件说明

闪灯的源代码位于 `/examples/03_basic_rgb_led/applications/main.c` 中。定义了以下宏：

```
#define PIN_LED_R (30)
#define PIN_LED_G (31)
#define PIN_LED_B (32)
```

在 main 函数中，将三个引脚配置为输出模式，并在下面的 while 循环中，每 500 毫秒变化一次 RGB 颜色，同时输出一些日志信息，一共有 8 组变化。

```
int main(void)
{
    unsigned int count = 1;
    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_G, PIN_MODE_OUTPUT);
    rt_pin_mode(PIN_LED_B, PIN_MODE_OUTPUT);

    while (count > 0)
    {
        /* 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_G, _blink_tab[group_current][1]);
        rt_pin_write(PIN_LED_B, _blink_tab[group_current][2]);

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

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

6.4 运行

6.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，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：

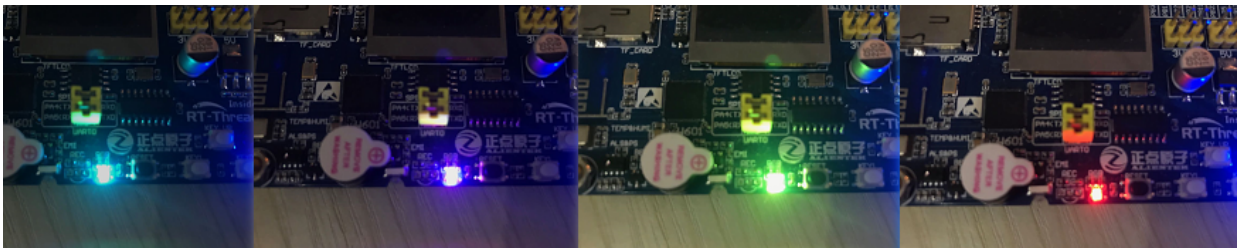


图 6.4: *RGB 灯周期闪烁*

此时也可以在 PC 端使用终端工具打开开发板的 `uart0` 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 ] | green led [ON ] | blue led [ON ]
[D/main] group: 1 | red led [OFF] | green led [ON ] | blue led [ON ]
[D/main] group: 2 | red led [ON ] | green led [OFF] | blue led [ON ]
[D/main] group: 3 | red led [ON ] | green led [ON ] | blue led [OFF]
[D/main] group: 4 | red led [OFF] | green led [OFF] | blue led [ON ]
[D/main] group: 5 | red led [ON ] | green led [OFF] | blue led [OFF]
[D/main] group: 6 | red led [OFF] | green led [ON ] | blue led [OFF]
[D/main] group: 7 | red led [OFF] | green led [OFF] | blue led [OFF]
```

6.5 注意事项

如果想要修改宏定义，可以参考 /drivers/pin_map.c 文件，该文件中里有定义单片机的其他引脚编号。

6.6 引用参考

- 《通用 GPIO 设备应用笔记》：docs/AN0002-RT-Thread-通用 GPIO 设备应用笔记.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 7 章

蜂鸣器控制例程

7.1 简介

本例程主要功能为使用按键控制蜂鸣器，KEY0 按键通过中断的方式控制蜂鸣器鸣叫。

7.2 硬件说明

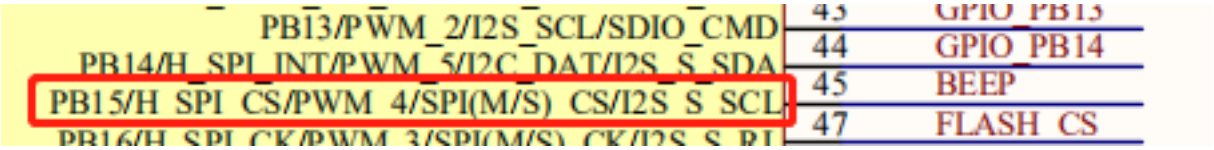


图 7.1: *BEEP 电路图*

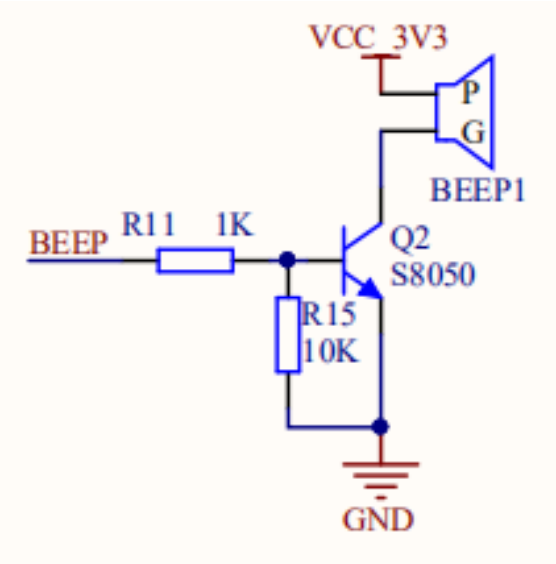


图 7.2: *BEEP 电路图*

如上图所示，BEEP 引脚连接单片机 PB15（45）引脚，可以通过输出高低电平控制蜂鸣器鸣叫。蜂鸣器及按键在开发板上的位置如下图所示：

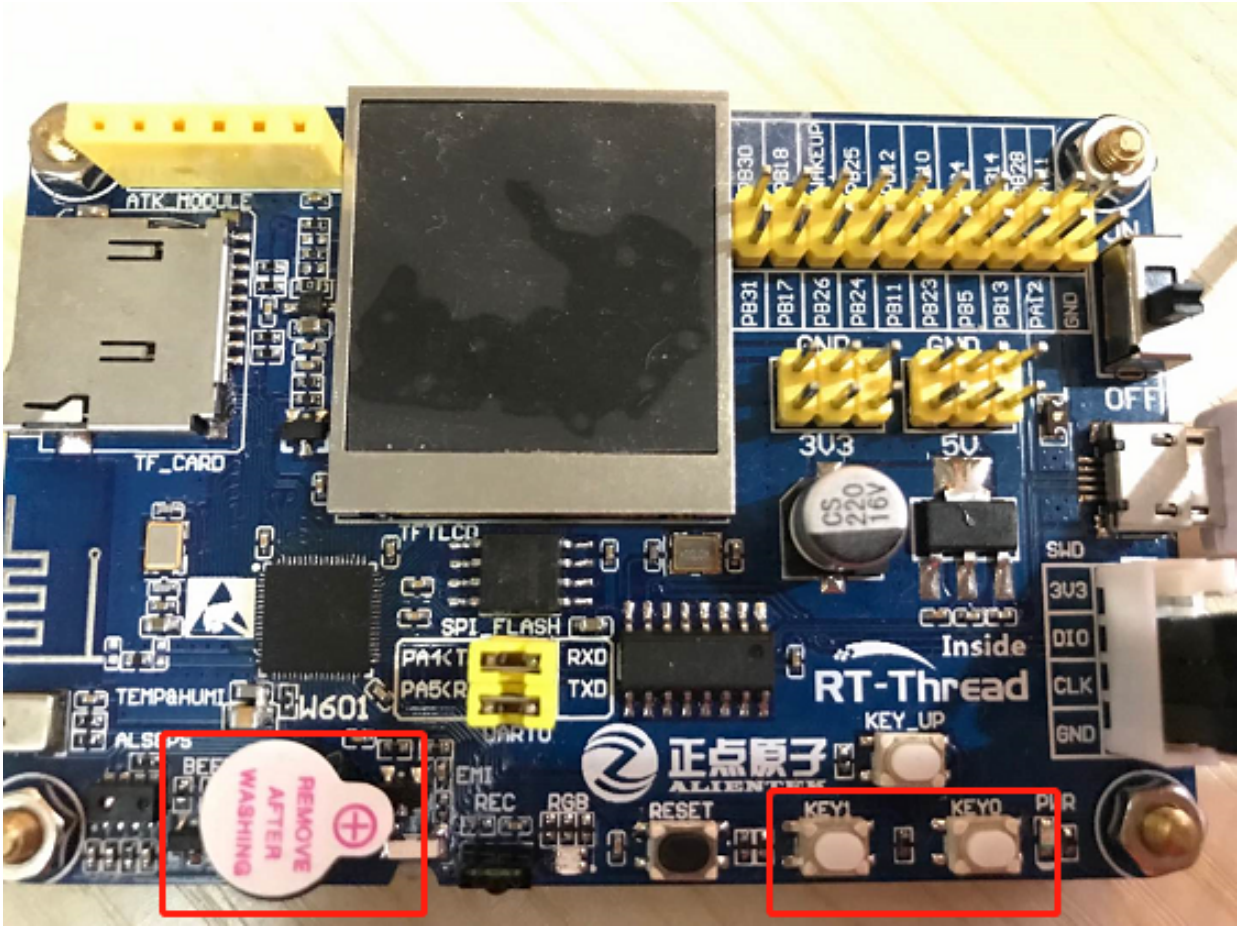


图 7.3: *BEEP 与 KEY 位置*

7.3 软件说明

闪灯的源代码位于 /examples/04_basic_beep/applications/main.c 中。定义了宏 PIN_KEY0 , KEY0 与单片机 35 号引脚相对应，BEEP 与单片机 45 号引脚相对应。

```
#define PIN_KEY0      (35)
#define PIN_BEEP      (45)
```

在 main 函数中，设置单片机 KEY0 引脚为中断方式触发，设置输入模式，设置中断触发方式与中断回调函数，使能中断。设置单片机 BEEP 引脚为输出模式，并默认输出低电平，使蜂鸣器在上电后默认处于不鸣叫的状态。

```
int main(void)
{
    /* 设置 KEY0 引脚为输入模式 */
    rt_pin_mode(PIN_KEY0, PIN_MODE_INPUT);
    /* KEY0 引脚绑定中断回调函数 */
    rt_pin_attach_irq(PIN_KEY0, PIN_IRQ_MODE_RISING_FALLING, beep_ctrl, RT_NULL);
```

```
    /* 使能中断 */
    rt_pin_irq_enable(PIN_KEY0, PIN_IRQ_ENABLE);

    /* 设置 BEEP 引脚为输出模式 */
    rt_pin_mode(PIN_BEEP, PIN_MODE_OUTPUT);
    /* 默认蜂鸣器不鸣叫 */
    rt_pin_write(PIN_BEEP, PIN_LOW);

    return 0;
}
```

当中断触发时，会执行中断回调函数，控制蜂鸣器。在中断触发时，通过读取 KEY0 引脚的电平确定当前按键是否按下，当按键按下时（低电平）使蜂鸣器鸣叫，打印相应的信息 `KEY0 pressed. beep on`，当松开按键时（高电平）使蜂鸣器不工作，打印相应的信息 `beep off`。

```
void beep_ctrl(void *args)
{
    if(rt_pin_read(PIN_KEY0) == PIN_LOW)
    {
        rt_pin_write(PIN_BEEP, PIN_HIGH);
        LOG_D("KEY0 pressed. beep on");
    }
    else
    {
        rt_pin_write(PIN_BEEP, PIN_LOW);
        LOG_D("beep off");
    }
}
```

7.4 运行

7.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

7.4.2 运行效果

按下复位按键重启开发板，按住 KEY0 可以听到蜂鸣器鸣叫；松开按键，蜂鸣器不鸣叫。

此时也可以在 PC 端使用终端工具打开开发板的 `uart0` 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. beep on
[D/main] beep off
[D/main] KEY0 pressed. beep on
```

```
[D/main] beep off
```

7.5 注意事项

如果想要修改 PIN_KEY0 或者 PIN_BEEP 宏定义，可以参考 /drivers/pin_map.c 文件，该文件中里有定义单片机的其他引脚编号。

7.6 引用参考

- 《通用 GPIO 设备应用笔记》：docs/AN0002-RT-Thread-通用 GPIO 设备应用笔记.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 8 章

红外遥控例程

8.1 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。

8.2 硬件说明

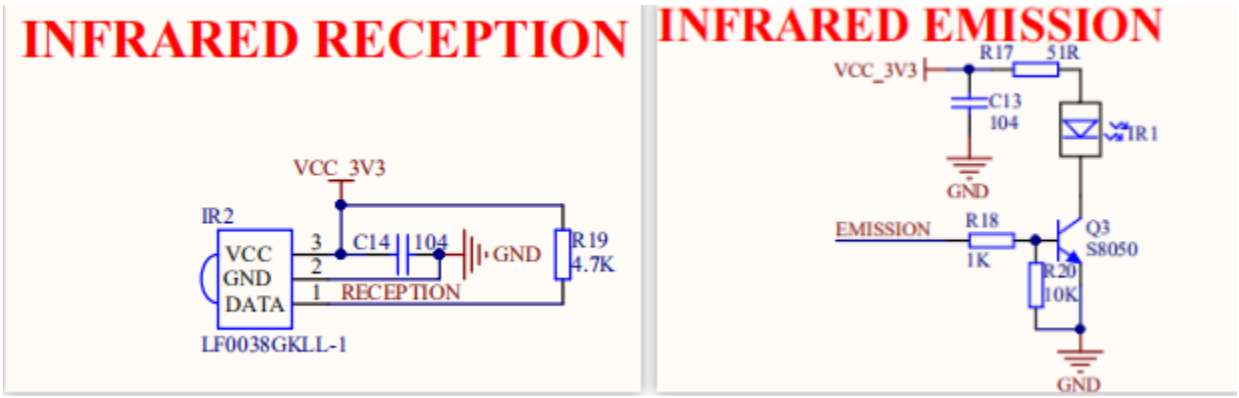


图 8.1: *红外原理图*

图 8.2: *红外原理图*

如上图所示，红外接收头引脚 RECEPTION 接单片机引脚 PA10（38 号，PWM_CH4），红外发射脚 EMISSION 接单片机引脚 PA9 （37 号）。

红外传感器在开发板中的位置如下图所示：

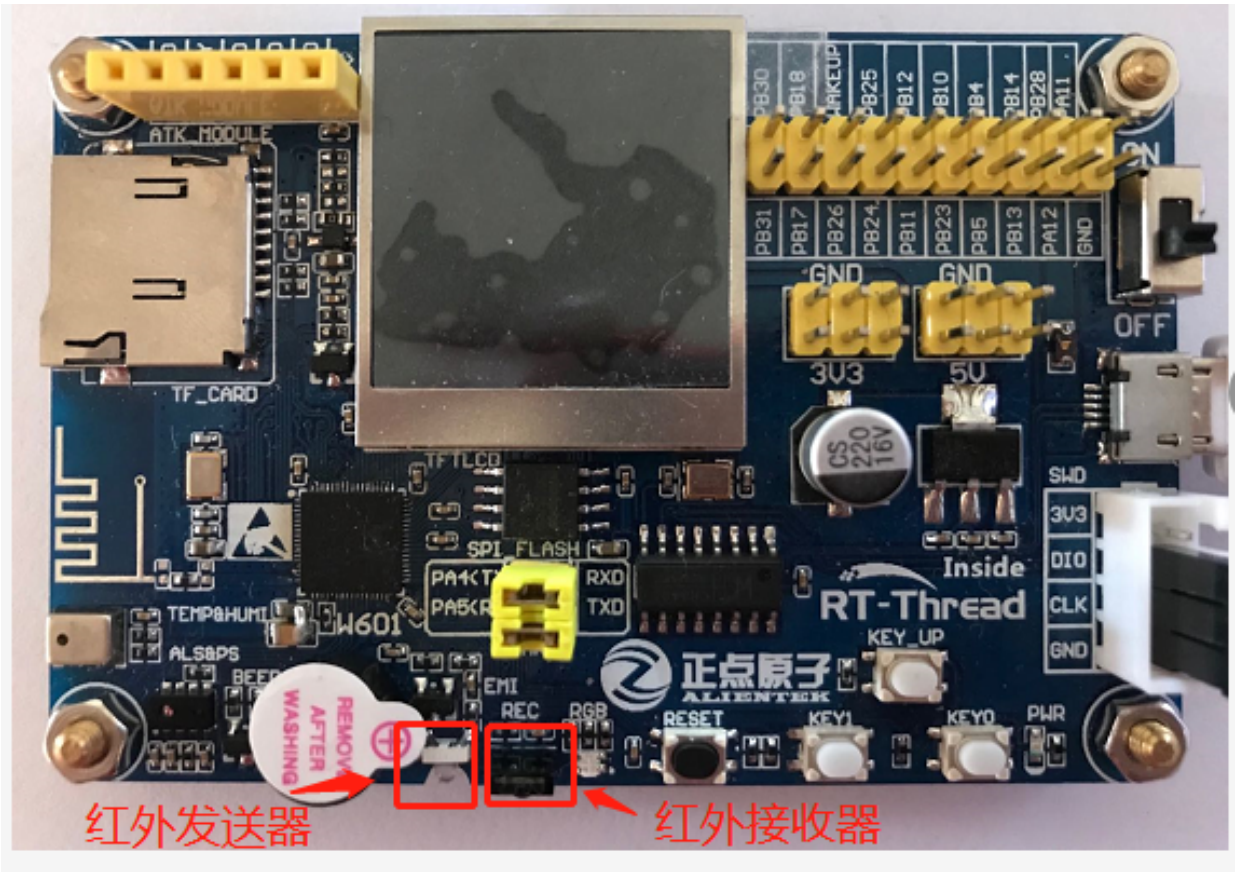


图 8.3: 红外位置

8.3 软件说明

红外例程源代码位于 /examples/05_basic_ir/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);

    /* 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
    );
```

8.4 运行

8.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，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，设置波特率：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。

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build May 28 2019
 2006 - 2019 Copyright by rt-thread team
msh >[I/main] RECEIVE OK: addr:0x00 key:0x18 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x7A repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x7A repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0x10 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x10 repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0x38 repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x38 repeat:1
[I/main] RECEIVE OK: addr:0x00 key:0x5A repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x5A repeat:1
[I/main] SEND    OK: addr:0x00 key:0x5A repeat:0
[I/main] RECEIVE OK: addr:0x00 key:0x5A repeat:0
```

8.5 注意事项

请使用 38KHZ 载波的红外遥控器实验。

8.6 引用参考

- 《通用 GPIO 设备应用笔记》：docs/AN0002-RT-Thread-通用 GPIO 设备应用笔记.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 9 章

LCD 显示例程

9.1 简介

本例程主要介绍了如何在 LCD 上显示文字和图片。

9.2 硬件说明

开发板板载一块 1.3 寸，分辨率为 240*240 的 LCD 显示屏，显示效果十分细腻。显示屏的驱动芯片是 ST7789，通过 4-line SPI 接口和单片机进行通讯。因为只需要往 LCD 写数据而不需要读取，所以，可以不接 MISO 引脚。

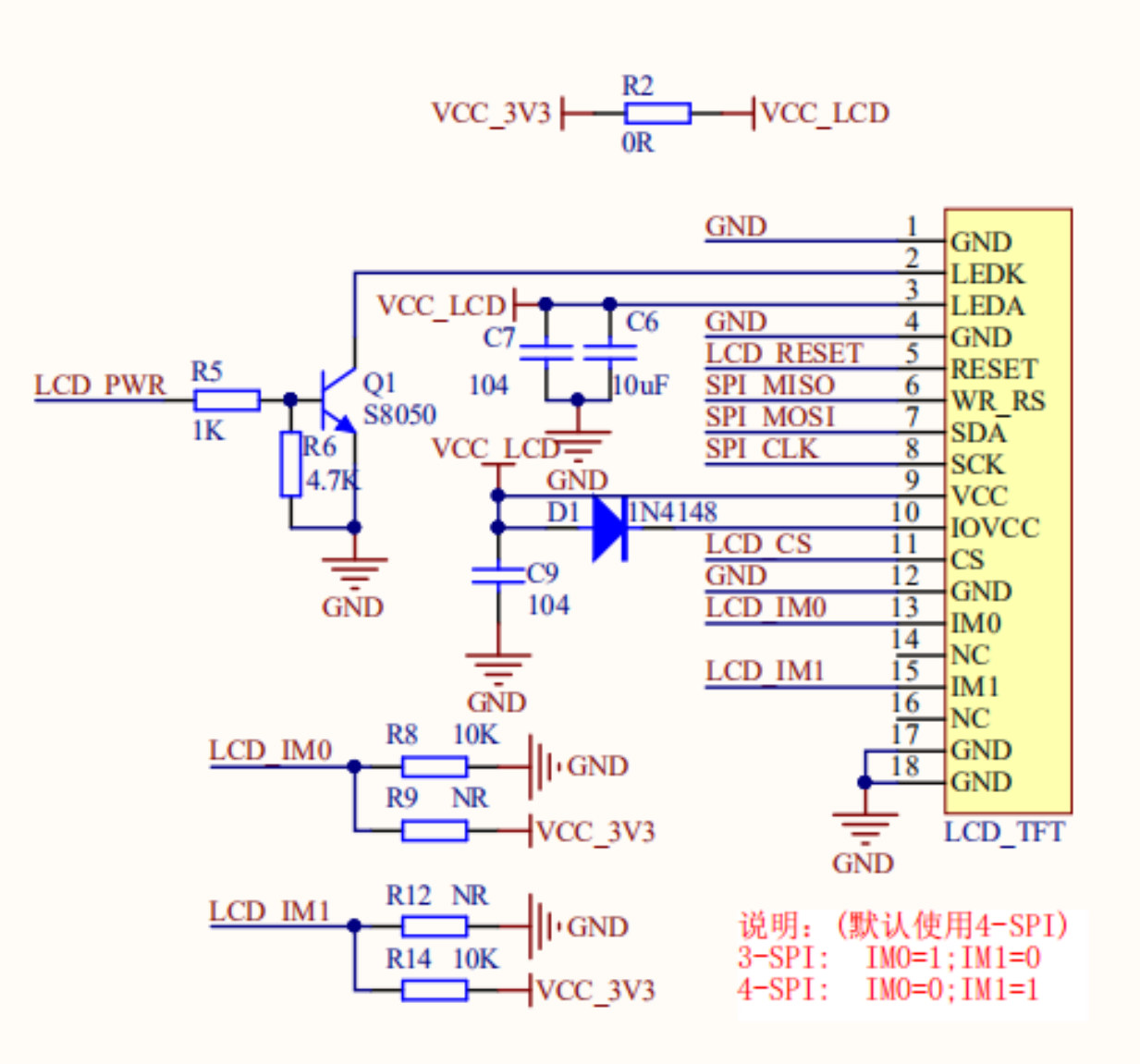


图 9.1: *LCD 原理图*

如上图所示，单片机通过以下管脚来控制 LCD：

名称	管脚	作用
SPI_MISO	PB1	SPI 命令数据选择
SPI_MOSI	PB2	SPI 数据线
SPI_CLK	PB27	SPI 时钟线
LCD_PWR	PB8	背光控制
LCD_RESET	PB3	LCD 复位
LCD_CS	PB9	SPI 片选信号

LCD 在开发板中的位置如下图所示：

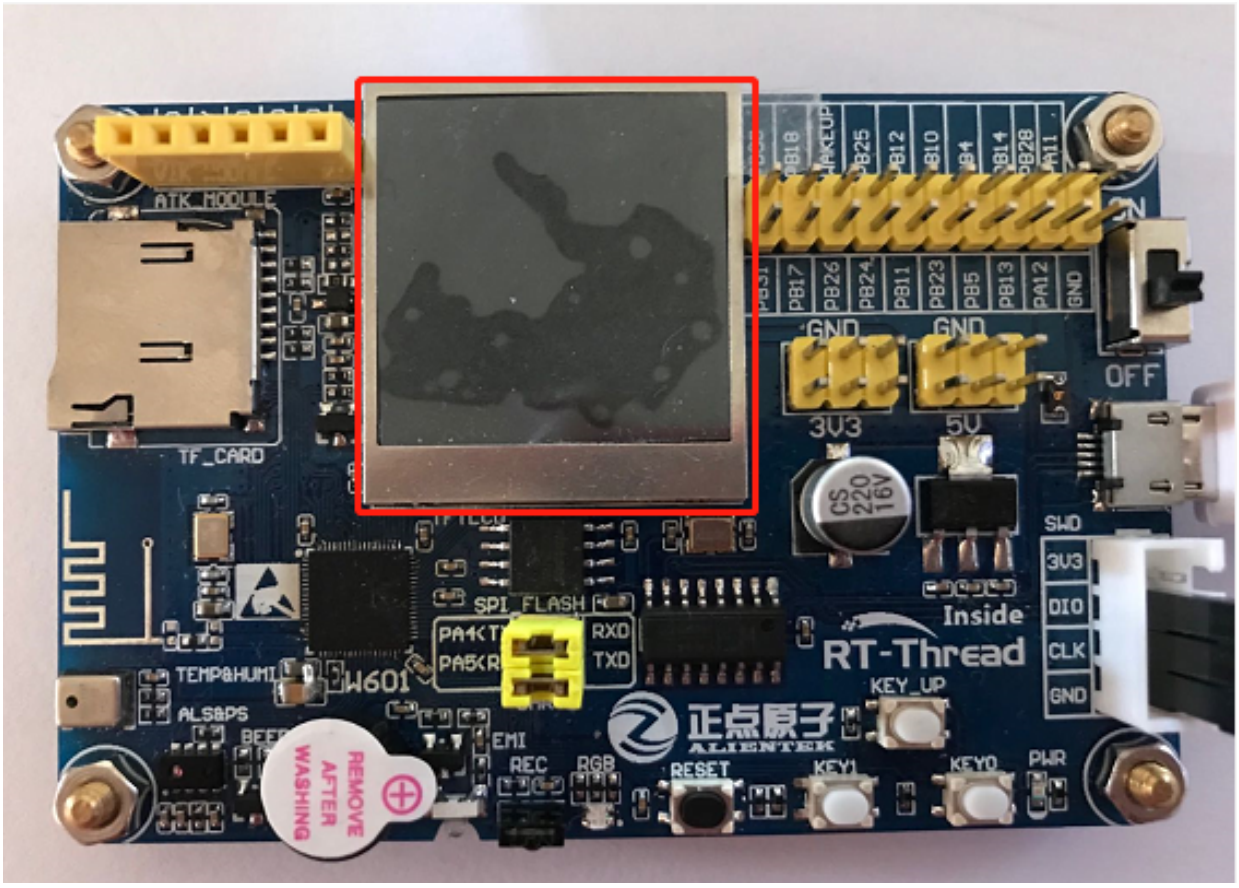


图 9.2: *LCD 位置*

9.3 软件说明

显示图片和文字的源代码位于 `/examples/06_driver_lcd/applications/main.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。然后显示 RT-Thread 的 LOGO，接着设置画笔的颜色为黑色，背景色为白色。最后会显示一些信息，包括 16*16 像素，24*24 像素和 32*32 像素的三行英文字符，一条横线和一个同心圆。

```
int main(void)
{
    /* 清屏 */
    lcd_clear(WHITE);

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

    /* 设置背景色和前景色 */
    lcd_set_color(WHITE, BLACK);

    /* 在 LCD 上显示字符 */
    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");
```

```
    /* 在 LCD 上画线 */
    lcd_draw_line(0, 69+16+24+32, 240, 69+16+24+32);

    /* 在 LCD 上画一个同心圆 */
    lcd_draw_point(120, 194);
    for (int i = 0; i < 46; i += 4)
    {
        lcd_draw_circle(120, 194, i);
    }

    return 0;
}
```

9.4 运行

9.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

9.4.2 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 9.3: *demo 效果图*

9.5 注意事项

屏幕的分辨率是 240*240，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。

9.6 引用参考

- 《SPI 设备应用笔记》：docs/AN0004-RT-Thread-SPI 设备应用笔记.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 10 章

AHT10 温湿度传感器例程

10.1 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` 所测量的温度（temperature）与湿度（humidity）。

10.2 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。

10.3 硬件说明

`aht10` 硬件原理图如下所示：

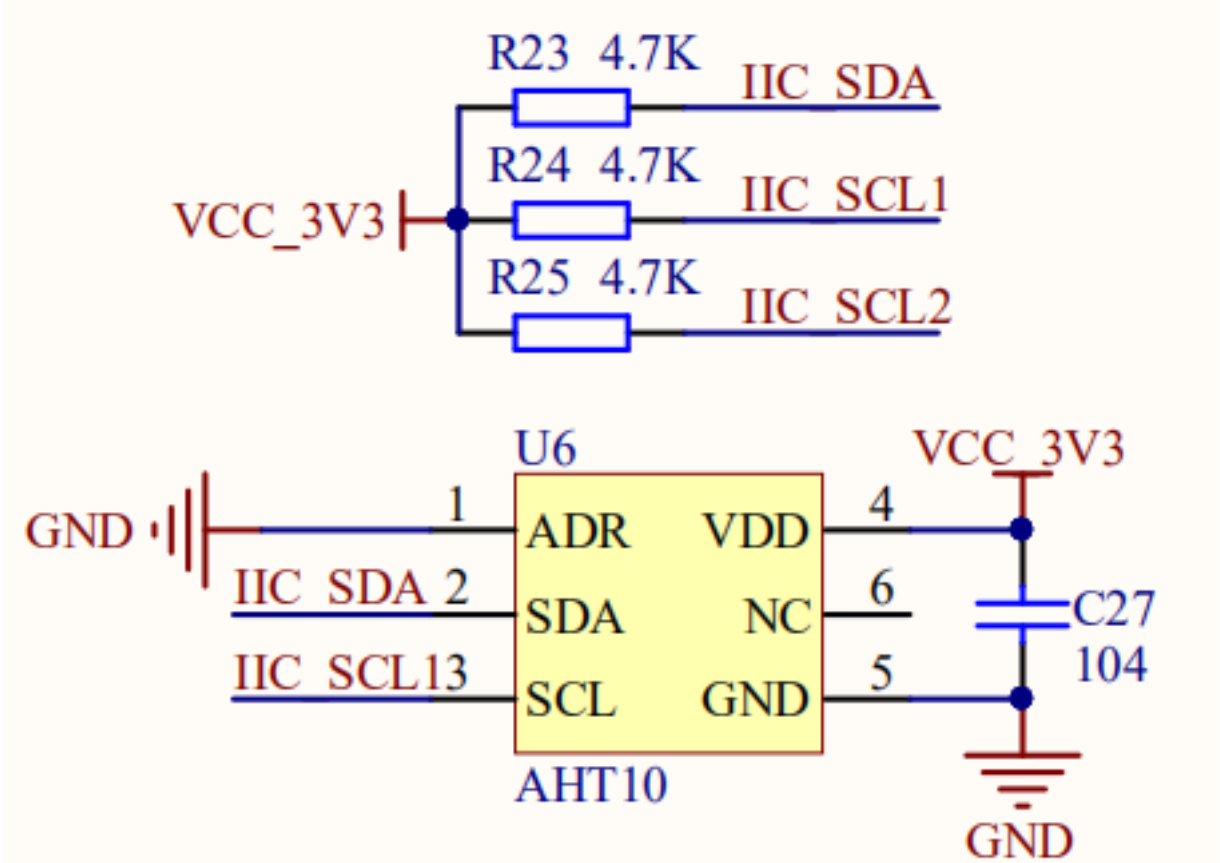


图 10.1: *温湿度传感器连接原理图 1*

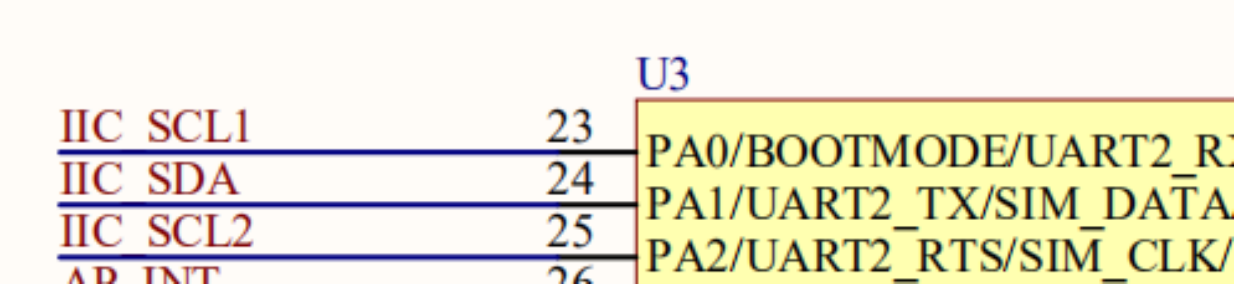


图 10.2: *温湿度传感器连接原理图 2*

如上图所示，单片机通过 IIC_SDA(PA1)、IIC_SCL1(PA0) 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：

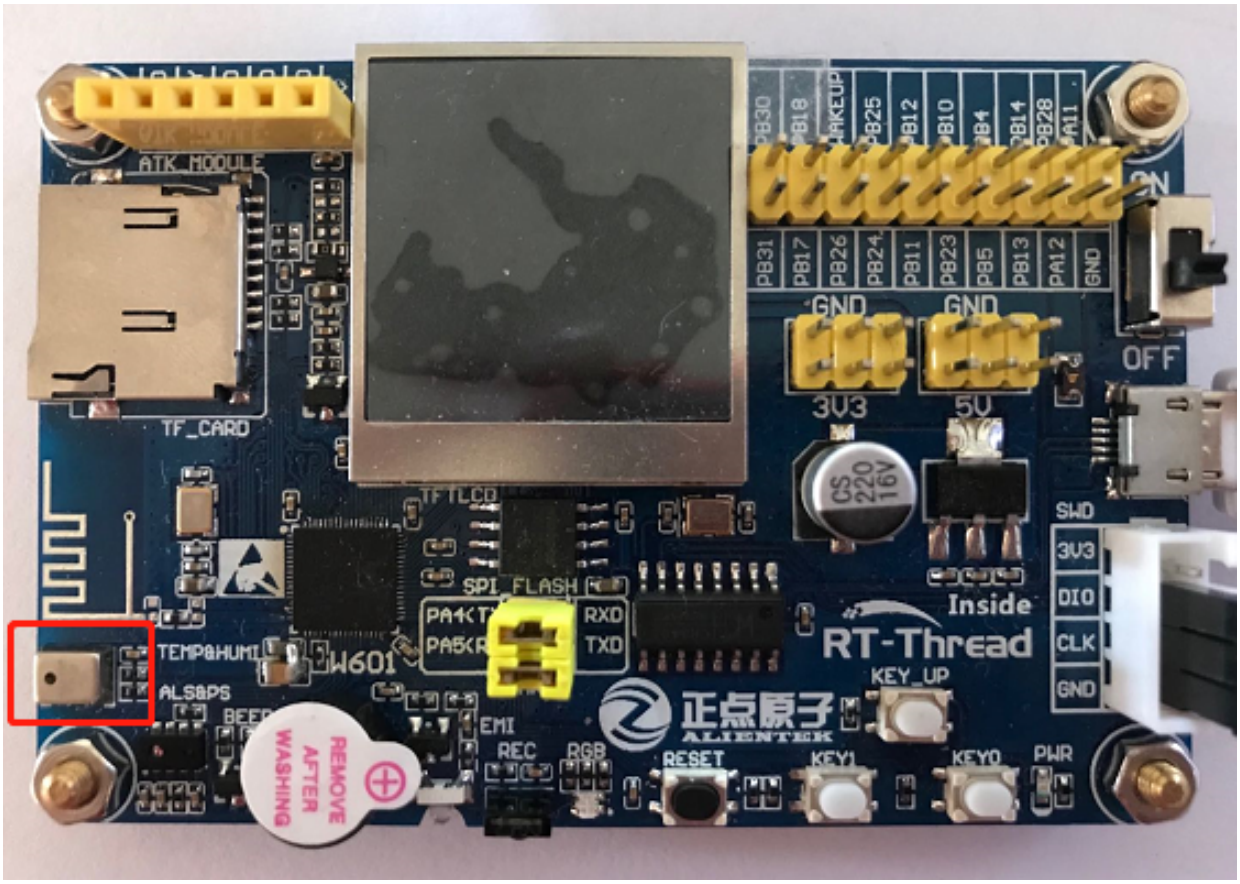


图 10.3: *温湿度传感器位置*

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 到 85	±0.5	摄氏度
相对湿度	0 到 100	±3	%

10.4 软件说明

温度与湿度传感器的示例代码位于 /examples/07_driver_temp_humi/applications/main.c 中，在 /examples/07_driver_temp_humi/ports/sensors/sensor_port.c 中有移植代码。

首先在 sensor_port.c 中已经完成了设备配置与初始化。设置 dev_name 为 i2c1soft，设置 user_data 为该传感器的设备地址 AHT10_I2C_ADDR，该地址定义在 aht10 软件包的 sensor_asair_aht10.h 中。

```
#include "sensor_asair_aht10.h"

#define AHT10_I2C_BUS  "i2c1soft"

int rt_hw_aht10_port(void)
{
```

```
    struct rt_sensor_config cfg;

    cfg.intf.dev_name  = AHT10_I2C_BUS;
    cfg.intf.user_data = (void *)AHT10_I2C_ADDR;

    rt_hw_aht10_init("aht10", &cfg);

    return RT_EOK;
}
INIT_ENV_EXPORT(rt_hw_aht10_port);
```

接着是 main.c 中的用户代码，由于在 rt_hw_aht10_init() 函数中注册了两个传感器设备，分别叫做 temp_aht 与 humi_aht，所以在查找设备时，首先寻找温度传感器 temp_aht ，然后打开温度传感器设备；接着寻找湿度传感器 humi_aht ，然后打开湿度传感器设备；最后每隔 1s 读取 1 次温湿度数据，共读取 20 次。在这过程中使用了 rt_device_find() ，rt_device_open() ，rt_device_read() 。

```
#define TEMP_DEV        "temp_aht"
#define HUMI_DEV        "humi_aht"

int main(void)
{
    int count = 0;
    rt_device_t temp_dev, humi_dev;
    static struct rt_sensor_data temp_dev_data, humi_dev_data;

    LOG_D("Temperature and Humidity Sensor Testing Start...");
    rt_thread_mdelay(2000);

    /* 寻找并打开温度传感器 */
    temp_dev = rt_device_find(TEMP_DEV);
    if(temp_dev == RT_NULL)
    {
        LOG_E("can not find TEMP device: %s", TEMP_DEV);
        return RT_ERROR;
    }
    else
    {
        if (rt_device_open(temp_dev, RT_DEVICE_FLAG_RDONLY) != RT_EOK)
        {
            LOG_E("open TEMP device failed!");
            return RT_ERROR;
        }
    }

    /* 寻找并打开湿度传感器 */
    humi_dev = rt_device_find(HUMI_DEV);
    if(humi_dev == RT_NULL)
    {
        LOG_E("can not find HUMI device: %s", HUMI_DEV);
```

```
        return RT_ERROR;
    }
    else
    {
        if (rt_device_open(humi_dev, RT_DEVICE_FLAG_RDONLY) != RT_EOK)
        {
            LOG_E("open HUMI device failed!");
            return RT_ERROR;
        }
    }

    while (count++ < 20)
    {
        rt_device_read(temp_dev, 0, &temp_dev_data, 1);
        LOG_D("temperature: %d.%d C", (int)(temp_dev_data.data.temp / 10), (int)(
            temp_dev_data.data.temp % 10));

        rt_device_read(humi_dev, 0, &humi_dev_data, 1);
        LOG_D("humidity    : %d.%d %%", (int)(humi_dev_data.data.humi / 10), (int)(
            humi_dev_data.data.humi % 10));

        rt_thread_mdelay(1000);
    }

    rt_device_close(temp_dev);
    rt_device_close(humi_dev);
    LOG_D("Temperature and Humidity Sensor Testing Ended.");

    return RT_EOK;
}
```

10.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

10.4.2 运行效果

烧录完成后，此时也可以在 PC 端使用终端工具打开开发板的 `uart0` 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。

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build May 15 2019
 2006 - 2019 Copyright by rt-thread team
[D/soft_i2c] software simulation i2c1soft init done, pin scl: 23, pin sda 24
```

```
[I/sensor] rt_sensor init success
[I/sensor] rt_sensor init success
[D/main] Temperature and Humidity Sensor Testing Start...
msh >[D/main] temperature: 30.4 C
[D/main] humidity   : 54.0 %
[D/main] temperature: 30.4 C
[D/main] humidity   : 53.9 %
[D/main] temperature: 30.5 C
[D/main] humidity   : 53.8 %
...
[D/main] Temperature and Humidity Sensor Testing Ended.
```

10.5 注意事项

请使用 aht10-v2.0.0 版本软件包，此软件包已经对接了 sensor 框架。

10.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《I2C 设备应用笔记》：docs/AN0003-RT-Thread-I2C 设备应用笔记
- 《aht10 软件包介绍》：https://github.com/RT-Thread-packages/aht10

第 11 章

AP3216C 接近与光强传感器例程

11.1 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）。

11.2 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。

11.3 硬件说明

ap3216c 硬件原理图如下所示：

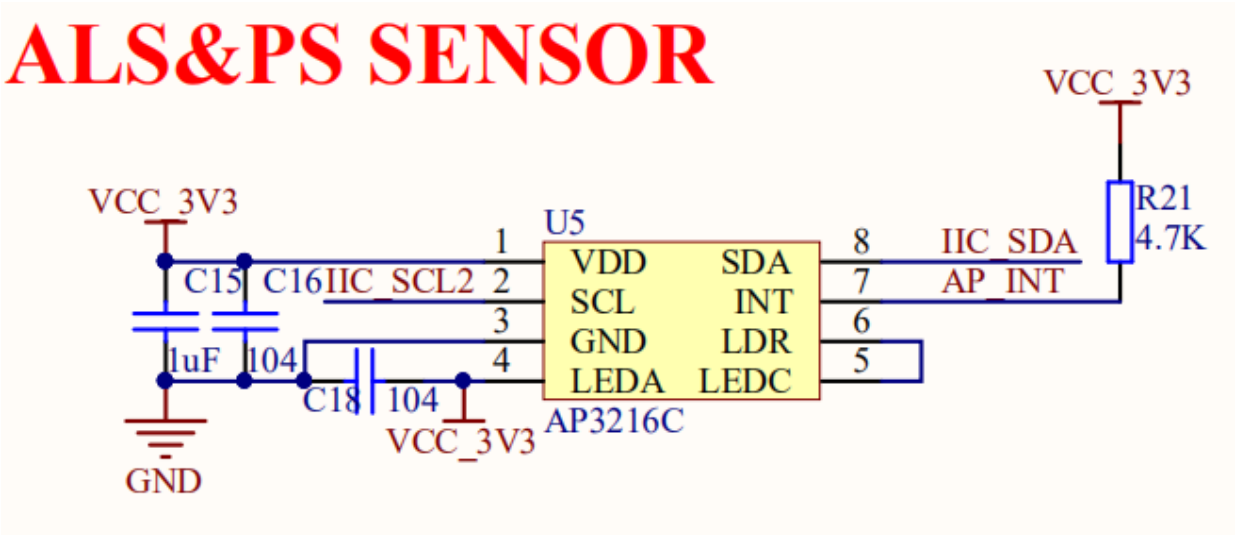


图 11.1: *接近与光强传感器连接原理图*

如上图所示，单片机通过 IIC_SDA(PA1)、IIC_SCL2(PA2) 对传感器 `ap3216c` 发送命令、读取数据等，AP_INT(PA3) 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：

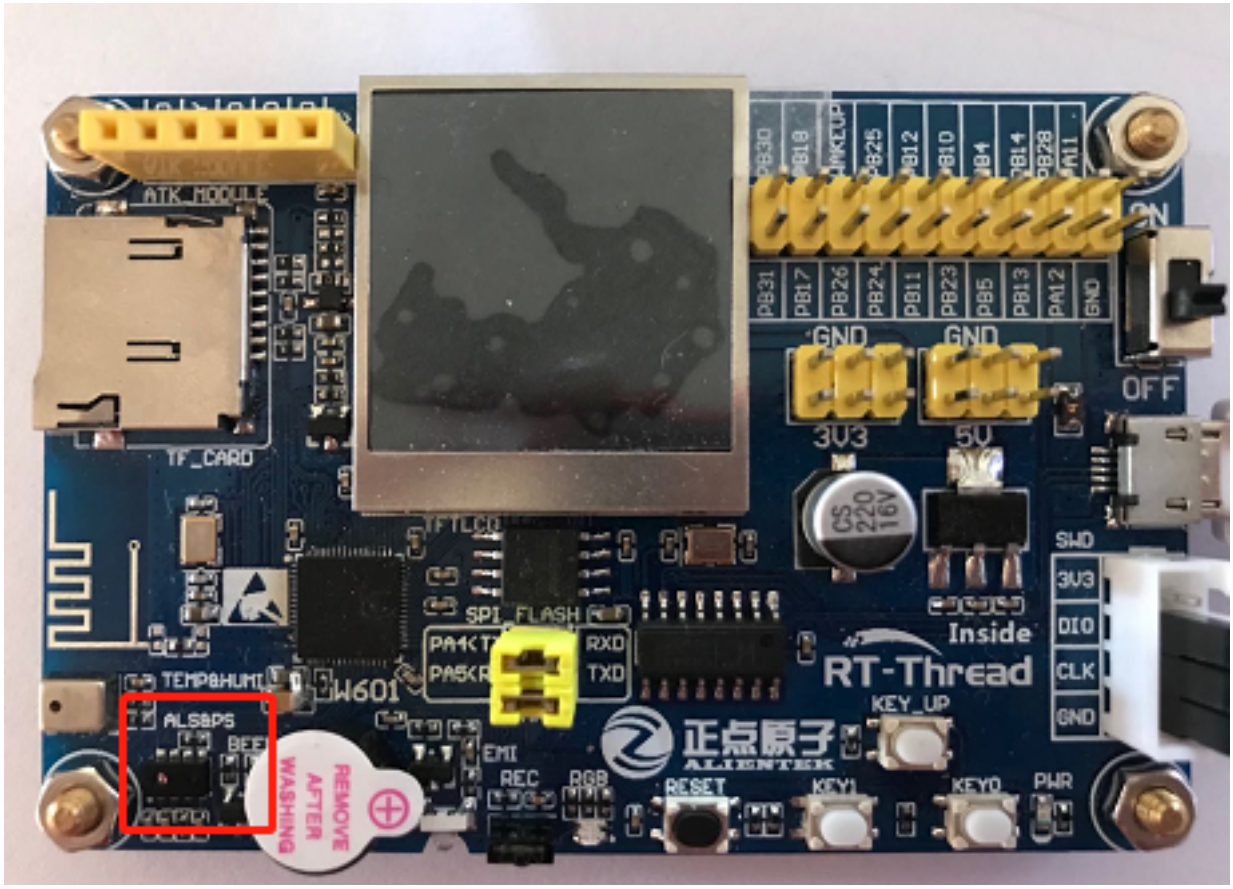


图 11.2: *接近与光强传感器位置*

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

11.4 软件说明

接近感应与光照强度传感器 `ap3216c` 的示例代码位于 `/examples/08_driver_als_ps/applications/main.c` 中，主要流程：

1. 查找并打开光照强度传感器设备；
2. 查找并打开接近感应传感器设备；
3. 读取两个设备的数据各 20 次；
4. 关闭两个设备。

示例代码如下：

```
int main(void)
{
    int count = 0;
    rt_device_t als_dev, ps_dev;
    struct rt_sensor_data als_dev_data, ps_dev_data;

    LOG_D("Als Ps Sensor Testing Start...");

    /* 查找并打开光强传感器 */
    als_dev = rt_device_find(ALS_DEV);
    if (als_dev == RT_NULL)
    {
        LOG_E("can not find ALS device: %s", ALS_DEV);
        return -RT_ERROR;
    }
    else
    {
        if (rt_device_open(als_dev, RT_DEVICE_FLAG_RDONLY) != RT_EOK)
        {
            LOG_E("open ALS device failed!");
            return -RT_ERROR;
        }
    }

    /* 查找并打开接近传感器 */
    ps_dev = rt_device_find(PS_DEV);
    if (ps_dev == RT_NULL)
    {
        LOG_E("can not find PS device: %s", PS_DEV);
        return -RT_ERROR;
    }
    else
    {
        if (rt_device_open(ps_dev, RT_DEVICE_FLAG_RDONLY) != RT_EOK)
        {
            LOG_E("open PS device failed!");
            return -RT_ERROR;
        }
    }

    /* 开始读取传感器数据 */
    while (count++ < 20)
    {
        rt_device_read(als_dev, 0, &als_dev_data, 1);
        LOG_D("current brightness: %d.%d(lux).", (int)(als_dev_data.data.light / 10)
            , (int)(als_dev_data.data.light % 10));

        rt_device_read(ps_dev, 0, &ps_dev_data, 1);
```

```
        if (ps_dev_data.data.proximity == 0)
        {
            LOG_D("no object approaching.");
        }
        else
        {
            LOG_D("current ps data   : %d.", ps_dev_data.data.proximity);
        }

        rt_thread_mdelay(1000);
    }

    rt_device_close(als_dev);
    rt_device_close(ps_dev);
    LOG_D("Als Ps Sensor Testing Ended.");

    return RT_EOK;
}
```

11.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

11.4.2 运行效果

烧录完成后，此时也可以在 PC 端使用终端工具打开开发板的 uart0 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。

```
  \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build May 31 2019
 2006 - 2019 Copyright by rt-thread team
[D/soft_i2c] software simulation i2c2soft init done, pin scl: 25, pin sda 24
[I/sensor] rt_sensor init success
[I/sensor] rt_sensor init success
[D/main] Als Ps Sensor Testing Start...
[D/main] current brightness: 8.7(lux).
[D/main] current ps data   : 14.
msh >[D/main] current brightness: 8.7(lux).
[D/main] current ps data   : 14.
[D/main] current brightness: 8.7(lux).
[D/main] current ps data   : 13.
[D/main] current brightness: 8.7(lux).
[D/main] current ps data   : 14.
[D/main] current brightness: 8.7(lux).
```

```
[D/main] current ps data   : 17.
[D/main] current brightness: 8.7(lux).
[D/main] current ps data   : 16.
[D/main] current brightness: 8.7(lux).
[D/main] current ps data   : 16.
...
[D/main] Als Ps Sensor Testing Ended.
```

11.5 注意事项

使用 ap3216c-v2.0.0 版本软件包，此软件包已经对接了 sensor 框架。

11.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《I2C 设备应用笔记》：docs/AN0003-RT-Thread-I2C 设备应用笔记
- 《ap3216c 软件包介绍》：https://github.com/RT-Thread-packages/ap3216c

第 12 章

TF 卡文件系统例程

12.1 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。

12.2 硬件说明

本次示例和存储器连接通过 SPI 接口，使用的是硬件的 SPI0，原理图如下所示：

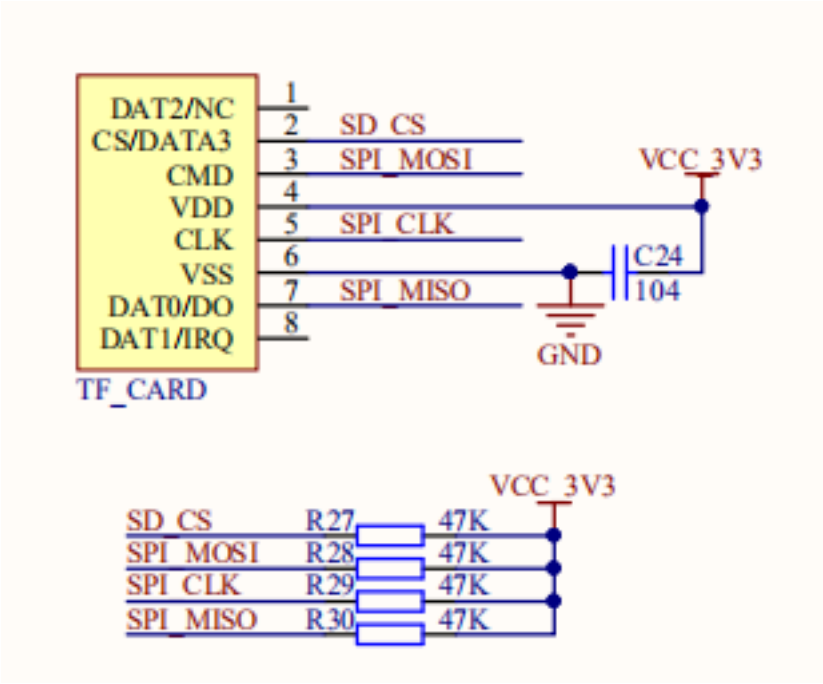


图 12.1: *TF 卡原理图*

TF 卡的卡槽在开发板中的位置如下图所示：

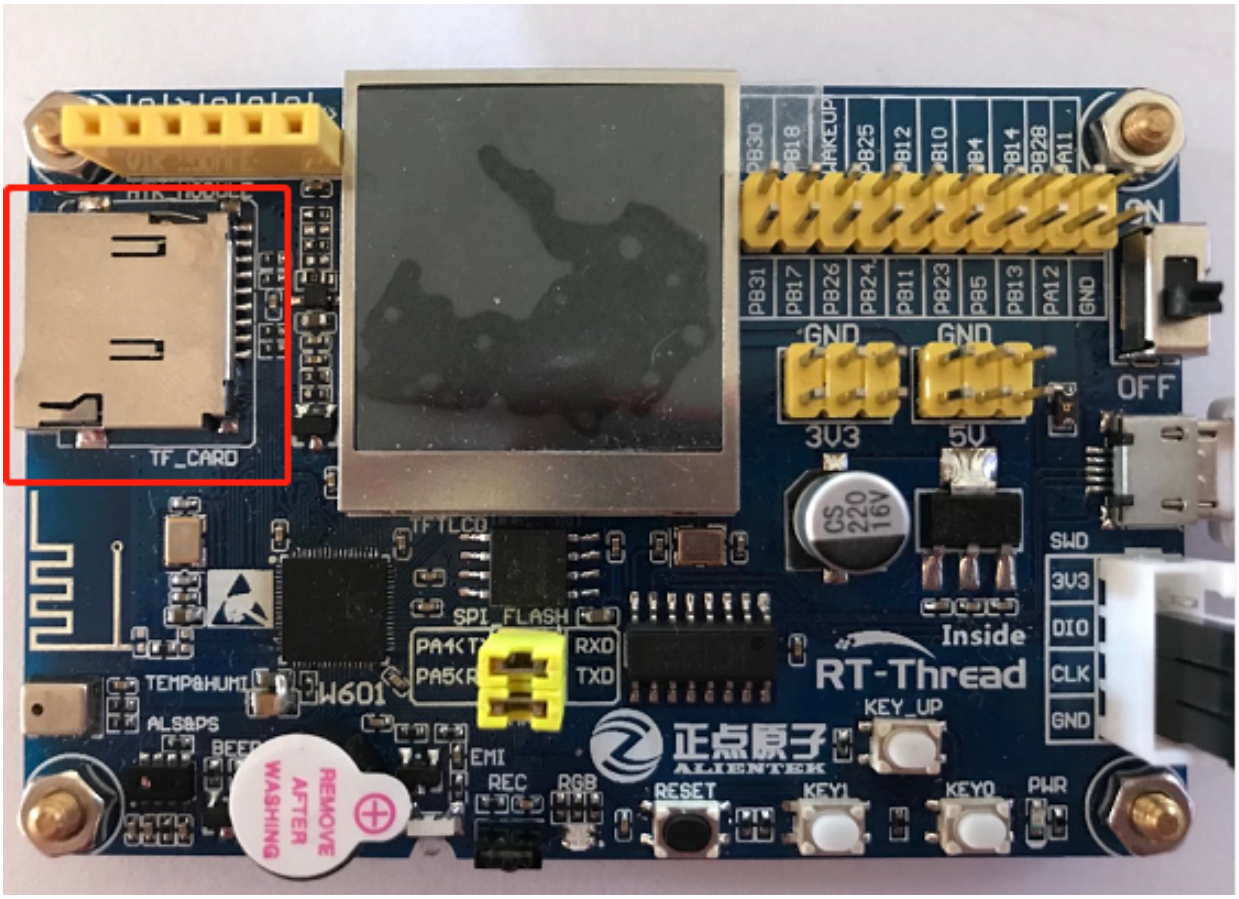


图 12.2: *TF 卡位置*

12.3 软件说明

12.3.1 挂载操作代码说明

挂载文件系统的源代码位于 `/examples/09_component_fs_tf_card/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)
{
#ifdef BSP_USING_TF_CARD
    /* 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;
}
```

12.3.2 创建块设备代码说明

在上面的挂载操作中所用的块设备 sd0 是基于 tf_spi 设备而创建的，创建块设备的代码在 drivers/drv_spi_tfcard.c 文件中。tf_spi 设备是挂载在硬件 SPI0 总线上的第一个 SPI 设备，因此命名为 tf_spi，该设备就是本次挂载的 SD 卡。msd_init 函数会在 tf_spi 设备上进行探测，并基于该设备创建名为 sd0 的块设备，用于文件系统的挂载，代码如下所示：

```
static int rt_hw_spi1_tfcard(void)
{
    return msd_init("sd0", "tf_spi");
}
```

12.3.3 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

12.3.4 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
 / | \     4.0.1 build May 22 2019
 2006 - 2019 Copyright by rt-thread team
[I/main] Filesystem initialized!
msh />
```

12.3.5 常用功能展示

12.3.6 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 /
```

12.3.7 mkdir: 创建文件夹

```
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>
```

12.3.8 echo: 将输入的字符串输出到指定输出位置

```
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  # 将字符串出输出到 hello.txt
msh />ls
Directory /:
rt-thread           <DIR>
hello.txt           18
msh />
```

12.3.9 cat: 查看文件内容

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

12.3.10 rm: 删除文件夹或文件

```
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 />
```

更多文件系统功能展示可以参考《文件系统应用笔记》。

12.4 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，否则会挂载失败。

12.5 引用参考

- 《文件系统应用笔记》：docs/AN0012-RT-Thread-文件系统应用笔记.pdf
- 《SPI 设备应用笔记》：docs/AN0004-RT-Thread-SPI 设备应用笔记.pdf

第 13 章

Flash 分区管理例程

本例程演示如何通过 RT-Thread 提供的 FAL 软件包对 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。

13.1 FAL 简介

FAL (Flash Abstraction Layer) Flash 抽象层，是 RT-Thread 的一个软件包，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

13.2 硬件说明

本例程使用到的硬件资源如下所示：

- UART0(Tx: PA4; Rx: PA5)

- 片内 FLASH (1MBytes)
- 片外 Nor Flash (16MBytes)

13.3 软件说明

fal 例程位于 `/examples/10_component_fal` 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（fal 例程程序）
ports	fal 移植文件
ports/fal/fal_cfg.h	fal 配置文件（Flash 设备配置和分区表配置）
ports/fal/fal_flash_sfud_port.c	fal 操作片外 Nor Flash 的移植文件（将 Flash 读写擦接口注册到 fal）
ports/fal/fal_flash_port.c	fal 操作片内 Flash 的移植文件（将 Flash 读写擦接口注册到 fal）
packages/fal	fal 软件包（fal 源码实现）
packages/fal/inc/fal.h	fal 软件包对外提供的操作接口

从上表中可以看到，如果要使用 fal 软件包需要进行必要的移植工作，移植文件存放在 **ports** 目录下。本例程已经完成相应的移植工作。

13.3.1 fal 配置说明

fal 配置存放在 `/examples/10_component_fal/ports/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，W601 芯片内的 Flash 和片外的 SPI Flash（Nor Flash）。

本例程的 Flash 设备列表定义如下所示：

```
extern const struct fal_flash_dev nor_flash0;
extern struct fal_flash_dev w60x_onchip;

/* flash device table */
#define FAL_FLASH_DEV_TABLE \
{                           \
    &w60x_onchip,           \
    &nor_flash0,            \
}
```

- **w60x_onchip** 是 W601 片内 Flash 设备，定义在 **/examples/10_component_fal/ports/fal_flash_port.c** 文件中，参考 **Flash 设备对接说明**章节

- **nor_flash0** 是板载 Nor FLASH 设备，定义在 **/examples/10_component_fal/ports/fal_flash_sfud_port.c** 文件中，参考 **Flash 设备对接说明**章节

13.3.2 分区表配置

分区表存放在 /examples/10_component_fal/ports/fal_cfg.h 文件中，如下所示：

```
/* partition table */
#define FAL_PART_TABLE                                                                                  \
{                                                                                                       \
    {FAL_PART_MAGIC_WROD, "app",        "w60x_onchip",                                    0, 959 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD, "easyflash",  FAL_NOR_FLASH_NAME,                               0, 1024 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD, "download",   FAL_NOR_FLASH_NAME,                    (1024) * 1024, 1024 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD, "font",       FAL_NOR_FLASH_NAME,          (1024 + 1024) * 1024, 7 * 1024 * 1024, 0}, \
    {FAL_PART_MAGIC_WROD, "filesystem", FAL_NOR_FLASH_NAME, (1024 + 1024 + 7 * 1024) * 1024, 7 * 1024 * 1024, 0}, \
}
#endif /* _FAL_CFG_H_ */
```

图 13.1: 分区表

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

成员变量	说明
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

13.3.3 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;
};
```

`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）
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

成员变量	说明
ops.write	Flash 设备数据写入接口
ops.erase	Flash 设备数据擦除接口

片内 Flash 对接说明

片内 Flash 设备实例定义在 **/examples/10_component_fal/ports/fal_flash_port.c** 文件中，如下所示：

```
struct fal_flash_dev w60x_onchip = {"w60x_onchip", 0, 0, 0, {init, read, write,
    erase}};
```

Flash 设备名称为 w60x_onchip，使用 init、read、write、erase 接口调用片上 flash 驱动进行初始化与操作。

片外 Nor Flash 对接说明

片外 Nor Flash 设备实例定义在 **/examples/10_component_fal/ports/fal_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：

```
const struct fal_flash_dev nor_flash0 = {
                    "norflash",
                    0,
                    (16 * 1024 * 1024),
                    4096,
                    {fal_sfud_init, read, write, erase}
                    };
```

Flash 设备名称为 norflash，设备容量为 16M，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。

13.3.4 例程使用说明

fal 例程代码位于 /examples/10_component_fal/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：

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**。

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。

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。

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。

13.4 运行

13.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

13.4.2 运行效果

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build May 28 2019
 2006 - 2019 Copyright by rt-thread team
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0
    x00000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offset
    @0x0000f0c8.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev   |   offset   |    length  |
[I/FAL] -------------------------------------------------------------
[I/FAL] | easyflash  | norflash    | 0x00000000 | 0x00100000 |
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash    | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash    | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash    | 0x00900000 | 0x00700000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[I/main] Flash device : norflash   Flash size : 16384K   Partition : easyflash
    Partition size: 1024K
msh />[I/main] Erase (easyflash) partition finish!
[I/main] Write (easyflash) partition finish! Write size 1048576(1024K).
[I/main] Fal partition (easyflash) test success!
[I/main] Flash device : norflash   Flash size : 16384K   Partition : download
    Partition size: 1024K
[I/main] Erase (download) partition finish!
[I/main] Write (download) partition finish! Write size 1048576(1024K).
[I/main] Fal partition (download) test success!
```

13.5 FinSH 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 norflash
  Probed a flash device | norflash | addr: 0 | len: 16777216 |.
  msh />fal probe download
  Probed a flash partition | download | flash_dev: norflash | offset: 1048576 |
      len: 1048576 |.
  msh />
  ```

- 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: norflash | offset: 1048576 |
      len: 1048576 |.
  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 .
  msh />
  ```

- 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 ................

msh />
```

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: 2.577S.
Writing 1048576 bytes data, waiting...
Write benchmark success, total time: 4.097S.
Reading 1048576 bytes data, waiting...
Read benchmark success, total time: 1.211S.
msh />
```

从日志上可以看到，`fal bench` 命令将 `download` 分区 1048576 字节大小的区域进行了擦、写、读测试，并给出了测试时间。

13.6 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
- 若写入的数据与读出的数据不一致，需要先擦除 Flash，然后再进行写入操作。

13.7 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《FAL 软件包介绍》：https://github.com/RT-Thread-packages/fal

第 14 章

KV 参数存储例程

14.1 简介

本例程将演示使用 EasyFlash 存储 KV 参数，记录开机次数。

14.2 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。

14.3 硬件说明

本例程使用到的硬件资源如下所示：

- UART0(Tx: PA4; Rx: PA5)
- 片内 FLASH (1MBytes)
- 片外 Nor Flash (16MBytes)

14.4 软件说明

14.4.1 EasyFlash 配置说明

EasyFlash 配置存放在 `/examples/11_component_kv/packages/EasyFlash-v3.3.0/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。

14.4.2 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下第 10 个例程 10_component_fal。

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 /examples/11_component_kv/packages/EasyFlash-v3.3.0/ports 复制到 /examples/11_component_kv/ports/easyflash。

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。

14.4.3 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 = NULL;
    char *c_old_boot_times, c_new_boot_times[11] = {0};

    /* 获得启动次数的值 */
    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 %d times", i_boot_times);
    LOG_D("================================================");
    /* 数字转字符串 */
    LOG_D(c_new_boot_times, "%d", i_boot_times);
    /* 保存开机次数的值 */
    ef_set_env("boot_times", c_new_boot_times);
    ef_save_env();
}
```

14.5 运行

14.5.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

14.5.2 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
 / | \     4.0.1 build May 29 2019
 2006 - 2019 Copyright by rt-thread team
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] EasyFlash V3.3.0 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] ============================================
msh />
```

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
```

14.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《FAL 软件包介绍》：https://github.com/RT-Thread-packages/fal

第 15 章

SPI Flash 文件系统例程

15.1 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《在 STM32L4 上应用 littlefs 文件系统》。

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。

15.2 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 SPI0，原理图如下所示：

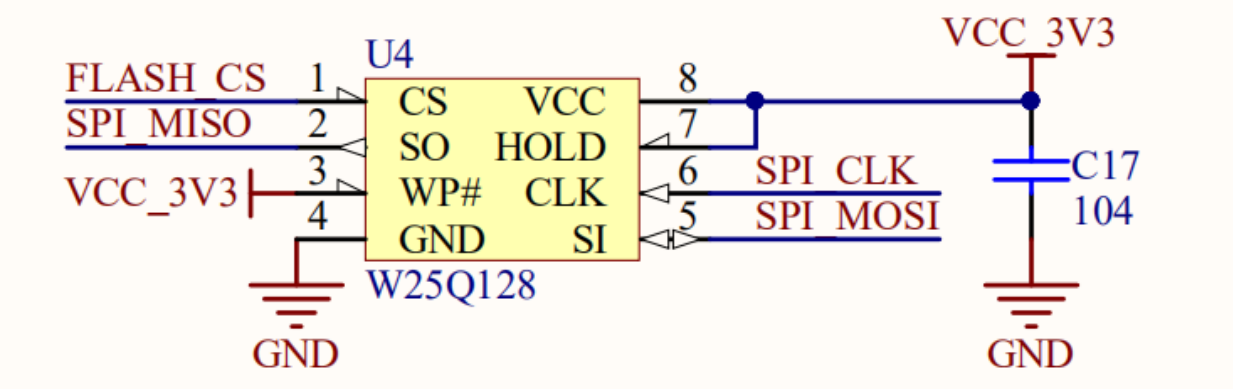


图 15.1: *SPI FLASH 原理图*

SPI FLASH 在开发板中的位置如下图所示：

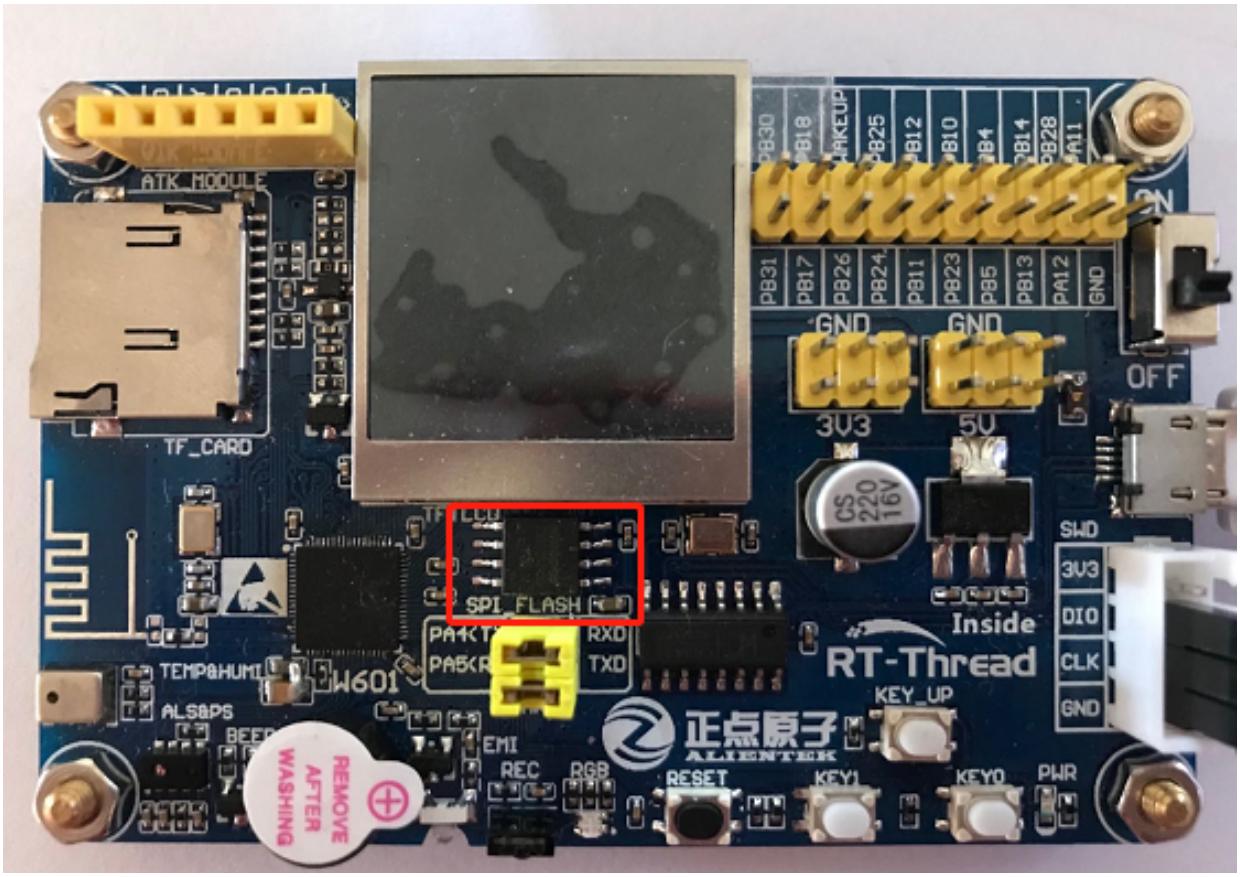


图 15.2: *SPI FLASH 位置*

15.3 软件说明

15.3.1 挂载操作代码说明

挂载文件系统的源代码位于 `/examples/12_component_fs_flash/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 `/` 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, "/", "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;
}
```

15.4 运行

15.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，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、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
 / | \     4.0.1 build May 29 2019
 2006 - 2019 Copyright by rt-thread team
```

```
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0x08
000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0x00
000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offse
t @0x0000f0c8.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev   |   offset    |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | easyflash  | norflash    | 0x00000000 | 0x00100000 |
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash    | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash    | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash    | 0x00900000 | 0x00700000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) 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!
msh />
```

15.4.3 常用功能展示

15.4.4 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 /
```

15.4.5 mkdir: 创建文件夹

```
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>
```

15.4.6 echo: 将输入的字符串输出到指定输出位置

```
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   # 将字符串出输出到 hello.txt
msh />ls
Directory /:
rt-thread           <DIR>
```

```
hello.txt           18
msh />
```

15.4.7 cat: 查看文件内容

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

15.4.8 rm: 删除文件夹或文件

```
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 />
```

更多文件系统功能展示可以参考《文件系统应用笔记》。

15.5 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。

15.6 引用参考

- 《文件系统应用笔记》：docs/AN0012-RT-Thread-文件系统应用笔记.pdf
- 《SPI 设备应用笔记》：docs/AN0004-RT-Thread-SPI 设备应用笔记.pdf
- 《FAL 软件包介绍》：https://github.com/RT-Thread-packages/fal

第 16 章

日志系统例程

本例程介绍如何使用 ulog 日志，详细介绍其使用方法以及部分常用功能。

16.1 简介

ulog 是一个非常简洁、易用的 C/C++ 日志组件，能做到最低 **ROM<1K, RAM<0.2K** 的资源占用。ulog 不仅有小巧体积，同样也有非常全面的功能，其设计理念参考的是另外一款 C/C++ 开源日志库：EasyLogger（简称 elog），并在功能和性能等方面做了非常多的改进。主要特性如下：

- 日志输出的后端多样化，可支持例如：串口、网络，文件、闪存等后端形式
- 日志输出被设计为线程安全的方式，并支持异步输出模式
- 日志系统高可靠，在中断 ISR 、Hardfault 等复杂环境下依旧可用
- 日志支持运行期 / 编译期设置输出级别
- 日志内容支持按关键词及标签方式进行全局过滤
- API 和日志格式可兼容 linux syslog
- 支持以 hex 格式 dump 调试数据到日志中
- 兼容 rtdbg （RTT 早期的日志头文件）及 EasyLogger 的日志输出 API

16.2 ulog 常规使用说明

ulog 功能强大，例程仅介绍常规使用方法，更多使用介绍与方法，请参考《ulog 日志组件应用笔记 - 基础篇》《ulog 日志组件应用笔记 - 进阶篇》。

以下说明对应例程文件位于 /examples/13_component_ulog/applications/ulog_example_a.c 。

16.2.1 日志标签

日志输出量比较大的情况下，使用标签（tag）给每条日志进行分类，避免日志杂乱无章。标签的定义是按照**模块化**的方式，每条日志的标签属性也可以被输出并显示出来，同时 ulog 还可以设置每个标签（模块）对应日志的输出级别，可以根据标签进行关闭。

在文件顶部有定义 LOG_TAG 宏：

```
#define LOG_TAG     "example.a"     // 该模块对应的标签，一般不同文件标签不一样
#define LOG_LVL     LOG_LVL_DBG     // 该模块对应的日志输出级别，默认为调试级别
#include <ulog.h>                   // 必须在 LOG_TAG 与 LOG_LVL 下面
```

- 定义日志标签必须位于 ##include <ulog.h> 的上方，否则会使用默认的 NO_TAG 与 LOG_LVL_DBG
- 日志标签的作用域是当前源码文件

16.2.2 日志级别

日志级别代表了日志的重要性，在 ulog 中由高到低，有如下几个日志级别：

级别	名称	描述
LOG_LVL_ASSE	断言	发生无法处理、致命性的的错误，以至于系统无法继续运行的断言日志
LOG_LVL_ERRO	错误	发生严重的、**不可修复**的错误时输出的日志属于错误级别日志
LOG_LVL_WAR	警告	出现一些不太重要的、具有**可修复性**的错误时，会输出这些警告日志
LOG_LVL_INFO	信息	给本模块上层使用人员查看的重要提示信息日志，例如：初始化成功，当前工作状态等。该级别日志一般在量产时依旧**保留**
LOG_LVL_DBG	调试	给本模块开发人员查看的调试日志，该级别日志一般在量产时**关闭**

在文件中代码如下

```
void ulog_example_a(void)
{
    char *RTOS = "RT-Thread is an open source IoT operating system from China.";

    /* 输出不同级别的日志 */
    LOG_D("Debug       : %s", RTOS);  /* 调试日志 */
    LOG_I("Information : %s", RTOS);  /* 信息日志 */
    LOG_W("Warning     : %s", RTOS);  /* 警告日志 */
    LOG_E("Error       : %s", RTOS);  /* 错误日志 */
}
```

- 注：断言使用 ASSERT(表达式)，触发后系统会停止运行

16.2.3 其他

LOG_RAW 输出不带任何格式的日志，不支持过滤；

LOG_HEX 日志为 DEBUG 级别，支持运行期的级别过滤与标签过滤

16.3 硬件说明

ulog 日志系统例程使用串口输入输出功能，无其他依赖。

16.4 运行

16.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译例程代码，然后将固件下载至开发板。

16.4.2 运行效果

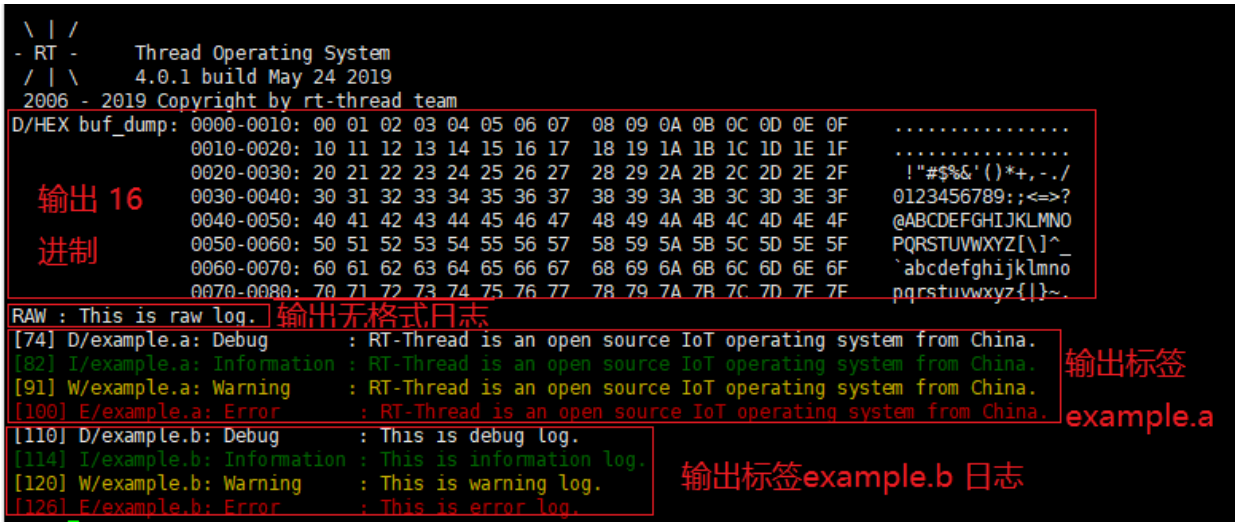


图 16.1: *启动日志*

16.4.3 按级别全局过滤

初始日志为 LOG_LVL_DBG 等级，对应的等级值为 7，对应的值可以通过命令 ulog_lvl 进行查询。下面将 log 等级设置为警告等级，则**低于设定级别**的日志都将停止输出，运行 ulog_example_a，如下

```
msh >ulog_lvl 4          # 设置日志等级
msh >ulog_example_a      # 执行 ulog_example_a 命令
[1638330] W/example.a: Warning     : RT-Thread is an open source IoT operating
    system from China.
[1638339] E/example.a: Error       : RT-Thread is an open source IoT operating
    system from China.
msh >
```

从上面日志可以看出，仅显示了日志等级大于或在等于 LOG_LVL_WARNING 的日志。

16.4.4 按关键词全局过滤

初始日志，过滤关键词为空，显示全部 log。如果查看某些特定的关键词，可以使用命令格式：ulog_kw [keyword] 进行过滤。下面关键词设为Warning，则仅**包含 Warning** 的日志都可以输出，运行 ulog_example_a，如下

```
msh >ulog_kw Warning              # 设置过滤关键词为 Warning
msh >ulog_example_a
[12961568] W/example.a: Warning     : RT-Thread is an open source IoT operating
    system from China.
```

16.4.5 其他常用命令

按模块的级别过滤：ulog_tag_lvl <tag> <level>

按标签全局过滤：ulog_tag [tag]

如果想了解更多详细内容，请参考《ulog 日志组件应用笔记 - 基础篇》与《ulog 日志组件应用笔记 - 进阶篇》。

16.5 注意事项

- 建议堆栈多预留 250 字节，系统运行崩溃
- 注意打开控制台后端，避免无输出日志
- 日志内容超出设定的日志的最大宽度，导致日志内容末尾缺失
- LOG_TAG 默认为 LOG_LVL ，LOG_LVL 默认为 LOG_LVL_DBG

16.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《ulog 日志组件应用笔记 - 基础篇》：docs/AN0022-RT-Thread-ulog 日志组件应用笔记 - 基础篇.pdf
- 《ulog 日志组件应用笔记 - 进阶篇》：docs/AN0024-RT-Thread-ulog 日志组件应用笔记 - 进阶篇.pdf

第 17 章

ADB 远程调试工具例程

本例程演示如何在 PC 端使用 ADB 工具发送和读取文件，远程 Shell 控制等功能。

17.1 工具简介

ADB 的全称为 Android Debug Bridge，是用于调试 Android 程序的 debug 工具。经过社区开发者的移植，在 RT-Thread 上也支持该工具。直接拉取 ADBD 软件包，使用 USB 或网络连接后，即可使用。

17.2 主要功能

- 文件同步

将 PC 端的文件同步到设备指定目录下。支持单文件同步与目录同步两种方式。同步完成后，设备端文件与 PC 端保持一致。

- 远程 Shell

在 ADB 工具中远程连接 Shell，用户无需连接串口设备即可完成对设备的控制、管理，满足用户对设备远程管理的需求。

- 自定义扩展模块

将远端路径封装，实现命令功能自定义。通过相应的脚本，可自由进行功能扩展。

17.3 硬件说明

本例程需要依赖 WiFi 功能完成网络通信，因此请确保硬件平台上的 WiFi 功能可以正常工作，并且能够连接网络。

17.4 配置说明

```
[*]   Using TCPIP transfer                          # 启用 TCP/IP 传输数据
[ ]   Using USB transfer                            # 启动 USB 传输数据
(2048) Set transfer thread stack size               # 设置传输线程栈大小
[*]   Enable Shell service                          # 开启 Shell 服务
-*-   Enable File service                           # 开启 文件 服务
(2304)  Set file service thread stack size          # 设置文件服务线程栈大小
(2000)  Set file service receive timeout(ms)        # 设置文件接收超时时间
[*]   Enable external MOD                           # 使能外部模块
[*]     Enable File SYNC Mod                        # 启用文件同步模块，支持校验MD5
    ，跳过相同文件
[*]     Enable File LIST Mod  --->                  # 启用获取文件目录模块
[ ]   Enable ADB service discovery                  # 启用局域网发现服务
     Version (v1.1.0)  --->
```

17.5 软件说明

ADB 例程代码位于 /examples/14_component_adbd/application/ 文件夹中，其中 main.c 主要完成文件系统的初始化，wlan 配置初始化，并等待设备联网成功。程序如下所示：

```c
int main(void)
{
    rt_device_t flash_dev;
    /* 初始化分区表 */
    fal_init();
    /* 初始化 easyflash */
    easyflash_init();

    /* 配置 wifi 工作模式 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);

    /* 初始化自动连接配置 */
    wlan_autoconnect_init();
    /* 使能 wlan 自动连接 */
    rt_wlan_config_autoreconnect(RT_TRUE);

    /* 在 filesystem 分区上创建一个 Block 设备 */
    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);
    }

    /* 挂载 FAT32 文件系统 */
    if (dfs_mount(FS_PARTITION_NAME, "/", "elm", 0, 0) == 0)
    {
```

```
        LOG_I("Filesystem initialized!");
    }
    else
    {
        /* 创建 FAT32 文件系统 */
        dfs_mkfs("elm", FS_PARTITION_NAME);
        /* 再次挂载 FAT32 文件系统 */
        if (dfs_mount(FS_PARTITION_NAME, "/", "elm", 0, 0) != 0)
        {
            LOG_E("Failed to initialize filesystem!");
        }
    }
    return 0;
}
```

17.6 运行

17.6.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

17.6.2 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
 / | \     4.0.1 build Jun  3 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[D/FAL] (fal_flash_init:63) Flash device |              w60x_onchip | addr: 0x08
000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:63) Flash device |                 norflash | addr: 0x00
000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offse
t @0x0000f0c8.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev   |   offset    |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | easyflash  | norflash    | 0x00000000 | 0x00100000 |
```

```
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash    | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash    | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash    | 0x00900000 | 0x00700000 |
[I/FAL] ============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/FAL] The FAL block device (filesystem) created successfully
[I/main] Filesystem initialized!
msh />[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.12.102
```

17.6.3 连接 ADB 设备

程序运行之后，网络连接成功并获得 IP 地址之后，在 env 终端中输入 `adb connect ip地址` 命令连接开发板。等待命令执行完成后，输入 `adb devices` 命令查看已经连接的设备。

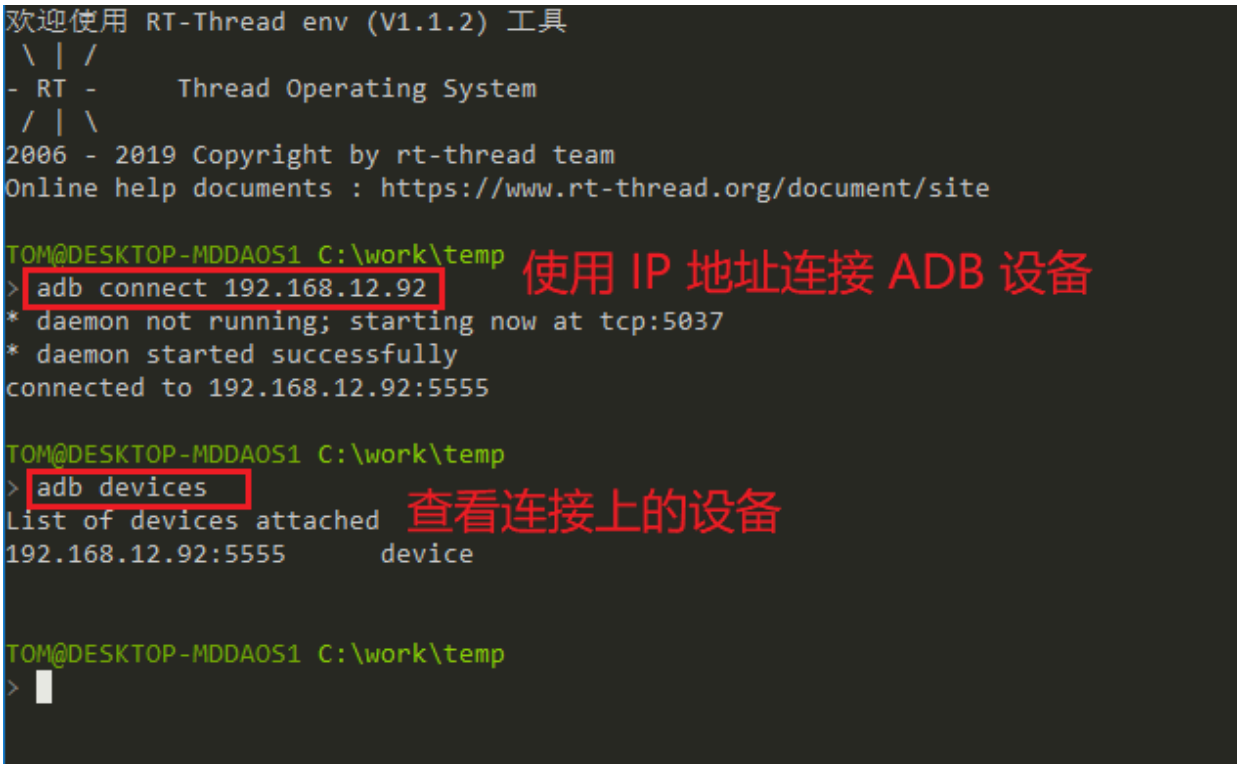


图 17.1: *ADB connect*

17.6.4 ADB Shell 功能

ADB 连接设备成功之后，在 env 终端中输入 `adb shell` 命令，进入开发板终端。此时即可实现远程 Shell 控制功能，用户无需连接串口设备即可完成设备的管理和调试，并且实时显示设备打印信息。输入 `exitas` 命令，退出远程 Shell。

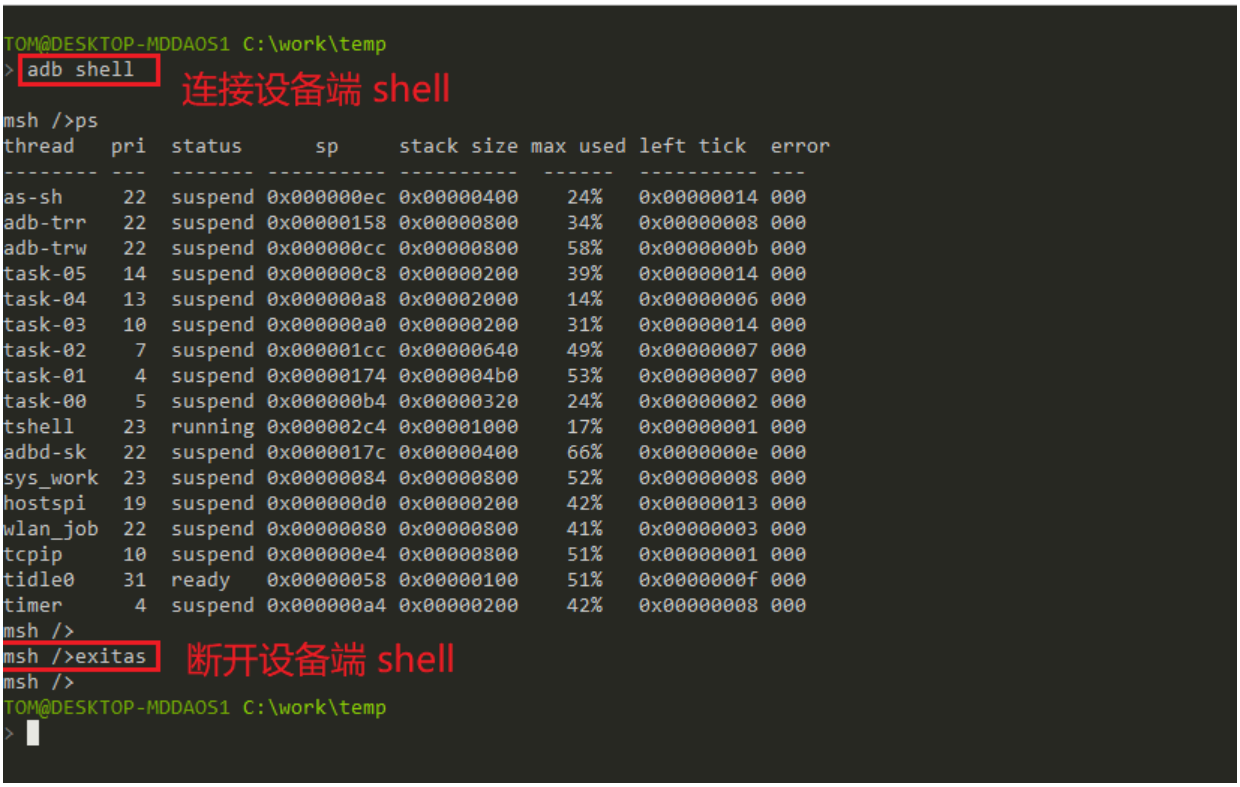


图 17.2: *ADB shell*

17.6.5 文件推送功能

文件推送功能是将 PC 端的文件或目录，传输到设备上。相关的命令为：`adb push LOCAL... REMOTE`。其中 `LOCAL` 表示 PC 端文件名或目录。`REMOTE` 表示发送到设备之后的文件名或目录。

例如：将 PC 的 hello.py 文件推送到设备根目录下，并命名为 hello_dev.py，在命令行输入 `adb push hello.py hello_dev.py` 即可完成。

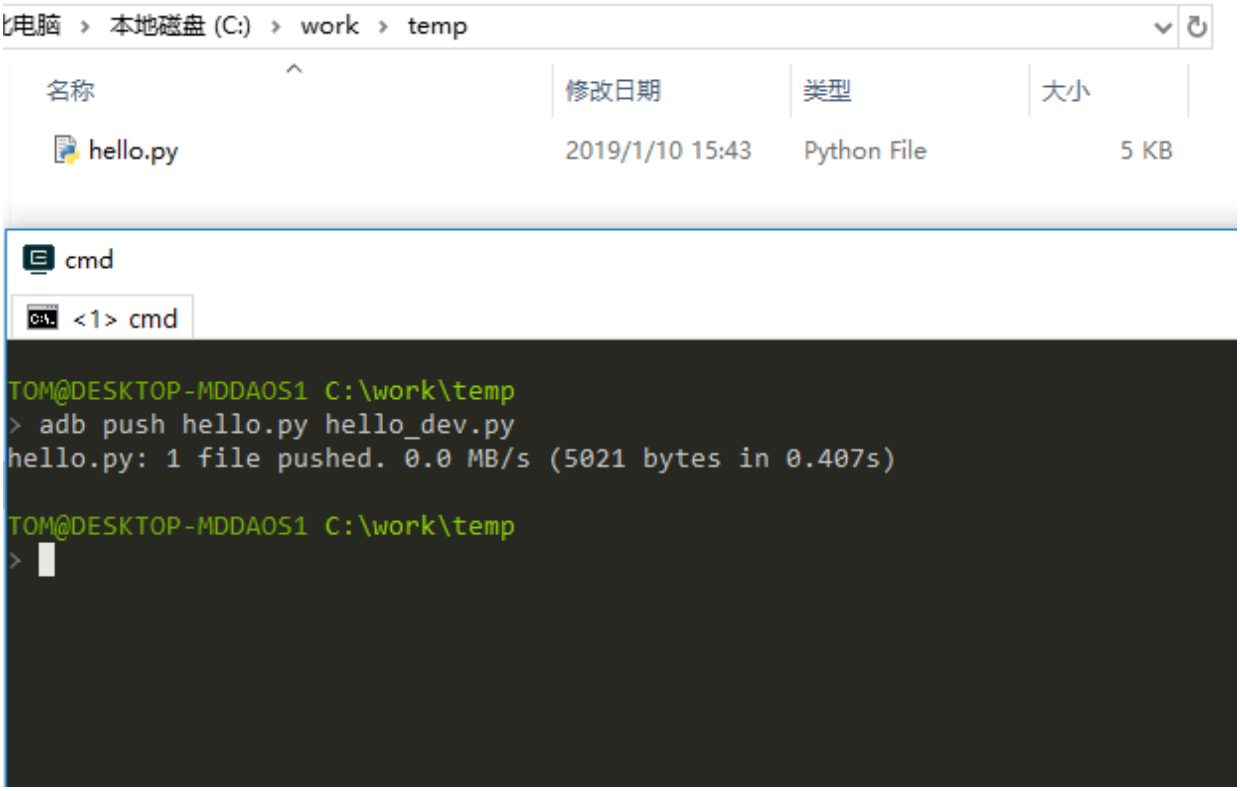


图 17.3: *ADB push*

传输文件完成之后，在设备端输入 `ls` 命令，查看文件。可以看到 PC 端推送过来的文件。

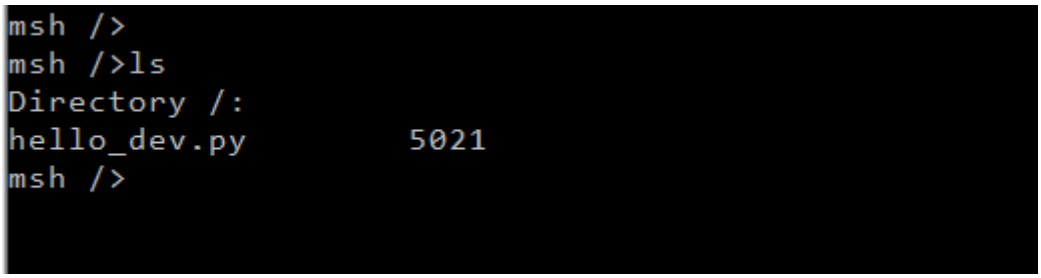


图 17.4: *ADB push*

17.6.6 文件拉取功能

文件拉取功能是将设备端的文件或目录，传输到 PC 上。相关的命令为：`adb pull REMOTE... LOCAL`。其中 `REMOTE` 表示设备端的文件名或目录。`LOCAL` 发送到 PC 端之后文件名或目录。

例如：将设备端的 hello_dev.py 文件传输到 PC 上，并命名为 hello_pc.py，在命令行输入 `adb pull hello_dev.py hello_pc.py` 即可完成。

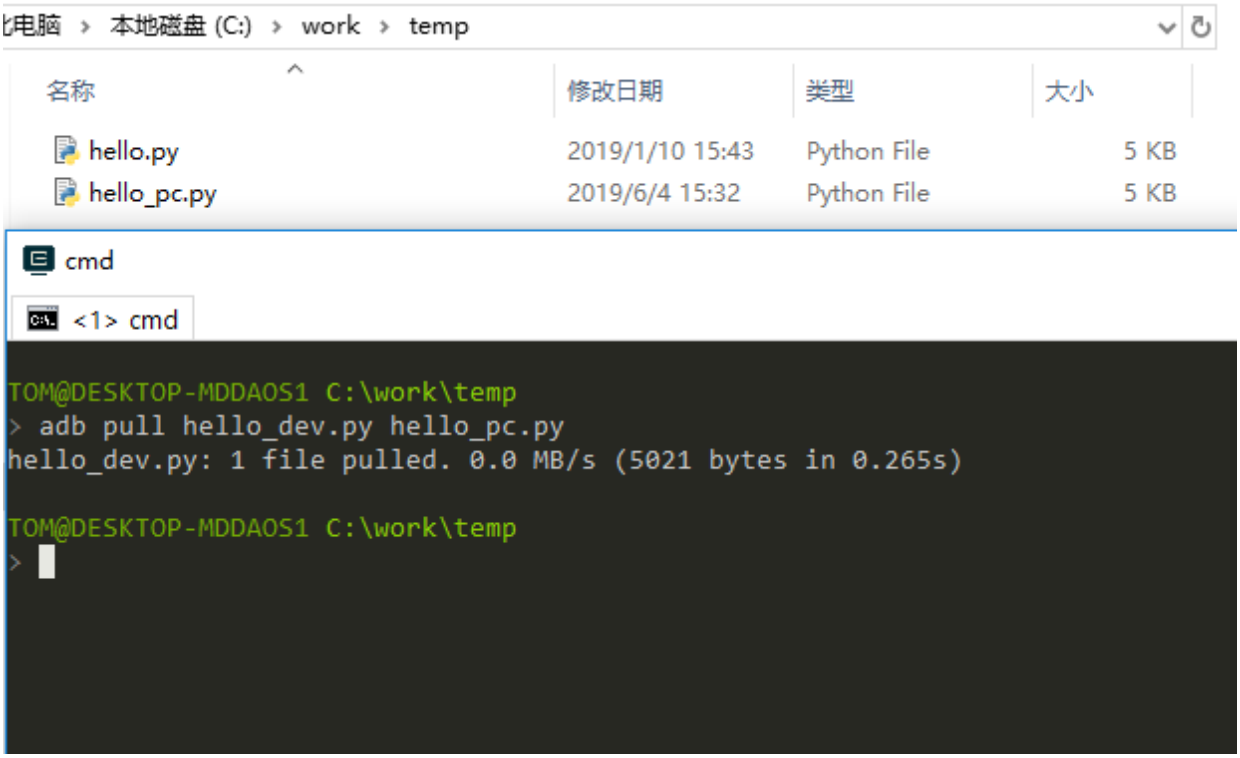


图 17.5: *ADB pull*

17.6.7 断开 ADB 设备

在 env 终端中，输入 `adb disconnect` 命令，断开所有连接的设备。

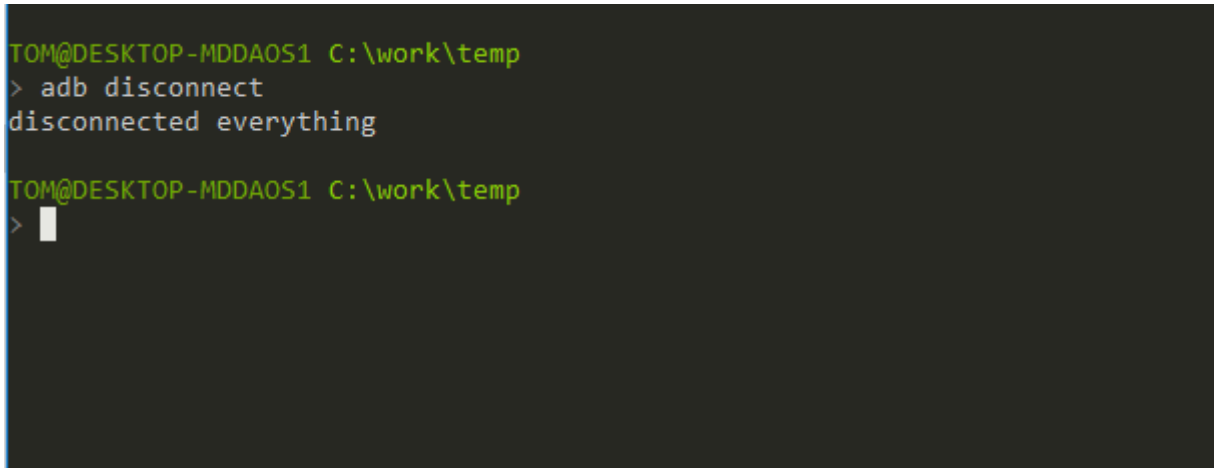


图 17.6: *ADB disconnect*

17.7 注意事项

本例程需要连接 WIFI ，开始例程之前确保 WIFI 已经连接并获取到 IP 地址。例程中已经启动 WIFI 自动连接功能，如果等待一段时间后，仍没有连接成功，则需要手动连接。使用 MSH 命令 `wifi join <ssid> <password>` 可让设备接入网络。

17.8 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 18 章

WiFi 管理例程

18.1 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。

18.2 硬件说明

本例程用到联盛德新一代支持多接口、多协议的无线局域网 IEEE802.11n（1T1R）低功耗、自带的 wifi SoC 芯片。芯片内置 Cortex-M3 CPU 处理器和 Flash，集成射频收发前端 RF Transceiver，CMOS PA 功率放大器，基带处理器/媒体访问控制，集成电源管理电路，支持丰富的外围接口，支持多种加解密协议。原理图如下所示

TODO :

18.3 软件说明

Wlan Manager 位于 `/examples/16_iot_wifi_manager` 目录下，WiFi 模组的部分代码以库文件的形式提供，重要文件摘要说明如下表所示：

文件	说明
applications/main.c	app 入口（wifi manager 例程程序）
ports/wifi	Wlan 配置信息储存的移植文件 (将已连接 AP 信息存储至 Flash)
../../drivers/drv_wifi.c	底层 wlan 驱动

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。

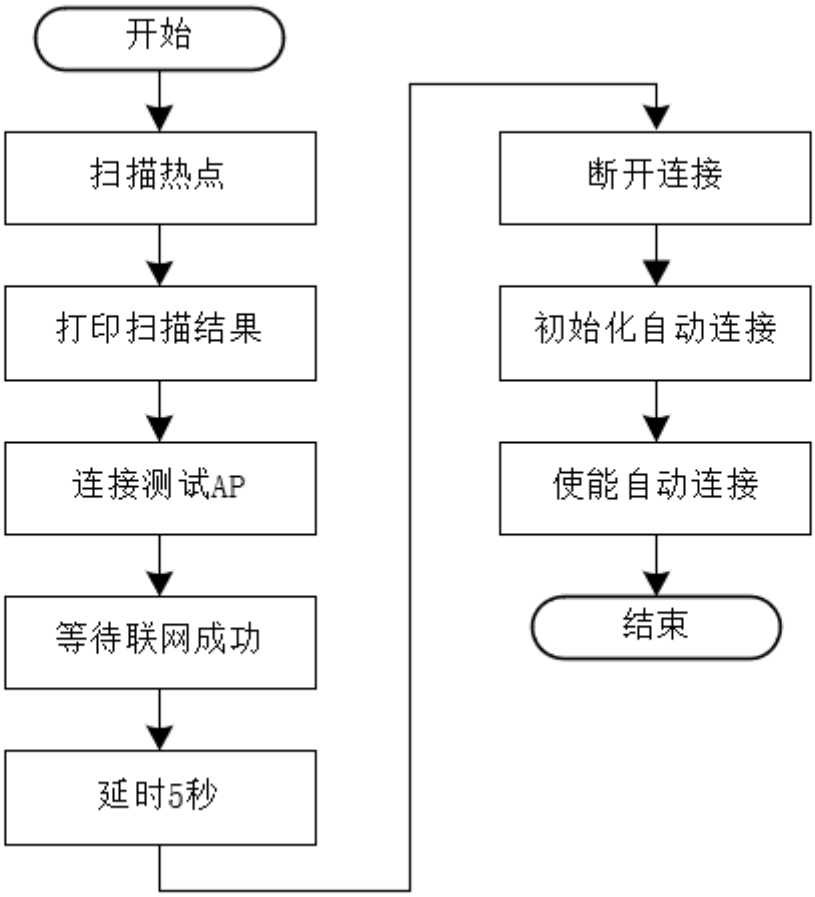


图 18.1: *wlan manager 流程*

18.3.1 热点扫描

下面代码段展示热点扫描功能，为同步式扫描，扫描的结果会通过接口直接返回。

```
    /* 扫描热点 */
    LOG_D("start to scan ap ...");
    /* 执行同步扫描 */
    scan_result = rt_wlan_scan_sync();
    if (scan_result)
    {
        LOG_D("the scan is complete, results is as follows: ");
        /* 打印扫描结果 */
        print_scan_result(scan_result);
        /* 清除扫描结果 */
        rt_wlan_scan_result_clean();
    }
    else
    {
```

```
        LOG_E("not found ap information ");
        return -1;
    }
```

18.3.2 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 */
    rt_thread_mdelay(100);
    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);
    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);
        /* 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。

18.3.3 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 ...");
    /* 初始化自动连接配置 */
    wlan_autoconnect_init();
    /* 使能 wlan 自动连接 */
    rt_wlan_config_autoreconnect(RT_TRUE);
```

自动连接功能需要用户实现参数信息存取的接口，本例中采用 KV 的方式进行存储，实现代码位于/examples/16_iot_wifi_manager/ports/wifi/wifi_config.c。

```
static int read_cfg(void *buff, int len);
static int get_len(void);
static int write_cfg(void *buff, int len);

static const struct rt_wlan_cfg_ops ops =
{
    read_cfg,
    get_len,
    write_cfg
};
```

- auto_connect 移植接口说明

接口	描述
read_cfg	从存储介质中读取配置信息
get_len	从存储介质中读取配置信息长度
write_cfg	写入 wlan 配置信息至存储介质

18.4 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
```

18.4.1 WiFi 扫描

- wifi 扫描命令格式如下

```
wifi scan
```

命令说明

字段	描述
wifi	有关 wifi 命令都以 wifi 开头
scan	wifi 执行扫描动作

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：

```
msh />wifi scan
SSID                          MAC            security    rssi chn Mbps
------------------------------ -----------------  -------------- ---- --- ----
rtt_test_ssid_1                c0:3d:46:00:3e:aa  OPEN           -14    8  300
test_ssid                      3c:f5:91:8e:4c:79  WPA2_AES_PSK   -18    6   72
rtt_test_ssid_2                ec:88:8f:88:aa:9a  WPA2_MIXED_PSK -47    6  144
rtt_test_ssid_3                c0:3d:46:00:41:ca  WPA2_MIXED_PSK -48    3  300
msh />
```

18.4.2 WiFi 连接

- wifi 连接命令格式如下

```
wifi join [SSID PASSWORD]
```

- 命令解析

字段	描述
wifi	有关 wifi 命令都以 wifi 开头
join	wifi 执行连接动作

字段	描述
SSID	热点的名字
PASSWORD	热点的密码，没有密码可不输入这一项

- 连接成功后，将在终端上打印获得的 IP 地址，如下图所示

```
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
msh />

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'字段。
```

18.4.3 WiFi 断开

- wifi 断开命令格式如下

```
wifi disc
```

- 命令解析

字段	描述
wifi	有关 wifi 命令都以 wifi 开头
disc	wifi 执行断开动作

- 断开成功后，将在终端上打印如下信息，如下图所示

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

18.5 运行

18.5.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 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

按下复位按键重启开发板，正常运行后，会依次执行扫描、联网、开启自动连接等功能，终端输出信息如下：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build Apr 30 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/SAL_SKT] Socket Abstraction Layer initialize success.
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0x08
000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0x00
000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offse
t @0x0000ffc0.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev   |   offset   |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | easyflash  | norflash    | 0x00000000 | 0x00100000 |
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash    | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash    | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash    | 0x00900000 | 0x00700000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x0000
0000, size is 4096 bytes.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number
is 0x9B83516C.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is
OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[D/main] start to scan ap ...            # 开始扫描
msh />[D/main] the scan is complete, results is as follows:
            SSID                     MAC            security    rssi chn Mbps
```

```
------------------------------ -----------------  -------------- ---- --- ----
aptest                         ec:88:8f:88:aa:9a  WPA2_MIXED_PSK -34   13  300
realthread                     dc:fe:18:67:4f:35  WPA2_MIXED_PSK -40    1  450
realthread_VIP                 0e:fe:18:67:4f:35  WPA2_MIXED_PSK -42    1  450

[D/main] start to connect ap ...          # 连接热点
[I/WLAN.mgnt] wifi connect success ssid:aptest
[D/main] station information:
[D/main] SSID : test_ssid
[D/main] MAC Addr: ec:88:8f:88:aa:9a
[D/main] Channel: 13
[D/main] DataRate: 144Mbps
[D/main] RSSI: -36
[D/main] networking ready!
network interface: w0 (Default)
MTU: 1500
MAC: 28 6d cd 3a 73 ec
FLAGS: UP LINK_UP ETHARP BROADCAST IGMP
ip address: 192.168.12.83
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
[I/WLAN.lwip] Got IP address : 192.168.12.83
[D/main] ready to disconect from ap ...
[I/WLAN.mgnt] disconnect success!         # 成功断开连接
[D/main] start to autoconnect ...         # 自动连接
[I/main] callback from the offline network!
[I/WLAN.mgnt] wifi connect success ssid:test_ssid
[I/WLAN.lwip] Got IP address : 192.168.12.83
```

18.6 联网失败处理

正常连接 WiFi 时，在连接成功后会打印分配到的 IP 信息，如 `[I/WLAN.lwip] Got IP address : 192.168.12.115`。当 AP 不存在或者密码错误，上述信息将不会打印，此时，请检查 SSID 、密码是否正确，以及网络配置。下面为 AP 不存在和密码错误时的信息：

```
/* AP不存在 */
msh />wifi join test_ssid 12345678
[W/WLAN.mgnt] F:rt_wlan_connect L:979 not find ap! ssid:test_ssid
msh />
msh />
/* 密码错误 */
msh />wifi join test_ssid 1234567890
join ssid:test_ssid
...
```

18.7 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。

18.8 引用参考

- 《WLAN 框架应用笔记》：docs/AN0026-RT-Thread-WLAN 框架应用笔记.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 19 章

使用 web 快速接入 WiFi 网络

本例程基于 WM SDK 提供的库文件，主要讲解 web 配网的基本原理、配网流程以及代码实现流程。

19.1 原理简介

支持 AP 模式的 WiFi 设备上内嵌一个简单的 web 服务器，在 web 网页里面提供一个配网的交互接口。WiFi 设备进入 AP 模式，其他设备包括手机等通过 AP 热点建立连接。建立连接之后，通过网页将其他路由器或者 AP 的 SSID 与 key 发送给 WiFi 设备。交互流程如下图所示

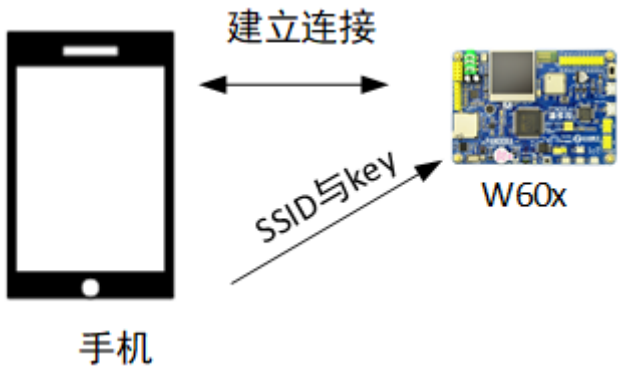


图 19.1: *设备联网示意图*

19.2 硬件说明

web 配网使用到 WiFi 相关功能，需要手机连接 WiFi 设备之后打开网页。

19.3 软件说明

AirKiss 使用例程位于 `/examples/17_iot_web_config_wifi/applications/main.c` 文件中，main 函数主要配置 WiFi 工作模式与启动一键配网功能，main 函数实现功能如下

```
int main(void)
{
    rt_err_t result = RT_EOK;
    /* 配置 wifi 工作模式 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    rt_wlan_set_mode(RT_WLAN_DEVICE_AP_NAME, RT_WLAN_AP);
    rt_sem_init(&web_sem, "web_sem", 0, RT_IPC_FLAG_FIFO);

    rt_thread_mdelay(2000);
    /* 一键配网：APWEB 模式 */
    result = wm_oneshot_start(WM_APWEB, oneshot_result_cb);
    if (result != 0)
    {
        LOG_E("web config wifi start failed");
        return result;
    }

    LOG_D("web config wifi start...");
    result = rt_sem_take(&web_sem, NET_READY_TIME_OUT);
    if (result != RT_EOK)
    {
        LOG_E("connect error or timeout");
        return result;
    };
    /* 连接 WiFi */
    result = rt_wlan_connect(ssid, passwd);
    if (result != RT_EOK)
    {
        LOG_E("\nconnect ssid %s key %s error:%d!", ssid, passwd, result);
    };
    return result;
}
```

启动一键配网函数的回调，实现了使用获取的 SSID 与 passwd 进行 WiFi 连接的功能。

```
static void oneshot_result_cb(int state, unsigned char *ssid_i, unsigned char *
    passwd_i)
{
    char *ssid_temp = (char *)ssid_i;
    char *passwd_temp = (char *)passwd_i;

    /* 配网回调超时返回 */
    if (state != 0)
    {
        LOG_E("Receive wifi info timeout(%d). exit!", state);
        return;
    }
    if (ssid_temp == RT_NULL)
    {
```

```
        LOG_E("SSID is NULL. exit!");
        return;
    }
    LOG_D("Receive ssid:%s passwd:%s", ssid_temp == RT_NULL ? "" : ssid_temp,
        passwd_temp == RT_NULL ? "" : passwd_temp);

    strncpy(ssid, ssid_temp, strlen(ssid_temp));
    if (passwd_temp)
    {
        strncpy(passwd, passwd_temp, strlen(passwd_temp));
    }
    /* 通知 ssid 与 key 接收完成 */
    rt_sem_release(&web_sem);
}
```

19.4 运行

19.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译例程代码，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
 / | \     4.0.1 build May 22 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w1
msh />[I/WLAN.mgnt] start ap successs!
[D/main] oneshot start...
[DHCP] dhcpd_start: w1          # 分配 ip
[DHCP] ip_start: [192.168.169.2]
[DHCP] ip_start: [192.168.169.254]
```

手机连接开发板的热点，例程中名称为 softap_73b2(具体的可能有所变化)，如下图所示

图 19.2: 二维码

然后打开浏览器，输入192.168.169.1，浏览器进入如下界面，然后再 ssid 栏中写入将要连接的热点名称，pwd 栏中输入对应 ssid 的密码，例程中 ssid 为 aptest，pwd 为 123456789，输入如下图所示

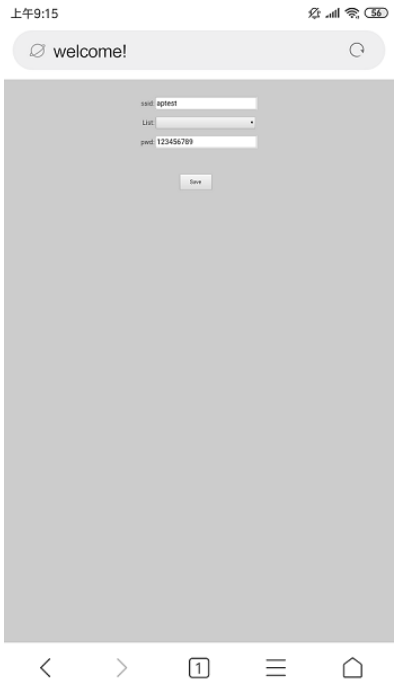


图 19.3: 二维码

然后点击按钮 save -> 确定，如下图所示

图 19.4: 二维码

点击完成后，查看日志，显示如下

```
[I/WLAN.mgnt] sta associated mac:f4:70:ab:72:20:66
[I/WLAN.mgnt] sta exit mac:f4:70:ab:72:20:66
[I/WLAN.mgnt] ap stop success!
Receive ssid:aptest passwd:123456789            # 打印接收的 ssid 与 key
connect wifi:aptest
[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.12.240
networking ready!                               # 网络连接成功
```

19.5 注意事项

- 待连接的路由器必须支持 2.4G

19.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《WM_W60X_ 一键配网使用指导 _V1.1》：libraries/WM_Libraries/Doc/WM_W60X_ 一键配网使用指导 _V1.1.pdf

第 20 章

使用 AirKiss 快速接入 WiFi 网络

本例程将介绍 AirKiss 快速接入 WiFi 网络，主要讲解配网的基本原理、配网流程以及代码实现流程。

20.1 简介

AirKiss 是微信硬件平台提供的一种 WiFi 设备快速入网配置技术，要使用微信客户端的方式配置设备入网，且需要设备支持 AirKiss 技术。

20.2 原理简介

AirKiss 配置设备联网过程中，主要包含以下基本内容

- 开发板以 station 混杂模式运行
- 手机微信客户端通过 AirKiss 发送 SSID 和密码
- 开发板通过抓包获取到 SSID 和密码，然后连接 WiFI

大致流程如下图所示

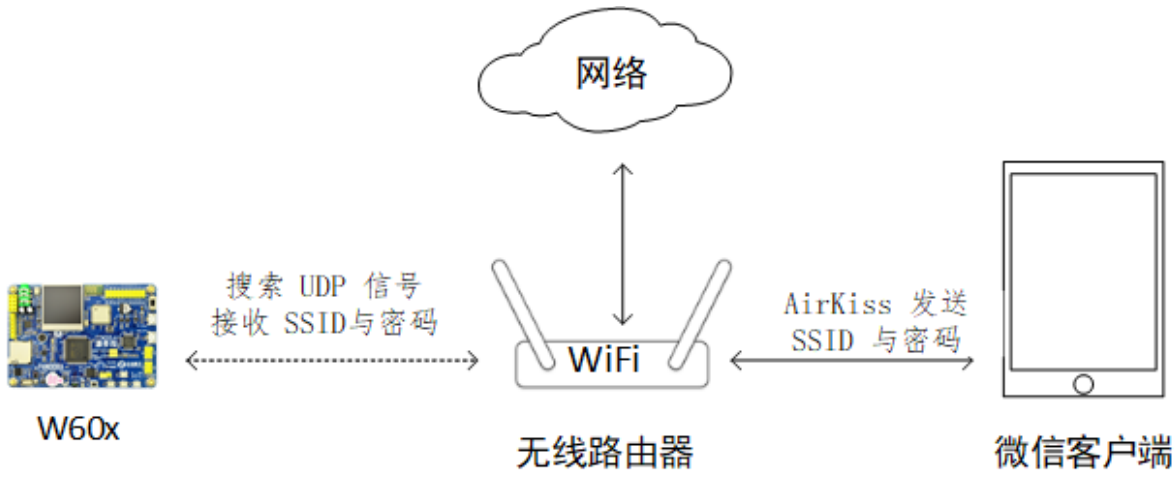


图 20.1: *设备联网示意图*

如上图所示，微信客户端将通过 AirKiss 通信协议将 WiFi 的 SSID 与密码，发送一系列的 UDP 广播包，路由器接收后进行转播，被置为 station 混杂模式的 WiFi 设备监听空中的广播包，WiFi 设备可根据 AirKiss 通信协议解析出需要的信息，获取到 SSID 与密码后进行网络连接。

更多关于 AirKiss 原理介绍参考链接 https://iot.weixin.qq.com/wiki/new/index.html?page=4-1-1。

20.3 硬件说明

AirKiss 使用到 LCD 显示二维码、WiFi 相关功能。

20.4 软件说明

AirKiss 使用例程位于 /examples/18_iot_airkiss/applications 目录下，配置 wifi 工作模式后启动一键配网功能，并且屏幕显示特定二维码图片，main 函数实现功能如下

```
int main(void)
{
    rt_err_t result;
    /* 清屏 */
    lcd_clear(WHITE);
    /* 配置 wifi 工作模式 */
    result = rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    if (result == RT_EOK)
    {
        LOG_D("Start airkiss...");
        /* 一键配网 demo */
        smartconfig_demo();
    }
    /* 显示二维码 */
    iotb_lcd_show_wechatscan();

    return 0;
}
```

其中一键配网 demo 具体实现位于../../libraries/smartconfig/smartconfig_demo.c，实现函数如下

```
void smartconfig_demo(void)
{
    rt_smartconfig_start(SMARTCONFIG_TYPE_AIRKISS, SMARTCONFIG_ENCRYPT_NONE, RT_NULL
        , smartconfig_result);
}
```

SMARTCONFIG_TYPE_AIRKISS 表示配网类型为 AirKiss；SMARTCONFIG_ENCRYPT_NONE 表示无加密；smartconfig_result 为配网回调函数，主要打印配网信息以及返回通知。

20.5 运行

20.5.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译例程代码，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
 / | \     4.0.1 build May 21 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/DBG] config_type:AIRKISS          # 初始化成功
```

屏幕显示二维码，如下图所示

图 20.2: 二维码

微信扫码，填入手机连接的 WiFi 密码，点击连接，如下图所示

图 20.3: 二维码

连接成功后显示日志如下

```
msh >[I/DBG] locked channel 10
[I/DBG] ssid:aptest
[I/DBG] passwd:123456789
[I/DBG] radom:d7
[I/DBG] airkiss finish!
[I/DBG] smartconfig finish!
type:0
ssid:aptest
passwd:123456789
user_data:0xd7
[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.12.240
networking ready!                        # 连接网络成功
airkiss notification thread exit!
```

20.6 注意事项

- 手机需要连接使用 2.4G 的路由器

20.7 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- qrcode 软件包：https://github.com/RT-Thread-packages/qrcode

第 21 章

AT 指令（服务器端）例程

本例程基于 RT-Thread 系统的 AT 组件，主要介绍如何使用 AT 服务器端，即通过添加 AT 指令，控制 LED 灯的开关。如果需要了解更多的 AT Server 使用例程，可以参考软件包 socket 指令的 AT 服务端。

21.1 简介

AT 指令（AT Commands）最早是为了控制拨号调制解调器（MODEM）的控制协议，经过系列的发展，逐步形成了一定的标准化，但没有完全统一，很难做到对上层网络应用接口的适配。

RT-Thread 提供方便用户使用 AT 命令组件，只需要简单适配即可实现 AT Server 和 AT Client 两部分功能。该组件完成 AT 命令的发送、命令格式及参数判断、命令的响应、响应数据的接收、响应数据的解析、URC 数据处理等整个 AT 命令数据交互流程。

本例程主要使用 **AT Server**，其主要功能特点如下：

- 基础命令：实现多种通用基础命令（ATE、ATZ 等）；
- 命令兼容：命令支持忽略大小写，提高命令兼容性；
- 命令检测：命令支持自定义参数表达式，并实现对接收的命令参数自检测功能；
- 命令注册：提供简单的用户自定义命令添加方式，类似于 `finsh/msh` 命令添加方式；
- 调试模式：提供 AT Server CLI 命令行交互模式，主要用于设备调试。

21.2 AT Server 使用说明

RT-Thread 的 AT 组件功能强大，例程仅使用了 AT Server 的相关功能，更详细介绍参考《AT 组件》。

AT 命令根据功能进行划分，有如下四种格式::

- 测试功能：`AT+<x>=?` 用于查询命令参数格式及取值范围；
- 查询功能：`AT+<x>?` 用于返回命令参数当前值；
- 设置功能：`AT+<x>=...` 用于用户自定义参数值；
- 执行功能：`AT+<x>` 用于执行相关操作。

每个命令的四种功能并不需要全部实现，用户自定义添加 AT Server 命令时，可根据自己需求实现一种或几种上述功能函数，未实现的功能可以使用 NULL 表示，再通过自定义命令添加函数添加到基础命令列表，添加方式类似于 finsh/msh 命令添加方式，添加函数如下：

```
AT_CMD_EXPORT(_name_, _args_expr_, _test_, _query_, _setup_, _exec_);
```

参数	描述
name	AT 命令名称
_args_expr_	AT 命令参数表达式；（无参数为 NULL，<> 中为必选参数，[] 中为可选参数）
test	AT 测试功能函数名；（无实现为 NULL）
query	AT 查询功能函数名；（同上）
setup	AT 设置功能函数名；（同上）
exec	AT 执行功能函数名；（同上）

21.3 硬件说明

AT 指令（服务器端）例程控制台使用串口 0，AT 指令交互使用串口 2，以及红色 LED 灯，其中串口 2 如下图所示

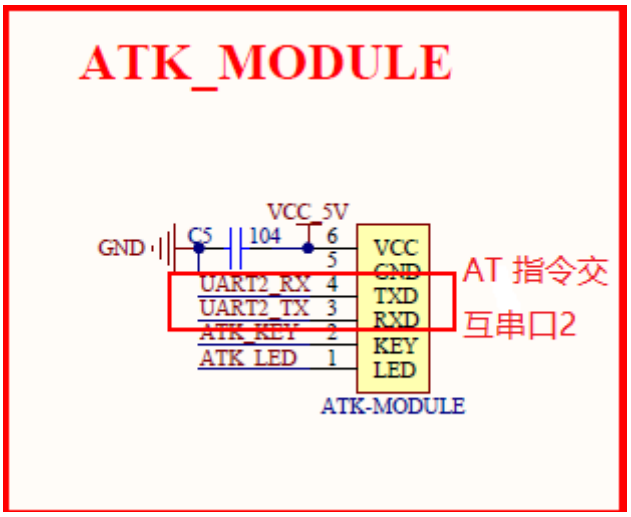


图 21.1: *LED 交互*

21.4 软件说明

AT server 使用例程位于 /examples/20_iot_at_server/applications 目录下，主要程序在 main.c 文件中，主要实现 AT 自定义命令的添加功能，完成通过 AT 指令控制灯的开关。

自定义 AT 命令

```
AT_CMD_EXPORT("AT+LED", "=<value>", led_parameter, led_query, led_setup, RT_NULL);
```

参数	描述
“AT+LED”	AT 命令名称
“=”	AT 命令参数表达式
led_parameter	AT 查询参数函数名，查询 LED 灯可设置状态
led_query	AT 查询功能函数名，获取 LED 灯的状态
led_setup	AT 设置功能函数名，设置 LED 灯的状态

查询 LED 参数

```
static at_result_t led_parameter()
{
    /* 返回当前灯可设置参数到 AT 客户端 */
    at_server_printf("AT+LED= 0, 1");
    return AT_RESULT_OK;
}
```

查询 LED 状态

```
static at_result_t led_query(void)
{
    /* 返回客户端获取 LED 状态 */
    at_server_printf("AT+LED: %c", led_status);
    return AT_RESULT_OK;
}
```

开关 LED

```
static at_result_t led_setup(const char *args)
{
    at_result_t  result;
    /* 解析得到结果存入 led_status */
    if (at_req_parse_args(args, "=%c", &led_status) > 0)
    {
        if ('0' == led_status)
        {
            /* LED 灯灭 */
            rt_pin_write(LED_PIN, PIN_HIGH);
        }
        else
        {
            /* LED 灯亮 */
            rt_pin_write(LED_PIN, PIN_LOW);
        }
        result = AT_RESULT_OK;
    }
    else
```

```
    {
        result = AT_RESULT_PARSE_FAILE;
    }
    return result;
}
```

21.5 运行

21.5.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译例程代码，然后将固件下载至开发板。

程序运行日志（串口 0 输出）如下所示：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build May 16 2019
 2006 - 2019 Copyright by rt-thread team
[I/at.svr] RT-Thread AT server (V1.2.0) initialize success.
msh >
```

AT 指令串口交互命令（串口 2 命令交互）如下图所示：

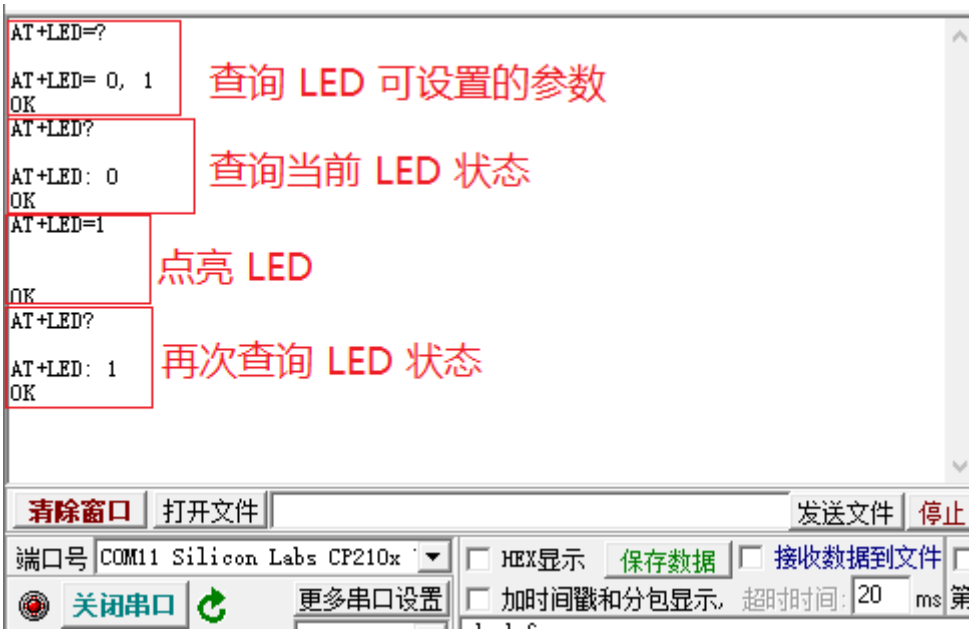


图 21.2: *LED 交互*

21.6 注意事项

- 避免 AT 指令中存在空格对指令识别造成影响
- 注意问号为半角
- AT 指令交互串口与控制台是两个不同的串口

21.7 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《AT 组件》：https://www.rt-thread.org/document/site/programming-manual/at/at/

第 22 章

MQTT 协议通信例程

22.1 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。

22.2 硬件说明

本例程需要依赖板 W601 WIFI 芯片完成网络通信。

22.3 软件说明

22.3.1 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。

22.3.2 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。

22.3.3 例程使用说明

本示例的源代码位于 /examples/21_iot_mqtt/applications/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：

```
int main(void)
{
    /* 配置 wifi 工作模式 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    /* 注册 MQTT 启动函数为 WiFi 连接成功的回调函数 */
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY,(void (*)(int , struct
        rt_wlan_buff *, void *))mq_start,RT_NULL);
    /* 初始化 WiFi 自动连接 */
    wlan_autoconnect_init();
    /* 使能 WiFi 自动连接 */
    rt_wlan_config_autoreconnect(RT_TRUE);
}

/* MQTT 启动函数 */
static void mq_start(void)
```

```
{
    ......

    /* 配置 MQTT 客户端参数 */
    {
        client.isconnected = 0;
        client.uri = MQTT_URI;

        /* 随机生成 ID 和 订阅&发布的主题 */
        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 = MQTT_USERNAME;
        client.condata.password.cstring = MQTT_PASSWORD;

        ......

        /* 设置回调函数 */
        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;

        ......
    }

    /* 启动 MQTT 客户端 */
    rt_kprintf("Start mqtt client and subscribe topic:%s\n", 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;
}

/* MQTT 在线回调函数 */
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!");
}
```

22.4 运行

22.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

22.4.2 运行效果

按下复位按键重启开发板，开发板会自动连上 WiFi ，可以看到板子会打印出如下信息：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.2 build Sep 10 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x0000
0000, size is 4096 bytes.
```

```
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number
is 0xD6363A94.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is
OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
msh />[I/WLAN.mgnt] wifi connect success ssid:test
[D/main] Start mqtt client and subscribe topic:/mqtt/test/rtthread5125        #启动
    MQTT 客户端
[I/main] Start to connect mqtt server
[D/MQTT] ipv4 address port: 1883
[D/MQTT] HOST = 'mqtt.rt-thread.com'
[I/WLAN.lwip] Got IP address : 192.168.12.49
[I/MQTT] MQTT server connect success
[I/MQTT] Subscribe #0 /mqtt/test/rtthread9660 OK!
[D/main] Connect mqtt server success
[D/main] Publish message: Hello,RT-Thread! to topic: /mqtt/test/rtthread9660   #发布
    消息
[D/main] Topic: /mqtt/test/rtthread9660 receive a message: Hello,RT-Thread!    #收到
    订阅消息
```

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。

提示：首次使用需要输入 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：

```
msh />wifi join test 12345678
join ssid:test
[I/WLAN.mgnt] wifi connect success ssid:test
.......
msh />[I/WLAN.lwip] Got IP address : 192.168.12.92
```

22.5 注意事项

- 使用本例程前需要先连接 WiFi。
- 如果串口终端显示 MQTT 连接错误（如下图），可能是 MQTT 服务器暂时被关闭了导致连接失败。

图 **22.1:** *1559533901730*

22.6 引用参考

- 《MQTT 软件包用户手册》：docs/UM1005-RT-Thread-Paho-MQTT 用户手册.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 23 章

HTTP Client 功能实现例程

23.1 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。

23.1.1 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。

23.1.2 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。

23.2 硬件说明

本例程需要依赖 IoTBoard 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

HTTP Client 例程位于 `/examples/22_iot_http_client` 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（WebClient 例程程序）
packages/webclient-v2.1.0	webclient 软件包
packages/webclient-v2.1.0/inc	webclient 软件包头文件
packages/webclient-v2.1.0/src	webclient 软件包源码文件

23.3.1 例程使用说明

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 `/examples/22_iot_http_client/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;

    /* 配置 wifi 工作模式 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);

    /* 初始化 wlan 自动连接功能 */
    wlan_autoconnect_init();

    /* 使能 wlan 自动连接功能 */
    rt_wlan_config_autoreconnect(RT_TRUE);

    /* 创建 'net_ready' 信号量 */
```

```
    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_data();
    /* HTTP POST 请求发送 */
    webclient_post_data();
}
```

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

1. GET 请求发送

本例程中使用 GET 方式请求 HTTP 服务器 www.rt-thread.com 中的 rt-thread.txt 文本文件，文件在服务器中的地址为 /server/rt-thread.txt，使用端口为默认 HTTP 端口 80。

例程中调用封装的 webclient_get_data() 函数发送 GET 请求，该函数内部直接使用 webclient_request() 完成整个 GET 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 GET 请求头部信息，响应的数据存储到 buffer 缓冲区并打印到 FinSH 控制台。

webclient_request() 函数一般用于响应数据较短的情况，因为该函数会将响应数据**一次性**全部读取并存储到缓冲区，更多 WebClient 软件包 GET 请求使用方式可查看 /examples/19_iot_http_client/applications/packages/webclient-v2.1.0/samples目录下 GET 请求例程文件。

```
#define HTTP_GET_URL          "http://www.rt-thread.com/service/rt-thread.txt"

int webclient_get_data(void)
{
    unsigned char *buffer = RT_NULL;
    int length = 0;

    /* 发送 GET 请求，使用默认头部信息，buffer 读取接收的数据 */
```

```
    length = webclient_request(HTTP_GET_URL, RT_NULL, RT_NULL, &buffer);
    if (length < 0)
    {
        LOG_E("webclient GET request response data error.");
        return -RT_ERROR;
    }

    LOG_D("webclient GET request response data :");
    LOG_D("%s", buffer);

    web_free(buffer);
    return RT_EOK;
}
```

2. POST 请求发送

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_data() 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 buffer 缓冲区并打印到 FinSH 控制台。更多 WebClient 软件包 POST 请求使用方式可查看 /examples/22_iot_http_client/applications/packages/webclient-v2.1.0/samples目录下 POST 请求例程文件。

```
#define HTTP_POST_URL         "http://www.rt-thread.com/service/echo"
/* POST 请求发送到服务器的数据 */
const char *post_data = "RT-Thread is an open source IoT operating system from China
    !";

int webclient_post_data(void)
{
    unsigned char *buffer = RT_NULL;
    int length = 0;

    /* 发送 POST 请求，使用默认头部信息，buffer 读取响应的数据 */
    length = webclient_request(HTTP_POST_URL, RT_NULL, post_data, &buffer);
    if (length < 0)
    {
        LOG_E("webclient POST request response data error.");
        return -RT_ERROR;
    }

    LOG_D("webclient POST request response data :");
    LOG_D("%s", buffer);

    web_free(buffer);
    return RT_EOK;
}
```

23.4 运行

23.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，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
 / | \     4.0.1 build Jun  2 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x0000
0000, size is 4096 bytes.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number
is 0xD6363A94.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is
OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
```

23.4.2 连接无线网络

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），如下所示：

```
msh />wifi join test 12345678
join ssid:test
[I/WLAN.mgnt] wifi connect success ssid:test
[I/WLAN.lwip] Got IP address : 192.168.12.83
```

23.4.3 发送 GET 和 POST 请求

WiFi 连接成功后，先运行 HTTP GET 请求，获取并打印 GET 请求接收的数据，接着运行 HTTP POST 请求，上传指定数据到服务器并获取服务器响应数据。如下所示：

```
[D/main] webclient GET request response data :              # GET 请求接收数据
```

```
[D/main] 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.

[D/main] webclient POST request response data :              # POST 请求响应数据
[D/main] RT-Thread is an open source IoT operating system from China!
```

23.5 注意事项

使用本例程前需要先连接 WiFi。

23.6 引用参考

- 《WebClient 软件包用户手册》：docs/UM1001-RT-Thread-WebClient 用户手册.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 24 章

使用 Web 服务器组件：WebNet

24.1 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。

24.2 硬件说明

本例程需要依赖 IoT Board 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。

24.3 准备工作

1. 在 TF 卡根目录下创建 webnet 文件夹。
2. 在 webnet 文件夹里创建 admin 和 upload 两个文件夹。
3. 将 `/examples/23_iot_web_server/ports/webnet/index.html` 复制到 TF 卡的 webnet 文件夹里。

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。

24.4 软件说明

WebNet 位于 `/examples/23_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
ports/webnet	示例网页
../../drivers/drv_spi_tfcard.c	TF 卡驱动

本例程主要展示了 WebNet 的几个常用功能，程序代码位于 /examples/23_iot_web_server/applications/main.c 文件中。

在 main 函数中，主要完成了以下几个任务：

- 挂载 SD 卡
- 初始化 WIFI，等待 WIFI 自动连接成功
- 启动 WebNet

main 函数代码如下所示：

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

    /* 挂载文件系统 */
    if (dfs_mount("sd0", "/", "elm", 0, 0) == 0)
    {
        LOG_I("Filesystem initialized!");
    }
    else
    {
        LOG_E("Failed to initialize filesystem!");
    }

    /* 配置 wifi 工作模式 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);

    /* 创建信号量 */
    rt_sem_init(&net_ready, "net_ready", 0, RT_IPC_FLAG_FIFO);

    /* 注册 WIFI 连接成功回调函数 */
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY, wlan_ready_handler, RT_NULL);

    /* 配置 WIFI 自动连接 */
    wlan_autoconnect_init();

    /* 使能 WIFI 自动连接 */
    rt_wlan_config_autoreconnect(RT_TRUE);

    /* 等待连接成功 */
    result = rt_sem_take(&net_ready, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        LOG_E("Wait net ready failed!");
        return -RT_ERROR;
    }

    /* 启动 webnet demo */
    webnet_demo();
```

```
    return 0;
}
```

当 WiFi 连接成功后，会调用 webnet_demo() 函数，该函数会开启 CGI，AUTH 验证和上传功能，然后启动 WebNet。webnet_demo() 函数代码如下所示：

```
void webnet_demo(void)
{
    /* 注册 CGI 处理函数 */
    webnet_cgi_register("hello", cgi_hello_handler);
    webnet_cgi_register("calc", cgi_calc_handler);

    /* 设置 AUTH 验证 */
    webnet_auth_set("/admin", "admin:admin");

    /* 添加上传入口 */
    webnet_upload_add(&upload_entry_upload);

    /* 启动 WebNet */
    webnet_init();
}
```

24.5 运行

24.5.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 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。

24.5.2 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：

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build Jun  6 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/main] Filesystem initialized!                          #文件系统挂载成功
[I/WLAN.dev] wlan init success
```

```
[I/WLAN.lwip] eth device init ok name:w0
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x0000
0000, size is 4096 bytes.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number
is 0xD6363A94.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is
OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
msh />[I/WLAN.mgnt] wifi connect success ssid:realthread
[I/wn] RT-Thread webnet package (V2.0.0) initialize success.
[I/WLAN.lwip] Got IP address : 192.168.12.29                #网络连接成功
```

24.5.3 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 /examples/23_iot_web_server/rtconfig.h 中的 WEBNET_ROOT 宏定义来修改，默认为 /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（/webnet）的 **index.html** 文件，如下图所示，页面文件正常显示：

图 24.1: *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 /webnet，想要访问在 TF 卡里的 /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。

24.5.4 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：

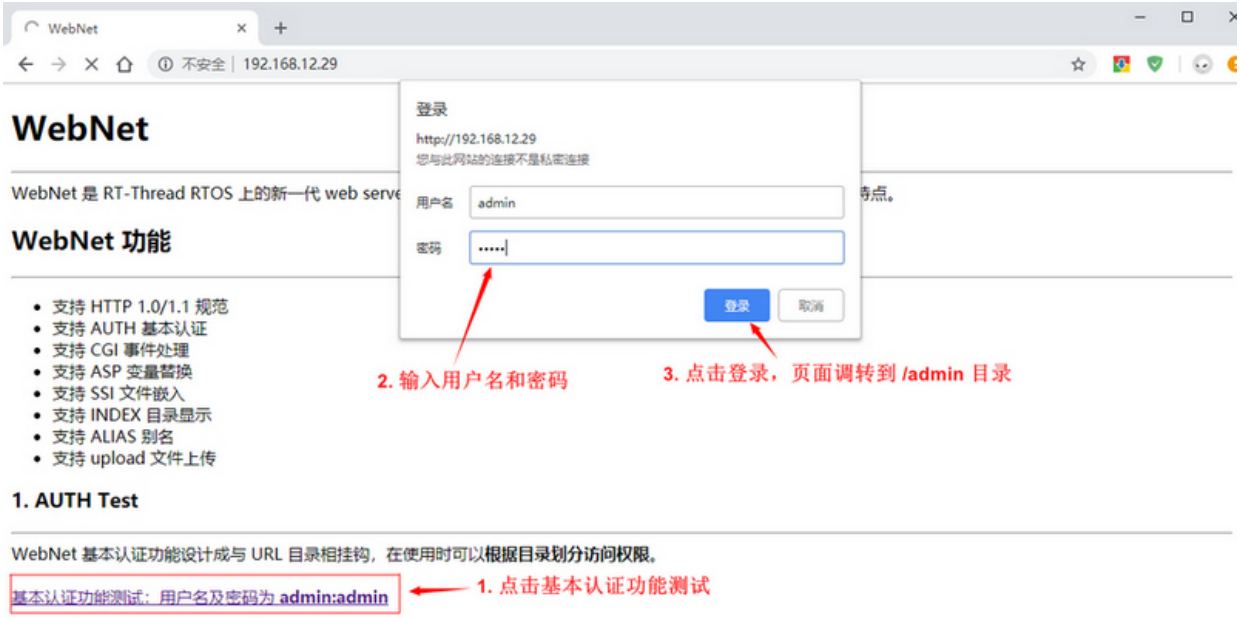


图 24.2: *auth*

24.5.5 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 24.3: *upload_file*

24.5.6 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 24.4: *index*

24.5.7 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 24.5: *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 24.6: *calc*

24.6 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。

24.7 引用参考

- 《文件系统应用笔记》：docs/AN0012-RT-Thread-文件系统应用笔记.pdf
- 《WLAN 框架应用笔记》：docs/AN0026-RT-Thread-WLAN 框架应用笔记.pdf
- 《WEBNET 用户手册》：docs/UM1010-RT-Thread-Web 服务器 (WebNet) 用户手册.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

24.8 使用 websocket 软件包通信

本例程使用 `librws` 软件包与 PC 模拟的 websocket 服务器进行数据交互，展示 `websocket` 软件包通信过程。

24.9 简介

WebSocket 是一种在单个 TCP 连接上进行全双工通信的协议，于 2011 年被 IETF 定为标准 RFC 6455，并由 RFC7936 补充规范。WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。

`librws` 软件包是小型、跨平台 `websocket` 客户端 C 语言库，具有以下特点：

- 无其他额外的依赖
- 单一头文件接口，位于 `librws.h` 中
- 线程安全
- 线程后台处理发送/接收逻辑

24.10 硬件说明

本例程需要依赖 WiFi 功能完成网络通信，因此请确保硬件平台上的 WiFi 功能可以正常工作，并且能够连接网络。

24.11 软件说明

24.11.1 主函数代码说明

在主函数中进行了如下操作：

1. 配置 wlan 的自动连接功能并开启自动连接。

2. 使用 websocket 与 PC 模拟的 websocket 服务器进行连接、数据发送、接收等。

```
int main(void)
{
    int i;
    static rws_socket socket;
    /* ！！！注意：要换成自己电脑的 IP 地址 */
    const char *host = "192.168.10.168";
    const char *send_text = "Hello RT-Thread!";

    /* 初始化分区表 */
    fal_init();
    /* 初始化 easyflash */
    easyflash_init();
    /* 配置 wifi 工作模式 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    /* 初始化自动连接配置 */
    wlan_autoconnect_init();
    /* 使能 wlan 自动连接 */
    rt_wlan_config_autoreconnect(RT_TRUE);

    /* 查询 wlan 是否处于连接状态 */
    while (rt_wlan_is_ready() != RT_TRUE)
    {
        rt_thread_delay(1000);
    }
    rt_sem_init(&rws_sem_rec, "rws_rec", 0, RT_IPC_FLAG_FIFO);
    /* 连接 echo.websocket.org */
    socket = rws_connect(host, 9999);
    if (RT_NULL == socket)
    {
        LOG_E("Can not connect %s",host);
        return 0;
    }

    for (i = 0; i < 10; i++)
    {
        /* 发送消息 */
        rws_send(socket, send_text);
        /* 等待接收完成 */
        rt_sem_take(&rws_sem_rec, RWS_RECEIVE_TIMEOUT);
        /* 比较发送消息与接收消息是否一致 */
        if (strcmp(text_rec, send_text) != 0)
        {
            LOG_E("Receive data: %s is different from send data: %s", text_rec,
                send_text);
        }
    }
```

```
    if (text_rec != RT_NULL)
    {
        rt_free(text_rec);
    }
    /* 断开连接 */
    rws_disconnect(socket);
}
```

24.11.2 websocket 连接说明

websocket 连接函数，需要传入 url 与端口号，本例程利用 RT-Thread_W60X_SDK/tools/websocket 的 websocket.exe 工具使电脑作为 websocket 服务端，默认端口号是 9999。该函数主要实现 socket 的创建与设置，以及连接、断开、接收数据等函数的实现，具体实现如下：

```
static rws_socket rws_connect(const char *host, const int port)
{
    static rws_socket socket;

    /* 创建 socket */
    socket = rws_socket_create();
    if (socket == RT_NULL)
    {
        LOG_E("Librws socket create failed. ");
        return RT_NULL;
    }
    /* 设置 socket url */
    rws_socket_set_url(socket, "ws", host, port, "/");
    /* 设置 socket 连接回调函数 */
    rws_socket_set_on_connected(socket, &rws_open);
    /* 设置socket 断开连接回调函数 */
    rws_socket_set_on_disconnected(socket, &rws_close);
    /* 设置接收消息回调函数 */
    rws_socket_set_on_received_text(socket, &message_text_rec);
    /* 设置自定义模式 */
    rws_socket_set_custom_mode(socket);

    if (rws_socket_connect(socket) == RT_FALSE)
    {
        LOG_E("Connect ws://%s:%d/ failed. ", host, port);
        return RT_NULL;
    }
    LOG_D("Connect ws://%s:%d/ success.", host, port);

    return socket;
}
```

24.11.3 websocket 断开连接说明

websocket 断开连接函数回收资源，断开 socket 连接，具体实现如下:

```
static void rws_disconnect(rws_socket socket)
{
    rws_socket_disconnect_and_release(socket);
    LOG_D("Try disconnect websocket connection.");
}
```

24.11.4 websocket 发送数据说明

websocket 发送数据函数主要是打印发送的数据、对数据进行发送，详细实现如下：

```
static int rws_send(rws_socket socket, const char *text)
{
    LOG_D("send message: %s, len: %d", text, rt_strlen(text));
    rws_socket_send_text(socket, text);

    return RT_EOK;
}
```

24.12 运行

24.12.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。
- **IAR**：双击 `project.eww` 打开 IAR 工程，执行编译。

编译完成后，将开发板与 PC 机连接，然后将固件下载至开发板。

24.12.2 运行效果

在电路板上电前，双击打开 `RT-Thread_W60X_SDK/tools/websocket/websocket.exe`

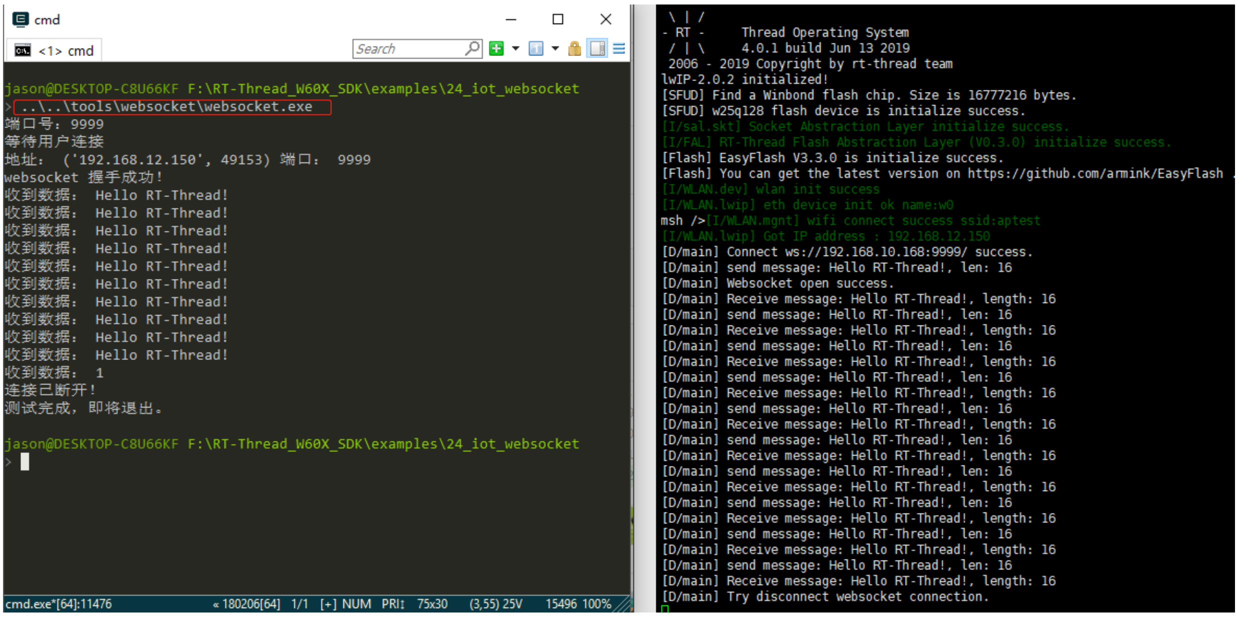


图 24.7: *websocket 运行效果图*

开发板串口日志如下所示:

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build Jun 13 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.12.150
[D/main] Connect ws://192.168.10.168:9999/ success.
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Websocket open success.
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
```

```
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] Try disconnect websocket connection.
```

24.12.3 连接无线网络

例程使能了 wlan 自动连接，之前有连接就会自动连接 wlan，运行效果日志之后，紧接着 wifi 连接成功日志，如下：

```
msh />[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.12.26
```

如果没有连接或者网络相关信息被擦除，则需要在程序运行后会进入 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：

```
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
```

24.12.4 websocket

数据完成交互日志如下：

```
[D/main] Connect ws://echo.websocket.org:80/ success.  # 成功连接服务器
[D/main] send message: Hello RT-Thread!, len: 16       # 发送数据
[D/main] Websocket open success.                       # 打开 websocket
[D/main] Receive message: Hello RT-Thread!, length: 16 # 接收数据
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
```

```
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] send message: Hello RT-Thread!, len: 16
[D/main] Receive message: Hello RT-Thread!, length: 16
[D/main] Try disconnect websocket connection.          # 断开连接
```

24.13 注意事项

- `main` 函数中的 host 要换成自己电脑的 IP 地址，如果 websocket 代码没有运行，检查 wifi 是否连接。

24.14 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 25 章

JSON 数据构建与解析

本例程介绍如何使用 cJSON 软件包，详细介绍 JSON 数据的构建及针对所构建的 JSON 进行解析。

25.1 简介

json 是一种完全独立于编程语言的文本格式，用来存储和表示数据，具有以下特点：

- 简洁和层次结构清晰
- 易于人阅读和编写
- 易于机器解析和生成，并有效地提升网络传输效率

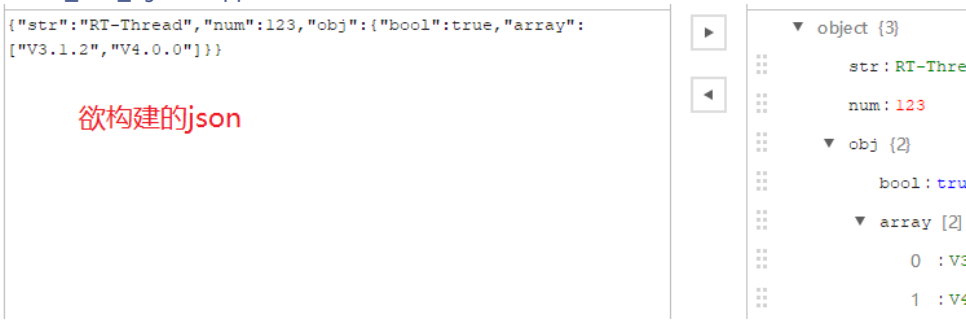
RT-Thread cJSON 软件包是一种超轻量级的 C 语言 JSON 解析库，具体 JSON 语法解析参考链接 http://www.json.org/json-zh.html，可以根据不同的语言习惯进行语言的选择。

25.2 硬件说明

json 数据构建与解析例程使用串口输出功能，无其他依赖。

25.3 软件说明

json 数据构建与解析位于 /examples/25_iot_cjson/applications 目录下，主要程序在 main 函数中，主要实现 json 的构建与解析，如下图所示



图中构建的 json 在构建函数中完成，解析的内容在解析函数中完成，具体如下。

构建 json 函数如下

```
/* 构建 json */
static char *make_json()
{
    char *p;
    cJSON *json_root = cJSON_CreateObject(); /* 创建根索引 json 对象 */
    cJSON *json_sub = cJSON_CreateObject();  /* 创建子索引 json 对象 */
    cJSON *json_array = cJSON_CreateArray(); /* 创建数组 json 对象 */

    /* 判断创建 cjson 是否创建成功 */
    if (RT_NULL == json_root || RT_NULL == json_sub || RT_NULL == json_array)
    {
        LOG_E("Fail to creat cjson");
        rt_free(json_root);
        rt_free(json_sub);
        rt_free(json_array);
        return RT_NULL;
    }
    /* 添加键值对 */
    cJSON_AddStringToObject(json_root, "str", "RT-Thread"); /* 添加字符串型键值对 */
    cJSON_AddNumberToObject(json_root, "num", 123); /* 添加整型键值对 */

    /* 添加子 json 对象到子 json 对象中 */
    cJSON_AddItemToObject(json_root, "obj", json_sub);
    /* 添加 bool 类型给子 json 对象 */
    cJSON_AddTrueToObject(json_sub, "bool"); /* 添加 bool 型键值对 */

    /* 添加数组 json 对象到根 json 对象中 */
    cJSON_AddItemToObject(json_sub, "array", json_array);
    cJSON_AddStringToObject(json_array, 0, "V3.1.2"); /* 添加数组成员 */
    cJSON_AddStringToObject(json_array, 0, "V4.0.0");

    /* 生成无格式 json 字符串 */
    p = cJSON_PrintUnformatted(json_root);

    cJSON_Delete(json_root);
    return p;

}
```

解析 json 函数如下

```
/* 解析构建的json */
static void parse_json(char *p)
{
    cJSON *root, *subordinate;
    RT_ASSERT(p);

    root = cJSON_Parse(p);
```

```
    if (RT_NULL == root)
    {
        LOG_E("Fail to parse root json");
        return;
    }

    /* 解析 json 中 CJSON_STR 键对应的字符串 */
    parse_string_kv(root, "str");
    /* 解析 json 中 "num" 键对应的数字 */
    parse_num_kv(root, "num");

    /* 解析子 json */
    subordinate = parse_key(root, "obj");
    /* 解析子 json 中 “bool” 键对应的 bool 值*/
    parse_bool_kv(subordinate, "bool");

    /* 解析 json 数组 */
    parse_array(subordinate, "array");

    cJSON_Delete(root);

}
```

```
int main(void)
{
   /* 构建 json */
   char *p = make_json();

    if (RT_NULL == p)
    {
        LOG_E("Fail to make cjson");
        return 0;
    }
    LOG_D("make json ->");
    LOG_D("%s", p);

    /* 解析组成的json */
    LOG_D("parse json ->");
    parse_json(p);

    rt_free(p);
    return 0;
}
```

25.4 运行

25.4.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译例程代码，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
 / | \     4.0.1 build May 14 2019
 2006 - 2019 Copyright by rt-thread team
[D/main] make json ->
[D/main] {"str":"RT-Thread","num":123,"obj":{"bool":true,"array":["V3.1.2","V4.0.0"
    ]}}
[D/main] parse json ->
[D/main] str : RT-Thread
[D/main] num : 123
[D/main] obj :
[D/main]     bool : 1
[D/main]     array:
[D/main]             V3.1.2
[D/main]             V4.0.0
msh >
```

25.5 注意事项

- 构建 json 过长，可能导致使用该 json 生成的字符串无法完全输出

25.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 26 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。

26.1 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。

26.2 硬件说明

mbedTLS 例程需要依赖 IoTBoard 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

fal 例程位于 `/examples/26_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
applications/tls_test.c	mbedTLS 例程程序
ports	移植文件
packages/mbedtls	mbedTLS 软件包（tls 源码实现）
packages/mbedtls/certs	TLS 证书存放目录

26.3.1 例程使用说明

mbedTLS 例程代码位于 /examples/26_iot_mbedtls/application/ 文件夹中，其中 main.c 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;

    /* 初始化 wlan 自动连接功能 */
    wlan_autoconnect_init();

    /* 使能 wlan 自动连接功能 */
    rt_wlan_config_autoreconnect(RT_TRUE);

    /* 创建 'net_ready' 信号量 */
    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)
    {
        rt_sem_detach(&net_ready);
        LOG_E("Wait net ready failed!");
        return -RT_ERROR;
    }

    /* 网络连接成功，启动 mbedTLS 客户端 */
    mbedtls_client_start();

    return 0;
}
```

设备成功接入网络后，会自动执行 mbedtls_client_start() 以启动 mbedTLS 客户端程序。

mbedTLS 客户端程序位于 /examples/26_iot_mbedtls/application/tls_test.c 中，核心代码说明如下：

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 mbedTLS 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。

26.4 运行

26.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。
- **IAR**：双击 `project.eww` 打开 IAR 工程，执行编译。

编译例程代码，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
 / | \     4.0.1 build Jun 11 2019
```

```
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />[I/WLAN.mgnt] wifi connect success ssid:realthread
[I/WLAN.lwip] Got IP address : 192.168.1.2
```

如果用户在此例程前已经为设备配置过网络，并且开启了自动连接网络功能，设备上电后，会自动连接网络，网络连接成功后，直接执行 mbedTLS 例程。

如果用户没有为设备配置过网络，请继续阅读下面章节，为设备配置网络。

26.4.2 连接无线网络

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），网络连接成功后会自动执行 mbedTLS 例程，如下所示：

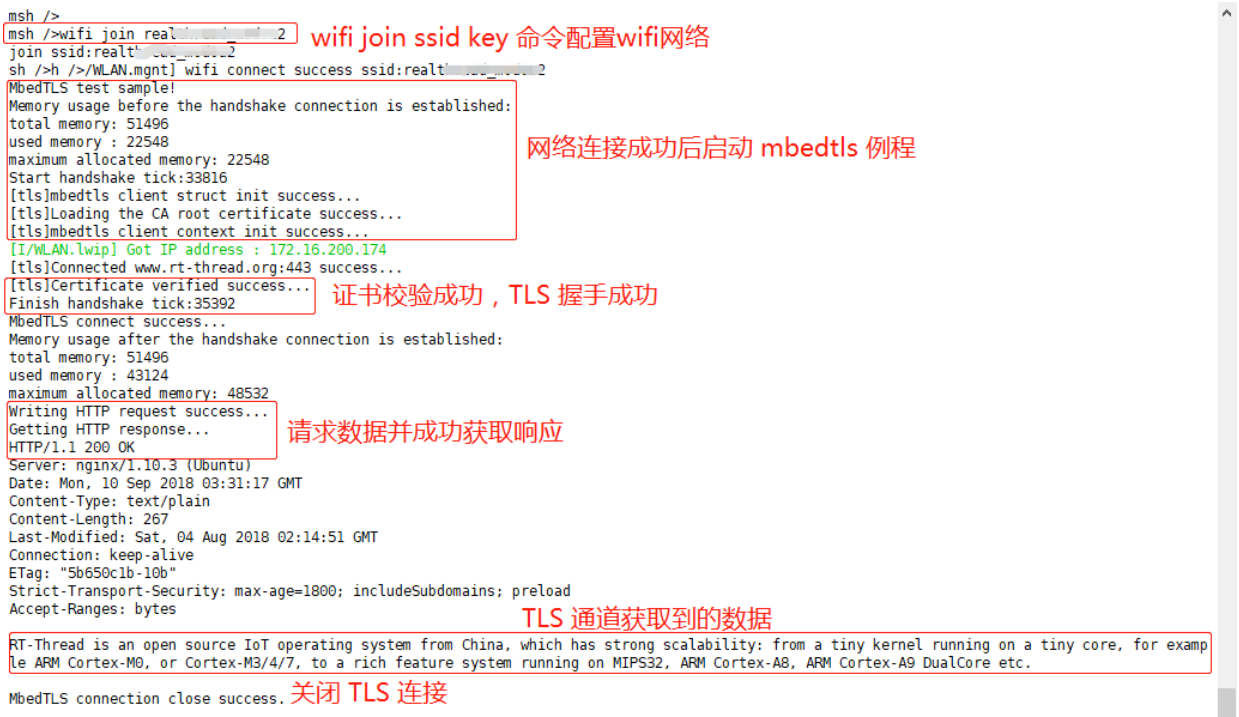


图 26.1: *连接无线网络*

26.5 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 `/examples/26_iot_mbedtls/mbedtls-v2.6.0/mbedtls/include/mbedtls/config.h`。

  如果用户不使用 scons 重新生成 IAR 或 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-v2.6.0/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。

26.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《mbedTLS 用户手册》：docs/UM1006-RT-Thread-MbedTLS 用户手册.pdf
- 已支持 TLS 加密连接的软件包

  RT-Thread 提供的与网络通讯相关的软件包大多都已经支持 TLS 加密连接，如 HTTP 客户端、MQTT 客户端 paho-mqtt、阿里云 iotkit、微软云 Azure等，如有需要请访问 RT-Thread 软件包主页获取。

第 27 章

硬件加解密例程

本例程将演示硬件 AES-CBC 加密及解密功能，以及使用硬件 MD5 和 SHA1 散列算法生成信息摘要。

27.1 简介

随机物联网大力发展，越来越多的设备接入网络，连接上云端。在使用网络通过的过程中，重要的原始明文被不法分子窃取到之后，将会产生不可估量的后果。此时信息安全显得格外重要。信息的安全交互往往伴随着各种加密解密。本例程将介绍如何使用常见的加解密算法及散列算法。

27.2 硬件说明

本硬件加解密例程需要使用串口输出功能，无其他依赖。

27.3 软件说明

硬件加解密的源代码位于 `/examples/27_iot_hw_crypto/applications/main.c` 中。

在 main 函数中，先使用 AES-CBC 将数据进行加密，然后对解密后的数据进行解密。加密完成后，使用 MD5 和 SHA1 两种散列算法生成信息摘要。同时输出一些日志信息。

```
int main(void)
{
    rt_uint8_t buf_in[32];
    rt_uint8_t buf_out[32];
    int i;

    /* 填充测试数据 */
    for (i = 0; i < sizeof(buf_in); i++)
    {
        buf_in[i] = (rt_uint8_t)i;
    }
    /* 打印填充的数据 */
```

```
    LOG_HEX("Data   ", 8, buf_in, sizeof(buf_in));

    memset(buf_out, 0, sizeof(buf_out));
    /* 对测试数据进行加密 */
    hw_aes_cbc(buf_in, buf_out, HWCRYPTO_MODE_ENCRYPT);

    /* 打印加密后的数据 */
    LOG_HEX("AES-enc", 8, buf_out, sizeof(buf_out));

    memset(buf_in, 0, sizeof(buf_in));
    /* 对加密数据进行解密 */
    hw_aes_cbc(buf_out, buf_in, HWCRYPTO_MODE_DECRYPT);

    /* 打印加密后的数据 */
    LOG_HEX("AES-dec", 8, buf_in, sizeof(buf_in));

    memset(buf_out, 0, sizeof(buf_out));
    /* 对测试数据进行 MD5 运算 */
    hw_hash(buf_in, buf_out, HWCRYPTO_TYPE_MD5);

    /* 打印 16 字节长度的 MD5 结果 */
    LOG_HEX("MD5    ", 8, buf_out, 16);

    memset(buf_out, 0, sizeof(buf_out));
    /* 对测试数据进行 SHA1 运算 */
    hw_hash(buf_in, buf_out, HWCRYPTO_TYPE_SHA1);

    /* 打印 20 字节长度的 SHA1 结果 */
    LOG_HEX("SHA1   ", 8, buf_out, 20);

    return 0;
}
```

硬件加解密具体实现代码在 main() 函数下方。硬件加解密使用流程大致分为 4 个步骤。第一步创建具体加解密类型的上下文。第二步对上下文进行配置，如设置密钥等操作。第三步执行相应的功能，获得处理后的结果。第四步删除上下文，释放资源。

1. AES-CBC 加解密

```
static void hw_aes_cbc(const rt_uint8_t in[32], rt_uint8_t out[32], hwcrypto_mode
    mode)
{
    struct rt_hwcrypto_ctx *ctx;

    /* 创建一个 AES-CBC 模式的上下文 */
    ctx = rt_hwcrypto_symmetric_create(rt_hwcrypto_dev_dufault(),
        HWCRYPTO_TYPE_AES_CBC);
    if (ctx == RT_NULL)
    {
        LOG_E("create AES-CBC context err!");
```

```
        return;
    }
    /* 设置 AES-CBC 加密密钥 */
    rt_hwcrypto_symmetric_setkey(ctx, key, 128);
    /* 执行 AES-CBC 加密/解密 */
    rt_hwcrypto_symmetric_crypt(ctx, mode, 32, in, out);
    /* 删除上下文，释放资源 */
    rt_hwcrypto_symmetric_destroy(ctx);
}
```

2. HASH 信息摘要

```
static void hw_hash(const rt_uint8_t in[32], rt_uint8_t out[32], hwcrypto_type type)
{
    struct rt_hwcrypto_ctx *ctx;

    /* 创建一个 SHA1/MD5 类型的上下文 */
    ctx = rt_hwcrypto_hash_create(rt_hwcrypto_dev_dufault(), type);
    if (ctx == RT_NULL)
    {
        LOG_E("create hash[%08x] context err!", type);
        return;
    }
    /* 将输入数据进行 hash 运算 */
    rt_hwcrypto_hash_update(ctx, in, 32);
    /* 获得运算结果 */
    rt_hwcrypto_hash_finish(ctx, out, 32);
    /* 删除上下文，释放资源 */
    rt_hwcrypto_hash_destroy(ctx);
}
```

27.4 运行

27.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

27.4.2 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
 / | \     4.0.1 build Jun  3 2019
```

```
 2006 - 2019 Copyright by rt-thread team
D/HEX Data   : 0000-0008: 00 01 02 03 04 05 06 07    ........
               0008-0010: 08 09 0A 0B 0C 0D 0E 0F    ........
               0010-0018: 10 11 12 13 14 15 16 17    ........
               0018-0020: 18 19 1A 1B 1C 1D 1E 1F    ........
D/HEX AES-enc: 0000-0008: 0A 94 0B B5 41 6E F0 45    ....An.E
               0008-0010: F1 C3 94 58 C6 53 EA 5A    ...X.S.Z
               0010-0018: 3C F4 56 B4 CA 48 8A A3    <.V..H..
               0018-0020: 83 C7 9C 98 B3 47 97 CB    .....G..
D/HEX AES-dec: 0000-0008: 00 01 02 03 04 05 06 07    ........
               0008-0010: 08 09 0A 0B 0C 0D 0E 0F    ........
               0010-0018: 10 11 12 13 14 15 16 17    ........
               0018-0020: 18 19 1A 1B 1C 1D 1E 1F    ........
D/HEX MD5    : 0000-0008: B4 FF CB 23 73 7C EC 31    ...#s|.1
               0008-0010: 5A 4A 4D 1A A2 A6 20 CE    ZJM... .
D/HEX SHA1   : 0000-0008: AE 5B D8 EF EA 53 22 C4    .[...S".
               0008-0010: D9 98 6D 06 68 0A 78 13    ..m.h.x.
               0010-0018: 92 F9 A6 42                ...B
```

27.5 注意事项

- AES 加解密是以块为单位进行计算的。一个块大小为 16 字节。加解密输入的数据大小需要是 16 字节的整数倍。
- 本例程为硬件加解密，需要运行在支持硬件 AES-CBC 、MD5 、SHA1 的芯片上，并且完成了驱动的对接，注册了硬件加解密设备驱动。

27.6 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 28 章

Ymodem 协议固件升级例程

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。

28.1 背景知识

28.1.1 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。

28.1.2 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。

28.1.3 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。

28.1.4 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。

28.1.5 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：

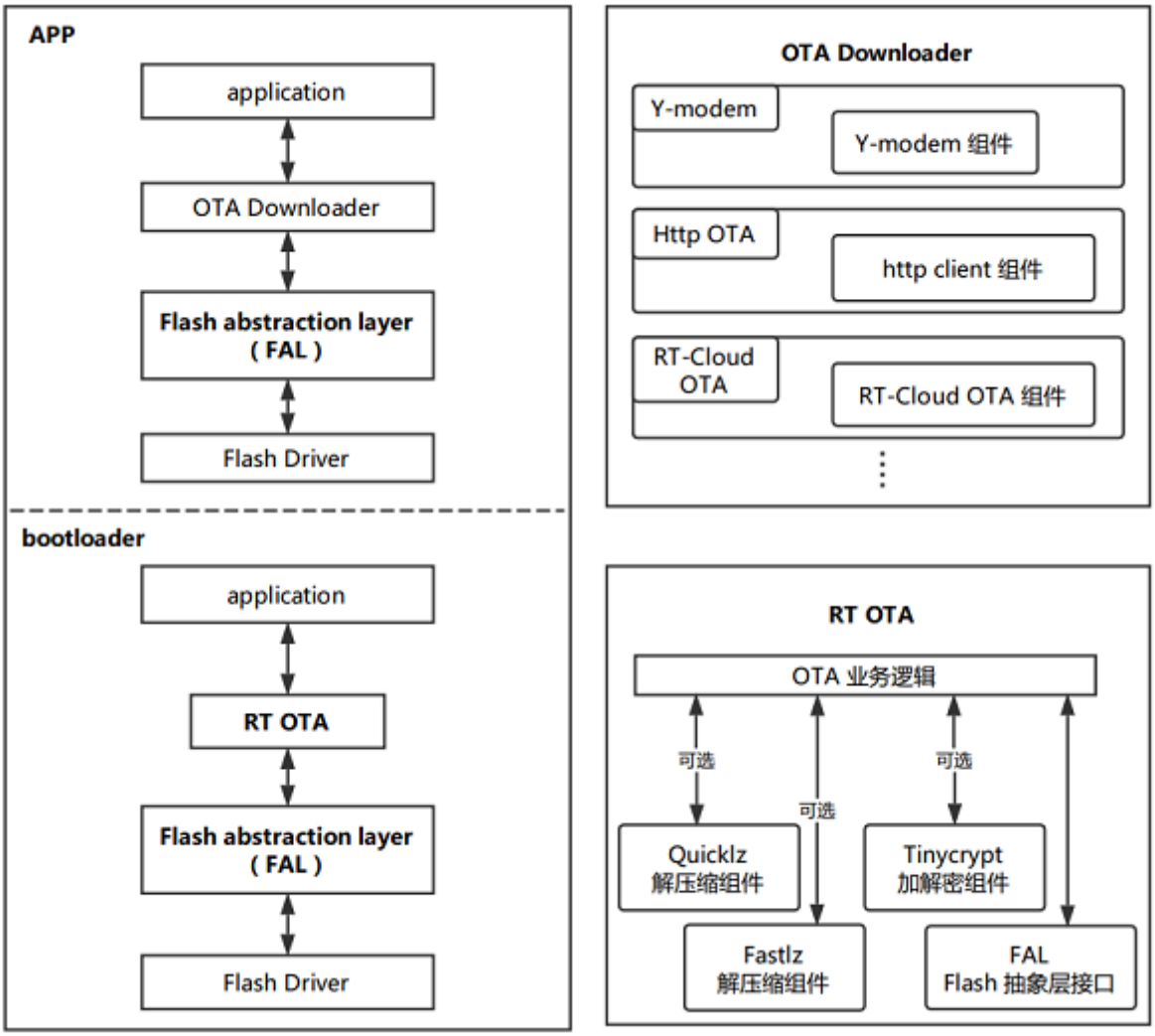


图 28.1: *OTA 框架图*

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。

28.1.6 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：

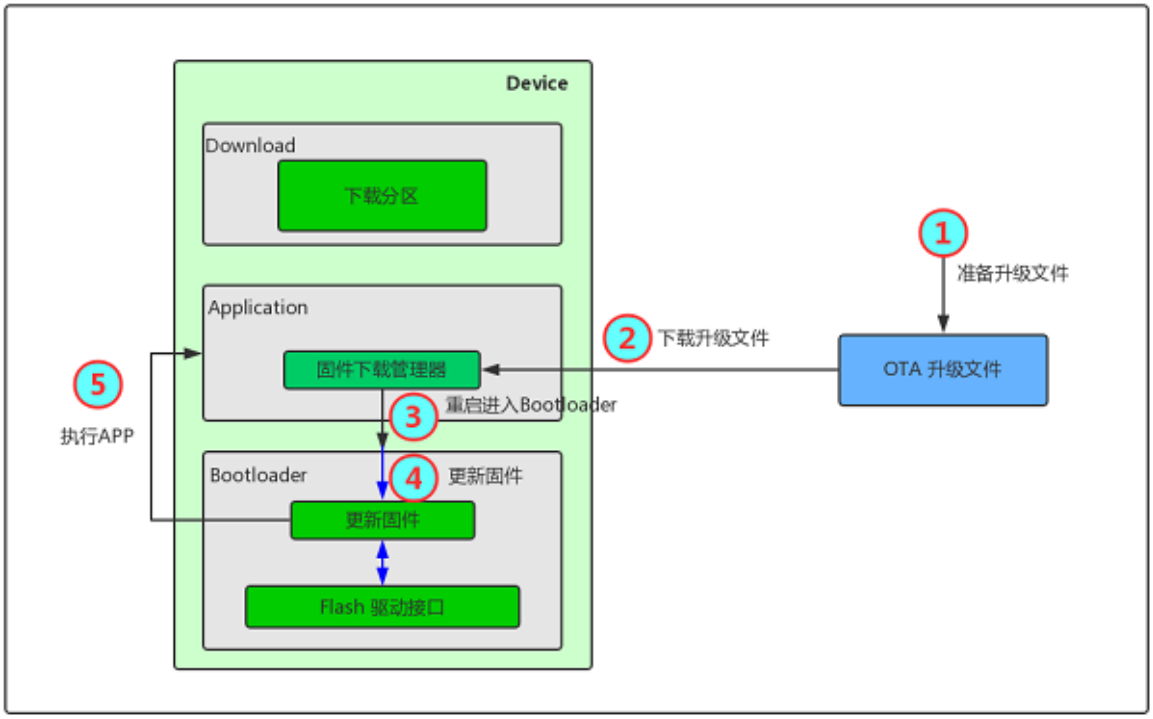


图 28.2: *OTA 升级流程*

28.2 硬件说明

本例程使用到的硬件资源如下所示：

- UART0(Tx: PA4; Rx: PA5)
- 片内 FLASH (1MBytes)
- 片外 Nor Flash (16MBytes)

28.3 分区表

分区名称	存储位置	起始地址	分区大小	结束地址	说明
app	片内 FLASH	0x08010100	950K	0x080fd900	app 应用程序存储区
easyflash	Nor FLASH	0x08000000	1M	0x08100000	easyflash 存储区
download	Nor FLASH	0x08100000	1M	0x08200000	download 下载存储区
font	Nor FLASH	0x08200000	7M	0x08900000	font 字库分区
filesystem	Nor FLASH	0x08900000	7M	0x09000000	filesystem 文件系统区

分区表定义在 **bootloader 程序** 中，如果需要修改分区表，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。

28.4 软件说明

Ymodem 例程位于 /examples/28_iot_ota_ymodem 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/ymodem_update.c	ymodem 应用，实现了 OTA 固件下载业务
ports/fal	Flash 抽象层软件包（fal）的移植文件
packages/fal	fal 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

28.4.1 Ymodem 代码说明

Ymodem 升级固件下载程序代码在 /examples/28_iot_ota_ymodem/applications/ymodem_update.c 文件中，仅有三个 API 接口，介绍如下：

update 函数

```
void update(uint8_t argc, char **argv);
MSH_CMD_EXPORT_ALIAS(update, ymodem_start, Update user application firmware);
```

update 函数调用底层接口 rym_recv_on_device 启动 Ymodem 升级程序，并使用 RT-Thread MSH_CMD_EXPORT_ALIAS API 函数将其导出为 **ymodem_start** MSH 命令。

固件下载完成后，通过底层接口 rym_recv_on_device 获取下载状态，下载成功则重启系统，进行 OTA 升级。

ymodem_on_begin 函数

```
static enum rym_code ymodem_on_begin(struct rym_ctx *ctx, rt_uint8_t *buf, rt_size_t
    len);
```

这是一个回调函数，通过底层接口 rym_recv_on_device 注册，在 Ymodem 程序启动后，获取到通过 Ymodem 协议发送给设备的文件后执行。主要是获取到文件大小信息，为文件存储做准备，完成相应的初始化工作（如 FAL download 分区擦除，为固件写入做准备）。

ymodem_on_data 函数

```
static enum rym_code ymodem_on_data(struct rym_ctx *ctx, rt_uint8_t *buf, rt_size_t
    len);
```

这是一个数据处理回调函数，通过底层接口 rym_recv_on_device 注册，在接收到通过 Ymodem 协议发送给设备的数据后，执行该回调函数处理数据（这里将接收到的数据写入到 download 分区）。

28.5 运行

本例程演示使用 Ymodem OTA 功能烧录 v2.0.0 版本的 app 程序。

28.5.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

28.5.2 运行效果

按下复位按键重启开发板，正常运行后，可以看到当前版本为 v1.0.0。

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build Jun  5 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0
    x00000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev    |   offset    |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | app        | w60x_onchip | 0x00000000 | 0x000efc00 |
[I/FAL] | easyflash  | norflash    | 0x00000000 | 0x00100000 |
[I/FAL] | download   | norflash    | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash    | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash    | 0x00900000 | 0x00700000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[D/main] The current version of APP firmware is 1.0.0
```

28.5.3 固件升级

修改例程中的 APP_VERSION 为 2.0.0，双击 project.uvprojx 打开 MDK5 工程，执行编译。编译完成后会在本例程的目录下生成 Bin 文件夹，该文件夹中已经生成了所需的升级固件 rtthread.rbl。

使用命令 ymodem_start 启动 Ymodem 升级。

- 打开支持 **Ymodem 协议**的串口终端工具（推荐使用 **Xshell**）
- 连接开发板串口，复位开发板，进入 MSH 命令行
- 在设备的命令行里输入 ymodem_start 命令启动 Ymodem 升级
- 选择升级使用 **Ymodem 协议**发送升级固件（选择刚才编译后生成的 **rt-thread.rbl** 固件）

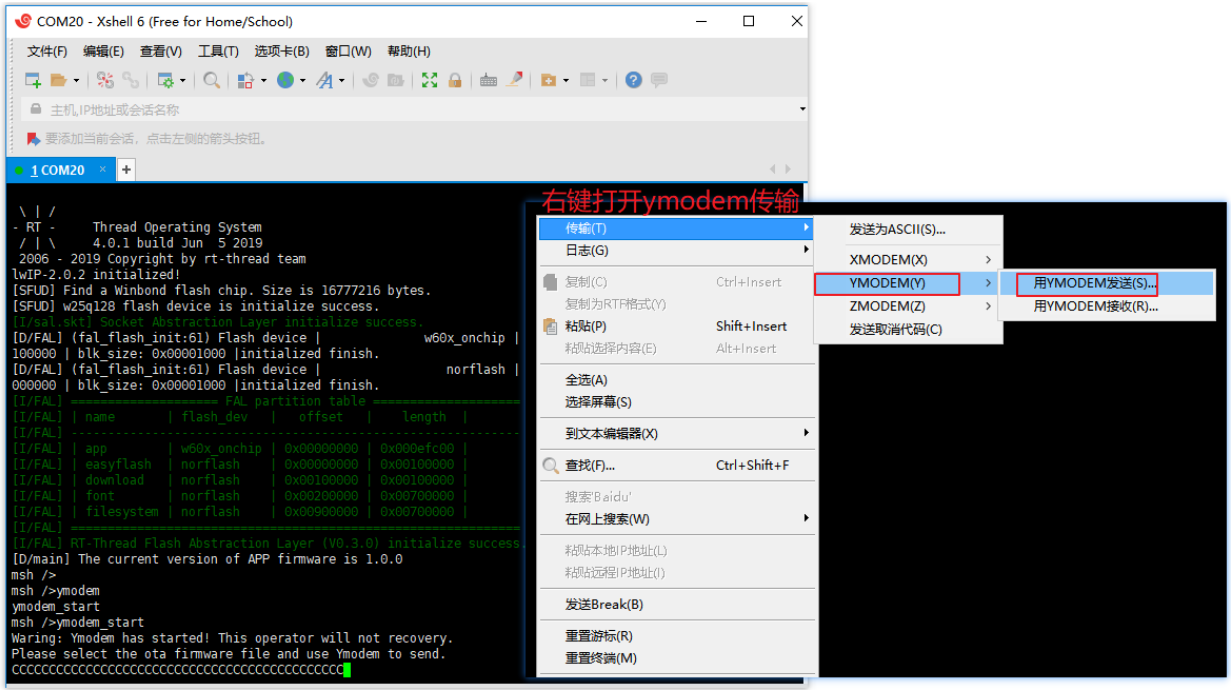


图 28.3: *选择 Ymodem 协议发送固件*

- 设备升级过程

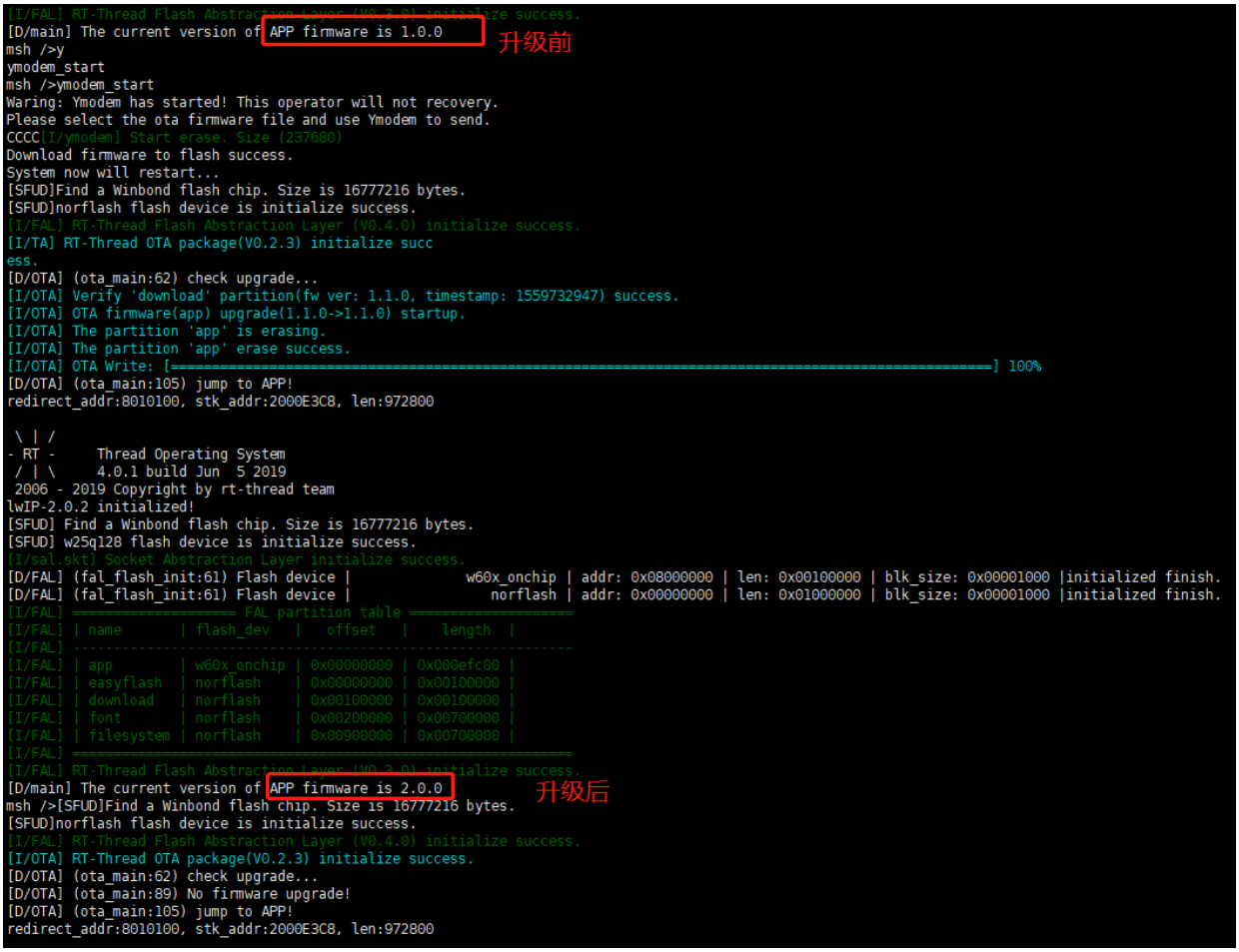


图 28.4: *升级固件*

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 Ymodem 固件下载功能的，因此可以一直使用 Ymodem 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。

28.6 注意事项

- 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

28.7 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《RT-Thread OTA 用户手册》：docs/UM1004-RT-Thread-OTA 用户手册.pdf

第 29 章

HTTP 协议固件升级例程

29.1 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** 。

29.2 背景知识

参考 Ymodem 固件升级例程。

29.3 硬件说明

本例程使用到的硬件资源如下所示：

- UART0(Tx: PA4; Rx: PA5)
- 片内 FLASH (1MBytes)
- 片外 Nor Flash (16MBytes)

29.4 分区表

分区名称	存储位置	起始地址	分区大小	结束地址	说明
app	片内 FLASH	0x08010100	950K	0x080fd900	app 应用程序存储区
easyflash	Nor FLASH	0x08000000	1M	0x08100000	easyflash 存储区

分区名称	存储位置	起始地址	分区大小	结束地址	说明
download	Nor FLASH	0x08100000	1M	0x08200000	download 下载存储区
font	Nor FLASH	0x08200000	7M	0x08900000	font 字库分区
filesystem	Nor FLASH	0x08900000	7M	0x09000000	filesystem 文件系统区

分区表定义在 `bootloader 程序` 中，如果需要修改分区表，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。

29.5 软件说明

http 例程位于 `/examples/29_iot_ota_http` 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
applications/ota_http.c	http ota 应用，基于 HTTP 协议实现 OTA 固件下载业务
ports/fal	Flash 抽象层软件包（fal）的移植文件
packages/fal	fal 软件包
packages/webclient	webclient 软件包，实现 HTTP 客户端

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

29.5.1 程序说明

HTTP OTA 固件下载器程序代码在 `/examples/29_iot_ota_http/applications/ota_http.c` 文件中，仅有三个 API 接口，介绍如下：

print_progress 函数

```
static void print_progress(size_t cur_size, size_t total_size);
```

该函数用于打印文件的下载进度。

http_ota_fw_download 函数

```
static int http_ota_fw_download(const char* uri);
```

http_ota_fw_download 函数基于 webclient API 实现了从指定的 **uri** 下载文件的功能，并将下载的文件存储到 **download** 分区。

uri 格式示例：`http://192.168.1.10:80/rt-thread.rbl`。非 80 端口需要用户指定。如果使用了 TLS 加密连接，请使用 `https://192.168.1.10:80/rt-thread.rbl`。

http_ota 函数

```
void http_ota(uint8_t argc, char **argv);
MSH_CMD_EXPORT(http_ota, OTA by http client: http_ota [url]);
```

HTTP OTA 入口函数，使用 `MSH_CMD_EXPORT` 函数将其导出为 `http_ota` 命令。

`http_ota` 命令需要传入固件下载地址，示例：`http_ota http://192.168.1.10:80/rt-thread.rbl`。

29.6 运行

本例程演示使用 http OTA 功能烧录 v2.0.0 版本的 app 程序。

29.6.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

29.6.2 运行效果

按下复位按键重启开发板，正常运行后，可以看到当前版本为 v1.0.0。

```
 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build Jun  6 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0
    x00000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offset
    @0x0000f0c8.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev   |   offset   |    length   |
[I/FAL] -------------------------------------------------------------
```

```
[I/FAL] | easyflash  | norflash     | 0x00000000 | 0x00100000 |
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash     | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash     | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash     | 0x00900000 | 0x00700000 |
[I/FAL] ============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x00000000
    , size is 4096 bytes.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number is 0
    xD6363A94.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[D/main] The current version of APP firmware is 1.0.0
msh />[I/WLAN.mgnt] wifi connect success ssid:realthread
[I/WLAN.lwip] Got IP address : 192.168.12.92
```

29.6.3 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. 双击 project.uvprojx 打开 MDK5 工程，修改本例程 mian.c 中的 APP_VERSION 为 2.0.0 ，执行编译。编译完成后，在本示例的目录下会生成 Bin 文件夹，其中含有 OTA 升级固件 rtthread.rbl 。
3. 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：

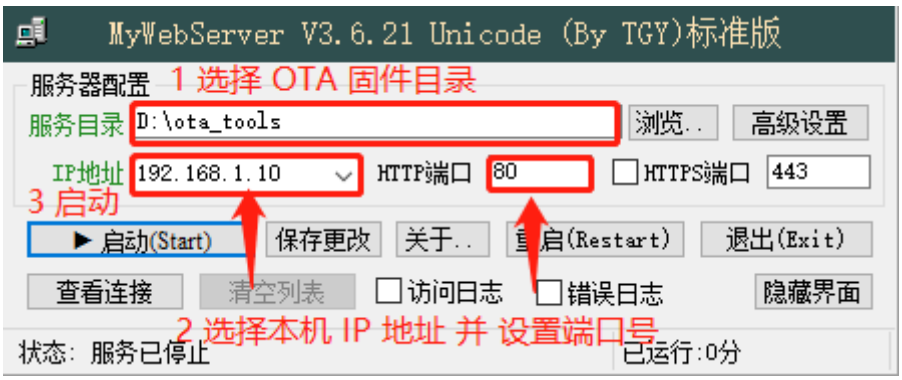


图 29.1: *启动 MyWebServer 软件*

4. 连接开发板串口，复位开发板，进入 MSH 命令行
5. 在设备的命令行里输入 **http_ota http://192.168.1.10:80/rt-thread.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。

6. 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。

```
msh />http_ota http://192.168.1.10:80/rt-thread.rbl
[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...
PPPPPPPPPPPPPPPCCCCCCCCCCCCCCCCCCCCCCCCCC
```

HTTP OTA 下载固件完成后，手动复位硬件。

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：

```
[SFUD]Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD]norflash flash device is initialize success.
[I/FAL] RT-Thread Flash Abstraction Layer (V0.4.0) initialize success.
[I/OTA] RT-Thread OTA package(V0.2.3) initialize success.
[D/OTA] (ota_main:62) check upgrade...
[I/OTA] Verify 'download' partition(fw ver: 1.1.0, timestamp: 1559803627) success.
[I/OTA] OTA firmware(app) upgrade(1.1.0->1.1.0) startup.
[I/OTA] The partition 'app' is erasing.
[I/OTA] The partition 'app' erase success.
[I/OTA] OTA Write: [============================================================] 100%
[D/OTA] (ota_main:105) jump to APP!
redirect_addr:8010100, stk_addr:2000ECD8, len:972800

 \ | /
- RT -     Thread Operating System
 / | \     4.0.1 build Jun  6 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0
    x00000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offset
    @0x0000f0c8.
[I/FAL] ==================== FAL partition table ====================
```

```
[I/FAL] | name       | flash_dev   |   offset   |    length  |
[I/FAL] -----------------------------------------------------------
[I/FAL] | easyflash  | norflash    | 0x00000000 | 0x00100000 |
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash    | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash    | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash    | 0x00900000 | 0x00700000 |
[I/FAL] ===========================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x00000000
    , size is 4096 bytes.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number is 0
    xD6363A94.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[D/main] The current version of APP firmware is 2.0.0
msh />[I/WLAN.mgnt] wifi connect success ssid:realthread
[I/WLAN.lwip] Got IP address : 192.168.12.92
```

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。

29.7 注意事项

- 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
- 如果要升级其他 APP 例程，请先将原 APP 例程移植到该例程工程，然后编译除 APP 固件，再进行升级操作

29.8 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《RT-Thread OTA 用户手册》：docs/UM1004-RT-Thread-OTA 用户手册.pdf
- 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 30 章

网络小工具集使用例程

30.1 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 IoT Board 开发板上使用 netutils 软件包的各种功能。

30.2 硬件说明

本例程需要依赖 WiFi 功能完成网络通信，因此请确保硬件平台上的 WiFi 功能可以正常工作，并且能够连接网络。

30.3 软件说明

30.3.1 主函数代码说明

在主函数中进行了如下操作：

1. 配置 wlan 的自动连接功能并开启自动连接。
2. 文件系统功能的初始化。

```
int main(void)
{
    struct rt_device *flash_dev;
    /* 初始化分区表 */
    fal_init();
    /* 初始化 easyflash */
    easyflash_init();
```

```
    /* 配置 wifi 工作模式 */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);

    /* 初始化自动连接配置 */
    wlan_autoconnect_init();
    /* 使能 wlan 自动连接 */
    rt_wlan_config_autoreconnect(RT_TRUE);

    /* 在 filesystem 分区上创建一个 Block 设备 */
    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);
    }

    /* 挂载 FAT32 文件系统 */
    if (dfs_mount(FS_PARTITION_NAME, "/", "elm", 0, 0) == 0)
    {
        LOG_I("Filesystem initialized!");
    }
    else
    {
        /* 创建 FAT32 文件系统 */
        dfs_mkfs("elm", FS_PARTITION_NAME);
        /* 再次挂载 FAT32 文件系统 */
        if (dfs_mount(FS_PARTITION_NAME, "/", "elm", 0, 0) != 0)
        {
            LOG_E("Failed to initialize filesystem!");
        }
    }
    return 0;
}
```

30.3.2 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

名称	分类	功能简介
Iperf	性能测试	测试最大 TCP 和 UDP 带宽性能，可以报告带宽、延迟抖动和数据包丢失
NetIO	性能测试	测试网络的吞吐量的工具
Telnet	远程访问	可以远程登录到 RT-Thread 的 Finsh/MSH Shell
tcpdump	网络调试	tcpdump 是 RT-Thread 基于 lwIP 的网络抓包工具

netutils 软件包文件结构如下所示：

```
netutils                // netutils 文件夹
    ├──iperf            // iperf 网络性能测试
    ├──netio            // netio 网络吞吐量测试
    ├──ntp              // ntp 时间同步功能
    ├──ping             // ping 功能
    ├──tcpdump          // 网络抓包工具
    ├──telnet           // telnet 服务器
    ├──tftp             // TFTP 功能
    └──tools            // 网络测试工具
```

30.4 运行

30.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。
- **IAR**：双击 `project.eww` 打开 IAR 工程，执行编译。

编译完成后，将开发板与 PC 机连接，然后将固件下载至开发板。

30.4.2 运行效果

30.4.2.1 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 lfs filesystem` 命令，该命令会在存储设备中名为 “filesystem” 的分区上创建 `LittleFS`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
 / | \     4.0.1 build Jun 11 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[I/FAL] The FAL block device (filesystem) created successfully
[I/main] Filesystem initialized!
msh />[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.1.2
msh />
```

30.4.2.2 连接无线网络

如果没有连接或者网络相关信息被擦除，则需要在程序运行后会进入 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：

```
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
```

30.4.2.3 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 />
```

30.4.2.4 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。

Finsh/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 2018
The system time is updated. Timezone is 8.
msh />date                    # 打印当前时间
Fri Sep 21 09:39:47 2018
```

30.4.2.5 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-v1.0.0/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_server
TFTP server start successfully.
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

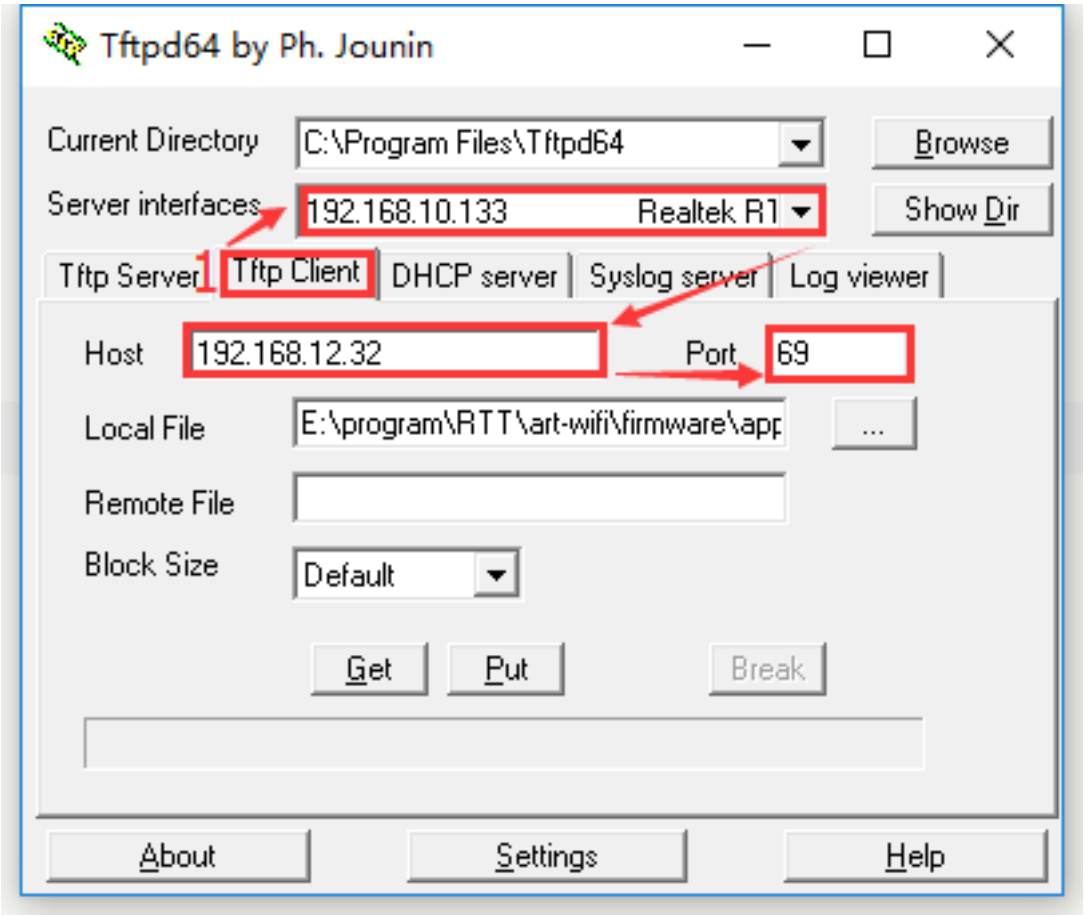


图 30.1: *tftpd config*

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 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 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**：

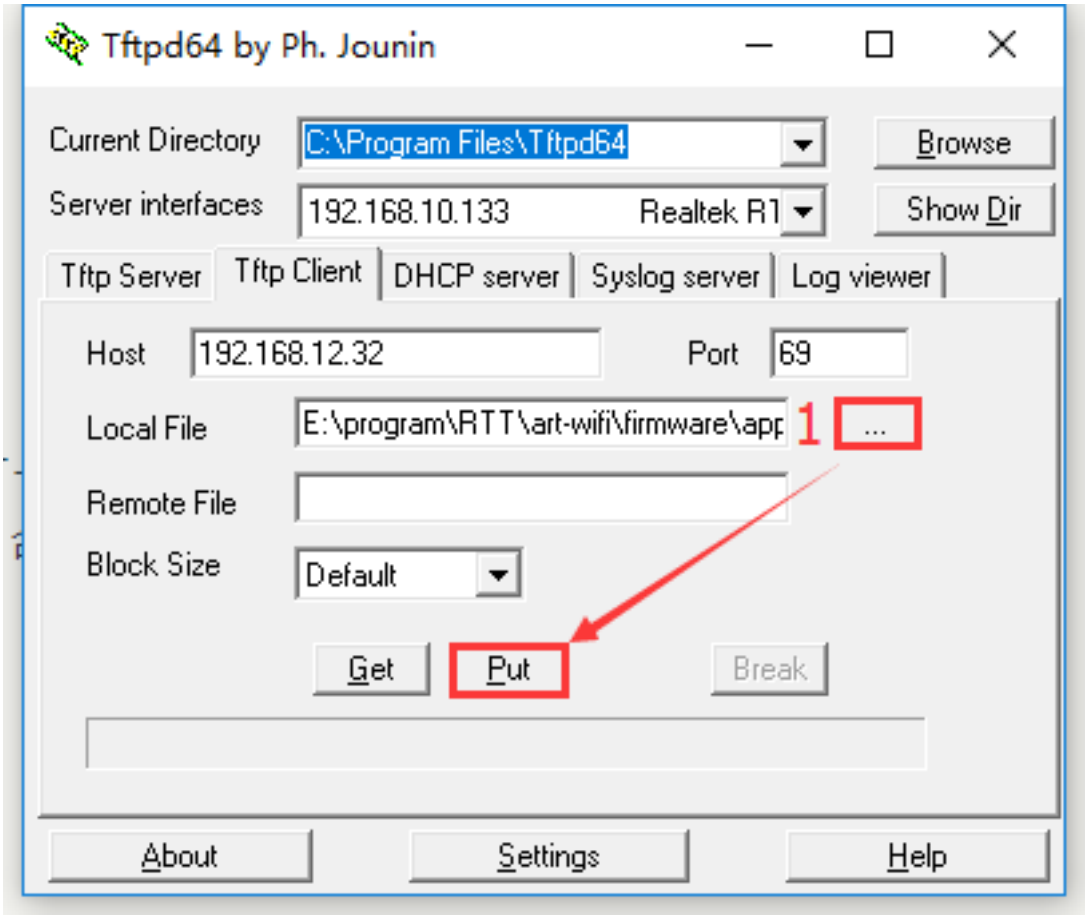


图 30.2: *tftpd put*

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>
```

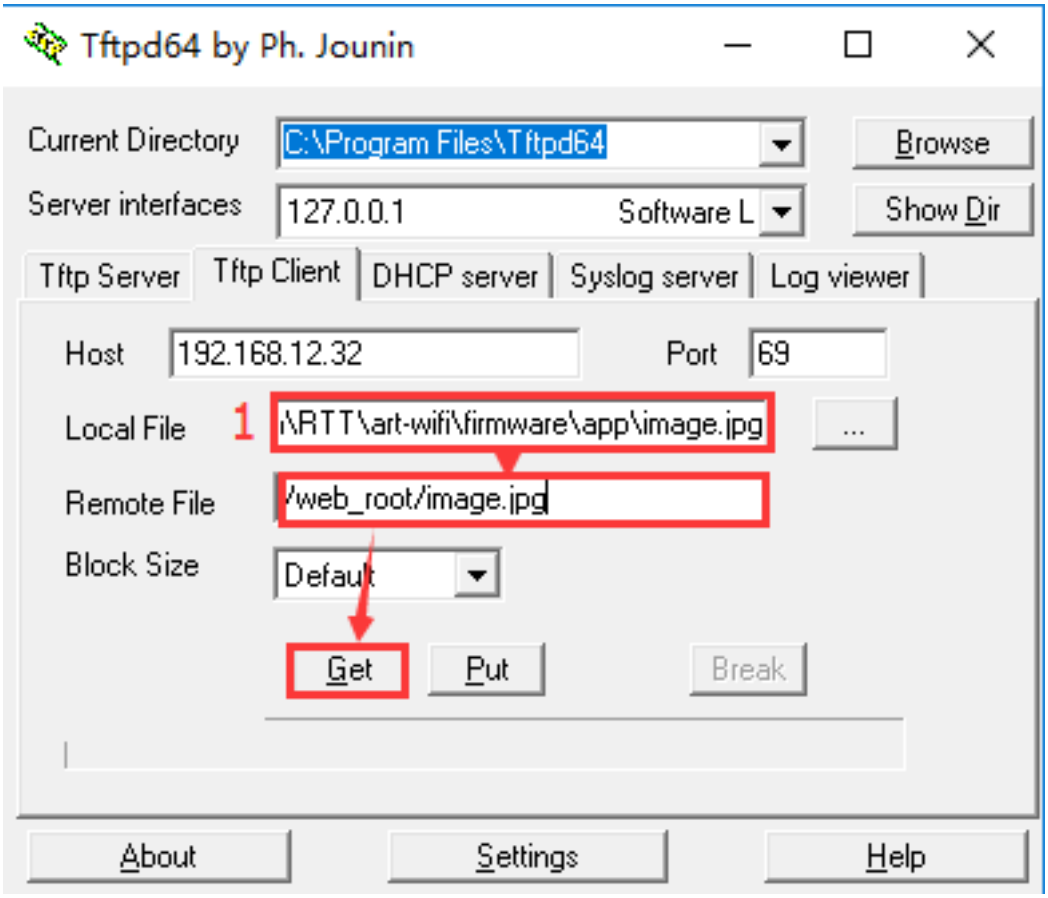


图 30.3: *tftpd get*

30.4.2.6 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 测试吗，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 30 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-v1.0.0/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。

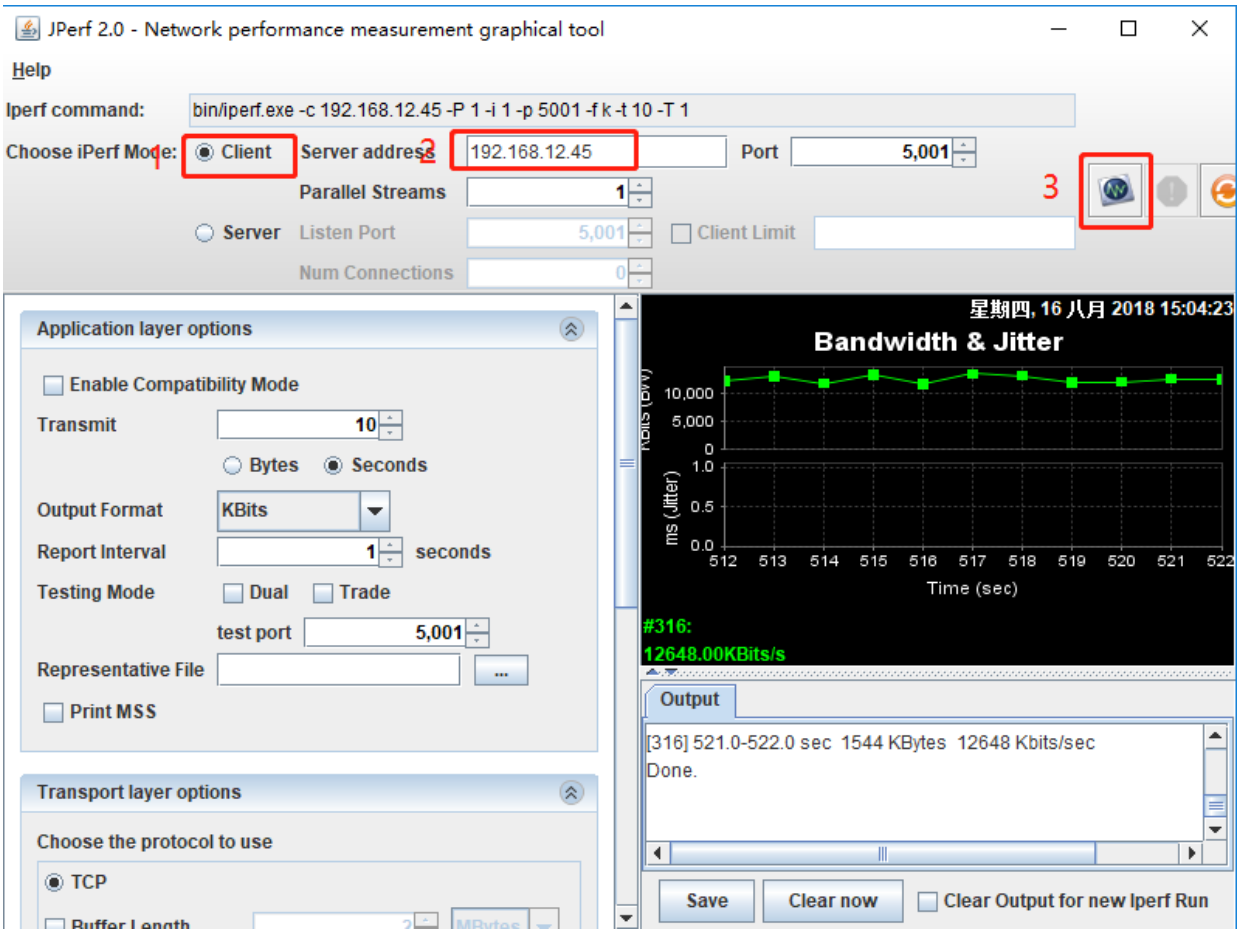


图 30.4: *iperf client*

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 192.168.10.138 -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。

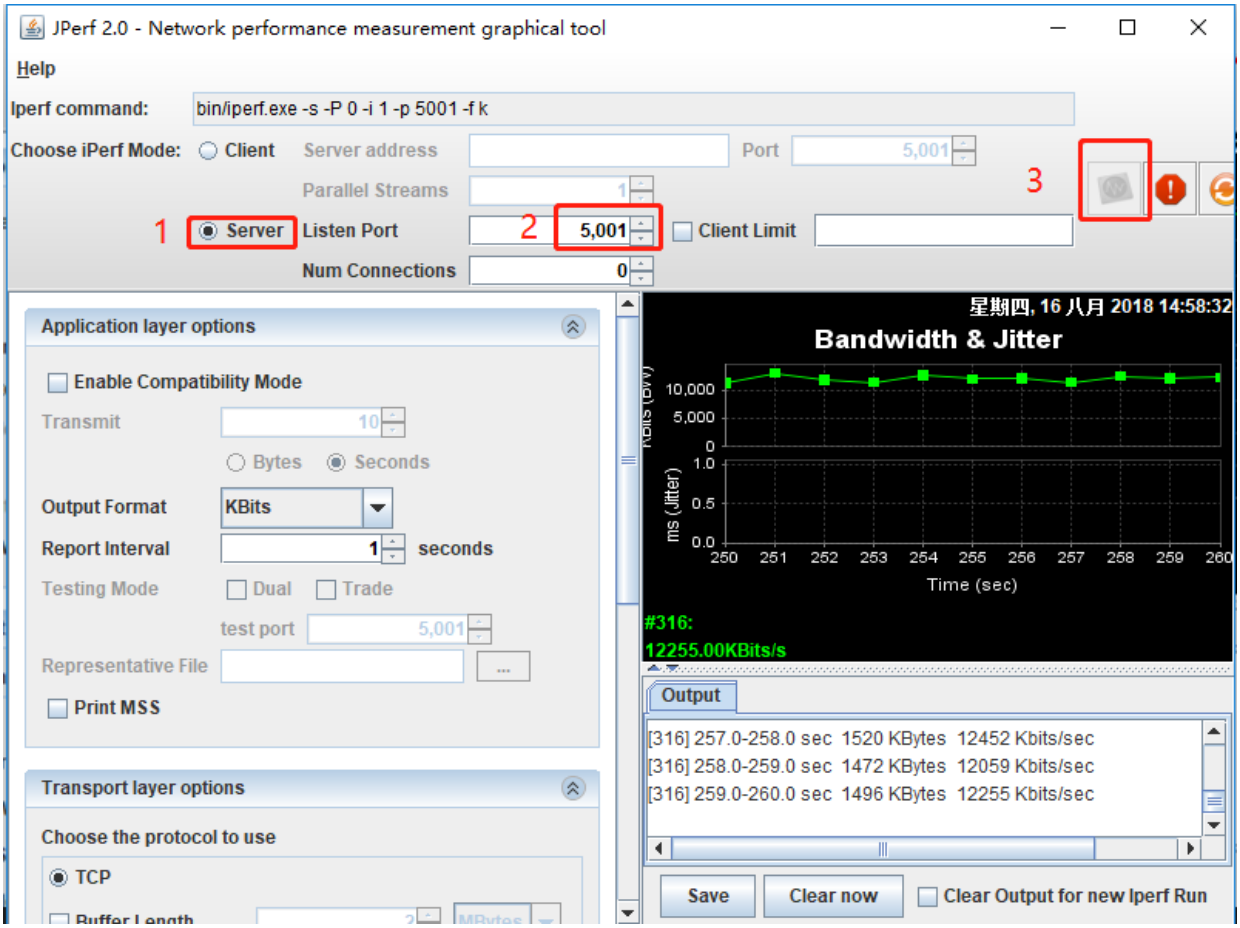


图 30.5: *iperf server*

30.4.2.7 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。

30.5 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。

30.6 引用参考

- 《RT-Thread 网络工具集 (NetUtils) 应用笔记》：docs/AN0018-RT-Thread-网络工具集应用笔记.pdf

第 31 章

RT-Thread 设备维护云平台接入例程

本例程演示如何使用 RT-Thread 提供的 CloudSDK 库接入 RT-Thread 云平台，实现远程 Shell 控制、远程 Log 存储和 OTA 升级功能。初次使用 RT-Thread 云平台的用户请先阅读《RT-Thread 云平台用户手册》(docs/UM1008-RT-Thread-设备维护云平台用户手册.pdf)。

31.1 平台简介

RT-Thread 云平台 是由上海睿赛德电子科技有限公司开发的一套物联网设备维护云平台。其目的是帮助开发者搭建安全有效的数据传输通道，方便设备终端和云端的双向通讯，在云端实现设备的升级、维护与管理功能。

该云平台旨在对接入产品和设备进行安全且高效的管理，可以实现对设备的远程操控、日志存储管理以及对固件的版本管理功能，帮助开发者快速搭建稳定可靠的物联网设备维护云平台。

31.2 主要功能

- **Web Shell 功能**

  RT-Thread 云平台实现远程 Shell 控制功能，用户无需连接串口设备即可完成对设备的控制、管理，满足用户对设备远程管理的需求。

- **Web Log 功能**

  RT-Thread 云平台实现设备日志的实时显示和存储功能，方便设备数据的采集以及设备状态的查看功能，用户可以通过 Web Log 随时查看设备动态及设备日志历史记录。

- **OTA 升级功能**

  RT-Thread 云平台实现设备远程升级功能，OTA 功能支持加密压缩升级、多固件升级、断点续传，满足用户对多种设备的 OTA 升级需求。

31.3 硬件说明

本例程需要依赖 WiFi 功能完成网络通信，因此请确保硬件平台上的 WiFi 功能可以正常工作，并且能够连接网络。

31.4 软件说明

31.4.1 准备工作

在使用本例程前需要先在 RT-Thread 云平台 注册账号，使用该账号在云平台中创建新产品，然后使用设备唯一标识符 **SN**（该示例中 **SN** 可以由用户自定义）在云端创建新设备，具体的流程参考《RT-Thread 云平台用户手册》(docs/UM1008-RT-Thread-设备维护云平台用户手册.pdf)。

产品和设备创建完成后，记录下**产品信息**页面的**产品 ID（ProductID）**和 **产品密钥（ProductKey，通常需要手动点开查看）**。下图为本次演示使用的 **ProductID** 和 **ProductKey** 位置：

图 31.1: *产品信息*

31.4.2 例程移植

31.4.2.1 移植流程

打开 `/examples/31_cloud_rtt/ports/cloudsdk/rt_cld_port.c` 文件，找到 `CLD_SN`，`CLD_PRODUCT_ID`，`CLD_PRODUCT_KEY` 这三个宏定义，将原来的内容替换成刚刚记录下来的 **ProductIdD** 和 **ProductKey**，**SN** 替换成云端创建设备时使用的 **SN**，保存文件，重新编译烧写程序，完成移植。（OTA 相关移植函数根据用户需求自定义实现，本例程使用默认接口不需要修改）

31.4.2.2 移植接口介绍

本例程移植主要是对 `/examples/31_cloud_rtt/ports/cloudsdk/rt_cld_port.c` 文件中用户自定义函数的实现。主要移植接口介绍如下：

SN 获取接口实现

```
void cld_port_get_device_sn(char *sn);
```

获取 SN（设备唯一标识符），注意与云端新建设备时使用的 SN 一致，格式为不大于 32 字节长的随机字符串形式，例如：`EF4016D6658466CA3E3610`。

产品 ID 获取接口实现

```
void cld_port_get_product_id(char *id);
```

获取产品 ID（ProductID），产品 ID 可以产品信息页面查询。

产品密钥获取接口实现

```
void cld_port_get_product_key(char *key);
```

获取产品密钥（ProductKey），产品密钥可以产品信息页面查询。

OTA 启动接口实现

```
void cld_port_ota_start(void);
```

用于 OTA 升级任务启动前，配置相关参数或执行相关操作，若不需要可置为空。

OTA 结束接口实现

```
void cld_port_ota_end(enum cld_ota_status status);
```

用于 OTA 升级结束后，根据 OTA 升级状态进行相应处理，例如：OTA 成功后设备复位进入 bootloader。

OTA 升级状态	介绍
CLD_OTA_OK	OTA 升级成功
CLD_OTA_ERROR	OTA 升级失败
CLD_OTA_NOMEM	OTA 升级内存不足

31.4.3 例程说明

本例程主要实现了连接 WiFi 成功后设备自动连接 RT-Thread 云平台。

在 main 函数中，主要完成了以下两个任务：

- 注册 CloudSDK 启动函数为 WiFi 连接成功的回调函数
- 启动 WiFi 自动连接功能

  当 WiFi 连接成功后，会调用 `/libraries/cloudsdk/libs` 目录下库文件中的 `rt_cld_init` 云端初始化函数，完成设备自动连接云端任务。

31.5 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译例程代码，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
 / | \     4.0.1 build May  6 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/SAL_SKT] Socket Abstraction Layer initialize success.
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[D/main] start to autoconnect ...
[D/main] The current version of APP firmware is 1.0.0
msh />[I/WLAN.mgnt] wifi connect success ssid:realthread
[I/OTA] RT-Thread OTA package(V0.2.3) initialize success.
[I/cld] The device has been activated successfully!
[I/cld.mqtt] CloudSDK MQTT server is startup!
[D/MQTT] ipv4 address port: 1883
[D/MQTT] HOST = 'iot.rt-thread.com'
[I/cld] RT-Thread CloudSDK package(V2.0.0) initialize success.
[I/WLAN.lwip] Got IP address : 192.168.12.108
[I/MQTT] MQTT server connect success
[I/MQTT] Subscribe #0 /device/12345678/abcdefgh/# OK!
```

31.5.1 连接无线网络

程序使用自动连接，如果没有连接 wifi，则需要手动连接。使用 MSH 命令 wifi join <ssid> <password> 可以让设备接入网络，如下所示：

```
msh />wifi join ssid_test 12345678
join ssid:ssid_test
[I/WLAN.mgnt] wifi connect success ssid:ssid_test
.......
msh />[I/WLAN.lwip] Got IP address : 152.10.200.224
```

31.5.2 设备自动上线

网络连接成功，程序会自动进行 RT-Thread 云平台初始化，设备自动上线，如下所示：

```
[I/cld] The device has been activated successfully!
```

```
[I/cld.mqtt] CloudSDK MQTT server is startup!
[I/cld] RT-Thread CloudSDK package(V2.0.0) initialize success.
[I/WLAN.lwip] Got IP address : 192.168.1.123
[I/MQTT] MQTT server connect success
[I/MQTT] Subscribe #0 /device/12345678/abcdefgh/# OK!
```

31.5.3 Web Shell 功能

Web Shell 的实现基于 TCP/IP 协议和 MQTT 协议，主要作用是实现远程 Shell 控制功能，用户无需连接串口设备即可在云端完成设备的管理和调试，并且实时显示设备打印信息。

设备上线成功，云端点击设备信息->（设备）详情->shell：连接‘，在云端实现 Shell 控制台功能：

图 31.2: *Web Shell 位置*

点击连接之后，设备端控制台将切换到云端显示。类似于 Shell 控制台，此时在云端输入命令可以得到相应响应，如下图所示：

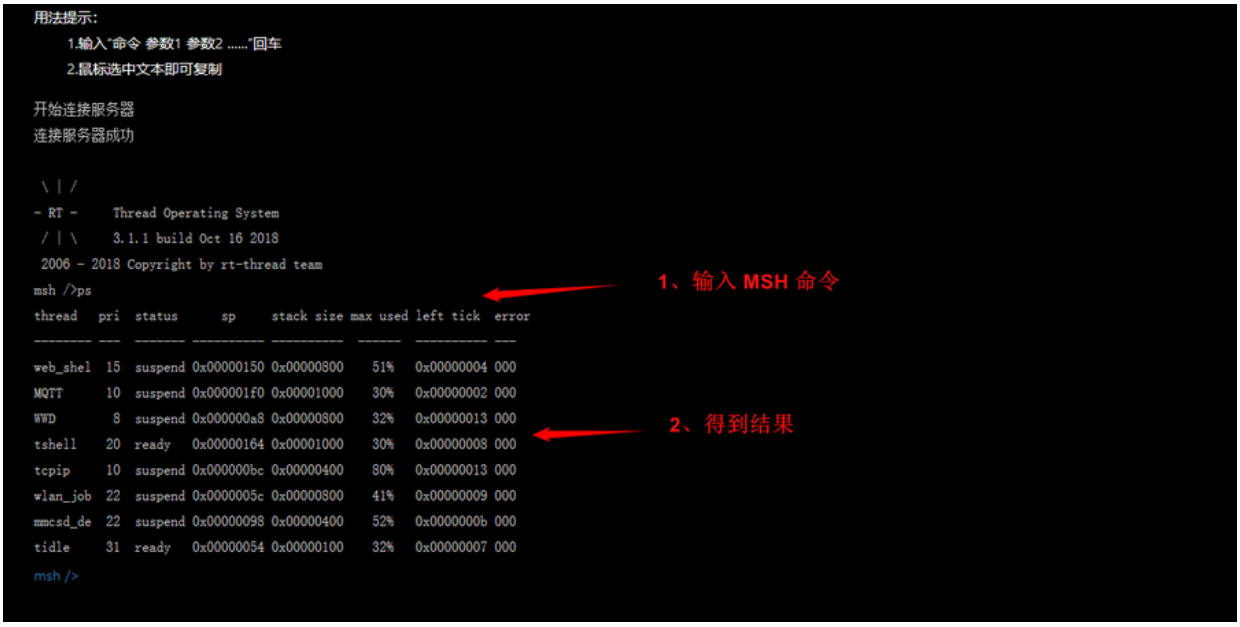


图 31.3: *启动 Web Shell*

31.5.4 Web Log 功能

Web Log 与 Web Shell 类似，主要作用是实现对 Shell 控制台输入输出日志的存储和查询功能。它与 Web Shell 主要的区别是 Web Shell 功能是对 Shell 控制台输入输出的实时显示与管理控制，Web Log 功能是对 Shell 控制台输入输出的记录存储，方便后期查看。

设备上线成功，云端点击设备信息->设备详情->开启日志功能：开启，云端开启设备 Web Log 日志记录功能，设备控制台的输入输出日志会发送到云端记录保存，再次点击开启日志功能：关闭可关闭日志功能。

Web Log 功能自带超时处理机制，开启 Web Log 功能后 5 分钟内无数据传输，服务器会主动关闭 Web Log 功能。

图 31.4: *Web Log 位置*

开启 Web Log 功能后，可在本地 MSH 命令行中输入 ps 命令查看当前线程状态，显示的日志会发送到云端并存储在日志列表中，之后在云端点击查看设备日志：日志列表，可以查看历史日志信息。

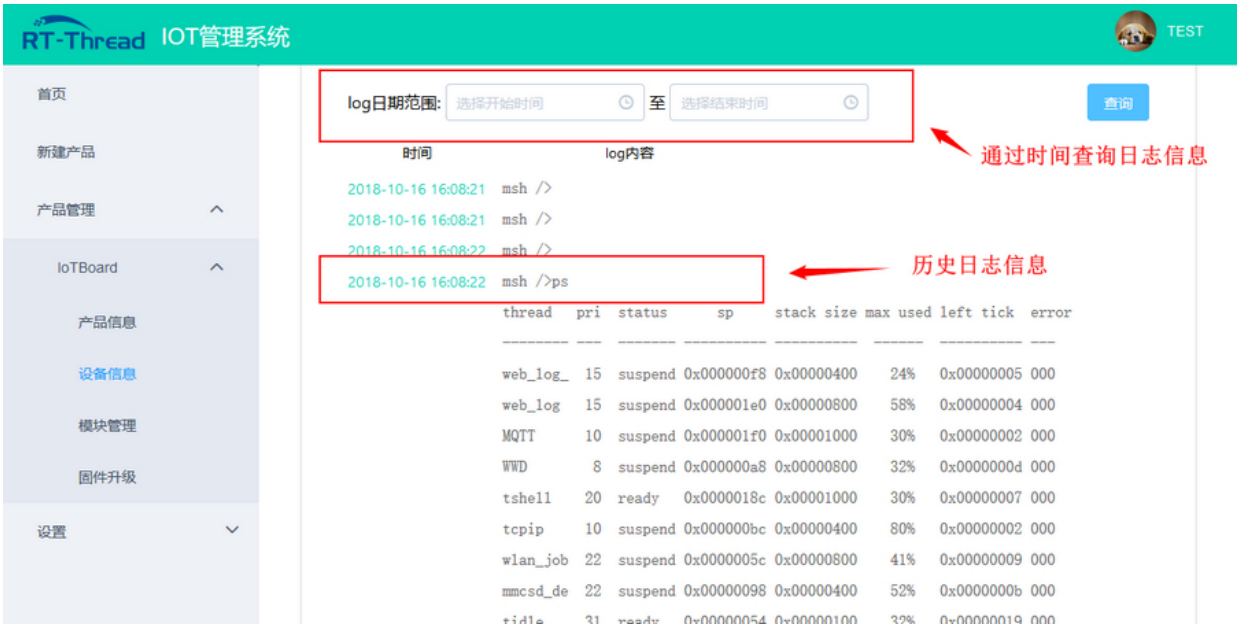


图 31.5: *查看历史日志*

31.5.5 OTA 升级功能

RT-Thread 云平台 OTA 升级功能可以实现设备远程升级。相比于其他的设备升级方式，RT-Thread 云平台具有以下特点：

- 可适配不同型号的 flash 或文件系统
- 支持云端加密数据传输
- 支持固件加密和压缩功能
- 支持**断点续传**功能
- 支持**多固件升级**功能

31.5.5.1 制作升级固件

云端 OTA 升级时所需的固件文件需要特定的格式固件支持，为此我们提供 RT-Thread OTA 固件打包器，位于 /tools/ota_packager/rt_ota_packaging_tool.exe 。固件打包工具可以将原格式固件文件做加密、压缩处理，生成特定格式（.rbl 后缀）的升级固件文件，用于后期上传至云端及在云端建立升级任务。

以 **31_iot_cloud_rtt** 例程为基础，制作用于 OTA 升级演示所用到的 **app 固件**。

1. **MDK**：双击 project.uvprojx 打开 MDK5 工程
2. **IAR**：双击 project.eww 打开 IAR 工程
3. 修改 /examples/31_iot_cloud_rtt/application/main.c 中的版本号 `##define APP_VERSION "1.0.0"` 为 `##define APP_VERSION "2.0.0"`
4. 编译得到 rt-thread.bin，文件位置 /examples/31_iot_cloud_rtt/rt-thread.bin

5. 使用 OTA 固件打包工具打包生成 `rt-thread.rbl` 文件

OTA 固件打包工具的界面如下图所示：

图 31.6: *OTA 打包工具*

工具使用方式

用户可以根据需求，选择是否对固件进行加密和压缩，工具提供多种压缩和加密算法支持。具体操作步骤如下：

1. 选择待打包的固件（`/examples/31_iot_cloud_rtt/rt-thread.bin`）
2. 选择生成固件的位置
3. 选择压缩算法（本例程仅支持 **gzip** 压缩，不压缩则留空）
4. 选择加密算法（本例程仅支持 **AES256** 加密，不加密则留空）

5. 配置加密密钥（不加密则留空）
6. 配置加密 IV （不加密则留空）
7. 填写固件名称（对应分区名称，这里为 app）
8. 固件版本号（填写 /examples/31_iot_cloud_rtt/application/main.c 中的版本号 2.0.0）
9. 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 支持加密压缩，使用的 **加密密钥**为 00000000000000000000000000000000，使用的 **加密 IV** 为 1111111111111111。
- 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 **font**，则填写 font。
- 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**。

31.5.5.2 OTA 升级流程

固件信息:

rtthread.rbl：固件名称为 **app**，固件版本为 **2.0.0**，压缩算法为 **gzip**，加密算法为 **AES256**

固件上传:

生成的固件需要上传到云端进行管理，用于在云端新建升级任务。点击模块管理->添加固件，选择 OTA 打包工具生成的 rtthread.rbl 文件，上传到云端，如下图所示方式：

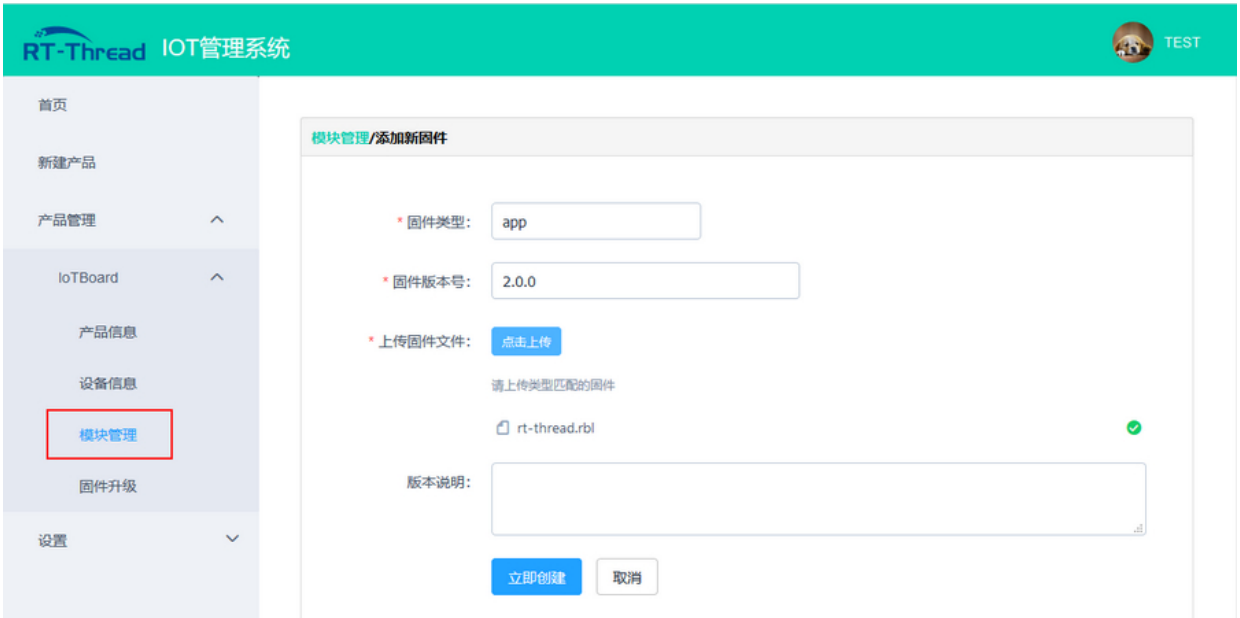


图 31.7: *固件上传到云端*

- 固件类型: 固件分区名称，与**工具生成固件时填入固件名称一致**；
- 固件版本号：需要与**工具生成固件时填入固件版本一致**，且**不同于**云端最新版本号；

新建 OTA 任务：

固件上传成功后，可以通过云端新建 OTA 升级任务（支持多固件升级），云端点击固件升级->新建OTA版本 选出刚才上传的固件建立 OTA 升级任务，之后云端会向设备推送升级请求，对设备进行升级。

图 31.8: *新建 OTA 升级任务*

- 升级模式：

```
强制升级：下发升级任务，设备立刻执行下载升级
可选升级：下发升级任务，用户可自定义执行下载升级的条件
静默升级：下发升级任务，设备无提示执行下载升级
```

- 升级顺序：多个固件升级时，用户可自定义多个固件的升级顺序，云端下发升级任务，设备会根据固件升级顺序依次来升级固件；
- 可升级列表：用户可自定义添加需要升级的设备到设备列表中，云端将下发本次升级任务到设备列表中的设备上；

设备 OTA 升级：

云端升级任务创建成功之后，云端会通过 MQTT 协议下发固件升级信息，设备获取升级信息后会下载新的固件。若为**多固件升级**，设备每次升级完一个固件会重启一次，直到最后一个固件升级成功。如果升级过程中若出现断电或者下载失败，设备支持**断点续传**功能，避免固件重复下载，减少固件升级时间。下图为固件下载过程：

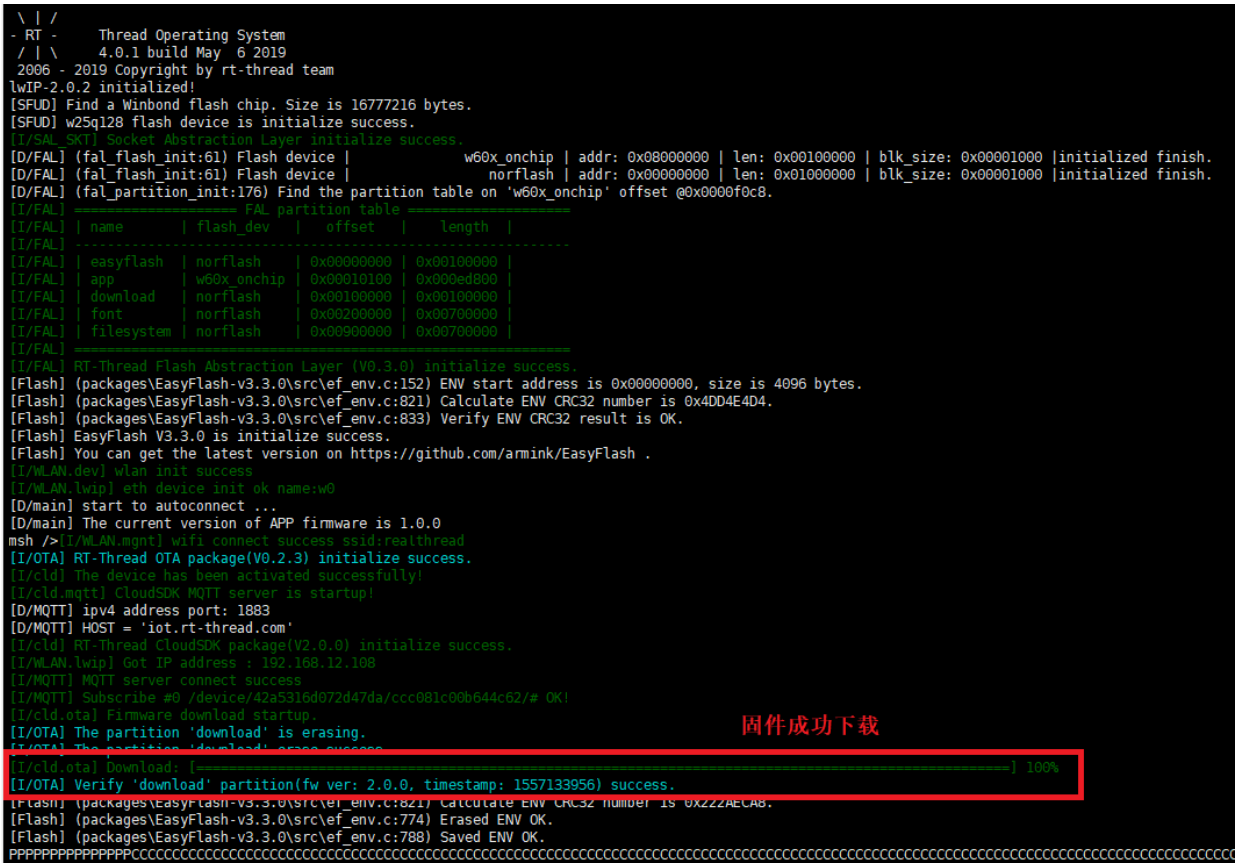


图 31.9: *OTA 开始升级*

设备固件下载完成之后，程序会自动跳转到 bootloader 中运行，bootloader 会对固件进行解压解密，并将固件拷贝到指定的应用分区（这里是 **app** 分区）。bootloader 中解压解密和升级成功后设备状态如下图所示：

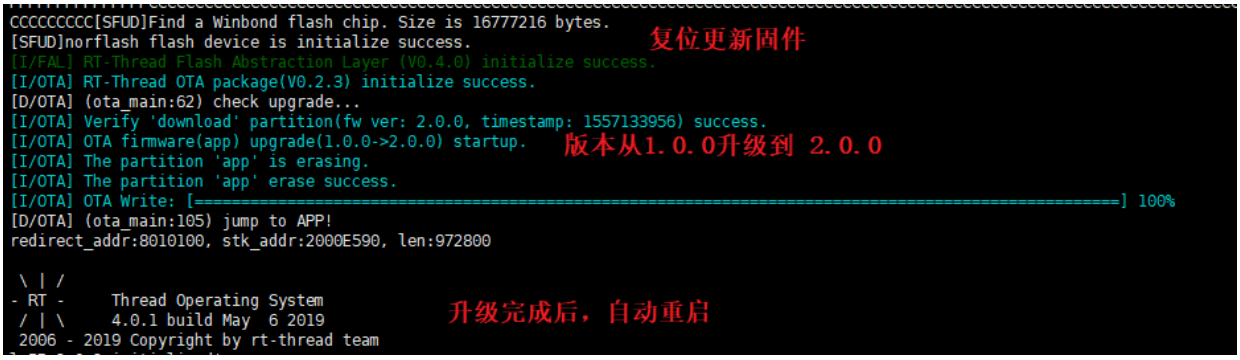


图 31.10: *OTA 升级完成*

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 **1.6.0 版本**升级到了 **1.6.6 版本**，设备 OTA 升级完成。

更多详细介绍，请阅读《RT-Thread 云平台用户手册》(docs/UM1008-RT-Thread-设备维护云平台用户手册.pdf)。

31.6 注意事项

- 使用本例程前请先修改 examples/31_cloud_rtt/ports/cloudsdk/rt_cld_port.c 里的两个 RT-Thread 云平台宏定义 (CLD_SN，CLD_PRODUCT_ID，CLD_PRODUCT_KEY)；
- 设备上电自动激活过程若出现 **400 错误**，检查 port 文件中修改的 SN 是否和云端注册时使用的一致；
- Web Log 功能开启后不能使用 Web Shell 功能；
- OTA 固件打包工具中使用的加密密钥和 IV ，需要使用默认的加密密钥和 IV，不支持自定义设置，如需定制可改动版请与 RT-Thread 官方联系。

31.7 引用参考

- 《RT-Thread 云平台用户手册》：docs/UM1008-RT-Thread-设备维护云平台用户手册.pdf

第 32 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 用户手册》。

32.1 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。

32.2 硬件说明

本例程需要依赖 WiFi 功能完成网络通信，因此请确保硬件平台上的 WiFi 功能可以正常工作，并且能够连接网络。

32.3 准备工作

32.3.1 创建设备

在使用本例程前需要在 OneNET 平台注册账号，并在帐号里创建产品，具体的流程参考《OneNET 示例说明》。产品创建完成后，记录下产品概况页面的产品 ID 和 APIKey。

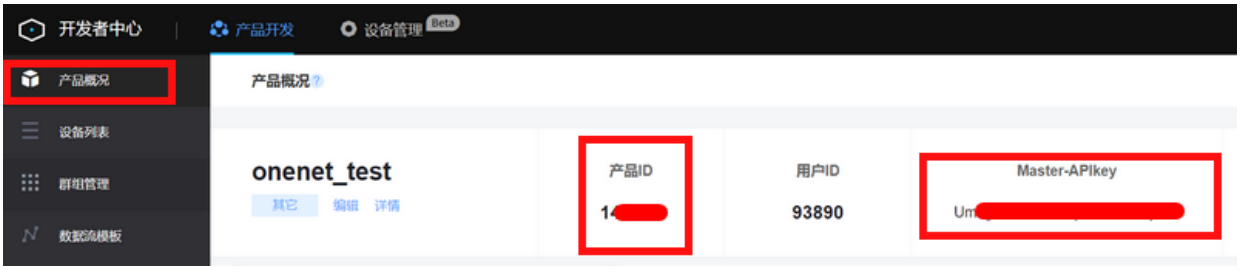


图 32.1: 设备信息

切换到设备管理界面，记录下设备注册码。

图 32.2: 环境注册码

打开 /examples/32_cloud_onenet/rtconfig.h，找到 ONENET_REGISTRATION_CODE，ONENET_INFO_PROID，ONENET_MASTER_APIKEY 这三个宏定义，将原来的内容分别替换成刚刚记录下来的设备注册码、产品 ID 和 APIKey，然后保存文件。

32.3.2 代码移植

设备注册，设备上线，需要用到一些需要移植的接口函数，下面将具体介绍需要用到的四个接口函数。IoT Board 关于 OneNET 的移植代码位于 /examples/32_cloud_onenet/applications/main.c 文件中。

32.3.2.1 保存设备信息

注册设备成功后，OneNET 平台会返回设备 ID 和 api key，用户需要将这两个信息保存起来，以便在下次开机时能读取这两个信息用于登录 OneNET 平台。除了保存 2 个设备信息外，用户还应该保存个已经注册成功的标志位，用来在开机时判断设备是否已经注册。

```
rt_err_t onenet_port_save_device_info(char *dev_id, char *api_key)
{
    EfErrCode err=EF_NO_ERR;

    /* 保存设备 ID */
```

```
    err = ef_set_and_save_env("dev_id", dev_id);
    if (err != EF_NO_ERR)
    {
        LOG_E("save device info(dev_id : %s) failed!", dev_id);
        return -RT_ERROR;
    }

    /* 保存设备 api_key */
    err = ef_set_and_save_env("api_key", api_key);
    if (err != EF_NO_ERR)
    {
        LOG_E("save device info(api_key : %s) failed!", api_key);
        return -RT_ERROR;
    }

    /* 保存环境变量：已经注册 */
    err = ef_set_and_save_env("already_register", "1");
    if (err != EF_NO_ERR)
    {
        LOG_E("save already_register failed!");
        return -RT_ERROR;
    }

    return RT_EOK;
}
```

32.3.2.2 获取注册设备信息

设备注册需要提供设备名字和鉴权信息，这里选取 mac 地址作设备名字和鉴权信息，除了赋值给 2 个入参外，还应该作为鉴权信息保存起来，用于下次开机登录 OneNET 平台。

```
rt_err_t onenet_port_get_register_info(char *dev_name, char *auth_info)
{
    rt_uint32_t udid[2] = {0}; /* 唯一设备标识 */
    EfErrCode err = EF_NO_ERR;

    /* 获得 MAC 地址 */
    if (rt_wlan_get_mac((rt_uint8_t *)udid) != RT_EOK)
    {
        LOG_E("get mac addr err!! exit");
        return -RT_ERROR;
    }

    /* 设置设备名和鉴权信息 */
    rt_snprintf(dev_name, ONENET_INFO_AUTH_LEN, "%d%d", udid[0], udid[1]);
    rt_snprintf(auth_info, ONENET_INFO_AUTH_LEN, "%d%d", udid[0], udid[1]);

    /* 保存设备鉴权信息 */
```

```
    err = ef_set_and_save_env("auth_info", auth_info);
    if (err != EF_NO_ERR)
    {
        LOG_E("save auth_info failed!");
        return -RT_ERROR;
    }

    return RT_EOK;
}
```

32.3.2.3 获取设备信息

获取用于登录的设备信息功能的代码如下所示：

```
rt_err_t onenet_port_get_device_info(char *dev_id, char *api_key, char *auth_info)
{
    char *info = RT_NULL;

    /* 获取设备 ID */
    info = ef_get_env("dev_id");
    if (info == RT_NULL)
    {
        LOG_E("read dev_id failed!");
        return -RT_ERROR;
    }
    else
    {
        rt_snprintf(dev_id, ONENET_INFO_AUTH_LEN, "%s", info);
    }

    /* 获取 api_key */
    info = ef_get_env("api_key");
    if (info == RT_NULL)
    {
        LOG_E("read api_key failed!");
        return -RT_ERROR;
    }
    else
    {
        rt_snprintf(api_key, ONENET_INFO_AUTH_LEN, "%s", info);
    }

    /* 获取设备鉴权信息 */
    info = ef_get_env("auth_info");
    if (info == RT_NULL)
    {
        LOG_E("read auth_info failed!");
        return -RT_ERROR;
```

```
    }
    else
    {
        rt_snprintf(auth_info, ONENET_INFO_AUTH_LEN, "%s", info);
    }

    return RT_EOK;
}
```

32.3.2.4 查询设备注册状态

检查设备是否已经注册的代码如下所示：

```
rt_bool_t onenet_port_is_registed(void)
{
    char *already_register = RT_NULL;

    /* 检查设备是否已经注册 */
    already_register = ef_get_env("already_register");
    if (already_register == RT_NULL)
    {
        return RT_FALSE;
    }

    return already_register[0] == '1' ? RT_TRUE : RT_FALSE;
}
```

32.4 软件说明

本例程主要实现了每隔 5s 往 OneNET 平台上传一次环境光强度，并可以执行 OneNET 下发命令的功能，程序代码位于 /examples/32_cloud_onenet/applications/main.c 文件中。

在 main 函数中，主要完成了以下几个任务：

- 初始化 LED 管脚
- 初始化 ap3216c 传感器
- 注册 OneNET 启动函数为 WiFi 连接成功的回调函数
- 启动 WiFi 自动连接功能

当 WiFi 连接成功后，会调用 /examples/32_cloud_onenet/packages/onenet-1.0.0/src/onenet_mqtt.c 文件中的 onenet_mqtt_init() 函数，该函数会获取设备信息完成设备上线的任务。如果设备未注册，会自动调用注册函数完成注册。

OneNET 设备上线后，会自动执行 /examples/32_cloud_onenet/applications/main.c中的 onenet_upload_cycle() 函数，函数会设置命令响应的回调函数，并启动一个 onenet_send 的线程，线程会每隔 5 秒获取一次 ap3216c 传感器的环境光强度数据，并将这个数据上传到 OneNET 的 light 数据流中，发送 100 次后线程自动结束。上传数据的代码如下所示：

```
static void onenet_upload_entry(void *parameter)
{
    struct rt_sensor_data als_dev_data = { 0 };
    int i = 0;

    /* 往 light 数据流上传环境光数据 */
    for (i = 0; i < NUMBER_OF_UPLOADS; i++)
    {
        if (als_dev)
        {
            rt_device_read(als_dev, 0, &als_dev_data, 1);
        }

        if (onenet_mqtt_upload_digit("light", als_dev_data.data.light / 10) < 0)
        {
            LOG_E("upload has an error, stop uploading");
            break;
        }
        else
        {
            LOG_D("buffer : {\"light\":%d}", als_dev_data.data.light / 10);
        }

        rt_thread_mdelay(5 * 1000);
    }
}
```

命令响应回调函数里主要完成的是命令打印，命令匹配的任务。当匹配到 ledon 和 ledoff 这两个命令时，会执行打开 led 和关闭 led 的动作，并向云端发送响应。具体代码如下所示：

```
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[20] = { 0 };

    LOG_D("recv data is %.*s\n", recv_size, recv_data);

    /* 命令匹配 */
    if (rt_strncmp(recv_data, "ledon", 5) == 0)
    {
        /* 开灯 */
        rt_pin_write(LED_PIN, PIN_LOW);

        rt_snprintf(res_buf, sizeof(res_buf), "led is on");

        LOG_D("led is on\n");
    }
    else if (rt_strncmp(recv_data, "ledoff", 6) == 0)
    {
```

```
        /* 关灯 */
        rt_pin_write(LED_PIN, PIN_HIGH);

        rt_snprintf(res_buf, sizeof(res_buf), "led is off");

        LOG_D("led is off\n");
    }

    /* 开发者必须使用 ONENET_MALLOC 为响应数据申请内存 */
    *resp_data = (uint8_t *) ONENET_MALLOC(strlen(res_buf) + 1);

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

    *resp_size = strlen(res_buf);
}
```

32.5 运行

32.5.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。
- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译完成后，将开发板与 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
 / | \     4.0.1 build Jun 26 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[D/soft_i2c] software simulation i2c2soft init done, pin scl: 25, pin sda 24
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[I/sensor] rt_sensor init success
[I/sensor] rt_sensor init success
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.12.30     # 自动成功连接 WiFi
```

32.5.2 连接无线网络

之前成功连接 WiFi 会自动连接，如果没有连接或者网络相关信息被擦除，则需要在程序运行后会进入 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：

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

32.5.3 清零注册标志

注册标志存储在 flash 中，连接之后，会保存注册信息与注册标志。第一次初始化下面的 OneNET mqtt 客户端，返回注册的设备信息，flash 对设备信息进行保存，并且将注册标志置为 1。如果需要重新注册设备，则需要将这个标志位清零。使用命令如下：

```
msh />setenv already_register 0     # 清零注册标志
msh />saveenv                       # 保存设置的变量
```

- 注：第一次初始化 OneNET mqtt 客户端建议清零，避免 already_register 为 1

32.5.4 初始化 OneNET mqtt 客户端

在 WiFi 连接成功后，输入命令 `onenet_mqtt_init` 对 OneNET mqtt 客户端进行初始化，日志如下：

```
msh />onenet_mqtt_init
[D/ONENET] (response_register_handlers:266) response is {"errno":0,"data":{"
    device_id":"532439787","key":"XGfOT4oeMZ0wzDsg2Y=ugQaUgD
M="},"error":"succ"}          #注册返回的设备信息
[D/ONENET] (mqtt_connect_callback:85) Enter mqtt_connect_callback!
[D/MQTT] ipv4 address port: 6002
[D/MQTT] HOST = '183.230.40.39'
[I/ONENET] RT-Thread OneNET package(V1.0.0) initialize success.
msh />[I/MQTT] MQTT server connect success
[D/ONENET] (mqtt_online_callback:90) Enter mqtt_online_callback!   # 链接成功
```

- 注：建立 WiFi 连接后才可能初始化成功

32.5.5 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] (onenet_upload_entry:82) buffer : {"light":20}    #上传的数据
[D/ONENET] (onenet_upload_entry:82) buffer : {"light":28}
```

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

图 32.3: *数据流*

利用物体盖住传感器或利用 led 照射传感器，可以在 FinSH 控制台和 OneNET 的数据流页面看到 light 数据会有一个大的波动。

32.5.6 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。例程支持 ledon 和 ledoff 两个命令，板子收到这两个命令后会打开和关闭开发板上的红色 led。示例效果如下所示：

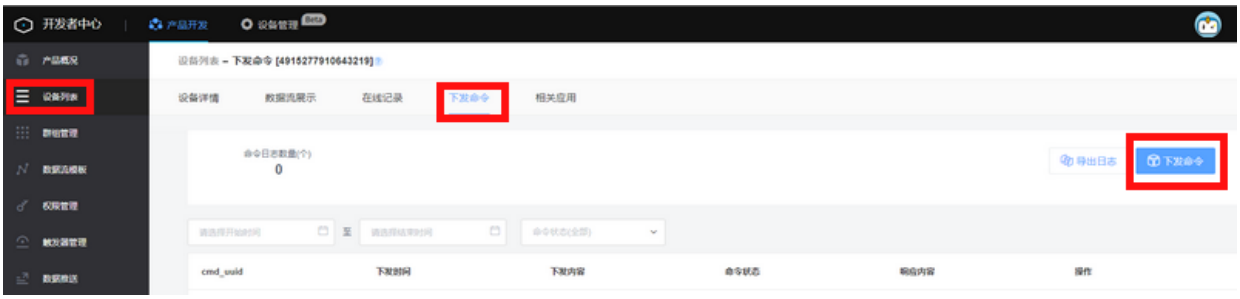


图 32.4: *设备控制界面*

图 32.5: 发送命令

```
[D/ONENET] (mqtt_callback:62) topic $creq/1798e855-c334-5f03-a81a-5d8f587a0bc9
    receive a message
[D/ONENET] (mqtt_callback:64) message length is 5
[D/ONENET] (onenet_cmd_rsp_cb:212) recv data is ledon
[D/ONENET] (onenet_cmd_rsp_cb:248) led is on
```

- 如果退出了数据流展示，无法直接使用**下发命令**按钮，按照下面操作：设备列表 -> 操作栏下 -> 更多操作 -> 拉框中使用**下发命令**。

32.6 注意事项

- 使用本例程前请先阅读《OneNET 用户手册》
- 使用本例程前请先修改 `rtconfig.h` 里的三个 OneNET 平台宏定义 (`ONENET_REGISTRATION_CODE`,`ONENET_INFO_PROID`,`ONENET_MASTER_APIKEY`)

32.7 引用参考

- 《GPIO 设备应用笔记》：docs/AN0002-RT-Thread-通用 GPIO 设备应用笔记.pdf
- 《OneNET 用户手册》：docs/UM1003-RT-Thread-OneNET 用户手册.pdf
- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf

第 33 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（LinkPlatform 和 LinkDevelop），以 LinkDevelop 平台为例，介绍如何通过 MQTT 协议接入 LinkDevelop 平台。初次使用阿里云物联网平台或者想了解更多使用例程，请阅读《ali-iotkit 用户手册》。

33.1 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

33.2 硬件说明

本例程需要依赖 WiFi 功能完成网络通信，因此请确保硬件平台上的 WiFi 功能可以正常工作，并且能够连接网络。

33.3 软件说明

ali-iotkit 例程位于 /examples/33_iot_cloud_ali_iotkit 目录下，重要文件摘要说明如下所示：

说明	文件
app 入口	applications/main.c
移植文件	ports
阿里云物联网平台接入软件包	packages/ali-iotkit-v2.0.3
阿里云物联网平台接入示例	packages/ali-iotkit-v2.0.3/samples

33.4 例程使用说明

该 ali-iotkit 演示例程程序代码位于 /examples/33_iot_cloud_ali_iotkit/packages/ali-iotkit-v2.0.3/samples/mqtt/mqtt-example.c 文件中，核心代码说明如下：

33.4.1 设置激活凭证

使用 menuconfig 设置设备激活凭证（从阿里云—— LinkDevelop 平台获取），如下图所示：

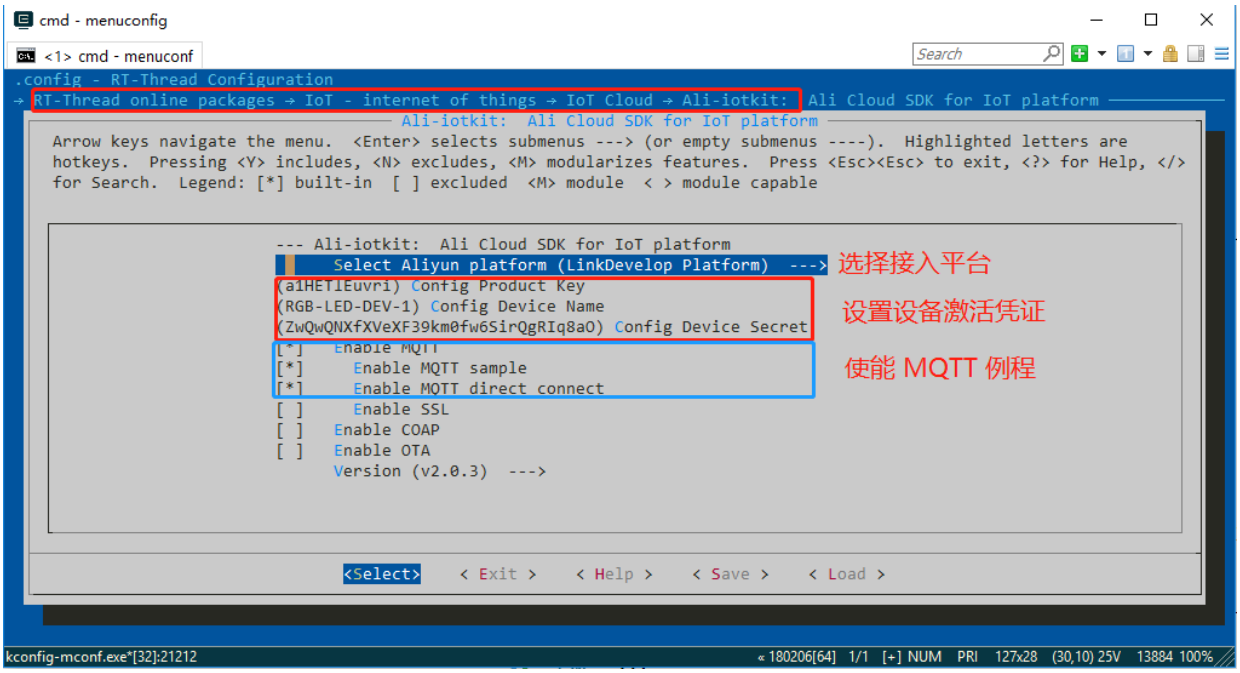


图 33.1: *设置激活凭证*

通过 menuconfig 工具完成配置的保存后，工具会将相关的配置项写入到 33_iot_cloud_ali_iotkit/rtconfig.h 文件中，如果用户无 ENV 工具，不能使用 menuconfig 进行配置，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"
```

33.4.2 注册 MQTT 事件回调

MQTT 事件回调函数 event_handle 用于处理 MQTT 状态事件（如上线、下线、重连等），在 MQTT 客户端创建的时候被注册，代码如下所示：

```
static void event_handle(void *pcontext, void *pclient, iotx_mqtt_event_msg_pt msg)
{
    iotx_mqtt_topic_info_pt topic_info = (iotx_mqtt_topic_info_pt)msg->msg;
    if (topic_info == NULL)
    {
        rt_kprintf("Topic info is null! Exit.");
        return;
    }
    uintptr_t packet_id = (uintptr_t)topic_info->packet_id;

    switch (msg->event_type) {
        case IOTX_MQTT_EVENT_UNDEF:
            EXAMPLE_TRACE("undefined event occur.");
            break;

        case IOTX_MQTT_EVENT_DISCONNECT:
            EXAMPLE_TRACE("MQTT disconnect.");
            break;

        case IOTX_MQTT_EVENT_RECONNECT:
            EXAMPLE_TRACE("MQTT reconnect.");
            break;
    /* 省略部分代码 */
    }
}
```

33.4.3 注册 MQTT 消息接收回调

每当 MQTT 客户端收到服务器发来的消息就会调用 _demo_message_arrive 函数，将收到的消息交由用户自定义处理，代码如下所示：

```
static void _demo_message_arrive(void *pcontext, void *pclient,
    iotx_mqtt_event_msg_pt msg)
```

```
{
    iotx_mqtt_topic_info_pt ptopic_info = (iotx_mqtt_topic_info_pt) msg->msg;

    /* 打印收到的消息的 Topic 主题和收到的消息内容 */
    EXAMPLE_TRACE("----");
    EXAMPLE_TRACE("packetId: %d", ptopic_info->packet_id);
    EXAMPLE_TRACE("Topic: '%.*s' (Length: %d)",
                  ptopic_info->topic_len,
                  ptopic_info->ptopic,
                  ptopic_info->topic_len);
    EXAMPLE_TRACE("Payload: '%.*s' (Length: %d)",
                  ptopic_info->payload_len,
                  ptopic_info->payload,
                  ptopic_info->payload_len);
    EXAMPLE_TRACE("----");
}
```

33.4.4 启动 MQTT 客户端

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（pconn_info），代码如下所示：

```
iotx_conn_info_pt pconn_info;
/* 设备鉴权 */
if (0 != IOT_SetupConnInfo(__product_key, __device_name, __device_secret, (void **)&
    pconn_info))
{
    EXAMPLE_TRACE("AUTH request failed!");
    rc = -1;
    goto do_exit;
}
```

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 */
memset(&mqtt_params, 0x0, sizeof(mqtt_params));
mqtt_params.port = pconn_info->port;
mqtt_params.host = pconn_info->host_name;
mqtt_params.client_id = pconn_info->client_id;
mqtt_params.username = pconn_info->username;
mqtt_params.password = pconn_info->password;
mqtt_params.pub_key = pconn_info->pub_key;
mqtt_params.request_timeout_ms = 2000;
```

```
mqtt_params.clean_session = 0;
mqtt_params.keepalive_interval_ms = 60000;
mqtt_params.pread_buf = msg_readbuf;
mqtt_params.read_buf_size = MQTT_MSGLEN;
mqtt_params.pwrite_buf = msg_buf;
mqtt_params.write_buf_size = MQTT_MSGLEN;
mqtt_params.handle_event.h_fp = event_handle;
mqtt_params.handle_event.pcontext = NULL;
```

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");
    rc = -1;
    goto do_exit;
}
```

订阅指定的 Topic

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

```
/* 订阅指定的 Topic */
rc = IOT_MQTT_Subscribe(pclient, ALINK_SERVICE_SET_SUB, IOTX_MQTT_QOS1,
    _demo_message_arrive, NULL);
if (rc < 0) {
    IOT_MQTT_Destroy(&pclient);
    EXAMPLE_TRACE("IOT_MQTT_Subscribe() failed, rc = %d", rc);
    rc = -1;
    goto do_exit;
}
```

ALINK_SERVICE_SET_SUB 消息主题为：##define ALINK_SERVICE_SET_SUB "/sys/"PRODUCT_KEY "/"DEVICE_NAME"/thing/service/property/set"，用于设置设备属性。

循环等待 MQTT 消息通知

使用 IOT_MQTT_Yield 函数接收来自云端的消息。

```
do {
    /* handle the MQTT packet received from TCP or SSL connection */
    IOT_MQTT_Yield(pclient, 200);
    HAL_SleepMs(2000);
} while (is_running);
```

33.4.5 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。

取消订阅指定的 Topic

使用 IOT_MQTT_Unsubscribe 取消已经存在的订阅。

```
IOT_MQTT_Unsubscribe(pclient, ALINK_SERVICE_SET_SUB);
```

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。

```
IOT_MQTT_Destroy(&pclient);
```

33.4.6 发布测试消息

构建 Alink 协议格式数据

本例程中构建一个 RGB 灯设备需要的 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 */
memset(msg_pub, 0x0, sizeof(msg_pub));
snprintf(msg_pub, sizeof(msg_pub),
        "{\"id\" : \"%d\",\"version\":\"1.0\",\"params\" : "
        "{\"RGBColor\" : {\"Red\":%d,\"Green\":%d,\"Blue\":%d},"
        "\"LightSwitch\" : %d},"
        "\"method\":\"thing.event.property.post\"}",
        ++pub_msg_cnt, rand_num_r, rand_num_g, rand_num_b, rgb_switch);
```

发布消息

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

```
memset(&topic_msg, 0x0, sizeof(iotx_mqtt_topic_info_t));
topic_msg.qos = IOTX_MQTT_QOS1;
topic_msg.retain = 0;
topic_msg.dup = 0;
topic_msg.payload = (void *)msg_pub;
topic_msg.payload_len = strlen(msg_pub);
rc = IOT_MQTT_Publish(pclient, ALINK_PROPERTY_POST_PUB, &topic_msg);
if (rc < 0) {
    IOT_MQTT_Destroy(&pclient);
    EXAMPLE_TRACE("error occur when publish");
    rc = -1;
    return rc;
}
```

33.5 运行

33.5.1 编译 & 下载

- **MDK**：双击 project.uvprojx 打开 MDK5 工程，执行编译。

- **IAR**：双击 project.eww 打开 IAR 工程，执行编译。

编译完成后，将开发板与 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
 / | \     4.0.1 build May  8 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/SAL_SKT] Socket Abstraction Layer initialize success.
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0
    x00000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offset
    @0x0000f0c8.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev   |   offset   |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | easyflash  | norflash    | 0x00000000 | 0x00100000 |
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash    | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash    | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash    | 0x00900000 | 0x00700000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x00000000
    , size is 4096 bytes.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number is 0
    xFDFEA040.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
[D/main] start to autoconnect ...
[D/main] The current version of APP firmware is 1.0.0
msh />[I/WLAN.mgnt] wifi connect success ssid:aptest
msh />
```

33.5.2 连接无线网络

如果没有连接或者网络相关信息被擦除，则需要在程序运行后会进入 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：

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

33.5.3 SHELL 命令

ali-iotkit 例程使用 MSH 命令演示与阿里云物联网平台的数据交互。例程使用了阿里云 LinkDevelop RGB 灯的示例，如果用户的 LinkDevelop 账户尚未创建 RGB 灯设备，请参阅《ali-iotkit 用户手册》例程使用章节完成 RGB 灯设备的创建。

命令列表

命令	说明
ali_mqtt_test start	启动 MQTT 示例
ali_mqtt_test pub open	开灯，并向云端同步开灯状态
ali_mqtt_test pub close	关灯，并向云端同步关灯状态
ali_mqtt_test stop	停止 MQTT 示例

1. 启动 MQTT

使用 `ali_mqtt_test start` 命令启动 MQTT 示例，成功后设备 log 显示订阅成功。

设备 log 如下所示：

```
msh />ali_mqtt_test start
[inf] iotx_device_info_init(40): device_info created successfully!
[dbg] iotx_device_info_set(50): start to set device info!
[dbg] iotx_device_info_set(64): device_info set successfully!
[dbg] guider_print_dev_guider_info(271):
    .................................................
[dbg] guider_print_dev_guider_info(272):           ProductKey : a1ClNfmK8VZ
[dbg] guider_print_dev_guider_info(273):           DeviceName : rgb_led_device
[dbg] guider_print_dev_guider_info(274):             DeviceID : a1ClNfmK8VZ.
    rgb_led_device
[dbg] guider_print_dev_guider_info(276):
    .................................................
[dbg] guider_print_dev_guider_info(277):        PartnerID Buf : ,partner_id=example.
    demo.partner-id  [dbg] guider_print_dev_guider_info(278):         ModuleID Buf :
    ,module_id=example.demo.module-id
[dbg] guider_print_dev_guider_info(279):           Guider URL :
[dbg] guider_print_dev_guider_info(281):       Guider SecMode : 3 (TCP + Direct +
    Plain)
[dbg] guider_print_dev_guider_info(283):     Guider Timestamp : 2524608000000
[dbg] guider_print_dev_guider_info(284):
    .................................................
```

```
[dbg] guider_print_dev_guider_info(290):
    ................................................
[dbg] guider_print_conn_info(248): ----------------------------------------
[dbg] guider_print_conn_info(249):        Host : a1ClNfmK8VZ.iot-as-mqtt.cn-shanghai
    .aliyuncs.com
[dbg] guider_print_conn_info(250):         Port : 1883
[dbg] guider_print_conn_info(253):         ClientID : a1ClNfmK8VZ.rgb_led_device|
    securemode=3,timestamp=2524608000000,signmethod=hma
csha1,gw=0,ext=0,partner_id=example.demo.partner-id,module_id=example.demo.module-id
    |
[dbg] guider_print_conn_info(258): ----------------------------------------
host: a1clnfmk8vz.iot-as-mqtt.cn-shanghai.aliyuncs.com
[inf] iotx_mc_init(1703): MQTT init success!
LINUXSOCK 83 HAL_TCP_Establish() | establish tcp connection with server(host=
    a1clnfmk8vz.iot-as-mqtt.cn-shanghai.aliyuncs.com msh />
LINUXSOCK 122 HAL_TCP_Establish() | success to establish tcp, fd=4
[inf] iotx_mc_connect(2035): mqtt connect success!
[dbg] iotx_mc_report_mid(2259): MID Report: started in MQTT
[dbg] iotx_mc_report_mid(2276): MID Report: json data = '{"id":"
    a1ClNfmK8VZ_rgb_led_device_mid","params":{"_sys_device_mid":"example
.demo.module-id","_sys_device_pid":"example.demo.partner-id"}}'
[dbg] iotx_mc_report_mid(2292): MID Report: topic name = '/sys/a1ClNfmK8VZ/
    rgb_led_device/thing/status/update'
[dbg] iotx_mc_report_mid(2309): MID Report: finished, IOT_MQTT_Publish() = 0
[inf] iotx_mc_subscribe(1388): mqtt subscribe success,topic = /sys/a1ClNfmK8VZ/
    rgb_led_device/thing/service/property/set!
[inf] iotx_mc_subscribe(1388): mqtt subscribe success,topic = /sys/a1ClNfmK8VZ/
    rgb_led_device/thing/event/property/post_reply!
[dbg] iotx_mc_cycle(1269): SUBACK
event_handle|139 :: subscribe success, packet-id=780
[dbg] iotx_mc_cycle(1269): SUBACK
event_handle|139 :: subscribe success, packet-id=8195
```

2. 设备发布消息

使用 **ali_mqtt_test pub open** 命令发送 LED 状态到云端。

设备 log 如下所示：

```
msh /> ali_mqtt_test_pub|571 ::
 publish message:
 topic: /sys/a1ClNfmK8VZ/rgb_led_device/thing/event/property/post
 payload: {"id" : "2","version":"1.0","p
msh />[dbg] iotx_mc_cycle(1260): PUBACK
event_handle|163 :: publish success, packet-id=1311
[dbg] iotx_mc_cycle(1277): PUBLISH
[dbg] iotx_mc_handle_recv_PUBLISH(1091):         Packet Ident : 00000000
[dbg] iotx_mc_handle_recv_PUBLISH(1092):         Topic Length : 63
[dbg] iotx_mc_handle_recv_PUBLISH(1093):           Topic Name : /sys/a1ClNfmK8VZ/
    rgb_led_device/thing/event/property/post_reply
```

```
[dbg] iotx_mc_handle_recv_PUBLISH(1097):     Payload Len/Room : 188 / 956
[dbg] iotx_mc_handle_recv_PUBLISH(1100):       Receive Buflen : 1024
[dbg] iotx_mc_handle_recv_PUBLISH(1111): delivering msg ...
[dbg] iotx_mc_deliver_message(866): topic be matched
_demo_message_arrive|197 :: ----
_demo_message_arrive|198 :: packetId: 0
_demo_message_arrive|199 :: Topic: '/sys/a1ClNfmK8VZ/rgb_led_device/thing/event/
    property/post_reply' (Length: 63)
_demo_message_arrive|203 :: Payload: '{"code":200,"data":{"LightSwitch":"tsl parse:
    params not exist","RGBColor":"tsl parse: params
not exist"},"id":"2","
_demo_message_arrive|207 :: ----
[inf] iotx_mc_keepalive_sub(2226): send MQTT ping...
[inf] iotx_mc_cycle(1295): receive
```

3. 云端查看发布的消息

在设备详情里的**运行状态**里可以查看设备的上报到云端的消息内容，一次有设备激活、设备上线与设备数据上报。

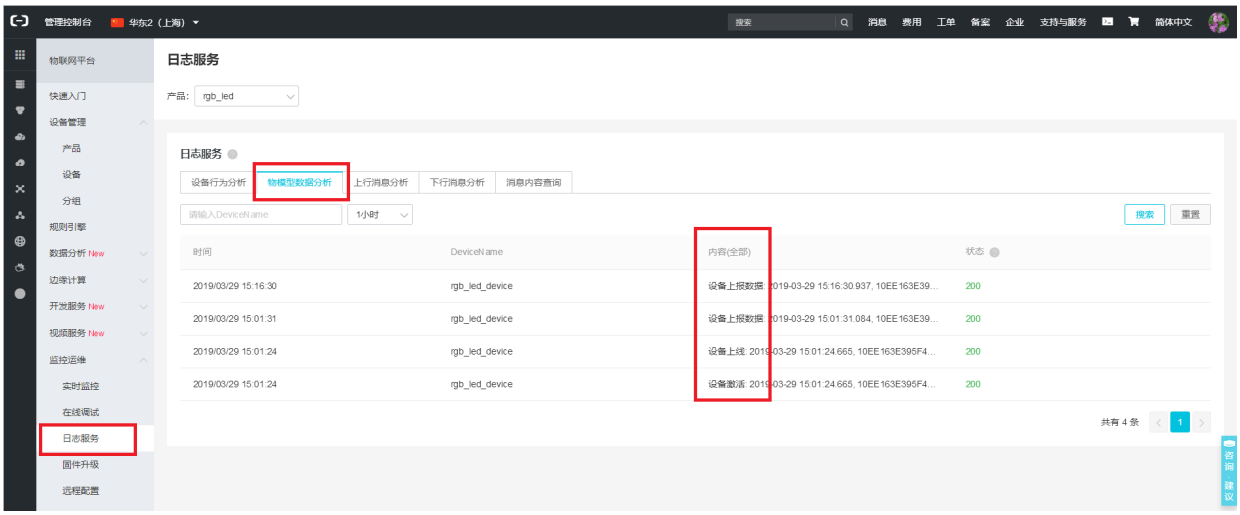


图 33.2: *日志服务*

鼠标移动至数据内容中，在数据上报中自动显示上报的数据内容。

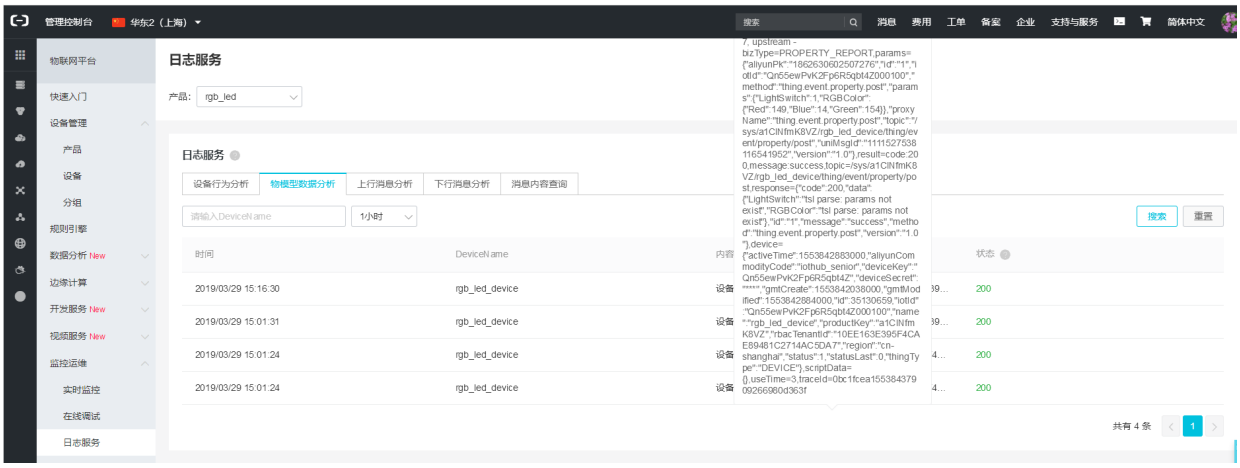


图 33.3: *查看设备接收数据*

4. 云端推送消息到设备

使用云端的在线调试给设备推送消息，如下图所示

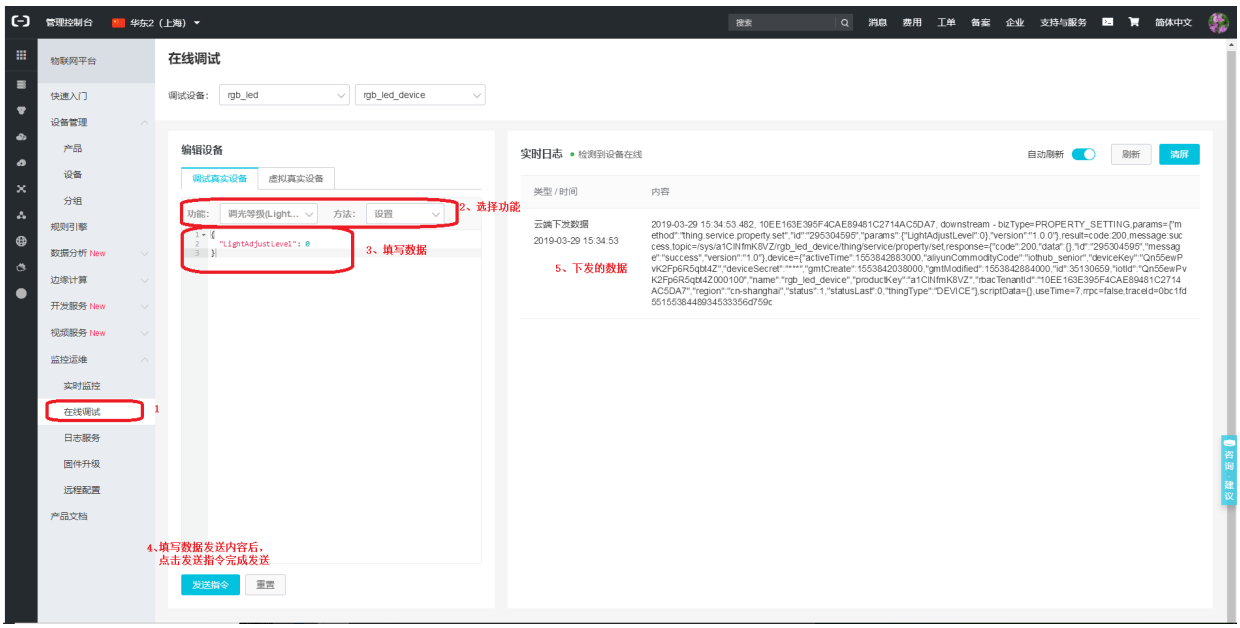


图 33.4: *查看设备接收数据*

5. 查看设备订阅日志

使用调试控制台发送命令后，设备可以接受到命令，log 如下所示：

```
[dbg] iotx_mc_handle_recv_PUBLISH(1091):          Packet Ident : 00000000
[dbg] iotx_mc_handle_recv_PUBLISH(1092):          Topic Length : 52
[dbg] iotx_mc_handle_recv_PUBLISH(1096):            Topic Name : /sys/a1Ayv8xhoIl/RGB
    -DEV1/thing/service/property/set
[dbg] iotx_mc_handle_recv_PUBLISH(1099):      Payload Len/Room : 100 / 967
[dbg] iotx_mc_handle_recv_PUBLISH(1100):         Receive Buflen : 1024
[dbg] iotx_mc_handle_recv_PUBLISH(1111): delivering msg ...
[dbg] iotx_mc_deliver_message(866): topic be matched
_demo_message_arrive|178 :: ----
```

```
_demo_message_arrive|179 :: packetId: 0
_demo_message_arrive|183 :: Topic: '/sys/a1Ayv8xhoIl/RGB-DEV1/thing/service/property
    /set' (Length: 52)
_demo_message_arrive|187 :: Payload:
'{"method":"thing.service.property.set","id":"35974024","params":{"LightSwitch":0},"
    version":"1.0.0"}' (Length: 100)
_demo_message_arrive|188 :: ----
```

6. 退出 MQTT 示例

使用 **ali_mqtt_test stop** 命令退出 MQTT 示例，设备 log 如下所示：

```
msh />ali_mqtt_test stop
[inf] iotx_mc_unsubscribe(1423): mqtt unsubscribe success,topic = /sys/a1HETlEuvri/
    RGB-LED-DEV-1/thing/event/property/post_reply!
[inf] iotx_mc_unsubscribe(1423): mqtt unsubscribe success,topic = /sys/a1HETlEuvri/
    RGB-LED-DEV-1/thing/service/property/set!
event_handle|136 :: unsubscribe success, packet-id=0
event_handle|136 :: unsubscribe success, packet-id=0
[dbg] iotx_mc_disconnect(2121): rc = MQTTDisconnect() = 0
[inf] _network_ssl_disconnect(514): ssl_disconnect
[inf] iotx_mc_disconnect(2129): mqtt disconnect!
[inf] iotx_mc_release(2175): mqtt release!
[err] LITE_dump_malloc_free_stats(594): WITH_MEM_STATS = 0
mqtt_client|329 :: out of sample!
```

33.6 注意事项

- 使用本例程前请先阅读《ali-iotkit 用户手册》
- 使用前请在 menuconfig 里配置自己的设备激活凭证（PRODUCT_KEY、DEVICE_NAME 和 DEVICE_SECRET）
- 使用 menuconfig 配置选择要接入的平台（**LinkDevelop** 或者 **LinkPlatform**）

33.7 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf
- 《ali-iotkit 用户手册》：docs/UM1002-RT-Thread-ali-iotkit 用户手册.pdf

第 34 章

微软 Azure 物联网平台接入例程

本例程演示如何使用 RT-Thread 提供的 azure-iot-sdk 软件包接入微软物联网平台。初次使用微软物联网平台或者想了解更多使用例程，请阅读《Azure 云平台软件包用户手册》。

34.1 简介

Azure 是 RT-Thread 移植的用于连接微软 Azure IoT 中心的软件包，原始 SDK 为：**azure-iot-sdk-c**。通过该软件包，可以让运行 RT-Thread 的设备轻松接入 Azure IoT 中心。

Azure IoT 中心运行在微软云服务器上，充当中央消息中心，用于 IoT 应用程序与其管理的设备之间的双向通信。通过 Azure IoT 中心，可以在数百万 IoT 设备和云托管解决方案后端之间建立可靠又安全的通信，生成 IoT 解决方案。几乎可以将任何设备连接到 IoT 中心。

使用 Azure 软件包连接 IoT 中心可以实现如下功能：

- 轻松连入 Azure IoT 中心，建立与 Azure IoT 的可靠通讯
- 为每个连接的设备设置标识和凭据，并帮助保持云到设备和设备到云消息的保密性
- 管理员可在云端大规模地远程维护、更新和管理 IoT 设备
- 从设备大规模接收遥测数据
- 将数据从设备路由到流事件处理器
- 设备上传文件到 IoT 中心
- 将云到设备的消息发送到特定设备

可以使用 Azure IoT 中心来实现自己的解决方案后端。此外，IoT 中心还包含标识注册表，可用于预配设备、其安全凭据以及其连接到 IoT 中心的权限。

34.2 硬件说明

Azure 例程需要依赖 IoT Board 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。

34.3 软件说明

azure 例程位于 `examples/34_iot_cloud_ms_azure` 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
ports	移植文件
packages/azure-iot-sdk-v1.2.8	azure 物联网平台接入软件包
packages/azure-iot-sdk-v1.2.8/samples	azure 物联网平台接入示例

34.3.1 准备工作

在运行本次示例程序之前需要准备工作如下：

1、注册微软 Azure 账户，如果没有 Azure 订阅，请在开始前创建一个试用帐户。

2、安装 DeviceExplorer 工具，这是一个 windows 平台下测试 Azure 软件包功能必不可少的工具。该工具的安装包为 packages/azure-iot-sdk-v1.2.8/tools 目录下的 SetupDeviceExplorer.msi，按照提示安装即可，成功运行后的界面如下图。

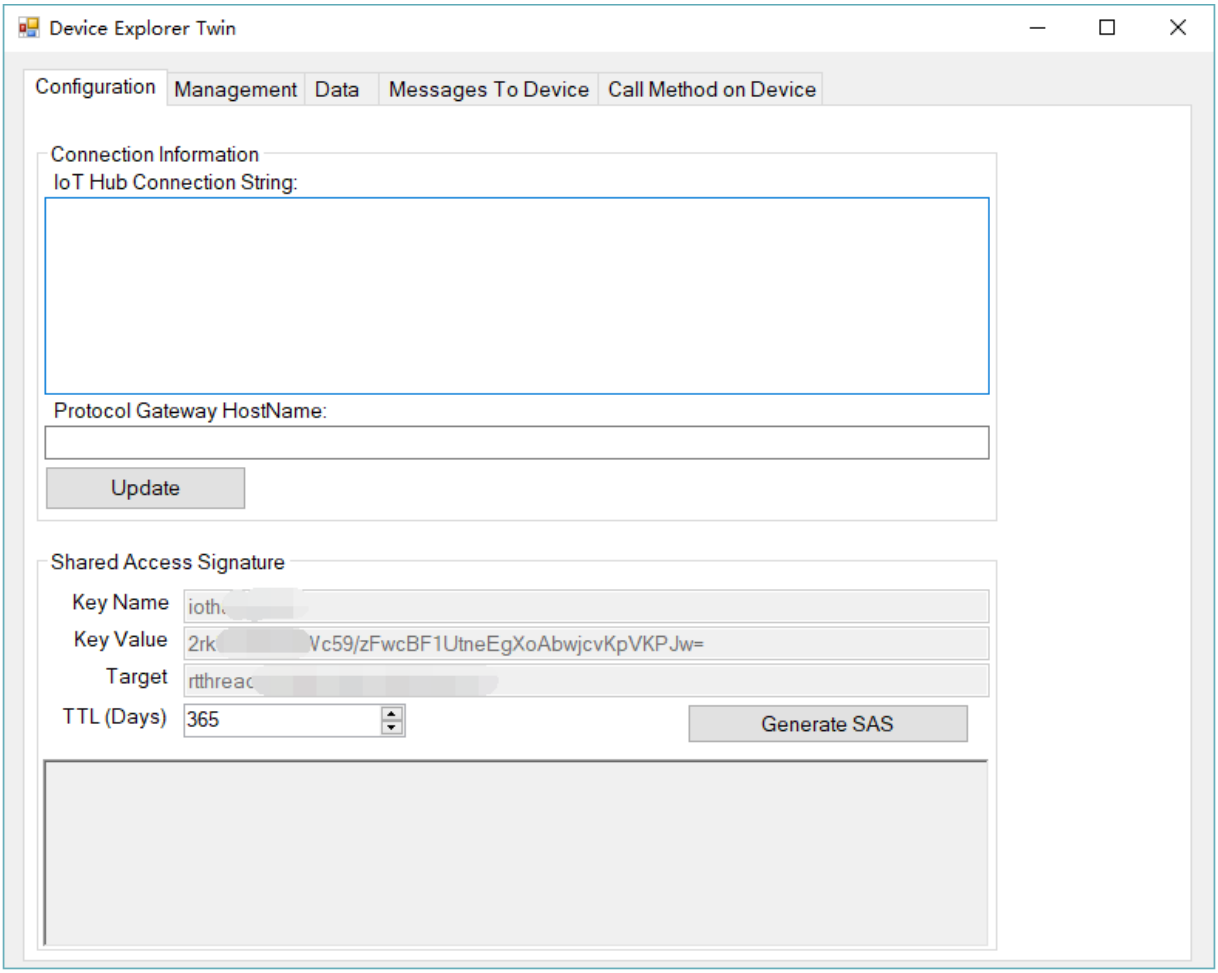


图 34.1: *DeviceExplorer 工具界面*

34.3.1.1 通信协议介绍

目前 RT-Thread Azure 软件包提供的示例代码支持 MQTT 和 HTTP 的通信协议，想要使用哪种协议，只需要在上面选项中选择相应的协议即可。在选择设备端通信协议时，需要注意以下几点：

1、当进行云到设备数据发送时，由于 HTTPS 没有用于实现服务器推送的有效方法。因此，使用 HTTPS 协议时，设备会在 IoT 中心轮询从云到设备的消息。此方法对于设备和 IoT 中心而言是低效的。根据当前 HTTPS 准则，每台设备应每 25 分钟或更长时间轮询一次消息。MQTT 支持在收到云到设备的消息时进行服务器推送。它们会启用从 IoT 中心到设备的直接消息推送。如果传送延迟是考虑因素，最好使用 MQTT 协议。对于很少连接的设备，HTTPS 也适用。

2、使用 HTTPS 时，每台设备应每 25 分钟或更长时间轮询一次从云到设备的消息。但在开发期间，可按低于 25 分钟的更高频率进行轮询。

3、更详细的协议选择文档请参考 Azure 官方文档《选择通信协议》。

34.3.1.2 创建 IoT 中心

首先要做的是使用 Azure 门户在订阅中创建 IoT 中心。IoT 中心用于将大量遥测数据从许多设备引入到云中。然后，该中心会允许一个或多个在云中运行的后端服务读取和处理该遥测数据。

1、登录到 Azure 门户。

2、选择 “创建资源” > “物联网” > “IoT 中心”。

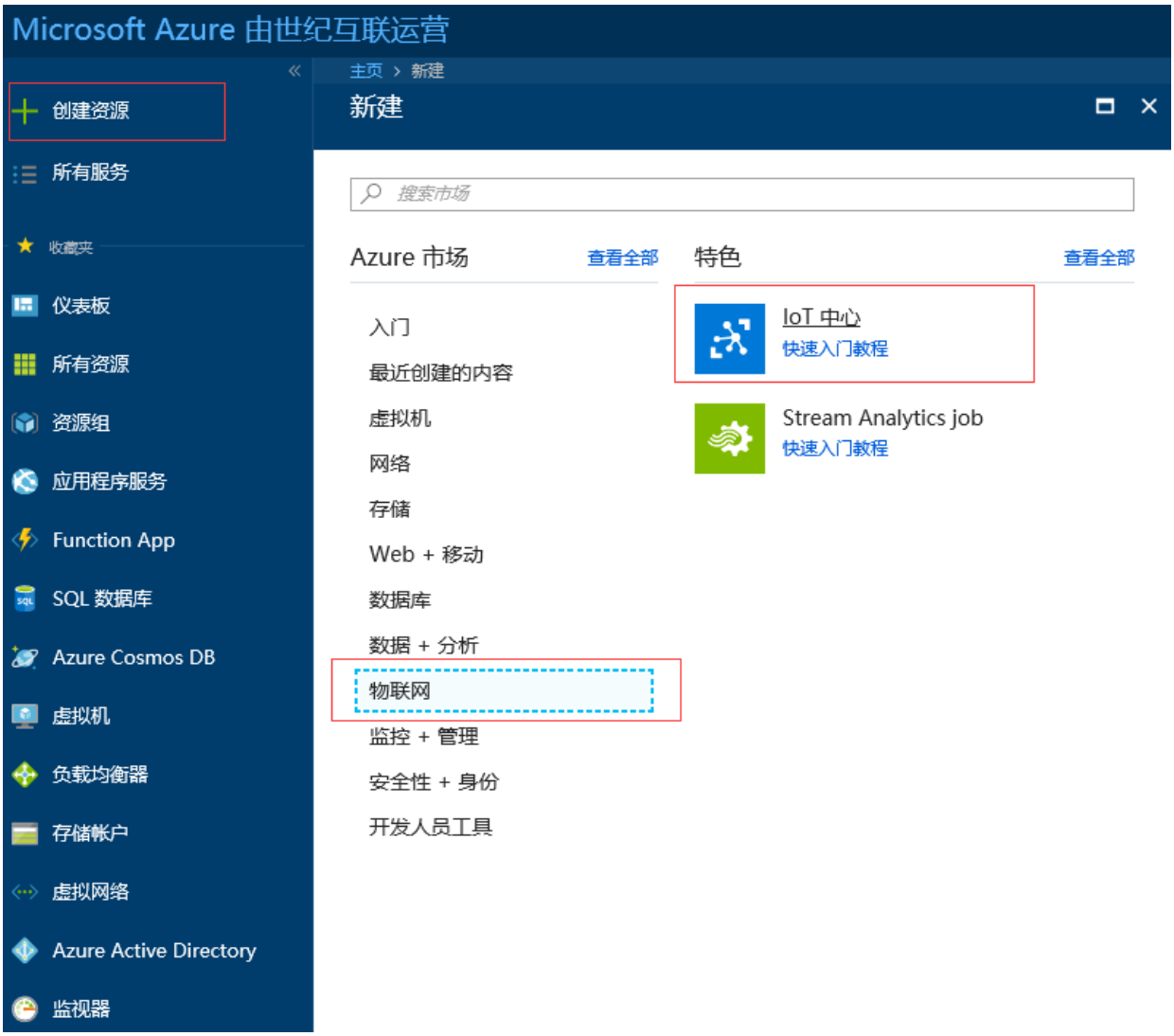


图 34.2: *创建物联网中心*

3、在 “IoT 中心” 窗格中，输入 IoT 中心的以下信息：

- **Subscription（订阅）**：选择需要将其用于创建此 IoT 中心的订阅。
- **Resource Group（资源组）**：创建用于托管 IoT 中心的资源组，或使用现有的资源组，在这个栏目中填入一个合适的名字就可以了。有关详细信息，请参阅使用资源组管理 Azure 资源。
- **Region（区域）**：选择最近的位置。
- **IoT Hub Name（物联网中心名称）**：创建 IoT 中心的名称，这个名称需要是唯一的。如果输入的名称可用，会显示一个绿色复选标记。

图 34.3: *填写 IoT 中心资料*

4、选择“下一步: **Size and scale（大小和规模）**”，以便继续创建 IoT 中心。

5、选择“**Pricing and scale tier（定价和缩放层）**”。就测试用途来说，请选择“**F1:Free tier（F1 - 免费）**”层（前提是此层在订阅上仍然可用）。有关详细信息，请参阅定价和缩放层。

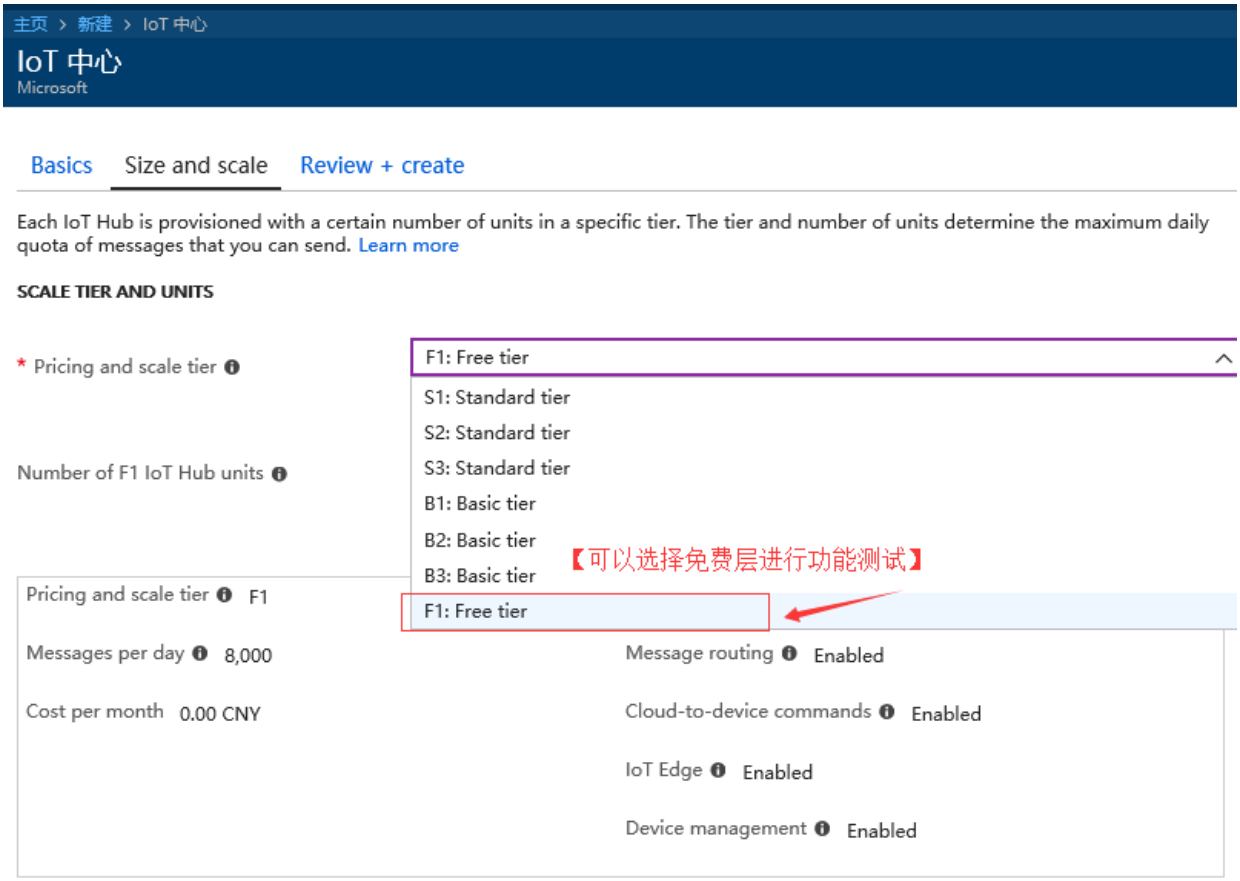


图 34.4: *选择功能层*

6、选择 “**Review + create（查看 + 创建）**”。

7、查看 IoT 中心信息，然后单击 “创建” 即可。创建 IoT 中心可能需要数分钟的时间。可在 “**通知**” 窗格中监视进度，创建成功后就可以进行下一步注册设备的操作了。

8、为了后续方便查找，可以手动将创建成功后的资源添加到**仪表盘**。

34.3.1.3 注册设备

要想运行设备端相关的示例，需要先将设备信息注册到 IoT 中心里，然后该设备才能连接到 IoT 中心。在本次示例中，可以使用 DeviceExplorer 工具来注册设备。

- 获得 IoT 中心的**共享访问密钥**（即 IoT 中心连接字符串）

1、IoT 中心创建完毕后，在设置栏目中，点击共享访问策略选项，可以打开 IoT 中心的访问权限设置。打开 iothubowner，在右侧弹出的属性框中获得 IoT 中心的共享访问密钥。

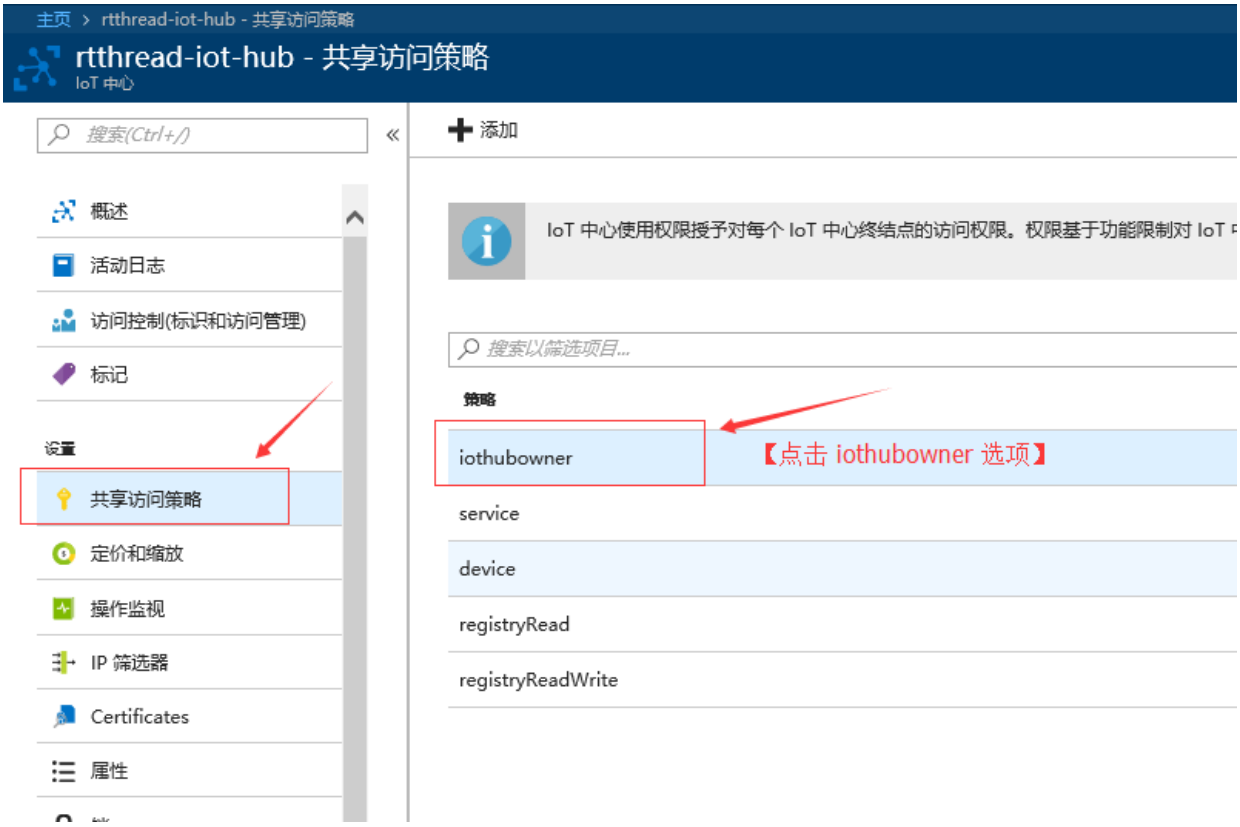


图 34.5: *查看物联网中心共享访问策略*

2、在右侧弹出的属性框中获取 IoT 中心连接字符串：

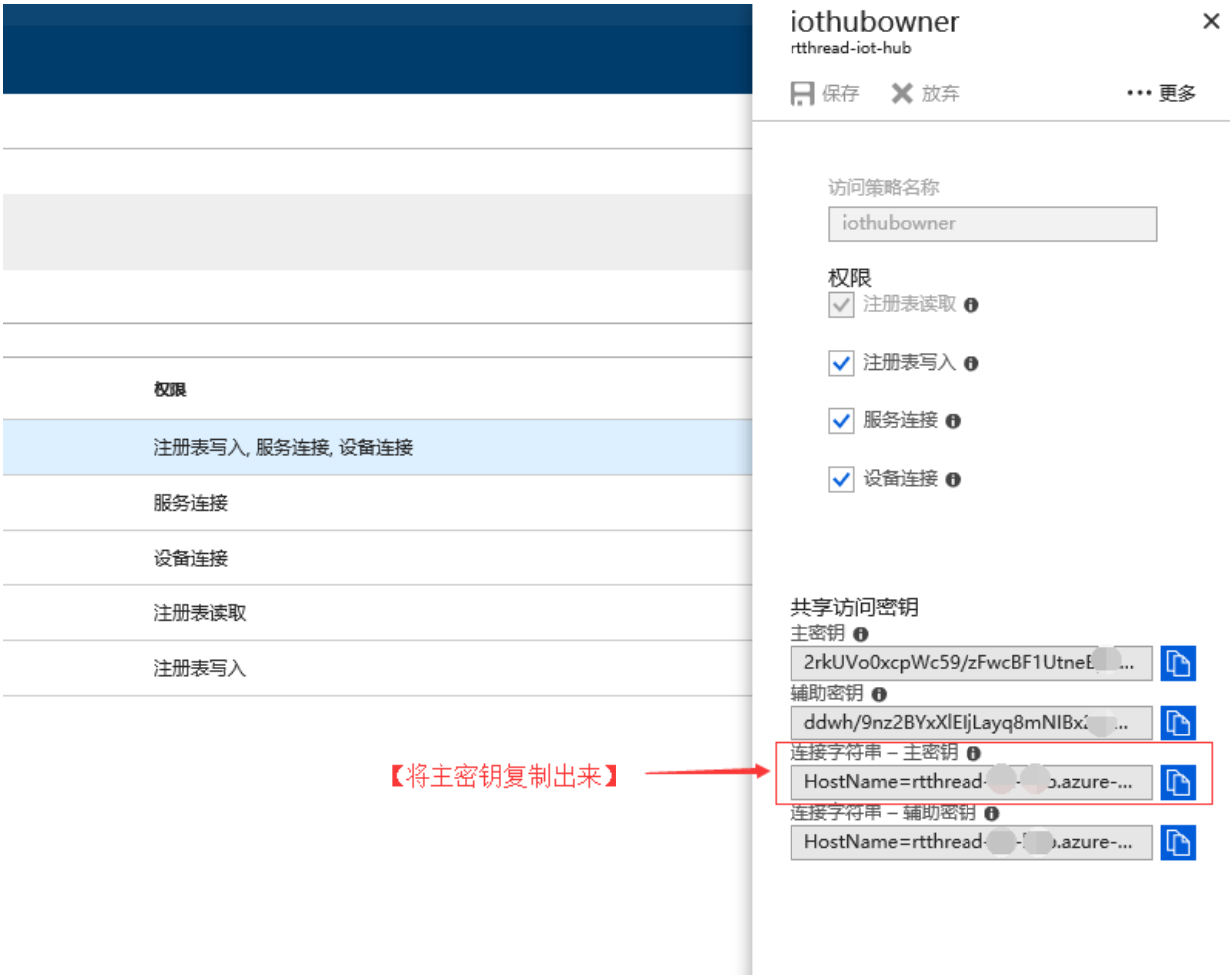


图 34.6: *复制访问主密钥*

- 创建设备

1、有了连接字符串后，我们便可以使用 DeviceExplorer 工具来创建设备，并测试 IoT 中心的功能了。打开测试工具，在配置选项中填入的连接字符串。点击 `update` 按钮更新本地连接 IoT 中心的配置，为下一步创建测试设备做准备。

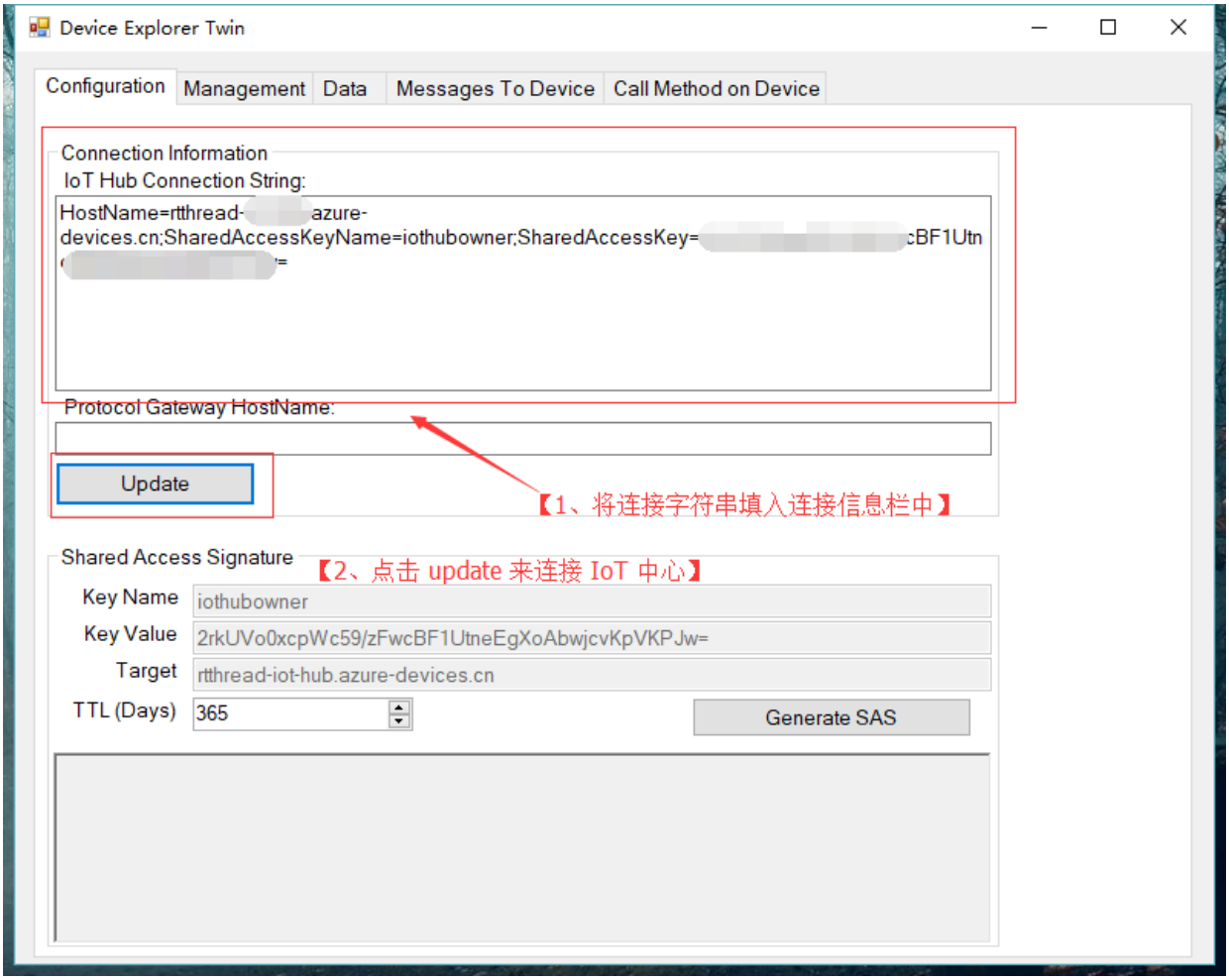


图 34.7: *配置 DeviceExplorer 工具*

2、打开 Management 选项栏，按照下图所示的步骤来创建测试设备。设备创建成功后，就可以运行设备的功能示例了。

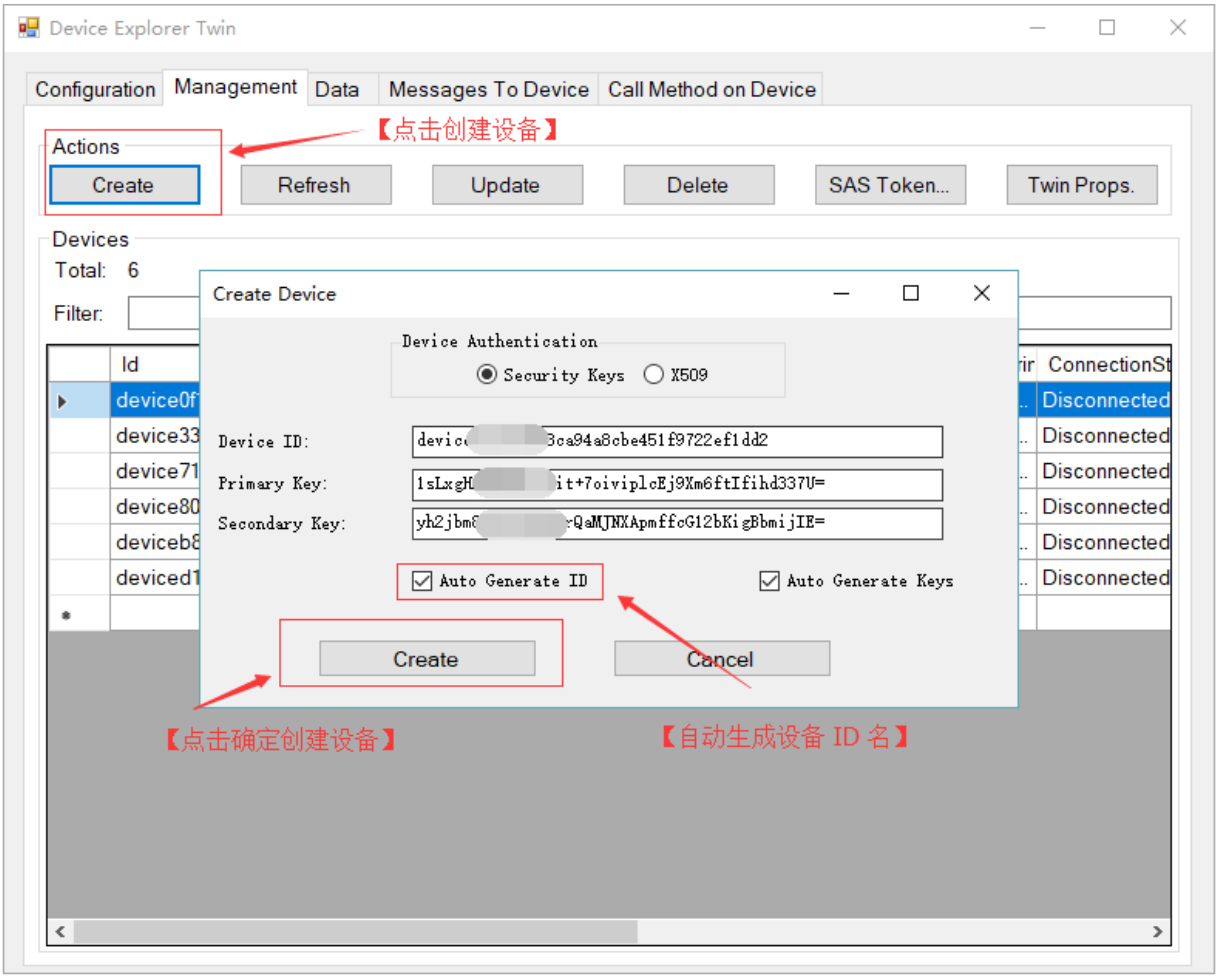

Device Explorer Twin
Configuration
Management
Data
Messages To Device
Call Method on Device
Actions
【点击创建设备】
Create
Refresh
Update
Delete
SAS Token...
Twin Props.
Devices
Total: 6
Filter:
Id
ConnectionSt
device0f
device33
device71
device80
deviceb8
deviced1
Disconnected
Create Device
Device Authentication
Security Keys
X509
Device ID:
Primary Key:
Secondary Key:
Auto Generate ID
Auto Generate Keys
Create
Cancel
【点击确定创建设备】
【自动生成设备 ID 名】

图 34.8: *创建设备*

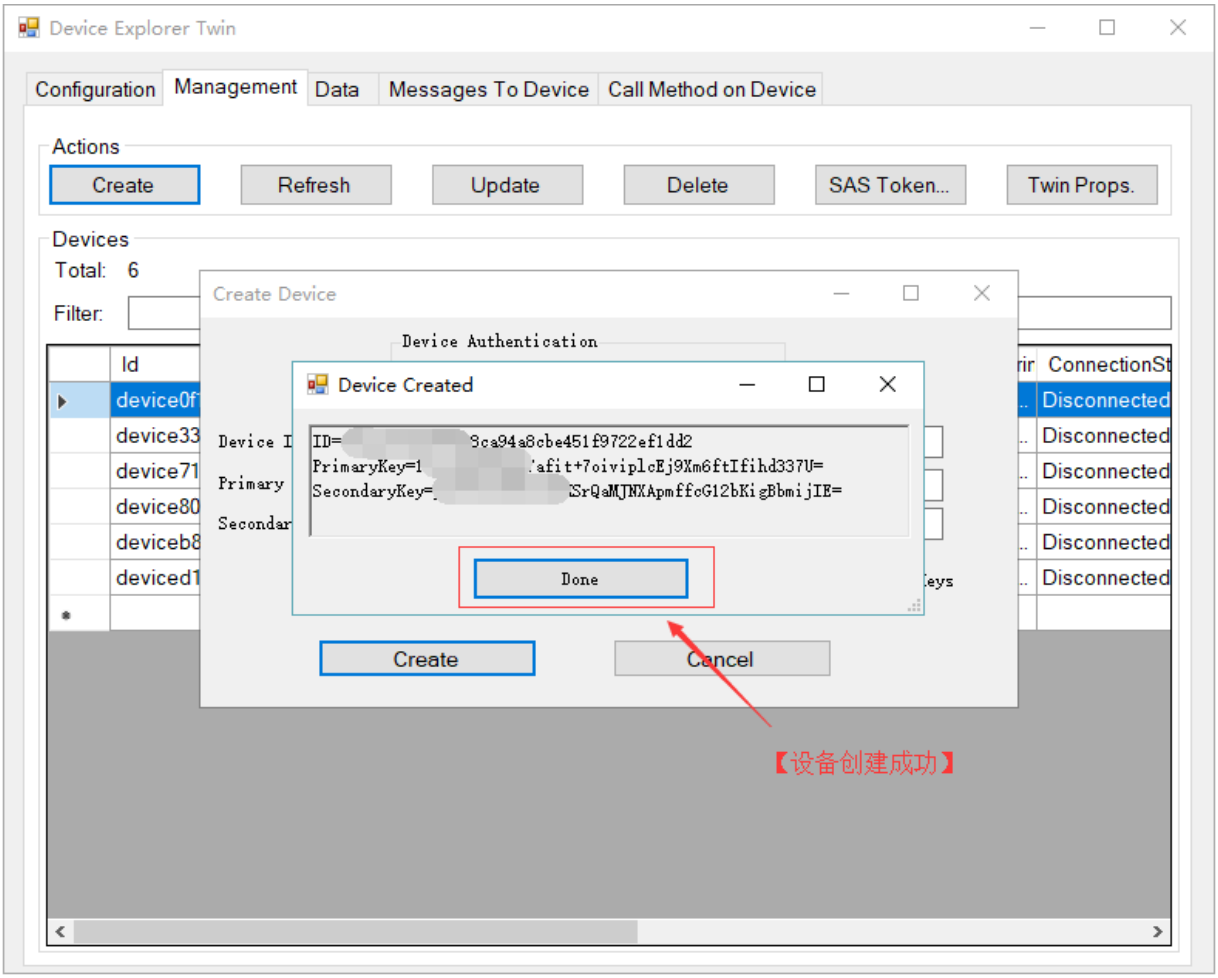


图 34.9: *创建设备成功*

34.4 运行

34.4.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。
- **IAR**：双击 `project.eww` 打开 IAR 工程，执行编译。

编译完成后，将开发板与 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
 / | \     4.0.1 build May 30 2019
 2006 - 2019 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 16777216 bytes.
[SFUD] w25q128 flash device is initialize success.
[I/sal.skt] Socket Abstraction Layer initialize success.
[E/main] 1005
```

```
[D/FAL] (fal_flash_init:61) Flash device |              w60x_onchip | addr: 0
    x08000000 | len: 0x00100000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_flash_init:61) Flash device |                 norflash | addr: 0
    x00000000 | len: 0x01000000 | blk_size: 0x00001000 |initialized finish.
[D/FAL] (fal_partition_init:176) Find the partition table on 'w60x_onchip' offset
    @0x0000f0c8.
[I/FAL] ==================== FAL partition table ====================
[I/FAL] | name       | flash_dev    |   offset    |    length   |
[I/FAL] -------------------------------------------------------------
[I/FAL] | easyflash  | norflash     | 0x00000000 | 0x00100000 |
[I/FAL] | app        | w60x_onchip | 0x00010100 | 0x000ed800 |
[I/FAL] | download   | norflash     | 0x00100000 | 0x00100000 |
[I/FAL] | font       | norflash     | 0x00200000 | 0x00700000 |
[I/FAL] | filesystem | norflash     | 0x00900000 | 0x00700000 |
[I/FAL] =============================================================
[I/FAL] RT-Thread Flash Abstraction Layer (V0.3.0) initialize success.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:152) ENV start address is 0x00000000
    , size is 4096 bytes.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:821) Calculate ENV CRC32 number is 0
    x12D2A372.
[Flash] (packages\EasyFlash-v3.3.0\src\ef_env.c:833) Verify ENV CRC32 result is OK.
[Flash] EasyFlash V3.3.0 is initialize success.
[Flash] You can get the latest version on https://github.com/armink/EasyFlash .
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />[I/WLAN.mgnt] wifi connect success ssid:aptest
[I/WLAN.lwip] Got IP address : 192.168.12.26     # 成功自动连接wifi
```

34.4.2 连接无线网络

如果没有连接网络，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 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
```

34.4.3 运行效果

34.4.3.1 功能示例一：设备发送遥测数据到物联网中心

示例文件

示例程序路径	说明
samples/iothub_ll_telemetry_sample.c	从设备端发送遥测数据到 Azure IoT 中心

云端监听设备数据

- 打开测试工具的 Data 选项栏，选择需要监听的设备，开始监听：

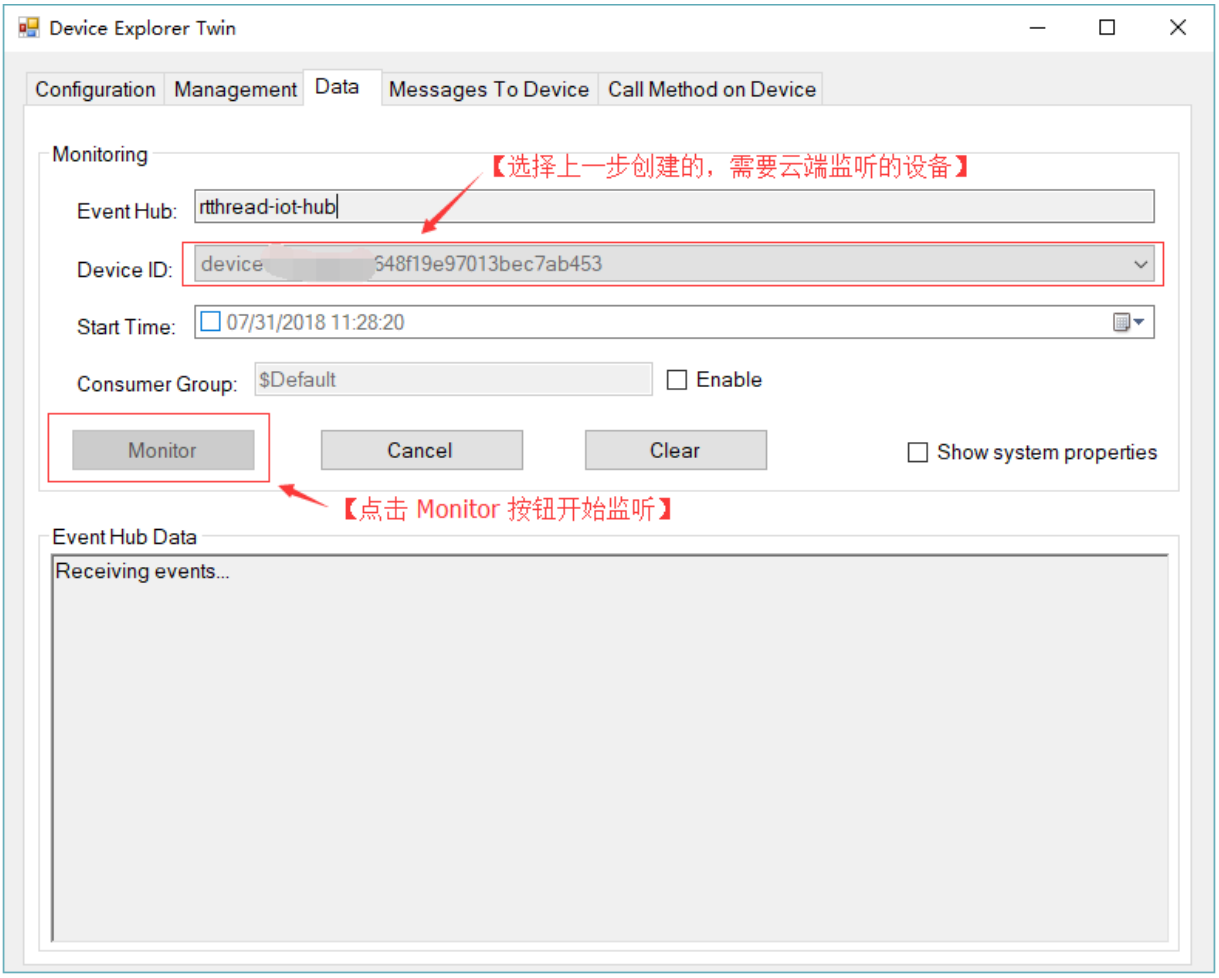


图 34.10: *监听设备遥测数据*

修改示例代码中的设备连接字符串

1、在运行测试示例前需要获取设备的连接字符串。

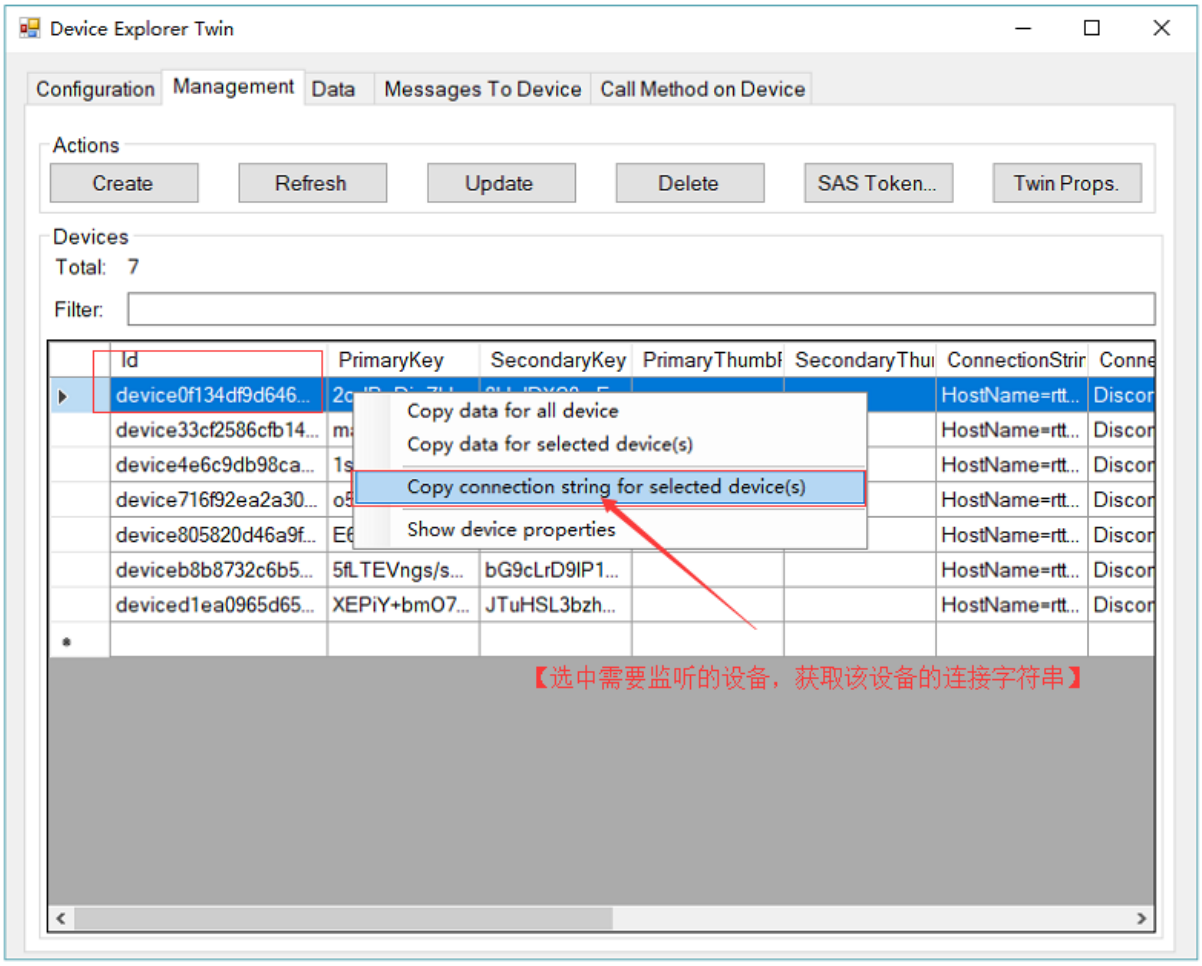


图 34.11: *获取设备连接字符串*

2、将连接字符串填入测试示例中的 connectionString 字符串中，重新编译程序，下载到开发板中。

```
iothub_ll_telemetry_sample.c
40 #ifdef SAMPLE_MQTT
41     #include "iothubtransportmqtt.h"
42 #endif // SAMPLE_MQTT
43 #ifdef SAMPLE_MQTT_OVER_WEBSOCKETS
44     #include "iothubtransportmqtt_websockets.h"
45 #endif // SAMPLE_MQTT_OVER_WEBSOCKETS
46 #ifdef SAMPLE_AMQP
47     #include "iothubtransportamqp.h"
48 #endif // SAMPLE_AMQP
49 #ifdef SAMPLE_AMQP_OVER_WEBSOCKETS
50     #include "iothubtransportamqp_websockets.h"
51 #endif // SAMPLE_AMQP_OVER_WEBSOCKETS
52 #ifdef SAMPLE_HTTP
53     #include "iothubtransporthttp.h"
54 #endif // SAMPLE_HTTP
55
56 /* Paste in the your iothub connection string  */
57 static const char* connectionString ="HostName=rtthread-        azure-devices.cn;DeviceId=device
58 #define MESSAGE_COUNT        5
59 static bool g_continueRunning = true;
60 static size_t g_message_count_send_confirmations = 0;
61
62 static void send_confirm_callback(IOTHUB_CLIENT_CONFIRMATION_RESULT result, void* userContextCal
63 {
```

【将连接字符串修改为上一步获取的设备连接字符串】

图 34.12: *填入设备连接字符串*

运行示例程序

1、在 msh 中运行 azure_telemetry_sample 示例程序：

```
msh />azure_telemetry_sample
msh />
ntp init
Creating IoTHub Device handle
Sending message 1 to IoTHub
-> 11:46:58 CONNECT | VER: 4 | KEEPALIVE: 240 | FLAGS: 192 |
USERNAME:
xxxxxxxxxx.azuredevices.cn/devicexxxxxxx9d64648f19e97013bec7ab453
/?api-version=2017-xx-xx-preview&
DeviceClientType=iothubclient%2f1.2.8%20
(native%3b%20xxxxxxxx%3b%20xxxxxx) | PWD: XXXX | CLEAN: 0
<- 11:46:59 CONNACK | SESSION_PRESENT: true | RETURN_CODE: 0x0
The device client is connected to iothub
Sending message 2 to IoTHub
Sending message 3 to IoTHub
-> 11:47:03 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE |
    TOPIC_NAME: devices/device0f134df9d64648f19e97013bec7ab453/messages/events
    /hello=RT-Thread | PACKET_ID: 2 | PAYLOAD_LEN: 12
-> 11:47:03 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE |
    TOPIC_NAME: devices/device0f134df9d64648f19e97013bec7ab453/messages/events
    /hello=RT-Thread | PACKET_ID: 3 | PAYLOAD_LEN: 12
-> 11:47:03 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE |
    TOPIC_NAME: devices/device0f134df9d64648f19e97013bec7ab453/messages/events
    /hello=RT-Thread | PACKET_ID: 4 | PAYLOAD_LEN: 12
<- 11:47:04 PUBACK | PACKET_ID: 2
Confirmation callback received for message 1 with result
    IOTHUB_CLIENT_CONFIRMATION_OK
<- 11:47:04 PUBACK | PACKET_ID: 3
Confirmation callback received for message 2 with result
    IOTHUB_CLIENT_CONFIRMATION_OK
<- 11:47:04 PUBACK | PACKET_ID: 4
Confirmation callback received for message 3 with result
    IOTHUB_CLIENT_CONFIRMATION_OK
Sending message 4 to IoTHub
-> 11:47:06 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE |
    TOPIC_NAME: devices/device0f134df9d64648f19e97013bec7ab453/messages/events
    /hello=RT-Thread | PACKET_ID: 5 | PAYLOAD_LEN: 12
<- 11:47:07 PUBACK | PACKET_ID: 5
Confirmation callback received for message 4 with result
    IOTHUB_CLIENT_CONFIRMATION_OK
Sending message 5 to IoTHub
-> 11:47:09 PUBLISH | IS_DUP: false | RETAIN: 0 | QOS: DELIVER_AT_LEAST_ONCE |
    TOPIC_NAME: devices/device0f134df9d64648f19e97013bec7ab453/messages/events
    /hello=RT-Thread | PACKET_ID: 6 | PAYLOAD_LEN: 12
<- 11:47:10 PUBACK | PACKET_ID: 6
Confirmation callback received for message 5 with result
```

```
    IOTHUB_CLIENT_CONFIRMATION_OK
-> 11:47:14 DISCONNECT
Error: Time:Tue Jul 31 11:47:14 2018 File:packages\azure\azure-port\pal\src\
socketio_berkeley.c Func:socketio_send Line:853
Failure: socket state is not opened.
The device client has been disconnected
Azure Sample Exit
```

2、此时可在 DeviceExplorer 工具的 Data 栏查看设备发到云端的遥测数据：

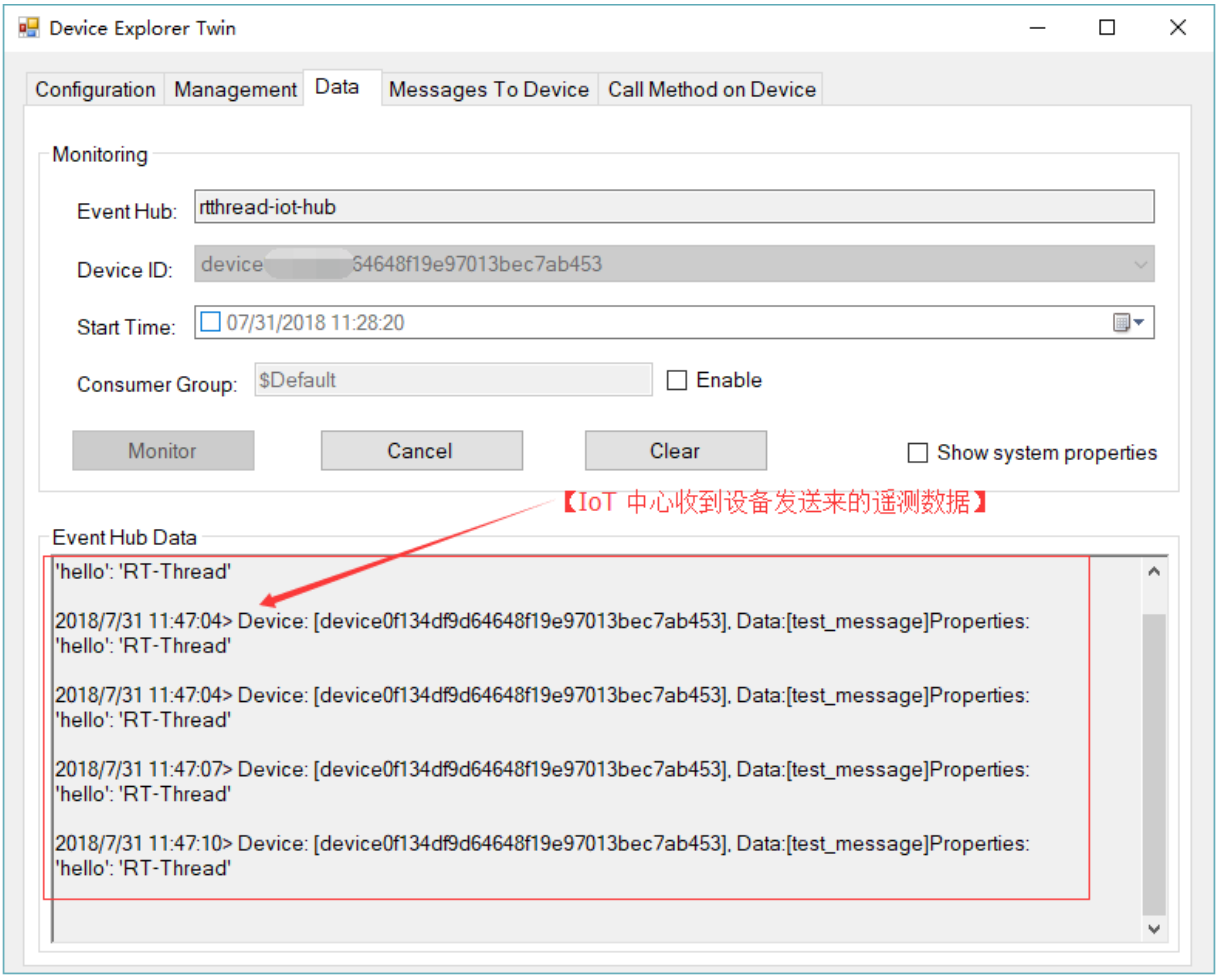


图 34.13: *收到遥测数据*

示例运行成功，在 DeviceExplorer 工具中看到了设备发送到物联网中心的 5 条遥测数据。

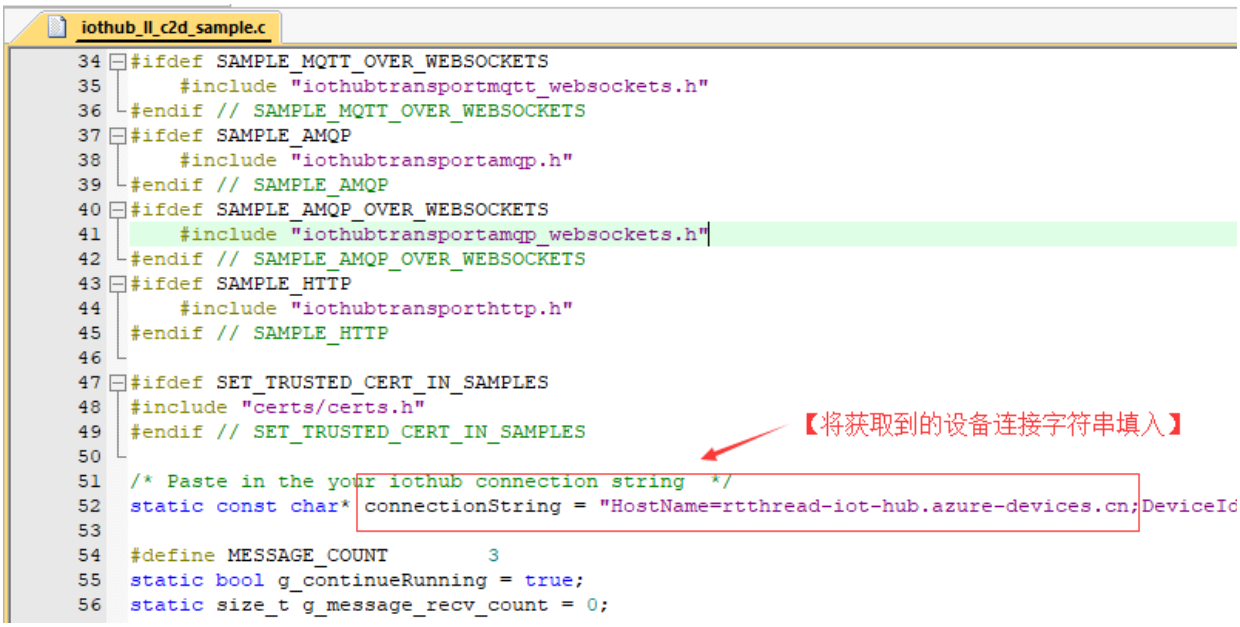
34.4.3.2 功能示例二：设备监听云端下发的数据

示例文件

示例程序路径	说明
samples/iothub_ll_c2d_sample.c	在设备端监听 Azure IoT 中心下发的数据

修改示例代码中的设备连接字符串

- 与上面的示例相同，本示例程序也需要填写正确的设备连接字符串，修改完毕后重新编译程序，下载到开发板中即可。修改内容如下所示：




```
34 #ifdef SAMPLE_MQTT_OVER_WEBSOCKETS
35     #include "iothubtransportmqtt_websockets.h"
36 #endif // SAMPLE_MQTT_OVER_WEBSOCKETS
37 #ifdef SAMPLE_AMQP
38     #include "iothubtransportamqp.h"
39 #endif // SAMPLE_AMQP
40 #ifdef SAMPLE_AMQP_OVER_WEBSOCKETS
41     #include "iothubtransportamqp_websockets.h"
42 #endif // SAMPLE_AMQP_OVER_WEBSOCKETS
43 #ifdef SAMPLE_HTTP
44     #include "iothubtransporthttp.h"
45 #endif // SAMPLE_HTTP
46
47 #ifdef SET_TRUSTED_CERT_IN_SAMPLES
48 #include "certs/certs.h"
49 #endif // SET_TRUSTED_CERT_IN_SAMPLES
50
51 /* Paste in the your iothub connection string  */
52 static const char* connectionString = "HostName=rtthread-iot-hub.azure-devices.cn;DeviceId
53
54 #define MESSAGE_COUNT        3
55 static bool g_continueRunning = true;
56 static size_t g_message_recv_count = 0;
```

图 34.14: *修改设备连接字符串*

设备端运行示例程序

- 在 msh 中运行 azure_c2d_sample 示例程序，示例程序运行后设备将会等待并接收云端下发的数据：

```
msh />azure_c2d_sample
msh />
ntp init
Creating IoTHub Device handle          # 等待 IoT 中心的下发数据
Waiting for message to be sent to device (will quit after 3 messages)
```

服务器下发数据给设备

1、打开 DeviceExplorer 工具的 Messages To Device 栏向指定设备发送数据：

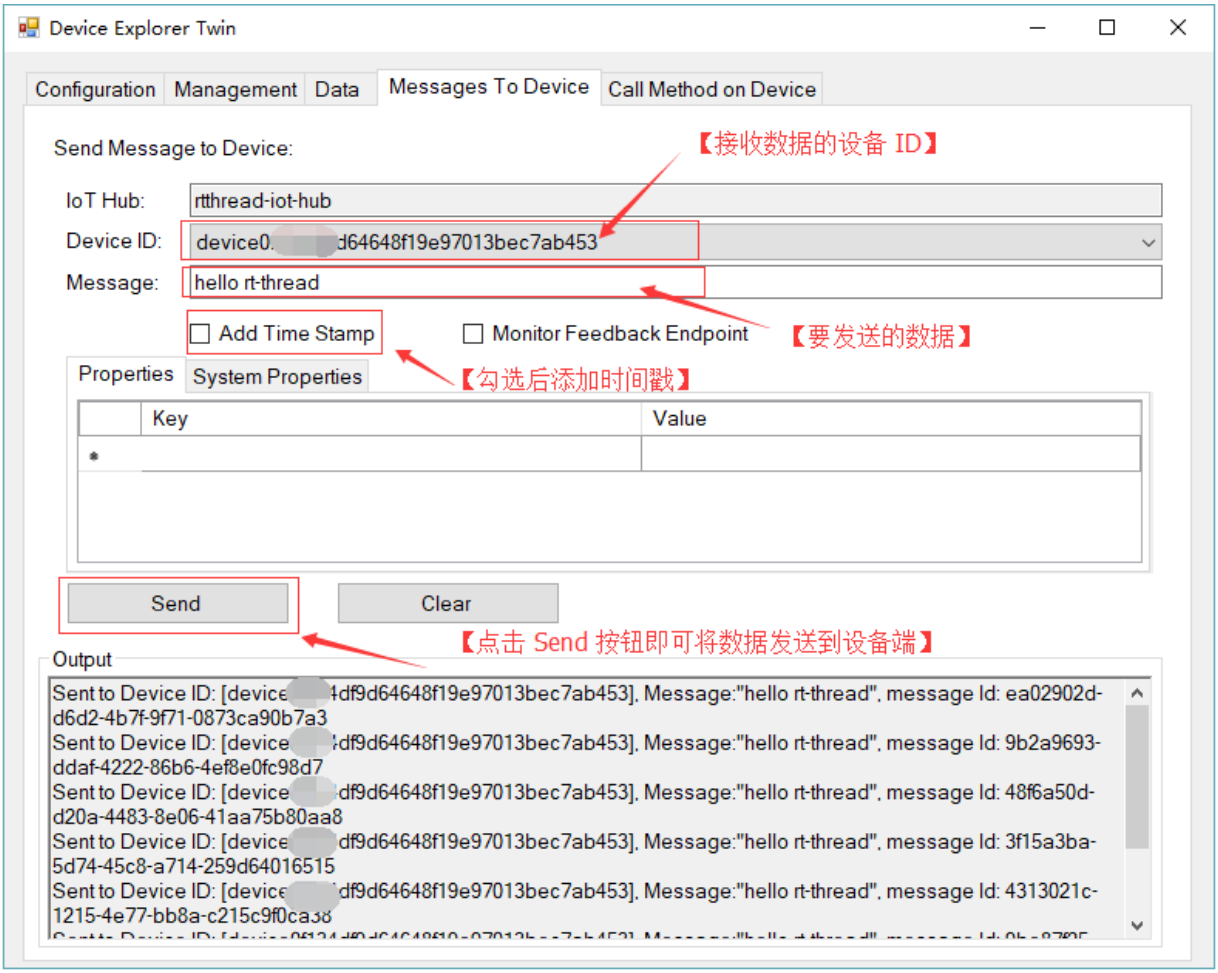


图 34.15: *服务器下发数据给设备*

2、此时在设备端查看从 IoT 中心下发给设备的数据：

```
msh />azure_c2d_sample
msh />
ntp init
Creating IoTHub Device handle
Waiting for message to be sent to device (will quit after 3 messages)
Received Binary message                              # 收到二进制数据
Message ID: ea02902d-d6d2-4b7f-9f71-0873ca90b7a3
 Correlation ID: <unavailable>
 Data: <<<hello rt-thread>>> & Size=15
Received Binary message
Message ID: 9b2a9693-ddaf-4222-86b6-4ef8e0fc98d7
 Correlation ID: <unavailable>
 Data: <<<hello rt-thread>>> & Size=15
Received Binary message
Message ID: 48f6a50d-d20a-4483-8e06-41aa75b80aa8
 Correlation ID: <unavailable>
 Data: <<<hello rt-thread>>> & Size=15
Received Binary message
Message ID: 3f15a3ba-5d74-45c8-a714-259d64016515
```

```
 Correlation ID: <unavailable>
 Data: <<<hello rt-thread>>> & Size=15
Received Binary message
Message ID: 4313021c-1215-4e77-bb8a-c215c9f0ca38
 Correlation ID: <unavailable>
 Data: <<<hello rt-thread>>> & Size=15
Received Binary message
Message ID: 9be87f25-2a6f-46b5-a413-0fb2f93f85d6
 Correlation ID: <unavailable>
 Data: <<<hello rt-thread>>> & Size=15
Error: Time:Tue Jul 31 13:54:14 2018
File:packages\azure\azure-port\pal\src\socketio_berkeley.c
Func:socketio_send Line:853 Failure: socket state is not opened.
Azure Sample Exit     #收到一定数量的下发数据，功能示例自动退出
```

34.5 注意事项

- 使用本例程前请先阅读《Azure 云平台软件包用户手册》

34.6 引用参考

- 《Azure 云平台软件包用户手册》：docs/UM1007-RT-Thread-Azure-IoT-SDK 用户手册.pdf

第 35 章

综合演示例程

本例程作为综合例程，在 IoT Board 上演示了 RT-Thread 传感器驱动、文件系统、网络、配网等功能，并通过 LCD 显示、按键输入与用户交互，在每个 LCD 界面展示不同的功能。

下面表格是每个界面的功能概览：

界面序号	界面标题	功能	说明
0	startup	启动界面	RT-Thread 与正点原子的 logo
1	IoT Board	主界面	显示 IoT Board 相关软硬件的版本信息
2	sensor	温湿度与光感传感器数据显示界面	显示温度、湿度、光感、接近感应的数据
4	BEEP/RGB	蜂鸣器/RGB 控制界面	按键控制蜂鸣器/ RGB 灯
5	SD card	SD 卡数据展示界面	扫描 sd 卡的内容，显示文件目录
6	Infrared	红外数据展示界面	展示红外自发自收数据
8	WiFi Scan	WiFi 扫描界面	显示扫描到的 wifi ssid
9	WeChat Scan	微信扫面二维码配网界面	用户可使用微信扫二维码，为 IoT Board 配置网络
10	WiFi Config	WiFi 配置等待界面	等待 wifi 连接成功

界面序号	界面标题	功能	说明
11	NetInfo	网络信息展示界面	IoT Board 联网成功后展示公网 IP 和网络时间
12	RT-Thread Cloud Scan & Config	RT-Thread Cloud 云平台扫码配置界面	需要设备先接入网络才会展示，通过微信扫码配置

35.1 硬件说明

IoT Board 的原理图位于：`RT-Thread_W60X_SDK/docs/board/W601_Board_V1.4_SCH.pdf`。

35.2 软件说明

iot_board_demo 综合例程位于 `examples/36_iot_board_demo` 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	应用程序入口
packages	依赖的 RT-Thread 软件包
packages/aht10-v1.0.0	温湿度传感器软件包
packages/ap3216c-v1.0.0	光强度传感器软件包
packages/EasyFlash-v3.2.1	轻量级 Flash 适配软件包
packages/fal-v0.2.0	Flash 抽象层软件包
packages/netutils-v1.0.0	网络小工具集合软件包
ports	移植文件
ports/cloudsdk	RT-Thread Cloud 云平台相关的软件移植
ports/easyflash	EasyFlash 软件包相关软件的移植
ports/fal	fal 软件包必要的软件移植
ports/wifi	wifi 功能组件必要的软件移植
modules	IoT Board 综合例程核心的软件模块
modules/event	综合例程事件分发处理模块
modules/infrared	综合例程红外自发自收处理模块
modules/iotb_workqueue	综合例程工作队列模块
modules/key	综合例程按键处理模块

文件	说明
modules/lcd	综合例程 LCD 显示屏模块
modules/sensor	综合例程传感器处理模块
modules/ymodem	综合例程 ymodem OTA 升级模块

程序入口：

```
int main(void)
{
    /* 显示启动页 */
    iotb_lcd_show_startup_page();

    /* 在 SD 卡上挂载文件系统 */
    if (iotb_sensor_sdcard_fs_init() != RT_EOK)
    {
        LOG_E("Init sdcard fs failed!");
    }

    /* 初始化 WIFI */
    if (iotb_sensor_wifi_init() != RT_EOK)
    {
        if (iotb_sdcard_wifi_image_upgrade() != RT_EOK)
        {
            /* 使用 'ymodem start' 命令升级 WIFI 固件 */
            LOG_E("sdcard upgrad 'wifi image' failed!");
            LOG_E("Input 'ymodem_start' cmd to try to upgrade!");
            lcd_set_color(BLACK, WHITE);
            lcd_clear(BLACK);
            lcd_show_string(0, 120 - 26 - 26, 24,  "SDCard upgrade wifi");
            lcd_show_string(0, 120 - 26, 24,  " image failed!");
            lcd_show_string(0, 120, 24,  "Input 'ymodem_start'");
            lcd_show_string(0, 120 + 26, 24,  "cmd to upgrade");
            return 0;
        }
    }

    LOG_E("iotb_workqueue_start!");
    /* 启动工作队列，异步处理耗时任务 */
    if (iotb_workqueue_start() != RT_EOK)
    {
        return -RT_ERROR;
    }

    iotb_workqueue_dowork(iotb_init, RT_NULL);


    /* 启动 LCD 线程，用于接收处理 menu 事件 */
```

```
    iotb_lcd_start();
    /* 启动事件处理器 */
    iotb_event_start();

    /* 启动按键处理线程 */
    iotb_key_process_start();
    LOG_E("Init key process start ok!");
    return 0;
}
```

示意图：

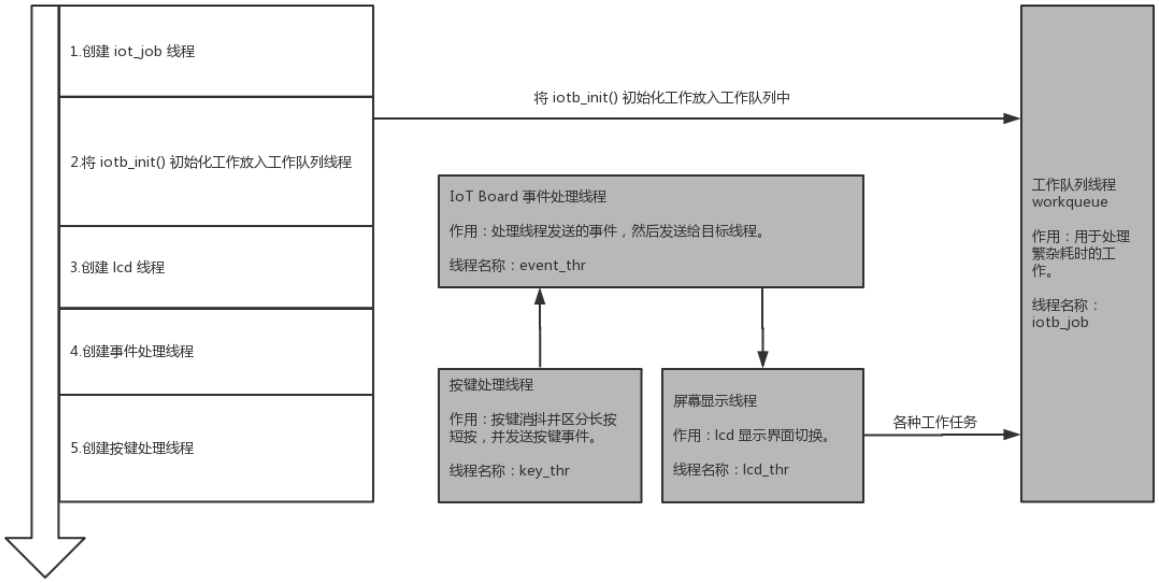


图 35.1: *程序示意图*

35.3 IoT Board 综合例程使用说明

35.3.1 编译 & 下载

- **MDK**：双击 `project.uvprojx` 打开 MDK5 工程，执行编译。

编译完成后，将固件下载至开发板。

35.3.2 按键使用说明

两个系统级按键：

- WK_UP 按键

  在任意页面长按 WK_UP 按键，进入微信扫码配网界面。

- KEY0 按键

  在任意页面单击 KEY0 按键，进入下一页。

其他功能按键：

- KEY1 按键

  根据具体界面提示使用，未提示此按键功能的页面不会响应该按键事件

35.3.3 SD 卡文件说明

综合例程在 SD 卡中需要一个必要的 `SYSTEM` 文件夹，综合例程在启动的时候会检查该目录，必要的时候进行相关固件的升级。目录结构如下：

目录	说明
SYSTEM	综合例程系统级目录
SYSTEM/WIFI	存放 WiFi 固件

35.3.4 LCD 界面说明

35.3.4.1 界面 0 启动界面

展示 RT-Thread 和正点原子的 LOGO。

图 35.2: *界面 0*

35.3.4.2 界面 1 主界面

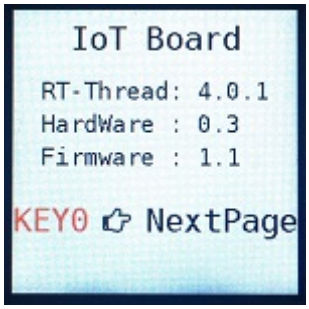


图 35.3: *界面 1*

该页面用于展示综合例程使用的软硬件版本及系统信息，并提示使用 **KEY0** 按键切换到下一页。

该界面对应的函数为 `static void iotb_lcd_show_index_page(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.3 界面 2 温湿度与光感

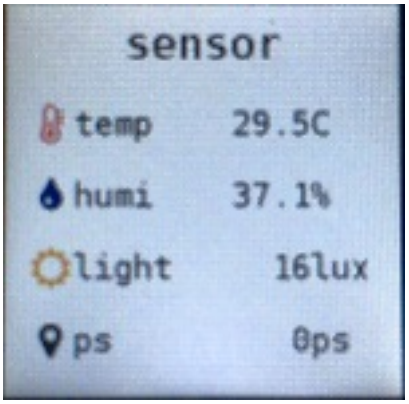


图 35.4: *界面 2*

展示从温湿度传感器 aht10 ，光照强度传感器 ap3216c 接收到的数据，按下 **KEY0** 按键可以切换到下一页。

这个界面功能用到了 RT-Thread package 提供的 **aht10-v1.0.0** 与 **ap3216c-v1.0.0** 这两个软件包。软件包的详细使用说明可参考 `/examples/07_driver_temp_humi` 与 `/examples/08_driver_als_ps` 中的例程。

该界面对应的函数为 `static void iotb_lcd_show_sensor(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.4 界面 3 蜂鸣器/RGB

图 35.5: *界面 4*

控制板载蜂鸣器、RGB 灯，按下 **KEY0** 按键可以切换到下一页。

- WK_UP 按键控制蜂鸣器
- KEY1 按键控制 RGB 灯调色

该界面对应的函数 `static void iotb_lcd_show_beep_motor_rgb(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.5 界面 4 SD card

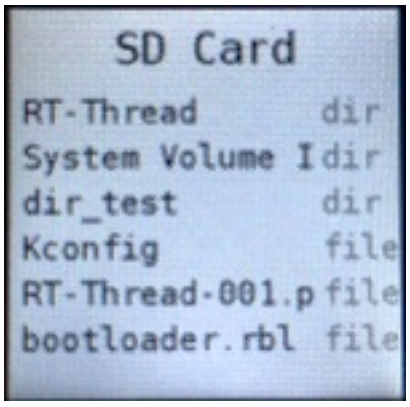


图 35.6: *界面 5*

该界面首先检测 SD 是否已经插入，然后展示 SD 卡文件系统内的文件，如果 IoT Board 未插入 SD 卡，将在屏幕中央一直显示 `Insert SD card before power-on` ，综合例程不支持热拔插，请在上电前插入 SD 卡。按下 **KEY0** 按键可以切换到下一页。

更多详细的文件系统使用说明可参考例程：`/examples/11_component_fs_tf_card`

该界面对应的函数 `static void iotb_lcd_show_sdcard(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.6 界面 5 红外收发

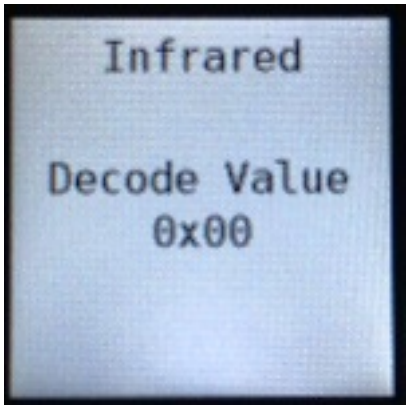


图 35.7: *界面 6*

使用 NEC 协议格式编码红外数据发送出去，然后再接收编码数据解码显示。

按下 **KEY0** 按键可以切换到下一页。

该界面对应的函数 `static void iotb_lcd_show_infrared(iotb_lcd_menu_t *lcd_menu);`。相关的处理业务逻辑请参考 `/36_iot_board_demo/modules/infrared` 中的程序以及 `/example/05_basic_ir` 例程。

35.3.4.7 界面 6 WiFi 扫描

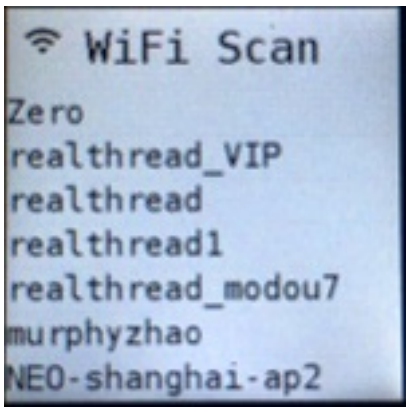


图 35.8: *界面 8*

使用设备上的 WiFi 模块扫描附近的 WiFi 热点，并在 LCD 上展示扫描到的 WiFi SSID。

按下 **KEY0** 按键可以切换到下一页。

该界面对应的函数 `static void iotb_lcd_show_wifiscan(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.8 界面 7 微信扫码配网

图 35.9: *界面 9*

只有在设备没有成功接入过网络的情况下会展示该页面，等待用户使用微信扫描二维码进行网络配置。设备成功接入网络后，不再显示该页面。

手机扫描二维码前，请设置手机接入 2.4G 频段的路由器热点。然后使用微信扫一扫扫描二维码，在微信中输入当前手机连入 wifi 的密码，点击 连接即可。IoT Board 会接收到 wifi 的广播信息自动连接到这个 WiFi 网络。

注意： 如果希望重新配置网络，请在任意页面下长按 WK_UP 按键约 6 秒。

按下 **KEY0** 按键可以切换到下一页。

手机配置步骤：

图 35.10: *界面 9*

该界面对应的函数 `static void iotb_lcd_show_wechatscan(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.9 界面 8 等待 WiFi 连接成功

图 35.11: *界面 10*

当 IoT Board 接收到 wifi 路由器的配网信息后，将会自动接入 wifi，接入成功后会保存 wifi 账号（以便下次开机自动连接此 wifi）并自动退出该界面。

该界面对应的函数 `static void iotb_lcd_show_wificonfig(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.10 界面 9 网络信息展示界面

图 35.12: *界面 11*

名称	说明
ssid	当前连接的 wifi 名称
LAN	设备 ip 地址
WAN	公网 ip 地址
DATE	年月日
TIME	时分秒

这里展示了 SSID、LAN、WAN、DATE、TIME 信息。按下 **KEY0** 按键可以切换到下一页。

该界面对应的函数 `static void iotb_lcd_show_network(iotb_lcd_menu_t *lcd_menu);`。

35.3.4.11 界面 10 扫描绑定设备到 RT-Thread 云平台

图 35.13: *界面 12-1*

如果设备已经正确接入网络，且没有被绑定过，则会在 LCD 菜单中展示该页面。用户需要使用手机扫描二维码（可以使用微信扫码）进入设备绑定页面，正确填写在 RT-Thread Cloud 平台注册的账户和密码既可以完成设备绑定。

设备成功被绑定后，会展示 `device online`，请在 `iot.rt-thread.com` 网站查看设备详细信息，如下图所示：

图 35.14: *界面 12*

35.4 注意事项

暂无

35.5 引用参考

- 《RT-Thread 编程指南》：docs/RT-Thread 编程指南.pdf