电源管理应用笔记

RT-THREAD 文档中心

WWW.RT-THREAD.ORG

Friday 19th October, 2018

目录

1 本文的目的背景和结构

1.1 本文的目的和背景

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

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

1.2 本文的结构

本文首先简要介绍了如何得到 RT-Thread 的电源管理组件（Power Management，以下简称 PM 组件），然后在 IoT Board 上运行相关的示例代码。最后深入介绍 PM 组件的设计思路和原理。

2 问题阐述

PM 组件从层次上划分，可以分成用户层、PM 组件层和 PM 驱动层。用户层包括了应用代码和驱动代码，它们通过 API 来决定芯片运行在什么模式。PM 驱动层主要是实现了 PM 驱动的支持以及与 PM 相关的外设功耗控制。PM 组件层里完成驱动的管理和对用户层提供支持。

本应用笔记将主要介绍用户层应该如何使用，而不过多设计到组件框架层和 PM 驱动层。应用层主要围绕了下面几个问题：

- PM 组件里面的模式是什么？有哪些不同类型的模式？
- 应用如何根据需求去管理模式？

3 在 IoT Board 上实现电源管理

本文的示例都是在 IoT Board 下运行。IoT Board 是 RT-Thread 和正点原子联合推出的硬件平台，该平台上专门为 IoT 领域设计，并提供了丰富的例程和文档。

本节主要展示了如何开启 PM 组件和相应的驱动，并通过例程来演示常见场景下，应用应该如何管理模式。

3.1 如何得到组件和相应的驱动

在 IoT Board 上运行电源管理组件，需要下载 IoT Board 的相关资料、RT-Thread 源码和 ENV 工具。

1. 下载 IoT Board 资料
2. 下载RT-Thread 源码
3. 下载ENV 工具

开启 env 工具，进入 IoT Board 的 **PM 例程目录**，在 env 命令行里输入 `menuconfig` 进入配置界面配置工程。

- 配置 PM 组件：勾选 BSP 里面的`Hareware Drivers Config ---> On-chip Peripheral Drivers ---> Enable Power Management`，使能了这个选项后，会自动选择 PM 组件和 PM 组件需要的 HOOK 功能：

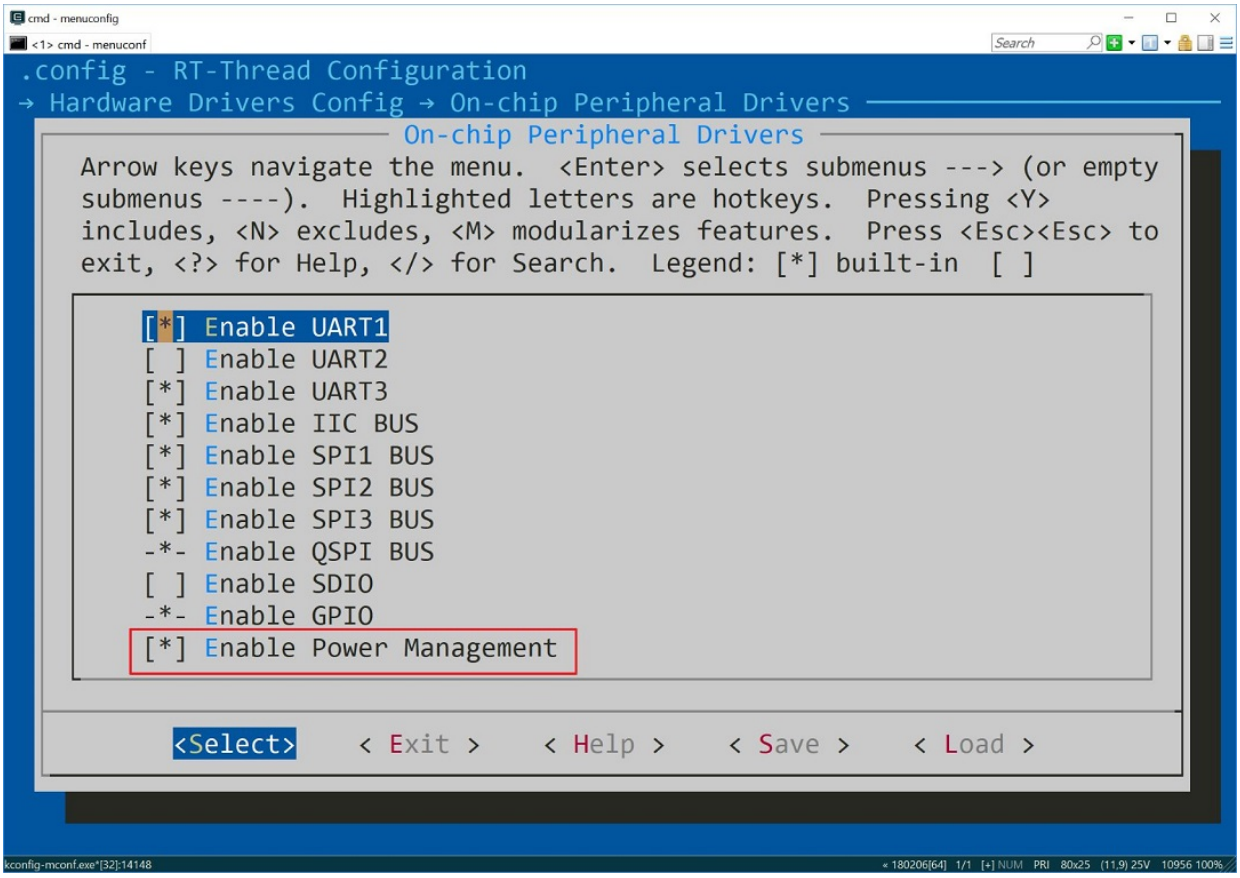

图 1：*配置组件*

- 配置内核选项：使用 PM 组件需要更大的 IDLE 线程的栈，这里使用了 1024 字节。例程里还使用 Software timer，所以我们还需要开启相应的配置

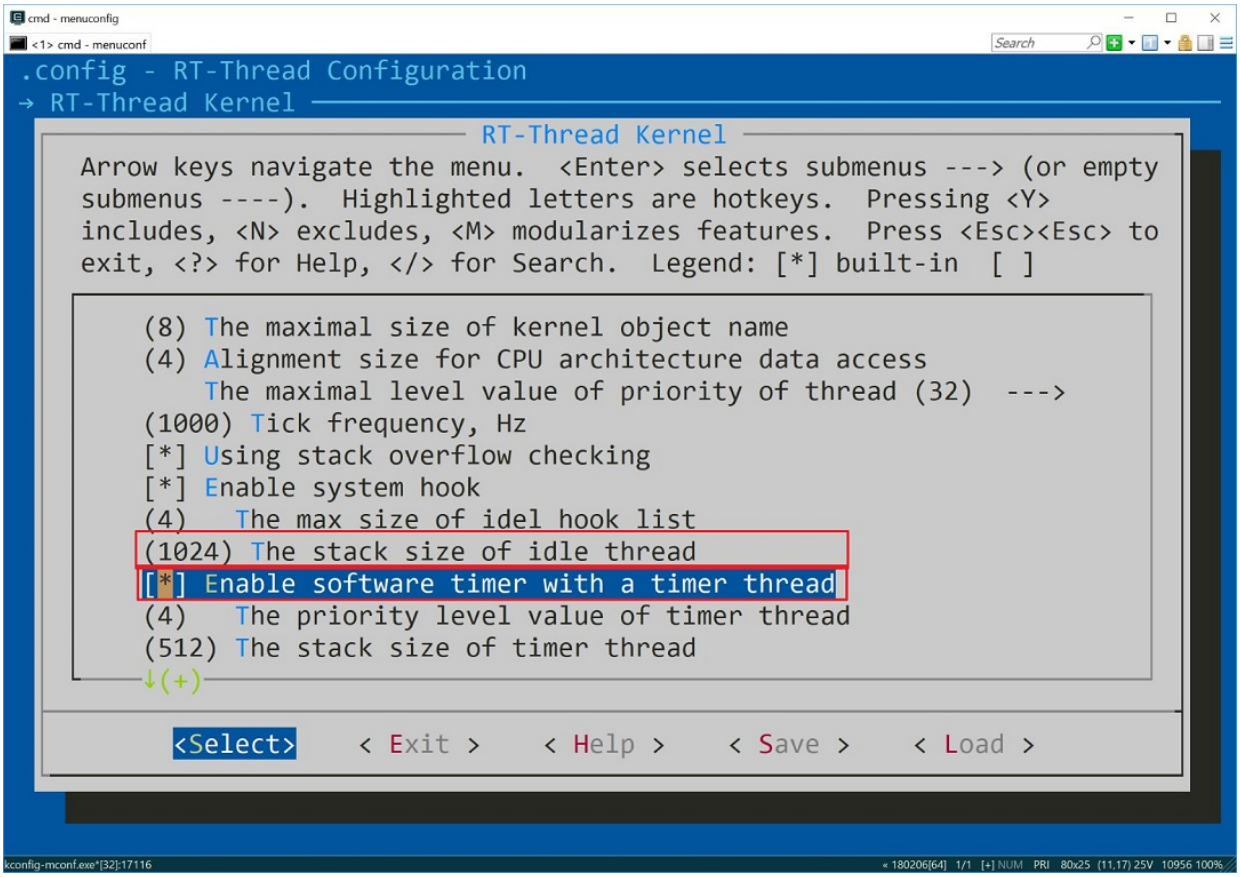

图 2: *配置内核选项*

- 配置完成，保存并退出配置选项，输入命令`scons --target=mdk5`生成 mdk5 工程；

打开 mdk5 工程可以看到相应的源码以及被添加进来：

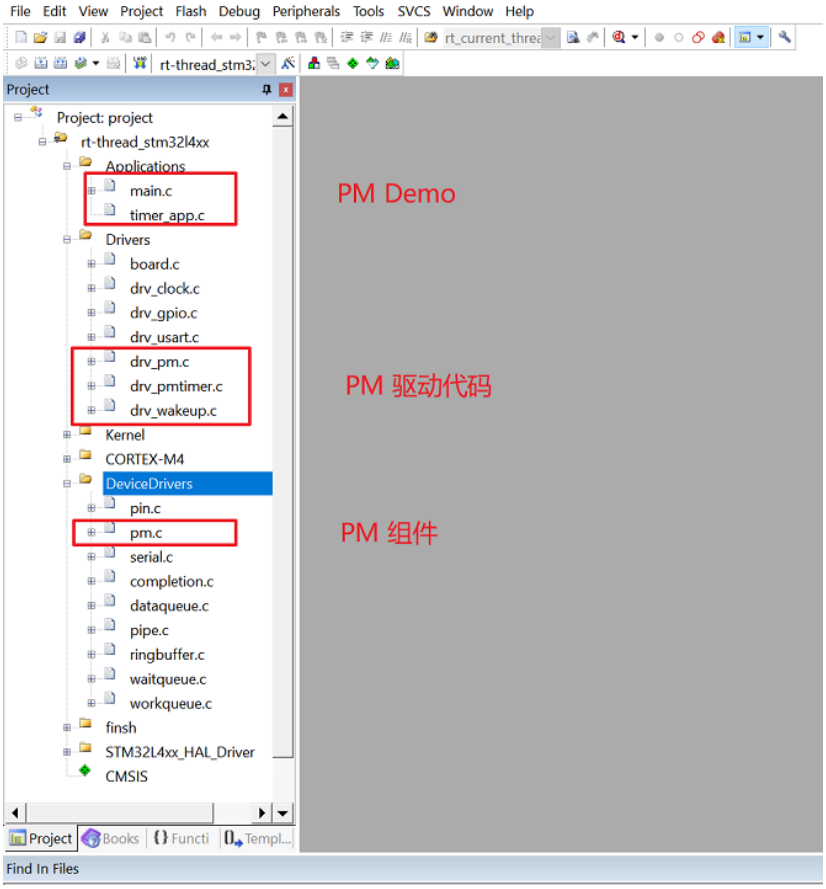

图 3: *MDK 工程*

3.2 IoT Board 上的低功耗例程

3.2.1. 定时应用 (timer_app)

在定时应用里，我们创建了一个周期性的软件定时器，定时器任务里周期性输出当前的 OS Tick。如果创建软件定时器成功之后，使用`rt_pm_request(PM_SLEEP_MODE_TIMER)`请求 TIMER 休眠模式。以下是示例核心代码：

```
#define TIMER_APP_DEFAULT_TICK  (RT_TICK_PER_SECOND * 2)

static rt_timer_t timer1;

static void _timeout_entry(void *parameter)
{
    rt_kprintf("current tick: %ld\n", rt_tick_get());
}

static int timer_app_init(void)
{
    timer1 = rt_timer_create("timer_app",
                             _timeout_entry,
                             RT_NULL,
                             TIMER_APP_DEFAULT_TICK,
```

```
                                RT_TIMER_FLAG_PERIODIC | RT_TIMER_FLAG_SOFT_TIMER);
    if (timer1 != RT_NULL)
    {
        rt_timer_start(timer1);

        /* keep in timer mode */
        rt_pm_request(PM_SLEEP_MODE_TIMER);

        return 0;
    }
    else
    {
        return -1;
    }
}
INIT_APP_EXPORT(timer_app_init);
```

按下复位按键重启开发板，打开终端软件，我们可以看到有定时输出日志：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Sep  7 2018
 2006 - 2018 Copyright by rt-thread team
[SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[SFUD] w25q128 flash device is initialize success.
sysclok: 80000000Hz
hclk:    80000000Hz
pclk1:   80000000Hz
pclk2:   80000000Hz
mmc1:    32000000Hz
msh />current tick: 2020
current tick: 4021
current tick: 6022
```

我们可以在 msh 里输入`pm_dump`命令观察 PM 组件的模式状态：

```
msh />pm_dump
| Power Management Mode | Counter | Timer |
+-----------------------+---------+-------+
|          Running Mode |       1 |     0 |
|            Sleep Mode |       1 |     0 |
|            Timer Mode |       1 |     1 |
|         Shutdown Mode |       1 |     0 |
+-----------------------+---------+-------+
pm current mode: Running Mode
```

以上的输出说明，PM 组件里所有 PM 模式都被请求了一次，现在正处于 Running 模式。`Running Mode`、`Sleep Mode`和`Shutdown Mode`都是启动的时候已经被默认请求了一次。`Timer Mode`在定时应用里被请求一次。

我们依次输入命令`pm_release 0`和`pm_release 1`手动释放 Running 和 Sleep 模式后，将进入`Timer Mode`。进入`Timer Mode`之后会定时唤醒。所以我们看到 shell 还是一直在输出：

```
msh />pm_release 0
msh />
msh />current tick: 8023
current tick: 10024
current tick: 12025

msh />pm_release 1
msh />
msh />current tick: 14026
current tick: 16027
current tick: 18028
current tick: 20029
current tick: 22030
current tick: 24031
```

我们可以通过功耗仪器观察功耗的变化。下图是基于 Monsoon Solutions Inc 的 Power Monitor 的运行截图，可以看到随着模式变化，功耗明显变化：

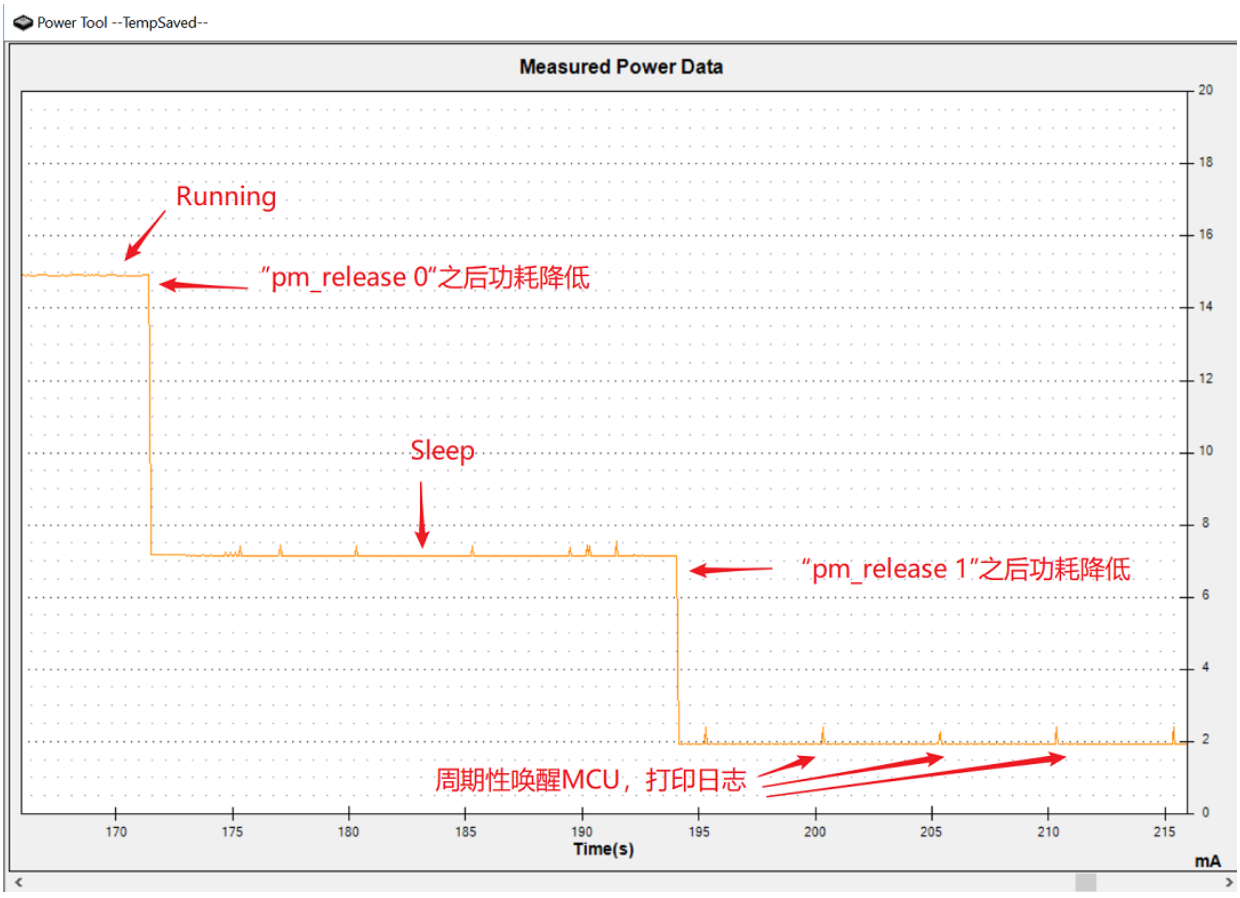

图 4: *功耗变化*

休眠时显示 2mA 是仪器的误差。

3.2.2. 按键唤醒应用

在按键唤醒应用里，我们使用 wakeup 按键来唤醒处于休眠模式的 MCU。一般情况下，在 MCU 处于比较深度的休眠模式，只能通过特定的方式唤醒。MCU 被唤醒之后，会触发相应的中断。以下例程是从 Timer 模式唤醒 MCU 并闪烁 LED 之后，再次进入休眠的例程。以下是核心代码：

```
#define WAKEUP_EVENT_BUTTON                 (1 << 0)

static rt_event_t wakeup_event;

static void wakeup_callback(void)
{
    rt_event_send(wakeup_event, WAKEUP_EVENT_BUTTON);
}

static void wakeup_app_entry(void *parameter)
{
    bsp_register_wakeup(wakeup_callback);

    while (1)
    {
        if (rt_event_recv(wakeup_event,
                          WAKEUP_EVENT_BUTTON,
                          RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                          RT_WAITING_FOREVER, RT_NULL) == RT_EOK)
        {
            rt_pm_request(PM_RUN_MODE_NORMAL);

            rt_pin_mode(PIN_LED_R, PIN_MODE_OUTPUT);
            rt_pin_write(PIN_LED_R, 0);
            rt_thread_delay(rt_tick_from_millisecond(100));
            rt_pin_write(PIN_LED_R, 1);
            _pin_as_analog();

#ifdef WAKEUP_APP_DEFAULT_RELEASE
            rt_pm_release(PM_RUN_MODE_NORMAL);
#endif
        }
    }
}

static int wakeup_app(void)
{
    rt_thread_t tid;

    wakeup_event = rt_event_create("wakup", RT_IPC_FLAG_FIFO);
    RT_ASSERT(wakeup_event != RT_NULL);

    tid = rt_thread_create("wakeup_app", wakeup_app_entry, RT_NULL,
```

```
                            WAKEUP_APP_THREAD_STACK_SIZE, RT_MAIN_THREAD_PRIORITY,
                                20);
    RT_ASSERT(tid != RT_NULL);

    rt_thread_startup(tid);

    return 0;
}
INIT_APP_EXPORT(wakeup_app);
```

上面的代码里，我们创建一个线程，这个线程里注册了按键中断唤醒回调函数，每当唤醒中断之后就会调用该函数。回调函数里会发送事件WAKEUP_EVENT_BUTTON。这样我们的线程里接收到这个事件之后，首先请求在 Normal 模式，然后完成 LED 闪烁功能之后，再去释放 Normal 。

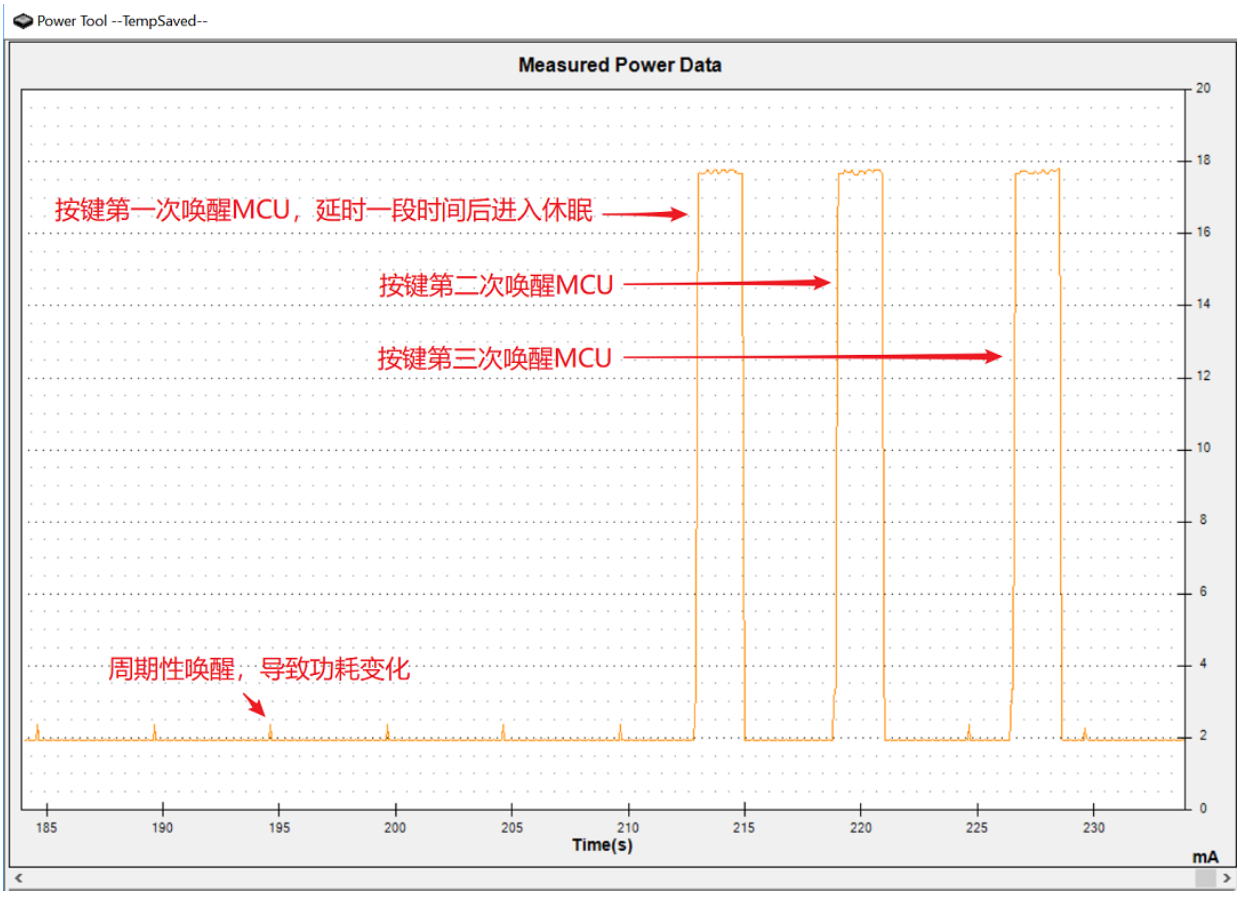

图 5: *功耗变化*

上图是我们三次按下 wakeup 按键的运行截图。每次按下按键，MCU 都会被唤醒点亮 LED 2 秒之后，再次进入休眠。

4 电源管理组件的深入理解

4.1 电源管理组件的模式是什么？

MCU 通常提供了多种时钟源供用户选择。例如 IoT Board 上板载的 STM32L475 就可以选择 LSI/MSI/HSI 等内部时钟，还可以选择 HSE/LSE 等外部时钟。MCU 内通常也集成了 PLL(Phase-locked loops)，基于不同的时钟源，向 MCU 的其他模块提供更高频率的时钟。

为了支持低功耗功能，MCU 里也会提供不同的休眠模式。例如 STM32L475 里，可以分成 SLEEP 模式、STOP 模式、STANDBY 模式。这些模式还可以有进一步的细分，以适应不同的场合。

以上是 STM32L475 的情况。在不同的 MCU 之间，它们的时钟和低功耗可能会有很大的差异。高性能的 MCU 可以运行在 600M 以上或者更高，低功耗的 MCU 可以在 1~2M 以极低的功耗运行。根据实际情况的不同，低功耗模式可以选择停止不同的外设，支持不同的外设在休眠模式里唤醒。

为了使得上层用户开发低功耗应用的时候，可以轻松处理芯片之间的差异，RT-Thread 的 PM 组件里基于模式来管理 MCU 的时钟和低功耗模式。不同的 MCU 根据实际情况定义不同的模式，例如高性能模式，普通模式，低功耗运行模式，休眠模式，定时器唤醒模式，关机模式等等。然后在驱动里和应用代码里，根据需要通过`rt_pm_request()`和`rt_pm_release()`来管理芯片的功耗。

4.2 运行模式和休眠模式是什么，有什么不同？

从模式的类型来区分，我们可以将 PM 组件的模式可以分成运行模式和休眠模式。在运行模式里，CPU 还是处于运行状态，根据 CPU 的频率不同可以分成不同的运行模式，根据运行的频率可以分成不同的运行模式。而在休眠模式里，CPU 已经停止工作，根据不同的外设是否还在工作可以分成不同的休眠模式。

4.3 模式的一票否决是什么？

在多个线程里，不同的线程可能请求不同的模式。例如线程 A 里请求运行在高性能运行模式，线程 B 和线程 C 里都请求运行在普通运行模式里。这种情况，PM 组件应该选择哪个模式？

这时候应该选择尽可能满足所有线程的请求的模式。高性能模式通常可以更好的完成在普通运行模式的功能，如果选择了高性能模式，那么线程 A/B/C 都可以正确运行。如果选择了普通模式，那么线程 A 的需求就无法得到满足。

因此在 RT-Thread 的电源管理组件里，只要有一个模式请求了更高的模式，就不会切换到比它低的模式。这就是模式的一票否决。

5 电源管理组件的 API 介绍

5.1 API 列表

PM 组件 API 列表	位置
rt_system_pm_init()	pm.c
rt_pm_request()	pm.c
rt_pm_release()	pm.c

PM 组件 API 列表	位置
rt_pm_register_device()	pm.c
rt_pm_unregister_device()	pm.c
rt_pm_enter()	pm.c
rt_pm_exit()	pm.c

5.2 API 详解

5.2.1. PM 组件初始化

```
void rt_system_pm_init(const struct rt_pm_ops *ops,
                       rt_uint8_t              timer_mask,
                       void                   *user_data);
```

PM 组件初始化函数，是由相应的 PM 驱动来调用，完成 PM 组件的初始化。

该函数完成的工作，包括底层 PM 驱动的注册、相应的 PM 组件的资源初始化、默认模式的请求，并对上层提供一个名字是 “pm” 的设备，还默认请求了三个模式，包括一个默认的 RUN 模式、一个 SLEEP 模式、以及最低的模式。

参数	描述
ops	底层 PM 驱动的函数集合
timer_mask	指定哪些模式包含了低功耗定时器
user_data	可以被底层 PM 驱动使用的一个指针

5.3 请求 PM 模式

```
void rt_pm_request(rt_ubase_t mode);
```

请求 PM 模式函数，是在应用或者驱动里调用的函数，调用之后 PM 组件确保当前模式不低于请求的模式。

参数	描述
mode	请求的模式

5.4 释放 PM 模式

```
void rt_pm_release(rt_ubase_t mode);
```

释放 PM 模式函数，是在应用或者驱动里调用的函数，调用之后 PM 组件并不会立即进行实际的模式切换，而是在rt_pm_enter()里面才开始切换。

参数	描述
mode	释放的模式

5.5 注册 PM 模式变化敏感的设备

```
void rt_pm_register_device(struct rt_device* device, const struct rt_device_pm_ops*
    ops);
```

该函数注册 PM 模式变化敏感的设备，每当 PM 的模式发生变化的时候，会调用设备的相应 API。

如果是切换到新的运行模式，会调用设备里的frequency_change()函数。

如果是切换到新的休眠模式，将会在进入休眠时调用设备的suspend()函数，在进入休眠被唤醒之后调用设备的resume()。

参数	描述
device	具体对模式变化敏感的设备
ops	设备的函数集合

5.6 取消注册 PM 模式变化敏感的设备

```
void rt_pm_unregister_device(struct rt_device* device);
```

该函数取消已经注册的 PM 模式变化敏感设备。

5.7 PM 模式进入函数

```
void rt_pm_enter(void);
```

该函数尝试进入更低的模式，如果没有请求任何运行模式，就进入休眠模式。这个函数已经在 PM 组件初始化函数里注册到 IDLE HOOK 里，所以不需要另外的调用。

5.8 PM 模式退出函数

```
void rt_pm_exit(void);
```

该函数在从休眠模式唤醒的时候被调用。它可能在从休眠唤醒的中断函数处理函数里由用户主动调用。无论中断处理函数有没有调用，在rt_pm_enter()都会调用一次。