# Power Management Application Notes

**RT-THREAD** Documentation Center



**WWW.RT-THREAD.ORG**

**Friday 19th October, 2018**

Table of contents

# 1 Purpose, Background and Structure of this Paper

## 1.1 Purpose and Background of this Paper

With the rise of the Internet of Things (IoT), the demand for power consumption of products is becoming increasingly strong. Sensor nodes for data collection usually need to work for a long time when powered by batteries, while SOCs for networking also need to have fast response functions and low power consumption.

At the beginning of product development, the first consideration is to complete the product function development as soon as possible. After the product functions are gradually improved, power management functions need to be added. In order to meet the needs of IoT, RT-Thread provides a power management framework. The concept of the power management framework is to be as transparent as possible, making it easier to add low-power functions to products.

## 1.2 Structure of this paper

This article first briefly introduces how to get the power management component (Power Management, hereinafter referred to as PM component) of RT-Thread. Then run the sample code on IoT Board. Finally, the design ideas and principles of PM components are introduced in depth.

# 2 Problem Statement

PM components can be divided into user layer, PM component layer and PM driver layer. The user layer includes application code and driver code, which use API to determine the mode in which the chip runs. The PM driver layer mainly implements PM driver support and PM-related peripheral power consumption control. The PM component layer completes driver management and provides support for the user layer.

This application note will mainly introduce how to use the user layer, without too much design of the component framework layer and PM driver layer. The application layer mainly revolves around the following issues:

- What are the modes in the PM component? What are the different types of modes? • How can applications manage modes based on their needs?

# 3. Implementing Power Management on IoT **Board**

The examples in this article are all run on IoT Board. IoT Board is a hardware platform jointly launched by RT-Thread and Zhengdian Atom, which is specially designed for the IoT field and provides rich routines and documents.

This section mainly shows how to enable the PM component and the corresponding driver, and uses routines to demonstrate how applications should manage modes in common scenarios.

## 3.1 How to get components and corresponding drivers

To run the power management component on the IoT Board, you need to download the IoT Board's related information, RT-Thread source code, and ENV tool.

1. Download IoT Board data 2.
Download RT-Thread source code
3. Download ENV tool

Open the env tool, enter the **PM** routine directory of IoT Board, and enter menuconfig in the env command line to enter the configuration interface.

Installation project.

- Configure PM components: Check Hareware Drivers Config in BSP ---> On-chip Peripheral Drivers --->

  Enable Power Management. After enabling this option, the PM component and the HOOK function required by the PM component will be automatically selected.

  able:

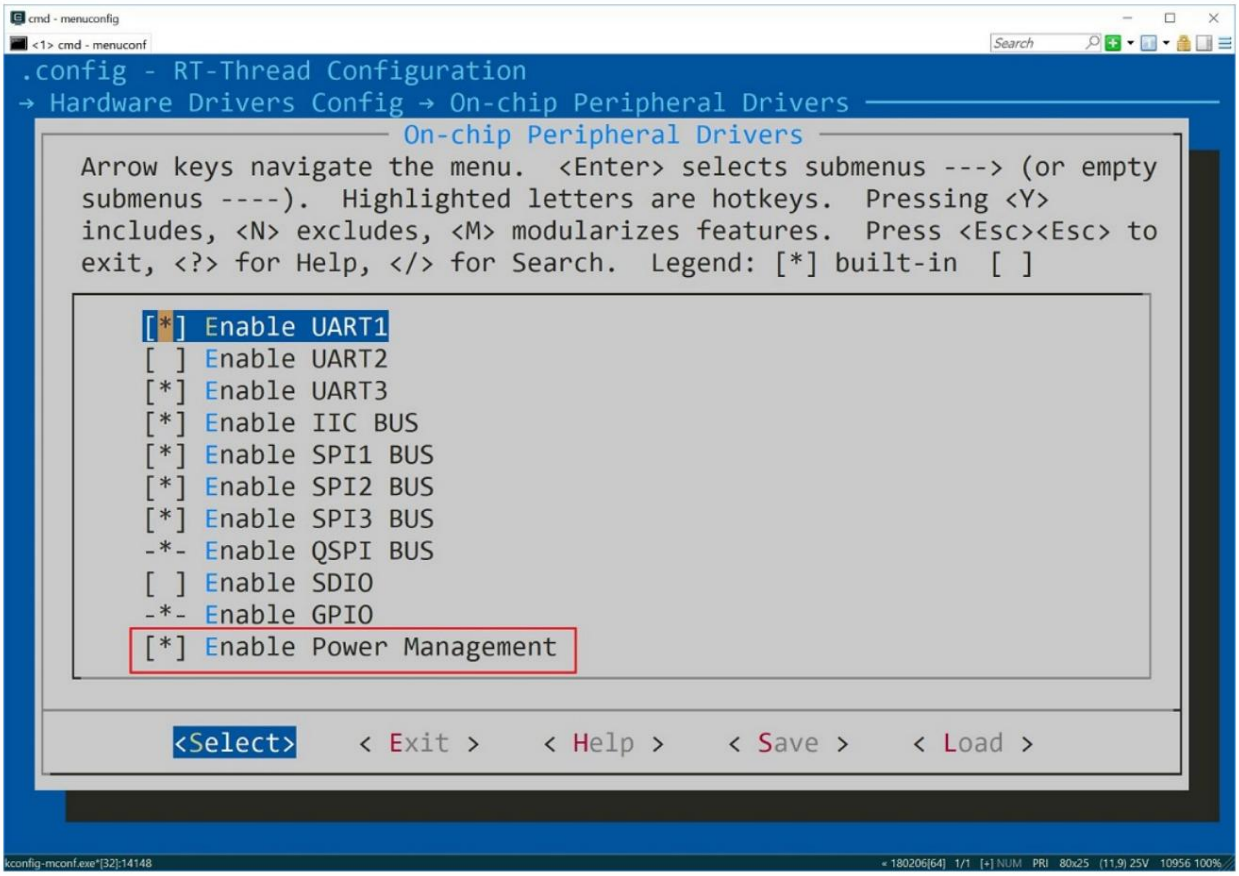


figure **1:** Configuring Components

- Configure kernel options: Using the PM component requires a larger IDLE thread stack, so 1024 bytes is used here.

  Software timer, so we also need to enable the corresponding configuration

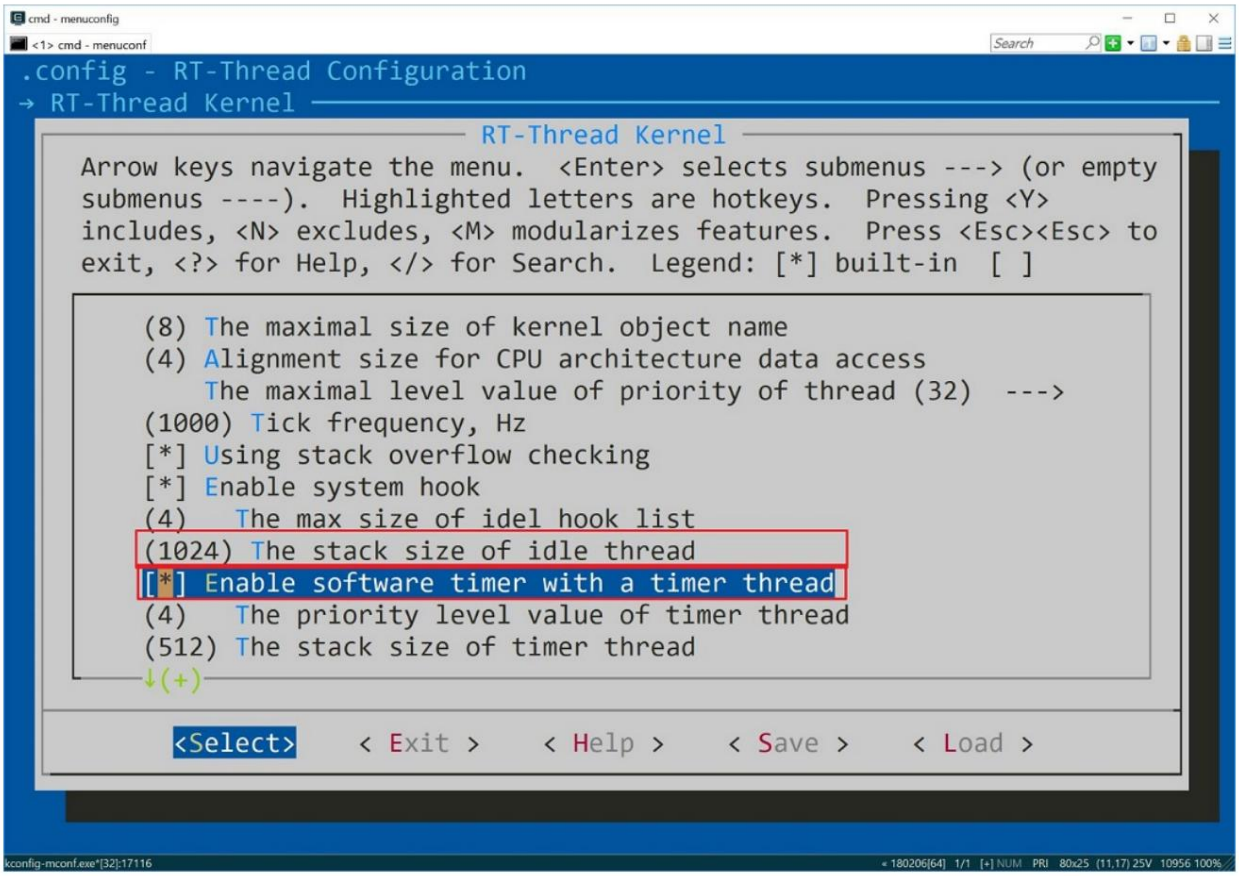


figure **2:** Configuring Kernel Options

• After configuration is complete, save and exit the configuration options, and enter the command scons --target=mdk5 to generate the mdk5 project;

Open the mdk5 project and you can see the corresponding source code and it has been added:

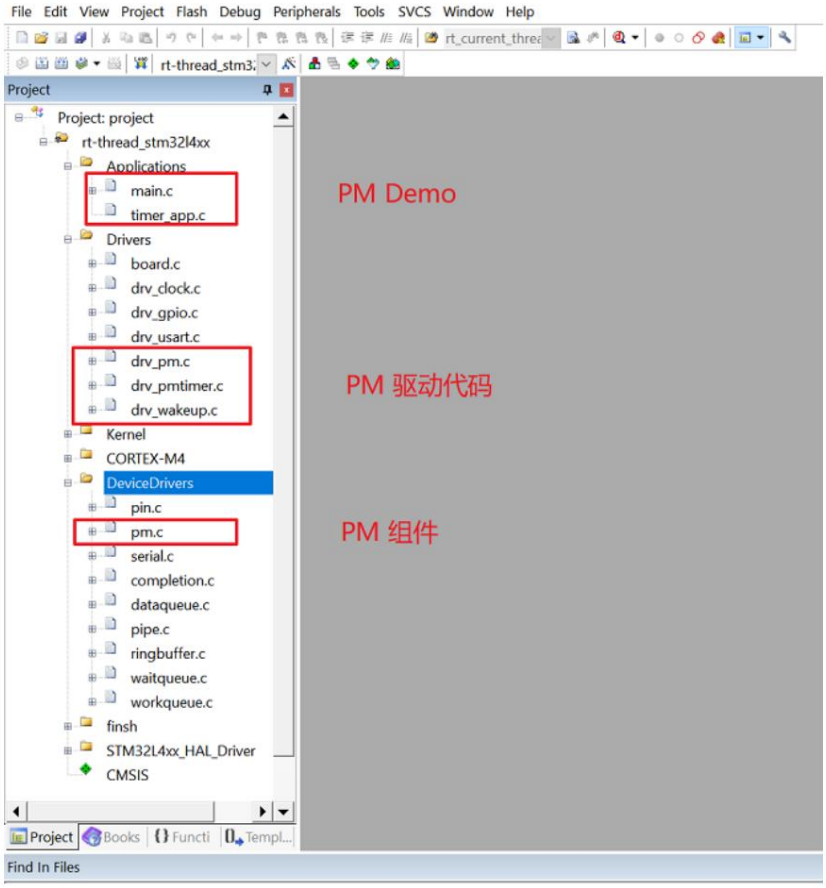


Figure **3:** *MDK* project

## 3.2 Low power consumption routines on **IoT Board**

### **3.2.1.** Timer application **(timer_app)**

In the timing application, we create a periodic software timer, and the timer task periodically outputs the current OS Tick. If the software timer is created successfully, use rt_pm_request(PM_SLEEP_MODE_TIMER) to request the TIMER sleep mode. The following is the sample core code:

```
#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);
```

Press the reset button to restart the development board, open the terminal software, and we can see the timed output log:

```
 \ | /
-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
```

We can enter the pm_dump command in msh to observe the mode status of the PM component:

```
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
```

The above output shows that all PM modes in the PM component have been requested once and are now in Running mode. Sleep Mode and Shutdown Mode are both requested once by default when the system is started. Timer Mode is requested once in the timing application .

Second-rate.

We enter the commands pm_release 0 and pm_release 1 to manually release the Running and Sleep modes, and then enter the Timer mode.
Mode. After entering Timer Mode, it will wake up at a fixed time. So we can see that the shell is still outputting:

```
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
```

We can observe the changes in power consumption through power consumption instruments. The following figure is based on the operation of Monsoon Solutions Inc's Power Monitor.

From the screenshot, you can see that the power consumption changes significantly as the mode changes:

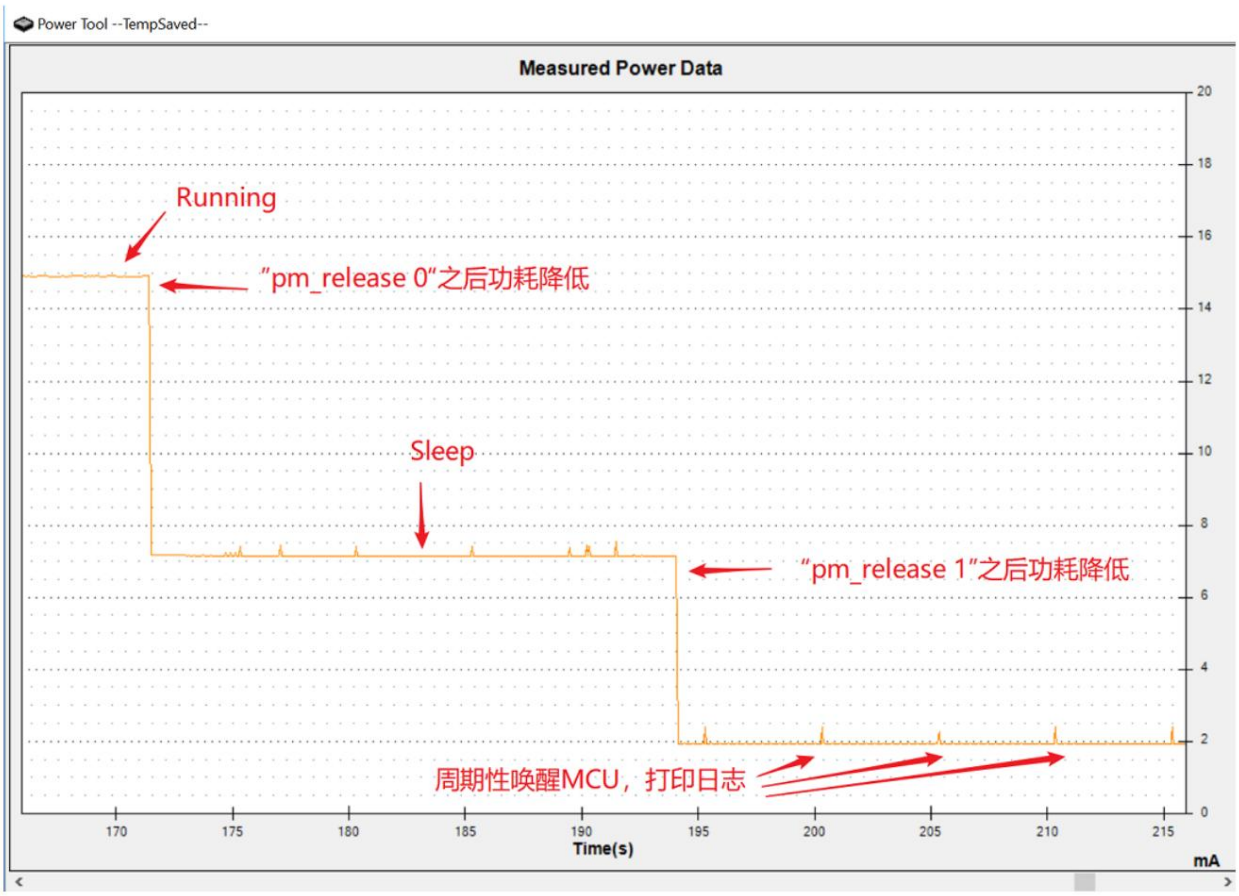


Figure 4: Power consumption changes

The 2mA displayed during sleep is the error of the instrument.

### 3.2.2. Button wake-up application

In the key wake-up application, we use the wakeup button to wake up the MCU in sleep mode. Generally, when the MCU is in a relatively deep sleep mode, it can only be awakened by a specific method. After the MCU is awakened, the corresponding interrupt will be triggered. The following routine is a routine that wakes up the MCU from the Timer mode and flashes the LED, and then enters sleep again. The following is the core code:

```
#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);
```

In the above code, we create a thread, register the key interrupt wakeup callback function in this thread, and call this function every time the interrupt is woken up . The callback function will send the event WAKEUP_EVENT_BUTTON. In this way, after receiving this event in our thread, we first request to be in Normal mode, and then release Normal after completing the LED flashing function.

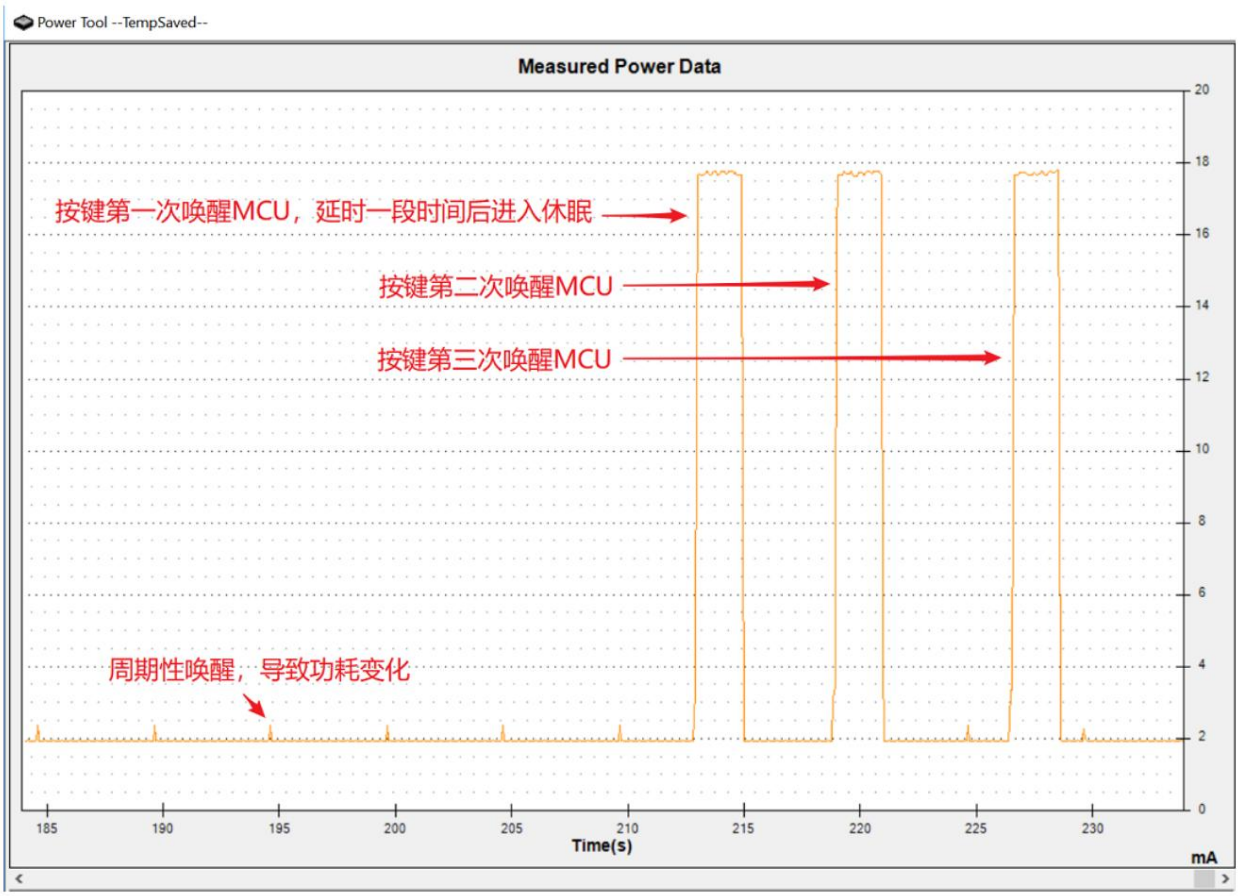


Figure 5: Power consumption changes

The above picture is a screenshot of the operation when we press the wakeup button three times. Each time the button is pressed, the MCU will be awakened and the LED will light up for 2 seconds before entering sleep again.

## 4. In-depth understanding of power management components

### 4.1 What are the modes of the power management component?

MCU usually provides multiple clock sources for users to choose from. For example, the STM32L475 on the IoT Board can choose internal clocks such as LSI/MSI/HSI, and external clocks such as HSE/LSE. MCUs usually also integrate PLLs (Phase-locked loops) to provide higher frequency clocks to other modules of the MCU based on different clock sources.

In order to support low power consumption, MCU also provides different sleep modes. For example, in STM32L475, it can be divided into SLEEP mode These modes can be further subdivided to suit different occasions.

The above is the case of STM32L475. The clock and low power consumption may vary greatly between different MCUs. High-performance MCUs can run at 600M or higher, and low-power MCUs can run at 1~2M with extremely low power consumption. Depending on the actual situation, the low-power mode can choose to stop different peripherals and support different peripherals to wake up in sleep mode.

In order to make it easy for upper-level users to handle differences between chips when developing low-power applications, the PM component of RT-Thread manages the clock and low-power mode of the MCU based on the mode. Different MCUs define different modes according to actual conditions, such as high-performance mode, normal mode, low-power operation mode, sleep mode, timer wake-up mode, shutdown mode, etc. Then, in the driver and application code , rt_pm_request() and rt_pm_release() are used to manage the power consumption of the chip as needed.

### 4.2 What are the running mode and the sleeping mode? What are the differences?

From the perspective of the mode type, we can divide the PM component modes into running mode and sleep mode. In the running mode, the CPU is still running, and can be divided into different running modes according to the CPU frequency. In the sleep mode, the CPU has stopped working, and can be divided into different sleep modes according to whether different peripherals are still working.

## What is the veto in 4.3 mode?

In multiple threads, different threads may request different modes. For example, thread A requests to run in high-performance mode, while thread B Both thread A and thread C request to run in normal operation mode. In this case, which mode should the PM component choose?

At this time, you should choose a mode that satisfies the requests of all threads as much as possible. High-performance mode can usually better complete the functions of normal operation mode. If high-performance mode is selected, threads A/B/C can all run correctly. If normal mode is selected, the needs of thread A cannot be met.

Therefore, in the power management component of RT-Thread, as long as a mode requests a higher mode, it will not switch to a lower mode. This is the veto of the model.

# 5. Introduction to the API of Power Management Components

## 5.1 API List

| PM component API list | Location |
|---|---|
| rt_system_pm_init() | pm.c |
| rt_pm_request() | pm.c |
| rt_pm_release() | pm.c |

| PM component API list | Location |
| --- | --- |
| 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 Detailed Explanation

#### 5.2.1. PM component initialization

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

The PM component initialization function is called by the corresponding PM driver to complete the initialization of the PM component.

The work completed by this function includes registration of the underlying PM driver, resource initialization of the corresponding PM component, request for the default mode, and A device named "pm" is provided to the upper layer, and three modes are requested by default, including a default RUN mode, a SLEEP mode, and a mode, and the lowest mode.

| parameter | describe |
| --- | --- |
| ops | Function set of the underlying PM driver |
| timer_mask | Specifies which modes include the low power timer |
| user_data | A pointer that can be used by the underlying PM driver |

## 5.3 Request PM Mode

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

The PM mode request function is a function called in the application or driver. After the call, the PM component ensures that the current mode is not lower than the requested mode. model.

| parameter | describe |
| --- | --- |
| mode | Requested Mode |

## 5.4 Release PM mode

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

The PM mode release function is called in the application or driver. After the call, the PM component will not immediately enter the actual mode.

Switching does not start in rt_pm_enter(), but in rt_pm_enter().

| parameter | describe |
| --- | --- |
| mode | Release Mode |

### 5.5 Registering devices sensitive to PM mode changes

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

This function registers devices that are sensitive to PM mode changes. Whenever the PM mode changes, the corresponding API of the device will be called.

If it is switching to a new operating mode, the frequency_change() function in the device will be called.

If you switch to a new sleep mode, the device's suspend() function will be called when entering sleep mode, and the device's suspend() function will be called after waking up from sleep mode.

Use resume() of the device.

| parameter | describe |
| --- | --- |
| device | Devices that are specifically sensitive to mode changes |
| ops | Function set of the device |

### 5.6 Unregister devices that are sensitive to PM mode changes

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

This function cancels the registered PM mode change sensitive device.

### 5.7 PM mode entry function

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

This function attempts to enter a lower mode, or sleep mode if no run mode is requested. This function is already in the PM group

It is registered to IDLE HOOK in the initialization function of the device, so no additional call is required.

### 5.8 PM mode exit function

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

This function is called when waking up from sleep mode. It may be called actively by the user in the interrupt function handler of waking up from sleep mode.

Regardless of whether the interrupt handling function is called or not, rt_pm_enter() will be called once.