RT-THREAD Power Management User Manual

RT-THREAD Documentation Center

WWW.RT-THREAD.ORG

Friday 19th October, 2018

Table of contents

1 Introduction

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, it is necessary to add power management (PM) functions. 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.

1.1 PM component features

• The PM component manages power consumption based on
the mode. • The PM component can automatically update the frequency configuration of the device according to the mode to ensure that it can work properly in different
operating modes. • The PM component can automatically manage the suspend and resume of the device according to the mode to ensure that it can suspend and resume correctly in
different sleep modes. • The PM component supports optional sleep time compensation, so that applications that rely on OS Tick can use
it transparently. • The PM component provides a device interface to the upper layer. If the devfs component is turned on, it can also be accessed through the file system interface.

2Get Started

This section mainly shows how to enable the PM component and the corresponding driver, and learns how to use the PM component through routines.

2.1 Choose the right BSP platform

Currently, the BSP platform that supports PM components is mainly IoT Board, and more platforms will be supported in the future. Therefore, the examples in this section are based on IoT Board demonstrations. 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.

2.2 How to get PM components and corresponding drivers

To run the power management component on the IoT Board, you need to download the IoT Board's related information and ENV tool:

1. Download IoT Board data 2. Download
ENV tool

Then copy the timer_app.c file attached to this article to the application directory of the **PM** routine directory of the IoT Board .

Finally, 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 configuration project:

• Configure PM components: Check Hareware Drivers Config ---> On-chip Peripheral Drivers in BSP
---> Enable Power Management. After enabling this option, the PM component and the IDLE required by the PM component will be automatically selected.
HOOK function:

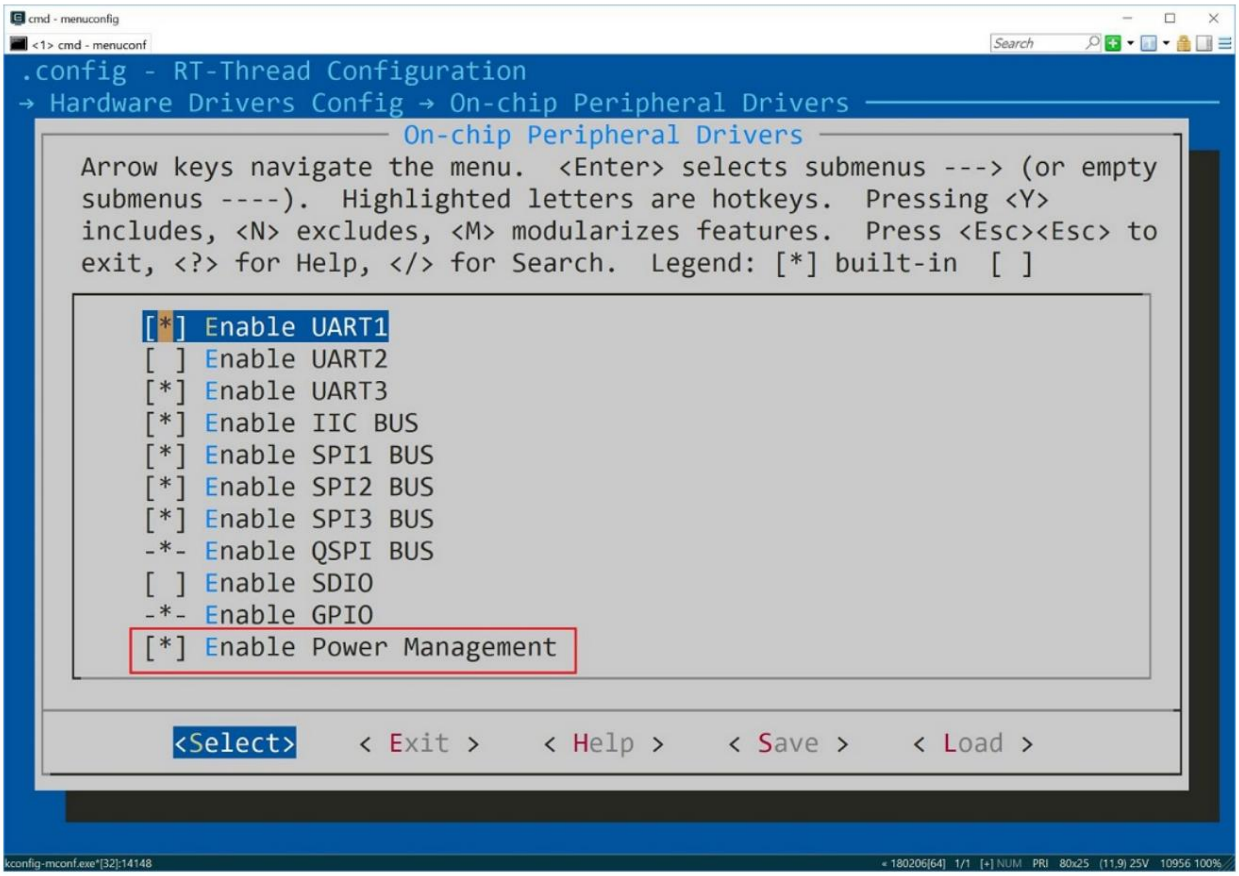

figure **1:** Configuring Components

• 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:

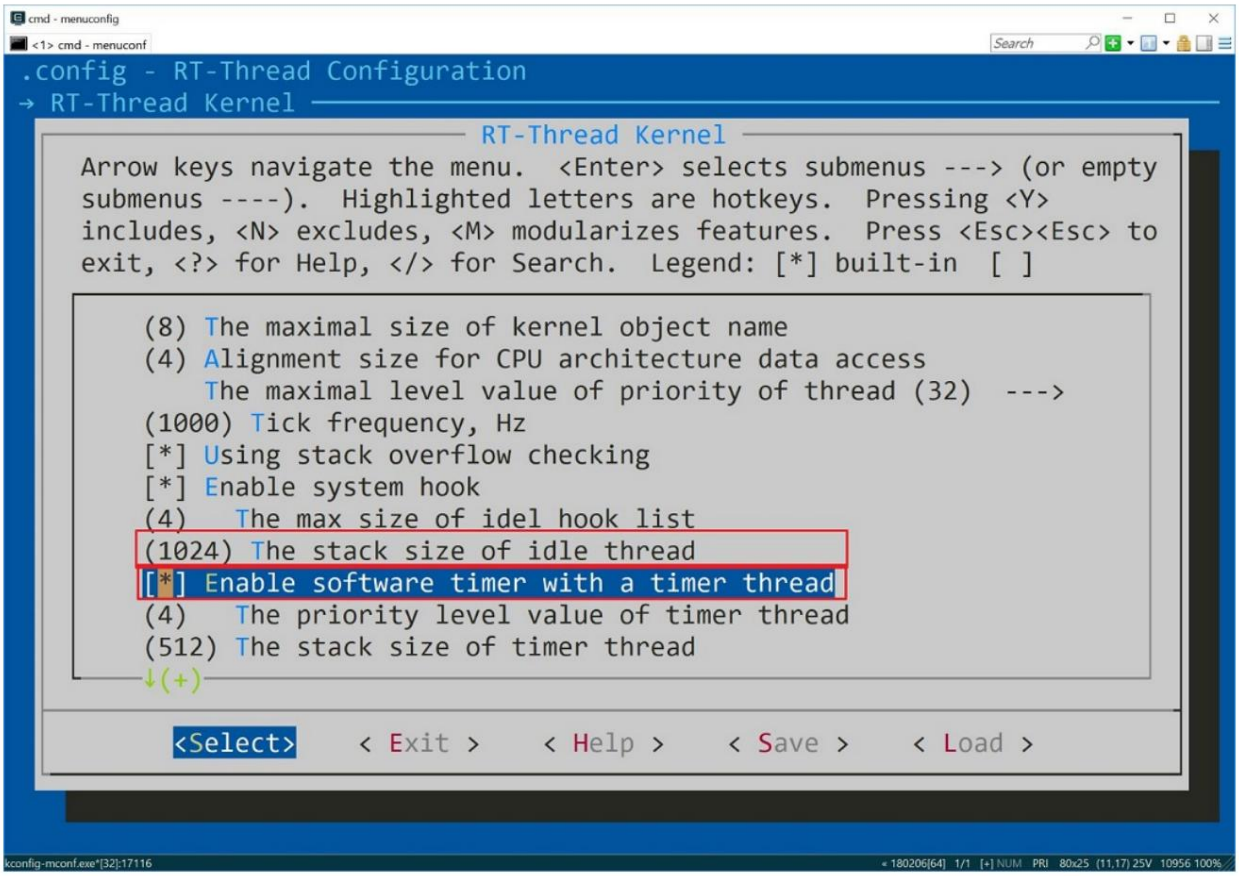

figure **2:** Configuring Kernel Options

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

When we open the mdk5 project, we can see that the corresponding source code has been added:

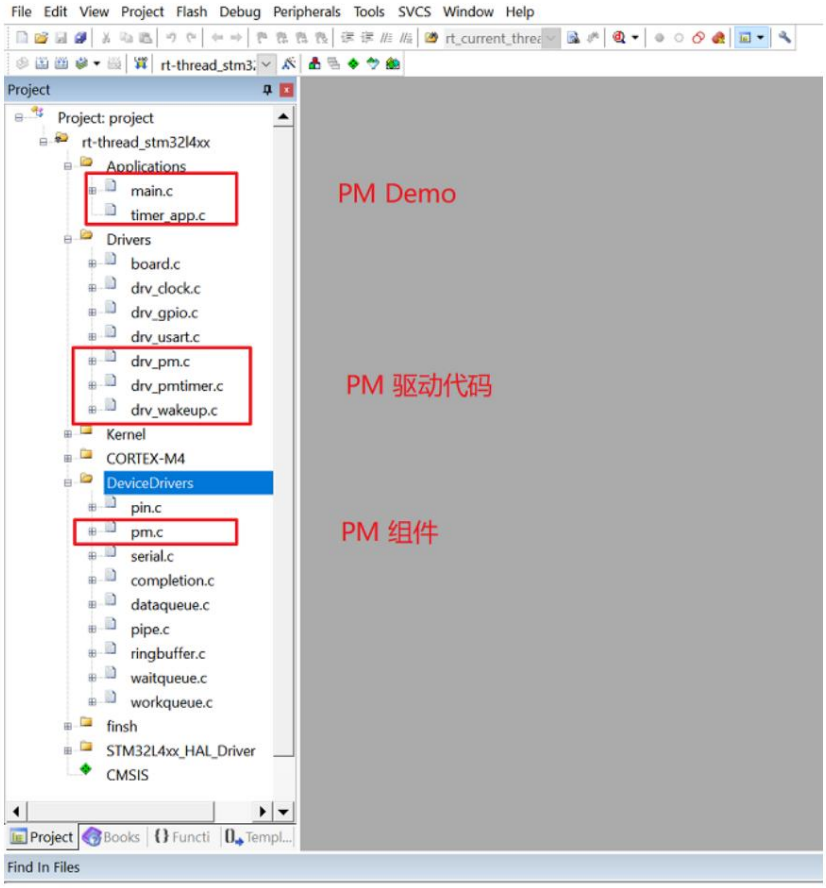

Figure **3:** *MDK* project

3 Example Description

This section introduces the timing application routine and the key wake-up routine.

The purpose of designing the timing application routine is to let everyone know that when implementing hardware-independent upper-layer applications in the PM component, there is basically no need to Too many low power consumption things. The purpose of the button wake-up routine is to let everyone understand how the applications related to the PM component are implemented.

3.0.1. Timer application **(timer_app)**

Periodic execution of tasks is a very common requirement. We use software timers to accomplish this. We will create a periodic software timer, and the timer timeout function will output the current OS Tick value.

In order for OS Tick to work properly in sleep mode, we hope that the sleep mode we enter also has a sleep timer. PM_SLEEP_MODE_TIMER mode in IoT Board can meet the requirements, so after successfully creating the software timer, we use rt_pm_request(PM_SLEEP_MODE_TIMER) to request the TIMER sleep mode. The following is the sample 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 you can see the timed output log:

```
 \ | /
-RT-     Thread Operating System
 / | \ 3.1.0 build Sep 7 2018
 2006 - 2018 Copyright by rt-thread team
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. In the mode list of pm_dump, the priority is from high to low.

To the lowest order, so now it is in Running Mode. Running Mode, Sleep Mode and Shutdown Mode are all requested once by default when starting. Timer Mode is requested once in the timer application.

We first enter the command pm_release 0 to manually release the Running Mode, then enter the command pm_release 1 to manually release the Sleep Mode; finally the PM component will enter the Timer Mode. In the Timer Mode, when the software timer times out, the chip will be awakened. So we 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:

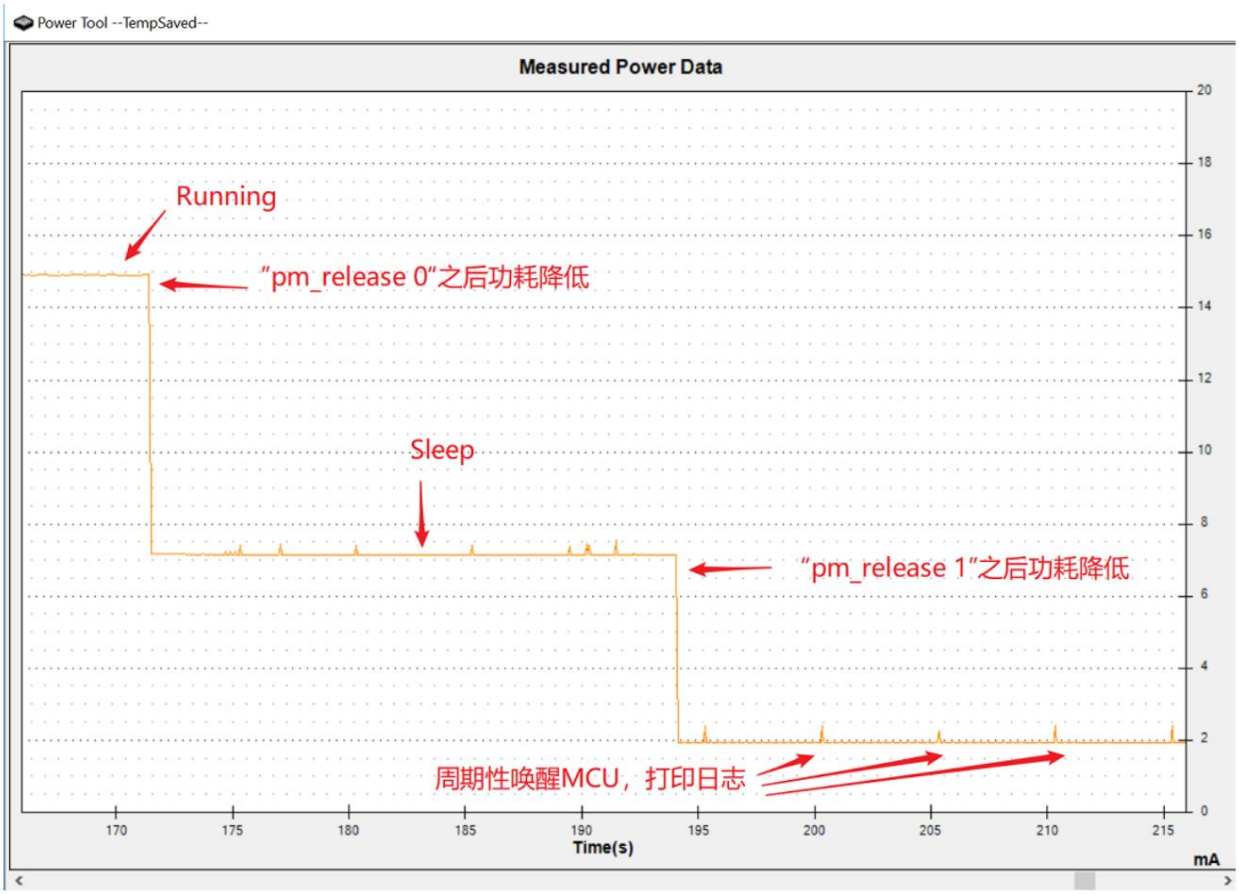

Figure 4: Power consumption changes

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

3.0.2. Press button to wake up the application

According to actual needs, waking up the chip from sleep mode to complete tasks is also a common low-power scenario. Chips usually support various wake-up methods, such as timed wake-up, peripheral interrupt wake-up, wake-up pin wake-up, etc. We will show how the PM component completes wake-up related applications based on key wake-up.

In the button wake-up application, we use the wakeup button to wake up the MCU in sleep mode. After the MCU is awakened, the corresponding wakeup interrupt will be triggered. The following routine is to use the wakeup button to wake up the MCU from Timer MODE and light up the LED for 2 seconds before entering sleep mode again.

The entry point of the routine is in the main() function:

```
int main(void) {

    /* wakeup event and callback init */ wakeup_init();


    /* pm mode init */
    pm_mode_init();

    while (1) {

        /* wait for wakeup event */ if
        (rt_event_recv(wakeup_event,
                                WAKEUP_EVENT_BUTTON,
                                RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
                                RT_WAITING_FOREVER, RT_NULL) == RT_EOK)
        {
            led_app();
        }
    }
}
```

The main() function first completes the initialization work: including the initialization of the wake-up function and the configuration of the PM mode; then it waits in a loop Wait for the event sent in the interrupt, if an event is received, execute led_task() once.

The initialization of the wakeup function wakeup_init() includes the initialization of the wakeup event and the initialization of the callback function in the wakeup interrupt:

```
static void wakeup_init(void) {

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

    bsp_register_wakeup(wakeup_callback);
}
```

PM_SLEEP_MODE_TIMER corresponds to the STM32L475's STOP2 mode, and LPTIM1 is turned on before entering. We want to stay in PM_SLEEP_MODE_TIMER mode, so we first need to call rt_pm_request() once to request this mode.

Since at the beginning, PM_SLEEP_MODE_SLEEP and PM_RUN_MODE_NORMAL modes are already set by default when the PM component is started. Requested once. In order not to stay in these two modes, we need to call rt_pm_release() to release them:

```
static void pm_mode_init(void) {

    rt_pm_request(PM_SLEEP_MODE_TIMER);
    rt_pm_release(PM_SLEEP_MODE_SLEEP);
    rt_pm_release(PM_RUN_MODE_NORMAL);
}
```

In led_app(), we want to light up the LED and delay it long enough for us to observe the phenomenon. During the delay, the CPU may be idle and will go to sleep if no run mode is requested. So we request PM_RUN_MODE_NORMAL and release it after the LED flashing is completed:

```
static void led_app(void) {

    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_mdelay(2000);
    rt_pin_write(PIN_LED_R, 1);
    _pin_as_analog();

    rt_pm_release(PM_RUN_MODE_NORMAL);
}
```

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 going back to sleep. The following is a screenshot of Power Monitor running during the wake-up process:

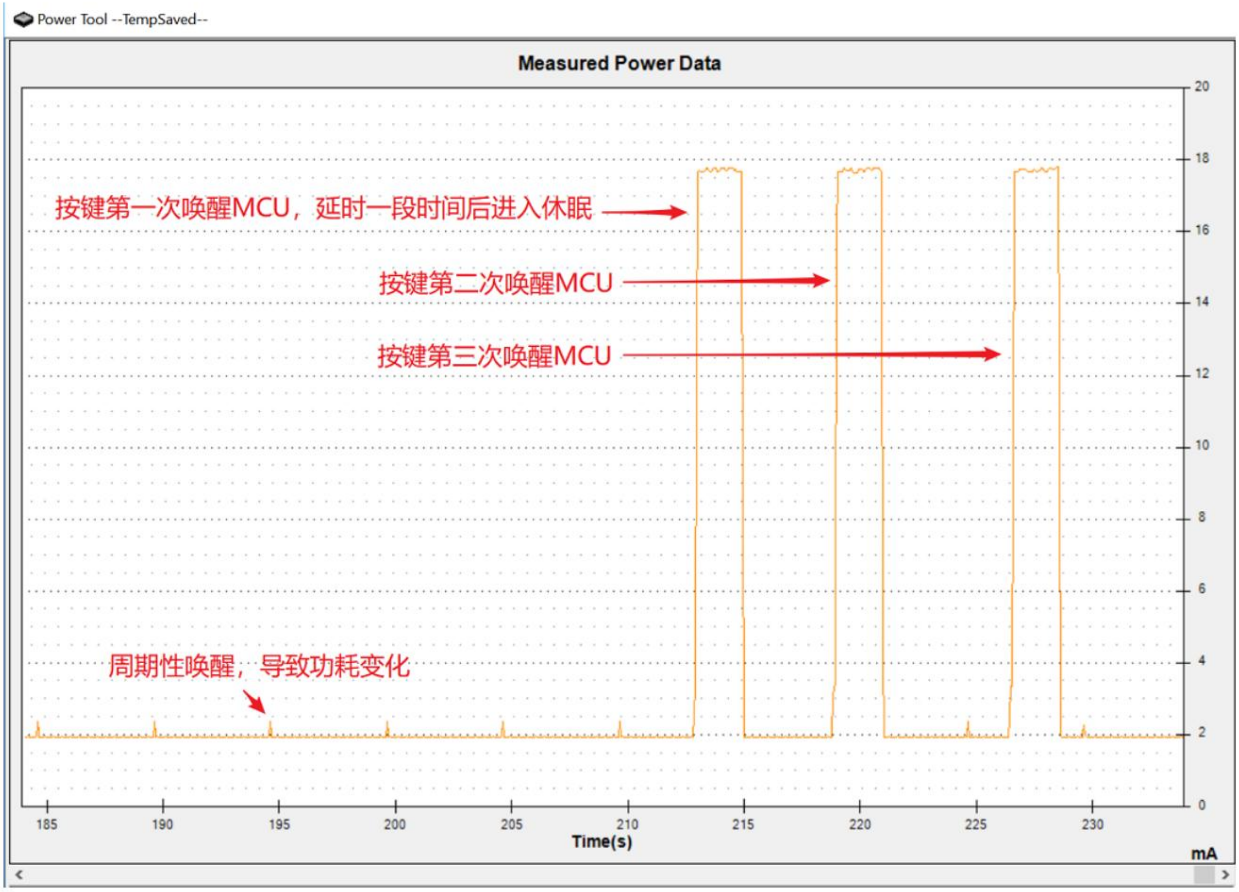

Figure 5: Power consumption changes

4 Working Principle

4.1 Chip operating frequency and sleep mode

Why does RT-Thread's PM component need to be managed based on the mode? How did it evolve? This section will start with the chip. Introduce the design of PM components of RT-Thread.

MCU usually provides multiple clock sources for users to choose. For example, STM32L475 can choose internal clocks such as LSI/MSI/HSI, and external clocks such as HSE/LSE. MCU usually also integrates PLL (Phase-locked loops), using different clock sources to provide higher frequency clocks to other modules of MCU.

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 only the clock and sleep situation 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 application can choose to let the chip run in high-performance, normal performance or extremely low-performance mode according to the needs of the task; when there is no task to be processed, the chip can be put into sleep mode. The sleep mode can choose to stop different peripherals and support different peripherals to wake up in sleep mode.

4.2 What is power consumption?

The previous section introduced that the chip can run at different frequencies and enter different sleep modes.
Shown as follows:

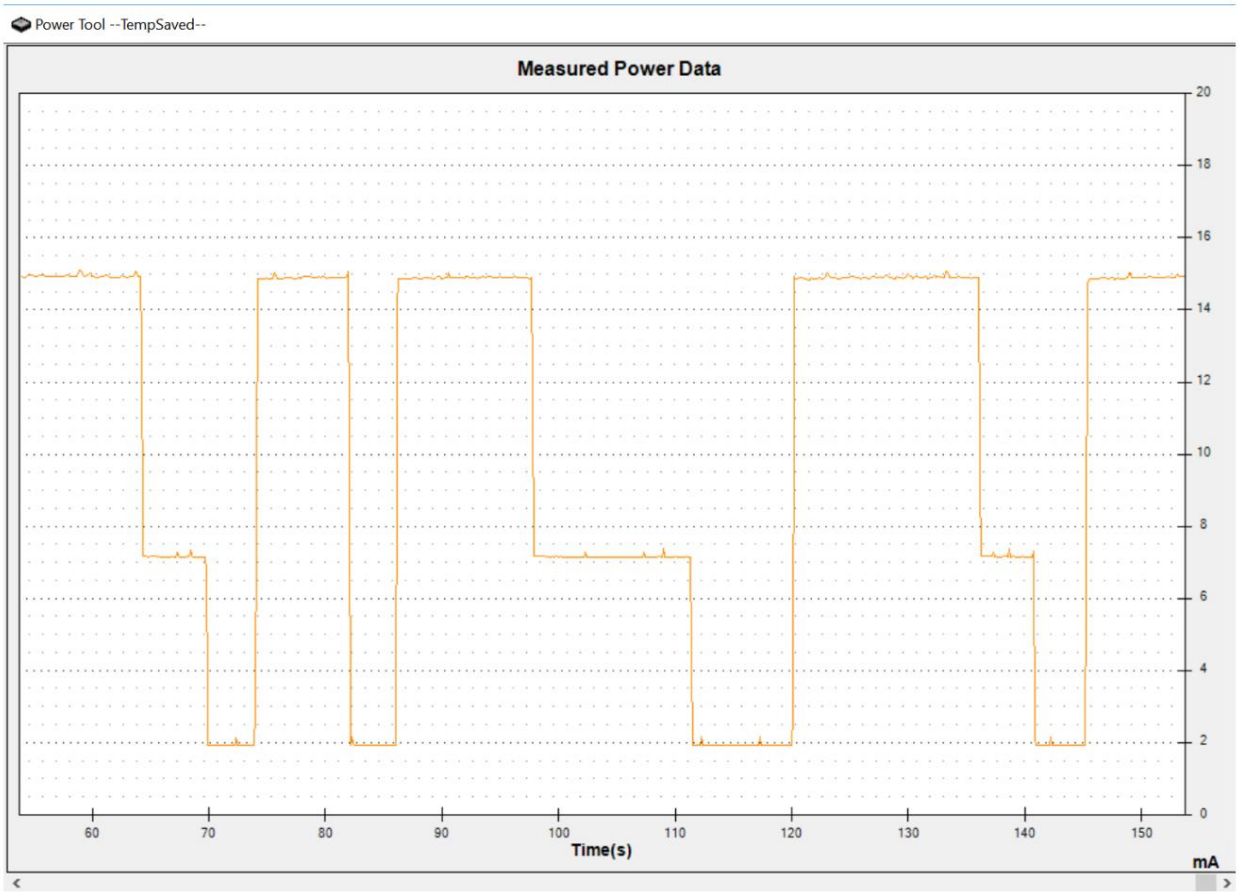

Figure 6: Power consumption changes

The horizontal axis of the graph is time, and the vertical axis is the current power of the chip. The power consumption is the area surrounded by the horizontal and vertical axes and the power curve. Reducing power consumption means making the corresponding area smaller. If a task can be completed at a lower frequency or in a lower sleep mode, the power consumption can be reduced; if it can be completed in a shorter time, the power consumption can also be reduced.

5. Dive Deep into PM Components

5.1 Definition of Pattern

We call the MCU clock frequency and sleep mode a mode. If the CPU is still working in the current mode, it is called running.
If the CPU stops working, it is called sleep mode.

In the running mode, the CPU is still running. We can subdivide different running modes according to the running frequency. In the sleep mode, the CPU has stopped working. We can subdivide different sleep modes according to whether different peripherals are still working. Different MCUs can customize different modes according to actual conditions.

5.2 Change of Mode

The PM component provides two APIs for users to decide the PM mode, one is rt_pm_request() for requesting the mode, and the other is rt_pm_release() for releasing the requested mode. The PM component records user requests and automatically handles the switching of operating frequency and sleep mode based on these requests using a veto mechanism.

5.3 Model veto

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 PM component, as long as a mode requests a higher priority mode, it will not switch to a lower priority mode. This is the veto of the mode.

5.4 Timing of Mode Change

Generally speaking, the PM model is implemented in the driver as much as possible, so that developers of ordinary applications do not need to worry about this part.

In the driver, if the peripheral does not want to enter low power mode when working, it needs to be adjusted before it works. Use rt_pm_request() to request the corresponding operating mode, and call rt_pm_release() to release it in time after completing the work.

For example, 100M Ethernet requires at least a 25M external clock, and they may share a PLL with the CPU. It is hoped that the Ethernet driver can request to work in a higher operating mode.

If it is not a low-power application, you don't need to worry about the mode.

5.5 PM Implementation

In PM, each mode has its own counter. If the value of a mode is not 0, it means that at least one request wants to work in this mode. According to the one-vote veto principle, the PM component chooses the highest mode to run, which is the first mode with a non-zero counter in the PM implementation.

If the user calls rt_pm_request(PM_RUN_MODE_XXX), the counter of PM_RUN_MODE_XXX mode will increase by one. If the mode is higher than the current mode, it will switch to the new mode immediately and the current mode will be modified to the new mode.

If rt_pm_release(PM_RUN_MODE_XXX) is called, the counter of PM_RUN_MODE_XXX mode will be reduced by 1. If the released mode is the current mode, and the counter value of the current mode becomes 0, it means that it is possible to switch to a lower mode. In the implementation of PM, this switch is not performed immediately, but is completed by calling rt_pm_enter() in IDLE HOOK when all tasks are idle.

Switching between modes may be different in different BSPs, so it is necessary to adapt to different hardware. For details, please refer to the porting instructions.

When the mode is switched, the peripherals may also be affected, so the PM component provides PM device functions that are sensitive to mode changes. Please refer to the next section for details.

5.6 PM devices sensitive to mode changes

In the PM component, switching to a new operating mode may cause the CPU frequency to change. If the peripherals and the CPU share a part of the clock, the peripheral clock will be affected; when entering a new sleep mode, most clock sources will be stopped. If the peripheral does not support the freeze function of sleep, then when waking up from sleep, the peripheral clock needs to be reconfigured. Therefore, the PM component supports PM devices that are sensitive to PM mode. Each PM device needs to implement the following functions:

```
struct rt_device_pm_ops { #if

PM_RUN_MODE_COUNT > 1 void
    (*frequency_change)(const struct rt_device* device); #endif


    void (*suspend)(const struct rt_device* device); void (*resume) (const
    struct rt_device* device);
};
```

Among them, frequency_change() is called when switching to a new operating mode. Suspend() is called before entering sleep mode, and resume() is called after waking up from sleep mode. After implementing the corresponding functions, we can register it to the PM component through APIrt_pm_register_device() . If you need to overflow this device, you can cancel the registration APIrt_pm_unregister_device() to complete it.

5.7 PM device interface

Generally, we use it directly through the API of PM component. At the same time, PM component also provides device interface upward, so

We can use rt_device_read, rt_device_write, and rt_device_control to use the PM component.

If the RT_USING_DFS_DEVFS option is turned on, access can also be based on the use of files.

6Transplantation Instructions

6.1 Basic transplantation

This section describes how to migrate PM components to the new BSP.

The underlying functions of the PM component are all completed through the functions in the struct rt_pm_ops structure:

```
struct rt_pm_ops {

    void (*enter)(struct rt_pm *pm); void (*exit)
    (struct rt_pm *pm);

#if PM_RUN_MODE_COUNT > 1
    void (*frequency_change)(struct rt_pm *pm, rt_uint32_t frequency);
#endif

    void (*timer_start)(struct rt_pm *pm, rt_uint32_t timeout); void (*timer_stop)(struct rt_pm
    *pm); rt_tick_t (*timer_get_tick)(struct rt_pm *pm);
```

```
};
```

To support PM components in the new BSP, you only need to implement the functions in struct rt_pm_ops and then use rt_system_pm_init () to complete the initialization work.

The following is a sample code for using rt_system_pm_init():

```
static int drv_pm_hw_init(void) {

    static const struct rt_pm_ops _ops = {

        _drv_pm_enter,
        _drv_pm_exit,
#if PM_RUN_MODE_COUNT > 1
        _drv_pm_frequency_change,
#endif
        _drv_pm_timer_start,
        _drv_pm_timer_stop,
        _drv_pm_timer_get_tick
    };

    rt_uint8_t timer_mask;

    /* initialize timer mask */
    timer_mask = 1UL << PM_SLEEP_MODE_TIMER;

    /* initialize system pm module */
    rt_system_pm_init(&_ops, timer_mask, RT_NULL);

    return 0;
}
```

The above code saves all related functions in the _ops variable and puts those functions that contain timer functions in the variable timer_mask. The position corresponding to the mode is 1, and finally rt_system_pm_init() is called to complete the initialization work.

The functions in **struct** rt_pm_ops that must be implemented are enter() and exit(). If the automatic frequency conversion function is not enabled, the frequency_change() function does not need to be implemented. If the timer function is not included in any mode, the timer_start(), timer_stop(), and timer_get_tick() functions do not need to be implemented .

The next section will introduce their specific implementations one by one according to the API.

6.1.0.1. Migration of **_drv_pm_enter()** and **_drv_pm_exit()** functions The function that needs to be completed in the _drv_pm_enter() function is to switch to the clock configuration of the new running mode or enter a new sleep mode according to the current mode.

The function that needs to be completed in the _drv_pm_exit() function is to complete the cleanup work of the mode exit. If there is no cleanup required, nothing needs to be done.

_drv_pm_enter() and _drv_pm_exit() functions are called when the PM component changes mode. There are two cases of PM component mode change : one is requesting a higher mode than the current mode in rt_pm_request(), and the other is decreasing to a lower mode than the current mode in rt_pm_enter().

Each time the mode changes, the PM component calls '_drv_pm_exit() to exit the current mode, then updates the mode variable pm-> current_mode, and finally calls _drv_pm_exit() to switch to the new mode.

Therefore, the _drv_pm_enter() and _drv_pm_exit() functions need to make different judgments based on the value of the current mode. The following are Implementation of _drv_pm_enter() in STM32L475 :

```
static void _drv_pm_enter(struct rt_pm *pm) {

    RT_ASSERT(pm != RT_NULL);
    switch (pm->current_mode) {

    case PM_RUN_MODE_NORMAL:
        break;

    case PM_SLEEP_MODE_SLEEP:
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
        break;

    case PM_SLEEP_MODE_TIMER:
        HAL_PWREx_EnterSTOP2Mode(PWR_STOPENTRY_WFI);
        break;

    case PM_SLEEP_MODE_SHUTDOWN:
        HAL_PWREx_EnterSHUTDOWNMode();
        break;

    default:
        RT_ASSERT(0);
        break;
    }
}
```

Since the BSP has only one operating mode and the clock has been configured when the system is started,

PM_RUN_MODE_NORMAL does not require any action. The remaining three sleep modes only require you to select different sleep modes according to the actual situation. Sleep mode.

The following is the implementation of _drv_pm_exit() in STM32L475 :

```
static void _drv_pm_exit(struct rt_pm *pm) {

    RT_ASSERT(pm != RT_NULL);
    switch (pm->current_mode) {

    case PM_RUN_MODE_NORMAL:
        break;

    case PM_SLEEP_MODE_SLEEP:
        break;

    case PM_SLEEP_MODE_TIMER:
```

```
            rt_update_system_clock();
            break;

        case PM_SLEEP_MODE_SHUTDOWN:
            break;

        default:
            RT_ASSERT(0);
            break;
        }
}
```

Since PM_SLEEP_MODE_SLEEP does not affect any peripherals, we do not need to do anything in this mode. We hope to use rt_kprintf() to output debugging information immediately after waking up from PM_SLEEP_MODE_TIMER, so we update the system clock immediately after waking up so that non-low-power uart can work normally.

Of course, if you want rt_kprintf() to work immediately after waking up, you can register this uart as a PM that is sensitive to mode changes. Device. For specific registration methods, please refer to the following chapters.

6.1.0.2. _drv_pm_timer_xxx() Before the PM component enters a sleep mode that includes a timer function, it calls the _drv_pm_timer_start() function according to the next wake-up time to enter sleep mode. Therefore, _drv_pm_timer_start() needs to complete the configuration of the sleep mode timer to ensure that it can wake up at the specified time, which is the timeout OS Tick.

```
void _drv_pm_timer_start(struct rt_pm *pm, rt_uint32_t timeout);
```

The chip is awakened after a period of sleep, which may be due to the timeout interrupt of the sleep mode timer or other peripheral interrupts. Therefore, the PM component will call the _drv_pm_timer_get_tick() function after the chip is awakened to get the actual sleep time. Finally, the PM component will call the _drv_pm_timer_stop() function to stop the timer.

The _drv_pm_timer_start() function and _drv_pm_timer_get_tick() both use OS Tick as the unit to determine the sleep time. However, there may be errors in the conversion between OS Tick and the sleep mode timer. Users can decide to ignore this error based on actual conditions, or correct the error based on the value of multiple OS Ticks.

The following is the implementation of _drv_pm_timer_start() in STM32L475 :

```
static void _drv_pm_timer_start(struct rt_pm *pm, rt_uint32_t timeout) {

    RT_ASSERT(pm != RT_NULL);
    RT_ASSERT(timeout > 0);

    /* Convert OS Tick to pmtimer timeout value */ timeout =
    stm32l4_pm_tick_from_os_tick(timeout); if (timeout >
    stm32l4_lptim_get_tick_max()) {

        timeout = stm32l4_lptim_get_tick_max();
    }

    /* Enter PM_TIMER_MODE */
    stm32l4_lptim_start(timeout);
}
```

This function first converts the value of the timeout variable from OS Tick to the Tick value of the low power timer in PM mode, and then determines this value The maximum access of the low power timer was not exceeded and the low power timer was finally turned on.

The following is the implementation of _drv_pm_timer_stop() in STM32L475 :

```
static void _drv_pm_timer_stop(struct rt_pm *pm) {

    RT_ASSERT(pm != RT_NULL);

    /* Reset pmtimer status */
    stm32l4_lptim_stop();
}
```

This function simply stops the timer.

The following is the implementation of _drv_pm_timer_stop() in STM32L475 :

```
static rt_tick_t _drv_pm_timer_get_tick(struct rt_pm *pm) {

    rt_uint32_t timer_tick;

    RT_ASSERT(pm != RT_NULL);

    timer_tick = stm32l4_lptim_get_current_tick();

    return stm32l4_os_tick_from_pm_tick(timer_tick);
}
```

In this function, the Tick value of the low power timer in PM mode is first obtained, and then converted into OS Tick. stm32l4_lptim_get_current_tick() is just a simple conversion. In the stm32l4_os_tick_from_pm_tick() function , the cumulative error is repaired:

```
static rt_tick_t stm32l4_os_tick_from_pm_tick(rt_uint32_t tick) {

    static rt_uint32_t os_tick_remain = 0; rt_uint32_t ret,
    freq;

    freq = stm32l4_lptim_get_countfreq(); ret = (tick *
                        RT_TICK_PER_SECOND + os_tick_remain) / freq;

    os_tick_remain += (tick * RT_TICK_PER_SECOND);
    os_tick_remain %= freq;

    return ret;
}
```

Each time this function changes lines, it saves the remainder of the conversion to the os_tick_remain variable and uses it in the next OS Tick to PM Tick's career change.

6.1.0.3. _drv_pm_frequency_change() PM component will call this function every time it switches to a new operation mode. So in the driver we can complete the CPU frequency adjustment of the chip in this function.

6.2 Support for PM devices that are sensitive to mode changes

After completing the basic migration, the BSP can also register devices that are sensitive to PM mode changes according to actual conditions, so that the devices It can work normally when switching to a new running mode or a new sleep mode.

The functionality of a new PM device is accomplished through the functions in struct rt_device_pm_ops:

```
struct rt_device_pm_ops { #if

PM_RUN_MODE_COUNT > 1 void
    (*frequency_change)(const struct rt_device* device);
#endif

    void (*suspend)(const struct rt_device* device); void (*resume)
    (const struct rt_device* device);
};
```

When switching to a new operating mode, the frequency_change() function of the registered devices will be called one by one. Before entering sleep mode, Call the suspend() function of the registered device. After the device wakes up, the resume() function of the registered device will be called one by one.

After completing the above functions, we can use rt_pm_register_device() and rt_pm_unregister_device() to manage PM devices.

To register a PM device, you need to pass in the corresponding device pointer and the pm_ops of the corresponding device. The following is a template:

```
#if PM_RUN_MODE_COUNT > 1
void _serial1_frequency_change(const struct rt_device* device) {

    /* do something */
}
#endif
void _serial1_suspend(const struct rt_device* device) {

    /* do something */

} void _serial1_resume(const struct rt_device* device) {

    /* do something */
}

int stm32_hw_usart_init(void) {

    static struct rt_device_pm_ops _pm_ops = { #if

    PM_RUN_MODE_COUNT > 1
        _serial1_frequency_change,
    #endif
```

```
        _serial1_suspend,
        _serial1_resume,
    };
    ......
    rt_pm_register_device(&serial1, &_pm_ops);
}
```

To unregister a PM device, just pass in the corresponding device pointer:

```
rt_pm_unregister_device(&serial1);
```

7 API Description

The following table shows the APIs in all PM components:

PM component API list	Location
rt_system_pm_init()	pm.c
rt_pm_request()	pm.c
rt_pm_release()	pm.c
rt_pm_register_device()	pm.c
rt_pm_unregister_device()	pm.c
rt_pm_enter()	pm.c
rt_pm_exit()	pm.c

7.1 API Detailed Explanation

7.1.1. 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

The upper layer provides a device named "pm" and requests three modes by default, including a default operating mode PM_RUN_MODE_DEFAULT, A default sleep mode PM_SLEEP_MODE_DEFAULT and a minimum mode PM_MODE_MAX. The default operating mode and sleep mode values are They can be defined as needed, the default value is the first mode.

parameter	describe
ops	Function set of the underlying PM driver

parameter	describe
timer_mask	Specifies which modes include the low power timer
user_data	A pointer that can be used by the underlying PM driver

7.2 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

7.3 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

7.4 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

7.5 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.

7.6 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.

7.7 PM mode exit function

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

This function is called by rt_pm_enter() when waking up from sleep mode. When waking up from sleep mode, it is possible to enter the interrupt handler of the wake-up interrupt first. The user can also actively call rt_pm_exit() here. rt_pm_exit() may be called multiple times after waking up from sleep mode .