RT-THREAD Power Management User Manual

RT-THREAD Document 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 efficiency in products is becoming increasingly urgent. Sensor nodes for data collection typically need to operate for long periods of time while operating on battery power, while networking SoCs also require fast response times and low power consumption.

At the beginning of product development, the primary priority is to quickly complete product functionality. As product functionality gradually improves, power management (PM) functionality becomes necessary. To meet these IoT requirements, RT-Thread provides a power management framework. The power management framework aims to be as transparent as possible, making it easier to incorporate low-power features into 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, ensuring 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, ensuring that it can suspend and resume correctly in

different sleep modes. • The PM component supports optional sleep time compensation, allowing applications that rely on OS Tick to use

it transparently. • The PM component provides a device interface to the upper layer. If the devfs component is enabled, it can also be accessed through the file system interface.

2. Get Started Quickly

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 supporting PM components is primarily the IoT Board, with support for more platforms planned in the future. Therefore, the examples in this section are based on the IoT Board. The IoT Board is a hardware platform jointly developed by RT-Thread and Zhengdian Atom, specifically designed for the IoT field and providing a wealth of examples and documentation.

2.2 How to obtain PM components and corresponding drivers

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

1. Download IoT Board information 2.

Download ENV tool

Then copy the timer_app.c file included in 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

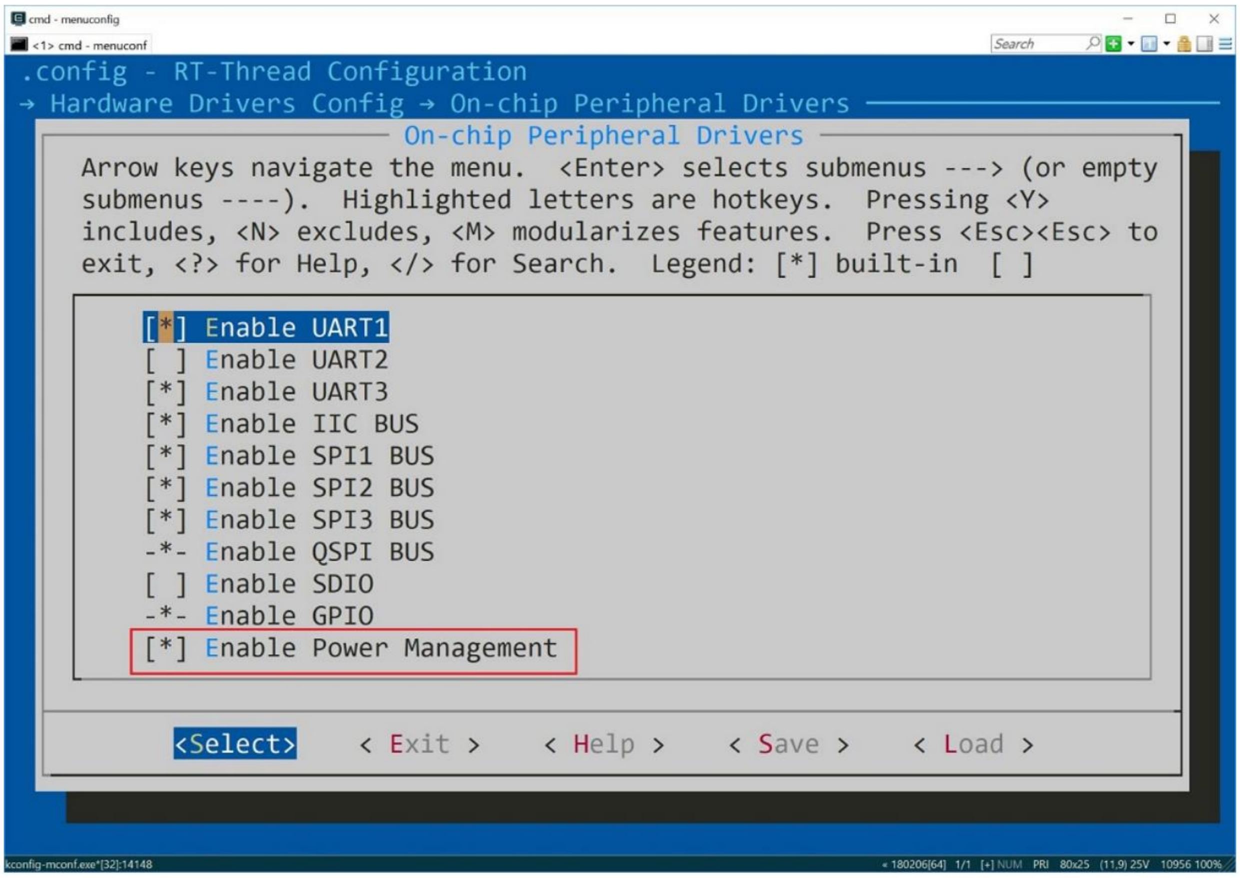
Interface configuration project:

• 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:

Machine Translated by Google



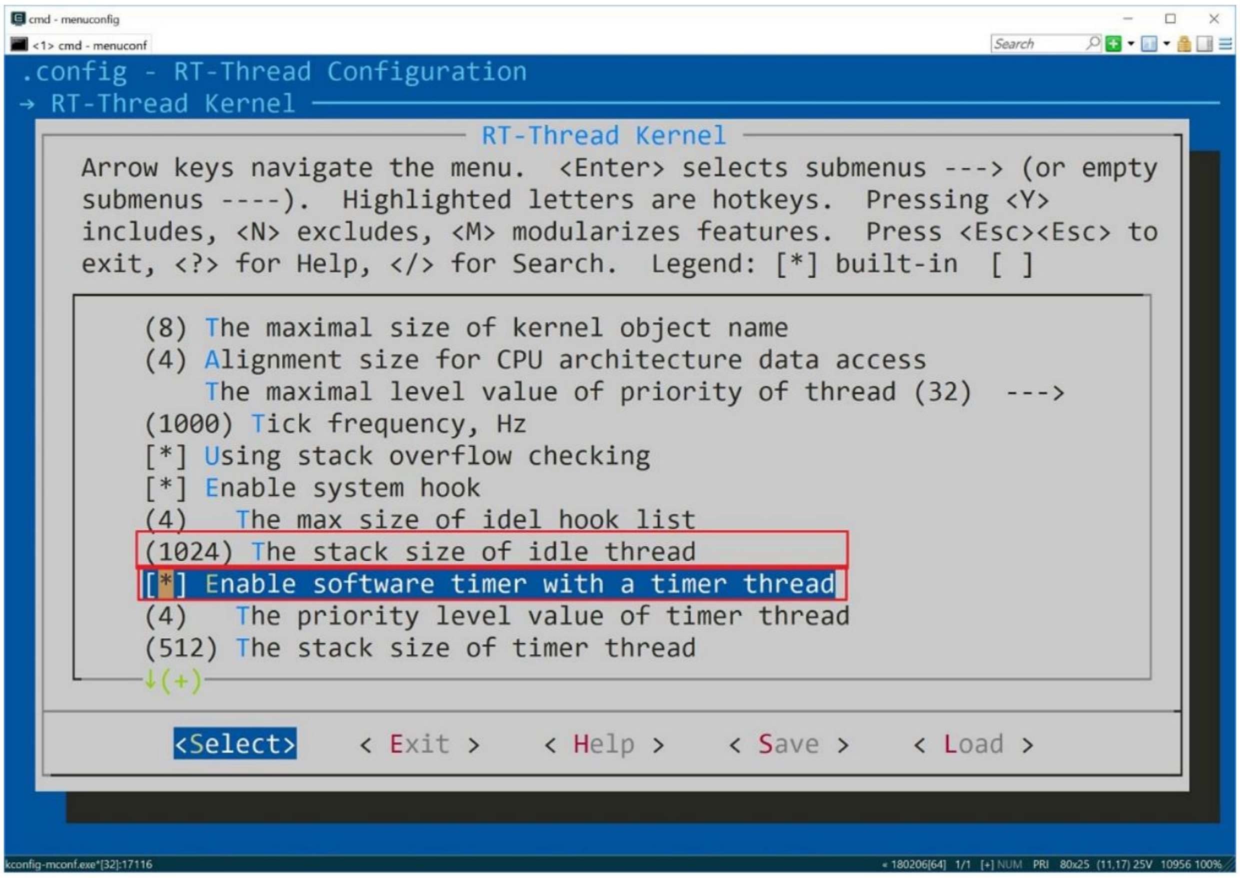
```
cmd - menuconfig
<1> cmd - menuconf
.config - RT-Thread Configuration
→ Hardware Drivers Config → On-chip Peripheral Drivers
                     On-chip Peripheral Drivers
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

        [*] Enable UART1
        [ ] Enable UART2
        [*] Enable UART3
        [*] Enable IIC BUS
        [*] Enable SPI1 BUS
        [*] Enable SPI2 BUS
        [*] Enable SPI3 BUS
        -*- Enable QSPI BUS
        [ ] Enable SDIO
        -*- Enable GPIO
        [*] Enable Power Management

        <Select>    < Exit >    < Help >    < Save >    < Load >
kconfig-mconf.exe*[32]:14148        « 180206[64]  1/1  [+] NUM  PRI  80x25  (11,9) 25V  10956 100%
```

Figure 1: Configuring Components

• Configure kernel options: Using PM components requires a larger IDLE thread stack, here we use 1024 bytes.

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

Machine Translated by Google



```
.config - RT-Thread Configuration
→ RT-Thread Kernel
                          RT-Thread Kernel
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]

      (8) The maximal size of kernel object name
      (4) Alignment size for CPU architecture data access
          The maximal level value of priority of thread (32)  --->
      (1000) Tick frequency, Hz
      [*] Using stack overflow checking
      [*] Enable system hook
      (4)   The max size of idel hook list
      (1024) The stack size of idle thread
      [*] Enable software timer with a timer thread
      (4)   The priority level value of timer thread
      (512) The stack size of timer thread
      ↓(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

cmd - menuconfig
<1> cmd - menuconf
Search
kconfig-mconf.exe*[32]:17116
« 180206[64] 1/1 [+] NUM PRI 80x25 (11,17) 25V 10956 100%

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:

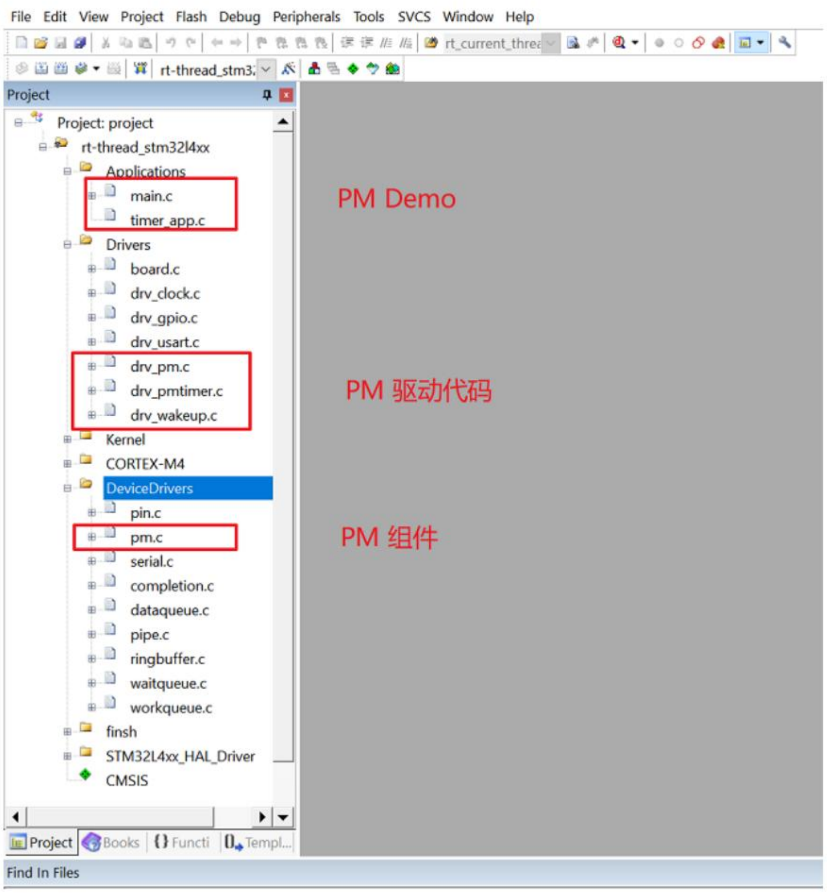


Figure 3: *MDK* project

3 Example Description

This section introduces the timing application routine and the button 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

The button wake-up routine is designed to help you understand how applications related to PM components are implemented.

3.0.1. Timer Application **(timer_app)**

Periodic task execution is a common requirement. We'll use a software timer to accomplish this. We'll create a periodic software timer whose timeout function outputs the current OS Tick value.

To ensure that the OS Tick function properly in sleep mode, we need to enter a sleep timer. PM_SLEEP_MODE_TIMER on the IoT Board meets this requirement. After successfully creating a software timer, we use rt_pm_request(PM_SLEEP_MODE_TIMER) to request 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 will 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.

The device is currently in Running Mode. Running Mode, Sleep Mode, and Shutdown Mode are all requested once by default at startup. Timer Mode is requested once in the timer application.

We first manually release Running Mode by entering the command pm_release 0 , then manually release Sleep Mode by entering the command pm_release 1. Finally, the PM component enters Timer Mode. In Timer Mode, when the software timer expires, the chip wakes up. Therefore, we see the shell continue to output:

```
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:

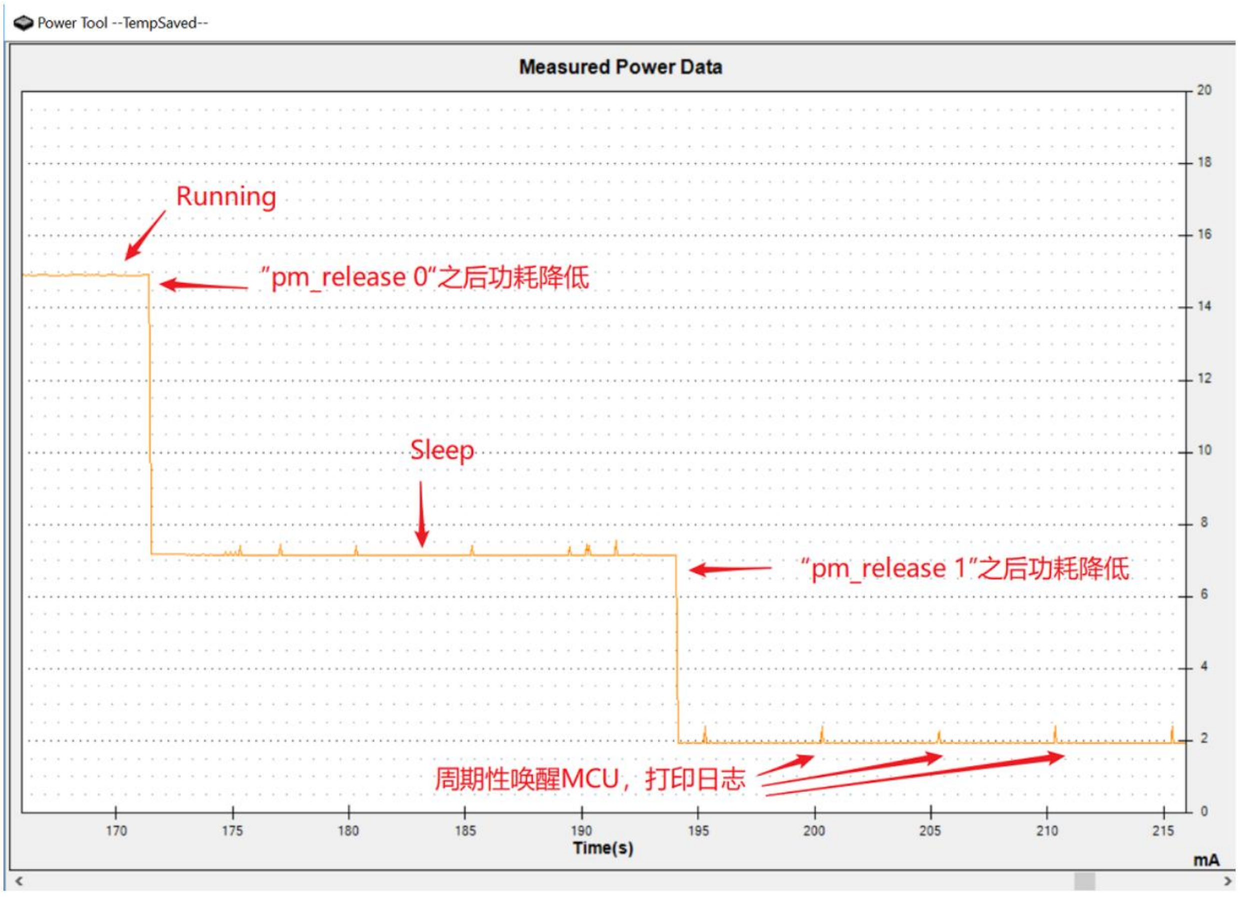


Figure 4: Power consumption changes

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

3.0.2. Button wake-up application

Waking up the chip from sleep mode to complete tasks is a common low-power scenario, depending on actual needs. Chips typically support various wakeup methods, such as timed wakeup, peripheral interrupt wakeup, and wakeup pin wakeup. This article demonstrates how the PM component accomplishes wakeup-related applications, based on keystroke wakeup.

In the button wakeup application, we use the wakeup button to wake up the MCU from sleep mode. After the MCU is awakened, the corresponding wakeup interrupt is triggered. The following example uses the wakeup button to wake up the MCU from Timer Mode , turn on the LED, and then put it back into sleep mode after 2 seconds.

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

```
int main(void) {

    /* wakup 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 until 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 STOP2 mode of STM32L475, 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 turn on the LED and delay it long enough for us to observe the phenomenon. During the delay, the CPU may be idle and will go into sleep if no run mode is requested. Therefore, 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 woken up 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:

Machine Translated by Google

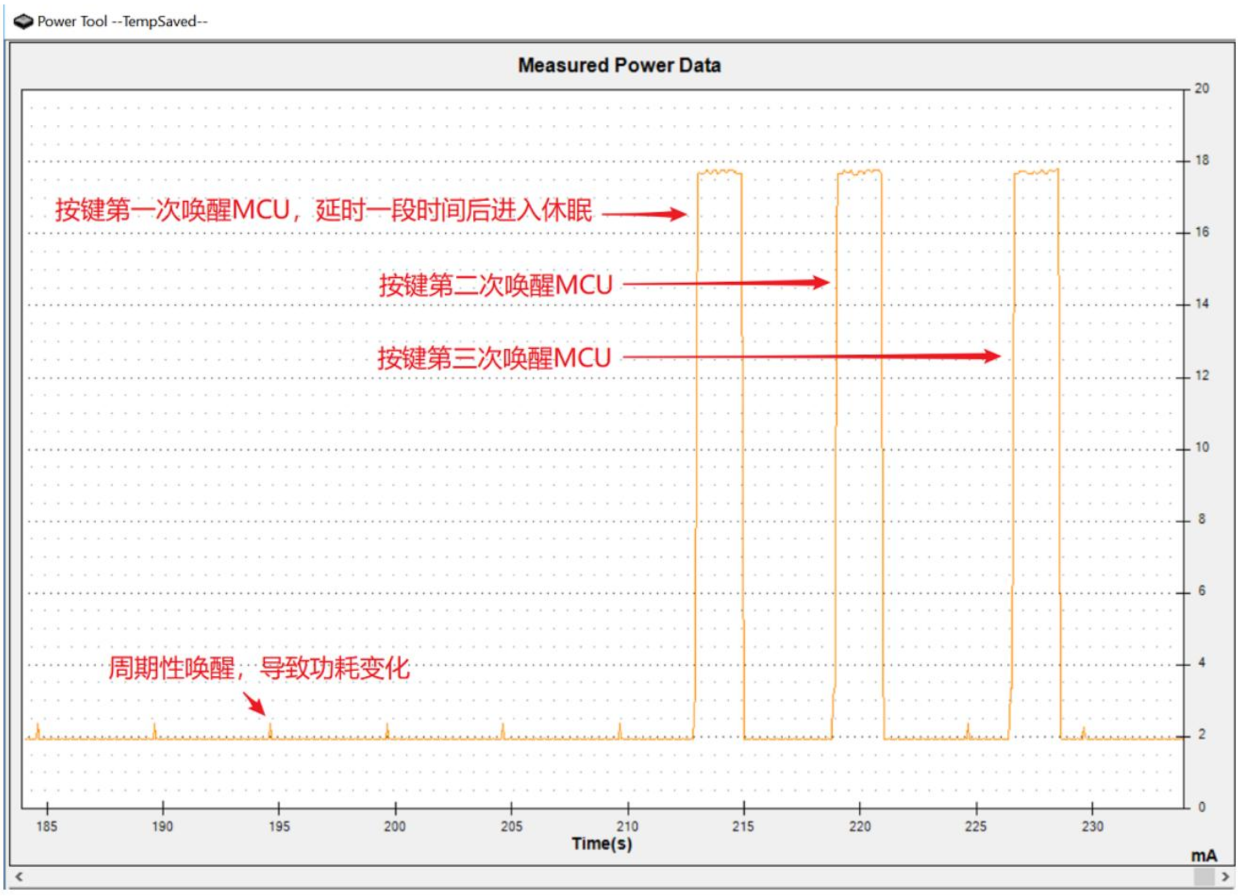


Figure 5: Power consumption changes

4 Working Principle

4.1 Chip operating frequency and sleep mode

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

MCUs typically offer a variety of clock sources for users to choose from. For example, the STM32L475 offers the option of selecting internal clocks such as LSI/MSI/HSI, as well as external clocks such as HSE/LSE. MCUs also typically integrate phase-locked loops (PLLs), which use different clock sources to provide higher-frequency clocks to other MCU modules.

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

The above only describes the clock and sleep characteristics of the STM32L475. Different MCUs can have significantly different clock and power consumption characteristics. High-performance MCUs can run at 600M or higher, while low-power MCUs can operate at 1-2M with very low power consumption.

Depending on the actual situation and the needs of the task, the application can choose to let the chip run in high-performance, normal performance or extremely low-performance mode; when there are no tasks 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:

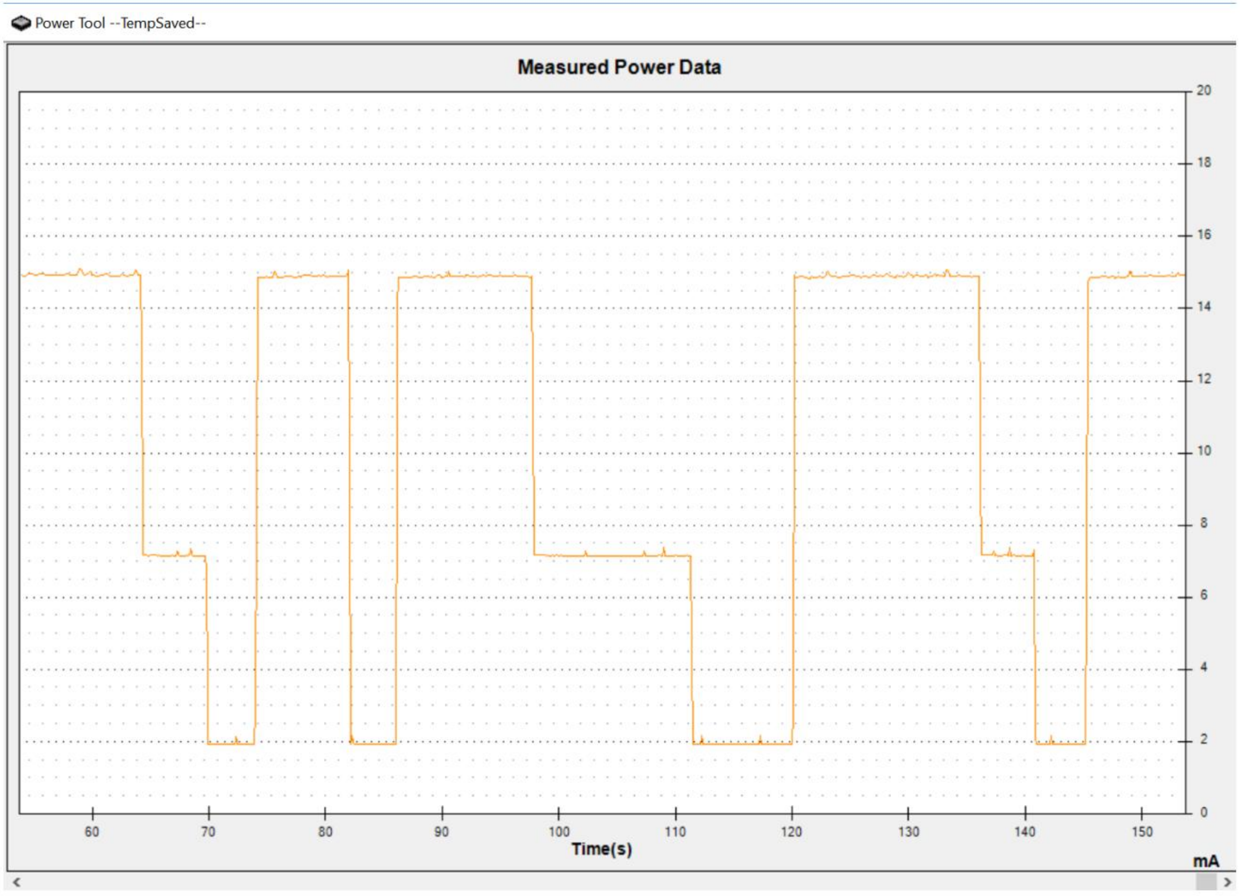


Figure 6: Power consumption changes

The horizontal axis of this graph is time, and the vertical axis is the chip's current power. Power consumption is the area enclosed by the horizontal and vertical axes and the power curve. Reducing power consumption means reducing the corresponding area. If a task can be completed at a lower frequency or in a lower sleep mode, power consumption can be reduced. If it can be completed in a shorter time, power consumption can also be reduced.

5. In-depth 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 run mode, the CPU remains operational, and different run modes can be categorized based on the operating frequency. In sleep mode, the CPU is stopped, and different sleep modes can be categorized based on whether different peripherals are still active. Different MCUs can customize different modes based on their specific needs.

5.2 Mode Change

The PM component provides two APIs for users to determine the PM mode: rt_pm_request() for requesting a mode, and rt_pm_release() for releasing the requested mode. The PM component records user requests and automatically handles switching between 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 Thread C and Thread C both request to run in normal operation mode. In this case, which mode should the PM component choose?

At this point, you should choose a mode that satisfies the needs of all threads as much as possible. High-performance mode generally performs better than normal mode. If high-performance mode is selected, threads A, B, and C will all run correctly. If normal mode is selected, thread A's needs will not be met.

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

5.4 Timing of Mode Change

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

In the driver, if the peripheral does not want to enter low power mode when it is 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, which may share a PLL with the CPU. In this case, the Ethernet driver can request to operate in a higher operating mode.

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

Implementation of 5.5 PM

In PM, each mode has its own counter. If the value of a mode is non-zero, it indicates that at least one request wishes to operate in that mode. Based on the one-vote veto principle, the PM component selects the highest-ranking mode to run. In the PM implementation, this is the mode with the first non-zero counter.

If the user calls rt_pm_request(PM_RUN_MODE_XXX), the PM_RUN_MODE_XXX mode counter 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 for the PM_RUN_MODE_XXX mode is decremented. If the released mode is the current mode and the current mode counter reaches 0, it means that a switch to a lower mode is possible. In the PM implementation, this switch is not performed immediately. Instead, it is completed by calling rt_pm_enter() in the 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 a PM device function that is sensitive to mode changes. Please refer to the next section for detailed introduction.

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 peripherals share a portion of the CPU clock, the peripheral clock will be affected. When entering a new sleep mode, most clock sources are stopped. If the peripheral does not support the freeze function during sleep, the peripheral clock will need to be reconfigured when waking from sleep. Therefore, the PM component supports PM mode-sensitive PM devices. 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);
};
```

frequency_change() is called when switching to a new operating mode. suspend() is called before entering sleep mode, and resume() is called after waking from sleep mode. After implementing the corresponding functions, we can register the device with the PM component using APIrt_pm_register_device() . If we need to overflow this device, we can unregister it using APIrt_pm_unregister_device().

5.7 PM device interface

In general, we use it directly through the API of PM component. At the same time, PM component also provides device interface to the upper level, 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.

6. Transplantation 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 an example 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 containing 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.

The mandatory functions in **struct** rt_pm_ops are enter() and exit(). If the automatic frequency conversion function is disabled, the frequency_change() function does not need to be implemented. If no mode includes the timer function, 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 _drv_pm_enter() 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 mode exit. If there is no cleanup required, nothing needs to be done.

The _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 downgrading to a lower mode than the current mode in rt_pm_enter().

Machine Translated by Google

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. 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 also 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. Specific registration methods can be found in the following chapters.

6.1.0.2. _drv_pm_timer_xxx() Before entering a sleep mode that includes a timer function, the PM component calls the _drv_pm_timer_start() function based on the next wakeup time to enter sleep mode. Therefore, _drv_pm_timer_start() must 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 wakes up after a period of sleep, possibly due to a timeout interrupt from the sleep mode timer or another peripheral interrupt. Therefore, the PM component calls the _drv_pm_timer_get_tick() function after the chip wakes up to obtain the actual sleep time. Finally, the PM component calls 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 ticks as the unit to determine the sleep time. However, there may be errors in the conversion between OS ticks and the sleep mode timer. Users can decide to ignore this error or correct it 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 limit for 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);
}
```

This function first obtains the PM mode low-power timer's tick value and then converts it to an OS tick. stm32l4_lptim_get_current_tick() simply performs the conversion.

The stm32l4_os_tick_from_pm_tick() function corrects the accumulated error:

```
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() The PM component will call this function every time it switches to a new operating mode. Therefore, in the driver, we can complete the chip

CPU frequency adjustment in this function.

6.2 Support for PM devices that are sensitive to mode changes

After completing the basic transplantation, 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, the frequency_change() function of the registered devices will be called one by one.

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
```

Machine Translated by Google

```
            _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 also requests three modes by default, including a default running mode PM_RUN_MODE_DEFAULT, A default sleep mode PM_SLEEP_MODE_DEFAULT and the lowest mode PM_MODE_MAX. The default operating mode and sleep mode values, I 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);
```

Release PM mode function is a function called in the application or driver. After calling, the PM component will not immediately enter the actual mode.

Switching does not start in rt_pm_enter(), but starts switching 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	Device function set

7.5 Unregistering 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 component, so no additional call is required.

7.7 PM Mode Exit Function

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

This function is called from rt_pm_enter() when waking from hibernation. When waking from hibernation, the system may first enter the wakeup interrupt handler. Users can also actively call rt_pm_exit() here. rt_pm_exit() may be called multiple times after waking from hibernation .