Power Management Application Notes

RT-THREAD Document 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 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 features need to be added. 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.

1.2 Structure of this paper

This article first briefly introduces how to get the power management component (PM component) of RT-Thread
Then, we will run the relevant sample code on the IoT Board. Finally, we will introduce the design ideas and principles of the PM component in depth.

2 Problem Description

The PM component can be divided into the user layer, the PM component layer, and the PM driver layer. The user layer includes application code and driver code, which use APIs to determine the chip's operating mode. The PM driver layer primarily implements PM driver support and power consumption control for PM-related peripherals. The PM component layer manages drivers and provides support for the user layer.

This application note will mainly introduce how to use the user layer, without going into detail about the component framework layer and PM driver layer. The application layer mainly focuses on the following issues:

• What are the modes in the PM component? What are the different types of modes? • How does the application manage modes based on its needs?

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

The examples in this article are all run on the IoT Board. The IoT Board is a hardware platform jointly launched by RT-Thread and Zhengdian Atom. It is designed specifically for the IoT field and provides rich examples and documentation.

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 information
2. Download RT-Thread source code 3. Download ENV tool

Machine Translated by Google

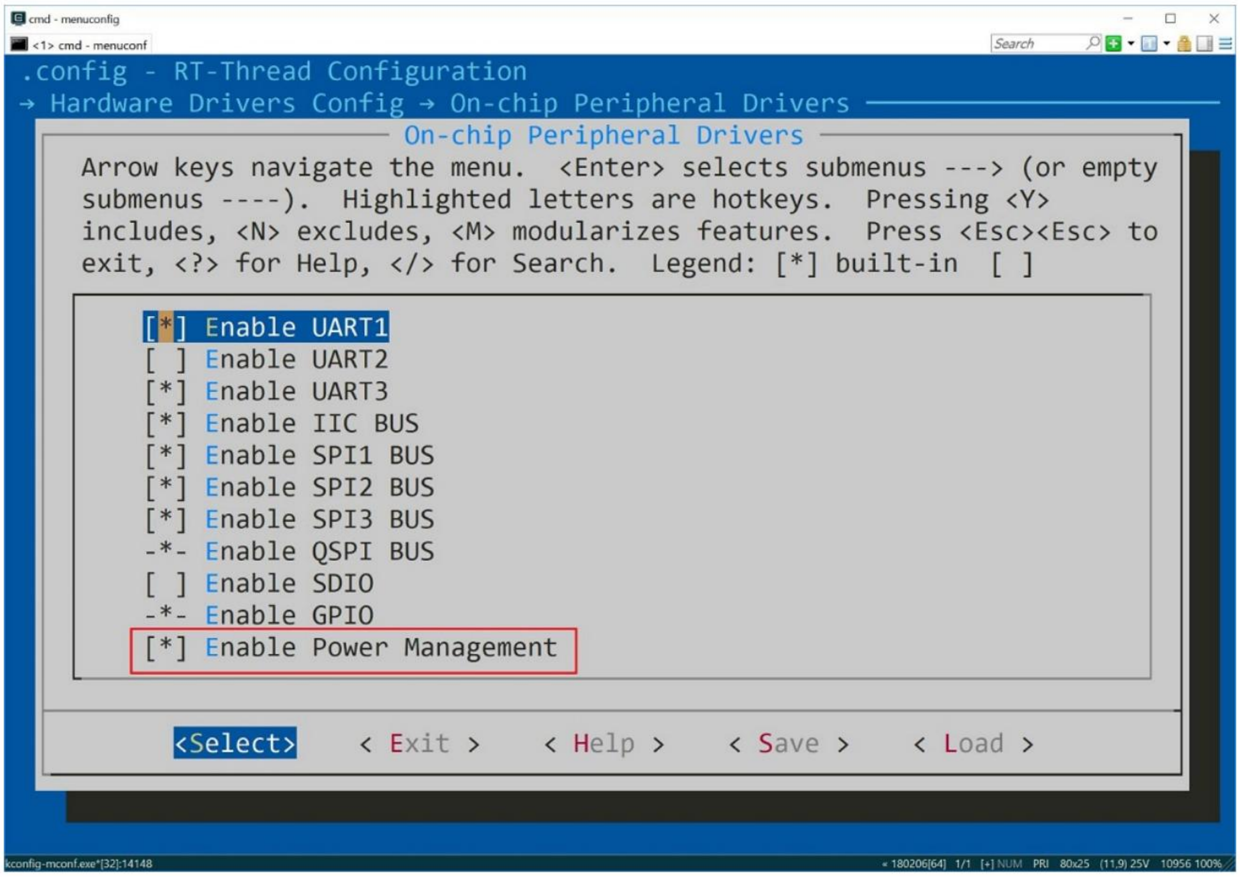
Open the env tool, enter the **PM** routine directory of IoT Board , and enter menuconfig in the env command line to enter the configuration interface.

Installation project.

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

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

able:

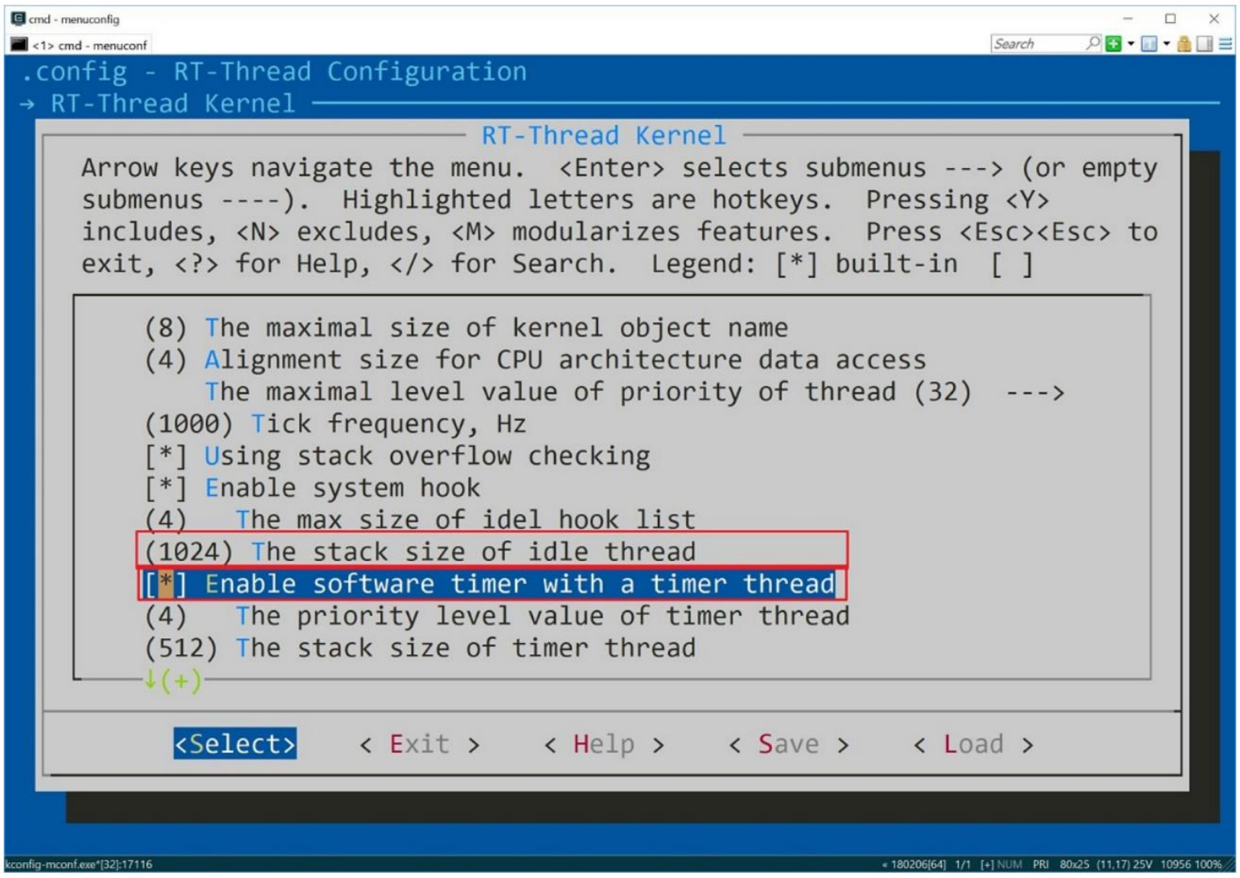


```
cmd - menuconfig
<1> cmd - menuconf
.config - RT-Thread Configuration
→ Hardware Drivers Config → On-chip Peripheral Drivers
                    On-chip Peripheral Drivers
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  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



```
.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 >
```

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:

Machine Translated by Google

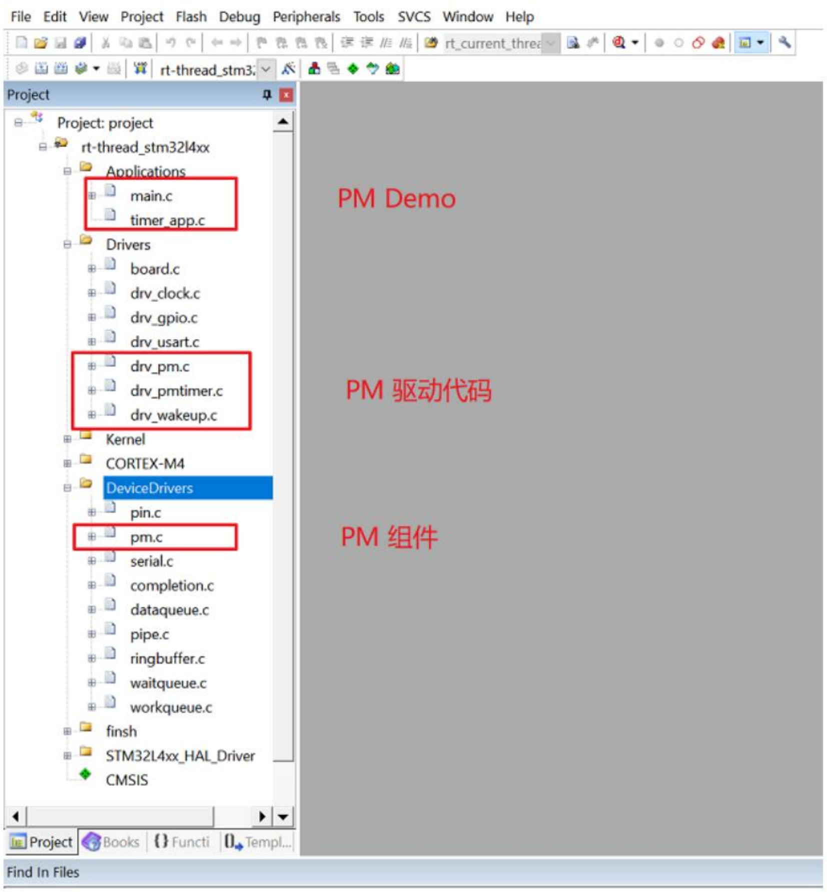


Figure 3: *MDK* project

3.2 Low Power Consumption Examples on IoT Board

3.2.1. Timer Application (timer_app)

In the timing application, we create a periodic software timer. The timer task periodically outputs the current OS tick. If the software timer is created successfully, we use rt_pm_request(PM_SLEEP_MODE_TIMER) to request the timer to enter 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 and open the terminal software. 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.

Machine Translated by Google

We enter the command 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 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:

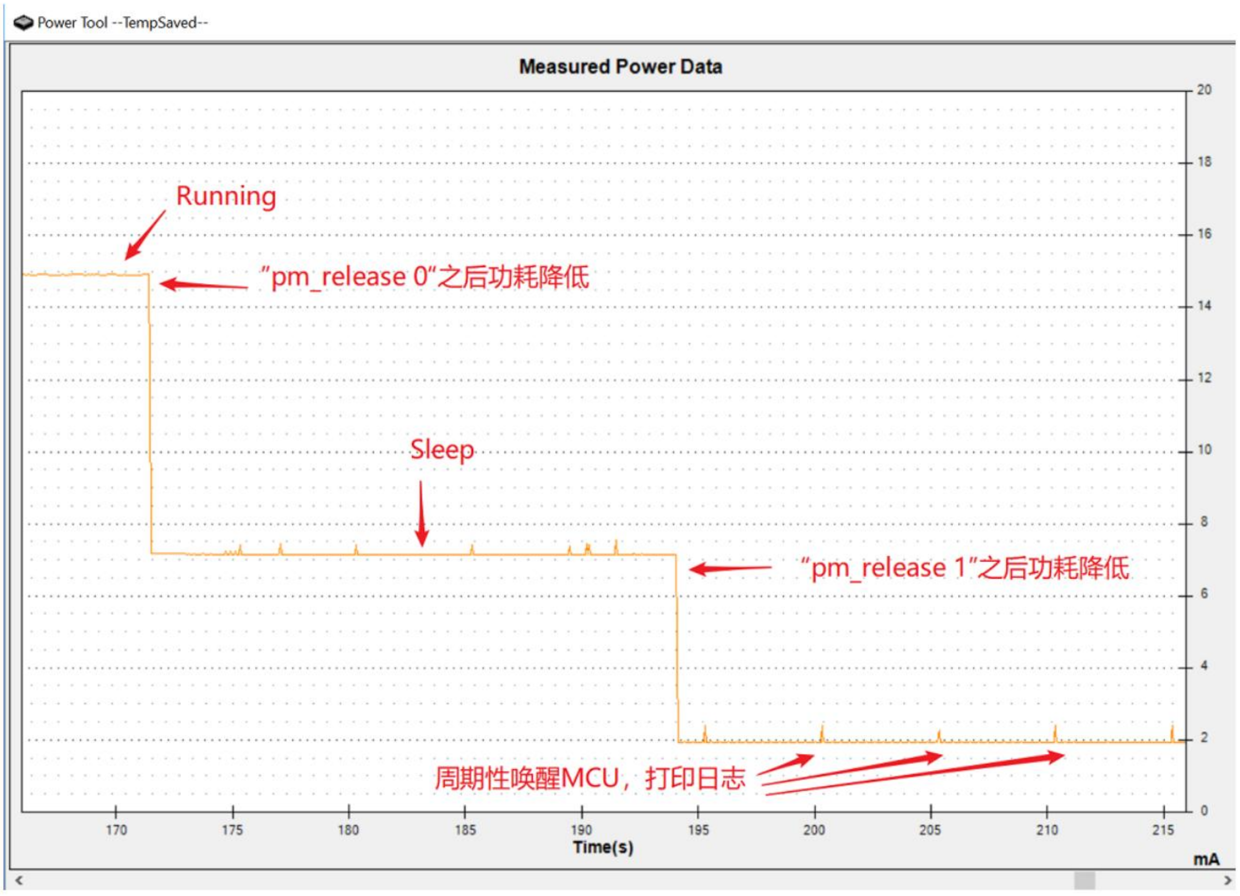


Figure 4: Power consumption changes

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

3.2.2. Button wake-up application

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

```
#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 code above, we create a thread and register a key interrupt wakeup callback function in this thread. This function is called every time a key interrupt is triggered . The callback function sends the event WAKEUP_EVENT_BUTTON. Upon receiving this event, our thread first requests Normal mode, then completes the LED blinking function and releases Normal mode.

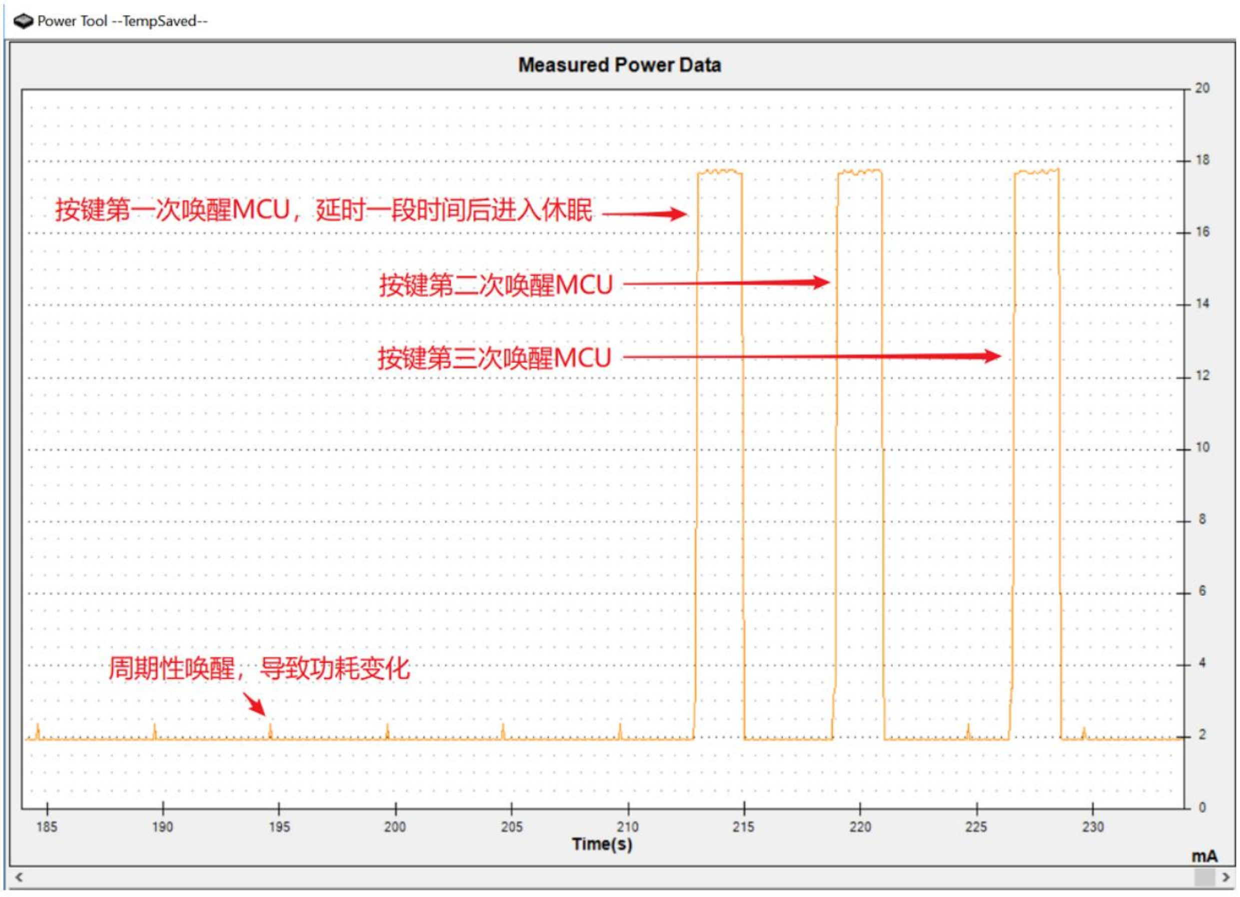


Figure 5: Power consumption changes

The above picture shows the running screenshot of pressing 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.

4. In-depth understanding of power management components

4.1 What are the modes of the power management component?

MCUs typically offer a variety of clock sources for users to choose from. For example, the STM32L475 on the IoT Board offers 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, based on different clock sources, 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 example describes the STM32L475. Different MCUs can have significantly different clock speeds and low-power consumption. High-performance MCUs can operate at 600M or higher, while low-power MCUs can operate at very low power consumption of 1-2M. Depending on the actual situation, low-power mode can selectively disable different peripherals and support waking up different peripherals from sleep mode.

To enable users to easily manage chip variations when developing low-power applications, RT-Thread's PM component manages MCU clocks and low-power modes based on their specific needs. Different MCUs define different modes, such as high-performance mode, normal mode, low-power run mode, sleep mode, timer wake-up mode, and shutdown mode. Chip power consumption is then managed as needed within the driver and application code using rt_pm_request() and rt_pm_release().

4.2 What are Run Mode and Sleep Mode? What are the differences?

Based on the mode type, PM components can be divided into run mode and sleep mode. In run mode, the CPU is still running, and different run modes are categorized based on the CPU frequency. In sleep mode, the CPU is stopped, and different sleep modes are categorized based on whether different peripherals are still functioning.

What is the veto in the 4.3 model?

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 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 Provide a device named "pm" to the upper layer, and request three modes by default, including a default RUN mode, a SLEEP mode 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);
```

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

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	Device function set

5.6 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.

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 component, 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 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.