RT-THREAD 电源管理用户手册

RT-THREAD 文档中心

WWW.RT-THREAD.ORG

Friday 19th October, 2018

目录

1 介绍

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

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

1.1 PM 组件功能特点

- PM 组件是基于模式来管理功耗
- PM 组件可以根据模式自动更新设备的频率配置，确保在不同的运行模式都可以正常工作
- PM 组件可以根据模式自动管理设备的挂起和恢复，确保在不同的休眠模式下可以正确的挂起和恢复
- PM 组件支持可选的休眠时间补偿，让依赖 OS Tick 的应用可以透明使用
- PM 组件向上层提供设备接口，如果打开了 devfs 组件，那么也可以通过文件系统接口访问

2 快速上手

本节主要展示了如何开启 PM 组件和相应的驱动，并通过例程来学习 PM 组件的使用。

2.1 选择合适的 BSP 平台

现在支持 PM 组件的 BSP 平台主要是 IoT Board，将来会支持更多的平台。所以本节的示例都是基于 IoT Board 演示。IoT Board 是 RT-Thread 和正点原子联合推出的硬件平台，该平台上专门为 IoT 领域设计，并提供了丰富的例程和文档。

2.2 如何得到 PM 组件和相应的驱动

在 IoT Board 上运行电源管理组件，需要下载 IoT Board 的相关资料和 ENV 工具：

1. 下载IoT Board 资料
2. 下载ENV 工具

然后复制本文附带的`timer_app.c`到进入 IoT Board 的 **PM 例程目录**的 application 目录里。

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

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

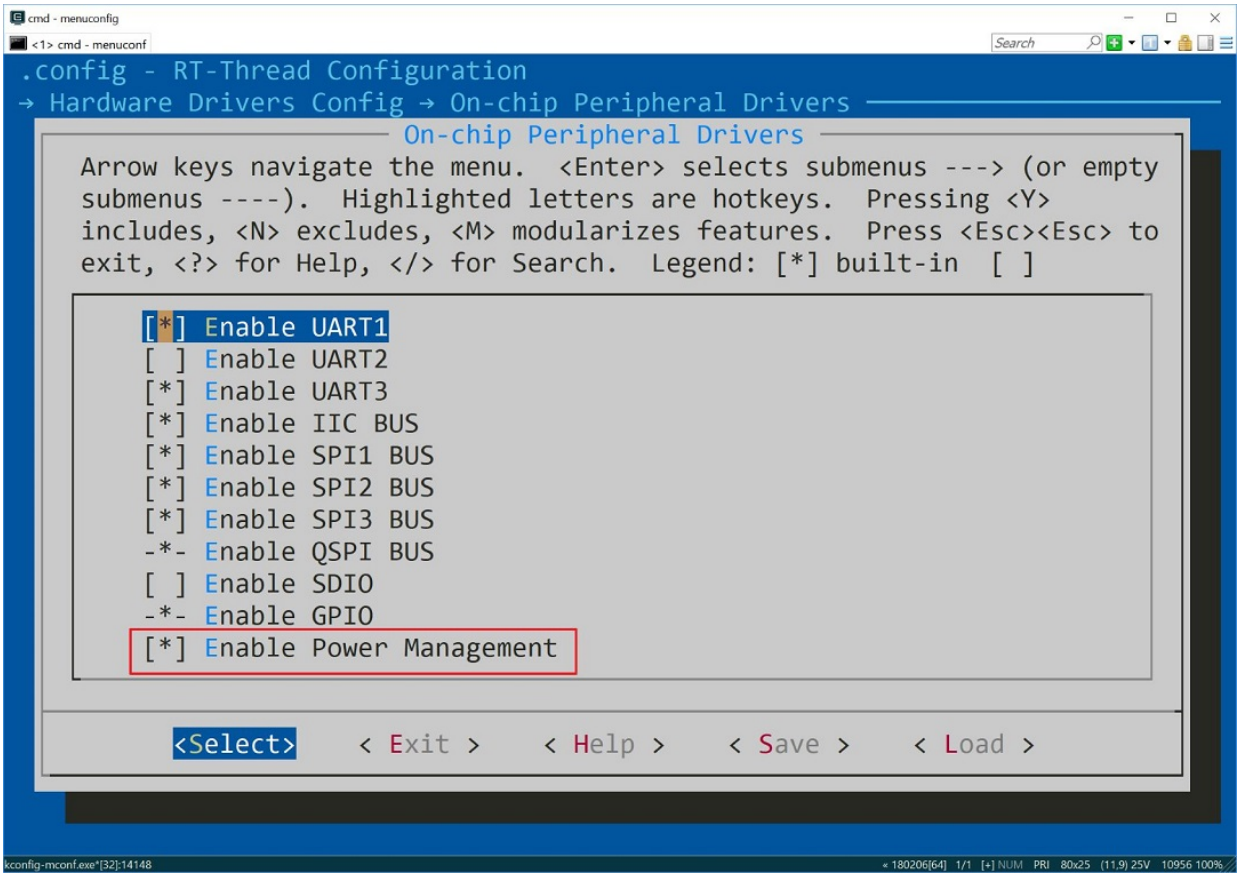

图 1：*配置组件*

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

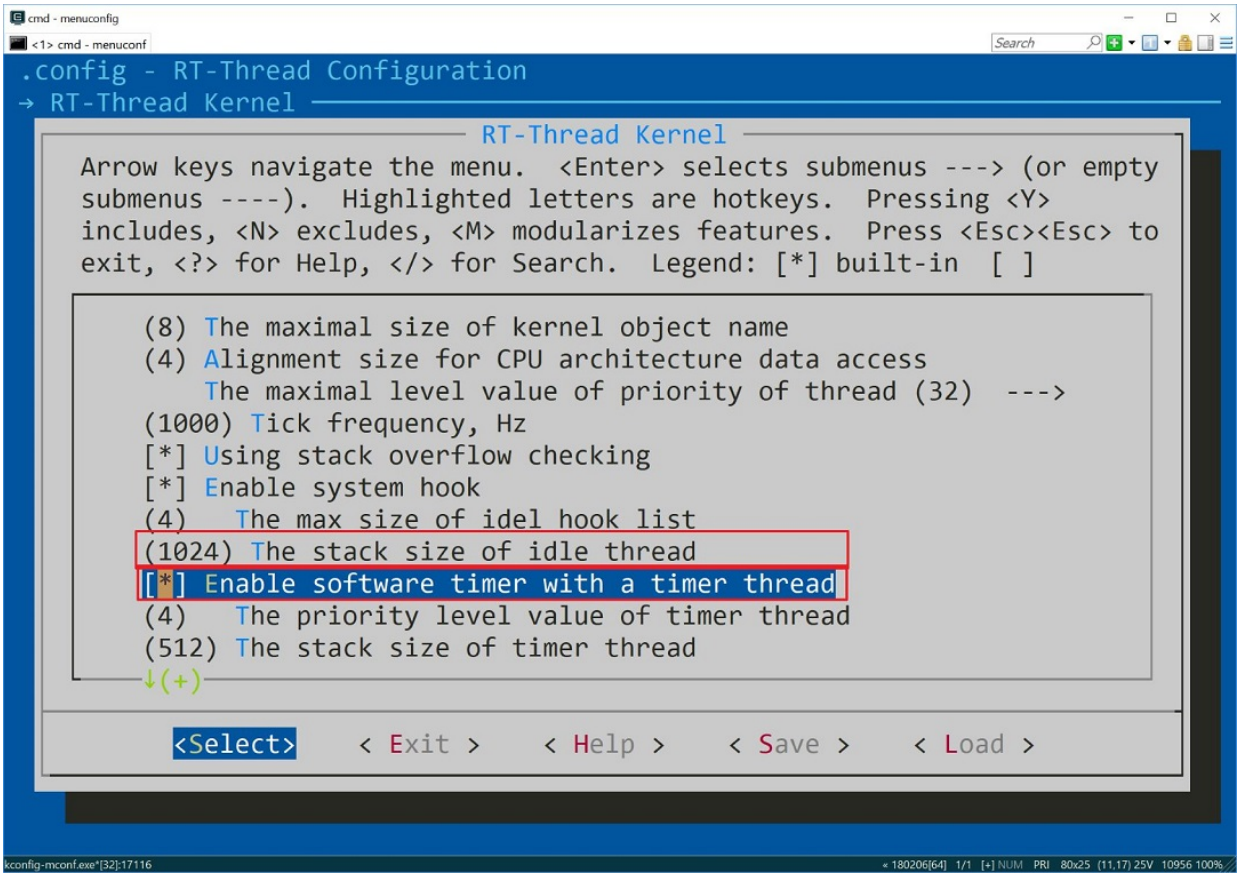

图 2: *配置内核选项*

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

我们打开 mdk5 工程，就可以看到相应的源码已经被添加进来：

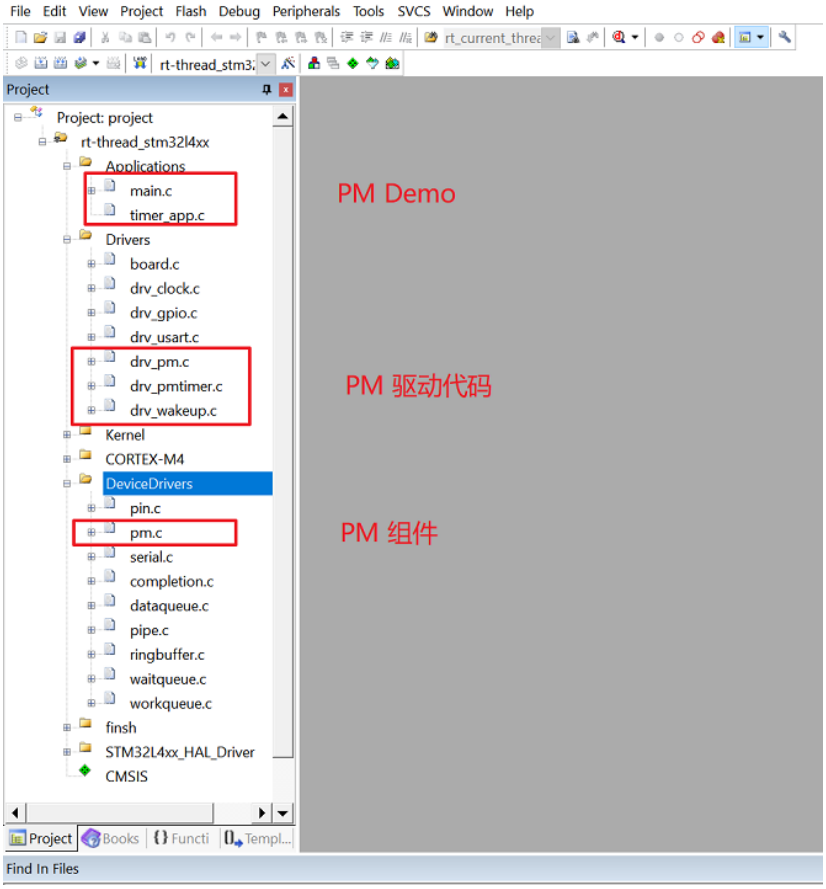

图 3: *MDK 工程*

3 示例说明

本节介绍了定时应用例程和按键唤醒例程。

定时应用例程的设计目的主要是让大家知道，PM 组件里的实现硬件无关的上层应用时，基本不需要过多低功耗的事情。按键唤醒例程的设计目的是让大家了解和 PM 组件相关的应用是如何实现的。

3.0.1. 定时应用 (timer_app)

周期性执行任务是一种很常见的需求。我们这里使用软件定时器来完成。我们将创建了一个周期性的软件定时器，定时器超时函数里输出当前的 OS Tick 当前值。

为了在休眠模式下，OS Tick 也能正常工作，我们希望进入的休眠模式也有休眠定时器。在 IoT Board 里`PM_SLEEP_MODE_TIMER`模式可以满足要求，所以我们在创建软件定时器成功之后，使用`rt_pm_request(PM_SLEEP_MODE_TIMER)`请求 TIMER 休眠模式。以下是示例代码：

```
#define TIMER_APP_DEFAULT_TICK  (RT_TICK_PER_SECOND * 2)

static rt_timer_t timer1;

static void _timeout_entry(void *parameter)
{
```

```
    rt_kprintf("current tick: %ld\n", rt_tick_get());
}

static int timer_app_init(void)
{
    timer1 = rt_timer_create("timer_app",
                             _timeout_entry,
                             RT_NULL,
                             TIMER_APP_DEFAULT_TICK,
                             RT_TIMER_FLAG_PERIODIC | RT_TIMER_FLAG_SOFT_TIMER);
    if (timer1 != RT_NULL)
    {
        rt_timer_start(timer1);

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

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

按下复位按键重启开发板，打开终端软件，可以看到有定时输出日志：

```
 \ | /
- RT -     Thread Operating System
 / | \     3.1.0 build Sep  7 2018
 2006 - 2018 Copyright by rt-thread team
msh />current tick: 2020
current tick: 4021
current tick: 6022
```

我们可以在 msh 里输入pm_dump命令观察 PM 组件的模式状态：

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

以上的输出说明，PM 组件里所有 PM 模式都被请求了一次，在pm_dump的模式列表里，优先级是从高

到低排列，所以现在正处于Running Mode。Running Mode、Sleep Mode和Shutdown Mode都是启动的时候已经被默认请求了一次。Timer Mode在定时应用里被请求一次。

我们首先输入命令pm_release 0来手动释放Running Mode，然后输入命令pm_release 1手动释放Sleep Mode后；最后 PM 组件将进入Timer Mode。在Timer Mode里，当软件定时器超时的时候，芯片会被唤醒。所以我们看到 shell 还是一直在输出：

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

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

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

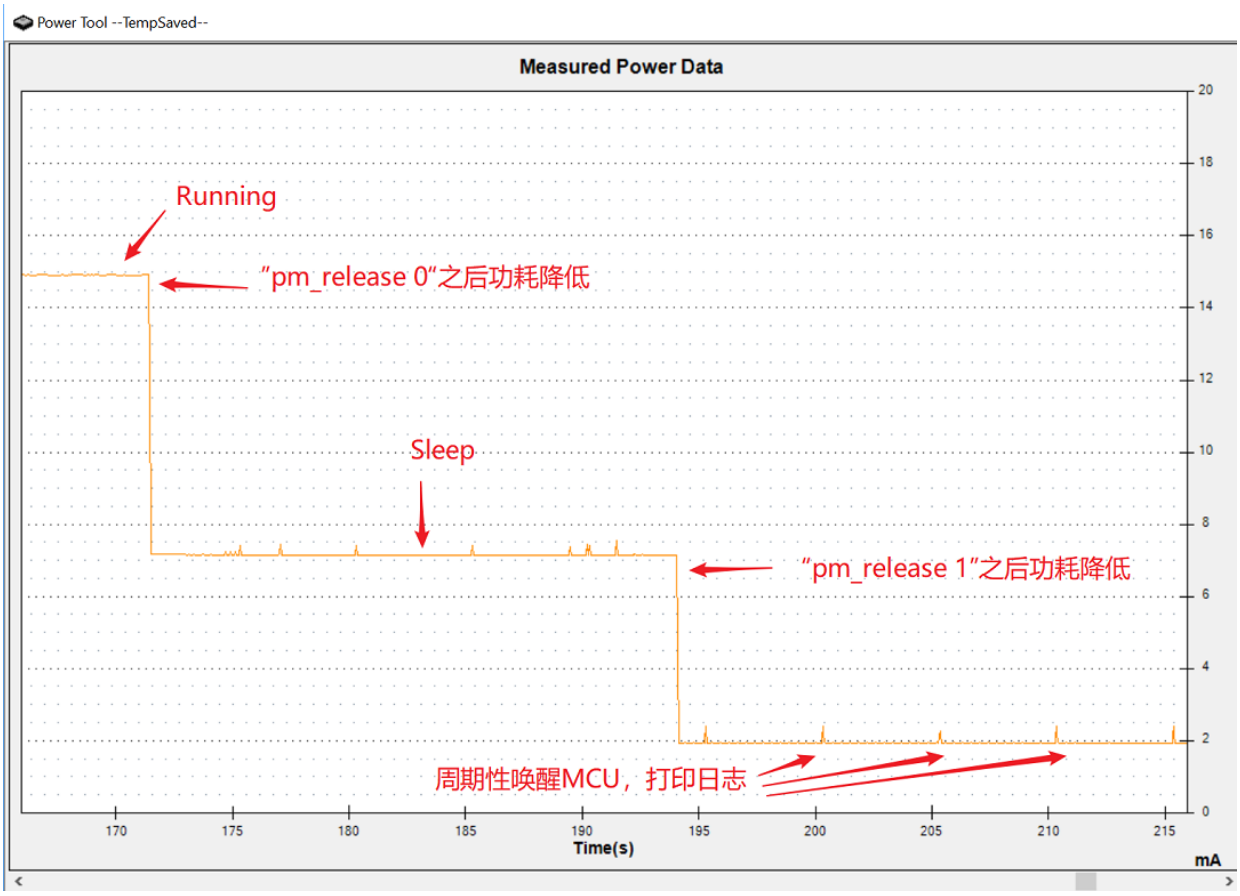

图 4: *功耗变化*

休眠时显示 2mA 是仪器的误差。

3.0.2. 按键唤醒应用

根据实际需求，从休眠模式下唤醒芯片完成任务，也是常见的低功耗场景。芯片通常支持各种唤醒方式，例如定时唤醒、外设中断唤醒、唤醒引脚唤醒等等。我们基于按键唤醒来展示 PM 组件是如何完成唤醒相关的应用。

在按键唤醒应用里，我们使用 wakeup 按键来唤醒处于休眠模式的 MCU。MCU 被唤醒之后，会触发相应的唤醒中断。以下例程是用 wakeup 按键从`Timer MODE`唤醒 MCU 并点亮 LED 之 2 秒后，再次进入休眠的例程。

例程的入口在`main()`函数里：

```
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();
        }
    }
}
```

`main()`函数里首先完成初始化工作：包括唤醒功能的初始化、PM 模式的配置；然后在循环里一直等待中断里发来的事件，如果接收到一次事件，就执行一次`led_task()`。

唤醒功能的初始化`wakeup_init()`包括了`wakeup`事件的初始化和唤醒中断里回调函数的初始化：

```
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`对应的是 STM32L475 的 STOP2 模式，并在进入之前打开了 LPTIM1。我们希望停留在`PM_SLEEP_MODE_TIMER`模式，所以首先需要调用一次`rt_pm_request()`来请求该模式。

由于在一开始，PM_SLEEP_MODE_SLEEP和PM_RUN_MODE_NORMAL模式都是 PM 组件启动的时候已经被默认请求了一次。为了不停留在这两个模式，我们需要调用 rt_pm_release() 释放它们：

```
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);
}
```

led_app()里我们希望点亮 LED 灯，并延时足够的时间以便我们观察到现象。在延时的时候，CPU 可能处于空闲状态，如果没有任何运行模式被请求，将会进入休眠。所以我们请求了PM_RUN_MODE_NORMAL，并在完成 LED 闪烁之后释放它：

```
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);
}
```

我们三次按下 wakeup 按键。每次按下按键，MCU 都会被唤醒点亮 LED 2 秒之后，再次进入休眠。下图是唤醒过程中的 Power Monitor 运行截图：

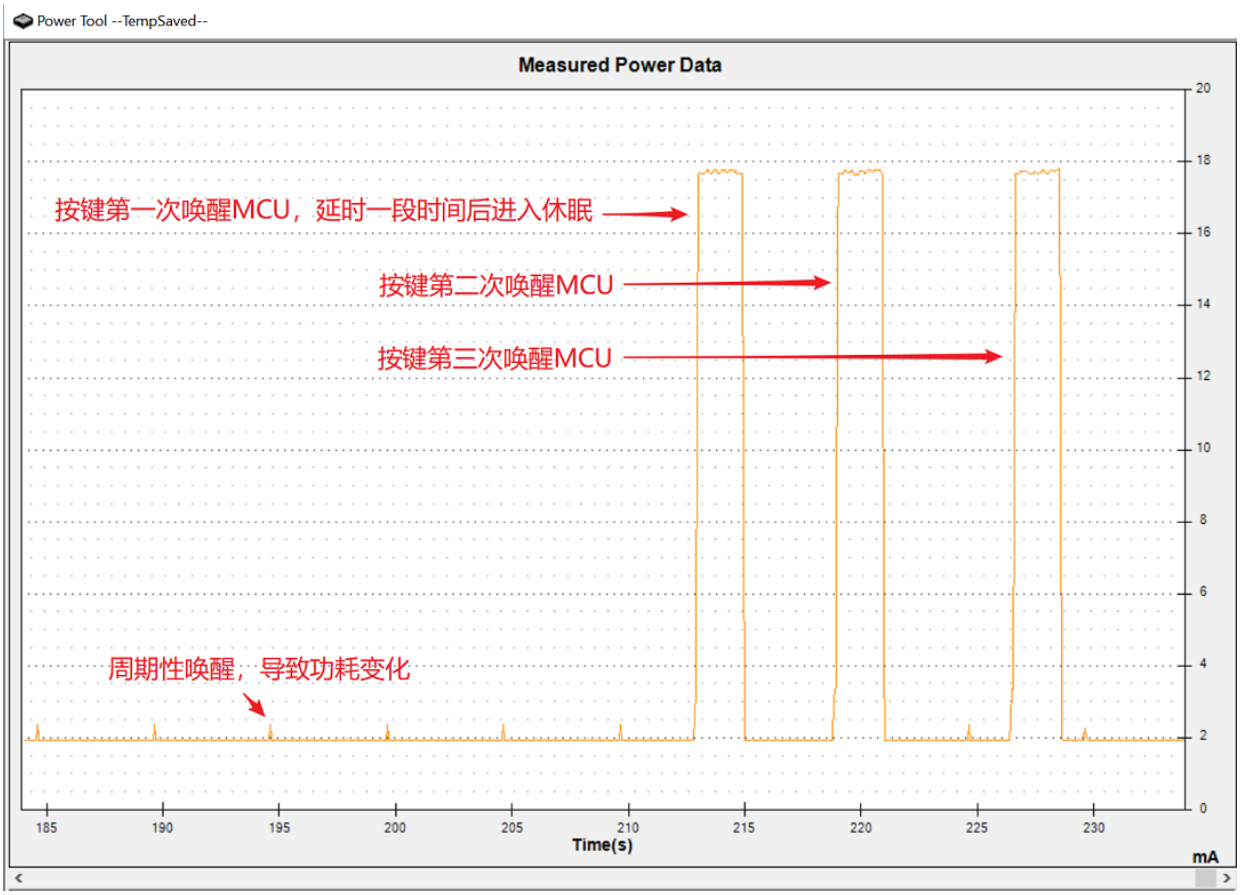

图 5: *功耗变化*

4 工作原理

4.1 芯片的运行频率和休眠模式

RT-Thread 的 PM 组件为什么要基于模式来管理呢？它是怎么演化出来的？本节将从芯片开始介绍，介绍 RT-Thread 的 PM 组件的设计。

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

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

以上只是 STM32L475 的时钟和休眠的情况。在不同的 MCU 之间，它们的时钟和低功耗可能会有很大的差异。高性能的 MCU 可以运行在 600M 以上或者更高，低功耗的 MCU 可以在 1~2M 以极低的功耗运行。

根据实际情况，应用里根据任务的需要，可以选择让芯片运行在高性能、普通性能或者极低性能模式；在当前没有需要处理的任务时，就可以让芯片进入休眠模式，休眠模式可以选择停止不同的外设，支持不同的外设在休眠模式里唤醒。

4.2 功耗是什么

上节介绍了芯片可以运行在不同的频率，可以进入不同的休眠模式。如果将芯片在不同时间的功耗展示出来，如下图：

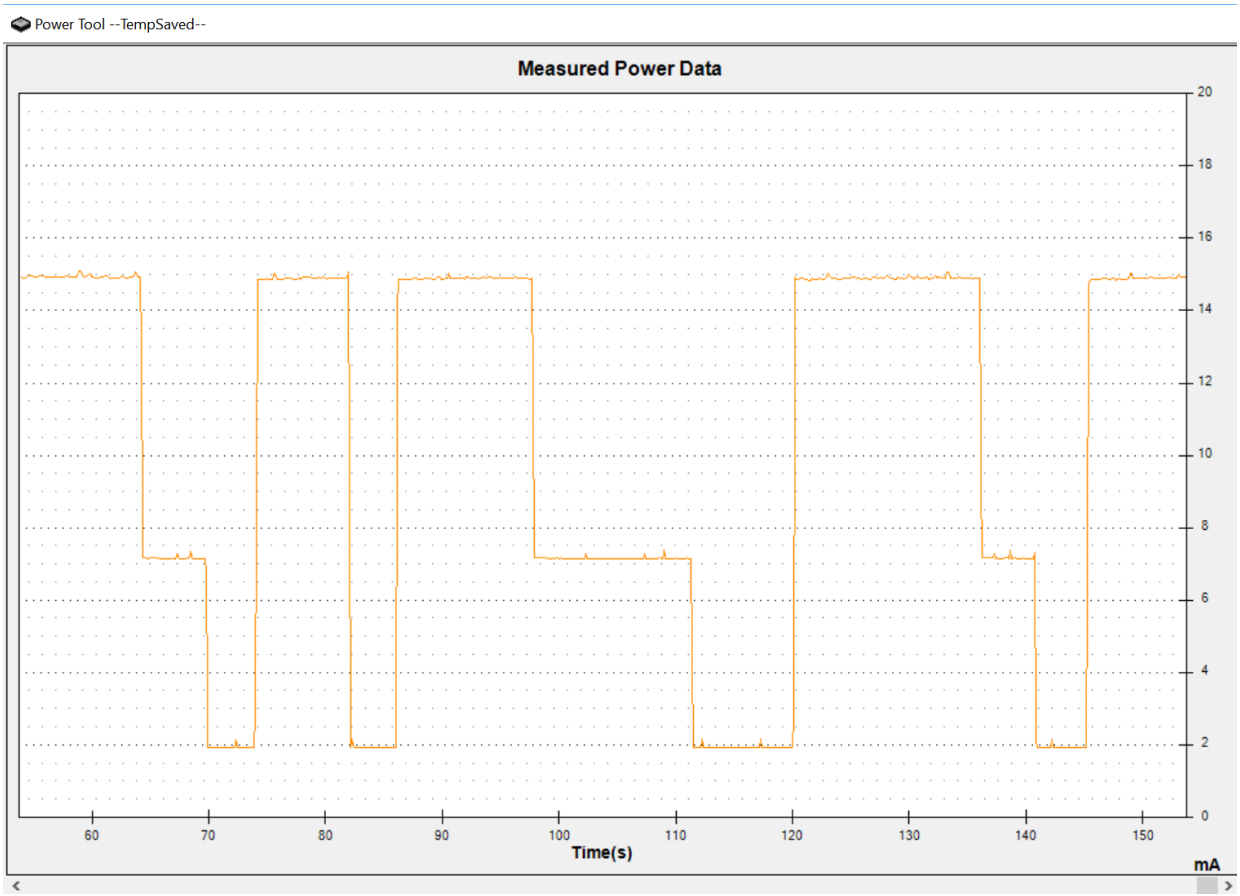

图 6: *功耗变化*

该图的横轴是时间，纵轴是芯片的当前功率，那么功耗就是被横轴和纵轴和功率曲线包裹起来的面积了。降低功耗也就是使得相应的面积更小。完成一个任务，如果可以在更低的频率下或者在更低休眠模式下进行，就可以降低功耗；如果可以在更短的时间内完成，也可以降低功耗。

5 深入 PM 组件

5.1 模式的定义

我们将 MCU 的时钟频率和休眠模式，统一叫做模式。如果当前模式下，CPU 还在工作的，叫做运行模式。如果 CPU 停止工作了，就叫做休眠模式。

在运行模式里，CPU 还是处于运行状态，根据运行的频率我们可以细分不同的运行模式。而在休眠模式里，CPU 已经停止工作，根据不同的外设是否还在工作我们可以细分不同的休眠模式。不同的 MCU 根据实际情况可以自定义不同的模式。

5.2 模式的改变

PM 组件里提供了两个 API 让用户来决定 PM 的模式，一个是rt_pm_request()用来请求模式，一个是rt_pm_release()用来释放请求的模式。PM 组件里记录了用户请求并根据这些请求，采用一票否决的机制自动处理运行频率和休眠模式的切换。

5.3 模式的一票否决

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

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

因此在 PM 组件里，只要有一个模式请求了更高优先级的模式，就不会切换到比它低的模式。这就是模式的一票否决。

5.4 模式改变的时机

PM 的模式一般来说，尽可能在驱动里完成，使得开发普通应用尽量不需要关心这部分。

在驱动里，如果外设希望在工作的时候不进入低功耗模式，那么在它工作的时候之前，就需要调用rt_pm_request()来请求相应的运行模式，在完成工作之后及时调用rt_pm_release()释放。

例如 100M 以太网至少需要 25M 的外部时钟，而它们可能和 CPU 共用一个 PLL，希望以太网驱动里就可以请求工作在比较高的运行模式。

如果是非低功耗相关的应用，基本不用关心模式。

5.5 PM 的实现

在 PM 里，每一个模式都有自己的计数器。如果某个模式的值不是 0，那就说明至少有一个请求希望工作在这个模式。根据一票否决的原则，PM 组件选择最高的模式运行，在 PM 的实现里就是第一个计数器非 0 的模式。

用户如果调用了rt_pm_request(PM_RUN_MODE_XXX)，PM_RUN_MODE_XXX模式的计数器会加一，如果请求模式比当前的模式更高，就会立即切换到新的模式，同时当前模式被修改成新的模式。

如果调用了rt_pm_release(PM_RUN_MODE_XXX)，PM_RUN_MODE_XXX模式的计数器会减一。如果释放的模式是当前模式，而且当前模式的计数器值变成 0，就意味着可以切换到更低的模式。在 PM 的实现里，这个切换并不会立即进行，而是在所有任务空闲的时候，在 IDLE HOOK 里调用rt_pm_enter()来完成。

模式之间切换，在不同的 BSP 可能是不一样的，所以需要针对不同的硬件进行适配，具体介绍参考移植说明。

在模式切换的时候外设也可能会收到影响，所以 PM 组件里提供了对模式变化敏感的 PM 设备功能，具体介绍参考下小节。

5.6 对模式变化敏感的 PM 设备

在 PM 组件里，切换到新的运行模式可能会导致 CPU 频率发生变化，如果外设和 CPU 共用一部分时钟，那外设的时钟就会受到影响；在进入新的休眠模式，大部分时钟源会被停止，如果外设不支持休眠的冻结功能，那么从休眠唤醒的时候，外设的时钟就需要重新配置外设。所以 PM 组件里支持了 PM 模式敏感的 PM 设备。每个 PM 设备需要实现以下函数：

```
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);
};
```

其中frequency_change()是在切换到新的**运行模式**时调用。suspend()是在进入休眠模式之前，resume()是在从休眠模式唤醒之后。在实现了相应的函数之后，我们可以通过注册 APIrt_pm_register_device()来注册到 PM 组件里。如果需要溢出这个设备，可以通过取消注册 APIrt_pm_unregister_device()来完成。

5.7 PM 的设备接口

一般情况我们是直接通过 PM 组件的 API 来使用的。同时 PM 组件也向上提供了设备接口，所以我们可以使用rt_device_read、rt_device_write、rt_device_control来使用 PM 组件。

如果打开了RT_USING_DFS_DEVFS选项，那么还可以基于使用文件的方式来访问。

6 移植说明

6.1 基本移植

本节介绍如何移植 PM 组件到新的 BSP 里。

PM 组件的底层功能全部都是通过**struct** rt_pm_ops结构体里的函数完成：

```
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);
```

```
};
```

在新的 BSP 里支持 PM 组件，只需要实现了`struct rt_pm_ops`里的函数，然后使用`rt_system_pm_init()`来完成初始化工作即可。

以下是`rt_system_pm_init()`的使用示例代码：

```
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;
}
```

上面的代码，将所有相关的函数保存在`_ops`变量里，并在变量`timer_mask`把那些包含了 timer 功能的模式对应的位置 1，最后调用`rt_system_pm_init()`完成初始化工作。

`struct rt_pm_ops`里面的函数里，必须实现的函数是`enter()`和`exit()`函数。如果没有打开自动变频功能，就不需要实现`frequency_change()`函数。如果所有模式里都不包含 timer 功能，那就不需要实现`timer_start()`、`timer_stop()`和`timer_get_tick()`函数。

下小节将按照 API 逐一介绍它们的具体实现。

6.1.0.1. __drv__pm__enter() 和 __drv__pm__exit() 函数的移植

`_drv_pm_enter()`函数里需要完成的功能是根据当前模式切换到新的运行模式的时钟配置或进入新的休眠模式。

`_drv_pm_exit()`函数里需要完成的功能是完成模式退出的清理工作，如果没有需要的清理，就不需要做任何事情。

`_drv_pm_enter()`和`_drv_pm_exit()`函数会在 PM 组件的模式变化的时候会被调用。PM 组件的模式变化有两种情况，一个是在`rt_pm_request()`里请求了比当前模式更高的模式，一个是在`rt_pm_enter()`里降低到比当前模式更低的模式。

每次模式变化的时候，PM 组件都会调用‘_drv_pm_exit()退出当前模式，然后更新模式变量pm->current_mode，最后调用了_drv_pm_exit()切换到新的模式。

所以_drv_pm_enter()和_drv_pm_exit()函数需要根据当前模式的值做不同的判断。以下是 STM32L475 里面的 _drv_pm_enter()的实现：

```
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;
    }
}
```

由于该 BSP 只有一个运行模式，而时钟已经在系统启动时完成了配置，所以

PM_RUN_MODE_NORMAL不需要做任何事情。剩下的三个休眠模式只需要根据实际情况选择进入不同的休眠模式即可。

以下是 STM32L475 里面的 _drv_pm_exit()的实现：

```
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;
    }
}
```

由于PM_SLEEP_MODE_SLEEP不会影响任何外设，所以在这个模式我们不需要做任何事情。我们希望在从PM_SLEEP_MODE_TIMER唤醒之后，就可以马上使用rt_kprintf()输出调试信息，所以在唤醒之后马上更新系统时钟，使得非低功耗的 uart 也能正常工作。

当然如果希望rt_kprintf()可以在唤醒之后马上工作，可以把这个 uart 注册为对模式变化敏感的 PM 设备。具体注册方式可以看后面章节。

6.1.0.2. __drv__pm__timer__xxx()

PM 组件在进入某个包含了 Timer 功能的休眠模式之前，根据下一次唤醒的时间来调用_drv_pm_timer_start()函数，才进入休眠。所以_drv_pm_timer_start()需要完成该休眠模式的定时器的配置，确保可以在指定的时间醒来，指定的时间是timeout个 OS Tick。

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

芯片在休眠一段时间之后被唤醒，可能是由于该休眠模式的定时器的超时中断，也可能是其他的外设中断。所以 PM 组件会在芯片被唤醒之后调用_drv_pm_timer_get_tick()函数，得到真正的休眠时间。最后 PM 组件会调用_drv_pm_timer_stop()函数来停止定时器。

_drv_pm_timer_start()函数和_drv_pm_timer_get_tick()都是使用 OS Tick 为单位来决定休眠时间的。但是 OS Tick 和休眠模式的定时器之间的转换可能存在误差，用户可以根据实际情况来决定忽略这部分误差，或者根据多次 OS Tick 的值来修正误差。

以下是 STM32L475 里面的 _drv_pm_timer_start()的实现：

```
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);
}
```

该函数首先将timeout变量的值由 OS Tick 转成 PM 模式的低功耗定时器的 Tick 值，然后确定这个值没有超过低功耗定时器的最大访问，最后打开了低功耗定时器。

以下是 STM32L475 里面的 _drv_pm_timer_stop()的实现：

```
static void _drv_pm_timer_stop(struct rt_pm *pm)
{
    RT_ASSERT(pm != RT_NULL);

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

该函数只是简单的停止了定时器。

以下是 STM32L475 里面的 _drv_pm_timer_stop()的实现：

```
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);
}
```

该函数里，首先去获取了 PM 模式的低功耗定时器的 Tick 值，然后转换成 OS Tick。stm32l4_lptim_get_current_tick()里只是简单的转换。而在stm32l4_os_tick_from_pm_tick()函数里，完成了累积误差的修复：

```
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;
}
```

该函数每次转行的时候，都将转换时的余数保存到os_tick_remain变量，并用在下次 OS Tick 到 PM Tick 的转行。

6.1.0.3. __drv_pm_frequency_change() PM 组件在每次切换到新的运行模式之后，会调用这个函数。所以在驱动里我们可以在这个函数里完成芯片的 CPU 频率调整。

6.2 支持对模式变化敏感的 PM 设备

在完成基本的移植之后，BSP 里还可以根据实际情况，注册对 PM 的模式变化敏感的设备，使得设备在切换到新的运行模式或者新的休眠模式都能正常的工作。

一个新的 PM 设备的功能是通过`struct rt_device_pm_ops`里的函数完成：

```
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);
};
```

在切换到新的运行模式，会逐个调用注册设备的`frequency_change()`函数。在进入休眠之前，会逐个调用注册设备的`suspend()`函数。在设备唤醒之后，会逐个调用注册设备的`resume()`函数。

在完成上述函数之后，我们可以使用`rt_pm_register_device()`和`rt_pm_unregister_device()`来管理 PM 设备。

注册 PM 设备，需要传入相应的设备指针以及相应的设备的 pm_ops，以下是一个模板：

```
#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);
}
```

取消注册 PM 设备，只需要传入相应的设备指针就可以了：

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

7 API 说明

下表是所有 PM 组件里面的 API：

PM 组件 API 列表	位置
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 详解

7.1.1. PM 组件初始化

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

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

该函数完成的工作，包括底层 PM 驱动的注册、相应的 PM 组件的资源初始化、默认模式的请求，并对上层提供一个名字是 “pm” 的设备，还默认请求了三个模式，包括一个默认运行模式PM_RUN_MODE_DEFAULT、一个默认休眠模式PM_SLEEP_MODE_DEFAULT和最低的模式PM_MODE_MAX。默认运行模式和休眠模式的值，我们可以根据需要来定义，默认值是第一个模式。

参数	描述
ops	底层 PM 驱动的函数集合

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

7.2 请求 PM 模式

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

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

参数	描述
mode	请求的模式

7.3 释放 PM 模式

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

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

参数	描述
mode	释放的模式

7.4 注册 PM 模式变化敏感的设备

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

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

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

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

参数	描述
device	具体对模式变化敏感的设备
ops	设备的函数集合

7.5 取消注册 PM 模式变化敏感的设备

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

该函数取消已经注册的 PM 模式变化敏感设备。

7.6 PM 模式进入函数

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

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

7.7 PM 模式退出函数

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

该函数在从休眠模式唤醒的时候被在`rt_pm_enter()`调用。在从休眠唤醒时，有可能先进入唤醒中断的中断处理函数里由。用户也可以在这里主动调用用户主动调用`rt_pm_exit()`。从休眠唤醒之后可能多次调用`rt_pm_exit()`。