XR871 PM User Guide

Revision 1.0

May 8, 2017

Declaration

THIS DOCUMENTATION IS THE ORIGINAL WORK AND COPYRIGHTED PROPERTY OF XRADIO TECHNOLOGY ("XRADIO"). REPRODUCTION IN WHOLE OR IN PART MUST OBTAIN THE WRITTEN APPROVAL OF XRADIO AND GIVE CLEAR ACKNOWLEDGEMENT TO THE COPYRIGHT OWNER.

THE INFORMATION FURNISHED BY XRADIO IS BELIEVED TO BE ACCURATE AND RELIABLE. XRADIO RESERVES THE RIGHT TO MAKE CHANGES IN CIRCUIT DESIGN AND/OR SPECIFICATIONS AT ANY TIME WITHOUT NOTICE. XRADIO DOES NOT ASSUME ANY RESPONSIBILITY AND LIABILITY FOR ITS USE. NOR FOR ANY INFRINGEMENTS OF PATENTS OR OTHER RIGHTS OF THE THIRD PARTIES WHICH MAY RESULT FROM ITS USE. NO LICENSE IS GRANTED BY IMPLICATION OR OTHERWISE UNDER ANY PATENT OR PATENT RIGHTS OF XRADIO. THIS DATASHEET NEITHER STATES NOR IMPLIES WARRANTY OF ANY KIND, INCLUDING FITNESS FOR ANY PARTICULAR APPLICATION.

THIRD PARTY LICENCES MAY BE REQUIRED TO IMPLEMENT THE SOLUTION/PRODUCT. CUSTOMERS SHALL BE SOLELY RESPONSIBLE TO OBTAIN ALL APPROPRIATELY REQUIRED THIRD PARTY LICENCES. XRADIO SHALL NOT BE LIABLE FOR ANY LICENCE FEE OR ROYALTY DUE IN RESPECT OF ANY REQUIRED THIRD PARTY LICENCE. XRADIO SHALL HAVE NO WARRANTY, INDEMNITY OR OTHER OBLIGATIONS WITH RESPECT TO MATTERS COVERED UNDER ANY REQUIRED THIRD PARTY LICENCE.

Revision History

Version	Data	Summary of Changes
1.0	2017-9-26	Initial Version

Table 1-1 Revision History

Contents

Tables

Figures

1 模块概要

1.1 模块功能介绍

XR871 PM(power manager)即低功耗模块主要用于降低系统功耗。

系统运行时降低功耗主要使用 WFI 和 tickless，默认都开启，不需要用户做任何配置。

WFI 是 cpu 没任务时进入 WFI(wait for interrupt) 状态，WFI 状态比正常工作状态功耗低不少。

Tickless 是在低负载时，关闭不必要的 tick 中断以让 cpu 更多时间在 WFI 状态。

休眠状态包括 sleep 和 standby，用户可以配置此系统支持与否。

休眠是在系统长时间无任务而又需要联网时，关闭多数外设，保留网络通信能力，在收到数据时，能尽快唤醒系统进行处理。

关机是系统只保留 prcm/RTC 部分的电源，只有特定的 wakeup gpio 和 wakeup timer 能复位系统，系统复位后，重新初始化。

1.2 软件结构

系统进入低功耗模式的流程，如果是进入 sleep 或 standby 模式，系统可被唤醒，唤醒后与休眠之前的配置一致，流程如下：

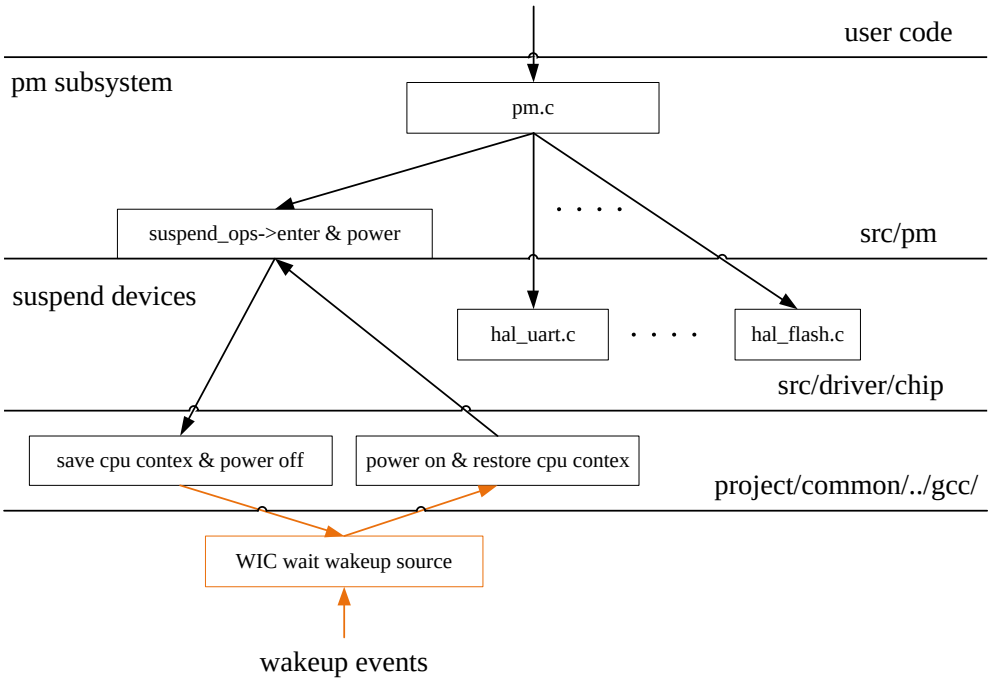

图 1-1 函数调用关系框图

1.3 调用流程

1. 用户触发休眠动作，即调用休眠函数： pm_enter_mode()。

2. 休眠函数会依次调用各设备注册的 suspend 和 suspend_noirq 函数，调用顺序为注册的相反顺序，保证资源的依赖关系不会错误。

各函数依次休眠完成且无错误后，最后一次判断是否有中断发生，如果有中断，则依次唤醒设备，退出休眠；如果无中断发生，则调用 cpu_suspend 函数，保存 cpu 上下文，设置 deepsleep mode，进入 WFI 状态。

3. 硬件模块 WIC 检测到 cpu 进入 deepsleep mode 且处于 WFI 状态时，关闭电源域。

4. WIC 等待唤醒事件的发生。有唤醒事件时，系统重新上电，cpu 从复位函数开始执行。

5. 复位函数检测到上次为休眠行为，则跳入恢复现场函数，现场恢复后，为 suspend_ops->enter() 执行的下一条语句，即开始执行唤醒流程。

6. 依次调用设备的 resume_noirq 和 resume 函数，调用流程与休眠顺序相反，即最后休眠的最先唤醒。

2 模块接口

2.1 模块接口

Reference	Functions	Detail
PM	HAL_Wakeup_GetEvent	uint32_t HAL_Wakeup_GetEvent(void) Purpose： 获取唤醒事件 Param : dev： None Returns： 事件定义见 hal_wakeup.h 中 PM_WAKEUP_SRC_xx 定义
	HAL_Wakeup_SetIO	void HAL_Wakeup_SetIO(uint32_t pn, uint32_t mode) Purpose： 设置 wakeup gpio Param : dev： Pn：port num，0~9，编号与实际 gpio 对应关系见 hal_gpio.h Mode: 0：下降沿唤醒，1：上升沿唤醒 Returns : None
	HAL_Wakeup_ClrIO	void HAL_Wakeup_ClrIO(uint32_t pn) Purpose： 清除 wakeup gpio Param : dev： Pn：port num，0~9，编号与实际 gpio 对应关系见 hal_gpio.h Returns : None
	HAL_Wakeup_SetTimer	int32_t HAL_Wakeup_SetTimer(uint32_t count_32k) Purpose： 设置 wakeup timer Param : count_32k：唤醒时间，基于 32k 计数 Returns： 返回状态（0：ok）
	HAL_Wakeup_SetTimer_mS	int32_t HAL_Wakeup_SetTimer_mS(ms) Purpose： 设置 wakeup timer Param : ms：唤醒时间，基于 ms Returns： 返回状态（0：ok）
	pm_register_ops	int pm_register_ops(struct soc_device *dev) Purpose： 注册设备的休眠唤醒函数实现 Param : dev：设备信息和休眠唤醒函数的实现

		必须要实现 dev->driver-> suspend/ resume, 或 dev->driver-> suspend_noirq / resume_noirq. Returns： 返回状态（0：ok）
	pm_unregister_ops	int pm_unregister_ops(struct soc_device *dev) Purpose： 注销设备的休眠唤醒函数 Param： dev：设备信息 Returns： 返回状态（0：ok）
	pm_enter_mode	int pm_enter_mode(enum suspend_state_t state) Purpose： 进入 state 低功耗模式 Param : state 为将要进入的模式 Returns : None
	pm_mode_platform_select	void pm_mode_platform_select(unsigned int select) Purpose： 选择此平台所使用的模式 Param : select 集 Returns : None
	pm_set_test_level	void pm_set_test_level(enum suspend_test_level_t level) Purpose： 测试时设置休眠到哪一层就返回 Param : level 为进入的最大层数 Returns : None
	pm_set_debug_delay_ms	void pm_set_debug_delay_ms(unsigned int ms) Purpose： 测试设置每个设备休眠之间的间隔，防止设备互相干扰 Param : ms 为间隔毫秒 Returns : None
	pm_stats_show	void pm_stats_show(void) Purpose： 打印休眠一些信息，如次数等 Param : None Returns : None
	HAL_Wakeup_SetIOHold	int32_t HAL_Wakeup_SetIOHold(uint32_t hold_io) Purpose： 设置 io hold，用户不需要调用 Param : dev：hold_io，io hold mask Returns： 返回状态（0：ok）

	HAL_Wakeup_SetSrc	int32_t HAL_Wakeup_SetSrc(void) Purpose： 设置唤醒源，用户不需要调用 Param : None Returns： 返回状态（0：ok）
	HAL_Wakeup_ClrSrc	void HAL_Wakeup_ClrSrc(void) Purpose： 清除唤醒源，用户不需要调用 Param : None Returns : None
	HAL_Wakeup_Init	void HAL_Wakeup_Init(void) Purpose： 初始化 wakeup，用户不需要调用 Param : None Returns : None
	HAL_Wakeup_DeInit	void HAL_Wakeup_DeInit(void) Purpose： 注销 wakeup，用户不需要调用 Param : None Returns : None
	pm_register_wlan_power_onoff	int pm_register_wlan_power_onoff(pm_wlan_power_onoff wlan_power_cb, unsigned int select) Purpose： 选择休眠时，是否要通过回调 wlan_power_cb 来开关网络，用户不需要调用 Param : wlan_power_cb 开关网络的回调函数 select 需要调用回调函数的模式集合 Returns： 返回状态（0：ok）
	pm_unregister_wlan_power_onoff	void pm_unregister_wlan_power_onoff(void) Purpose： 注销来开关网络功能，用户不需要调用 Param : None Returns : None
	pm_init	int pm_init(void) Purpose： 初始化 pm 模块，用户不需要调用 Param： None Returns： 返回状态（0：ok）

	pm_set_sync_magic	void pm_set_sync_magic(void) Purpose : 设置同步标志，用户不需要调用 Param : None Returns : None

3 使用示例

3.1 新加设备的休眠唤醒函数

新加设备，如果休眠时不能掉电，而且在休眠时不使用设备唤醒系统，就需要添加设备的休眠唤醒函数，以在休眠状态下降低系统功耗。为节省空间，可以服用初始化和注销函数，uart 示例代码如下：

```
#include “pm/pm.h”
.....
#ifdef CONFIG_PM
static int8_t g_uart_suspending = 0;
static UART_InitParam g_uart_param[UART_NUM];

static int uart_suspend(struct soc_device *dev, enum suspend_state_t state)
{
    UART_ID uartID = (UART_ID)dev->platform_data;

    g_uart_suspending |= (1 << uartID);

    switch (state) {
    case PM_MODE_SLEEP:
    case PM_MODE_STANDBY:
    case PM_MODE_HIBERNATION:
    case PM_MODE_POWEROFF:
        HAL_DBG("%s id:%d okay\n", __func__, uartID);
        HAL_UART_DeInit(uartID);
        break;
```



```
        default:
                break;
        }
        return 0;
}

static int uart_resume(struct soc_device *dev, enum suspend_state_t state)
{
        UART_ID uartID = (UART_ID)dev->platform_data;

        switch (state) {
        case PM_MODE_SLEEP:
        case PM_MODE_STANDBY:
        case PM_MODE_HIBERNATION:
                HAL_UART_Init(uartID, &g_uart_param[uartID]);
                HAL_DBG("%s id:%d okay\n", __func__, uartID);
                break;
        default:
                break;
        }

        g_uart_suspending &= ~(1 << uartID);

        return 0;
}

static struct soc_device_driver uart_drv = {
        .name = "uart",
        .suspend_noirq = uart_suspend,
        .resume_noirq = uart_resume,
};
```



```
static struct soc_device uart_dev[UART_NUM] = {
    {.name = "uart0", .driver = &uart_drv, .platform_data = (void *)UART0_ID,},
    {.name = "uart1", .driver = &uart_drv, .platform_data = (void *)UART1_ID,},
};

#define UART_DEV(id) (&uart_dev[id])

#endif /* CONFIG_PM */
......

HAL_Status HAL_UART_Init(UART_ID uartID, const UART_InitParam *param)
{......
#ifdef CONFIG_PM
    if (!(g_uart_suspending & (1 << uartID))) {
        memcpy(&g_uart_param[uartID], param, sizeof(UART_InitParam));
        pm_register_ops(UART_DEV(uartID));
    }
#endif
......
}

HAL_Status HAL_UART_DeInit(UART_ID uartID)
{......
#ifdef CONFIG_PM
    if (!(g_uart_suspending & (1 << uartID))) {
        pm_unregister_ops(UART_DEV(uartID));
        memset(&g_uart_param[uartID], 0, sizeof(UART_InitParam));
    }
#endif
......
}
```

设备休眠唤醒函数编写注意事项：

1. 设备 suspend 函数功能是将设备设为低功耗模式，如果在休眠期间不在使用，则应处于关闭状态，如果

需要作为唤醒源，则应保留最低能工作的资源（power、clk、gating），关闭不需要的资源。

2. 设备 resume 函数是将设备设置为正常工作模式，如果休眠时已经完全关闭，则此函数应该重新初始化。如果休眠时仍为工作状态，则此函数应该谨慎处理，不应将中断丢失。

3.2 设置唤醒源

唤醒源设置需要包含头文件："driver/chip/hal_wakeup.h"

设置 wakeup gpio：

```
HAL_Wakeup_SetIO(io_num, mode);
```

Wakeup gpio 只在休眠期间使用，正常工作时可配置为其他模式，此函数不会改变对应 gpio 的功能，唤醒后，gpio 为初始状态，需要重新初始化。另外只要配置一次，以后每次休眠都有效。参加见接口模块说明。

如果不再继续使用此 gpio 作为唤醒源，需要清除掉：

```
HAL_Wakeup_ClrIO(io_num);
```

设置 wakeup timer：

```
HAL_Wakeup_SetTimer_mS(ms);
```

此函数从调用后就开始计时，下次休眠时要重新配置，在 32k 时钟精确的情况下，时间范围为：10ms ~ 18h，另外精度受 32k 时钟影响，如果使用内部晶振，常温下频率为 37.5k，在温度变化比较大时，可能会漂移至 60k，对应的时间要重新计算。

3.3 进入 sleep 状态

Sleep 状态，外设中断可以随时唤醒系统，如果 sleep 期间，外设仍处于工作状态，则外设产生中断，系统即退出 sleep 模式。

```
pm_enter_mode(PM_MODE_SLEEP)
```

3.4 进入 standby 状态

Standby 状态，可以唤醒系统的中断有三种，第一是 wakeup gpio，需要配置为唤醒源状态才起效果；第二是 wakeup timer，需要在休眠前配置好唤醒时间；第三是网络部分，如果 standby 期间仍然需要联网，则网络部分会保持连接状态并进入低功耗模式，在休眠期间有网络数据就会唤醒系统，另外 sta 状态下，所连接 AP 发送的一些广播数据也会唤醒系统，比如 group key 更新等，如果不是需要的唤醒事件，用户需要再次进入休眠。


```
pm_enter_mode(PM_MODE_STANDBY)
```

3.5 进入 poweroff 状态

Poweroff 状态，可以有三种方式复位系统，第一是直接按 reset 键或重新断电上电，第二是 wakeup gpio，需要配置为唤醒源状态才起效果；第三是 wakeup timer，需要在休眠前配置好唤醒时间；

```
pm_enter_mode(PM_MODE_POWEROFF)
```

4 常见问题及调试方法

4.1 进入低功耗模式后功耗偏高

检查所有设备是否处于合理状态，休眠期间不使用的设备需要设置处于低功耗模式或关闭状态。休眠期间需要工作并唤醒系统的设备，需要设置成可工作的低功耗模式。

上述检查都进行后，功耗仍然不能降低至合理值的，需要测试各个分量的功耗，依次找出不合理的分量。

4.2 系统不能不醒

检查唤醒源配置是否正确，wake gpio 编号和状态是否正确，wakeup timer 的时间是否合理，如果 wakeup timer 时间较短，系统还没完全进入休眠，则 wakeup timer 会提前产生中断并被清理，不能起到唤醒系统的作用。

4.3 系统进入不了低功耗模式

检查系统中断产生情况，如果休眠过程中，系统产生了中断，系统休眠流程会被打断而退出。

