



■ **PHY622X**

PWR MGR Application Note

Version 1.0

Author: Bingliang.lou

Security: Internal / Public

Date: 2020.12

PhyPlus

Copyright © 2020 Phyplus Microelectronics Limited All rights reserved.
Reproduction in whole or in part is prohibited without the prior written permission of the copyright holder.



Revision History

Revision	Author	Date	Description
v0.1	Bingliang.lou	2020.12.09	Draft
v1.0	Bingliang.lou, Zhongqi.yang	2021.3.17	First Edition.

目录

1	概述.....	1
1.1	PWR_MGR 原理框图.....	1
2	PWR_MGR 模块 API.....	3
2.1	数据结构和类型.....	3
2.1.1	MODULE_e 类型.....	3
2.1.2	pwrmgr_Ctx_t.....	4
2.1.3	pwroff_cfg_t.....	4
2.2	APIs.....	4
3	低功耗模式使用注意事项.....	6
4	功耗评估及实测数据.....	6
4.1	功耗评估模型.....	6
4.2	功耗估算.....	7
4.3	实际测算的功耗.....	8
4.4	系统上电启动时间.....	9

图表目录

Figure 1:	深度休眠模式原理框图.....	1
Figure 2:	standby 模式原理框图.....	2
Figure 3:	功耗评估模型.....	7
Table 1	advertising 功耗测算值.....	8
Figure 4:	功耗实测图.....	8
Table 2	上电启动时间.....	9



PHY622X PWR MGR Application Note v1.0

1 概述

PHY622X 低功耗相关的功能在 PWR_MGR 模块实现，对应的 API 代码存放在 SDK 的 components\driver\pwrmgr 目录下的 pwrmgr.c 和 pwrmgr.h 中。

PHY622X 有四类功耗模式：

- **普通模式**：CPU 和外设全速运行，不休眠
- **CPU 休眠模式**：只有 CPU 会进入休眠，可由中断或事件唤醒。该模式由 OS 自行控制，应用程序无需干预
- **深度休眠模式**：CPU 和大部分外设都会进入休眠。应用程序应根据需要设置休眠唤醒源（GPIO pin 和触发方式）和内存保持(memory retention,以保持运行时上下文)
- **standby 模式**：除了 AON 和一块 RAM 区域内内存保持外，CPU 和其它外设都进入休眠。仅维持 RAM0 区域内容。应用程序应根据需要设置唤醒源（GPIO pin 和触发方式）
- **关机模式**：除了 AON 外，CPU 和其它外设都进入休眠。唤醒后相当于系统重启，不能维持运行时上下文。应用程序应根据需要设置唤醒源（GPIO pin 和触发方式）

1.1 PWR_MGR 原理框图

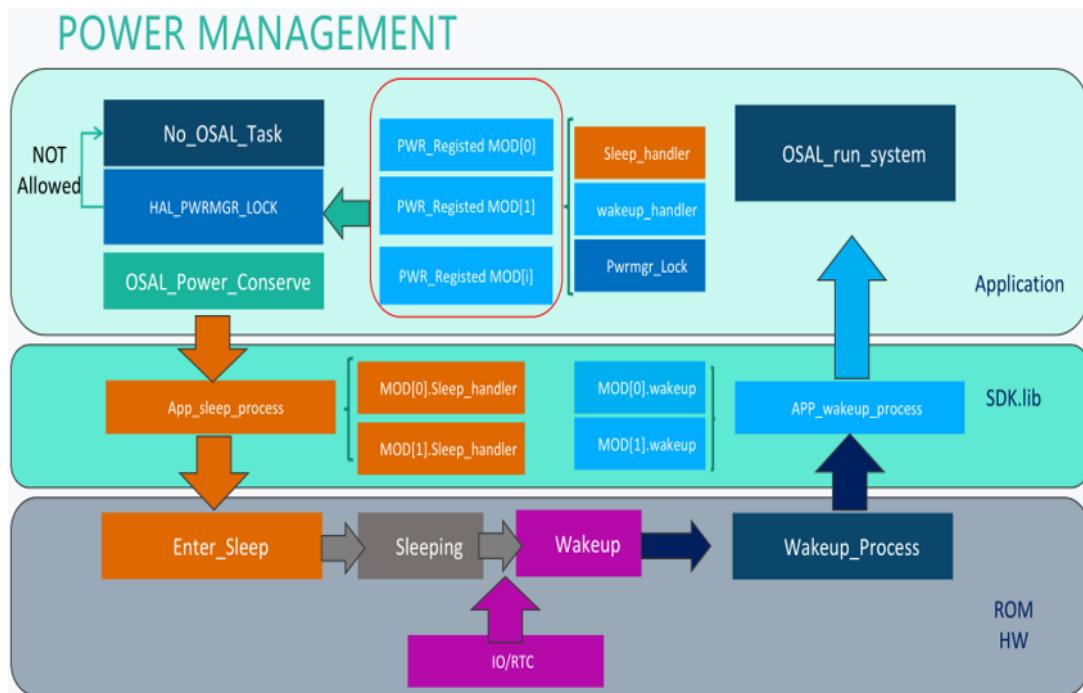


Figure 1: 深度休眠模式原理框图

在 CPU/深度休眠模式下，当系统处于 IDLE 状态时，调用 sleep_process() 尝试进入休眠模式。若符合 CPU 休眠条件，则调用 __WFI() 等待中断唤醒后返回；否则的话，系统将进入深度休眠，在此之前，应用程序通过 hal_pwrmgr_register() API 注册的 sleep_handler() 函数都会被回调，然后系统进入休眠；而当有 IO/RTC 触发唤醒时，系统会被唤醒并做相应的初始化。在系统唤醒过程中，应用程序通过 hal_pwrmgr_register() API 注册的所有 wakeup_handler() 函数也都会被回调，然后由 OSAL 调度 task。

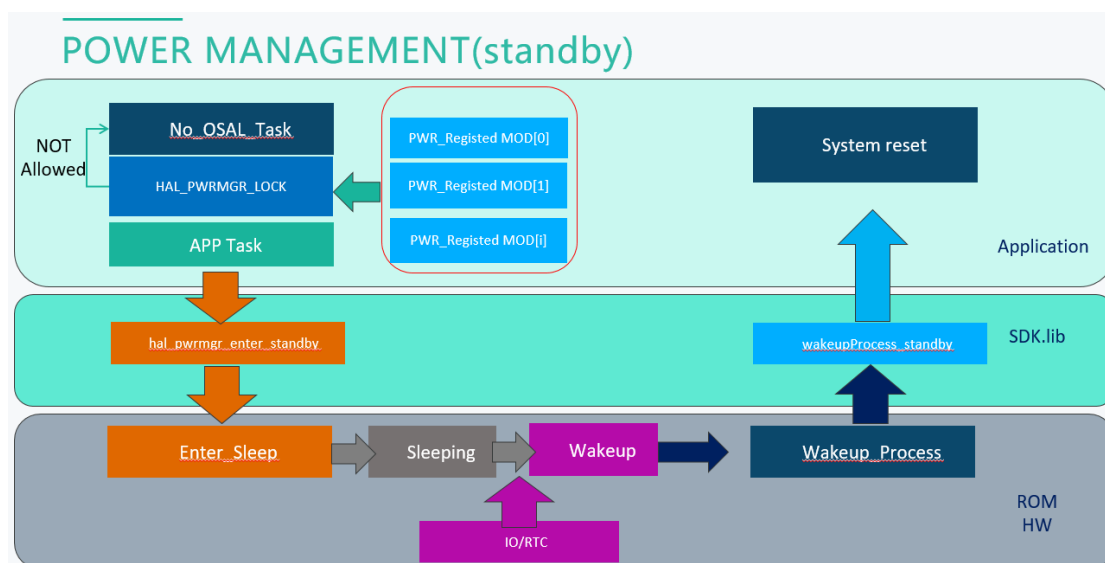


Figure 2: standby 模式原理框图

在 standby 模式下，APP Task 调用 `hal_pwrmgr_enter_standby()` 进入 standby 模式；而当有 IO 触发唤醒时，系统会被唤醒并调用 `wakeupProcess_standby()`，若系统满足唤醒的条件，则触发系统 reset。

2 PWR_MGR 模块 API

2.1 数据结构和类型

2.1.1 MODULE_e 类型

在 mcu_phy_bumbee.h 文件中定义了下列 module ID.

```
typedef enum {
    MOD_NONE          = 0, MOD_CK802_CPU    = 0,
    MOD_DMA            = 3,
    MOD_AES            = 4,
    MOD_IOMUX          = 7,
    MOD_UART0          = 8,
    MOD_I2C0           = 9,
    MOD_I2C1           = 10,
    MOD_SPI0           = 11,
    MOD_SPI1           = 12,
    MOD_GPIO           = 13,
    MOD_QDEC           = 15,
    MOD_ADCC           = 17,
    MOD_PWM            = 18,
    MOD_SPIF           = 19,
    MOD_VOC            = 20,
    MOD_TIMER5         = 21,
    MOD_TIMER6         = 22,
    MOD_UART1          = 25,

    MOD_CP_CPU         = 0 + 32,
    MOD_BB             = MOD_CP_CPU + 3,
    MOD_TIMER          = MOD_CP_CPU + 4,
    MOD_WDT            = MOD_CP_CPU + 5,
    MOD_COM            = MOD_CP_CPU + 6,
    MOD_KSCAN          = MOD_CP_CPU + 7,
    MOD_BBREG          = MOD_CP_CPU + 9,
    BBLL_RST           = MOD_CP_CPU + 10, //can reset,but not gate in here
    BBTX_RST           = MOD_CP_CPU + 11, //can reset,but not gate in here
    BBRX_RST           = MOD_CP_CPU + 12, //can reset,but not gate in here
    BBMIX_RST          = MOD_CP_CPU + 13, //can reset,but not gate in here
    MOD_TIMER1         = MOD_CP_CPU + 21,
    MOD_TIMER2         = MOD_CP_CPU + 22,
    MOD_TIMER3         = MOD_CP_CPU + 23,
    MOD_TIMER4         = MOD_CP_CPU + 24,
```

```

MOD_PCLK_CACHE = 0 + 64,
MOD_HCLK_CACHE = MOD_PCLK_CACHE + 1,

MOD_USR0        = 0 + 96,
MOD_USR1        = MOD_USR0 + 1,
MOD_USR2        = MOD_USR0 + 2,
MOD_USR3        = MOD_USR0 + 3,
MOD_USR4        = MOD_USR0 + 4,
MOD_USR5        = MOD_USR0 + 5,
MOD_USR6        = MOD_USR0 + 6,
MOD_USR7        = MOD_USR0 + 7,
MOD_USR8        = MOD_USR0 + 8,
} MODULE_e;

```

2.1.2 pwrmgr_Ctx_t

PWR_MGR 模块为每个注册的模块(与 MODULE_e 对应)维护一个该结构类型的变量。最多 10 个。

```

typedef struct _pwrmgr_Context_t {
    MODULE_e      moudle_id;
    bool          lock; // 为 TRUE 时表示禁止休眠；反之，允许休眠
    pwrmgr_Hdl_t sleep_handler; // 该模块对应的进入休眠之前会被调用的回调函数
    pwrmgr_Hdl_t wakeup_handler; // 该模块对应的唤醒之前会被调用的回调函数
} pwrmgr_Ctx_t;

```

2.1.3 pwroff_cfg_t

在系统调用 hal_pwrmgr_poweroff() API 进入关机模式之前，需要设置的唤醒源(GPIO pin)和触发方式保存在该类型的变量中。

```

typedef struct {
    gpio_pin_e pin;
    gpio_polarity_e type; // POL_FALLING or POL_RISING
} pwroff_cfg_t;

```

2.2 APIs

PWR_MGR 模块的接口函数如下：

1. int hal_pwrmgr_init(void);
模块初始化函数
2. bool hal_pwrmgr_is_lock(MODULE_e mod);
查询模块 mod 的 lock 状态。TRUE:禁止休眠；FALSE:使能休眠

3. `int hal_pwrmgr_lock(MODULE_e mod);`
设置模块 `mod` 的 `lock` 为 `TRUE`,并禁止休眠
4. `int hal_pwrmgr_unlock(MODULE_e mod);`
设置模块 `mod` 的 `lock` 为 `FALSE`,并使能休眠
5. `int hal_pwrmgr_register(MODULE_e mod, pwrmgr_Hdl_t sleepHandle, pwrmgr_Hdl_t wakeupHandle);`
注册模块 `mod`,并提供相应的休眠/唤醒回调函数
6. `int hal_pwrmgr_unregister(MODULE_e mod);`
取消注册模块 `mod`
7. `int hal_pwrmgr_wakeup_process(void) __attribute__((weak));`
8. `int hal_pwrmgr_sleep_process(void) __attribute__((weak));`
休眠/唤醒过程中 `PWR_MGR` 模块定义的处理函数，应用程序不需要也不应该调用它们
9. `int hal_pwrmgr_RAM_retention(uint32_t sram);`
配置需要保持的 `RAM` 区域，可选的有 `RET_SRAM0` | `RET_SRAM1` | `RET_SRAM2`
10. `int hal_pwrmgr_clk_gate_config(MODULE_e module);`
配置在唤醒时需要使能的时钟源。
11. `int hal_pwrmgr_RAM_retention_clr(void);`
12. `int hal_pwrmgr_RAM_retention_set(void);`
使能/清除对配置的 `RAM` 区域的保持(`retention`)功能
13. `int hal_pwrmgr_LowCurrentLdo_enable(void);`
14. `int hal_pwrmgr_LowCurrentLdo_disable(void);`
使能/禁用调节 `LowCurrentLDO` 的输出电压。
15. `int hal_pwrmgr_poweroff(pwroff_cfg_t *pcfg, uint8_t wakeup_pin_num);`
配置唤醒源后系统进入关机模式
16. `void wakeupProcess_standby(void);`
系统在 `standby` 模式下的唤醒函数。应用程序不需要也不应该调用它。
17. `void hal_pwrmgr_enter_standby(pwroff_cfg_t* pcfg,uint8_t wakeup_pin_num);`
让系统进入 `standby` 模式的 `API` 函数。应用程序需要在合适的时机调用它进入 `standby`

3 低功耗模式使用注意事项

为使用低功耗模式，编程时需要注意以下几个方面：

- **配置 CFG_SLEEP_MODE 宏：**

在工程中需要设置 CFG_SLEEP_MODE=PWR_MODE_SLEEP 以使能休眠模式，而在其它模式下系统将不会进入休眠

- **初始化 pwrmgr 模块：**

若使用低功耗模式，在系统初始化时须调用 hal_pwrmgr_init()以初始化 pwrmgr 模块

- **配置不同 RAM 的 retention 属性：**

若使用低功耗模式，在系统初始化时需要调用 hal_pwrmgr_RAM_retention()以在系统休眠后保留对应 memory 区域的内容。可选的 RAM 区域有 RET_SRAM0，RET_SRAM1，和 RET_SRAM2，用户根据需要自行指定要保留的 RAM 区域

- **选择模块 ID 并注册休眠或唤醒回调函数：**

PHY62 系列 SDK 在 mcu_phy_bumbee.h 文件中定义了一些 MODULE_e 类型的模块名/ID，在 pwrmgr 模块中作为模块 ID 使用。APP task 可以在 MOD_USR1 和 MOD_USR8 之间任选一个作为模块 ID 使用。

若要使用低功耗模式，APP task 在初始化时需要调用以下函数进行注册：

```
hal_pwrmgr_register(MODULE_e mod, pwrmgr_Hdl_t sleepHandle,
    pwrmgr_Hdl_t wakeupHandle);
```

其中 mod 就是模块 ID，是必须项，在 MOD_USR1 和 MOD_USR8 之间任选一个即可；

sleepHandle()和 wakeupHandle ()分别对应的是可选的休眠和唤醒的回调函数。

一个常用的做法是：用户可以在 sleepHandle()函数中做一些休眠唤醒使用的 pin 和相应属性的设置；而在 wakeupHandle()函数中做唤醒源的判定和初始化，以便系统唤醒后能恢复回到休眠前的状态

- **控制是否允许 APP task 进入休眠：**

在使用低功耗模式时，APPtask 可以调用 pwrmgr 模块提供的下列接口来查询或控制是否允许进入休眠：

```
hal_pwrmgr_is_lock(MODULE_e mod):  查询是否允许模块进入休眠；
hal_pwrmgr_lock(MODULE_e mod):    禁止模块进入休眠；
hal_pwrmgr_unlock(MODULE_e mod):  允许模块进入休眠
```

4 功耗评估及实测数据

4.1 功耗评估模型

为了方便评估功耗，我们引入下图中的一个休眠唤醒周期作为功耗模型来估算平均功耗。

在此模型中，一个休眠唤醒周期由休眠时间(X1)，唤醒时间(X2)，工作时间(X3)，射频发送时间(X4, Tx RF)，和射频接收时间(X5, Rx RF)五部分组成。而 Y1,Y2,Y3,Y4,和 Y5 则表示其对应的各个部分的功耗。

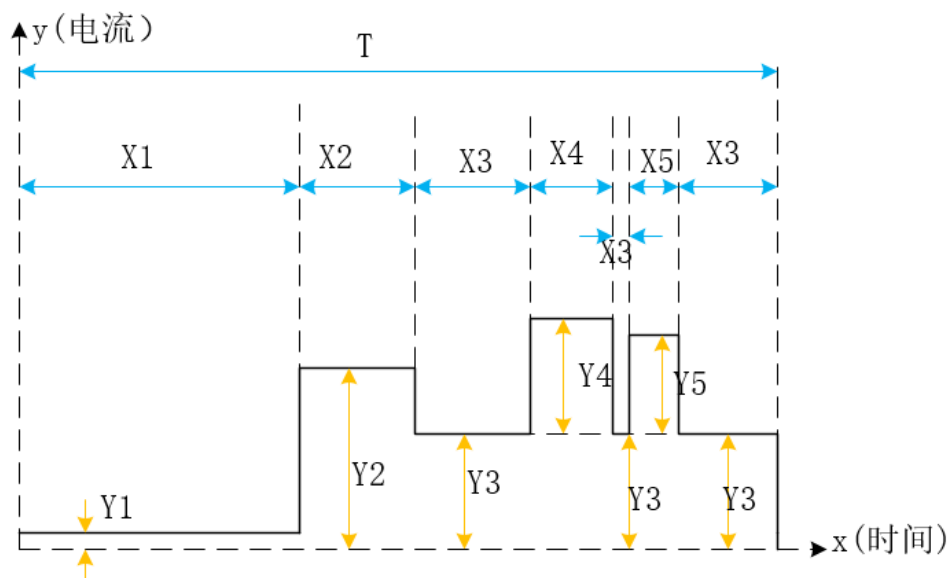


Figure 3: 功耗评估模型

其中:

X1: 休眠时间, 即系统处于休眠阶段的时间。这阶段对应的功耗为 Y1,影响较大的因素为 RAM retention 的数量, 需要 retention 的 RAM 越多, Y1 越大;

X2: 唤醒时间, 即系统被唤醒到系统时钟(hclk)切换之前的这段时间。这阶段对应的功耗为 Y2, 由于时钟源固定为 32m RC, Y2 的值相对稳定, 不过用户可以通过适当地调整唤醒时的参数, 缩小唤醒时间 X2, 从而达到减小这部分功耗的目的;

X3: 工作时间, 即系统切换好系统时钟且不处于射频发送/接收阶段的时间总和。这阶段对应的功耗为 Y3, 主要的影响因素为: 系统时钟 (16/32/48/64M) 和 LowCurrentLdo(工程里默认为 enable)。应用程序也需要尽量缩短这部分的时间以减小功耗;

X4: 射频发送时间, 即系统处于射频发送阶段的时间。这阶段对应的功耗为 Y4.对于实际的应用而言, 这个阶段是可选项, 可以没有, 有一个或多个。主要的影响因素为: PA 发射功率;

X5: 射频接收时间, 即系统处于射频接收阶段的时间。这阶段对应的功耗为 Y5.对于实际的应用而言, 这个阶段是可选项, 可以没有, 有一个或多个。接收功率相对稳定。

4.2 功耗估算

在 4.1 节描述的模型的基础上, 我们可以估算功耗如下:

平均功耗 = 总功耗 / 总时间

总功耗 = 休眠时的电流 X 休眠时间 + 唤醒时的电流 X 唤醒时间 + 工作电流 X (工作时间 + 射频发送时间 + 射频接收时间) + 射频发送时的电流 X 射频发送时间 + 射频接收时的电流 X 射频接收时间

总时间 = 休眠时间 + 唤醒时间 + 工作时间 + 射频发送时间 + 射频接收时间

电池使用时间 = 电池容量 X 3600 / 平均功耗 (秒)

4.3 实际测算的功耗

这里以 simpleBlePeripheral 工程为例，描述功耗估测的大致过程：

HCLK	RF time	wakeup time	work time	sleep time	total time	wakeup curt	RF curt	work curt	sleep curt	averag power
64M	676X3us	854us	728x3us	500ms	505ms	3.07ma	3.65ma	2.55ma	5.8ua	0.046
48M	684X3us	886us	738x3us	500ms	505ms	3.03ma	3.60ma	2.26ma	5.6ua	0.044
16M	683X3us	964us	736x3us	500ms	505ms	3.03ma	3.60ma	1.65ma	5.6ua	0.039

Table 1 advertising 功耗测算值

由于 simpleBlePeripheral 工程在唤醒后有三组射频发送/接收的过程，实际测试的波形图如下：

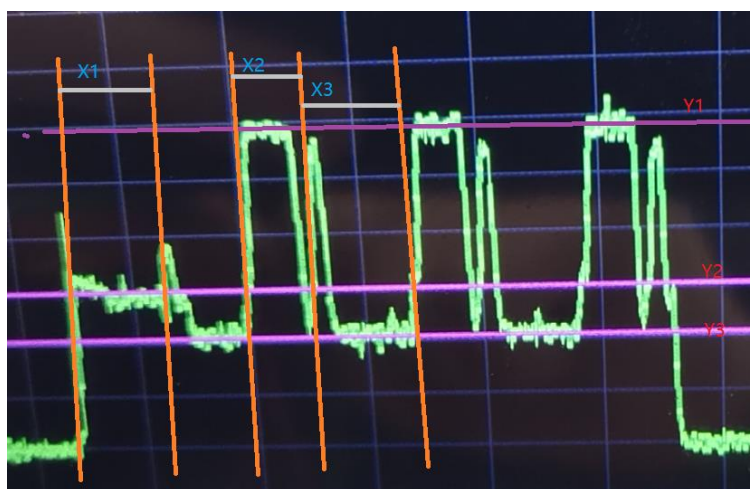


Figure 4: 功耗实测图

其中：

X1: 系统唤醒时间，对应的功耗为 Y2;

X2: 一次射频收发的估计时间，对应的功耗为(Y1 – Y3),对于 simpleBlePeripheral 工程，每个休眠唤醒周期有三组射频发送/接收的过程；

X3: 系统唤醒且切换好时钟后，到再次进入休眠之前的这段时间减去射频收发的总时间，对应的功耗为 Y3；

系统在休眠状态的时间及其功耗可以在功率计上直接读取。

参考上一节的功耗模型，可得

总时间 = 休眠时间 + 唤醒时间 + 工作时间 + 射频收发时间(Tx/Rx RF 的总时间)

总功耗 = 休眠时的电流 X 休眠时间 + 唤醒时的电流 X 唤醒时间 + 工作电流 X (工作时间 + 射频收发时间) + 射频发送接收时的电流 X 射频收发时间

平均功耗 = 总功耗 / 总时间

电池使用时间 = 电池容量 X 3600 / 平均功耗 (秒)

例如：电池容量为 520 mAh，平均功耗为 0.0398，则电池使用时间为

520 X 3600 / 0.0398 = 47035175 秒 = 13065 小时 = 544 天 = 1.5 年

4.4 系统上电启动时间

过程	时间	示例工程
冷启动	91ms	simpleBLEPeripheral
软启动	44ms	simpleBLEPeripheral
Wakeup	3.4ms	bleuart_at
Main 函数入口到 ble 初始化完成	31ms	simpleBLEPeripheral

Table 2 上电启动时间

系统上电可分成以下几种情况：

1. 冷启动，第一次上电的启动模式，启动时间与需要初始化的代码有关系，这份过程由 rom code 完成。
2. 软启动，software reset。可以配置 AON 寄存器绕过 dwc（大概耗时 50ms），剩余时间是代码初始化时间。
3. Wakeup，这份是从 sram 启动，系统回复速度最快，没有从 flash 搬运代码的时间和 dwc 的时间