PRBMD02 Application Note

Power management application note

Disclaimer 4

1. Introduction 5

2. PWR_MGR module API 6

- **2.1. data structure and type** 6
  - **2.1.1. MODULE_e type** 6
  - **2.1.2. pwrmgr_Ctx_t** 8
  - **2.1.3. pwroff_cfg_t** 8
- **2.2. APIs** 8

3. Point to note when using low power mode 9

4. Power consumption and result of actual test 10

- **4.1. Power consumption model** 10
- **4.2. Power consumption estimation** 11
- **4.3. Actual power consumption from test** 12
- **4.4. System startup time** 13

Disclaimer

Liability Disclaimer
K-Solution Consulting Co. Ltd reserves the right to make changes without further notice to the product to improve reliability, function or design. K-Solution Consulting Co. Ltd does not assume any liability arising out of the application or use of any product or circuits described herein.

Life Support Applications
K-Solution Consulting Co. Ltd's products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. K-Solution Consulting Co. Ltd customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify K-Solution Consulting Co. Ltd for any damages resulting from such improper use or sale.

1.Introduction

The functions related to PHY622X low power consumption are implemented in the PWR_MGR module, and the corresponding API codes are stored in pwrmgr.c and pwrmgr.h in the components\driver\ pwrmgr directory of the SDK.

There are FOUR power module for PRBMD02:

- Normal mode: CPU and peripherals run at full speed, no sleep
- CPU Sleep Mode: Only the CPU will go to sleep and can be woken up by interrupts or events. The mode is controlled by the OS itself without application intervention
- Deep Sleep Mode: The CPU and most peripherals will go to sleep. The application should set the sleep wakeup source (GPIO pin and trigger method) and memory retention (memory retention, to keep the runtime context) as needed
- Standby mode: Except for AON and a RAM area memory retention, the CPU and other peripherals go to sleep. Only the RAM0 area contents are maintained. Application should set wakeup source (GPIO pin and trigger method) as needed
- Shutdown Mode: Except for AON, the CPU and other peripherals go to sleep. After waking up, it is equivalent to a system restart, and the runtime context cannot be maintained. Application should set wakeup source (GPIO pin and trigger method) as needed

1.1.PWR_MGR Schematic diagram

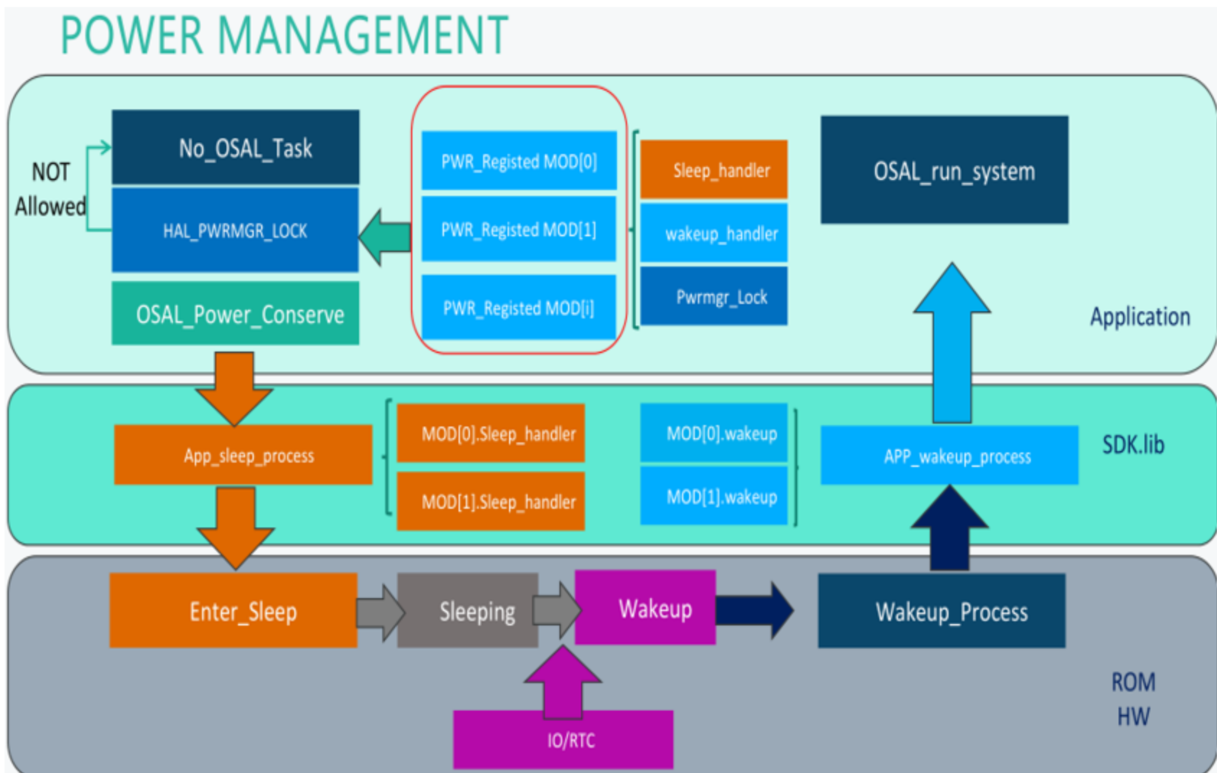


Figure 1: Deep Sleep Mode Block Diagram

In CPU deep sleep mode, when the system is in IDLE state, call sleep_process() to try to enter sleep mode. If the CPU sleep conditions are met, call __WFI() to wait for the interrupt to wake up and return; otherwise, the system will enter deep sleep. Before that, the sleep_handler() function registered by the application through the

hal_pwrmgr_register() API will be called back, and then the system will Go to sleep; and when IO/RTC triggers wake-up, the system will be woken up and initialized accordingly. During the system wakeup process, all wakeup_handler() functions registered by the application through the hal_pwrmgr_register() API will also be called back, and then the task will be scheduled by OSAL.

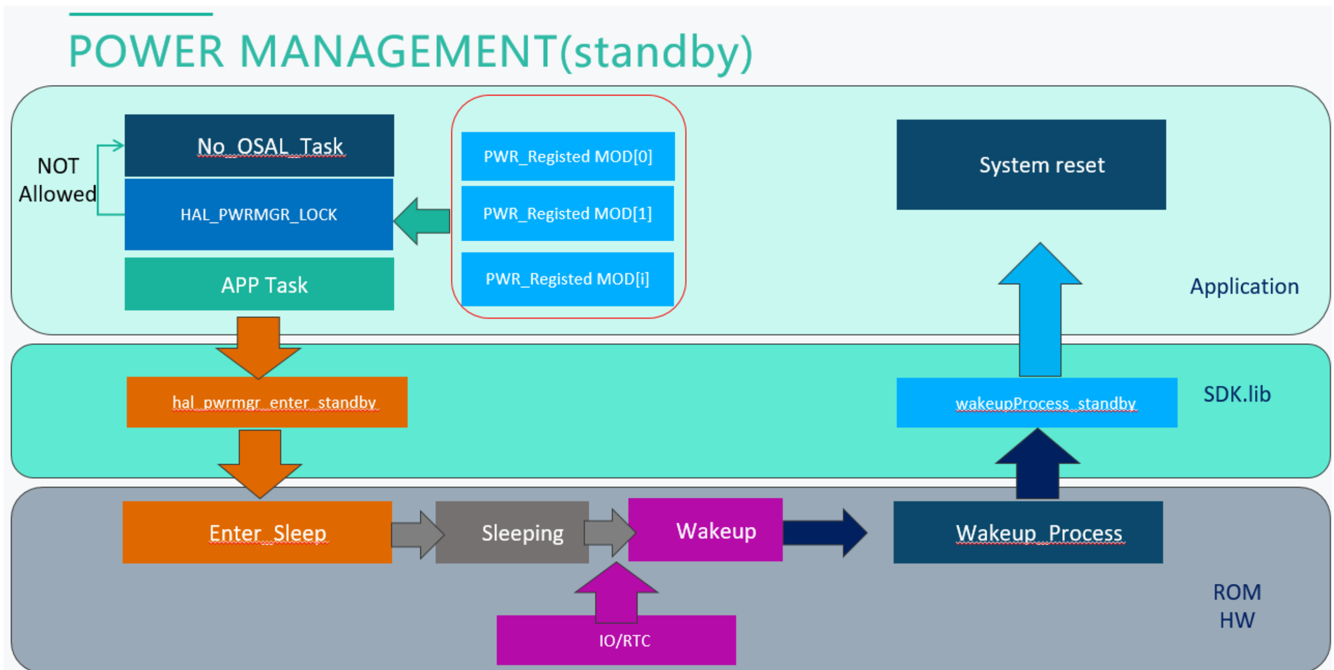


Figure 2: Standby Mode Block Diagram

In standby mode, APP Task calls hal_pwrmgr_enter_standby() to enter standby mode; When IO triggers wake-up, the system will wake up and call wakeupProcess_standby(). If the system meets the wake-up conditions, it will trigger system reset.

2.PWR_MGR module API

2.1. data structure and type

2.1.1.MODULE_e type

The following module IDs are defined in the mcu_phy_bumbee.h file.

```
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 + 8,
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

The PWR_MGR module maintains a variable of this structure type for each registered module (corresponding to MODULE_e). Up to 10.

```
typedef struct _pwrmgr_Context_t {
    MODULE_e            moudle_id;
    bool lock;       // When it is TRUE, it means that sleep is prohibited; otherwise,
    sleep is allowed pwrmgr_Hdl_t sleep_handler; // The callback function
    corresponding to this module that will be called before going to sleep
    pwrmgr_Hdl_t wakeup_handler; // The callback function that will be called before
    the module's corresponding wakeup
} pwrmgr_Ctx_t;
```

2.1.3.pwroff_cfg_t

Before the system calls the hal_pwrmgr_poweroff() API to enter the power-off mode, the wake-up source (GPIO pin) and trigger mode that need to be set are stored in this type of variable.

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

2.2.APIs

The interface functions of the PWR_MGR module are as follows:

1. int hal_pwrmgr_init(void);
   module initialisation
2. bool hal_pwrmgr_is_lock(MODULE_e mod);
   Query the lock status of module mod. TRUE: disable sleep; FALSE: enable sleep
3. int hal_pwrmgr_lock(MODULE_e mod);
   Set module mod's lock to TRUE and disable sleep
4. int hal_pwrmgr_unlock(MODULE_e mod);
   Register the module mod and provide the corresponding sleep/wake callback function
5. int hal_pwrmgr_register(MODULE_e mod, pwrmgr_Hdl_t sleepHandle, pwrmgr_Hdl_t wakeupHandle);
   Register the module mod and provide the corresponding sleep/wake callback function
6. int hal_pwrmgr_unregister(MODULE_e mod);
   unregister module mod
7. int hal_pwrmgr_wakeup_process(void) __attribute__((weak));
8. int hal_pwrmgr_sleep_process(void) __attribute__((weak));

Handler functions defined by the PWR_MGR module during sleep/wakeup, the application does not need and should not call them

9. int hal_pwrmgr_RAM_retention(uint32_t sram);
   Configure the RAM area that needs to be kept, optional RET_SRAM0 | RET_SRAM1 | RET_SRAM2
10. int hal_pwrmgr_clk_gate_config(MODULE_e module);
   Configure the clock source that needs to be enabled on wakeup.
11. int hal_pwrmgr_RAM_retention_clr(void);
12. int hal_pwrmgr_RAM_retention_set(void);
   Enable/Clear Retention for configured RAM regions
13. int hal_pwrmgr_LowCurrentLdo_enable(void);
14. int hal_pwrmgr_LowCurrentLdo_disable(void);
   Enable/disable regulation of the output voltage of the LowCurrentLDO.
15. int hal_pwrmgr_poweroff(pwroff_cfg_t *pcfg, uint8_t wakeup_pin_num);
   After configuring the wake-up source, the system enters shutdown mode
16. void wakeupProcess_standby(void);
   The wake-up function of the system in standby mode. The application does not need and should not call it.
17. void hal_pwrmgr_enter_standby(pwroff_cfg_t* pcfg,uint8_t wakeup_pin_num);
   API function to put the system into standby mode. The application needs to call it to enter standby at the right time

3. Point to note when using low power mode

In order to use the low power mode, the following aspects need to be paid attention to when programming:

- **Configure the CFG_SLEEP_MODE macro:**
  In the project, you need to set CFG_SLEEP_MODE=PWR_MODE_SLEEP to enable sleep mode, and the system will not enter sleep mode in other modes
- **Initialize the pwrmgr module:**
  To use low power mode, call hal_pwrmgr_init() to initialize the pwrmgr module during system initialisation
- **Configure retention properties for different RAMs:**
  To use the low power consumption mode, you need to call hal_pwrmgr_RAM_retention() during system initialization to retain the contents of the corresponding memory area after the system sleeps. The optional RAM areas are RET_SRAM0, RET_SRAM1, and RET_SRAM2. Users can specify the RAM area to be reserved according to their needs
- **Select the module ID and register the sleep or wake callback function:**

The PHY62 series SDK defines some module names/IDs of type MODULE_e in the mcu_phy_bumbee.h file, which are used as module IDs in the pwrmgr module. APP task can use any one between MOD_USR1 and MOD_USR8 as the module ID.
To use the low power mode, the APP task needs to call the following functions to register when it is initialized:

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

Where mod is the module ID, which is a required item, you can choose one between MOD_USR1 and MOD_USR8; sleepHandle() and wakeupHandle() correspond to the optional sleep and wake-up callback functions respectively. A common practice is: the user can set some pins and corresponding properties for sleep wake-up in the sleepHandle() function; and determine and initialize the wake-up source in the wakeupHandle() function, so that the system can return to the state after waking up. state before sleep

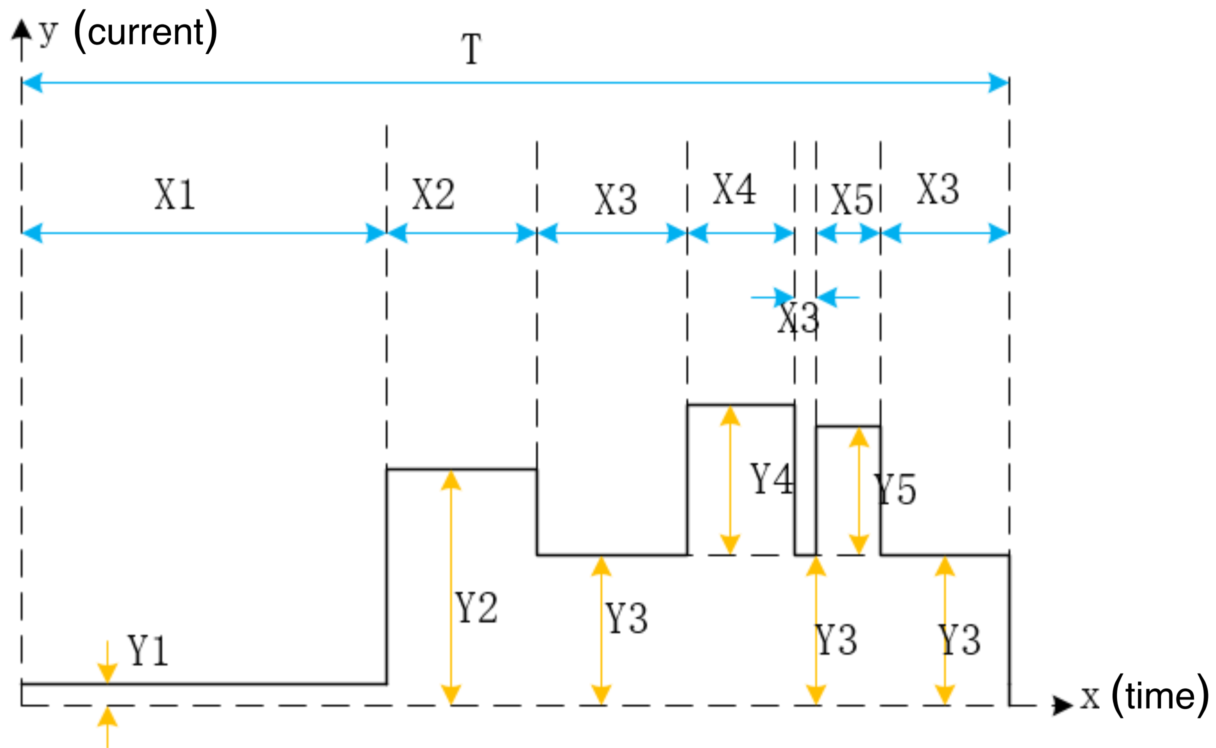
- **Controls whether APP task is allowed to sleep:**
  When using low power mode, APPtask can call the following interfaces provided by the pwrmgr module to query or control whether to allow sleep:
  hal_pwrmgr_is_lock(MODULE_e mod): Query whether the module is allowed to go to sleep;
  hal_pwrmgr_lock(MODULE_e mod): prohibit power module enter sleep;
  hal_pwrmgr_unlock(MODULE_e mod): allow power module enter sleep

4. Power consumption and result of actual test

4.1. Power consumption model

To facilitate power consumption estimation, we introduce a sleep-wake cycle in the figure below as a power consumption model to estimate average power consumption. In this model, a sleep-wake-up cycle consists of five parts: sleep time (X1), wake-up time (X2), working time (X3), RF transmission time (X4, Tx RF), and RF reception time (X5, Rx RF) composition. And Y1, Y2, Y3, Y4, and Y5 represent the power consumption of their corresponding parts.

Fig3: power consumption estimation model



where:
X1: Sleep time, that is, the time when the system is in the sleep phase. The power consumption corresponding to this stage is Y1, and the most influential factor is the amount of RAM retention. The more RAM that needs to be retained, the larger Y1 is;
X2: Wake-up time, which is the period of time before the system is woken up until the system clock (hclk) is switched. The corresponding power consumption at this stage is Y2. Since the clock source is fixed at 32m RC, the value of Y2 is relatively stable. However, the user can reduce the wake-up time X2 by properly adjusting the parameters during wake-up, so as to achieve the purpose of reducing this part of the power consumption. ;
X3: Working time, that is, the total time when the system has switched the system clock and is not in the RF transmit/receive phase. The power consumption corresponding to this stage is Y3, and the main influencing factors are: system clock (16/32/48/64M) and LowCurrentLdo (the default is enable in the project). The application also needs to shorten this part of the time as much as possible to reduce power consumption;
X4: RF transmission time, that is, the time when the system is in the RF transmission phase. The corresponding power consumption of this stage is Y4. For practical applications, this stage is optional, there may be none, one or more. The main influencing factors are: PA transmit power;
X5: RF reception time, that is, the time when the system is in the RF reception stage. The power consumption corresponding to this stage is Y5. For practical applications, this stage is optional, there may be none, one or more. The received power is relatively stable.

4.2. Power consumption estimation

On the basis of the model described in Section 4.1, we can estimate the power consumption as follows:

Average Power Consumption = Total Power Consumption / Total Time
Total power consumption = current during sleep X sleep time + current during wake-up X wake-up time + working current X (working time + RF sending time + RF receiving time) + current during RF sending X RF sending time + RF receiving time Current X RF reception time

Total time = sleep time + wake-up time + working time + RF transmit time + RF receive time

Battery life = battery capacity X 3600 / average power consumption (seconds)

4.3. Actual power consumption from test

Here we take the simpleBlePeripheral project as an example to describe the general process of power consumption estimation:

Since the simpleBlePeripheral project has three sets of RF transmission/reception processes after waking up, the actual test waveform is as follows:

Since the simpleBlePeripheral project has three sets of RF transmission/reception processes after waking up, the actual test waveform is as follows:

Table 1: power estimation on advertising

HCLK	RF time	wake up time	work	sleep time	total time	wakeup curt	RF curt	work curt	sleep curt	avg power
64M	676X3us	854us	728x3us	500ms	505ms	3.07mA	3.65mA	2.55mA	5.8uA	0.046mA
48M	684X3us	886us	728x3us	500ms	505ms	3.03mA	3.60mA	2.26mA	5.6uA	0.044mA
16M	683X3us	964us	728x3us	500ms	505ms	3.03mA	3.60mA	1.65mA	5.6uA	0.039mA

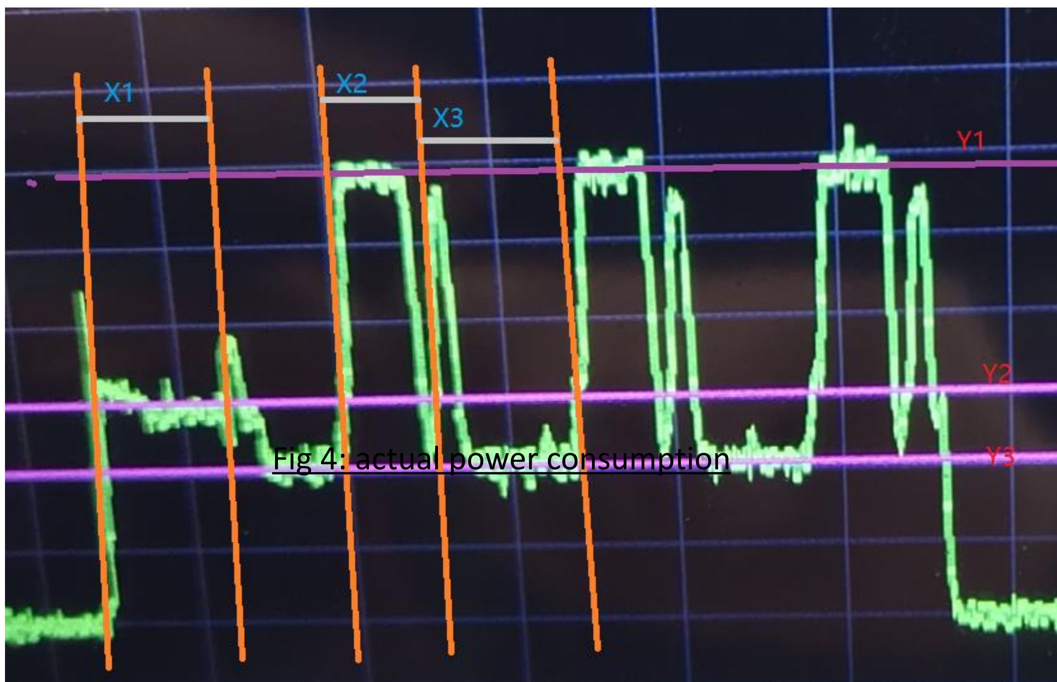


Fig 4: actual power consumption

where:

X1: System wake-up time, the corresponding power consumption is Y2;
X2: The estimated time of one RF transmission and reception, and the corresponding power consumption is (Y1 – Y3). For the simpleBlePeripheral project, there are three groups of RF transmission/reception processes in each sleep-wake cycle;
X3: After the system wakes up and the clock is switched, the total time for radio frequency transmission and reception is subtracted from the time before going to sleep again, and the corresponding power consumption is Y3;
The time the system is in sleep state and its power consumption can be read directly on the power meter.

Referring to the power consumption model in the previous section, we can get

Total time = sleep time + wake-up time + working time + RF transceiver time (total time of Tx/Rx RF)
Total power consumption = sleep current X sleep time + wake-up current X wake-up time + working current X (working time + RF transceiver time) + RF transmission and reception current X RF transceiver time
Average power consumption = total power consumption / total timeSystem power up timing

Battery usage time = battery capacity X 3600 / average power consumption (seconds)
For example: battery capacity is 520 mAh, average power consumption is 0.0398, then battery usage time is 520X3600/0.0398=47035175 seconds = 13065 hours = 544 days = 1.5 years

4.4.System startup time

process	time	demo process
cold reset	91ms	simpleBLEPeripheral
soft reset	44ms	simpleBLEPeripheral
Wakeup	3.4ms	bleuart_at
from Main function to BLE initialisation complete	31ms	simpleBLEPeripheral

Table 2: power up start time

System power-on can be divided into the following situations:

1. Cold boot, the startup mode of the first power-on, the startup time is related to the code that needs to be initialized, and this process is completed by the rom code.
2. Soft start, software reset. The AON register can be configured to bypass the dwc (about 50ms), and the remaining time is the code initialization time.
3. Wakeup, this one starts from sram, the system recovers the fastest, there is no time to move code from flash and dwc time