

WS8100

应用指南

修订记录

修订版本	描述	作者
1.0	初稿完成	Zxb
1.1	增加快速上手及 AT 指令章节	Zxb
1.2	增加 OTA 章节详细描述	Zxb
1.3	1、补充 DEEP_SLEEP+ 模式 待机 2、增加串口 IO 注意事项	Zxb
1.4	1、增加 ota 对 boot 要求描述 2、更新到最新文件 3、增加串口驱动，keil 安装 要求 4、移除旧版本 ota 升级方式 5、修复 at 指令格式	Zxb

目录

1、 文档目的作用范围.....	9
2、 复位/时钟.....	9
2.1 NST 复位电平.....	9
2.2 NST 电路处理.....	9
2.3 晶振时钟.....	9
2.4 外设时钟及复位.....	10
2.4.1 外设时钟/复位例程代码.....	10
2.5 复位重启.....	13
2.6 看门狗复位.....	14
3、 仿真/下载.....	15
3.1 Jlink 接线方式.....	15
3.2 KIEL 编译配置.....	15
3.2.1 Kiel 安装.....	15
3.2.2 Kiel 链接配置.....	16
3.2.3 程序调试配置.....	16
3.3 KIEL 下载.....	17
3.4 串口下载.....	18
3.4.1 串口驱动.....	18
3.4.2 调试烧录.....	18
3.4.3 量产烧录.....	19
3.4.4 用户自定义蓝牙地址，蓝牙名称量产烧录.....	20
3.5 单步调试.....	21
3.6 串口调试.....	21
3.6.1 调试接口.....	21
3.6.2 推荐调试方法：.....	21
4、 内存部分.....	22
4.1 WS8100 RAM 共 40K,其中掉电不丢失 RAM 24K ,掉电不保留数据 16K。	22
4.2 栈顶：	22
4.3 重量级任内存.....	22
4.4 具备内存回收能力的函数.....	23

4.4.1 释放服务.....	23
4.4.2 接收数据回应.....	23
4.4.3 发送蓝牙通知数据.....	23
4.4.4 写蓝牙数据.....	23
4.4.5 恢复配对密钥.....	23
4.4.6 命令发送.....	23
4.5 内存申请，内存释放.....	23
5、任务部分.....	23
5.1 定时任务.....	23
5.2 等待 cpu 空闲任务.....	23
6、低功耗.....	24
6.1 低功耗配置例程.....	24
6.2 低功耗运行例程.....	26
6.3 DEEP_SLEEP 进入外设掉电前，关键外设备份，gpio 省电操作。.....	29
6.4 即将进入 DEEP_SLEEP，外设掉电前，gpio 省电操作。.....	30
6.5 退出 DEEP_SLEEP 恢复外设，恢复关键区域.....	30
6.6 DEEP_SLEEP 模式 外设正常工作.....	31
6.6.1 设置外设工作事件.....	32
6.6.2 清除外设工作事件.....	33
6.6.3 示例代码.....	33
6.7 退出 DEEP_SLEEP 恢复外设，恢复用户外设，及应用数据.....	33
6.8 DEEP_SLEEP 模式 用户中断.....	34
6.8.1 蓝牙中断事件.....	34
6.8.2 常开电源域外设中断事件.....	34
6.8.4 固定打断事件.....	34
6.9 DEEP_SLEEP 模式 漏电流调试.....	35
6.9.1 漏电流查找.....	35
6.9.2 漏电流处理方法.....	35
6.10 浅睡眠低功耗，CPU 运行频率仍然不够.....	35
7、待机.....	35
7.1 DEEP_SLEEP+ 模式待机.....	36

7.2 DEEP_SLEEP 模式待机.....	36
7.2.1 进入 DEEP_SLEEP 模式待机示例代码.....	36
7.2.2 退出 DEEP_SLEEP 模式待机示例代码.....	38
8、 中断.....	41
8.1 WS8100 中断命名方法.....	41
8.2 WS8100 中断管理.....	41
8.3 中断应用示例代码.....	41
8.4 WS8100 清中断.....	43
9、 SYS_STICK 及 RTC.....	43
9.1 SYS_STICK 示例代码.....	43
9.2 RTC 示例代码.....	46
10、 GPIO.....	47
10.1 GPIO 简介.....	47
10.2 GPIO 映射方法.....	47
10.3 WS8100 GPIO 的两种表示方法.....	48
10.3.1 独立 GPIO 示意.....	48
10.3.2 集合示意.....	48
10.4 WS8100 GPIO 输出的两种方式.....	48
10.4.1 低功耗模式下的输出写法.....	48
10.4.2 常规模式的写法.....	49
10.5 GPIO 中断.....	50
10.5.1 中断服务函数.....	50
10.5.2 低功耗中断注册.....	51
10.5.3 普通 gpio 中断注册，低功耗情况无法唤醒.....	52
10.5.4 GPIO 中断应用示例代码.....	53
11、 UART 应用.....	55
11.1 设置 UART 工作状态.....	55
11.2 APP_USART_IRQ 示例代码.....	56
11.3 注意事项.....	58
12、 TIMER.....	58
12.1 设置 TIMER 工作状态.....	59
12.2 TIMER 示例代码.....	59
13、 ADC.....	61

13.1	ADC 简介.....	61
13.2	ADC 查询模式示例代码.....	61
13.3	中断模式示例代码.....	62
14、	PWM.....	63
14.1	PWM 简介.....	63
14.2	PWM 示例代码.....	63
15、	SPI.....	66
15.1	SPI 简介.....	66
16、	I2C.....	66
16.1	I2C 简介.....	66
16.2	I2C 示例代码.....	66
17、	FLASH 读写.....	68
17.1	FLASH 擦除.....	68
17.2	写操作.....	69
17.3	读操作.....	69
17.4	Flash 读写示例代码：.....	69
17.5	加载量产工具用户自定义蓝牙名称.....	70
17.6	加载量产工具用户自定义蓝牙地址.....	71
18、	蓝牙部分.....	72
18.1、	全向广播.....	72
18.1.1、	广播类型.....	72
18.1.2、	广播包.....	74
18.2.3、	广播回应包.....	74
18.3.4、	广播间隔.....	75
18.2、	定向广播.....	75
18.3、	停止广播.....	75
18.4、	发射功率.....	75
18.5、	蓝牙连接.....	76
18.5.1、	连接上事件.....	76
18.5.2、	断开连接事件.....	77
18.6、	蓝牙安全.....	77
18.7、	蓝牙地址.....	77
18.7.1	蓝牙开机地址设置示例代码.....	78
18.7.2	蓝牙用户地址设置示例代码.....	79

18.7.3 蓝牙随机地址设置示例代码.....	79
18.9、蓝牙名称.....	80
18.9.1 修改蓝牙名称示例代码:.....	80
18.10、蓝牙 UUID.....	80
18.10.1 修改蓝牙 UUID 广播示例代码:.....	80
18.10.2 GAP 层 UUID:.....	81
18.11、蓝牙服务.....	84
18.11.1 添加服务.....	84
18.11.2 保存服务中的所有句柄，并释放服务实例.....	88
18.12、蓝牙发送.....	93
18.12.1 通知方式发送.....	93
18.12.2 写数据方式发送.....	94
18.13、蓝牙接收.....	95
18.13.1 蓝牙接收函数注册.....	95
18.13.2 蓝牙接收数据.....	96
19、蓝牙连接断开常见错误处理.....	96
20、异常错误处理.....	97
21、晶振频偏校准.....	97
21.1、校准文件.....	97
21.2、校准烧录方法.....	97
21.3、校准流程.....	97
21.4、校准结果.....	98
21.5、校准结果写入程序.....	98
22、OTA 升级.....	99
22.1、OTA 简介.....	99
22.2、OTA 升级开启.....	99
22.3.1、创建固件.....	99
22.3.2、升级操作.....	100
22.4、密钥创建.....	102
22.5、固件 OTA 密钥配置.....	102
22.5、OTA 编译固件版本.....	102
23、SDK 快速上手.....	103

23.1、程序下载运行.....	103
23.2、程序修改.....	103
23.4、逻辑应用流程简介.....	103
24、WS8100 AT 指令.....	105
24.1、透传模块工作方式.....	105
24.2、AT 指令集格式.....	107
24.3、AT 测试指令.....	107
24.3、AT 测试指令生成.....	108
24.4、AT 测试指令使用.....	109
24.5、WS8100 透传模块 SDK 宏定义配置简介.....	109

1、文档目的作用范围

本文档作用范围为开启蓝牙，及低功耗的应用程序开发过程。

不带蓝牙，及低功耗例程，直接基于 MCU_DEMO 例程开发。

基于文档，文件中没有描述的外设，请将 MCU_DEMO 例程中的例程移植到蓝牙工程。

2、复位/时钟

2.1 NST 复位电平

WS8100 为低电平复位，默认高电平。

2.2 NST 电路处理

WS8100 内部自带上拉电阻，不可接下拉电阻。

2.3 晶振时钟

WS8100 核心高频支持 16M 或 32M 时钟。

WS8100 低功耗低频支持内置 32K 时钟或外置 32K 时钟。

/******

函 数 名 : syscfg_configuration

功能描述 : 系统配置

输入参数 : void

输出参数 : 无

返 回 值 :

*****/

void syscfg_configuration(void)

{

#if(USE_EXT_XTAL_16M)

// 16M 晶振

syscfg_external_16M_xtal();

//< 配置晶振匹配电容,外部无须电容

syscfg_set_oscxtal_config(RF_ADJUST);

#endif

#if(USE_EXT_XTAL_32M)

```
// 32M 晶振

syscfg_external_32M_xtal();

//< 配置晶振匹配电容，外部无须电容

syscfg_set_oscxtal_config(0x57);

#endif

#if (USE_EXT_XTAL_32K)

// 外部 32K 晶振

syscfg_external_32k_xtal();

radio_set_32k_freq(32768);

radio_set_32k_ppm(20);

#endif

#if (USE_INTER_RC_32K)

// 内部 32K 晶振

syscfg_internal_32k_rc();

radio_set_32k_ppm(300);

#endif

// 配置低功耗模式

low_power_sleep_config_ex(SYSCFG_Retention_Sram_24k,SDK_ENABLE_SLEEP);

}
```

2.4 外设时钟及复位

芯片上电初始化外设，使能时钟后，需要复位。

2.4.1 外设时钟/复位例程代码

```
/**
 *
 * 功能描述：外设时钟列表
 *
 */

const uint32_t ws_clock[18][3]=

{

//蓝牙时钟
```

```
        { ws_bt,    CRM_PeriphClock_BT,    CRM_PeriphReset_BT    },
//ADC 时钟
        { ws_adc,    CRM_PeriphClock_QPADC,  CRM_PeriphReset_GPADC },
//QDEC 时钟
        { ws_qdec,    CRM_PeriphClock_QDEC,  CRM_PeriphReset_QDEC   },
//GPIO 时钟
        { ws_gpio,    CRM_PeriphClock_GPIO,  CRM_PeriphReset_GPIO   },
//RTC 时钟
        { ws_rtc,    CRM_PeriphClock_RTC,    CRM_PeriphReset_RTC    },
//WDT 时钟
        { ws_wdt,    CRM_PeriphClock_WDT,    CRM_PeriphReset_WDT    },
//定时器 3 时钟
        { ws_timer3, CRM_PeriphClock_TIMER23,CRM_PeriphReset_TIMER23 },
//定时器 2 时钟
        { ws_timer2, CRM_PeriphClock_TIMER23,CRM_PeriphReset_TIMER23 },
//定时器 1 时钟
        { ws_timer1, CRM_PeriphClock_TIMER01,CRM_PeriphReset_TIMER01 },
//定时器 0 时钟
        { ws_timer0, CRM_PeriphClock_TIMER01,CRM_PeriphReset_TIMER01 },
//硬件随机数时钟
        { ws_rng,    CRM_PeriphClock_RNG,    CRM_PeriphReset_RNG    },
//系统配置时钟, 暂时不要使用
        { ws_syscfg,CRM_PeriphClock_SYSCFG,  CRM_PeriphReset_SYSCFG },
//键盘矩阵时钟
        { ws_kpc,    CRM_PeriphClock_KPC,    CRM_PeriphReset_KPC    },
//SPI1 时钟
        { ws_spi1,    CRM_PeriphClock_SPI1,    CRM_PeriphReset_SPI1    },
//SPI0 时钟
        { ws_spi0,CRM_PeriphClock_SPI0,    CRM_PeriphReset_SPI0    },
//串口 1 时钟
        { ws_uart1,CRM_PeriphClock_UART1,  CRM_PeriphReset_UART1   },
//串口 0 时钟
```

第 12 页 共 114 页

```
    }

}

if(found == 0){
    dbg_printf("err config clock arg invaild\n");
}

}

/*****

void user_uart_example(void)
    功能描述   ：时钟举例

    输入参数   ：void

    输出参数   ：无

*****/

void user_uart_example(void)
{
    dbg_block_printf("app service initial timer ready\r\n");

    user_uart_cfg_parameter_t uart_param;

    // 【1】 开启串口 0 时钟

    clock_config(ws_uart1,ENABLE);

/*
* To DO
*/
}
```

2.5 复位重启

```
/*****

    功能描述   ：管理单元复位

*****/

const uint32_t reset[]=
{
    //片内复位，重启

    CRM_PeriphReset_GLB,

    //CPU 内核复位

    CRM_PeriphReset_CM3,
```

//时钟复位管理单元复位

CRM_PeriphReset_CRM,

};

/*****

函 数 名 : system_rest

功能描述 : 系统复位

输入参数 : WS8100_RST rst

输出参数 : 无

返 回 值 :

调用函数 :

被调函数 :

*****/

void system_rest(WS8100_RST rst)

{

crm_reset_periph(reset[rst]);

}

2.6 看门狗复位

WS8100 看门狗复位分为 2 种。

系统复位模式: 喂狗超时直接复位

中断模式: 喂狗超时产生不可屏蔽中断, 用户没有处理, 继续超时, 系统复位。

具体情况查询 WS8100 芯片数据手册。

#define WDT_TIMEOUT 12000 // 12s watchdog timeout

/*****

函 数 名 : wdt_user_example

功能描述 : 看门狗例程

输入参数 : void

输出参数 : 无

返 回 值 :

*****/

void wdt_user_example(void)

{

//【1】开启 wdt 时钟

```

        clock_config(ws_wdt,ENABLE);

// 【2】 设置超时时间

        wdt_set_reload(WDT_TIMEOUT*16000);

// 【3】 设置看门狗超时触发 CPU 复位

        wdt_mode_config(WDT_Mode_CPUReset);

// 【4】 使能看门狗

        wdt_enable();

    }

/*

*喂狗示例

*/

    while(1)
    {

        lib_schedule(); /* run task schedule. */

        feed_dog();      /* run feed dog */

        low_power_sleep(lib_sleep);

    }
    
```

3、仿真/下载

3.1 Jlink 接线方式

WS8100 支持 jlink swd 模式仿真

接线方式

JLINK			WS8100	
1	VCC			VCC
7	TMS			TMS
9	TCLK			TCK
15	RST			RST
20	GND			GND

RST 可不接，遇到不可预料的错误，将 RST 线接好。

3.2 KIEL 编译配置

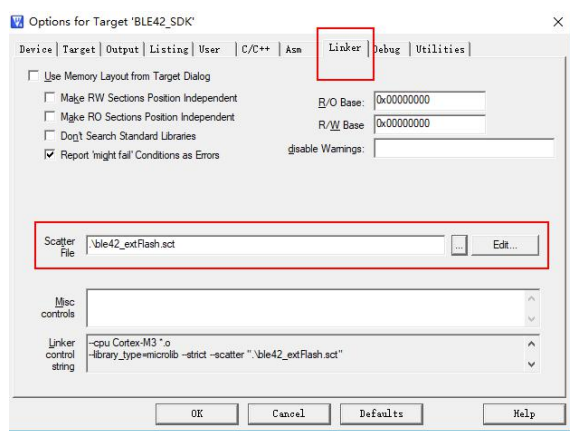
3.2.1 Kiel 安装

Keil 下载地址 TOOLS\KEIL 工具\Keil 目录下 Keil 工具说明.txt

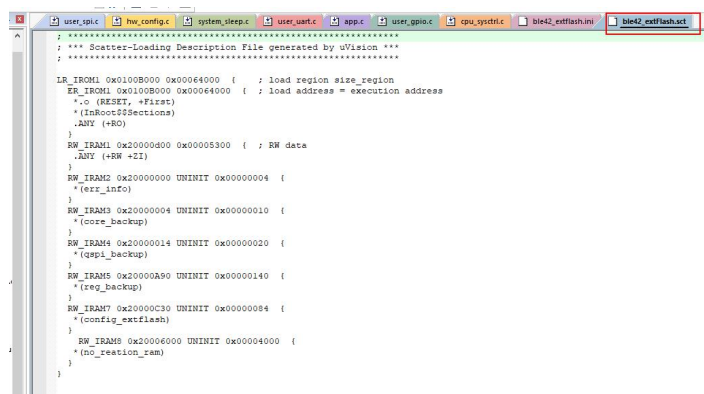
保持和我们一致，我们所有代码用 keil 4.72 编译验证过。

3.2.2 Kiel 链接配置

程序链接配置文件

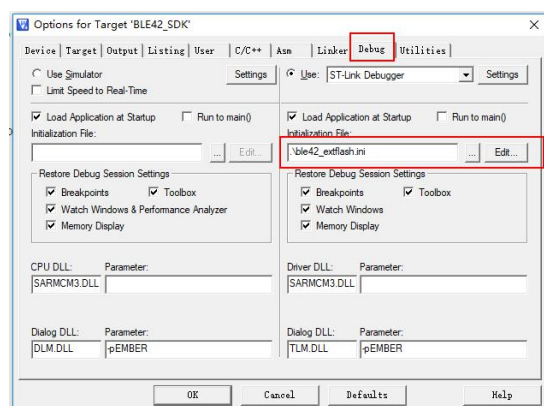


配置内容

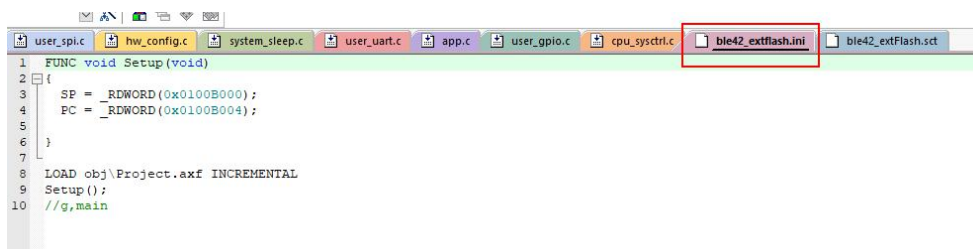


3.2.3 程序调试配置

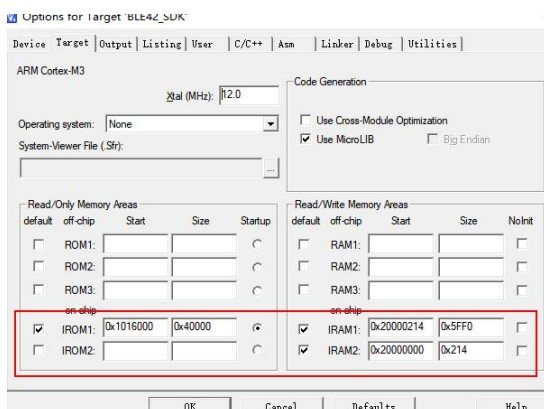
程序调试配置文件【和 project 同级目录】



程序调试配置内容

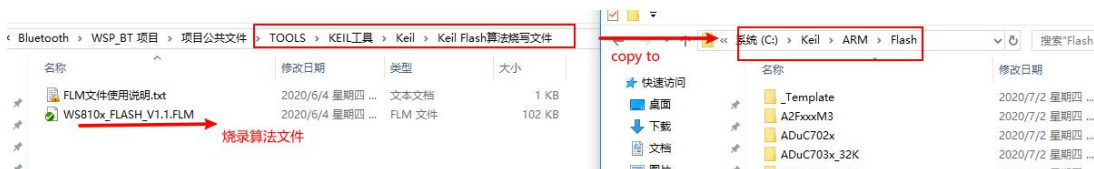


WS8100 制定了配置文件，故不采用下图中 IROM1 和 IRAM1 等的配置。

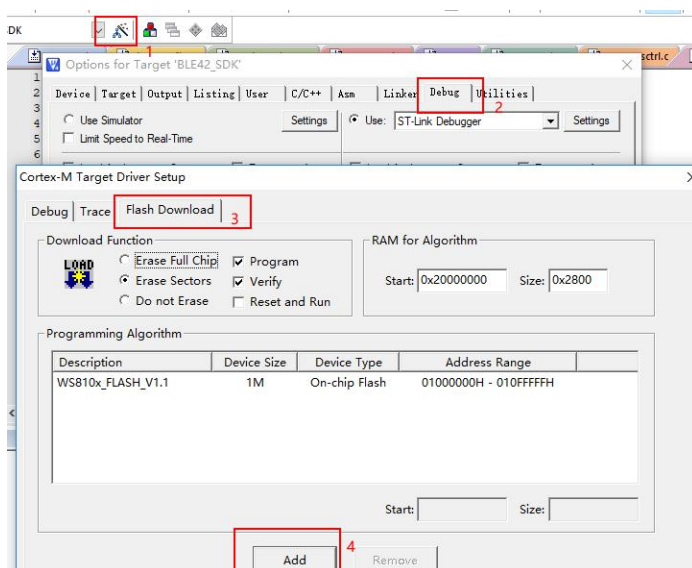


3.3 KIEL 下载

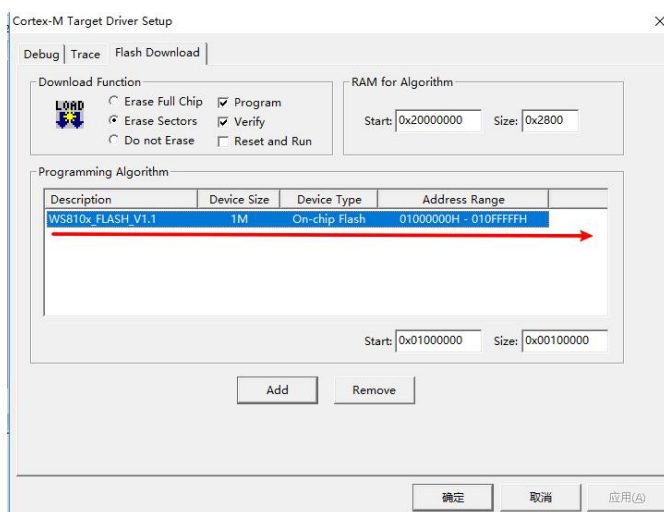
拷贝 ws8100 烧录 配置算法文件到 keil flash 目录中



配置 keil 烧录算法文件



添加完成后的效果



要使用编译器的烧录规则检测目标文件是否超出了 flash 大小可以自行将 0x00100000 修改芯片实际 flash 大小。

3.4 串口下载

ws8100 支持串口线下载。



3.4.1 串口驱动

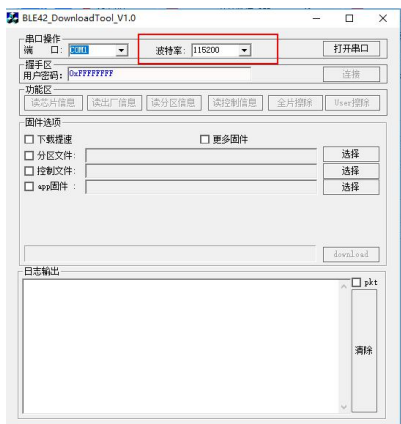
WS8100 串口驱动统一用，TOOLS 目录下串口驱动文件。

3.4.2 调试烧录

3.4.2.1 普通调试串口接线

UART		WS8100	
2	3V3		VCC
3	TXD		RXD
4	RXD		TXD
5	RTS		RST
6	GND		GND

3.4.2.2 波特率选择



3.4.2.3 烧录文件

32M 晶振选择 230400 波特率 16M 晶振选择 115200 波特率!!!

首次烧录需要烧录分区、控制、app、boot 四个文件，后面只需要烧录 app 即可。

8100 支持串口烧录加密功能，默认密码 0xFFFFFFFF,加密信息在控制文件中，一旦选错了文件，可能造成误锁被加密!!! 部分公司有加密系统，需要注意这一点。

Boot 文件: WS8100_ble_boot_v1.3.hex

分区文件: ws8100_ble_part_config_v1.3.cfg

控制文件: ws8100_ble_access_config_v1.3.cfg

应用程序: ws8100_ble_app_ws_V1.30_Project.hex

3.4.3 量产烧录

3.4.3.1 量产工具接线

UART			WS8100	
1	RXD			TXD
2	TXD			RXD
3	GND			GND
4	VCC			VCC

使用量产工具，不要接它供电源。IO 口可以为芯片引入供电!!!，需要断开

3.4.3.1 量产工具配置

配置文件: config.ini

配置语法: [xx] = "content"

屏蔽方式: ;[stack] = "stack.hex";STACK 固件

主要修改内容:

[freq] = "16000000"

[app] = "^WS8%d+_w?%w+_app_.*%.hex\$" ;APP 固件

:[stack] = "^WS8%d+_w?%w+_stack.*_v%x+%.hex\$" ;STACK 固件

[boot] = "^WS8%d+_w?%w+_boot.*%.hex\$" ;BOOT 固件

[part] = "^WS8%d+_w?%w+_part.*%.cfg\$" ;分区配置

[userdata] = "^WS8%d+_w?%w+_userdata.*%.cfg\$" ;USERDATA

[factory] = "^WS8%d+_w?%w+_product_.*%.hex\$" ;芯片出厂射频校准与产

测固件

[disable_rftest] = "true" ;dongle 射频测试 量产时候用到

[check_io] = "false" ;模块 io 连锡检测

上述正则表达式可以直接替换成 hex 实际名称 如

ws8100_ble_app_BM880_V1.31_Project(1).hex

正则表达式要求文件命名规则 ws8100_xx_app_xxx.hex

正则表达式文件名称屏蔽方式 文件名称前加 ++ 即可排除在正则表达式外。

3.4.4 用户自定义蓝牙地址，蓝牙名称量产烧录

3.4.4.1 量产工具接线

2.4.2.1 描述

3.4.4.2 量产工具配置

配置文件: config.ini

配置语法: [xx] = "content"

屏蔽方式:

:[stack] = "stack.hex";STACK 固件

主要修改内容:

[ome_script] = "user_oem.lua"

3.4.4.3 蓝牙地址名称规则定义

脚本文件:user_oem.lua

蓝牙地址起始及变化规律:

```

00027:
00028:    --起始地址
00029:    --若stroages不存起始地址字段, 则创建它, 否则自增2 可任意修改
00030:    if stroages.address == nil then
00031:        stroages.address = 0xC225361204C0
00032:    else
00033:        --增长规律, 不需要的, 连else一起 直接用 -- 屏蔽掉
00034:        stroages.address = stroages.address + 2    --增2
00035:        -- stroages.address = stroages.address - 2    --减2
00036:    end
00037:
    
```

用户可自行修改。

蓝牙名称起始及变化规律:

```

00038:    --蓝牙名称
00039:    if stroages.ble_name == nil then
00040:        stroages.ble_name = 0
00041:    else
00042:        --增长规律, 不需要的, 连else一起 直接用 -- 屏蔽掉
00043:        stroages.ble_name = stroages.ble_name + 2 --增2
00044:    end
00045:
    
```

蓝牙名称一部分信息

完整蓝牙名称:

```

-----
0085:
0086: --BLE用户自定义蓝牙名称
0087: local name = {
0088:     offset = 0x0040,    --data分区偏移
0089:     --data = "WS8100-BLE,    --固定蓝牙名称
0090:     data = "WS8100-BLE_".string.format("%x", stroages.ble_name)    --动态蓝牙名称 "WS8100-BLE_" 字符串可任意修改 .. 为字符串拼接符
0091: }
0092:
    
```

user_oem.lua 文件中有详细注释, 有需要请自行查阅修改。

3.5 单步调试

WS8100 支持 keil 环境下单步调试, 单步调试需要关闭看门狗及低功耗。

3.6 串口调试

3.6.1 调试接口

dbg_printf、dbg_block_printf 和 dbg_block_printf_hex

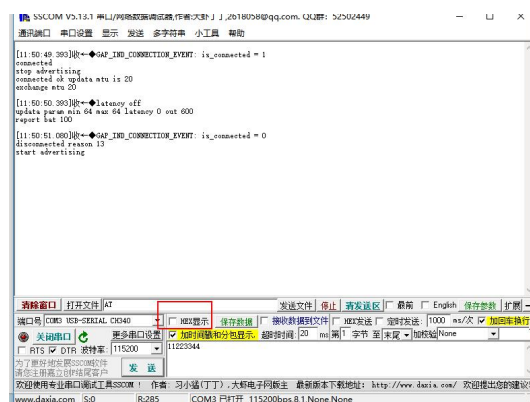
中断中, 推荐使用 dbg_block_printf

3.6.2 推荐调试方法:

因为烧录工具 BLE_DownloadTool.exe 仅仅为烧录所用, 对调试的支持不是很好, 因此建议用 BLE_DownloadTool.exe 和 sscom5.13.1.exe 组合使用。

具体操作流程为

- 1、在 BLE_DownloadTool.exe 烧录完成
- 2、点击 BLE_DownloadTool.exe 断开
- 3、关闭 BLE_DownloadTool.exe 串口
- 4、打开 sscom5.13.1.exe 串口



去掉 HEX 显示，显示为 ASCII

4、内存部分

4.1 WS8100 RAM 共 40K,其中掉电不丢失 RAM 24K ,掉电不保留数据 16K。

超低低功耗【DEEP_SLEEP 3uA】，关闭外设，RAM 掉电。

浅睡眠低功耗：cpu 休眠【NORMAL_SLEEP 900uA+】，外设不掉电，RAM 不掉电

掉电，针对的是 DEEP_SLEEP 的超低低功耗模式，及保留 RAM 模式的待机。

不保留 RAM 模式的待机，唤醒后从复位处开始运行。

4.2 栈顶：

程序栈顶，__initial_sp,必须位于掉电不丢失 RAM 24K 区间内。否则引起进入低功耗，唤醒后，出现堆栈破坏，引起程序复位。

程序编译完成，需要查看 map 文件

CODE > WS8100_TRANSPOND > example > ws8100_app_standard_at > project > list					搜索"list"
名称	修改日期	类型	大小		
Project.map	2020/10/13 星期...	MAP 文件	479 KB		
startup_cpu_deepsleep.lst	2020/10/13 星期...	LST 文件	41 KB		
04738: GFSM_Queue	0x20001e2c	Data	12	fsm.o(.bss)	
04739: glb_customize_para	0x20001e38	Data	60	rom_lib.o(.bss)	
04740: smp_root_	0x20004b64	Data	56	smp_fsm.o(.bss)	
04741: att_root_	0x20004b9c	Data	40	att_pdu.o(.bss)	
04742: smp_ecc_env	0x20004bdc	Data	788	smp_ecc.o(.bss)	
04743: __initial_sp	0x20005df0	Data	0	startup_cpu_deepsleep.o(STACK)	

确保 __initial_sp 不超过 0x20005FFF

4.3 重量级任内存

较大的运算的数据量处理，将内存分配到掉电不保留区域。

```
/*define to no reation ram*/
```

```
#define NO_REACTION_RAM(a) a __attribute__((section("no_reaction_ram")))
```

```
NO_REACTION_RAM(user_mic_env_t mic_env);
```

任务处理完成后，才能进入 超低低功耗，如，采集 10s 心率数据，计算心率，采集期间，设置低功耗阻止，处理完成，清除低功耗阻止标记。

4.4 具备内存回收能力的函数

4.4.1 释放服务

```
void gatt_free_service(struct gatt_service_stru *service);
```

4.4.2 接收数据回应

```
void gatt_server_pend_ind_response(struct att_pend_ind_stru *in);
```

4.4.3 发送蓝牙通知数据

```
void gatt_server_notify_indicate(struct gatt_server_notify_indicate_stru *in);
```

4.4.4 写蓝牙数据

```
void gatt_write_value(struct gatt_tl_connect_stru *tlin, struct att_handle_value_stru *in, UINT8 task);
```

4.4.5 恢复配对密钥

```
void smpc_peerkeys_recover(void *context, struct list_stru *list);
```

4.4.6 命令发送

```
void hci_command_send_amp(void *context, void *par, UINT32 opcode);
```

调用以上函数之前，参数实例必须用 NEW 申请。

4.5 内存申请，内存释放

内存申请： NEW

内存释放： FREE

5、任务部分

5.1 定时任务

创建 1000ms 后执行 app_task 任务的 软定时器任务 parm 作为 app_task 函数参数

```
ft = FsmAddTimerCx(1000,1,app_task,parm);
```

5.2 等待 cpu 空闲任务

创建 CPU 空闲后调度任务，DM_KEY_EV_EVENT_REPORT_IND rpt_ind 分别为参数

```
FsmEventCx(DM_KEY_EV_EVENT_REPORT_IND, (fsm_func *)key_event_fsm, rpt_ind);
```

传递参数中有指针，需要用全局指针实例 或者用 NEW 申请实例，并注意释放，防止内存泄漏。

ws8100 支持如下低功耗模式:

注意 : WS8100 非常开电源域外设:

/*****

返回值：

{

```
if(retention sram == SYSCFG noRetention Sram) // 低功耗不允许不配置不保留内存
```

{

return;

}

```
syscfg deepsleep set ram retention(retention sram);
```

```
syscfg deepsleep cmd wakeup source(SYSCFG_DeepSleepWakeUpSource_GPIO,DISABLE);//禁用 IO 唤醒源
```

```
cur_power_mode_type = type;
```

```
set low power mode(sys power mode != NONE SLEEP);
```

```
if (cur_power_mode_type == DEEP_SLEEP) //深度睡眠
```

```
        {  
            syscfg_deepsleep_set_mode(SYSCFG_DeepSleepMode_DEEPSLEEP);  
        }  
        else if (cur_power_mode_type == NORMAL_SLEEP) //浅度睡眠  
        {  
            syscfg_deepsleep_set_mode(SYSCFG_DeepSleepMode_SLEEP);  
        }  
    }  
}  
  
/*****
```

函 数 名 : syscfg_configuration

功能描述 : 系统配置

输入参数 : void

输出参数 : 无

返 回 值 :

```
*****/
```

```
void syscfg_configuration(void)  
{  
    // 【1】 配置晶振  
  
    #if (USE_EXT_XTAL_16M)  
        // 16M 晶振  
        syscfg_external_16M_xtal();  
        //< 配置晶振匹配电容,外部无须电容  
        syscfg_set_oscxtal_config(RF_ADJUST);  
    #endif  
  
    #if (USE_EXT_XTAL_32M)  
        // 32M 晶振  
        syscfg_external_32M_xtal();  
        //< 配置晶振匹配电容, 外部无须电容  
        syscfg_set_oscxtal_config(0x57);  
    #endif  
  
    #if (USE_EXT_XTAL_32K)
```

```
// 外部 32K 晶振

syscfg_external_32k_xtal();

radio_set_32k_freq(32768);

radio_set_32k_ppm(20);

#endif

#if (USE_INTER_RC_32K)

// 内部 32K 晶振

syscfg_internal_32k_rc();

radio_set_32k_ppm(300);

#endif

// 配置低功耗模式

low_power_sleep_config_ex(SYS_CFG_Retention_Sram_24k, SDK_ENABLE_SLEEP);

}
```

6.2 低功耗运行例程

```
/*
*****
函数名 : deep_sleep_not_available
功能描述 : 低功耗自适应调整函数
输入参数 : void
输出参数 : 无
*****
static uint8_t deep_sleep_not_available(void)
{
// 【1】 检测低功耗系统开关

if(get_low_power_mode() == 0){ // 不允许低功耗休眠

return 1;

}

// 【2】 系统只允许浅睡眠低功耗

if(sys_power_mode == NORMAL_SLEEP){

*((__IO uint32_t *) (0xE000ED10)) = 0x0; //浅睡眠

__wfi();

return 1;

}

}
```

```
// 系统允许深睡眠模式

// 【3】 任务处理完 切换到深睡眠模式

if((cur_power_mode_type == NORMAL_SLEEP) && (cpu_prevent_sleep_get() == 0)){
    low_power_type_config_ex(SYSCFG_Retention_Sram_24k,DEEP_SLEEP);
    *((__IO uint32_t *) (0xE000ED10)) = 0x4; //深睡眠
    return 1;
}

// 【4】 深睡眠模式 有事情要处理，需要切换到浅睡眠模式

if (cur_power_mode_type == DEEP_SLEEP && cpu_prevent_sleep_get()){
    low_power_type_config_ex(SYSCFG_Retention_Sram_24k,NORMAL_SLEEP);
    *((__IO uint32_t *) (0xE000ED10)) = 0x0; //浅睡眠
    return 1;
}

// 【5】 当前处于无低功耗状态

if(cur_power_mode_type == NONE_SLEEP){
    return 1;
}

// 【6】 当前处于浅睡眠低功耗状态

if(cur_power_mode_type == NORMAL_SLEEP){
    *((__IO uint32_t *) (0xE000ED10)) = 0x0; //浅睡眠
    __wfi();
    return 1;
}

// 【7】 SDK 底层校准中 不允许深度睡眠

if(get_calibrate_ing()){
    return 1;
}

// 【8】 允许 DEEP_SLEEP 的情况

return 0;
}
```

```
/******
```

```
函 数 名 : low_power_sleep
```

```
功能描述 : 低功耗处理函数
```

```
输入参数 : ptr_sleep lib_sleep
```

```
输出参数 : 无
```

```
*****/
```

```
void low_power_sleep( ptr_sleep lib_sleep )
```

```
{
```

```
// 【1】低功耗自适应函数
```

```
    if(deep_sleep_not_available())
```

```
        return;
```

```
// 【2】深度低功耗过程屏蔽外设中断，常开电源域仍然可以中断
```

```
    /* mask interrupt */
```

```
    __set_faultmask(1);
```

```
// 【3】检测系统任务，是否允许低功耗
```

```
    /* is into sleep */
```

```
    if(lib_sleep())
```

```
    {
```

```
// 【4】进入低功耗，非常开电源域外设会掉电，需要进行数据或者省电处理
```

```
//    外设寄存器少或关键外设部分，放入 retainram 区域中
```

```
        deepsleep_prepare();           /* prepare for into sleep. */
```

```
// 【5】ws8100 进入低功耗状态，高频部分停止工作，32k 监测唤醒事件
```

```
        deep_sleep_enter();           /* into sleep. */
```

```
// 【6】ws8100 检测到唤醒事件，高频部分开始工作，恢复外设，恢复运行
```

```
//    外设寄存器少或关键外设部分，从 retainram 区域中恢复
```

```
        deepsleep_exit();
```

```
/* exit sleep */
```

```
    }
```

```
// 【7】开启中断源
```

```
        /* cancel mask interrupt */
        __set_faultmask(0);
    }
    // 入口函数
    void main(void)
    {
        /* To Do */

        while(1)
        {
            lib_schedule();          /* run task schedule. */

            #if (WDT_ENABLE)

                feed_dog();          /* run feed dog */

            #endif

            low_power_sleep(lib_sleep);/* low power mode schedule */

        }
    }
```

6.3 DEEP_SLEEP 进入外设掉电前，关键外设备份，gpio 省电操作。

函 数 名 : deepsleep_prepare

功能描述 : DEEP_SLEEP 模式，非常开电源域及非 retain_ram 区域会掉电，需要对关键外设或者较少寄存器的外设，寄存器进行备份，关闭 flash 烧录功能

输入参数 : void

输出参数 : 无

返 回 值 :

*****/

```
void deepsleep_prepare(void)
{
    uint32_t i = 0;

    // 【1】 DEEP_SLEEP 模式 待机事件处理
    retain_ram_standbye_prepare();

    // 【2】 /*备份需要备份的寄存器到不掉电的 ram 区域*/
    for(i = 0; i < listnum;i++)
```

```
{  
    backup_table.backupmem[i] = *(volatile uint32_t *)needbackuplist[i];  
}  
  
// 【3】 /*进入休眠之前，GPIO 省电操作及应用层进行的一些准备工作*/  
user_deepsleep_prepare();  
  
// 【4】 /* 关闭 flash burst 功能 */  
flash_burst_disable();  
}
```

6.4 即将进入 DEEP_SLEEP，外设掉电前，gpio 省电操作。

```
void user_deepsleep_prepare(void)  
{  
    // 有其它配置，需要依此处理  
    #if is_uart_dev(DBG_UART)  
    #if USART1_FlowCtrl_EN  
        gpio_remap_config(GPIOD_Pin0,GPIO_Remap_GPIO);  
        gpio_remap_config(GPIOD_Pin1,GPIO_Remap_GPIO);  
    #endif  
  
    //  gpio_remap_config(GPIOD_Pin2,GPIO_Remap_GPIO);  
    //  gpio_remap_config(GPIOD_Pin3,GPIO_Remap_GPIO);  
    #endif  
}
```

6.5 退出 DEEP_SLEEP 恢复外设，恢复关键区域

```
/**/  
  
函 数 名 : deepsleep_exit  
功能描述 : 退出 DEEP_SLEEP 模式，外设重新恢复  
输入参数 : void  
输出参数 : 无  
  
/**/
```

```
void deepsleep_exit(void)
{
    uint32_t i = 0;

    // 【1】 /* 恢复 CG_CTRL 寄存器*/

    *(volatile uint32_t *)needbackuplist[i] = backup_table.backupmem[i];    i++;

#ifdef WDT_ENABLE
    wdt_init();
#endif

    // 【2】 /* 重新时能 flash burst */

    flash_burst_enable();

    // 【3】 /* 恢复用户外设，及其他事件处理*/

    user_deepsleep_exit();

    // 【4】 /* 恢复 patch*/

    patch_configuration();

    // 【5】 /*恢复备份的关键寄存器及寄存器较少的外设*/

    for(; i < listnum;i++)
    {
        *(volatile uint32_t *)needbackuplist[i] = backup_table.backupmem[i];
    }

    // 【1】 DEEP_SLEEP 模式 退出待机事件处理

    retain_ram_standby_exit();
}
```

6.6 DEEP_SLEEP 模式 外设正常工作

DEEP_SLEEP 模式，外设会掉电，因此外设需要正常工作，需要设置阻止进入 DEEP_SLEEP 模式。

```
typedef enum cpu_prevent_sleep
```

```
{
    /* Flag indicating that the amic is ongoing */

    AMIC_ONGOING                = 0x00000001,                // ADC 采集

    UART0_RX_ONGOING            = 0x00000002,

    UART0_TX_ONGOING            = 0x00000004,
```

```

    UART1_RX_ONGOING          = 0x00000008,
    UART1_TX_ONGOING          = 0x00000010,
    SPI0_RX_ONGOING           = 0x00000020,
    SPI0_TX_ONGOING           = 0x00000040,
    SPI1_RX_ONGOING           = 0x00000080,
    SPI1_TX_ONGOING           = 0x00000100,
    IIC_RX_ONGOING             = 0x00000200,
    IIC_TX_ONGOING             = 0x00000400,
    QDEC_ONGOING               = 0x00000800, //正交解码器
    DMA1_RX_ONGOING            = 0x00001000,
    DMA1_TX_ONGOING            = 0x00002000,
    DMA2_RX_ONGOING            = 0x00004000,
    DMA2_TX_ONGOING            = 0x00008000,
    DMA3_RX_ONGOING            = 0x00010000,
    DMA3_TX_ONGOING            = 0x00020000,
    DMA4_RX_ONGOING            = 0x00040000,
    DMA4_TX_ONGOING            = 0x00080000,
    TIMER0_ONGOING             = 0x00100000,
    TIMER1_ONGOING             = 0x00200000,
    TIMER2_ONGOING             = 0x00400000,
    TIMER3_ONGOING             = 0x00800000,
    GPADC_ONGOING              = 0x01000000,
    // ... 用户自行添加事件          ...
    ADV_ONGOING                = 0x20000000,
    USER_TSK_ONGOING           = 0x40000000,
}cpu_prevent_sleep_t;

```

6.6.1 设置外设工作事件

```

/*
 * 简介:设置外设阻止休眠的对应 bit 位。
 */
void cpu_prevent_sleep_set(cpu_prevent_sleep_t prv_slp_bit)
{
    cpu_prevent_sleep |= prv_slp_bit;
}

```

```
}
```

6.6.2 清除外设工作事件

```
/*
```

```
* 简介:清除阻止休眠的 bit 位。
```

```
*/
```

```
void cpu_prevent_sleep_clear(cpu_prevent_sleep_t prv_slp_bit)
```

```
{
```

```
    cpu_prevent_sleep &= ~prv_slp_bit;
```

```
}
```

6.6.3 示例代码

```
void adc_process(void)
```

```
{
```

```
    #if SDK_GADC_ENABLE
```

```
        int i;
```

```
        uint16_t val;
```

```
// 【1】停止温度校准，内部温度校准复用 ADC
```

```
        hwadjust_get_temperature_adc_data_period_stop();
```

```
// 【2】设置 adc 采集标志，停止进入 DEEP_SLEEP 模式
```

```
        cpu_prevent_sleep_set(AMIC_ONGOING);
```

```
// 【3】配置 ADC
```

```
        gpio_dac_configuration(GPIOA_Pin4);
```

```
// 【4】等待转换完成，读出来数值为 0 表示，转换未完成，需要加延时
```

```
        for(i=0;i<500;i++);
```

```
        val = GPADC->GPADC_DATA;
```

```
// 【5】清除 adc 工作标志，允许进入 DEEP_SLEEP 模式
```

```
        cpu_prevent_sleep_clear(AMIC_ONGOING);
```

```
// 【6】开启片内温度校准
```

```
        hwadjust_get_temperature_adc_data_period_start(NULL,NULL);
```

```
        dbg_printf("adc %d\n",val);
```

```
#endif
```

```
}
```

6.7 退出 DEEP_SLEEP 恢复外设，恢复用户外设，及应用数据

/******

函 数 名 : user_deepsleep_exit

功能描述 : 退出 DEEP_SLEEP 用户外设恢复, 及用户数据恢复

输入参数 : void

输出参数 : 无

*****/

void user_deepsleep_exit(void)

{

#if (is_uart_dev(APP_UART))

 app_uart_configuration(ON);

#endif

#if (is_uart_dev(DBG_UART) && (APP_UART != DBG_UART))

 dbg_uart_configuration(ON);

#endif

}

6.8 DEEP_SLEEP 模式 用户中断

进入 DEEP_SLEEP 状态后, 蓝牙事件, 常开电源域外设中断事件, 32K 校准事件, 温度校准事件, 软定时器事件, 可打断 DEEP_SLEEP 模式。

6.8.1 蓝牙中断事件

蓝牙中断事件包含

- 1、广播间隔 时间到了起来广播
- 2、interval 及 latency 时间到了自动起来响应协议
- 3、蓝牙超时断开。

6.8.2 常开电源域外设中断事件

程序在 DEEP_SLEEP 模式, 仅有常开电源域外设能够产生中断, 其余外设因为掉电, 不在工作状态, 所以不会产生中断。例程详见第五章中断部分。

3.8.3 常开电源域 GPIO 电路设计注意

GPIO 注册了低功耗中断, GPIO 为低电平, 可正常进入低功耗, GPIO 为高电平, 进入了低功耗, 会立刻退出低功耗, 因此注意空闲状态, 将 GPIO 电路部分设置为低电平, 以适应低功耗。

6.8.4 固定打断事件

WS8100 具备较强的环境适应能力，内部具备温度校准，32K 时钟校准功能，该功能为不可屏蔽功能。

6.9 DEEP_SLEEP 模式 漏电流调试

6.9.1 漏电流查找

DEEP_SLEEP 模式，单个 IO 漏电流 70uA 左右，内置 70K 下拉电阻，无上拉。

功耗指标参考参考 DEEP_SLEEP 待机模式 开启调试串口，功耗 3uA，以此作为参考依据，判断硬件设计是否有漏电流。

6.9.2 漏电流处理方法

漏电流情况	处理方法	举例
串口等外设 IO 漏电	进入 DEEP_SLEEP 模式，外设失电，外设 IO 退化成普通 GPIO，输入端，可能处于悬浮状态，悬浮状态不接负载，会造成漏电流，可焊接 3M 上拉电阻处理。	如 UART RX 如果不接负载 TX，进入 DEEP_SLEEP 会造成 70uA 漏电流，需要接 3M 上拉电阻解决。
GPIO 默认高电平	1、检测下拉电阻大小，是否符合理论需求，或者移除下拉电阻。 2、检测 GPIO 是否接了存在电压差的负载。 3、进入低功耗等状态，将输出高电平的外设配置为下拉输入状态。	如部分设计电路在复位电路上加下拉电阻，会造成漏电流。解决方法，移除下拉电阻，或者将下拉电阻增加到 M 级或者符合设计需求的值。
GPIO 默认低电平	检测 GPIO 如外部发生灌电流，将 GPIO 设置为悬浮输入。	

6.10 浅睡眠低功耗，CPU 运行频率仍然不够

// 【1】关闭低功耗

```
low_power_sleep_config_ex(SYSCFG_Retention_Sram_24k,NONE_SLEEP);
```

// 【2】 to do user Task

// 【3】 处理完了，恢复低功耗

```
low_power_sleep_config_ex(SYSCFG_Retention_Sram_24k,SDK_ENABLE_SLEEP);
```

7、待机

待机模式	模式说明	功耗
DEEP_SLEEP	1、停止用户任务 2、关闭所有外设 3、关闭蓝牙	备注：该模式程序以极低功耗模式运行。 1.3 uA 3 uA 【开启调试串口】

DEEP_SLEEP+	进入 DEEP_SLEEP+ 模式后，外设，蓝牙 ram 断电 注册 gpio 可唤醒	0.9uA
-------------	--	-------

7.1 DEEP_SLEEP+ 模式待机

DEEP_SLEEP+模式为不保留 RAM 的待机方式，功耗最低可达 0.9uA，芯片级 0.6uA。因此设计时候注意 GPIO 漏电流

标准 SDK, PA0 拉高进入待机，PB4 唤醒。

注意事项：

注册待机唤醒 IO 用

```
void setup_gpio_wake_irq(irq_bank start,uint32_t pins_map,gpio_irq_type type)
```

注册普通中断 IO 用【不可唤醒待机】

```
void setup_gpio_irq(irq_bank start,uint32_t pins_map,gpio_irq_type type)
```

进入待机前调用【禁用非唤醒 IO 中断】

```
void cmd_goto_standbye_prepare(void)
```

7.2 DEEP_SLEEP 模式待机

7.2.1 进入 DEEP_SLEEP 模式待机示例代码

函 数 名 : set_retain_ram_to_standbye

功能描述 : 设置进入保留 RAM 方式待机

输入参数 : void

输出参数 : 无

*****/

```
void set_retain_ram_to_standbye(void)
```

```
{
```

```
    goto_stand_bye = 1;
```

```
}
```

函 数 名 : retain_ram_enter_standbye_process

功能描述 : 进入保留 RAM 待机方式，用户事件处理

输入参数 : void

输出参数 : 无

返 回 值 :

```

*****/

void retain_ram_enter_standbye_process(void)                                //停掉所有的任务
{
//【1】 断开蓝牙连接

    if(app_env.state == BT_STATE_CONNECTED)
    {
        app_ble_disconn_req();
    }

//【2】 处理用户 GPIO

/*Todo Stop User Gpio*/

//【3】 移除定时器

    timer_del_byp_free(&ft);

//【4】 设置蓝牙断开事件中，不重启广播

    app_env.state = BT_STATE_STOP;
}

/*****
```

函 数 名 : retain_ram_standbye_prepare

功能描述 : 执行进入保留 RAM 方式待机

输入参数 : void

输出参数 : 无

返 回 值 :

```

*****/

void retain_ram_standbye_prepare(void)
{
    if(goto_stand_bye == 1){
        if(in_stand_bye != goto_stand_bye){
            in_stand_bye = goto_stand_bye;
            retain_ram_enter_standbye_process();
        }
    }
}

/*****
```

函 数 名 : deepsleep_prepare

功能描述 : DEEP_SLEEP 模式, 非常开电源域及非 retain_ram 区域会掉电, 需要对关
键外设或者较少寄存器的外设, 寄存器进行备份, 关闭 flash 烧录功能

输入参数 : void

输出参数 : 无

返 回 值 :

*****/

void deepsleep_prepare(void)

```
{
    uint32_t i = 0;

    // 【1】 DEEP_SLEEP 模式 进入待机事件处理
    retain_ram_standbye_prepare();

    // 【2】 /*备份需要备份的寄存器到不掉电的 ram 区域*/
    for(i = 0; i < listnum; i++)
    {
        backup_table.backupmem[i] = *(volatile uint32_t *)needbackuplist[i];
    }

    // 【3】 /*进入休眠之前, GPIO 省电操作及应用层进行的一些准备工作*/
    user_deepsleep_prepare();

    // 【4】 /* 关闭 flash burst 功能 */
    flash_burst_disable();
}
```

7.2.2 退出 DEEP_SLEEP 模式待机示例代码

/*****/

函 数 名 : clear_retain_ram_to_standbye

功能描述 : 退出保留 RAM 方式待机

输入参数 : void

输出参数 : 无

返 回 值 :

调用函数 :

*****/

```
void clear_retain_ram_to_standbye(void)
```

```
{
```

```
    goto_stand_bye = 0;
```

```
}
```

```
/******
```

函 数 名 : retain_ram_exit_standbye_process

功能描述 : 退出保留 RAM 待机方式, 用户事件处理

输入参数 : void

输出参数 : 无

```
*****/
```

```
void retain_ram_exit_standbye_process(void)
```

```
//恢复所有的任务
```

```
{
```

```
//【1】设置断开连接重启广播标志
```

```
    app_env.state
```

```
= BT_STATE_ADV_ON;
```

```
//【2】设置断开连接重启广播标志
```

```
    app_bt_enable(TRUE);
```

```
//【3】处理用户 GPIO
```

```
/*Todo Configure User Gpio*/
```

```
//【4】创建用户任务
```

```
    app_task_looper(0,NULL);
```

```
}
```

```
/******
```

函 数 名 : retain_ram_standbye_exit

功能描述 : 执行退出保留 RAM 方式待机

输入参数 : void

输出参数 : 无

返 回 值 :

```
*****/
```

```
void retain_ram_standbye_exit(void)
```

```
{
```

```
    if(goto_stand_bye == 0){
```

```
        if(in_stand_bye != goto_stand_bye){
```

```
        in_stand_bye = goto_stand_bye;

        retain_ram_exit_standbye_process();

    }

}

}

/*****

函 数 名   : deepsleep_exit

功能描述   : 退出 DEEP_SLEEP 模式，外设重新恢复

输入参数   : void

输出参数   : 无

*****/

void deepsleep_exit(void)
{
    uint32_t i = 0;

    // 【1】 /* 恢复 CG_CTRL 寄存器*/

    *(volatile uint32_t *)needbackuplist[i] = backup_table.backupmem[i];    i++;

#ifdef WDT_ENABLE
    wdt_init();
#endif

    // 【2】 /* 重新时能 flash burst */

    flash_burst_enable();

    // 【3】 /* 恢复用户外设，及其他事件处理*/

    user_deepsleep_exit();

    // 【4】 /* 恢复 patch*/

    patch_configuration();

    // 【5】 /*恢复备份的关键寄存器及寄存器较少的外设*/

    for(; i < listnum;i++)
    {
        *(volatile uint32_t *)needbackuplist[i] = backup_table.backupmem[i];
    }

    // 【6】 DEEP_SLEEP 模式 退出待机事件处理

    retain_ram_standbye_exit();

}
```

8、中断

8.1 WS8100 中断命名方法

XXX_IRQHandler, 通过 IRQHandler 关键字查找自己想要的中断，中断向量定义位于 startup_cpu_deepsleep.s

对于没有的中断，用户可自行补充定义。

8.2 WS8100 中断管理

WS8100 中断管理比较简单

typedef struct

```
{  
    uint32_t nvic_irq_channel;           //中断通道  
    uint32_t nvic_irq_channel_preemption_priority; //中断优先级  
    uint32_t nvic_irq_channel_subpriority; //中断次级优先级  
    functional_state_t nvic_irq_channel_cmd; //中断使能  
} nvic_init_type_def;
```

8.3 中断应用示例代码

void nvic_configuration(void)

```
{  
    nvic_init_type_def nvic_init_structure;  
    // 【1】配置中断优先级格式，初级优先级 0~3 次级优先级 0~1 值越小优先级越高  
    NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);  
    // 【2】初级中断优先级 1 (0~3)  
    nvic_init_structure.nvic_irq_channel_preemption_priority = 1;  
    // 【3】初级中断优先级 0 (0~1)  
    nvic_init_structure.nvic_irq_channel_subpriority = 0;  
    // 【4】使能中断配置 (DISABLE~ENABLE)  
    nvic_init_structure.nvic_irq_channel_cmd = ENABLE;  
  
    // 【5】配置中断通道 (DISABLE~ENABLE)  
    nvic_init_structure.nvic_irq_channel = ROM_LIB_0_IRQN;  
    // 【5】中断配置生效 (DISABLE~ENABLE)  
    nvic_init(&nvic_init_structure);
```

// 【6】 ROM_LIB_XX 中断为系统中断，不要修改

```
NVIC_ClearPendingIRQ(ROM_LIB_1_IRQN);

nvic_init_structure.nvic_irq_channel = ROM_LIB_1_IRQN;
nvic_init(&nvic_init_structure);

nvic_init_structure.nvic_irq_channel = ROM_LIB_2_IRQN;
nvic_init(&nvic_init_structure);

nvic_init_structure.nvic_irq_channel = ROM_LIB_3_IRQN;
nvic_init(&nvic_init_structure);

nvic_init_structure.nvic_irq_channel = ROM_LIB_4_IRQN;
nvic_init(&nvic_init_structure);

nvic_init_structure.nvic_irq_channel = ROM_LIB_5_IRQN;
nvic_init(&nvic_init_structure);

nvic_init_structure.nvic_irq_channel = ROM_LIB_6_IRQN;
nvic_init(&nvic_init_structure);

nvic_init_structure.nvic_irq_channel = ROM_LIB_7_IRQN;
nvic_init(&nvic_init_structure);

nvic_init_structure.nvic_irq_channel = ROM_LIB_13_IRQN;
nvic_init(&nvic_init_structure);
```

// 【7】 串口中断

```
#if (DBG_UART == ws_uart1 || APP_UART == ws_uart1)

    nvic_init_structure.nvic_irq_channel = UART1_IRQN;
    nvic_init_structure.nvic_irq_channel_preemption_priority = 2;
    nvic_init_structure.nvic_irq_channel_subpriority = 0;
    nvic_init(&nvic_init_structure);

#endif

#if (DBG_UART == ws_uart0 || APP_UART == ws_uart0)

    nvic_init_structure.nvic_irq_channel = UART0_IRQN;
    nvic_init_structure.nvic_irq_channel_preemption_priority = 2;
    nvic_init_structure.nvic_irq_channel_subpriority = 0;
    nvic_init(&nvic_init_structure);

#endif
```

```
//【8】定时器中断

#if TIMER01_IRQ_ENABLE

    nvic_init_structure.nvic_irq_channel = Timer0_IRQn;

    nvic_init(&nvic_init_structure);

#endif

#if EXT_GPIO_IRQ

    nvic_init_structure.nvic_irq_channel = GPIO_INT0_IRQn;

    nvic_init(&nvic_init_structure);

    nvic_init_structure.nvic_irq_channel = GPIO_INT1_IRQn;

    nvic_init(&nvic_init_structure);

    nvic_init_structure.nvic_irq_channel = GPIO_INT2_IRQn;

    nvic_init(&nvic_init_structure);

    nvic_init_structure.nvic_irq_channel = GPIO_INT3_IRQn;

    nvic_init(&nvic_init_structure);

#endif

}
```

8.4 WS8100 清中断

清中断

```
__STATIC_INLINE void NVIC_ClearPendingIRQ(IRQn_Type IRQn)
{
    NVIC->ICPR[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F)); /* Clear pending interrupt */
}
```

9、SYS_STICK 及 RTC

9.1 SYS_STICK 示例代码

```
sys_stick_handler stick_handler = NULL;

/*****

函 数 名   : SysTick_Handler

功能描述   : sys_stick 中断函数

输入参数   : void

输出参数   : 无

*****/
```

```
void SysTick_Handler(void)
```

```
{  
    NVIC_ClearPendingIRQ(SysTick_IRQn);  
    if(stick_handler)  
    {  
        stick_handler();  
    }  
}
```

```
/******
```

函 数 名 : start_system_stick

功能描述 : 启动 sysstick

输入参数 : uint32_t ms

输出参数 : 无

返 回 值 :

```
*****/
```

```
void start_system_stick(uint32_t ms)
```

```
{  
    SysTick_Config(1600*ms);  
    NVIC_EnableIRQ(SysTick_IRQn);  
    cpu_prevent_sleep_set(SYS_STICK_ONGOING);  
}
```

```
/******
```

函 数 名 : stop_system_stick

功能描述 : 停止 sysstick

输入参数 : void

输出参数 : 无

返 回 值 :

```
*****/
```

```
void stop_system_stick(void)
```

```
{  
    NVIC_DisableIRQ(SysTick_IRQn);
```

```
    cpu_prevent_sleep_clear(SYS_STICK_ONGOING);  
}
```

```
/******
```

函 数 名 : register_sys_stick_handler

功能描述 : 注册 sysstick 回调函数

输入参数 : sys_stick_handler handler

输出参数 : 无

```
*****/
```

```
void register_sys_stick_handler(sys_stick_handler handler)
```

```
{  
    stick_handler = handler;  
}
```

```
/******
```

函 数 名 : sys_stick_handler

功能描述 : sysstick 中断处理函数

输入参数 : void

输出参数 : 无

返 回 值 :

```
*****/
```

```
void sys_stick_handler(void)
```

```
{  
    dbg_printf("*");  
}
```

```
/******
```

函 数 名 : user_stick_example

功能描述 : sysstick 示例代码

输入参数 : void

输出参数 : 无

返 回 值 :

*****/

```
void user_stick_example(void)
{
    dbg_printf("stick example ... \n");

    // 【0】初始化 sysstick 10ms                并使能
    start_system_stick(10);

    // 【1】注册回调函数
    register_sys_stick_handler(sys_stick_handler);
}
```

9.2 RTC 示例代码

RTC 为常开电源域，低功耗情况下，可正常使用。

/*****

函 数 名 : rtc_event_cbk

功能描述 : 用户 rtc 中断代码，常开电源域，可打断低功耗唤醒

输入参数 : void

输出参数 : 无

返 回 值 :

*****/

```
void rtc_event_cbk(void)
{
    static int gpio_out=0;

    const int gpio_map = gpio_to_value(GPIOA_Pin3);

    gpio_out ^= gpio_map;

    user_gpio_dir_out(gpio_map, gpio_out);
}
```

/*****

函 数 名 : user_rtc_example

功能描述 : 用户 rtc 示例代码

输入参数 : void

输出参数 : 无

返 回 值 :

注 意：检查是否 需要定义 #define RTC_SURPORT_ENABLE 1

```

*****/

void user_rtc_example(void)
{
    nvic_init_type_def nvic_init_structure;

// 【1】开启 rtc 时钟

    clock_config(ws_rtc,ENABLE);

// 【2】开启 rtc 中断

    NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);

    nvic_init_structure.nvic_irq_channel_preemption_priority= 1;

    nvic_init_structure.nvic_irq_channel_subpriority= 0;

    nvic_init_structure.nvic_irq_channel_cmd = ENABLE;

    nvic_init_structure.nvic_irq_channel = RTC_IRQn;

    nvic_init(&nvic_init_structure);

// 【3】配置 rtc 秒中断

    set_rtc_config(glb_customize_para.sleepclk_center_freq, rtc_event_cbk);
}
    
```

10、GPIO

10.1 GPIO 简介

类型	最大电流	备注
拉电流	15mA	
灌电流	11mA	
悬浮不接负载漏电流	50~70uA	接负载确定电平无漏电流

WS8100 GPIO 可配置到任意外设 IO 上，灵活映射。

10.2 GPIO 映射方法

```

void gpio_remap_config(gpio_pin gpio_pinx, gpio_remap_type gpio_remap)
{
    gpio_type_def *gpiox      = gpio_bank_reg[gpio_pinx/8];

    unsigned char  gpio_pin_x  = gpio_pinx%8;
    
```

```
// 每个 IO 占用 8 位，低 6 位为配置，高 2 位位状态读取

// 8 个一组， 占用 64bit 0-3  占用 0-31 bit 4-7  占用 32-63 bit

#define IO_BIT_WIDE                8

#define BIT_NIO                    4

if(gpio_pinx > GPIO_MAXS )

    return;

uint32_t pos                      = gpio_pin_x/BIT_NIO;

uint32_t mask                    = 0x3f                << ((gpio_pin_x%BIT_NIO)*IO_BIT_WIDE);

uint32_t val                     = gpio_remap<< ((gpio_pin_x%BIT_NIO)*IO_BIT_WIDE);

gpiox->map_ctrl->map_mode[pos] &= ~mask;

gpiox->map_ctrl->map_mode[pos] |= val;

}
```

10.3 WS8100 GPIO 的两种表示方法

10.3.1 独立 GPIO 示意

GPIOA_Pin1~GPIOA_Pin7

GPIOB_Pin0~GPIOB_Pin7

GPIOC_Pin0~GPIOC_Pin7

GPIOD_Pin0~GPIOD_Pin7

10.3.2 集合示意

uint32_t pins_map

PA0~PA7 对应到 bit00~bit07

PB0~PB7 对应到 bit08~bit15

PC0~PC7 对应到 bit16~bit23

PD0~PD7 对应到 bit24~bit31

10.4 WS8100 GPIO 输出的两种方式

10.4.1 低功耗模式下的输出写法

/**

函 数 名 : user_gpio_out

功能描述 : 低功耗方式下的输出配置，低电平为输入下拉配置，不具备驱动能力

输入参数 : gpio_pin GPIO_Pin

functional_state_t enable

输出参数 : 无

返回值 :

*****/

```
void user_gpio_out(gpio_pin GPIO_Pin, functional_state_t enable)
{
    if(enable){
        gpio_remap_config(GPIO_Pin,GPIO_Remap_GPIO);
        gpio_init(GPIO_Pin,GPIO_Mode_Output);
        gpio_set(GPIO_Pin);
    }else{
        gpio_remap_config(GPIO_Pin,GPIO_Remap_GPIO);
        gpio_init(GPIO_Pin,GPIO_Mode_Input_Pull_Down_Resistor);
    }
}
```

10.4.2 常规模式的写法

*****/

函数名 : gpio_set

功能描述 : 设置高电平输出

输入参数 : gpio_pin gpio_pinx

输出参数 : 无

*****/

```
void gpio_set(gpio_pin gpio_pinx)
{
    if(gpio_pinx > GPIO_MAXS )
        return;

    gpio_type_def      *gpiox      = gpio_bank_reg[gpio_pinx/8];
    unsigned char      GPIO_Pin    = gpio_pinx%8;
    gpiox->inout_ctrl->BSR        |= (1 << GPIO_Pin);
}
```

*****/

函数名 : gpio_reset

功能描述 : 设置低电平输出

输入参数 : gpio_pin gpio_pinx

输出参数 : 无

*****/

```
void gpio_reset(gpio_pin gpio_pinx)
{
    if(gpio_pinx > GPIO_MAXS )
        return;

    gpio_type_def      *gpiox      = gpio_bank_reg[gpio_pinx/8];
    unsigned char      GPIO_Pinx    = gpio_pinx%8;
    gpiox->inout_ctrl->BSR          |= ((1 << GPIO_Pinx) << 8);
}
```

10.5 GPIO 中断

WS8100 GPIO 中断共 4 组，每组中断可映射任意 GPIO。

GPIO 中断方式可分为上升沿触发，下降沿触发，双边触发。

滤波方式分为，即时【无滤波】、32 个时钟滤波

DEEP_SLEEP 低功耗模式下，需要按照低功耗的中断注册的方式注册，高电平进入了低功耗会立刻退出。低功耗状态，电平变高，触发中断。

10.5.1 中断服务函数

```
void GPIO_INT0_IRQHandler()
{
    gpio_clear_it_pending(GPIO_Int0);
    if(gpio_env.gpio_handler != NULL)
        gpio_env.gpio_handler(GPIO_Int0);
}
```

```
void GPIO_INT1_IRQHandler()
{
    gpio_clear_it_pending(GPIO_Int1);
    if(gpio_env.gpio_handler != NULL)
        gpio_env.gpio_handler(GPIO_Int1);
}
```

```
void GPIO_INT2_IRQHandler()
{
    gpio_clear_it_pending(GPIO_Int2);
    if(gpio_env.gpio_handler != NULL)
        gpio_env.gpio_handler(GPIO_Int2);
}

void GPIO_INT3_IRQHandler()
{
    gpio_clear_it_pending(GPIO_Int3);
    if(gpio_env.gpio_handler != NULL)
        gpio_env.gpio_handler(GPIO_Int3);
}
```

10.5.2 低功耗中断注册

/******

函 数 名 : setup_gpio_irq

功能描述 : 低功耗模式下的中断配置

输入参数 : irq_bank start 可选参数:

GPIO_Int0

GPIO_Int1

GPIO_Int2

GPIO_Int3

uint32_t pins_map 可选参数:

gpio_to_value(GPIOA_Pin1)

gpio_to_value(GPIOA_Pin2)

...

gpio_to_value(GPIOD_Pin7)

gpio_irq_type type 可选参数:

GPIO_Int_Type_Disable,

GPIO_Int_Type_RisingEdge,

GPIO_Int_Type_FallingEdge,

GPIO_Int_Type_BothEdge

输出参数 : 无

返回值 :

```
*****/

void setup_gpio_irq(irq_bank start,uint32_t pins_map,gpio_irq_type type)
{
    int i=0,k=start;
    for(i=0; i<32; i++)
    {
        if((pins_map & ((0x00000001)<< i)) && (k<4))
        {
            syscfg_deepsleep_set_gpiopin_wakeup_src(SYSCFG_GPIOWakeUpSource_GPIO_
INT0+k,SYSCFG_GPIOWakeUpType_One32KCycle_AntiShake,i);

            gpio_set_irq_src_type((irq_bank)(GPIO_Int0+k),(gpio_pin)i,type);

            k++;
        }
    }
}
```

10.5.3 普通 gpio 中断注册, 低功耗情况无法唤醒

函数名 : gpio_set_irq_src_type

功能描述 : 注册普通中断, 浅睡眠模式, 及正常模式可用

输入参数 : irq_bank GPIO_Intx

gpio_pin GPIOx_Pinx

gpio_irq_type GPIO_Int_Type

输出参数 : 无

```
*****/

void gpio_set_irq_src_type(irq_bank GPIO_Intx,gpio_pin GPIOx_Pinx,gpio_irq_type
GPIO_Int_Type)
{
    uint32_t idx = GPIO_Intx - GPIO_Int0;
    uint32_t mask = (0x03U)<<(idx*8);
    GPIO_MODULE->INTP_TYPE &= ~mask;
    GPIO_MODULE->INTP_TYPE |= (GPIO_Int_Type<<(idx*8));
}
```

```
mask = (0x1fU)<<(idx*8);

GPIO_MODULE->INTP_SRC  &= ~mask;

GPIO_MODULE->INTP_SRC  |= (GPIOx_Pinx<<(idx*8));

}
```

10.5.4 GPIO 中断应用示例代码

```
//=====

//****                      GPIO 应用举例

//****  中断中，用阻塞打印

//****

//=====

void example_user_gpio_isr(uint8_t intter_num)
{
    switch(intter_num){
        case GPIO_Int0:
            dbg_block_printf("int 0\n");
            break;
        case GPIO_Int1:
            dbg_block_printf("int 1\n");
            break;
        case GPIO_Int2:
            dbg_block_printf("int 2\n");
            break;
        case GPIO_Int3:
            dbg_block_printf("int 3\n");
            break;
    }
}

void user_gpio_example(void)
{
    uint32_t i;

    //【1】 开启 GPIO 时钟

    clock_config(ws_gpio,ENABLE);
```

// 【2】 开启 GPIO 中断

```
nvic_init_type_def nvic_init_structure;  
  
NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);  
  
nvic_init_structure.nvic_irq_channel_preemption_priority = 1;  
  
nvic_init_structure.nvic_irq_channel_subpriority          = 0;  
  
nvic_init_structure.nvic_irq_channel_cmd                 = ENABLE;  
  
nvic_init_structure.nvic_irq_channel                     = GPIO_INT0_IRQn;  
  
nvic_init(&nvic_init_structure);  
  
nvic_init_structure.nvic_irq_channel                     = GPIO_INT1_IRQn;  
  
nvic_init(&nvic_init_structure);  
  
nvic_init_structure.nvic_irq_channel                     = GPIO_INT2_IRQn;  
  
nvic_init(&nvic_init_structure);  
  
nvic_init_structure.nvic_irq_channel                     = GPIO_INT3_IRQn;  
  
nvic_init(&nvic_init_structure);
```

// 【3】 映射 pin 为 GPIO

```
gpio_remap_config(GPIOA_Pin1,GPIO_Remap_GPIO);  
  
gpio_remap_config(GPIOA_Pin2,GPIO_Remap_GPIO);  
  
gpio_remap_config(GPIOA_Pin3,GPIO_Remap_GPIO);  
  
gpio_remap_config(GPIOA_Pin4,GPIO_Remap_GPIO);
```

// 【4】 注册回掉函数

```
user_gpio_cfg_parameter_t p_parameter;  
  
p_parameter.gpio_handler = example_user_gpio_isr;  
  
user_gpio_config(&p_parameter);
```

// 【5】 配置 GPIO 下拉输入内置 70K 下拉，目前输入模式支持 下拉 悬浮两种

```
gpio_init(GPIOA_Pin1,GPIO_Mode_Input_Pull_Down_Resistor);  
  
gpio_init(GPIOA_Pin2,GPIO_Mode_Input_Pull_Down_Resistor);
```

// 【6】 注册低功耗唤醒中断高电平无法进入休眠，进入待机，必须保证配置了唤醒中断的 GPIO 处于低电平

```
setup_gpio_irq(GPIO_Int0,gpio_to_value(GPIOA_Pin1),GPIO_Int_Type_BothEdge);    // PA1  
setup_gpio_irq(GPIO_Int1,gpio_to_value(GPIOA_Pin2),GPIO_Int_Type_BothEdge);    // PA2
```

// 【7】 注册普通中断 普通中断 在非低功耗状态下才能响应，

// 测试普通 PA3 PA4 两个中断，需要关闭低功耗，或者按下 PA1 不动测试 PA3 PA4

```

    gpio_set_irq_src_type(GPIO_Int2,GPIOA_Pin3,GPIO_Int_Type_BothEdge);
    gpio_set_irq_src_type(GPIO_Int3,GPIOA_Pin4,GPIO_Int_Type_BothEdge);
}

```

11、UART 应用

WS8100 支持 2 个等同性能的 uart。

WS8100 UART 为非常开电源域设备, 因此 DEEP_SLEEP 模式, 串口无法接收数据, 需要外部 GPIO 通知, 有数据来临。接收完毕, 撤掉 GPIO 通知。

WS8100 UART 支持流控，管脚可任意映射。

11.1 设置 UART 工作状态

设置发送状态，SDK 已有处理，用户不用处理

```
cpu prevent sleep set(UART0 TX ONGOING);
```

清除发送状态，SDK 已有处理，用户不用处理

```
cpu_prevent_sleep_clear(UART0_TX_ONGOING);
```

/*****

函数名 : set_uart_recv_busy

功能描述：设置 UART 接收工作状态

输入参数 : ws dev dev

uint8_t busy

输出参数：无

返回值：

```
void set_uart_recv_busy(ws_dev dev,uint8_t busy)
```

{

```
if(busy){
```

```
if(dev == ws_uart0){
```

```
cpu_prevent_sleep set(UART0 RX ONGOING);
```

}

```
if(dev == ws_uart1){
```

```
cpu_prevent_sleep_set(UART1_RX_ONGOING);
```

```

    }
}
else{
    if(dev == ws_uart0){
        cpu_prevent_sleep_clear(UART0_RX_ONGOING);
    }
    if(dev == ws_uart1){
        cpu_prevent_sleep_clear(UART1_RX_ONGOING);
    }
}
}

```

11.2 APP USART IRQ 示例代码

```
//=====
//****                               串口应用举例
//****
//****
//=====

void user_uart_rcv_cbk(void *param,uint8_t ch)
{

}

/*****

函 数 名   : user_timer_example

功能描述   : 定时器应用举例

输入参数   : void

*****/

void user_uart_example(void)
{

    dbg_block_printf("app service initial timer ready\r\n");

    user_uart_cfg_parameter_t uart_param;

//【1】 开启串口 0 时钟
```

```
        clock_config(ws_uart1,ENABLE);

// 【2】 开启串口 0 中断

        nvic_init_type_def nvic_init_structure;

        NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);

        nvic_init_structure.nvic_irq_channel_preemption_priority = 1;

        nvic_init_structure.nvic_irq_channel_subpriority           = 0;

        nvic_init_structure.nvic_irq_channel_cmd                  = ENABLE;

        nvic_init_structure.nvic_irq_channel

        nvic_init(&nvic_init_structure);


// 【3】 初始化开串口参数以及注册回调函数

        uart_config Config={

            .baud_rate = 115200,

            .length_bit = UART_WordLength_8b,

            .stop_bit   = UART_StopBits_1,

            .parity_bit = UART_Parity_No,

            .mode        = APP_USART_IRQ,

#ifdef USART0_FlowCtrl_EN

            .rtsnPin     = GPIOD_Pin0, //串口接收流控脚

            .ctsnPin     = GPIOD_Pin1, //串口发送流控脚

#else

            .rtsnPin     = GPIO_INVAILD, //串口接收流控脚

            .ctsnPin     = GPIO_INVAILD, //串口发送流控脚

#endif

            .rxPin       = GPIOB_Pin2, //串口接收脚

            .txPin       = GPIOB_Pin3 //串口发送脚

        };

        Config.baud_rate = get_baudrate();

        ui_uart_configuration(ws_uart1,&Config,1);

        uart_param.uart_handler = user_uart_rcv_cbk;

        reg_uart_rx_callback(ws_uart1,&uart_param);

// 【4】 备注:

//串口 RX 为输入
```

```
//TX 为输出

//在深度低功耗状态下，串口外设失电,串口控制寄存器数据丢失,RX 退化为悬浮输入的普通 GPIO
状态

//基于上述条件 需要注意:

//A:退出低功耗串口外设需要重新初始化

//B:不接负载，RX TX 需要接 3M 上拉电阻

//C:接入串口负载，串口默认电平为高电平，不影响。

}
```

11.3 注意事项

中断接收方式，分为

APP_USART_IRQ 字符中断模式

APP_USART_FIFO_IRQ FIFO 中断模式

FIFO 中断模式可配置下列方式

RX 中断方式

UART_FIFO_RX_Trigger_1_Char

UART_FIFO_RX_Trigger_1_4_Full

UART_FIFO_RX_Trigger_1_2_Full

UART_FIFO_RX_Trigger_2_CharLessFull

TX 中断方式

UART_FIFO_TX_Trigger_Empty

UART_FIFO_TX_Trigger_2_Chars

UART_FIFO_TX_Trigger_1_4_Full

UART_FIFO_TX_Trigger_1_2_Full

注意：16M 晶振下，少量数据才可用 APP_USART_IRQ

大量数据需要使用 APP_USART_FIFO_IRQ FIFO 中断模式。

APP_USART_FIFO_IRQ 需要使用中断模式和查询模式相结合，查询模式，防止丢数据，同时防止 FIFO 中有数据进入 DEEP_SLEEP，中断模式，实时性高，两者结合取得较好性能，ws8100_app_standard_at 使用该模式。

12、TIMER

WS8100 timer 属于非常开电源域设备。

WS8100 支持 2 个 TIMER 分频器，每路分频器，配置 2 个 timer,共 4 路 timer

Timer 寄存器比较少，直接放入 backup_table_s backup_table，退出 DEEP_SLEEP 低功耗模式，不需要手动恢复，自动完成恢复。

12.1 设置 TIMER 工作状态

```
void example_timer_set(functional_state_t enable)
{
    if(enable){
        //          dbg_printf("TimerStart\n");
// 【1】启动 timer
        timer_enable_ctrl_cmd(TIMER0, TIMER_START);
// 【2】设置 timer 工作状态，阻止进入 DEEP_SLEEP
        cpu_prevent_sleep_set(TIMER0_ONGOING);
    }else{
        //          dbg_printf("TimerStop\n");
// 【3】停止 timer
        timer_enable_ctrl_cmd(TIMER0, TIMER_STOP);
// 【2】清除 timer 工作状态，解决阻止进入 DEEP_SLEEP
        cpu_prevent_sleep_clear(TIMER0_ONGOING);
    }
}
```

12.2 TIMER 示例代码

```
//=====
//****          定时器应用举例
//=====

void example_timer_cbk(void)
{
    dbg_block_printf("*");
}

void example_timer_set(functional_state_t enable)
{
    if(enable){
        //          dbg_printf("TimerStart\n");
```

```
        timer_enable_ctrl_cmd(TIMER0, TIMER_START);

        cpu_prevent_sleep_set(TIMER0_ONGOING);//
    }else{

        //dbg_printf("TimerStop\n");

        timer_enable_ctrl_cmd(TIMER0, TIMER_STOP);

        cpu_prevent_sleep_clear(TIMER0_ONGOING);    //

    }

}

/*****

函 数 名   : user_timer_example

功能描述   : 定时器应用举例

输入参数   : void

输出参数   : 无

*****/

void user_timer_example(void)
{
    dbg_block_printf("app service initial timer ready\r\n");

    timer_config_t timer_param;

// 【1】 开启定时器 0 时钟

    clock_config(ws_timer0,ENABLE);

// 【2】 开启定时器 0 中断

    nvic_init_type_def nvic_init_structure;

    NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);

    nvic_init_structure.nvic_irq_channel_preemption_priority = 1;

    nvic_init_structure.nvic_irq_channel_subpriority          = 0;

    nvic_init_structure.nvic_irq_channel_cmd                  = ENABLE;

    nvic_init_structure.nvic_irq_channel                      = Timer0_IRQn;

    nvic_init(&nvic_init_structure);

// 【3】 初始化定时器参数以及注册回调函数

    timer_param.timer_idx = IDX_TIMER0;

    timer_param.timer_param.timer_period = 10* 16000; // 10 ms
```

```
timer_config(TIMER0, &timer_param, example_timer_cbk);
```

```
//【4】 使能定时器
```

```
example_timer_set(ENABLE);
```

```
}
```

13、ADC

13.1 ADC 简介

WS8100 有一路 ADC，可分时复用。

ADC 管脚，可配置 GPIOA_Pin0~GPIOA_Pin7

ADC 电压采集范围：0~1200mV

ADC 位数 10 位

ADC 工作模式：查询模式，中断模式。

注意：

WS8100 为保证工作稳定，配备了温度漂移频偏补偿功能，用到了 ADC，因此使用 ADC 前，需要重新配置，禁用 WS8100 晶振校准，ADC 使用完成，开启 WS8100 晶振校准。

13.2 ADC 查询模式示例代码

```
// 推荐使用方式
```

```
/******
```

```
函 数 名 : user_adc_inquire_example
```

```
功能描述 : adc 查询模式示例代码
```

```
输入参数 : void
```

```
输出参数 : 无
```

```
*****/
```

```
void user_adc_inquire_example(void)
```

```
{
```

```
int i;
```

```
//【1】 开启 ADC 时钟
```

```
clock_config(ws_adc,ENABLE);
```

```
//【2】 查询模式
```

```
while(1){
```

```
//【3】 停用温度校准
```

```
hwadjust_get_temperature_adc_data_period_stop();

// 【4】 设置 ADC 工作事件标志

cpu_prevent_sleep_set(AMIC_ONGOING);

// 【5】 配置 ADC

user_gpio_dac_configuration(GPIOA_Pin7);

// 【6】 适当延时

for(i=0;i<100;i++);

// 【7】 获取 ADC 值

dbg_block_printf("%d\n",get_dac_value());

// 【8】 清除 ADC 工作事件标志

cpu_prevent_sleep_clear(AMIC_ONGOING);

// 【9】 恢复温度校准

hwadjust_get_temperature_adc_data_period_start(NULL,NULL);

}

}
```

13.3 中断模式示例代码

```
/*
*****

函 数 名 : GPADC_IRQHandler

功能描述 : adc 转换完成中断函数

输入参数 : 无

输出参数 : 无

*****
*/

void GPADC_IRQHandler()
{
    uint16_t gpadc_data;

    gpadc_data = GPADC->GPADC_FIFODATA;//从 fifo 中读取数据

    dbg_block_printf("%d\n",gpadc_data);

    gpadc_clear_it_pending_bit(GPADC_IT_ALERT_CLEAR);//清除中断标志位

}

/*
*****

函 数 名 : user_int_adc_example
```

功能描述 : adc 中断模式示例代码

输入参数 : void

```
*****/

void user_int_adc_example(void)
{
    int i;

    // 【1】 开启 ADC 时钟
    clock_config(ws_adc,ENABLE);

    // 【2】 开启 ADC 中断
    nvic_init_type_def nvic_init_structure;
    NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);
    nvic_init_structure.nvic_irq_channel_preemption_priority = 1;
    nvic_init_structure.nvic_irq_channel_subpriority          = 0;
    nvic_init_structure.nvic_irq_channel_cmd                  = ENABLE;
    nvic_init_structure.nvic_irq_channel                      = GPADC_IRQn;

    // 【3】 使能 ADC 转换完成中断
    gpadc_it_config(GPADC_IT_ALERT_EN,ENABLE);
    nvic_init(&nvic_init_structure);

    // 【4】 中断模式配置
    gpadc_irq_configuration(GPIOA_Pin7);

    // 【5】 一旦温度校准重新配置了 ADC 后，用户需要再次配置，因此此处用 while(1) 来阻止
    while(1){
    }
}
```

14、PWM

14.1 PWM 简介

WS8100 有两路 PWM，每路 PWM 可映射一个或多个 IO 同时输出。PWM 时钟来源 TIMER 0 号分频器。

任意 IO 可做 PWM 输出控制用

14.2 PWM 示例代码

第 64 页 共 114 页

输出参数 : 无

```
*****/

void pwm_breath_led_config(void)
{
    timer_config_t  rgb_timer_param;

    rgb_env.rgb_param.scan_interval = 160000*4; // scan interval defaule 20ms (ms) 定时器周期

    rgb_env.rgb_param.scan_period    = 20;// scan times//

    rgb_env.rgb_param.scan_duty_start = 0;// io row map// 动态变化

    rgb_env.rgb_param.scan_duty_end  = 20;// io row map//

    rgb_env.rgb_param.next_timer_cnt = 0;// 持续多久切换到下一个周期 3 秒

    rgb_env.rgb_param.r_pin= GPIOC_Pin0;// 由 PD2 更换为 PD0

    rgb_env.rgb_param.g_pin      = GPIOC_Pin1;//

    rgb_env.rgb_param.b_pin= GPIOC_Pin2;// 由 PD3 更换为 PD1

    rgb_env.step

    dbg_printf("rgb config \n");
}
```

// 【1】 开启时钟

```
clock_config(ws_timer0,ENABLE);
clock_config(ws_timer1,ENABLE);
```

// 【2】 开启中断

```
nvic_init_type_def nvic_init_structure;
NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);
nvic_init_structure.nvic_irq_channel_preemption_priority= 1;
nvic_init_structure.nvic_irq_channel_subpriority= 0;
nvic_init_structure.nvic_irq_channel_cmd= ENABLE;
```

//PWM 中断

```
nvic_init_structure.nvic_irq_channel = Timer0_IRQn;
nvic_init(&nvic_init_structure);
```

//Adjust Timer 中断

```
NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);

nvic_init_structure.nvic_irq_channel_preemption_priority = 1;

nvic_init_structure.nvic_irq_channel_subpriority          = 0;

nvic_init_structure.nvic_irq_channel_cmd                  = ENABLE;

nvic_init_structure.nvic_irq_channel                      = Timer1_IRQn;

nvic_init(&nvic_init_structure);


// 【3】 timer 初始化

rgb_timer_param.timer_idx                                = IDX_TIMER1;

rgb_timer_param.timer_param.timer_period                  = rgb_env.rgb_param.scan_interval;

timer_config(TIMER1,&rgb_timer_param,breath_timer_callback);


// 【4】 timer 使能

timer_enable_ctrl_cmd(TIMER1, TIMER_START);

cpu_prevent_sleep_set(TIMER1_ONGOING);


// 【5】 pwm 初始化并启动

rgb_breath_pwm_start();

}
```

15、SPI

15.1 SPI 简介

WS8100 支持 2 路 SPI

SPI 最高速都支持 2 分频，最高速度和晶振有关，方案评估，据最最高要求速度，选择 16M 晶振还是 32M 晶振。

WS8100 SPI 支持 MASTER 模式和 SLAVER 模式。

16、I2C

16.1 I2C 简介

WS8100 支持一路 I2C,可配置主从模式，速率 100~400K，SCL SDA 可映射任意 IO

16.2 I2C 示例代码

```

/*****

函 数 名   : iic_configuration

功能描述   : iic 配置

输入参数   : void

输出参数   : 无

返 回 值   :

*****/

void iic_configuration(void)
{
    i2c_init_type_def i2c_init_struct;

    i2c_struct_init(&i2c_init_struct);

    i2c_init_struct.i2c_clock_speed    = I2C_ClockSpeed_300KHz;//速率配置

    i2c_init_struct.i2c_mode           = I2C_Mode_Master;      //模式选择

    i2c_init_struct.i2c_duty_cycle     = I2C_DutyCycle_1;      //时钟控制

    i2c_init_struct.i2c_target_address = 0x24;

    i2c_init_struct.i2c_target_address_mode = I2C_TargetAddressMode_7bit; //目标地址模式: 7bit

    //I2C GPIO 管脚映射

    gpio_remap_config(GPIOD_Pin4,GPIO_Remap_I2C_SCL);

    gpio_remap_config(GPIOD_Pin5,GPIO_Remap_I2C_SDA);

    i2c_init(I2C,&i2c_init_struct);//初始化

    i2c_it_config(I2C,I2C_IT_RXF      |      I2C_IT_TXE      |      I2C_IT_RD_REQ      |
    I2C_IT_RX_DONE,ENABLE);//中断使能

    i2c_cmd(I2C,ENABLE);//I2C 使能

}

/*****

函 数 名   : user_iic_example

功能描述   : iic 用户举例

输入参数   : void

输出参数   : 无

*****/

void user_iic_example(void)
```

```
{  
    uint8_t buf[10];  
    // 【1】 开启 I2C 时钟  
    clock_config(ws_i2c,ENABLE);  
  
    // 【2】 开启 I2C 中断  
    nvic_init_type_def nvic_init_structure;  
    NVIC_SetPriorityGrouping(NVIC_PriorityGroup_2);  
    nvic_init_structure.nvic_irq_channel_preemption_priority= 1;  
    nvic_init_structure.nvic_irq_channel_subpriority= 0;  
    nvic_init_structure.nvic_irq_channel_cmd= ENABLE;  
    nvic_init_structure.nvic_irq_channel = I2C_IRQn;  
    nvic_init(&nvic_init_structure);  
  
    // 【3】 初始化 IIC  
    iic_configuration();  
  
    // 【4】 从 0x01 寄存器开始读取数据  
    i2c_send_data(I2C, 0x01, I2C_DataEndCondition_Stop);  
  
    // 【5】 读取数据  
    i2c_receive_bytes(I2C, buf, 5, I2C_DataEndCondition_Stop);  
  
    // 【4】 向 0x01 寄存器开始写入数据  
    i2c_send_data(I2C, 0x01, I2C_DataEndCondition_Stop);  
  
    // 【7】 发送数据  
    i2c_send_bytes(I2C, buf, 5, I2C_DataEndCondition_Stop);  
}
```

17、FLASH 读写

17.1 FLASH 擦除

```
uint32_t flash_erase(uint32_t address, uint32_t size)
```

17.2 写操作

```
uint32_t flash_write(uint32_t address, uint32_t size, uint8_t *buf)
```

17.3 读操作

```
uint32_t flash_read(uint32_t address, uint32_t size, uint8_t *buf)
```

17.4 Flash 读写示例代码：

1.1 默认分区大小

表 1-1 512K Flash 分区表

512K Flash 分区表	起始地址	大小
UserData2	0x0107f000	4k
UserData1	0x0107a000	20k
Free	0x01048000	200k
App	0x01016000	200k
Stack	0x01002000	80k
Sbl	0x01001000	4k
PartInfo	0x01000000	4k

用户可操作空间为 0x0107A0000~0x0107EFFF 共 20K。

UserData2 分区用来加载量产工具烧录的用户信息。

```
/******
```

函 数 名 : flash_example

功能描述 : flash 读写操作示例代码

输入参数 : void

输出参数 : 无

返 回 值 :

```
*****/
```

```
void flash_example(void)
```

```
{
```

```
    #define TEST_ADDR_FLASH
```

```
    char buf[64];
```

```
    char read_buf[64];
```

```
    char i;
```

```
    for(i=0;i<64;i++){
```

```
        buf[i] = i*2;
```

```
        read_buf[i] = 0;
```

```
    }

    // 【1】擦除 falsh。    起始地址要求 4K 对齐，擦除大小也要求 4K 对齐
    flash_erase(TEST_ADDR_FLASH, 0x1000);

    // 【2】写 falsh。    要求写在擦出的区域，起始地址要求 4K 对齐
    flash_write(TEST_ADDR_FLASH, sizeof(buf), (uint8_t *) (buf));

    // 【3】读取 falsh。

    flash_write(TEST_ADDR_FLASH, sizeof(read_buf), (uint8_t *) (read_buf));

    for(i=0;i<64;i++){
        if(buf[i] != read_buf[i]){
            dbg_printf("err ...\n");
        }
    }
}
```

17.5 加载量产工具用户自定义蓝牙名称

```
/*
*****

函 数 名   : flash_smp_info_read_name
功能描述   : 加载用户烧录蓝牙名称
输入参数   : uint32_t address
              uint32_t offset
              char *name_buf
              uint16_t name_size
返回参数   : 加载是否成功
*****

uint8_t flash_smp_info_read_name(uint32_t address, uint32_t offset, char *name_buf, uint16_t
name_size)
{
    char i, read_buf[64];

    char sz = 0;

    flash_read(address + offset, 64, (uint8_t *) (&read_buf));

    for(i = 0; i < 64; i++)
    {
        if(read_buf[i] != 0xFF){
```

```
        sz = 1;

        break;

    }

}

if(read_buf[0] == 0xFF){

    return 0;

}

if(sz){

    for(i = 0;i < 64;i ++){

        {

            name_buf[i] = read_buf[i];

            if(read_buf[i] == 0xFF)

            {

                read_buf[i]  = '\0';

                name_buf[i] = '\0';

                break;

            }

        }

    }

}

return sz;

}
```

17.6 加载量产工具用户自定义蓝牙地址

/******

函 数 名 : flash_smp_info_read_address

功能描述 : 加载用户烧录地址信息

输入参数 : uint32_t address

uint32_t offset

struct hci_address_stru *addr

返回参数 : 加载是否成功

*****/

```
uint8_t flash_smp_info_read_address(uint32_t address, uint32_t offset,struct hci_address_stru
*addr)
```

```
{
```

```
char i,read_buf[6];

uint8_t sz;

flash_read(address + offset,6,(uint8_t *)read_buf);

dbg_block_printf("addr :: ",read_buf,6);

sz = 0;

for(i = 0;i < 6;i++)
{
    if(read_buf[i] != 0xFF)
    {
        sz = 1;
        break;
    }
}

if(sz == 0)

    return 0;

for(i=0;i<6;i++){

    addr->bd[i] = read_buf[5-i];

}

addr->bd[5] |= 0xC0;

addr->atype = 1;//          0:公共地址          1:私有地址

dbg_printf("Random Adress :: %02X:%02X:%02X:%02X:%02X:%02X\n",

          addr->bd[0],addr->bd[1],addr->bd[2],

          addr->bd[3],addr->bd[4],addr->bd[5]);

return sz;

}
```

18、蓝牙部分

18.1、全向广播

18.1.1、广播类型

有限可发现不可连接广播

APP_BLE_MODE_LIMITED_DISCOVERABLE_NONE_CONNECTABLE

常规可发现不可连接广播

APP_BLE_MODE_GENERAL_DISCOVERABLE_NONE_CONNECTABLE

不可发现，可连接广播

APP_BLE_MODE_NONE_DISCOVERABLE_CONNECTABLE

有限可发现可连接广播

APP_BLE_MODE_LIMITED_DISCOVERABLE_CONNECTABLE

常规可发现可连接广播

APP_BLE_MODE_GENERAL_DISCOVERABLE_CONNECTABLE

/******

函 数 名 : set_ble_adv_mode

功能描述 : 设置广播模式

输入参数 : app_ble_mode_t mode 可选参数:

APP_BLE_MODE_GENERAL_DISCOVERABLE_NONE_CONNECTABLE:普通可发现不可连接广播

APP_BLE_MODE_GENERAL_DISCOVERABLE_CONNECTABLE普通可发现可连接广播

APP_BLE_MODE_LIMITED_DISCOVERABLE_NONE_CONNECTABLE 有
限不可连接广播

APP_BLE_MODE_LIMITED_DISCOVERABLE_CONNECTABLE 有限
可连接广播

输出参数 : 无

返 回 值 :

*****/

void set_ble_adv_mode(app_ble_mode_t mode)

{

 UINT16 eir_mode = HCI_EIR_FLAGS_BREDR_NOT_SUPPORTED;

 switch (mode) {

case APP_BLE_MODE_GENERAL_DISCOVERABLE_NONE_CONNECTABLE:

 case APP_BLE_MODE_GENERAL_DISCOVERABLE_CONNECTABLE:

 eir_mode |= HCI_EIR_FLAGS_LE_GENERAL_DISCOVERABLE_MODE;

 break;

case APP_BLE_MODE_LIMITED_DISCOVERABLE_NONE_CONNECTABLE:

 case APP_BLE_MODE_LIMITED_DISCOVERABLE_CONNECTABLE:

 eir_mode |= HCI_EIR_FLAGS_LE_LIMITED_DISCOVERABLE_MODE;

 break;

```
        default:
            break;
    }

    bt_set_info(HCI_EIR_DATATYPE_FLAGS,&eir_mode,1);//修改广播名称
}
```

18.1.2、广播包

```
typedef enum _ADV_TYPE{
    ADV_NORMAL, //用户不必指定 广播数组，根据蓝牙名称等信息自动合成广播包
    ADV_USER,   //用户指定广播数组 必须符合格式的数组 const uint8_t adverData
    ADV_DAT,    //用户指定广播数组 用户自定义广播数据
}ADV_TYPE;

ADV_USER 广播包数据举例

const uint8_t adverData[31] =
{
    0x02,//length of this data
    HCI_EIR_DATATYPE_FLAGS,
    APP_BLE_MODE_GENERAL_BIT | APP_BLE_MODE_ADVERTISING_BIT,
    //complete name
    0x0e,//length of this data
    HCI_EIR_DATATYPE_COMPLETE_LOCAL_NAME,
    'B','L','E','-','t','r','a','n','s','p','o','n','d',
    0x0
};

ADV_DAT
数据来源于 ble_inf.ad_inf.user_data
```

18.2.3、广播回应包

```
static uint8_t scanRspData[] =
{
    0x00
};

数据格式遵循 const uint8_t adverData[31];
```

广播内容过多，放不下，可以放在此处，等待广播查询的时候，响应广播数据，效果等同，放在广播数组中。

18.3.4、广播间隔

```
void start_advertising
(
    app_ble_mode_t mode,UINT16 adv_min_interval, UINT16 adv_max_interval)
{
    app_gap_set_ble_operation_modes
    (mode, adv_min_interval, adv_max_interval,NULL);
}

默认广播间隔

//=====默认广播间隔===== 广播间隔单位 0.625ms 理论值范围 6~16384

//=====受限于连接间隔 【7.5ms~4s】=====，可连接广播 要求 20~3999ms 之间

#define GENERAL_ADV_MIN_INTERVAL      1600 // 1 秒

#define GENERAL_ADV_MAX_INTERVAL      1600 // 1 秒

// 尽量最大广播间隔，最小广播间隔，不一致，减少广播撞车概率。
```

18.2、定向广播

定向广播无法广播附加信息，广播只能载地址信息，只有高周期广播和低周期广播之分。两次广播间隔要求> 1.3s

定向广播用作回连用。回连速度比全向广播快。

18.3、停止广播

```
void stop_advertising(void)
{

    app_gap_set_ble_operation_modes(APP_BLE_MODE_NONE_DISCOVERABLE_NONE_CONNECTABLE, NULL, NULL,NULL);

    //stop_advertising_event();
}
```

18.4、发射功率

WS8100 支持以下发射功率配置：

```
TX_POWER_MINIUX_20,
TX_POWER_MINIUX_15,
TX_POWER_MINIUX_10,
```

```
TX_POWER_MINUX_5,  
TX_POWER_MINUX_4,  
TX_POWER_MINUX_3,  
TX_POWER_MINUX_2,  
TX_POWER_MINUX_1,  
TX_POWER_0,  
TX_POWER_1,  
TX_POWER_2,  
TX_POWER_3,  
TX_POWER_4,  
TX_POWER_7,
```

18.4.1 设置蓝牙发射功率：

```
radio_set_tx_power(TX_POWER_0);
```

18.5、蓝牙连接

蓝牙连接为回调执行。

18.5.1、连接上事件

```
void app_ble_connected_ind(struct hci_address_stru *addr)  
{  
    dbg_block_printf("connected\n");  
    bt_stop_advertising();  
    remove_adv_time_out_tsk();  
    memcpy(&app_env.addr,addr,sizeof(struct hci_address_stru));  
    ble_connect_event(TRUE,addr);  
  
    #if 1  
    #if SDK_LEANCY_ENABLE  
        app_latency_enable(1);  
    #else  
        app_latency_enable(0);  
    #endif  
    #endif  
}
```

15.5.2、断开连接事件

```
void app_cbk_ble_disconn_ind(struct hci_address_stru *addr, UINT8 reason)
{
    // HCI_STATUS_REMOTE_USER_TERMINATED_CONNECTION 用户主动断开

    dbg_block_printf("disconnected reason %0x\n",reason);

    // 【1】 进入待机之前，先断开蓝牙，否则连接断开超时过大，主机端检测到断开会延时很久。

    if(enter_standbye_flg){
        enter_standbye_deepsleep(SYSCFG_noRetention_Sram);

        enter_standbye_flg = 0;
    }

    // 【2】 蓝牙非停止工作状态

    if(get_app_stat() != BT_STATE_STOP){
        set_app_stat(BT_STATE_CONNECT_BROKEN);

        task_latency = 0xFF;
    }

    // 【3】 蓝牙非停止工作状态 系统停止工作，准备重启了，防止蓝牙事件打断重启过程

    if(task_stop){
        return;
    }

    // 【4】 检测到断开重启广播

    if(ble_inf.bt_enable && ble_inf.bt_adv_enable)
    {
        bt_start_advertising();
    }else{
        bt_stop_advertising();
    }
}

ble_connect_event(FALSE,addr);
}
```

18.6、蓝牙安全

18.7、蓝牙地址

WS8100 支持三种模式的蓝牙地址设置

1、片上 OTP 地址

2、用户地址

3、随机地址

备注：蓝牙地址设置后，需要重启广播生效。

18.7.1 蓝牙开机地址设置示例代码

```
/*
*****

函 数 名   : app_gap_set_default_public_bd_addr
功能描述   : 写入公共地址
输入参数   : struct hci_address_stru *bd
输出参数   : 无
返 回 值   :

*****

void app_gap_set_default_public_bd_addr(struct hci_address_stru *bd)
{
    struct hci_le_vector_set_dbaddr_stru *req;

    req = NEW(sizeof(struct hci_le_vector_set_dbaddr_stru));

    req->type      = 1; //< only le addr

    memcpy(req->addr,bd->bd,6);

    gap_vendor_commanda(NULL,req,HCI_EVI_COMMAND_COMPLETE,4,req,sizeof(struct
    hci_le_vector_set_dbaddr_stru),app_gap_vector_set_bd_addr_cfm);
}

/*
*****

函 数 名   : app_gap_set_default_public_bd_addr
功能描述   : 设置公共地址
输入参数   : struct hci_address_stru *bd
输出参数   : 无
返 回 值   :

*****

void app_gap_set_default_public_bd_addr(struct hci_address_stru *bd)
{
    struct hci_le_vector_set_dbaddr_stru *req;
```

```
req = NEW(sizeof(struct hci_le_vector_set_dbaddr_stru));

req->type          = 1; //< only le addr

memcpy(req->addr,bd->bd,6);

gap_vendor_commanda(NULL,req,HCI_EVI_COMMAND_COMPLETE,4,req, sizeof(struct
hci_le_vector_set_dbaddr_stru),app_gap_vector_set_bd_addr_cfm);

}
```

18.7.2 蓝牙用户地址设置示例代码

```

/*****

函 数 名   : app_gap_set_user_addr
功能描述   : 设置用户地址
输入参数   : struct hci_address_stru *addr
输出参数   : 无
返 回 值   :

*****/

void app_gap_set_user_addr(struct hci_address_stru *addr)
{
    struct hci_address_stru *r_addr = NEW(sizeof(struct hci_address_stru));

    memcpy(r_addr,addr,sizeof(struct hci_address_stru));

    app_gap_clear_random_addr();

    gap_le_set_adv_address(0,GAP_LE_ADDRTYPE_STATIC);

    FsmEvent_ExternalTx(app_update_addr_exec, r_addr);
}

```

18.7.3 蓝牙随机地址设置示例代码

```

/*****

函 数 名   : app_gap_set_random_addr
功能描述   : 设置随机地址
输入参数   : struct hci_address_stru *addr
输出参数   : 无
返 回 值   :

*****/

```

```
void app_gap_set_random_addr(void)
{
    struct hci_address_stru *r_addr = NEW(sizeof(struct hci_address_stru));
    user_get_random_addr(r_addr);

    // 保存随机地址，写入 flash，作为下次开机随机函数发生器的随机种子
    updata_mac(r_addr);

    app_gap_clear_random_addr();
    gap_le_set_adv_address(0,GAP_LE_ADDRTYPE_STATIC);
    FsmEvent_ExternalTx(app_update_addr_exec, r_addr);
}
```

18.9、蓝牙名称

18.9.1 修改蓝牙名称示例代码:

```
/*
*****

函 数 名   : set_ble_name
功能描述   : 修改蓝牙名称
输入参数   : char *name
输出参数   : 无
*****
*/

void set_ble_name(char *name)
{
    // 【1】更新广播中蓝牙名称
    bt_set_info(HCI_EIR_DATATYPE_COMPLETE_LOCAL_NAME,name, strlen(name));//修改广播名称

    // 【2】更新 GAP 层蓝牙名称
    app_gap_set_local_name(name, strlen(name));
}
```

蓝牙名称过长，广播包只能有 31 字节，可将蓝牙名称放入 scanRspData 中，通过交互的方式获取蓝牙名称。

18.10、蓝牙 UUID

18.10.1 修改蓝牙 UUID 广播示例代码:

```
/*
*****
```

函 数 名 : set_ble_name

功能描述 : 修改蓝牙名称

输入参数 : char *name

输出参数 : 无

```
*****/

void set_ble_uuid16(uint16_t uuid)
{
    // 【1】更新广播中蓝牙 UUID

    bt_set_info(HCI_EIR_DATATYPE_COMPLETE_LIST_OF_16BIT_SERVICE_CLASS_UUIDS,&
uuid, 2);                                //修改广播名称

    // 【1】gap 层 uuid 为服务注册时候写入的
}
```

18.10.2 GAP 层 UUID:

GAP 层 uuid 为 128 位，可简化位 32 位或 16 位

```
/******

函 数 名 : att_uuid16_to_u128
功能描述 : uuid16 转 uuid128
输入参数 : uint8_t uuid128[16]

            struct att_uuid_stru *out
            uint8_t base[12]
            uint16_t uuid16

输出参数 : 无

*****/

void att_uuid16_to_u128(uint8_t uuid128[16],struct att_uuid_stru *out,uint8_t base[12],uint16_t
uuid16)
{
    uint8_t i;

    memset(out,0,sizeof(struct att_uuid_stru));

    ENCODE2BYTE_LITTLE (&(out->u[12]),uuid16);

    for(i=0;i<12;i++){
        out->u[i]=base[i];
    }
}
```

```
        memcpy(&uuid128,out->u,16);
    }

    void app_gatts_register_iot_service(void)
    {
        struct gatt_service_stru *service;

        struct gatt_characteristic_stru *character;

        struct att_uuid_stru uuid;

        UINT8 val[2] = {0, 0};

        // 6E40 0001-B5A3-F393-E0A9-E50E-24DC-CA9E
        // 6E40 0002-B5A3-F393-E0A9-E50E-24DC-CA9E
        // 6E40 0003-B5A3-F393-E0A9-E50E-24DC-CA9E
        // 0000-1000-8000-0080-5F9B-34FB
        //小端编码

        #if (UUID_TEST16 || UUID_TEST32)

        //uint8_t base_uuid[12]={0xFB,0x34,0x9B,0x5F,
        //
        //                0x80,0x00,0x00,0x80,
        //
        //                0x00,0x10,0x00,0x00};

        uint8_t Base_Uuid[12]={0x9E,0xCA,0xDC,0x24,
        //
        //                0x0E,0xE5,0xA9,0xE0,
        //
        //                0x93,0xF3,0xA3,0xB5};

        #endif

        // 【1】 申请服务实例

        service = (struct gatt_service_stru *)List_NodeNew(sizeof(struct gatt_service_stru));

        // 【2】 注册服务回调函数

        service->cbk = app_gatts_iot_service_cbk;

        #if UUID_TEST16

        // 【3】 base 部分用户指定，用 16 位 uuid 转 128 位 service uuid

        att_uuid16_to_u128(Service_Uuid,&(service->uuid),Base_Uuid,APP_ATT_UUID_IOT_SERVICE);

        #elif UUID_TEST32

        // 【3】 base 部分用户指定，用 32 位 uuid 转 128 位 service uuid
```

```
att_uuid32_to_u128(Service_Uuid,&(service->uuid),Base_Uuid,APP_ATT_UUID_IOT_SERVICE_
32);

#else

// 【3】 base 部分用系统默认的转 128 位 service uuid

att_sys_uuid16_to_u128(Service_Uuid,&(service->uuid), APP_ATT_UUID_IOT_SERVICE);

#endif


// ===== TX uuid =====

#if UUID_TEST16

// 【4】 base 部分用户指定, 用 16 位 uuid 转 128 位 character uuid

att_uuid16_to_u128(TxCharacteristic_Uuid,&uuid,Base_Uuid,APP_ATT_UUID_IOT_TX_PORT);

#elif UUID_TEST32

// 【4】 base 部分用户指定, 用 32 位 uuid 转 128 位 character uuid

att_uuid32_to_u128(TxCharacteristic_Uuid,&uuid,Base_Uuid,APP_ATT_UUID_IOT_TX_PORT_
32);

#else

// 【4】 base 部分用系统默认的转 128 位 character uuid

att_sys_uuid16_to_u128(TxCharacteristic_Uuid,&uuid, APP_ATT_UUID_IOT_TX_PORT);

#endif

// 【5】添加 character 到 service 树上

character = gatt_add_characteristic(service, val, 2, &uuid, ATT_NOTIFY);


// 【6】添加发送 character ccc

gatt_add_descriptor_ccc(character, ATT_CCCMASK_NOTIFICATION,
ATT_READ|ATT_WRITE);


// ===== RX uuid =====

#if UUID_TEST16

// 【7】 base 部分用户指定, 用 16 位 uuid 转 128 位 character uuid

att_uuid16_to_u128(RxCharacteristic_Uuid,&uuid,Base_Uuid,APP_ATT_UUID_IOT_RX_PORT);

#elif UUID_TEST32

// 【7】 base 部分用户指定, 用 32 位 uuid 转 128 位 character uuid

att_uuid32_to_u128(RxCharacteristic_Uuid,&uuid,Base_Uuid,APP_ATT_UUID_IOT_RX_PORT_
32);
```

```
#else

// 【7】 base 部分用系统默认的转 128 位 character uuid
att_sys_uuid16_to_u128(RxCharacteristic_Uuid,&uuid, APP_ATT_UUID_IOT_RX_PORT);

#endif

// 【8】添加 character 到 service 树上

gatt_add_characteristic(service, val, 2, &uuid, ATT_WRITE_NORSP);


// 【9】注册 service

gatt_server_reg_tree(service);

}
```

18.11、蓝牙服务

18.11.1 添加服务

18.11.1.1 服务创建

WS8100 支持多个服务，每个服务树下面可以挂多个 character

服务创建流程：

- 1、申请服务实例
- 2、设置服务事件回调函数
- 3、设置服务 uuid
- 4、在服务树下挂 character
- 5、注册服务树

character 创建流程：

- 1、设置 character UUID
- 2、基于 service 创建 character
- 3、具备发送属性的 character 需要写 ccc

18.11.1.2 添加服务示例代码

```
/******

函 数 名 : app_gatts_register_defl_service

功能描述 : 添加自定义服务 1

输入参数 : void

输出参数 : 无

*****/
```

```
void app_gatts_register_defl_service(void)
{
    struct gatt_service_stru *service;

    struct gatt_characteristic_stru *character;

    struct att_uuid_stru uuid;

    UINT8 val[2] = {0, 0};

    // 【1】 申请服务

    service=(structgatt_service_stru *)List_NodeNew(sizeof(struct gatt_service_stru));

    // 【2】 注册服务回调函数

    service->cbk = app_gatts_def_service_cbk;///

    // 【3】 设置服务 UUID

    att_sys_uuid16_to_u128(Service_defl_Uuid0,&(service->uuid),
APP_ATT_UUID_AE00_SERVICE);


    // 【3】 添加 character1

    // ===== uuid RX ae01=====

    // character step 【1】 设置 character uuid

    att_sys_uuid16_to_u128(Characteristic_RX_AE01_Uuid,&uuid, APP_ATT_UUID_RX_AE01);


    // character step 【2】 设置 character 属性, 将 character 挂到 service 树上

    character = gatt_add_characteristic(service, val, 2, &uuid, ATT_WRITE_NORSP);


    // 【4】 添加 character2

    // ===== uuid TX ae02=====

    // character step 【1】 设置 character uuid

    att_sys_uuid16_to_u128(Characteristic_TX_AE02_Uuid,&uuid, APP_ATT_UUID_TX_AE02);


    // character step 【2】 设置 character 属性, 将 character 挂到 service 树上

    character = gatt_add_characteristic(service, val, 2, &uuid, ATT_NOTIFY);


    // character step 【3】 具备发送 character 需要添加 ccc, 激活通知属性, 接收 character 无
    此特性

    gatt_add_descriptor_ccc(character, ATT_CCCMASK_NOTIFICATION,
ATT_READ|ATT_WRITE);
```

```
// 【5】 添加 character3

// ===== uuid RX AE03=====

    att_sys_uuid16_to_u128(Characteristic_RX_AE03_Uuid,&uuid,
APP_ATT_UUID_RX_AE03);

    character = gatt_add_characteristic(service, val, 2, &uuid, ATT_WRITE_NORSP);


// 【6】 添加 character4

// ===== uuid TX AE04=====

    att_sys_uuid16_to_u128(Characteristic_TX_AE04_Uuid,&uuid,
APP_ATT_UUID_TX_AE04);

    character = gatt_add_characteristic(service, val, 2, &uuid, ATT_NOTIFY);

    gatt_add_descriptor_ccc(character,                ATT_CCCMASK_NOTIFICATION,
ATT_READ|ATT_WRITE);


// 【7】 添加 character5

// ===== uuid INDICATE AE05=====

    att_sys_uuid16_to_u128(Characteristic05_Uuid,&uuid, APP_ATT_UUID_AE05);

    gatt_add_characteristic(service, val, 2, &uuid, ATT_INDICATE);


// 【8】 添加 character6

// ===== uuid RTX AE10=====

    att_sys_uuid16_to_u128(Characteristic_RXTX_AE10_Uuid,&uuid,
APP_ATT_UUID_AE10);

    character = gatt_add_characteristic(service, val, 2, &uuid, ATT_READ|ATT_WRITE);

    gatt_add_descriptor_ccc(character,                ATT_CCCMASK_NOTIFICATION,
ATT_READ|ATT_WRITE);


// 【8】 注册 service 树

    gatt_server_reg_tree(service);

}

/*****

函 数 名   : app_gatts_register_def2_service

功能描述   : 添加自定义服务 2

*****/
```

输入参数 : void

输出参数 : 无

返回值 :

```
*****/

void app_gatts_register_def2_service(void)
{
    struct gatt_service_stru *service;

    struct gatt_characteristic_stru *character;

    struct att_uuid_stru uuid;

    UINT8 val[2] = {0, 0};

    service = (struct gatt_service_stru *)List_NodeNew(sizeof(struct gatt_service_stru));
    service->cbk = app_gatts_def_service_cbk;

    // ===== service uuid =====

    att_sys_uuid16_to_u128(Service_def2_Uuid0,&(service->uuid),
APP_ATT_UUID_CE00_SERVICE);

    // ===== uuid =====

    att_sys_uuid16_to_u128(Characteristic_RXTX_CE01_Uuid,&uuid,
APP_ATT_UUID_RXTX_CE01);

    character = gatt_add_characteristic(service, val, 2, &uuid, ATT_WRITE);

    gatt_add_descriptor_ccc(character, ATT_CCCMASK_NOTIFICATION,
ATT_READ|ATT_WRITE);

    gatt_server_reg_tree(service);
}

/*****
```

函数名 : app_server_reg_service

功能描述 : 注册蓝牙服务

输入参数 : void

输出参数 : 无

```
*****/

void app_server_reg_service(void)
{
```

```
#if SDK_STD_IOT_SURPORT

// 【1】 注册 iot 服务

    app_gatts_register_iot_service();

// 【2】 保存最后一个服务 uuid

    s_last_service = get_iot_uuid16();

#endif


#if SDK_DEF_IOT_SURPORT

// 【3】 注册自定义服务 1

    app_gatts_register_def1_service();

// 【4】 注册自定义服务 2

    app_gatts_register_def2_service();

// 【5】 保存最后一个服务 uuid

    s_last_service = get_def2_uuid16();

#endif


#if SDK_OTA_SURPORT

// 【6】 注册 ota 服务

    ota_service_register();

// 【7】 保存最后一个服务 uuid

    s_last_service = APP_ATT_UUID_OTA_SERVICE;

#endif


#if SDK_BAT_SURPORT

// 【6】 注册电池电量服务

    ble_gatts_register_bat_service();

// 【7】 保存最后一个服务 uuid

    s_last_service = GATT_UUID_BATTERY_SERVICE;

#endif

}
```

18.11.2 保存服务中的所有句柄，并释放服务实例

```
/*****
```

函 数 名 : save_ae_def_iot_handle

功能描述 : 提取服务 2 句柄

输入参数 : struct gatt_service_stru *service

输出参数 : 无

*****/

```
uint8_t save_ae_def_iot_handle(struct gatt_service_stru *service)
{
    uint8_t save_ok = 0;

    struct gatt_characteristic_stru *character;

    if(NULL == def_iot){
        def_iot = NEW(sizeof(app_gatt_def_iot_service_data_stru));
        memset(def_iot, 0, sizeof(app_gatt_def_iot_service_data_stru));

        def_iot->tx_mode    = APP_IOT_MODE_NORMAL;
        def_iot->tx_credit   = APP_GATT_MAX_TX_CREDITS;
        def_iot->min_att_mtu = APP_ATT_DEFAULT_MTU;
    }

    character=                                app_gatt_find_characteristic_by_uuid128(service,
Characteristic_RXTX_CE01_Uuid);

    def_iot->ce01_tx_value_handle = character->value_hdl;

    dbg_block_printf("ce01_tx_value_handle %d \n",def_iot->ce01_tx_value_handle);

    return 1;
}
```

*****/

函 数 名 : save_ce_def_iot_handle

功能描述 : 提取服务 1 句柄

输入参数 : struct gatt_service_stru *service

输出参数 : 无

*****/

```
uint8_t save_ce_def_iot_handle(struct gatt_service_stru *service)
{
    struct gatt_characteristic_stru *character;
```

// 【1】 申请句柄存储实例

```
if(NULL == def_iot){  
    def_iot = NEW(sizeof(app_gatt_iot_service_data_stru));  
    memset(def_iot, 0, sizeof(app_gatt_iot_service_data_stru));  
    def_iot->tx_mode = APP_IOT_MODE_NORMAL;  
    def_iot->tx_credit= APP_GATT_MAX_TX_CREDITS;  
    def_iot->min_att_mtu = APP_ATT_DEFAULT_MTU;  
}
```

// 【2】 根据 UUID 查找 character

```
character= app_gatt_find_characteristic_by_uuid128(service,  
Characteristic_TX_AE02_Uuid);
```

// 【3】 提取 character 中的 value 句柄

```
def_iot->ae02_tx_value_handle = character->value_hdl;  
dbg_block_printf("ae02_tx_value_handle  %d \n",def_iot->ae02_tx_value_handle);
```

// 【4】 查找下一个 character

```
character= app_gatt_find_characteristic_by_uuid128(service,  
Characteristic_RX_AE03_Uuid);
```

```
def_iot->ae03_rx_value_handle = character->value_hdl;  
dbg_block_printf("ae03_tx_value_handle  %d \n",def_iot->ae03_rx_value_handle);
```

```
character= app_gatt_find_characteristic_by_uuid128(service,  
Characteristic_TX_AE04_Uuid);
```

```
def_iot->ae04_tx_value_handle = character->value_hdl;  
dbg_block_printf("ae04_tx_value_handle  %d \n",def_iot->ae04_tx_value_handle);
```

```
character= app_gatt_find_characteristic_by_uuid128(service, Characteristic05_Uuid);  
def_iot->ae05_indcate_value_handle = character->value_hdl;  
dbg_block_printf("ae05_tx_value_handle  %d \n",def_iot->ae05_indcate_value_handle);
```

```
character= app_gatt_find_characteristic_by_uuid128(service,  
Characteristic_RXTX_AE10_Uuid);
```

```
def_iot->ae10_tx_value_handle = character->value_hdl;
```

```
    dbg_block_printf("ae10_tx_value_handle %d \n",def_iot->ae10_tx_value_handle);

    return 1;
}
```

```
/******
```

函 数 名 : app_gatts_save_service

功能描述 : 内存转换, service 消耗内存较大, 将关键信息存储后, 销毁 service 保存简体
service

输入参数 : struct gatt_service_stru *service

输出参数 : 无

返 回 值 :

```
*****/
```

```
void app_gatts_save_service(struct gatt_service_stru *service)
```

```
{
```

```
// 【1】申请简化服务实例
```

```
    app_gatt_service_inst_stru *item = List_NodeNew(sizeof(app_gatt_service_inst_stru));
```

```
    //app_gatt_dbg_services(service);
```

```
// 【2】保存服务句柄
```

```
    item->service_handle = service->hdl;
```

```
// 【3】保存服务 UUID 16
```

```
    item->service_uuid = att_uuid_get_u2(&service->uuid);
```

```
// 【4】保存服务树下的句柄
```

```
    switch (item->service_uuid)
```

```
    {
```

```
        case GATT_UUID_DEVICE_INFORMATION:
```

```
            dbg_block_printf("device information \n");
```

```
            break;
```

```
        case GATT_UUID_GENERIC_ATTRIBUTE:
```

```
            dbg_block_printf("device attribute \n");
```

```
            break;
```

```
case GATT_UUID_GENERIC_ACCESS:

    dbg_block_printf("generic access\n");

    save_generic_handle(service);

    break;

case APP_ATT_UUID_IOT_SERVICE:

    dbg_block_printf("iot service \n");

#ifdef SDK_STD_IOT_SURPORT

    save_iot_handle(service);

#endif

    break;

case APP_ATT_UUID_AE00_SERVICE:

#ifdef SDK_DEF_IOT_SURPORT

    save_ae_def_iot_handle(service);

#endif

    break;

case APP_ATT_UUID_CE00_SERVICE:

#ifdef SDK_DEF_IOT_SURPORT

    save_ce_def_iot_handle(service);

#endif

    break;

case APP_ATT_UUID_OTA_SERVICE:

    dbg_block_printf("ota service \n");

#ifdef SDK_OTA_SURPORT

    save_ota_handle(service);

#endif

    break;

case GATT_UUID_LINK_LOSS:

case GATT_UUID_IMMEDIATE_ALERT:

    dbg_block_printf("immediate alert \n");

    save_link_lost_handle(service);

    break;

case GATT_UUID_BATTERY_SERVICE:

    dbg_block_printf("batter service \n");
```

```
        save_battery_handle(service);

        break;

    default:

        dbg_block_printf("default service \n");

        save_iot_handle(service);

        break;

    }

    item->service_data = NULL;

// 【5】添加服务实例至链表

    List_AddTail(APP_GATT_SERVICE_LIST, item);

    if (s_last_service == item->service_uuid) {

// 【6】所有服务添加完成，调用 app_server_register_cbk_ready

        app_gap_gatt_register_services_cfm(NULL, 0, NULL);

    }

}
```

18.12、蓝牙发送

18.12.1 通知方式发送

/**

函 数 名 : app_gatts_notify_data_ind

功能描述 : 通知数据

输入参数 : UINT16 tx_hdl

uint8_t *value

uint32_t value_len

输出参数 : 无

**/

void app_gatts_notify_data_ind(UINT16 tx_hdl, uint8_t *value, uint32_t value_len)

{

// 【1】判断蓝牙连接状态，及参数有效性

if((is_connected() == 0) || value == NULL || value_len <= 0)

return;

// 【2】 分配动态内存，底层会释放

```
struct gatt_server_notify_indicate_stru *pkt = NEW(sizeof(struct gatt_server_notify_indicate_stru) +
value_len);
```

```
memset(pkt, 0, sizeof(struct gatt_server_notify_indicate_stru) + value_len);
```

// 【3】 完成数据拷贝

```
pkt->hdl = tx_hdl;
```

```
pkt->val.len= value_len;
```

```
memcpy(pkt->val.value, value, value_len);
```

// 【4】 通知发送

```
gatt_server_notify_indicate(pkt);
```

```
}
```

通知发送成功，会在 GATT_EV_HDLVALUE_NOTIFY_CONFIRM 事件中通知。

18.12.2 写数据方式发送

/******

函 数 名 : app_gatts_write_data_ind

功能描述 : 写数据

输入参数 : UINT16 tx_hdl

uint8_t *value

uint32_t value_len

uint8_t send_type eg. GATT_TASK_WRITE_WITHOUT_RESPONSE

输出参数 : 无

*****/

```
void app_gatts_write_data_ind(UINT16 tx_hdl,uint8_t *value, uint32_t value_len, uint8_t
send_type)
```

```
{
```

```
struct gatt_tl_connect_stru tlin;
```

```
struct att_handle_value_stru *val_in;
```

```
memset(&tlin,0,sizeof(struct gatt_tl_connect_stru));
```

// 【1】 判断蓝牙连接状态，及参数有效性

```
if(is_connected() == 0)
```

```
        return;

    tlin.cbk      = NULL;

    tlin.context  = NULL;

    tlin.require  = GATT_TLREQUIRE_LE;

    tlin.sec_mode = LE_GAP_SECURITY_MODE1_LEVEL1;//安全级别

    tlin.timeout = 0xFFFFFFFF; /* Disconnect manually(keep link), 0 for disconnect now, 10~60000
for idle timeout */

    //【2】分配动态内存，底层会释放

    val_in      = NEW(sizeof(struct att_handle_value_stru) + value_len);

    //【3】完成数据拷贝

    val_in->hdl   = tx_hdl;

    val_in->offset = 0;

    val_in->val.len = value_len;

    memcpy(val_in->val.value, value, value_len);

    //【4】数据写入发送

    gatt_write_value(&tlin, val_in, send_type);

}
```

18.13、蓝牙接收

18.13.1 蓝牙接收函数注册

```
/******

函 数 名  : app_register_le_rcv_handler

功能描述  : 注册蓝牙接收到的数据接口

输入参数  : le_recive_data_handler handler

输出参数  : 无

返 回 值  :

*****/

void app_register_le_rcv_handler(le_recive_data_handler handler)

{

    le_rcv_handler = handler;
```

```
}
```

18.13.2 蓝牙接收数据

```
/******
```

函 数 名 : app_gatts_recv_data_ind

功能描述 : 蓝牙接收数据

输入参数 : struct att_pend_ind_stru *req

输出参数 : 无

返 回 值 :

```
*****/
```

```
void app_gatts_recv_data_ind(struct att_pend_ind_stru *req)
```

```
{
    // 遇到意外情况 可以用 req 中辅助条件判断
    // req 中有辅助条件可加以判断
    // UINT8 index;
    // UINT8 op; /* etc, ATT_OP_READ_BLOB_REQ */
    // UINT8 mask; /* etc, ATT_PENDMASK_FINAL */
    // UINT8 errcode; /* etc: ATT_ERR_OK, for upper set error code */
    // UINT8 require; /* UPF45, for dual mode, etc, GATT_TLREQUIRE_LE */
    if(req == NULL)
        return;
    if(le_rcv_handler){
        le_rcv_handler(req->it.hdl, req->it.val.value, req->it.val.len);
    }
}
```

通过句柄来判断数据来源，进行处理。

19、蓝牙连接断开常见错误处理

HCI_STATUS_REMOTE_USER_TERMINATED_CONNECTION

远端用户断开连接。

HCI_STATUS_CONNECTION_TIMEOUT

连接超时，查看信号强度，信号强度过低，检查天线或者，校准频偏，一般参考值 30cm > -60dbm

HCI_STATUS_CONNECTION_LIMIT_EXCEEDED

WS8100 支持一个端点连接，出现该错误为逻辑错误，请检查代码逻辑

HCI_STATUS_UNSPECIFIED_ERROR

未知错误，连接建立过程收到了意外的数据，

1、链路干扰

2、连接建立之前，用法调用了发送数据接口

HCI_STATUS_CONNECTION_FAILED_TO_BE_ESTABLISHED

连接建立失败，对方信号丢失，对方关闭了蓝牙，或者 master 死机，会出现这个错误。

20、异常错误处理

1、烧录程序，通过串口打印广播包确定为正常运行，发现不了蓝牙

解决方法：用频偏校准工具，查看频偏及频率幅度，1、主频是否被噪声信号淹没，2、主频外是否有过高的谐波。

2、开启调试串口，串口没输出

解决方法：检查烧录文件是否正确，程序运行地址是否正确。

3、进入待机无法唤醒

解决方法：通过复位强行唤醒，并检查具备唤醒能力的 gpio 在进入待机前是否为高电平

4、Keil 无法下载程序

解决方法：查看是否因为程序进入了低功耗，又没开启 jTAG 唤醒，通过复位唤醒，用烧录调试工具连接 WS8100 后，再用 keil 下载程序，或者按住具备唤醒能力的按键。

21、晶振频偏校准

21.1、校准文件

校准 PC 客户端：BLE_RF_Test_V1.2.exe

校准固件 16M 晶振：BLE_RF_Test_16M.hex

校准固件 32M 晶振：BLE_RF_Test_32M.hex

控制文件：ws8100_access_config.cfg

分区文件：ws8100_part_config.cfg

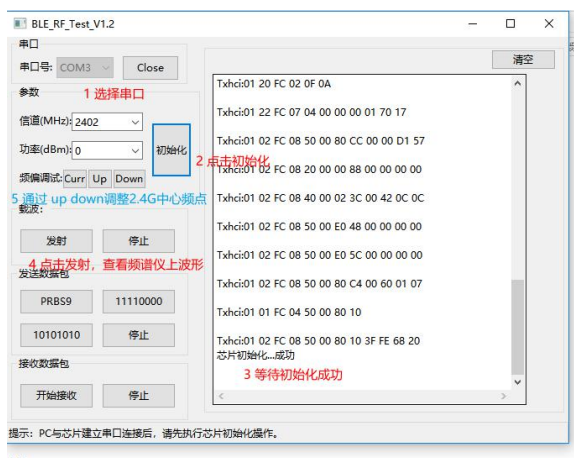
启动文件：WS8100_boot.hex

21.2、校准烧录方法

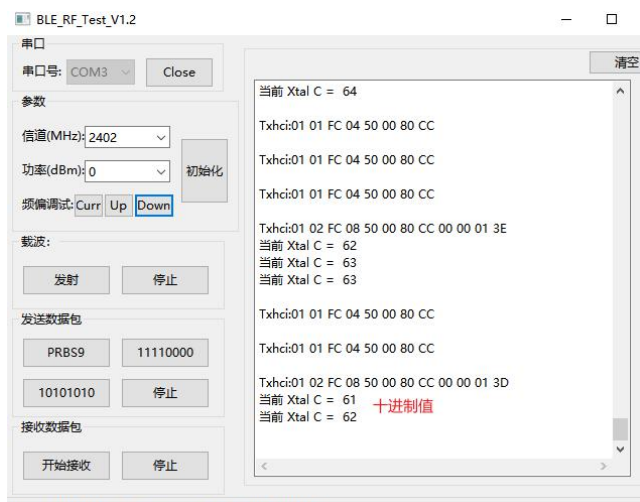
同调试烧录

21.3、校准流程

烧录校准固件后，按照下述流程操作。



21.4、校准结果



21.5、校准结果写入程序

```

00045: #define GENERAL_ADV_MIN_INTERVAL 1000 //
00046: #define GENERAL_ADV_MAX_INTERVAL 1600 //
00047:
00048: //=====晶振一般的都有误差，需要校准，默认64 有效范围 0~255=====
00049: #define RF_ADJUST 62
00050:
00051:
L07:
L08: *****
L09: void syscfg_configuration(void)
L10: {
L11: #if (USE_EXT_XTAL_16M)
L12: // 16M晶振
L13: syscfg_external_16M_xtal();
L14: //< 配置晶振匹配电容，外部无须电容
L15: syscfg_set_oscxtal_config(RF_ADJUST); 16M晶振位置
L16: #endif
L17:
L18: #if (USE_EXT_XTAL_32M)
L19: // 32M晶振
L20: syscfg_external_32M_xtal();
L21: //< 配置晶振匹配电容，外部无须电容
L22: syscfg_set_oscxtal_config(0x57); 32M晶振位置
L23: #endif
L24:
L25: #if (USE_EXT_XTAL_32K)
L26: // 外部32K晶振
L27: syscfg_external_32k_xtal();
L28: radio_set_32k_freq(32768);
L29: radio_set_32k_ppm(20);
L30: #endif
L31:
L32: #if (USE_INTER_RC_32K)
L33: // 内部32K晶振
    
```

22、OTA 升级

22.1、OTA 简介

WS8100 支持 OTA 升级，配置加密，文件校验功能。

OTA 升级需要 boot 支持，

经过验证的 OTA 文件为

Boot 文件：WS8100_ble_boot_v1.3.hex

分区文件：ws8100_ble_part_config_v1.3.cfg

控制文件：ws8100_ble_access_config_v1.3.cfg

应用程序：ws8100_ble_app_ws_V1.30_Project.hex

ota 要求打包版本高于 sdk 中版本，如 sdk 中版本为 1.30 那么制作 ota 固件版本应该为 131 以上。

22.2、OTA 升级开启

```

25:
26: //=====OTA升级功能=====
27: #define SDK_OTA_SUPPORT 1
28:
    
```

在 app_config.h 文件中定义

```

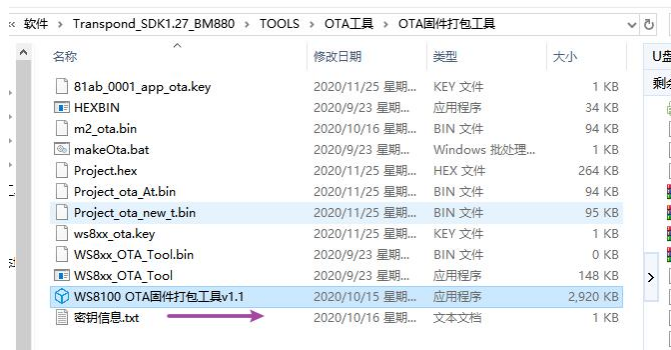
//=====OTA 升级功能=====

#define SDK_OTA_SUPPORT 1
    
```

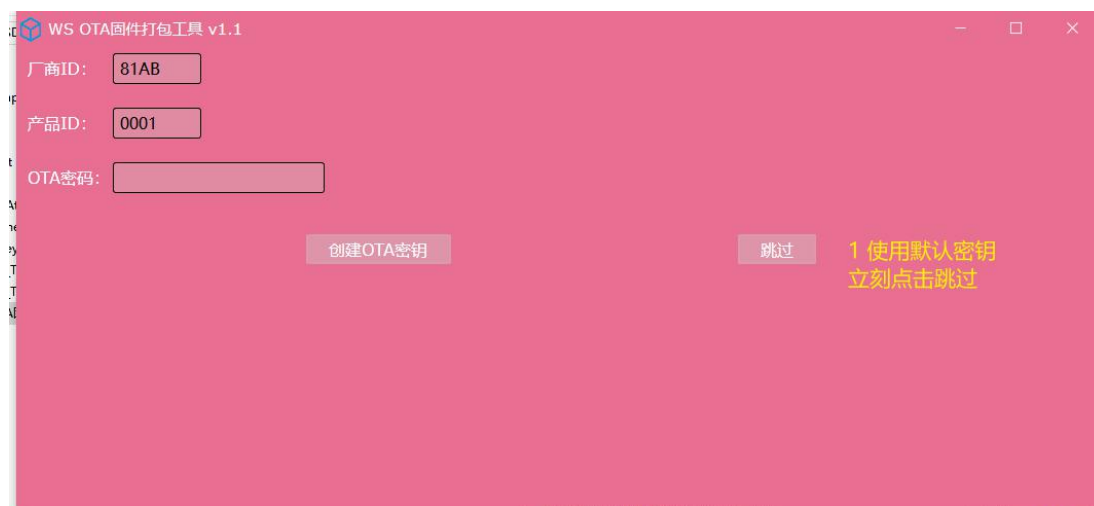
22.3、新版本升级方式

22.3.1、创建固件

打开 WS8100 OTA 固件打包工具 v1.1.exe



1、使用默认密钥，点击跳过



2、使用默认密钥，选择 OTA 密钥，选择 ws8xx_ota.key 文件，选择需要转换的 hex 固件，生成固件。



3、拷贝，81ab_0001_app_ota.key 及生成的 x_ota.bin 文件到手机指定目录。

22.3.2、升级操作

1、安装 WS8100_OTA.apk，如果已经安装 ws8100_ota_prev.apk 先移除掉，再安装，否则会提示失败。

2、选择刚才拷贝好的 x_ota.bin 文件，密钥 81ab_0001_app_ota.key 需要和 x_ota.bin 文件放同一目录，密钥生成，按照界面提示操作就可以了。



22.4、密钥创建



22.5、固件 OTA 密钥配置

在 ota.c 文件中

```

App.h  Bie_adv_ui.h  Module_Cmd.c  Module_Cmd.h  Project.h  App_gap.h  App_server.c  Bie_get_100_ui.c
lash.h  Generic_ui.c  Ota_server.c  全部关闭
00009: #if SDK_OTA_SURPORT
00010:
00011: #define APP_VERSION          0
00012: #define STACK_VERSION        0
00013:
00014: typedef struct {
00015:     unsigned short company_id;
00016:     unsigned short product_id;
00017:     char password[12];
00018: } ws_ota_pid_key_t;
00019:
00020: //Key值自定义
00021: #define OTA_VENDOR_ID        0x81AB
00022: #define OTA_PRODUCT_ID        0x0001
00023:
00024: #define OTA_PRODUCT_PASS      "rs#58da+s1@0"
00025:
00026: ws_ota_pid_key_t Key = {OTA_VENDOR_ID, OTA_PRODUCT_ID, OTA_PRODUCT_PASS};
00027:
00028: uint16_t crc_init = 0x0000; //CRC初始值
00029: uint16_t ota_reboot = 0x0000;
    
```

用记事本中内容替换掉上述内容，否则无法升级。

22.5、OTA 编译固件版本

sdk_include.h 中

```
153: ^/  
154: #define SOFT_VERSION 0x0130  
155:
```

23、SDK 快速上手

23.1、程序下载运行

A、WS8100 第一次下载，需要烧录四个文件：

Boot 文件：WS8100_boot.hex

分区文件：ws8100_part_config.cfg

控制文件：ws8100_access_config.cfg

控制文件：Project.hex

B、烧录的时候，点击了全片擦除，也需要烧录上述四个文件

C、下载过了 Boot 文件、分区文件、控制文件，后面修改应用程序重新编译，只需要下载 Project.hex 文件

D、DownLoadTool 界面，勾选了才会下载，不勾选，不会下载。

23.2、程序修改

1、外设初始化位置，hw_config.c 中 void hardware_init(void)

2、蓝牙服务注册位置 app_server.c 中 void app_server_reg_service(void)

3、逻辑应用入口 app.c 中 app_server_register_cbk_ready(void)

4、程序配置位置 app_config.h 中

5、低功耗部分，用户无需修改，系统自适应，默认开启超低低功耗，低功耗部分开关在 app_config.h 文件中。

23.4、逻辑应用流程简介

1、WS8100 片内启动引导启动 boot 程序，boot 功能为烧录，ota 升级使用，boot 启动用户程序。

2、WS8100 蓝牙逻辑，

Step1 rom_lib_init()

Step2 host_init();

Step3 开启中断后，蓝牙从 app_gap_init_regiser_tl_cfm 处开始运行

完成蓝牙复位、基础信息初始化、服务注册

Step4 蓝牙事件注册

```
void app_gatt_init(void)
{
    app_gattc_init();//事件通知，如发送数据成功，
    app_gatts_register_services();//服务注册
}
```

Step5 用户蓝牙句柄保存

```
void app_server_reg_service(void)
    服务完成后，保存需要使用的 hdl
```

Step6 蓝牙广播

```
start_advertising();//启动广播
stop_advertising();//停止广播
```

Step7 蓝牙连接事件

```
void app_ble_connected_ind(struct hci_address_stru *addr)           //蓝牙连接
void app_cbk_ble_disconn_ind(struct hci_address_stru *addr, UINT8 reason) //蓝牙断开
```

蓝牙连接逻辑，蓝牙广播，master 扫描到广播，发起连接，蓝牙响应连接。因此连接之前要先广播，断开连接后，要恢复连接，也要广播。

Step8 蓝牙数据事件

1、注册蓝牙接收数据

```
app_register_le_rcv_handler(le_rcv_data);
//里面需要根据句柄来判断数据来源
```

2、发送数据

// 默认方式发送

```
void app_gatt_iod_notify_data(uint8_t *value, uint32_t value_len)
```

// 句柄方式发送

```
void app_gatts_notify_indicate_data_ind(UINT16 tx_hdl, uint8_t *value, uint32_t value_len)
```

3、WS8100 外设逻辑

WS8100 外设分为常开电源域和非常开电源域

处于超低功耗下，非常开电源域设备，需要工作在浅睡眠或者无睡眠模式下，常开电源域设备不受此限制。非常开电源域设备，工作完成，系统切换到超低功耗模式下。

4、WS8100 内存逻辑

WS8100 为适应市场低成本需求，做了 16K 非掉电保存区间，低功耗要求运行在 24K 掉电保存区域。

[应用举例：](#)

切换到浅睡眠或者无睡眠模式下，连续采集 10s 数据，可放在 16K 非掉电保存区间，数据处理完了，再切换到超低功耗睡眠模式。

5、WS8100 任务逻辑

- ①、区别于常见 main.c 中 while(1)循环中调用，ws8100 为事件驱动机制。
- ②、开启低功耗，用户代码不可放在 main.c 中 while(1)中，用户函数需要在定时器、gpio、蓝牙事件、软定时器等中断回调函数中。
- ③、蓝牙回调函数中，不能有影响时序的较重的任务函数，较重任务需要用 FsmEventCx 等方式调用。

24、WS8100 AT 指令

WS8100 支持 AT 指令

24.1、透传模块工作方式

- 1、连接上以后，蓝牙接收到数据自动转发给串口
- 2、透传模块 IO 说明

引脚序号	引脚定义	引脚类型	说明
1	3VC	P	模块电源正极 1.8V-3.6V， 典型 3.3V
2	TCK	J	JTAG TCK
3	TMS	J	JTAG TMS
4	RST	R	模块运行复位,低电平复位， 内置看门狗。 对于极低功耗要求，完成复位后，应该将控制 io 设置为悬浮输入，降低功耗。
5	PC1 UART_CTSN	O	模块信号输入，模块串口发送允许 0: MCU 可接受来自模块的数据 1: MCU 不可接受来自模块的数据
6	PC0 UART_RTSN	I	模块信号输出，模块串口接收允许 0: 通知用户 MCU，此时模块串口可接收数据 1: 通知用户 MCU，此时模块串口不可接收数据
7	PD2	O	模块串口数据发送端

	UART_RX		
8	PD3 UART_TX	I	模块串口数据接收端
9	PD1 READY	O	模块信号输出，模块初始化状态指示 1: 模块就绪。 0: 模块未就绪。
10	GND		模块电源地
11	PA3 BT_EN	I	模块信号输入，模块蓝牙使能 0: 模块开始广播，直到有设备与之连接 1: 关闭蓝牙，如有连接，则断开，且关闭广播
12	PA2 UART_EN		1: 串口准备接收数据 0: 串口空闲，降低功耗
13	PA1 MODE_CTRL	I	模块信号输入，串口模式切换 1: 串口为指令模式 0: 串口为透传模式
14	PA0 STANDBYE	I	1: 进入待机。 0: 工作模式。 【极低功耗要求备注：输出 2 秒高电平后，设置 IO 为输入下拉，否则会增加功耗， 该操作带检测信号：生效后 PD1 为低电平】
15	PB4 WAKE_UP	I	1: 模块唤醒 0: 进入待机前要求为低电平 【极低功耗要求备注：输出 2 秒高电平后，设置 IO 为输入下拉，否则会增加功耗 该操作带检测信号：生效后 PD1 为高电平】
16	PB3	I	未定义 低电平/DEBUG_UART_TX
17	PB2	I	未定义 低电平/DEBUG_UART_RX
18	PB1	O	//方案 1【默认方案】

19	PB0	O	B1 B0 分别为 BIT1 BIT0 00: 停止广播 01: 正在广播 02: 完成连接 03: 连接断开
20	GND	P	模块地

备注：上述功能针对通用透传。

3、简单测试接线

PA1：命令使能，向模块发送命令，该引脚需要拉高

PA2：串口使能，向串口写数据，该引脚需要拉高。

PD2：RX 接串口模块 TX

PD3：TX 接串口模块 RX

24.2、AT 指令集格式

AT + CMD:param\r\n 或 AT + CMD\r\n

如 版本查询

ASC 码 指令 AT+VERS\r\n

十六进制指令 41 54 2b 56 45 52 53 0D 0A

24.3、AT 测试指令

位于 doc 目录下有 sscom

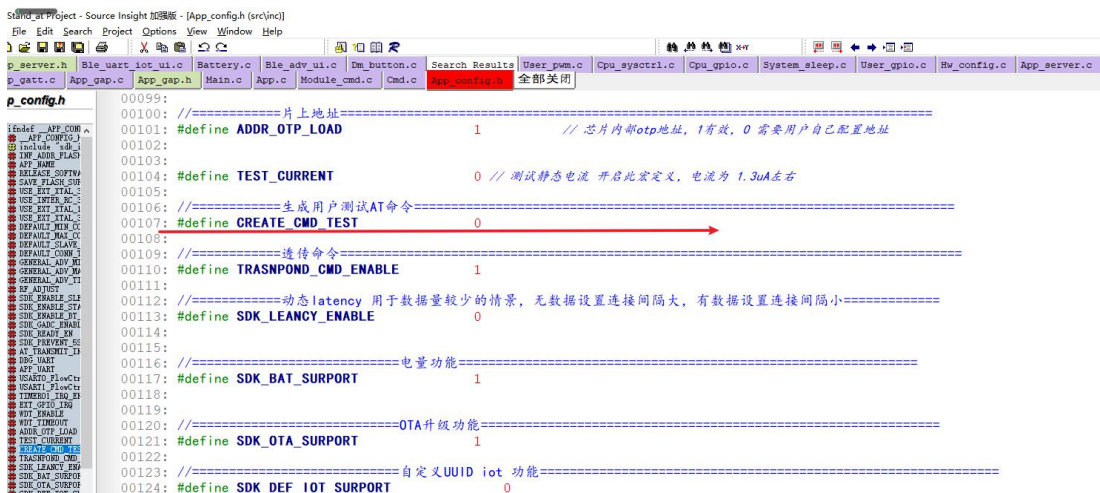
该工具默认配置文件包含 AT 指令配置

默认波特率 115200



24.3、AT 测试指令生成

WS8100 SDK 代码中已经包含了生成测试命令代码



```

120:  nvic_configuration();          /* 配置
121:  radio_update_32k_freq_and_ppm(); /* 初始
122:
123:  #if TEST_CURRENT
124:
125:  #else
126:
127:  #if (USE_INTER_RC_32K)
128:      hwadjust_calc_32krc_once(); /* 进行
129:      radio_32k_auto_cali();      /* 打开
130:  #endif
131:
132:  #if CREATE_CMD_TEST
133:      test_main();
134:  #endif
135:
136:  /*打开温度自动校准默认校准周期2s*/
137:  radio_temperature_auto_cali();
138:
139:  #endif
    
```

24.4、AT 测试指令使用

将生成指令拷贝到 sscom51.ini 文件中

```
00136:      char i;
00137:  #if CMD_TEST
00138:  //      for(i=0;i<sizeof(test);i++){
00139:  //          dbg_block_printf("%x ",test[i]);
00140:  //      }
00141:  //      while(1);
00142:  #endif
00143:
00144:      dbg_block_printf ("生成 sscom5.13.1.exe 工具的配置文件!\n");
00145:      dbg_block_printf ("将打印出来的信息复制到 sscom51.ini文件中 替换掉N1XX 最大支持100条命令!\n");
00146:      dbg_block_printf ("请按照该模板添加自己要的数据到上述数组中, 进行AT指令测试!\n");
00147:      dbg_block_printf ("AT 查询命令 \n");
00148:      for(index=0;index<sizeof(cmd_get)/sizeof(cmd_get[0]);index++)
00149:      {
00150:          dbg_block_printf("Nd=%d,%s,1000 \n",n,0,cmd_get[index].info);
00151:          dbg_block_printf("Nd=H," ,ncmd+1);
00152:
00153:          ptr = buf;
00154:          while(*ptr){
00155:              dbg_block_printf("%02x ",*ptr);
00156:              ptr ++;
00157:          }
00158:
00159:          ptr = cmd_get[index].cmd;
00160:          while(*ptr){
00161:              dbg_block_printf("%02X ",*ptr);
00162:              ptr ++;
00163:          }
00164:      }
```

24.5、WS8100 透传模块 SDK 宏定义配置简介

宏定义文件 app_config.h

```
// 【1】 用户 user2 分区 从 0x7A000~0x7F000,
```

```
// INF_ADDR_FLASH 用于保存配置文件信息
```

```
#define INF_ADDR_FLASH (0x0007a000) //flash 读写地址
```

```
// 【2】 默认 APP 应用名称,
```

```
//          优先原则, 优先使用 AT 指令设置的蓝牙名称, 次优先量产工具烧录的蓝牙名称, 前两者找不到有效名称, 使用 APP_NAME
```

```
#define APP_NAME "WS8100_BLE5.0"
```

```
// 【3】 =====配置文件掉电存储=====
```

```
#define SAVE_FLASH_SUPPORT 1
```

```
// 【4】 =====32M/16M 晶振修改此处==时钟选择 =====
```

```
#define USE_EXT_XTAL_32K 0 ///use external 32.768k xtal.
```

```
#define USE_INTER_RC_32K 1 ///use internal 32k rc.
```

```
#define USE_EXT_XTAL_16M      1    ///use external 16M xtal

#define USE_EXT_XTAL_32M      0    ///use external 32M xtal


// 【5】=====默认连接间隔=====单位 1.25ms 数值范围 6~3200=====

// 1600 20uA

#define                      DEFAULT_MIN_CONN_INTERVAL                      64 //800

#define                      DEFAULT_MAX_CONN_INTERVAL                      64 //800


// 【6】=====默认跳过连接信号序列 单位 1.25ms 0~499 =====

//=====很多手机不支持，不建议使用，特殊情况才用的，潜伏 17.5ms=====

#define DEFAULT_SLAVE_LATENCY      0


// 【7】=====默认连接超时时间=====单位 10ms 数值范围 1~1024

// 用户可以通过心跳包来检测连接

#define DEFAULT_CONN_TIMEOUT  600    // 单位 10ms 10s 最大 3200


// 【8】=====默认广播间隔===== 广播间隔单位 0.625ms 数值范围 6~16384

//                                  受限于连接间隔，可连接广播 要求 20~3999ms 之间

// 3000 24uA

#define GENERAL_ADV_MIN_INTERVAL      160

#define GENERAL_ADV_MAX_INTERVAL      160


// 【9】bt_start_advertising 启动广播，广播超时后，关闭广播，

#define GENERAL_ADV_TIME_OUT          0    // 0 表示无超时限制，


//【10】=====晶振一般的都有误差，需要校准，默认 64 有效范围 0~255=====

// layout 完成后，没有工具的情况下，和原厂工程师联系，校准频偏

#define RF_ADJUST                      64


//【11】=====低功耗选择，不在意功耗，关闭低功耗，发挥芯片的高性能=====

#define SDK_ENABLE_SLEEP    DEEP_SLEEP //DEEP_SLEEP //低功耗使能
```

```
// 【12】=====待机选择=====
#define SDK_ENABLE_STANDBYE          1    //待机使能

// 【13】=====蓝牙开关，允许 PA3 蓝牙使能=====
#define SDK_ENABLE_BT_SWT            1        //允许 PA3 控制蓝牙使能

// 【14】=====电量检测=====
#define SDK_GADC_ENABLE                0        //adc 检测使能 电量检测代码 battery

// 【15】=====READY 输出=====
#define SDK_READY_EN                  1

// 【16】=====调试串口=====
#define DBG_UART                      ws_uart1    //可选 WS_UART0 WS_UART1 0【关闭】

// 【17】=====透传串口=====
#define APP_UART                      ws_uart1

// 【18】=====串口方式=====
#define USART0_FlowCtrl_EN            0    //不开流控
#define USART1_FlowCtrl_EN            0    //不开流控

// 【19】=====定时器测试=====
#define TIMER01_IRQ_ENABLE            1        //定时器使能, 定时器发送数据 透传必须开启

// 【20】=====GPIO 中断=====
#define EXT_GPIO_IRQ                  1    // GPIO 中断使能

// 【21】=====看门狗=====
#define WDT_ENABLE                    0
```

```
/* wdt will reset system if wait this time don't feed dog*/

#define WDT_TIMEOUT                12000    // 12s watchdog timeout 低功耗建议设为 3

// 【22】=====片上地址=====

#define ADDR_OTP_LOAD              1  // 芯片内部 otp 地址, 1 有效, 0 需要用户自己配置地址

#define TEST_CURRENT                0  // 测试静态电流 开启此宏定义, 电流为 1.3uA 左右

// 【23】=====生成用户测试 AT 命令=====

#define CREATE_CMD_TEST            0

// 【24】=====透传命令=====

#define TRASNPOND_CMD_ENABLE        1

// 【25】==动态 latency 用于数据量较少的情景, 无数据设置连接间隔大, 有数据设置连接间隔小==

#define SDK_LEANCY_ENABLE           0

// 【26】=====电量功能=====

#define SDK_BAT_SURPORT            1

// 【27】=====OTA 升级功能=====

#define SDK_OTA_SURPORT            1

// 【28】=====自定义 UUID iot 功能=====

#define SDK_DEF_IOT_SURPORT        0

// 【29】=====标准 iot 功能=====

#define SDK_STD_IOT_SURPORT        1

// 【30】=====转发完整串口包 等串口发送完, 延时 2ms 启动蓝牙数据发送=====

#define UART_FULL_PACK_ENABLE      1
```

```
//===== 自动处理宏定义部分 =====

//=====DEBUG_UART START=====

#if TEST_CURRENT

#undef DBG_UART

#undef APP_UART

#define DBG_UART 0

#define APP_UART 0

#endif

#if ((is_uart_dev(DBG_UART) && is_uart_dev(APP_UART)) && (DBG_UART != APP_UART))

#define UART_USE_NUM          2

#define UART0_IDX             0

#define UART1_IDX             1

#elif (is_uart_dev(DBG_UART) || is_uart_dev(APP_UART))

#define UART_USE_NUM          1

#define UART0_IDX             0

#define UART1_IDX             0

#else

#define UART_USE_NUM          0

#endif

//【31】=====开启调试串口，开关默认关闭，可以 AT 指令开启=====

#define LOG_DEFAULT_ON          0    // 1 为默认开启打印信息 0 为默认关闭打印信息

#define LOG_SWITCH              1    // 日志控制开关，为 1 的时候，日志开关命令有效, 允许命令切换

#define SPI_USE_NUM            2

#define APP_SPI                 ws_spi0

#define SPI0_IDX                0
```

```
#define SPI1_IDX 1  
  
#endif /// __APP_CONFIG_H__
```