深圳市启明云端科技有限公司

WT5105

SDK 应用指南

Version 0.2

Wireless-Tag Technology Co., Limited
2019/01/24

版本控制信息

版本/状态	作者	参与者	起止日期	备注
V0.1	赵红梅		03/30/2018	文档初稿
V0.2	赵红梅		01/24/2019	基于 WT51XX_SDK_1.1.8 的文档更新

目录

图表

1 简介

本文档用于说明如何使用 WT5105 SDK 进行应用的开发，它能够帮助您了解和理解 SDK 提供的组件，样例的使用方法，并且帮助您如何从提供的样例开始进行 BLE 产品的固件开发。

SDK 提供的样例仅仅作为设计的参考，对于产品的固件开发，请务必使用自己设计的固件！

2 快速开始

SDK 提供一组示例程序，他们可以直接在开发板上正常运行，您可以以这些实例程序为参考，并开始自己的应用开发。

通过运行这些预编译示例，并结合智能手机通过 BLE 测试程序进行交互实验，您能够快速了解开发板和样例所提供的具体功能。

2.1 SDK 目录结构

```
WT51XXSDK
├─components                ;SDK组件，包括BLE API，GATT Profiles，芯片硬件驱动和其他软件组件
├─example                   ;例程
│  ├─ble_central            ;
│  ├─ble_peripheral         ;
│  │  ├─alternate_iBeacon   ;alternate iBeacon样例
│  │  ├─ancs                ;苹果消息中心（ANCS）例程
│  │  ├─bleI2C_RawPass      ;I2C透传例程
│  │  ├─bleSmartPeripheral  ;综合Peripheral例程
│  │  ├─bleUart-RawPass     ;UART透传例程
│  │  ├─eddystone           ;eddystone例程
│  │  ├─HIDKeyboard         ;HID例程
│  │  ├─hrs                 ;Heart rate profile演示例程
│  │  ├─iBeacon             ;iBeacon例程
│  │  ├─otaDemo             ;最基本的OTA演示例程
│  │  ├─pwmLight            ;通过BLE命令控制PWM进行LED亮度调整的例程
│  │  ├─RawAdv              ;简单的广播例程，用于无线胎压监测一类的应用
│  │  ├─Sensor_Broadcast    ;
│  │  ├─wrist               ;综合样例，用于运动手环一类的应用
│  │  ├─wrist_aptm          ;基于综合样例，通过AP Timer+OSAL Timer实现实时性较高的时钟
│  │  └─XIPDemo             ;部分实时性要求较低的代码直接运行于内部flash的例程
│  ├─OTA                    ;
│  │  ├─OTA_internal_flash  ;OTA bootloader
│  │  └─OTA_upgrade_2ndboot ;特殊应用，用于升级OTA bootloader自身
│  └─peripheral             ;
│     ├─adc                 ;ADC驱动应用样例
│     ├─ap_timer            ;AP Timer驱动应用样例
│     ├─fs                  ;文件系统样例
│     ├─gpio                ;GPIO演示例程
│     ├─kscan               ;4x4按键演示例程
│     ├─lcd_ST7789VW        ;240x240 TFT彩屏演示例程
│     ├─pwm                 ;PWM演示例程
│     ├─qdec                ;QDEC演示例程
│     ├─spiflash            ;外部SPI演示例程
│     ├─voice               ;语音采集演示例程
│     ├─voice_sbc           ;SBC格式语音采集编码演示例程
│     └─watchdog            ;看门狗演示例程
├─lib                       ;lib文件和.h文件，包括蓝牙协议栈更新，字库
│  └─font                   ;字库的资源文件和升级文件
└─misc                      ;ROM symble table和跳转表
```

2.2 开发板和预编译样例

我们提供三种开发板，其中贴 32pin 封装的开发板，用于用户扩展功能 验证。48pin 封装芯片的开发板，该开发板集成了加速度传感器， 64*128 分辨率的 OLED 屏幕，心率传感器，可调亮度的 LED 灯，键盘，SPI FFlash 等等外设，可以用于比较复杂应用的验证，以

及演示。还有一种更小的贴 32pin 封装芯片的开发 板为 mesh 开发板。

在 SDK 所在的目录中，，example 目录包含了样例项目，其中每个样例有一个 bin 目录，其中的 hex 文件为可执行的程序固件，初次使用本 SDK 的用户可以直接烧录该文件进行实验。程序的烧录方法请参考章节：调试和烧写

2.3 安装开发环境

开发环境安装请按照以下步骤进行：

- 拷贝 SDK 至工作目录。
- 安装 MDK Keil5 for ARM 开发环境。
- 通过 MDK 打开 SDK 目录中的样例的项目文件即可对项目进行编译调试等操作。

2.4 编译和运行样例

- 使用 WT51-RFtools 工具删除开发板已经烧录的固件（WT51-RFtools 工具使用方法请参考工具 使用 指南：<WT51-RFtools.pdf>）。
- 从浏览器的 SDK 安装目录→example→ble_perpheral 选择一个样例，比如 iBeacon，打开 MDK 项目文件：

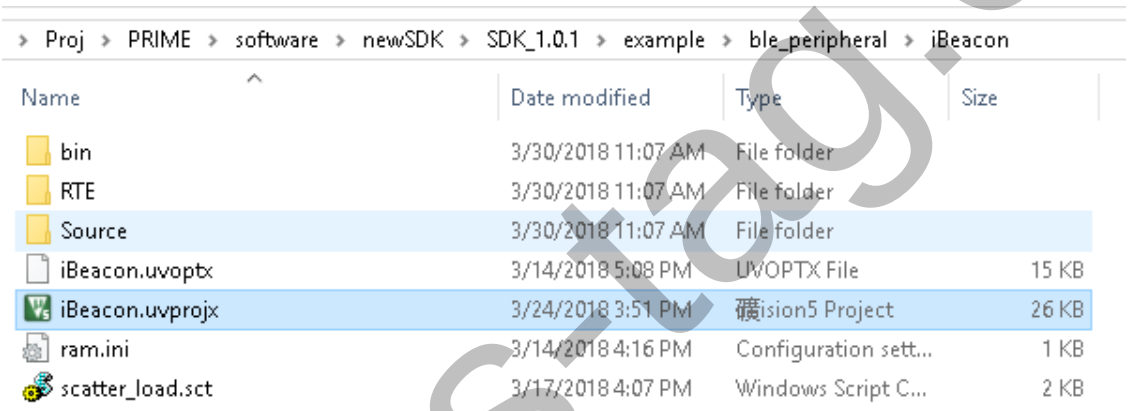

图 4

- 编译项目，然后单击调试按钮加载固件进行调试，然后单机运行按钮，运行程序。
- 此时固件程序已经在开发板上运行，可以通过手机的 BLE 工具进行交互。

2.5 调试和烧写

- 在 MDK 工具栏按钮，点击 Option for target 按钮 ，打开项目的 option 对话框。

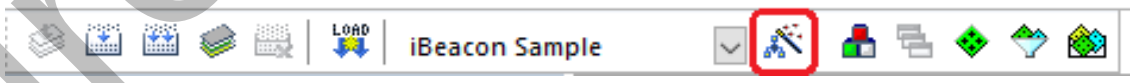

图 5

- 在 C/C++标签页的 Preprocessor Symbols→Define 里面，开发者可以改变对应的预编译宏：
  - CFG_SLEEP_MODE=PWR_MODE_SLEEP ：使能低功耗模式，固件程序执行过程中，会在空闲过程进入睡眠，睡眠之后调试器无法进行调试跟踪，断点也失效

- CFG_SLEEP_MODE=PWR_MOD_ENO_SLEEP ：关闭低功耗模式，固件程序执行过程中，处理器一直处于唤醒状态。
- DEBUG_INFO=1：使能调试信息，默认通过串口输出：P9(Tx),P10(Rx)
- DEBUG_INFO=0：关闭调试信息。

- 调试代码：
- 请使用 WT51-RFtools 工具擦除 Flash 中已烧录的固件，擦除之前请确认开发板的 TM 跳线连接高电平，并且按 Reset 键。
- 确认 Flash Erase 成功之后，TM 跳线连接低电平，按 Reset 键重启芯片
- 点击 MDK 工具栏的 Debug button，下载 Image 文件到 SRAM，进行在线 Debug。

- 固件烧写：
- 需要先擦除 Flash，然后进行烧写，烧写方法参考 WT51-RFtools 使用文档。

3 平台和驱动

3.1 平台简介

WT5105 的固件架构分为三部分，ROM 部分，OTA Bootloader，应用固件，其中 OTA Boot loader 为可选过程，下图为两种启动过程流程。

支持 OTA 模式启动：

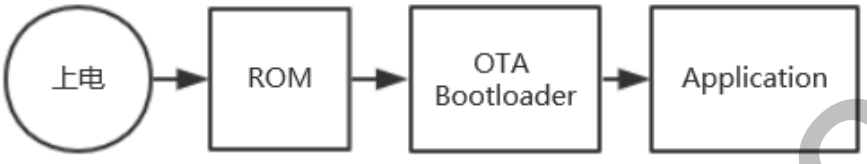

图 6

非 OTA 模式启动：

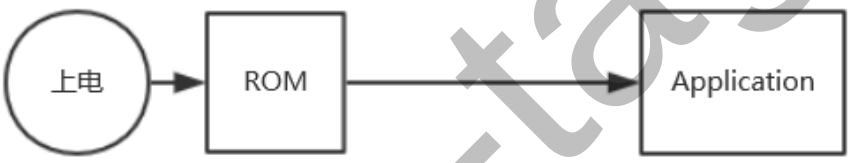

图 7

- ROM 部分：启动并引导-或者 Boot loader 或者 Application 和 BLE 协议栈 API，启动通过 TM 管脚的高低电平选择编程模式（高电平）还是正常启动模式（低电平）。
- OTA Boot loader：用于引导 Application 以及处理 OTA 升级。
- Application：应用代码，绝大部分二次开发工作都集中在 Application 部分。

3.2 软件框图

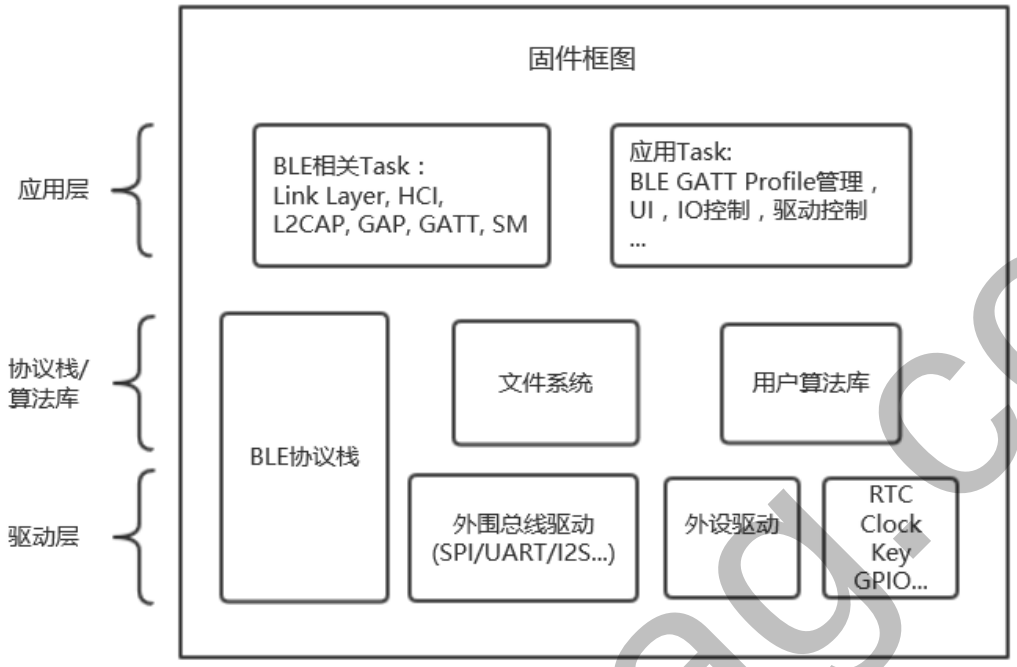

图 8

SDK 没有使用第三方的 RTOS，不过在应用层抽象了 Task 概念，对于 BLE 应用，以下 Task 是必须的，每个 Task 分别包括一个初始化函数和一个事件处理函数，具体说明请参考下表。

应用一般定义一到多个 Task，典型场景（例程）一般只用一个应用 Task。人机交互，外设控制，BLE 广播和连接的配置，GATT Profile 的加载等事务都在应用 Task 实现。任务以内以及任务之间可以通过 OSAL 提供的 API 进行交互和通信。

Tasks (Task 初始化&Task 事件响应函数)	说明
LL_Init() LL_ProcessEvent(uint8, uint16)	Link layer 初始化和对应的事件处理函数。
HCI_Init() HCI_ProcessEvent(uint8, uint16)	HCI 层初始化和对应的事件处理函数
L2CAP_Init() L2CAP_ProcessEvent(uint8, uint16)	L2CAP 初始化和对应的事件处理函数

GAP_Init() GAP_ProcessEvent(uint8, uint16)	GAP 初始化和对应的事件处理函数
GATT_Init() GATT_ProcessEvent(uint8, uint16)	GATT 初始化和对应的事件处理函数
SM_Init() SM_ProcessEvent(uint8, uint16)	SM（安全管理）初始化和对应的事件处理函数
GAPRole_Init() GAPRole_ProcessEvent(uint8, uint16)	GAP 配置初始化和对应的事件处理函数
GATTServApp_Init() GATTServApp_ProcessEvent(uint8, uint16)	GATT 服务初始化和对应的事件处理函数

如果需要支持 SMP 那么还需要加入 Bond Manager task：

GAPBondMgr_Init() GAPBondMgr_ProcessEvent(uint8, uint16)	用于对 SMP 的支持

3.2.1 OSAL

SDK 提供一组 API，用于系统层面的操作，包括消息机制，定时器，堆管理。

APIs

3.2.1.1 uint8 osal_set_event(uint8 task_id, uint16 event_flag)

用于向一个 Task 发送事件，之后 Task 的事件处理函数会响应该事件。

- 参数

类型	参数名	说明
uint8	task_id	任务 ID
uint16	event_flag	事件 ID，每个 Bit 位对应一个事件，一个 Task 能声明最多 16 个事件类型。

- 返回值

0	成功。
其他数值	参考<comdef.h>

3.2.1.2 uint8 osal_clear_event(uint8 task_id, uint16 event_flag)

清除一个事件。

- 参数

类型	参数名	说明
uint8	task_id	任务 ID
uint16	event_flag	事件 ID，每个 Bit 位对应一个事件，一个 Task 能声明最多 16 个事件类型。

- 返回值

0	成功。
其他数值	参考<comdef.h>

3.2.1.3 uint8 osal_msg_send(uint8 task_id, uint8 *msg_ptr)

向一个 Task 发送消息。

- 参数

类型	参数名	说明
uint8	task_id	任务 ID
Uint8*	msg_ptr	消息指针

- 返回值

0	成功。
其他数值	参考<comdef.h>

3.2.1.4 uint8 *osal_msg_receive(uint8 task_id)

接收消息，该函数应用在 Task 的事件处理函数里对应的事件为 SYS_EVENT_MSG (0x8000)。这个事件是专门预留给消息处理的。

收到事件 SYS_EVENT_MSG 之后再通过该函数获取消息指针，如果该消息的 buffer 是通过 osal_mem_alloc 分配的，那么使用之后需要通过 osal_mem_free 释放。

- 参数

类型	参数名	说明
uint8	task_id	任务 ID

- 返回值

uint8*	消息指针。
NULL	未收到消息

3.2.1.5 uint8 osal_start_timerEx(uint8 task_id, uint16 event_id, uint32 timeout_value)

开始一个应用 Timer，到达超时时间系统会向指定的 task 发送一个事件，该 timer 完成一次事件之后自动关闭，不再重发。

- 参数

类型	参数名	说明
uint8	task_id	Task ID。

uint16	event_id	事件 ID，需要在指定的 Task 声明
uint32	timeout_value	超时时间，单位毫秒。

- 返回值

SUCCESS	成功。
NO_TIMER_AVAIL	启动 timer 失败

3.2.1.6 uint8 osal_start_reload_timerEx(uint8 task_id, uint16 event_id, uint32 timeout_value)

开始一个应用 Timer，到达超时时间系统会向指定的 task 发送一个事件，该类型 timer 在开始之后会按照指定的时间间隔向 task 发送事件，直到函数 osal_stop_timerEx()停止该 timer。

- 参数

类型	参数名	说明
uint8	task_id	Task ID。
uint16	event_id	事件 ID，需要在指定的 Task 声明
uint32	timeout_value	超时时间，单位毫秒。

- 返回值

SUCCESS	成功。
NO_TIMER_AVAIL	启动 timer 失败

3.2.1.7 uint8 osal_stop_timerEx(uint8 task_id, uint16 event_id)

停止一个应用 Timer。

- 参数

类型	参数名	说明
uint8	task_id	Task ID。
uint16	event_id	事件 ID，需要在指定的 Task 声明

- 返回值

SUCCESS	成功。
INVALID_EVENT_ID	该 timer ID 并不存在（未启动）。

3.2.1.8 uint32 osal_get_timeoutEx(uint8 task_id, uint16 event_id)

获取一个正在运行 Timer 剩余的超时时间，时间单位为毫秒。

- 参数

类型	参数名	说明
uint8	task_id	Task ID。
uint16	event_id	事件 ID，需要在指定的 Task 声明

- 返回值

uint32	剩余的等待时间，如果该 timer 并没有启动，返回 0。

3.3 硬件驱动

3.3.1 概述

本章节介绍硬件模块驱动和外部总线驱动，硬件模块定义请参考章节 3.1.3.1，另外在实际硬件驱动以外还提供了虚拟模块，用于应用进行虚拟控制，关于虚拟控制请参用手环综合用例，其中有对于虚拟模块 MOD_USR0 的应用。

3.3.1.1 模块 ID

3.3.1.2 MODULE_e

模块 ID。

值	名称	说明
0	MOD_NONE MOD_SOFT_RESET MOD_CPU	通常情况下是无效设备。 MOD_SOFT_RESET 和 MOD_CPU 这两个别名暂时预留，并没有生效。
1	MOD_LOCKUP_RESET_EN	暂时无效。
2	MOD_WDT_RESET_EN	暂时无效。
3	MOD_DMA	DMA 模块。
4	MOD_AES	AES 模块。
5	MOD_TIMER	Timer 模块。
6	MOD_WDT	Watchdog 模块。
7	MOD_COM	暂时无效。
8	MOD_UART	UART 模块。
9	MOD_I2C0	I2C 总线 1。
10	MOD_I2C1	I2C 总线 2。
11	MOD_SPI0	SPI 总线 1。
12	MOD_SPI1	SPI 总线 2。
13	MOD_GPIO	GPIO 模块。
14	MOD_I2S	I2S 模块。
15	MOD_QDEC	QDEC（正交解码器）模块。
16	MOD_RNG	随机数模块
17	MOD_ADCC	ADC 模块
18	MOD_PWM	PWM 模块。
19	MOD_SPIF	内建 SPI Flash 模块。
20	MOD_VOC	VOC 模块。
31	MOD_KSCAN	Key Scan 模块
32	MOD_USR0	虚拟模块 0，预留给系统使用，用于管理中断优先级。

33~39	MOD_USR1~8	应用根据需要使用，可以管理应用的休眠，管理应用的唤醒和睡眠的附加操作。

3.3.2 Clock

Clock 模块主要提供系统时钟相关的配置，包括模块的时钟开关，32K 时钟源选择等操作。

3.3.2.1 枚举&宏

3.3.2.1.1 CLK32K_e

32K 时钟源选择。

CLK_32K_XTAL	选择外部晶振作为 32K 时钟源。
CLK_32K_RCOSC	选择内部 RC 作为时钟源。

3.3.2.2 数据结构

无。

3.3.2.3 APIs

3.3.2.3.1 void clk_gate_enable(MODULE_e module)

使能硬件模块的时钟。

- 参数

类型	参数名	说明
MODULE_e	module	硬件模块 ID。

- 返回值

无。

3.3.2.3.2 void clk_gate_disable(MODULE_e module)

关闭硬件模块的时钟。

- 参数

类型	参数名	说明
MODULE_e	module	硬件模块 ID。

- 返回值

无。

3.3.2.3.3 void clk_reset(MODULE_e module)

重置模块。

- 参数

类型	参数名	说明
MODULE_e	module	硬件模块 ID。

- 返回值

无。

3.3.2.3.4 uint32_t clk_hclk(void)

获取 HCLK 数值。

- 参数

无。

- 返回值

返回 HCLK 数值。

3.3.2.3.5 uint32_t clk_pclk(void)

获取 PCLK 数值。

- 参数

无。

- 返回值

返回 PCLK 数值。

3.3.2.3.6 void hal_rtc_clock_config(CLK32K_e clk32Mode)

配置 RTC 的时钟源。

- 参数

类型	参数名	说明
CLK32K_e	clk32Mode	RTC 的时钟源。

- 返回值

无。

3.3.2.3.7 uint32_t hal_systick(void)

获得系统 tick 值，tick 值为 32bit 无符号数，时间单位为 625 微秒。

- 参数

无

- 返回值

32bit 无符号系统 tick 值。

3.3.2.3.8 uint32_t hal_ms_intv(uint32_t tick)

和上一个时间快照之前的时间差，单位为毫秒，输入参数为系统 tick。

- 参数

类型	参数名	说明
uint32_t	tick	需要对比的 tick 值，在之前某个时间记录下的系统 tick。

- 返回值

毫秒为单位的时间差。

3.3.3 retention SRAM

休眠模式下，SRAM 可以根据需要配置为保持或者不保持数据，每块 SRAM 可以独立开关，如果配置为保持，休眠唤醒之后，RAM 数据能够保持，否则数据会丢失。

通常应用会根据实际 SRAM 的使用量配置 retention。一般在<main.c>的 hal_init 函数进行配置，请参考 API：hal_pwrmgr_RAM_retention(uint32_t sram)

WT5105 总共有 5 块可用的 SRAM，具体信息请参考下表：

RAM ID	Size	地址空间	说明
SRAM0	32K Byte	1FFF_0000~1FFF_7FFF	应用与协议栈公用，休眠模式下必选打开。
SRAM1	32K Byte	1FFF_8000~1FFF_FFFF	SRAM1 ~ SRAM4 可以根据应用选择关闭或者使能，建议不需要的情况下关闭，有助于降低功耗。
SRAM2	64K Byte	2000_0000~2000_FFFF	
SRAM3	8K Byte	2001_0000~2001_1FFF	
SRAM4	2K Byte	2001_2000~2001_27FF	

3.3.4 GPIO

提供 GPIO 相关的操作，包括 GPIO 配置，上下拉设置，GPIO 输入模式的事件驱动模型，GPIO 唤醒操作等。

3.3.4.1 枚举&宏

3.3.4.1.1 GPIO pin 定义

其中 GPIO_DUMMY 定义为虚拟 pin，一般作为无效 pin 使用。

```
typedef enum{
    GPIO_P00    =   0,      P0   =   0,
    GPIO_P01    =   1,      P1   =   1,
    GPIO_P02    =   2,      P2   =   2,
    GPIO_P03    =   3,      P3   =   3,
    GPIO_P04    =   4,      P4   =   4,
    GPIO_P05    =   5,      P5   =   5,
    GPIO_P06    =   6,      P6   =   6,
    GPIO_P07    =   7,      P7   =   7,
    TEST_MODE   =   8,      P8   =   8,
    GPIO_P09    =   9,      P9   =   9,
    GPIO_P10    =   10,     P10  =   10,
    GPIO_P11    =   11,     P11  =   11,    Analog_IO_0 = 11,
    GPIO_P12    =   12,     P12  =   12,    Analog_IO_1 = 12,
    GPIO_P13    =   13,     P13  =   13,    Analog_IO_2 = 13,
    GPIO_P14    =   14,     P14  =   14,    Analog_IO_3 = 14,
    GPIO_P15    =   15,     P15  =   15,    Analog_IO_4 = 15,
    GPIO_P16    =   16,     P16  =   16,    XTALI = 16,
    GPIO_P17    =   17,     P17  =   17,    XTALO = 17,
    GPIO_P18    =   18,     P18  =   18,    Analog_IO_7 = 18,
    GPIO_P19    =   19,     P19  =   19,    Analog_IO_8 = 19,
    GPIO_P20    =   20,     P20  =   20,    Analog_IO_9 = 20,
    GPIO_P21    =   21,     P21  =   21,
    GPIO_P22    =   22,     P22  =   22,
    GPIO_P23    =   23,     P23  =   23,
    GPIO_P24    =   24,     P24  =   24,
    GPIO_P25    =   25,     P25  =   25,
    GPIO_P26    =   26,     P26  =   26,
    GPIO_P27    =   27,     P27  =   27,
    GPIO_P28    =   28,     P28  =   28,
    GPIO_P29    =   29,     P29  =   29,
    GPIO_P30    =   30,     P30  =   30,
    GPIO_P31    =   31,     P31  =   31,
    GPIO_P32    =   32,     P32  =   32,
    GPIO_P33    =   33,     P33  =   33,
    GPIO_P34    =   34,     P34  =   34,
    GPIO_DUMMY  =   0xff,
}GPIO_Pin_e;
```

3.3.4.1.2 GPIO_ioe

GPIO 输入或者输出配置。

IE	配置为输入。
OEN	配置为输出

3.3.4.1.3 BitAction_e

IO pin 配置为 GPIO 模式或者配置为功能 pin。

Bit_DISABLE	配置为 GPIO 模式。
Bit_ENABLE	配置为功能 pin。

3.3.4.1.4 IO_Pull_Type_e

配置 pin 的上下拉模式。

FLOATING	配置为无上下拉，pin 悬空。
WEAK_PULL_UP	配置为弱上拉。
STRONG_PULL_UP	配置为强上拉。
PULL_DOWN	配置为下拉。

3.3.4.1.5 IO_Wakeup_Pol_e

配置 pin 的唤醒极性，上升沿唤醒或者下降沿唤醒。

POSEDGE	配置为上升沿唤醒。
NEGEDGE	配置为下降沿唤醒。

3.3.4.1.6 Fmux_Type_e

配置 pin 的功能设置。

3.3.4.2 数据结构

3.3.4.2.1 gpioin_Hdl_t

GPIO 输入模式中的回调函数原型。

3.3.4.3 APIs

3.3.4.3.1 int hal_gpio_init(void)

GPIO 模块初始化：初始化硬件，使能中断，配置中断优先级等等。

该函数需要在系统初始化时候设置，一般是在 hal_init()函数中调用，具体请参考例程。

- 参数

无。

- 返回值

PPlus_SUCCESS	初始化成功。
其他数值	参考<error.h>

3.3.4.3.2 void hal_gpio_wakeup_set (GPIO_Pin_e pin,IO_Wakeup_Pol_e type)

配置 GPIO 唤醒模式：上升沿唤醒或者下降沿唤醒。

- 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
IO_Wakeup_Pol_e	type	唤醒模式。

- 返回值

无。

3.3.4.3.3 void hal_gpio_pin_init(GPIO_Pin_e pin,GPIO_ioe type)

配置 GPIO 模式，输入或者输出。

- 参数

类型	参数名	说明
GPIO_Pin_e	pin	GPIO pin。
GPIO_ioe	type	配置 GPIO 为输入或者输出。

- 返回值

无。

3.3.4.3.4 void hal_gpio_write(GPIO_Pin_e pin, uint8_t en)

向某一个 GPIO 写 1 或者 0。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
uint8_t	en	0：写 0，其他值：写 1。

- 返回值

无。

3.3.4.3.5 uint32_t hal_gpio_read(GPIO_Pin_e pin)

读取某一个 GPIO 的值。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。

- 返回值

0：低电平， 1：高电平。

3.3.4.3.6 int hal_gpio_cfg_analog_io(GPIO_Pin_e pin, BitAction_e value)

配置 IO 为 analog 模式。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
BitAction_e	value	使能或者关闭 IO 的 analog 模式

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.4.3.7 int hal_gpio_pull_set(GPIO_Pin_e pin,IO_Pull_Type_e type);

设置 IO 的上下拉。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
Pull_Type_e	type	IO 上下拉设置。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.4.3.8 int hal_gpio_fmux_set(GPIO_Pin_e pin,Fmux_Type_e type)

配置 IO 的功能模式。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
Fmux_Type_e	type	IO 的功能模式

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.4.3.9 int hal_gpioin_register(GPIO_Pin_e pin, gpioin_Hdl_t posedgeHdl, gpioin_Hdl_t negedgeHdl)

注册 GPIO 的输入模式，该模式下支持中断和唤醒回调，包括上升沿回调和下降沿回调函数。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。
gpioin_Hdl_t	posedgeHdl	上升沿的回调函数，可以为 NULL。
gpioin_Hdl_t	negedgeHdl	下降沿的回调函数，可以为 NULL。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.4.3.10 int hal_gpioin_unregister(GPIO_Pin_e pin)

注销 GPIO 的输入模式，该模式下支持中断和唤醒回调，包括上升沿回调和下降沿回调函数，注销之后这些功能无效。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.4.3.11 int hal_gpioin_enable(GPIO_Pin_e pin)

对于已经注册为 gpioin 的 pin，如果该 pin 已经停用（参考 hal_gpioin_disable），通过该函数可以启用 gpioin 功能。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.4.3.12 int hal_gpioin_disable(GPIO_Pin_e pin)

对于已经注册为 gpioin 的 pin，如果该 pin 已经使能，通过该函数可以停止 gpioin 功能。

- 参数

类型	参数名	说明
GPIO_Pin_e	Pin	GPIO pin。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.5 I2S

3.3.5.1 枚举&宏

3.3.5.2 数据结构

3.3.5.3 APIs

3.3.6 I2C Master

本章节介绍 I2C Master 模式。

WT5105 有两个 I2C 控制器可以使用，通过 API 可以选择使用其中一个。

本章节主要举例介绍 I2C Master 的基本使用方法，以下通过实际用例介绍模块的应用，以加速度传感器 KX023 为例介绍。

3.3.6.1 参考代码

3.3.6.1.1 I2C 初始化

初始化过程主要包括 IO 配置，I2C 初始化。初始化需要配置 I2C 总线的速率: I2C_CLOCK_100K 或者 I2C_CLOCK_400K。

```
static void* kxi2c_init(void)
{
    void* pi2c;
    hal_i2c_pin_init(I2C_0, P26, P27);
    pi2c = hal_i2c_init(I2C_0,I2C_CLOCK_400K);
    return pi2c;
}
```

3.3.6.1.2 I2C Deinit

Deinit 过程要释放相关的资源，请参考以下代码。

```
static int kxi2c_deinit(void* pi2c)
{
    int ret;
    ret = hal_i2c_deinit(pi2c);
    hal_gpio_pin_init(P26,IE);
    hal_gpio_pin_init(P27,IE);
    return ret;
}
```

3.3.6.1.3 I2C 读写

读写过程需要输入从设备的 Slave address，具体实现请参考以下代码，由于有实时性要求，传输过程 FIFO 不能为空，所以写 FIFO 的过程不允许被打断。

```
static int kxi2c_read(void* pi2c, uint8_t reg, uint8_t* data, uint8_t size)
{
    return hal_i2c_read(pi2c, KX023_SLAVE_ADDR, reg, data, size);
}

static int kxi2c_write(void* pi2c, uint8_t reg, uint8_t val)
{
    uint8_t data[2];
    data[0] = reg;
    data[1] = val;
    hal_i2c_addr_update(pi2c, KX023_SLAVE_ADDR);
    {
        HAL_ENTER_CRITICAL_SECTION();
        hal_i2c_tx_start(pi2c);
        hal_i2c_send(pi2c, data, 2);
        HAL_EXIT_CRITICAL_SECTION();
    }
    return hal_i2c_wait_tx_completed(pi2c);
}
```

3.3.7 I2C Slave

TBD.

3.3.8 PWM

3.3.9 枚举&宏

3.3.9.1 PWMN_e

枚举 PWM 物理通道 0~5。

3.3.9.2 PWM_CLK_DIV_e

时钟分频。

PWM_CLK_NO_DIV	16MHz 时钟，无分频
PWM_CLK_DIV_2	2~128 分频。
PWM_CLK_DIV_4	
PWM_CLK_DIV_8	
PWM_CLK_DIV_16	
PWM_CLK_DIV_32	
PWM_CLK_DIV_64	
PWM_CLK_DIV_128	

3.3.9.3 数据结构

3.3.9.4 APIs

3.3.9.4.1 void hal_pwm_init(PWMN_e pwmN, PWM_CLK_DIV_e pwmDiv, PWM_CNT_MODE_e pwmMode, PWM_POLARITY_e pwmPolarity)

PWM 初始化，包括开启时钟，配置通道，配置分频参数，工作模式，极性。

- 参数

类型	参数名	说明
PWMN_e	pwmN	通道参数。

PWM_CLK_DIV_e	pwmDiv	分频参数。
PWM_CNT_MODE_e	pwmMode	计数模式，递增或者递减。
PWM_POLARITY_e	pwmPolarity	输出极性。

- 返回值

无。

3.3.9.4.2 void hal_pwm_open_channel(PWMN_e pwmN,GPIO_Pin_e pwmPin)

使能 PWM 通道，绑定 IO pin。

- 参数

类型	参数名	说明
PWMN_e	pwmN	通道参数。
GPIO_Pin_e	pwmPin	IO 管脚。

- 返回值

无。

3.3.9.4.3 void hal_pwm_close_channel(PWMN_e pwmN)

关闭 PWM 通道。

- 参数

类型	参数名	说明
PWMN_e	pwmN	通道参数。

- 返回值

无。

3.3.9.4.4 void hal_pwm_destroy(PWMN_e pwmN)

关闭 PWM 通道，并且通道配置清零。

- 参数

类型	参数名	说明
PWMN_e	pwmN	通道参数。

- 返回值

无。

3.3.9.4.5 void hal_pwm_set_count_val(PWMN_e pwmN, uint16_t cmpVal, uint16_t cntTopVal)

配置计数器数据，用于配置 PWM 占空比。

- 参数

类型	参数名	说明
PWMN_e	pwmN	通道参数。
uint16	cmpVal	计数器
uint16_t	cntTopVal	计数总数

- 返回值

无。

3.3.9.4.6 void hal_pwm_start(void)

启动 PWM 计数器。由于 PWM 工作需要依赖系统时钟，所以工作过程中会锁定不进入休眠。

- 参数

无。

- 返回值

无。

3.3.9.4.7 void hal_pwm_stop(void)

关闭 PWM 计数器，在此之前，需要关闭开启的 PWM 通道。

- 参数

无。

- 返回值

无。

3.3.10QDEC

TBD

3.3.11ADC

ADC 驱动提供异步的 AD 采集功能，数据采集完成之后通过回调函数回传 ADC 采集结果，ADC 驱动支持单端和差分两种模式。

如果需要使用 ADC 模块，一般需要在 main.c 的 hal_init()函数初始化，然后在 Application Task 按照下图进行使用，下图是假定同时采集两个 ADC 通道。

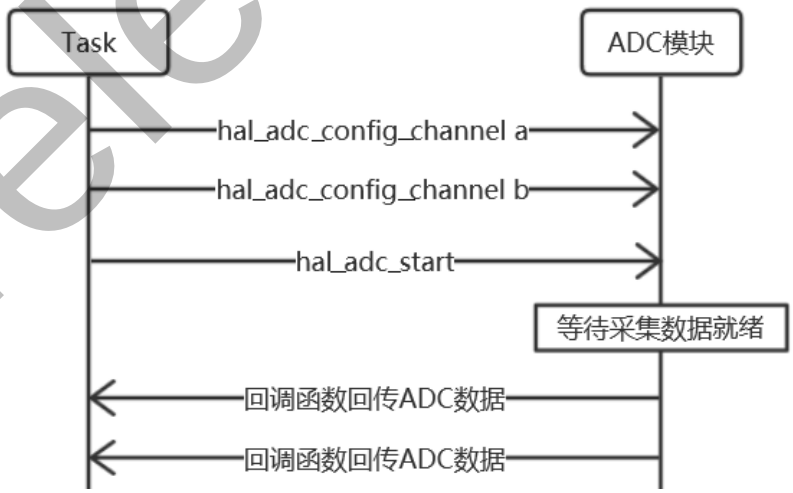

图 9

3.3.11.1 枚举&宏

3.3.11.1.1 adc_CH_t

ADC 物理通道。

ADC_CH0	暂不支持此通道。
ADC_CH1	暂不支持此通道。
ADC_CH1P 或 ADC_CH1DIFF	单端模式下中作为通道 1P，独立工作，差分模式下作为 1DIFF 通道，和 1N 通道组合使用。
ADC_CH1N	单端模式下作为通道 1N，差分模式下无效
ADC_CH2P 或 ADC_CH2DIFF	单端模式下中作为通道 2P，独立工作，差分模式下作为 2DIFF 通道，和 2N 通道组合使用。
ADC_CH2N	单端模式下作为通道 2N，差分模式下无效
ADC_CH3P 或 ADC_CH3DIFF	单端模式下中作为通道 3P，独立工作，差分模式下作为 3DIFF 通道，和 3N 通道组合使用。
ADC_CH3N	单端模式下作为通道 3N，差分模式下无效
ADC_CH_VOICE	语音通道，采集模拟麦克风使用。

3.3.11.1.2 ADC 事件

ADC 驱动事件，会在 ADC 驱动的回调函数抛出。

HAL_ADC_EVT_DATA	ADC 采样数据，如果采样数据就绪，会调用已注册的 ADC 回调函数，送出该事件。
HAL_ADC_EVT_FAIL	ADC 采样失败。

3.3.11.2 数据结构

3.3.11.2.1 adc_Evt_t

ADC 驱动事件的数据结构。

Int	type	ADC 事件类型。
adc_CH_t	ch	ADC 通道
uint16_t*	data	ADC 采样数据。
uint8_t	size	ADC 采样数据的大小，数据单位为 16bit。

3.3.11.2.2 adc_Hdl_t

ADC 回调函数类型。

typedef void (*adc_Hdl_t)(adc_Evt_t* pev)

3.3.11.2.3 adc_Cfg_t

ADC 配置参数。

Bool	is_continue_mode	是否连续采集模式，如果为 TRUE，ADC 采集在手动停止之前会一直工作
Bool	is_differential_mode	是否为差分模式，如果是差分模式，需要 P 端和 N 端成对配置。
Bool	is_high_resolution	是否采用衰减，如果该值为 TRUE，则采用衰减，量程为 0V~1V，，否则量程为 0V~4V。
Bool	is_auto_mode	暂不支持此功能。

3.3.11.3 APIs

3.3.11.3.1 void hal_adc_init(void)

ADC 模块初始化，模块其他函数需要配置之后方可使用，否则无法预测结果，或者返回错误。

- 参数

无。

- 返回值

无。

3.3.11.3.2 int hal_adc_config_channel (adc_CH_t channel, adc_Cfg_t cfg, adc_Hdl_t evt_handler)

配置 ADC 采集通道。

- 参数

类型	参数名	说明
adc_CH_t	channel	ADC 通道。
adc_Cfg_t	cfg	ADC 配置信息。
adc_Hdl_t	evt_handler	事件响应回调函数。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.11.3.3 int hal_adc_start(void)

开始采集。

- 参数

无。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.11.3.4 int hal_adc_stop(void)

停止采集。

- 参数

无。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.11.3.5 float hal_adc_value(uint16_t* buf, uint8_t size, bool high_resol, bool diff_mode)

计算 ADC 数值，输出为浮点数，为采样点的电压值，并且是输入 buffer 的算术平均值。

- 参数

类型	参数名	说明
uint16_t*	buf	采样原始数据。
uint8_t	size	采样数据的个数。
bool	High_resol	是否使用衰减（TRUE：量程为 0~1V；FALSE：量程为 0~4V）
bool	diff_mode	采样数据是否为差分数据。

- 返回值

float	采样点的电压值。

3.3.12SPI

SPI 提供 spi0、spi1 用于通信，当外围电路使用两组 SPI 时，软件分时切换两组 SPI 即可。

SPI 传输方式，可以选择轮询方式或中断方式。

如果数据量小，一次就传输完成，直接使用轮询方式方便简单一些。

如果数据量大，多次才传输完成，轮询方式或中断方式都可以。轮询方式会依次传输每帧数据，直到数据传输完毕，程序才继续往下执行。中断方式则在发送第一帧后便向下执行，数据每传递完成一次向应用索要数据，并同时接收数据，数据发送 buffer 和接收 buffer 在传输前配置，传输第一帧后程序可以往下执行但此时不允许进入休眠，因为 SPI 传输正在进行，等到数据传输完毕后才允许休眠使能，SPI 通过回调函数告知应用传输完成。

使用中断方式发送数据时，需要为 tx 分配内存，内存大小根据单次传输最大值配置。

SPI 使能信号，可以选择手动配置或由 SPI controller 自动配置，这个需要根据外设的特性决定。

具体使用请参考 SPI flash 实例。

3.3.12.1 枚举&宏

3.3.12.1.1 SPI_INDEX_e

SPI 模块选择。

SPI0	spi 0
SPI1	spi 1

3.3.12.1.2 SPI_TMOD_e

SPI 使用场景。

SPI_TRXD	Transmit & Receive
SPI_TXD	Transmit Only
SPI_RXD	Receive Only
SPI_EEPROM	EEPROM Read

3.3.12.1.3 SPI_SCMOD_e

SPI 工作模式。

SPI_MODE0	SCPOL=0,SCPH=0
SPI_MODE1	SCPOL=0,SCPH=1
SPI_MODE2	SCPOL=1,SCPH=0
SPI_MODE3	SCPOL=1,SCPH=1

3.3.12.1.4 spi_Hdl_t

SPI 传输完成回调函数，以中断方式传输时需要配置。

3.3.12.1.5 spi_Type_t

SPI 传输完成回调函数参数。

TRANSMIT_COMPLETED	传输完成

3.3.12.2 数据结构

3.3.12.2.1 spi_Cfg_t

SPI 配置参数。

GPIO_Pin_e	sclk_pin	配置 clk 引脚
GPIO_Pin_e	ssn_pin	配置 cs 引脚
GPIO_Pin_e	MOSI	配置 mosi 引脚
GPIO_Pin_e	MISO	配置 miso 引脚
uint32_t	baudrate	设置 spi 波特率
SPI_TMOD_e	spi_tmod	配置 spi 工作模式
SPI_SCMOD_e	spi_scmod	配置 spi 工作模式
bool	int_mode	使用中断模式/非中断模式传输

bool	force_cs	使用手工设/IP 拉动 cs 引脚
spi_Hdl_t	evt_handler	使用中断方式传输时，传输完毕时回调函数

3.3.12.3 APIs

3.3.12.3.1 void hal_spi_init(void)

SPI 模块初始化，模块其他函数需要配置之后方可使用，否则无法预测结果，或者返回错误。

- 参数

无。

- 返回值

无。

3.3.12.3.2 int hal_spi_bus_init(hal_spi_t* spi_ptr,spi_Cfg_t cfg)

配置使能 SPI 模块。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄。
spi_Cfg_t	cfg	SPI 配置信息。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.12.3.3 int hal_spi_bus_deinit(hal_spi_t* spi_ptr)

禁止 SPI 模块。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.12.3.4 int hal_spi_set_int_mode(hal_spi_t* spi_ptr,bool en)

设置 SPI 传输方式，中断/非中断。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄。
bool	en	true-中断方式 false—非中断方式

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.12.3.5 int hal_spi_set_force_cs(hal_spi_t* spi_ptr,bool en)

设置 SPI cs 引脚控制方式，建议使用手动方式。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄。
bool	en	true-手动设置 cs 状态 false—IP 设置 cs 状态

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.12.3.6 int hal_spi_int_set_tx_buf(hal_spi_t* spi_ptr,uint8_t* tx_buf,uint16_t len)

使用中断方式发送数据时，需要为其分配一段 buffer。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄。
uint8_t*	tx_buf	发送 buffer 指针
uint16_t	len	发送 buffer 长度

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.12.3.7 int hal_spi_transmit(hal_spi_t* spi_ptr,uint8_t* tx_buf,uint8_t* rx_buf,uint16_t len)

SPI 启动数据传输。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄。
uint8_t*	tx_buf	发送 buffer 指针
uint8_t*	rx_buf	接收 buffer 指针
uint16_t	len	buffer 长度，发送 buffer 和接收 buffer 长度需要相等

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.12.3.8 bool hal_spi_get_transmit_bus_state(hal_spi_t* spi_ptr)

查询 SPI 状态，是否有数据正在用中断方式传输。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄

- 返回值

true	SPI 空闲
false	SPI 忙，当前有数据尚未传输完成

3.3.12.3.9 int hal_spi_module_select(hal_spi_t* spi_ptr)

SPI 模块选择，只有两组 SPI 都会用到时才会调用，两组 SPI 外围应用电路独立，软件分时复用。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.12.3.10 void hal_spi_send_byte(hal_spi_t* spi_ptr,uint8_t data)

SPI 发送一个数据，接口预留满足向前兼容性。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄
uint8_t	data	要发送的数据

- 返回值

无。

3.3.12.3.11 void hal_spi_TxComplete(hal_spi_t* spi_ptr)

等待 SPI 传输完成，接口预留满足向前兼容性。

- 参数

类型	参数名	说明
hal_spi_t*	channel	SPI 通道句柄

- 返回值

无。

3.3.13UART

UART 提供一组 API 用于 UART 同步或者异步模式的控制。

对于 Rx，本模块只提供异步模式接口，在有 Rx 数据就绪的情况下，会通过初始化时候注册的回调函数抛出 UART_EVT_TYPE_RX_DATA 事件或者 UART_EVT_TYPE_RX_DATA_TO（超时：当最后一笔数据不足 FIFO Threshold 时）。

对于 Tx，本模块提供同步和异步两种接口，如果初始化时候配置为 cfg. use_tx_buf 为 FALSE，Tx 同步，可以调用函数 hal_uart_send_buff (uint8_t *buff,uint16_t len)，以同步方式发送数据，函数为阻塞。

如果初始化时候配置 cfg. use_tx_buf 为 TRUE，并且 cfg. use_fifo 为 TRUE，那么 Tx 为异步，初始化完成之后还需要调用 hal_uart_set_tx_buf(uint8_t* buf, uint16_t size)配置 Tx 缓存，之后才能使用 hal_uart_send_buff (uint8_t *buff,uint16_t len)发送数据，此时该函数非阻塞，发送完成之后会通过回调函数抛出 UART_EVT_TYPE_TX_COMPLETED 事件，应用程序根据需要响应该事件。

3.3.13.1 枚举&宏

3.3.13.1.1 uart_Evt_Type_t

对于异步传输，回调函数会抛出以下事件类型之一。

UART_EVT_TYPE_RX_DATA	UART Rx 数据就绪。
UART_EVT_TYPE_RX_DATA_TO	UART Rx 数据就绪，接收超时。
UART_EVT_TYPE_TX_COMPLETED	UART Tx 传输完成。

3.3.13.2 数据结构

3.3.13.2.1 uart_Cfg_t

UART 配置参数。

GPIO_Pin_e	tx_pin	设置 Tx 管脚
GPIO_Pin_e	rx_pin	设置 Rx 管脚
uint32_t	Baudrate	设置传输波特率
bool	use_fifo	是否使用 FIFO，建议使用。
bool	use_tx_buf	是否使用 Tx 缓存，如果使用缓存，Tx 为异步模式
uart_Hdl_t	evt_handler	设置回调函数。
GPIO_Pin_e	ts_pin	暂不支持。
GPIO_Pin_e	cts_pin	暂不支持。
bool	hw_fwctrl	暂不支持。
bool	parity	暂不支持。

3.3.13.3 APIs

3.3.13.3.1 int hal_uart_init (uart_Cfg_t cfg)

用于 UART 模块初始化，模块其他函数需要配置之后方可使用，否则无法预测结果，或者返回错误。

- 参数

类型	参数名	说明
uart_Cfg_t	cfg	串口配置信息

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.13.3.2 int hal_uart_set_tx_buf(uint8_t* buf, uint16_t size)

设置 Tx 缓存，如果初始化时候 cfg. use_tx_buf 为 TRUE，本函数需要执行。

- 参数

类型	参数名	说明
uint8_t*	buf	需要配置的缓存指针。
uint16_t	size	缓存大小。

- 返回值

PPlus_SUCCESS	缓存配置成功。
其他数值	参考<error.h>

3.3.13.3.3 int hal_uart_get_tx_ready(void)

用于用于查询一个异步的 Tx 传输是否完成，或者说查询 Tx 是否空闲。

- 参数

无。

- 返回值

PPlus_SUCCESS	Tx 传输完成。
PPlus_ERR_BUSY	Tx 忙。

3.3.13.3.4 hal_uart_send_buff(uint8_t *buff,uint16_t len)

Tx 发送数据，如果为同步模式，函数阻塞，发送完成或者发送超时返回，如果为异步模式，函数非阻塞，数据传输完成之后通过回调函数抛出 UART_EVT_TYPE_TX_COMPLETED 事件。

- 参数

类型	参数名	说明
uint8_t*	buf	需要发送的数据。
uint16_t	len	大小。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.3.14Key Scan

TBD.

3.3.15Flash

3.3.15.1 枚举&宏

3.3.15.2 数据结构

3.3.15.3 APIs

3.3.15.3.1 uint8_t flash_write_ucds(uint32_t addr,uint32_t value)

用于用户写 flash，用户地址范围为 0x0000~0x3ffc，超出此范围报错。

- 参数

类型	参数名	说明
uint32_t	addr	Flash 地址（0x0000~0x3ffc）
uint32_t	value	写入 flash 的值

- 返回值

PPlus_SUCCESS	写入成功
其他数值	参考<error.h>

3.3.15.3.2 uint32_t flash_read_ucds(uint32_t addr)

用于用户读 flash，用户地址范围为 0x0000~0x3ffc，超出此范围报错。

- 参数

类型	参数名	说明
uint32_t	addr	Flash 地址（0x0000~0x3ffc）

- 返回值

0xffffffff	Flash 值为 0xffffffff 或者地址错误
其他数值	Flash 地址的数值

3.3.15.3.3 void flash_erase_ucds_all(void)

用于用户擦除全部 flash，用户地址范围为 0x0000~0x3ffc

- 参数

无。

- 返回值

无。

3.3.15.3.4 uint8_t flash_erase_ucds(uint32_t addr)

用于用户擦除指定地址的 sector，用户地址范围为 0x0000~0x3ffc

- 参数

类型	参数名	说明
uint32_t	addr	Flash 地址（0x0000~0x3ffc）

- 返回值

PPlus_SUCCESS	写入成功
其他数值	参考<error.h>

3.4 休眠管理

WT5105 系列芯片对外提供两种休眠模式：休眠模式和 Power Off 模式

对于休眠模式，SDK 对于休眠和唤醒进行集中管理，并通过预编译宏使能或者禁止休眠，如果应用代码需要某个过程禁止休眠，可参考章节进行配置。

对于 Power Off 模式，SDK 提供 API 用于进入 Power Off 模式，并且可以通过参数配置。

3.4.1 使能和禁止休眠

可以再 MDK 的项目→Option 对话框→C/C++页面→Preprocessor Symbol→Define 进行配置：

CFG_SLEEP_MODE=PWR_MODE_SLEEP	使能休眠
CFG_SLEEP_MODE=PWR_MODE_NO_SLEEP	禁止休眠

3.4.1.1 Application 如何控制休眠

应用代码可以通过注册 PwrMgr 模块的方式获得休眠的控制权，需要注意的是 MOD_VOC 以下模块预留给实际的硬件模块， MOD_USR0 是预留给 rfPhy，应用可以使用的模块是 MOD_USR1 以及以上的模块，具体模块定义请参考模块 ID：

具体代码参考如下：

```
//注册应用模块
hal_pwrmgr_register(MOD_USR1, NULL, rf_wakeup_handler);

//应用禁止休眠
hal_pwrmgr_lock(MOD_USR1);
//应用使能休眠
hal_pwrmgr_lock (MOD_USR1);
```

3.4.2 Power Off 模式

如果需要进入 Power Off 模式，可以调用函数 hal_pwrmgr_poweroff(pwroff_cfg_t cfg)进入睡眠，参数 cfg 用于配置唤醒 pin。

在 Power off 模式下，系统处于 Power off 状态，Power off 之前的 BLE 广播或者连接都会断开，唤醒之后系统相当于冷启动。

3.4.3 相关的 APIs

3.4.3.1 APIs

3.4.3.1.1 int hal_pwrmgr_init(void)

模块初始化。

- 参数

无。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.4.3.1.2 bool hal_pwrmgr_is_lock(MODULE_e mod)

查询某个模块（硬件或者应用模块）是否禁止休眠。

- 参数

类型	参数名	说明
MODULE_e*	mod	模块 ID。

- 返回值

TRUE	该模块禁止休眠。
FALSE	该模块允许休眠

3.4.3.1.3 int hal_pwrmgr_lock(MODULE_e mod)

模块禁止休眠。

- 参数

类型	参数名	说明
MODULE_e*	mod	模块 ID。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.4.3.1.4 int hal_pwrmgr_unlock(MODULE_e mod)

模块允许休眠。

- 参数

类型	参数名	说明
MODULE_e*	mod	模块 ID。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.4.3.1.5 int hal_pwrmgr_register(MODULE_e mod, pwrmgr_Hdl_t sleepHandle, pwrmgr_Hdl_t wakeupHandle)

向休眠管理系统注册，包括注册模块，注册休眠回调函数，注册唤醒回调函数，其中模块是必选项，休眠和唤醒回调函数为可选参数，如果不需要设置，参数设置为 NULL。

- 参数

类型	参数名	说明
MODULE_e*	mod	模块 ID。
pwrmgr_Hdl_t	sleepHandle	回调函数，进入休眠之前会调用该函数。
pwrmgr_Hdl_t	wakeupHandle	回调函数，从休眠唤醒之后会调用该函数。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.4.3.1.6 int hal_pwrmgr_unregister(MODULE_e mod)

模块从休眠管理系统注销。

- 参数

类型	参数名	说明
MODULE_e*	mod	模块 ID。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.4.3.1.7 int hal_pwrmgr_RAM_retention(uint32_t sram)

配置休眠过程中能够保持的 RAM，Bit 0~4 有效，分别对应 RAM0~RAM4，如果对应的 Bit 为 1，这块 RAM 在休眠过程中数据能够保持，如果为 0，那么休眠过程中，数据会丢失。

- 参数

类型	参数名	说明
uint32_t	sram	配置休眠过程中能够保持的 RAM，Bit 0~4 有效，分别对应 RAM0~RAM4。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.5 Libraries

3.5.1 文件系统

基于内部 flash 的轻量级文件系统，该文件系统提供一组同步 API，支持 16bit 文件 ID，文件查找，读、写、删除、以及文件系统查询，垃圾文件回收。

3.5.1.1 枚举&宏

3.5.1.1.1 FS_SETTING

FS 每条记录长度，该宏定义可以在工程的配置中设置，如果设置默认每条记录长度 16 字节。

FS_ITEM_LEN_16BYTE	每条记录长度 16 字节
FS_ITEM_LEN_32BYTE	每条记录长度 32 字节
FS_ITEM_LEN_64BYTE	每条记录长度 64 字节

3.5.1.2 数据结构

无。

3.5.1.3 APIs

3.5.1.3.1 int hal_fs_init(uint32_t fs_start_address,uint8_t sector_num)

用配置参数初始化文件系统，该函数需要在系统初始化时候设置，具体请参考例程。

- 参数

类型	参数名	说明
uint32_t	fs_start_address	FS 起始地址。 需要 4096 字节对齐，空间上不能和其他使用有冲突。
uint8_t	sector_num	FS 扇区数量，有效值 3~78。 举例： 将 FS 分配 4 个扇区，起始地址为 0x11005000 hal_fs_init(0x11005000,4)

- 返回值

PPlus_SUCCESS	初始化成功。
其他	参考<error.h>

3.5.1.3.2 int hal_fs_item_read(uint16_t id,uint8_t* buf,uint16_t buf_len,uint16_t* len)

读取 FS 文件。

- 参数

类型	参数名	说明
uint16_t	id	读取文件的 id。
uint8_t*	buf	传入 buffer 起始地址。

buf_len	buf_len	传入 buffer 起始长度
uint16_t*	len	文件实际长度

- 返回值

PPlus_SUCCESS	初始化成功。
其他	参考<error.h>

3.5.1.3.3 int hal_fs_item_write(uint16_t id,uint8_t* buf,uint16_t len)

写入 FS 文件。

- 参数

类型	参数名	说明
uint16_t	id	写入文件的 id。
uint8_t*	buf	传入 buffer 起始地址。
uint16_t	len	传入 buffer 起始长度

- 返回值

PPlus_SUCCESS	初始化成功。
其他数值	参考<error.h>

3.5.1.3.4 uint32_t hal_fs_get_free_size(void)

FS 可用来存储文件数据的空间大小，单位为字节。

- 参数

无。

- 返回值

FS 可用存储文件空间，单位为字节。

3.5.1.3.5 int hal_fs_get_garbage_size(uint32_t* garbage_file_num)

FS 已删除文件数据区大小，单位为字节。

- 参数

类型	参数名	说明
uint32_t*	garbage_file_num	已删除文件的数量。

- 返回值

FS 已删除文件所占空间，单位为字节。

3.5.1.3.6 int hal_fs_item_del (uint16_t id)

删除文件。

- 参数

类型	参数名	说明
uint16_t	id	删除文件的 id。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.5.1.3.7 int hal_fs_garbage_collect(void)

垃圾回收，将 FS 中已经删除的文件所占有的空间释放。

该函数会遍历整个 FS，还会对多个扇区进行擦除操作，耗时相对较多。

建议在 CPU 空闲时且 garbage 较多时执行，执行时间和主频、FS 大小都有关系。

- 参数

无。

- 返回值

PPlus_SUCCESS	初始化成功。
其他数值	参考<error.h>

3.5.1.3.8 int hal_fs_format (uint32_t fs_start_address,uint8_t sector_num)

格式化 FS，所有文件会被擦除，使用需谨慎。

如果必须调用，建议在 CPU 空闲时调用，执行时间和主频、FS 大小都有关系。

- 参数

类型	参数名	说明
uint32_t	fs_start_address	FS 起始地址。 需要 4096 字节对齐，空间上不能和其他使用有冲突。
uint8_t	sector_num	FS 扇区数量，有效值 3~78。 举例： 将 FS 分配 4 个扇区，起始地址为 0x11005000 hal_fs_init(0x11005000,4)

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.5.1.3.9 bool hal_fs_initialized(void)

查询 FS 初始化状态。

- 参数

无。

- 返回值

true	已初始化，可以使用。
false	未初始化，不能使用。

3.5.2 日期时间

实现万年历功能，datetime_t 为精确到秒的时间，该库作为后台运行的服务，受 osal_timer 驱动，实现时间的自动更新，并提供 API 用于时间获取和时间设置。

3.5.2.1 枚举&宏

3.5.2.1.1 USE_SYS_TICK

TRUE	使用 osal_sys_tick 作为时钟基准计数，在使用 RC 作为 32K 时钟源的情况下，osal_sys_tick 作为经过校准的 tick 值，有较高的精度
FALSE	使用 RTC 计数器作为时钟的基准计数。

3.5.2.2 数据结构

3.5.2.2.1 datetime_t

日期时间的数据格式。

uint8_t	seconds	秒。
uint8_t	minutes	分钟。
uint8_t	hour	小时（0~23）。
uint8_t	day	日期（day of month）。
uint8_t	month	月份（1~12）。

uint16_t	year	年。

3.5.2.3 APIs

3.5.2.3.1 void app_datetime_init(void)

日历应用初始化，通常在应用 task 初始化时候调用。

- 参数

无。

- 返回值

无。

3.5.2.3.2 void app_datetime_sync_handler(void)

日历应用同步函数，要求一分钟左右同步一次（或者小于一分钟）。不需要精确的调用。

- 参数

无。

- 返回值

无。

3.5.2.3.3 int app_datetime_set(datetime_t dtm)

根据 dtm 设置系统的时间。

- 参数

类型	参数名	说明
datetime_t	dtm	需要设置的时间值，精确到秒

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.5.2.3.4 int app_datetime(datetime_t* pdtm)

获取当前的系统时间。

- 参数

类型	参数名	说明
datetime_t*	pdtm	输出参数，如果返回成功，pdtm 会被写入最新的系统时间。

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.5.2.3.5 int app_datetime_diff(const datetime_t* pdtm_base, const datetime_t* pdtm_cmp)

比较两个时间，获取差值，单位为秒。

- 参数

类型	参数名	说明
const datetime_t*	pdtm_base	被比较的时间。
const datetime_t*	pdtm_cmp)	比价的时间，公式为 pdtm_cmp – pdtm_base

- 返回值

int	时间的差值，如果被比较的时间比较新，那么返回值为负

3.5.3 点阵字库

WT5105 Font library 提供多语言，可配置的字库方案。

字库方案包括两个部分：

- WT5105 Font library + API
  - 字库驱动库和 API 函数，提供字库的注册，UTF-8 解析和字库 bitmap 接口
  - 字库 Bitmap 接口支持可变宽度文字。
- WT5105 多语言字库
  - 根据客户需要生成多语言字库，在 Unicode 基本多文种平面（BMP 0x0000~0xffff）可以根据需要配置任意语种组合，并且可以根据客户需要配置文字分辨率，扫描方式。
  - 对于中文，可以支持 2009 版本的 3500 常用字生成方案。
  - 字库文件提供二进制格式的文件（.bin）和升级文件(.res)。

3.5.3.1 APIs

3.5.3.1.1 void* ui_font_load(uint32_t flash_addr)

加载字库，将已烧录在内部 flash 的字库加载，加载之后会获得一个字库句柄，可以通过该句柄调用函数 ui_font_unicode()获取 Unicode 字符的 bitmap。

- 参数

类型	参数名	说明
uint32_t	flash_addr	字库数据存储的 flash 地址

- 返回值

返回 void*类型字库句柄，如果数值为 NULL 表示加载失败。

3.5.3.1.2 int utf8_to_unicode(const char* utf8, uint16_t *unicode)

将 UTF-8 字符串转换为 Unicode 字符。

- 参数

类型	参数名	说明
const char*	utf8	输入参数，输入的 UTF-8 字符串，以'\0'结束。
uint16_t *	unicode	输出参数，转换 UTF-8 得到的一个 Unicode 字符。

- 返回值

返回 int 类型。

0	字符串解析完成，输出参数为无效数据。
1~n	转换获得一个 Unicode 字符，对应的 UTF-8 字符串解析消耗 n 个字节。

3.5.3.1.3 int ui_font_unicode(void* font, uint16_t unicode,uint8_t *bitmap)

根据 Unicode 字符获取 bitmap 数据。

- 参数

类型	参数名	说明
void*	font	字库句柄。
uint16_t	unicode	Unicode 字符，需要转换为 bitmap
uint8_t *	bitmap	Unicode 字符的 bitmap. 其中字节 0~3 为分辨率信息，字节 4 及其以后数据为 bitmap 数据。 字节 0~3 具体内容： Byte0： bitmap 数据宽度 Byte1： bitmap 数据高度 Byte2： bitmap 实际有效数据宽度 Byte4： reserved。

- 返回值

返回 int 类型。

0	操作成功。
其他数值	错误码。

3.5.3.1.4 const char* ui_font_version(void)

返回字库 lib 的版本信息和字库文件的版本信息。

- 参数

无。

- 返回值

const char* 类型字符串。

4 BLE

4.1 GAP

通用访问配置文件（GAP）：

Ble 协议栈中的 GAP 层负责处理设备访问模式，包括设备发现、建立连接、终止连接、初始化安全管理和设备配置，所以在 ble 协议栈中有不少函数均是以 GAP 为前缀，这些函数会负责以上的内容。

GAP 层总是作为下面四个钟角色之一：

- Broadcaster 广播者——不可以连接的一直在广播的设备；
- Observer 观测者——可扫描广播设备，但不能发起建立连接的设备；
- Peripheral 从机 ——可被连接的广播设备，可以在单个链路层连接中作从机。
- Central 主机 ——可以扫描广播设备并发起连接，在单个链路层或多链路层中作为主机。

在典型的蓝牙低功耗系统中，从机设备广播特定的数据，以便让主机知道他是一个可以连接的设备，广播内容包括设备地址以及一些额外的数据，如设备名、服务等。主机收到广播数据后，会向从机发送扫描请求 ScanRequest，然后从机将特定的数据回应给主机，称为扫描回应 ScanResponse。主机收到扫描回应后，便知道这是一个可以建立连接的外部设备，这就是设备发现的全过程。此时，主机可以向从机发起建立连接的请求，连接请求包括下面一些参数。

- 连接间隔——在两个 BLE 设备的连接中使用调频机制，两个设备使用特定的信道收发数据，然后过一段时间后再使用新的信道。（链路层处理信道切换），两设备在信道切换后收发数据称之为连接事件，即使没有应用数据的收发，两个设备任然会通过交换链路层数据来维持连接，连接间隔就是两个连接事件之间的时间间隔，连接间隔以 1.25ms 为单位，连接间隔的值为 6（7.5ms）~3200（4s）。
- 从机延时——这个参数的设置可以使从机跳过若干连接事件，这给了从机更多的灵活度，如果它没有数据发送时，可以选择跳过连接时间继续休眠，以节省功耗。
- 管理超时——这是两个成功连接事件之间的最大允许的间隔，如果超过了这个时间（这个值的单位是 10ms）而没有成功的连接事件，设备被认为丢失连接，返回到未连接状态，管理超时的范围是 100（100ms）~3200（32s）另外，超时值必须大于有效的连接间隔[有效的连接间隔=连接间隔*（1+从机延时）]。
- 安全管理——只有已认证的连接中，特定的数据数据才能被读写，一旦连接建立，两个设备进行配对，当配对完成后，形成加密连接的密钥，在典型的应用中，外设请求集中器提供密钥来完成配对工作。密钥是一个固定的值，如 000000，也可以随机生成一个数据提供给使用者，当主机设备发送正确的密钥后，两设备交换安全密钥并加密认证链接。在许多情况下，同一对外设和主机会不时的连接和断开，ble 的安全机制中有一项特性，允许两个设备之间建立长久的安全密钥信息，这种特性称为绑定，他允许两设备连接时快速的完成加密认证，而不需要每次连接时执行配对的完整过程。

4.2 GATT

GATT(Generic Attribute Profile)：通用属性配置文件，是在属性协议(ATT)基础上构建，为属性协议传输和存储数据简历了一些通用的操作和框架。

4.2.1 如何实现自定义服务

本章节介绍如何通过 SDK 实现自定义服务。

我们通过 BLE-Uart 例程介绍如何实现一个服务，其中包含写、Notify、Indicate 三种类型的特征值。

4.2.1.1 建立属性表

属性表为一个 gattAttribute_t 类型的静态数组，包含：

- 主服务。
- 特征值 1 的属性(GATT_PROP_WRITE_NO_RSP| GATT_PROP_WRITE)。
- 特征值 1 的值
- 特征值 2 的属性(GATT_PROP_NOTIFY| GATT_PROP_INDICATE)。
- 特征值 2 的客户端特性配置描述符(Client Characteristic Configuration Descriptor)。
- 特征值 2 的值。

具体内容请参考 SDK 例程代码。

```
static gattAttribute_t bleuart_ProfileAttrTbl[] =
{
  // 主服务
  {
    { ATT_BT_UUID_SIZE, primaryServiceUUID }, /* type */
    GATT_PERMIT_READ,                         /* permissions */
    0,                                        /* handle */
    (uint8 *)&bleuart_Service                 /* pValue */
  },
    // 特征值 1：属性（写，写-无需应答）
    {
      { ATT_BT_UUID_SIZE, characterUUID },
      GATT_PERMIT_READ,
      0,
      &bleuart_RxCharProps
    },
      // 特征值 1：值
      {
        { ATT_UUID_SIZE, bleuart_RxCharUUID },
        GATT_PERMIT_WRITE,
        0,
        &bleuart_RxCharValue[0]
      },
    // 特征值 2：属性（Notify）
    {
      { ATT_BT_UUID_SIZE, characterUUID },
      GATT_PERMIT_READ,
      0,
      &bleuart_TxCharProps
```

```
    },
      // 特征值 2：值
      {
        { ATT_UUID_SIZE, bleuart_TxCharUUID },
        0,
        0,
        (uint8 *)&bleuart_TxCharValue
      },
      // 特征值 2：客户端特性配置描述符(Client Characteristic Configuration Descriptor)
      {
        { ATT_BT_UUID_SIZE, clientCharCfgUUID },
        GATT_PERMIT_READ|GATT_PERMIT_WRITE,
        0,
        (uint8*)&bleuart_TxCCCD          },
};
```

4.2.1.2 需要实现的函数

完成属性表配置之后，我们需要添加服务，并且注册读写回调函数，以下为这部分函数的说明。

4.2.1.2.1 bStatus_t bleuart_AddService(bleuart_ProfileChangeCB_t cb)

用于添加本服务，主要实现以下功能：

- 注册 link status 的回调，用于响应连接状态的改变。
- 注册服务，通过向协议栈传递属性表、GATT 服务回调函数，实现服务的注册。
- 保存应用层的回调函数指针，用于向应用层。

4.2.1.2.2 static bStatus_t bleuart_WriteAttrCB(uint16 connHandle, gattAttribute_t *pAttr, uint8 *pValue, uint8 len, uint16 offset)

回调函数，响应 Central 端的写(Write 或者 Write no response)操作，其中包含两部分，对特征值 1 的写操作；对于特征值 2 的客户端特征值配置描述符的写操作。

具体实现请参考例程代码。

4.2.1.2.3 static void bleuart_HandleConnStatusCB (uint16 connHandle, uint8 changeType)

回调函数，响应 link status 的改变。

4.2.1.2.4 bStatus_t bleuart_Notify(uint16 connHandle, attHandleValueNoti_t *pNoti, uint8 taskId)

函数，用于发送 Notify 给 Central。

4.2.1.3 应用层调用服务接口

应用层在应用 Task 初始化函数调用 bleuart_AddService()函数，在 BLE-Uart 例程是在函数 bleuart_Init()调用。

4.3 OTA

OTA 即 Over the Air，指的是通过 BLE 无线升级固件的功能，WT5105 的 OTA 提供一套完善可靠的固件空中升级方案，升级包括应用固件升级、资源文件升级、OTA bootloader 升级。

- 应用固件升级

  升级应用固件，支持非加密升级和加密升级。

- 资源文件升级

  升级资源文件，资源文件存储区域为非程序区和系统保护区，对于 NVM 区域，需要用户自主保护，OTA 升级过程不会对此区域保护。

- OTA Bootloader 升级

  升级 OTA 引导程序可以看作是一种特殊的应用固件升级，通常这部分区域不需要进行升级，如果需要升级该区域，请参考例程< OTA_upgrade_2ndboot>。OTA bootloader 升级之后，应用固件也会失效，所以，在完成 OTA bootloader 升级之后，还需要进行应用固件升级。

4.3.1 OTA 模式

应用固件升级过程需要在 OTA 模式下进行。

有三种方法可以进入 OTA 模式：

- 未检测到应用固件，如果应用固件损坏或者仅仅烧写了 OTA Bootloader 固件，那么设备 reset 或者上电之后默认进入 OTA 模式。
- 在应用模式下，通过 OTA App Service 的指令可以使系统进入 OTA 模式，支持 OTA 的应用固件要求加载 OTA App Service，通过该 Service，host 对该服务下的特征值发送控制命令使应用固件跳转到 OTA 模式
- 第三种模式需要根据用户需求定制，通常是通过识别特定 IO 的状态进入 OTA 模式。

4.3.2 OTA Resource 模式

资源文件升级过程需要在 OTA Resource 模式下进行。

应用模式通过 OTA App Service 的指令可以进入 OTA Resource 模式。

4.3.3 OTA Service

应用固件需要加载如下图所示的 OTA Service，通过该 service，host 可以通过命令控制设备跳转到 OTA 模式。

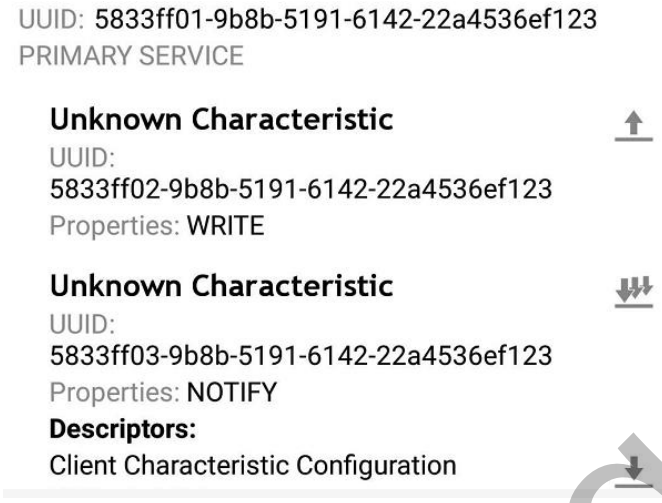

图 10

OTA App Service 代码位于 SDK\components\profiles\ota_app

4.3.4 OTA Bootloader

OTA bootloader 是二级引导程序，通过该程序可以实现应用固件的加载、加密应用固件的加载、加密和非加密 OTA 功能、OTA resource 功能。该固件在 OTA 模式下会运行在 RAM 的高端地址区域，应用固件引导完毕之后会释放 RAM 的控制权，所有 RAM 交由应用管理。

OTA bootloader 针对不同的应用场景，又分为以下几种模式（针对 512KB Flash 以及以上版本），对于不同 OTA 模式，应用固件不需要做任何调整。

4.3.4.1 OTA Dual Bank Has FCT

为应用固件在内部 flash 上开辟两个对称的 128K 空间，并且为 FCT 固件（功能测试固件）单独开辟了 120K 空间，这种模式的 Flash 地址映射如下：

	512K 版本（Dual bank)(Has FCT)	
Reserved By wireless-tag	00000~09fff	40k
OTA bootloader	0a000~11fff	32k
FCT App	12000~2ffff	120k

App Bank0	30000~4ffff	128k
App Bank1	50000~6ffff	128k
NVM	70000~7ffff	64k

4.3.4.2 OTA Dual Bank No FCT

为应用固件在内部 flash 上开辟两个对称的 128K 空间，但是不单独提供 FCT 固件，这种模式 Flash 地址映射如下：

	512K 版本 (Dual bank) (No FCT)	
Reserved By wireless-tag	00000~09fff	40k
OTA bootloader	0a000~11fff	32k
App Bank0	12000~31fff	128k
App Bank1	32000~51fff	128k
NVM	52000~7ffff	184k

4.3.4.3 OTA Single Bank Has FCT

该模式仅为应用固件开辟 1 块 128K Flash 空间，同时还提供用于功能测试固件的 120K 空间，该模式 Flash 空间映射如下：

	512K 版本(Dual bank)(Has FCT)	
Reserved By wireless-tag	00000~09fff	40k
OTA bootloader	0a000~11fff	32k
FCT App	12000~2ffff	120k
App Bank0	30000~4ffff	128k
NVM	50000~7ffff	192k

4.3.4.4 OTA Single Bank No FCT

该模式仅为应用固件开辟 1 块 128K Flash 空间，不提供用于功能测试固件的空间，该模式 Flash 空间映射如下：

	512K 版本(Dual bank) (No FCT)	
Reserved By wireless-tag	00000~09fff	40k
OTA bootloader	0a000~11fff	32k
App Bank0	12000~31fff	128k
NVM	32000~7ffff	312k

4.3.5 加密 OTA

OTA 支持加密传输和加密存储，对于加密模式的 OTA，需要通过 PC 工具对应用固件进行加密，并且将密钥植入芯片内部，从应用角度看升级过程和非加密的 OTA 没有区别，如果密钥不匹配，升级过程会报告错误报文。

4.3.6 如何实现 OTA

如果芯片烧写 OTA Bootloader，就具备了进行 OTA 的能力，对于应用固件来说，需要加载 OTA App Service，使得应用固件能够和 OTA Bootloader 进行交互。

4.3.6.1 应用固件和 OTA

应用固件需要加载 OTA App Service，一般来说我们会在应用 Task 初始化的过程加载该服务，请参考样例代码：SDK\example\ble_peripheral\otaDemo

具体代码片段 如下：

```
void otaDemo_Init( uint8 task_id )
{
  otaDemo_TaskID = task_id;

  // 此处省略部分代码


  // Initialize GATT attributes
  GGS_AddService( GATT_ALL_SERVICES );            // GAP
  GATTServApp_AddService( GATT_ALL_SERVICES ); // GATT attributes
  GATTServApp_RegisterForMsg(otaDemo_TaskID);
  DevInfo_AddService();
  otaDemo_AddService();
  ota_app_AddService();


  // Setup a delayed profile startup
  osal_set_event( otaDemo_TaskID, START_DEVICE_EVT );
}
```

4.3.7 烧写应用固件和 OTA bootloader

通过 WT51-RFtools工具或者 WT51 烧写器硬件都可以进行应用固件和 OTA Bootloader 的烧 写，具体使用请参考 WT51-RFtools 使用指南和烧写器的使用文档。

4.3.8 OTA 总结

总得来说实现 OTA 需要三个步骤:

- 编译 OTA Bootloader 获得 ota.hex，例程提供了四种模式的 hex 文件（For 512K flash 版本的硬件）。
- 应用固件添加 OTA App Service: ota_app_AddService();
- 通过 WT51-RFtools 或者 WT51 烧写器进行芯片的编程。

完成以上步骤之后，可以通过安卓或者 iOS 版本的 WTApp 进行应用固件或者资源文件（比如字库文件）的升级更新。

5 样例程序

在 SDK 中提供下表所示应用例程：

①：例程可以在开发板 WT5105_32_V1.4 以及以上版本运行；

②：例程可以在开发板 WT5105_48_Bracelet_V1.0 以及以上版本运行；

③：例程可以在开发板 WT_BLE_Mesh_V1.0 以及以上版本运行。

ble_peripheral		
alternate_iBeacon	alternate iBeacon 样例。	①②③
ancs	苹果消息中心（ANCS）例程	①②③
bleI2C_RawPass	I2C 透传例程	①②③
bleSmartPeripheral	综合 BLE Peripheral 例程	①②③
bleUart-RawPass	UART 透传例程	①②③
eddystone	eddystone 例程	①②③
HIDKeyboard	HID 例程	①②③
hrs	Heart rate profile 演示例程	①②③
iBeacon	iBeacon 例程	①②③
otaDemo	最基本的 OTA 演示例程	①②③
pwmLight	通过 BLE 命令控制 PWM 进行 LED 亮度调整的例程	②③
RawAdv	简单的广播例程，用于无线胎压监测一类的应用	①②③
Sensor_Broadcast		①②③
wrist	综合样例，用于运动手环一类的应用	②
wrist_aptm	基于综合样例，通过 AP Timer+OSAL Timer 实现实时性较高的时钟	②
XIPDemo	部分实时性要求较低的代码直接运行于内部 flash 的例程	②③
OTA		
OTA_internal_flash	OTA bootloader	①②③
OTA_upgrade_2ndboot	特殊应用，用于升级 OTA bootloader 自身	①②③
peripheral		
adc	ADC 驱动应用样例	①②③
ap_timer	AP Timer 驱动应用样例	①②③
fs	文件系统样例	①②③
gpio	GPIO 演示例程	①②③
kscan	4x4 按键演示例程	②
lcd_ST7789VW	240x240 TFT 彩屏演示例程	①

pwm	PWM 演示例程	①②③
qdec	QDEC 演示例程	①②③
spiflash	外部 SPI 演示例程	②
voice	语音采集演示例程	②
voice_sbc	SBC 格式语音采集编码演示例程	②
watchdog	看门狗演示例程	①②③