PHY622X

EVB 用户开发手册

Version 1.0

Author: Bingliang.Lou

Security: Public

Date: 2020.12

PhyPlus

Revision History

Revision	Author	Date	Description
v1.0	Bingliang.Lou	2020.12	First Edition

目录

图表目录

1 简介

本文档用于说明如何使用 PHY62XX 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

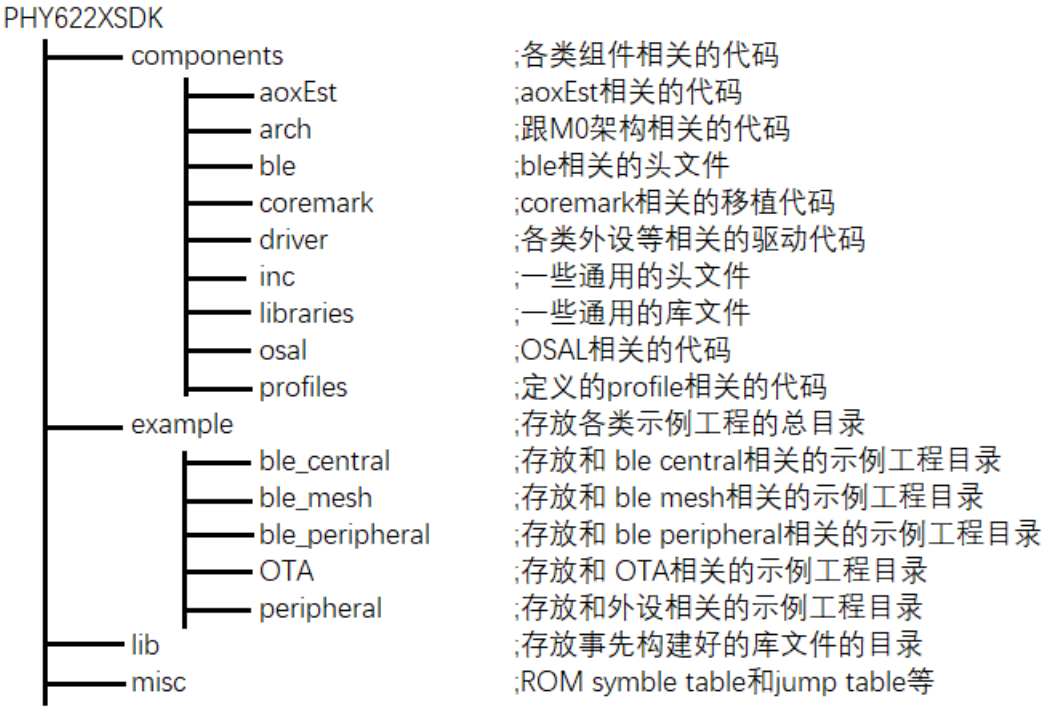


图 1：SDK 目录结构

2.2 开发板

PHY 为不同的芯片系列(PHY6222/6252，和 PHY6220 等)提供若干开发板，供用户根据特定的应用场合做功能验证。常用的有：

- **PHY6222/PHY6220 EVB 开发板资源介绍如下：**

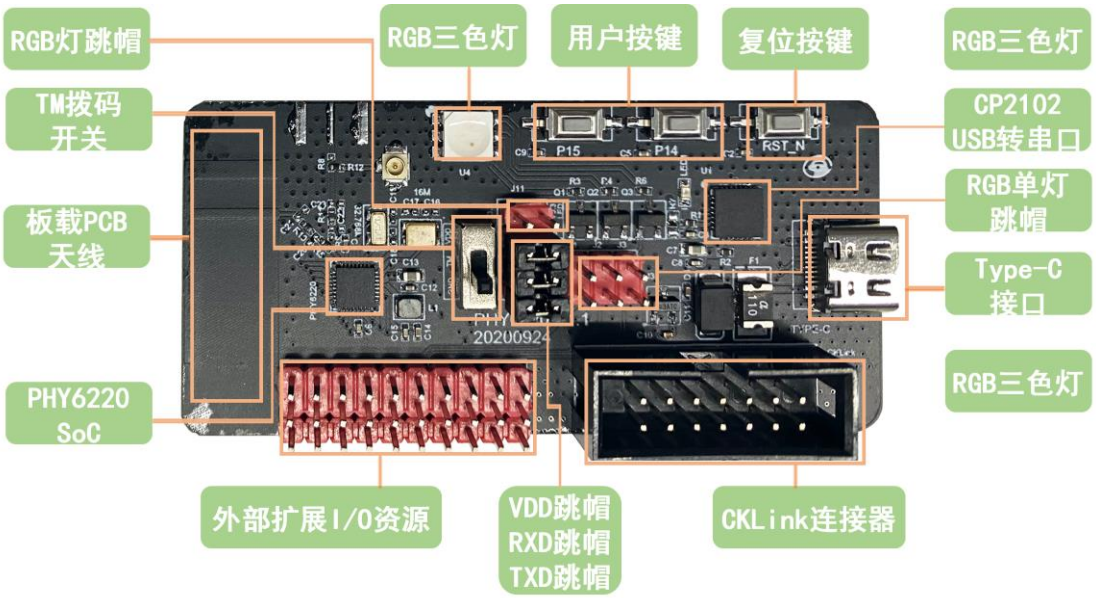


图 2：PHY622X EVB 开发板

序号	资源	说明
1	Type-C 接口	Type C 接口，可用于供电和串口输入输出
2	CP2102 USB 转串口	USB-UART 转换芯片
3	用户按键	普通 GPIO 按键（P14/P15），开发板已并连一个 1μF 电容接地
4	复位按键	芯片复位按键
5	外部扩展 I/O 资源	GPIO 引脚，定义见开发板背部
6	PHY6220 SoC	PHY6220 低功耗蓝牙 SoC
7	RGB 三色灯	RGB 三色灯
8	RGB 三色灯跳帽	三色灯跳帽，去除后断开三色灯的供电
9	RGB 单灯跳帽	单 LED 灯跳帽，去除后断开单个 LED 和 GPIO 连接
10	VDD 跳帽	VDD 跳帽，去除后断开芯片供电
11	UART TXD 跳帽	UART TXD 跳帽，去除后断开 UART 和 P10 连接
12	UART RXD 跳帽	UART RXD 跳帽，去除后断开 UART 和 P9 连接
13	TM 拨码开关	烧录模式选择开关，正常模式拨至 GND，烧录模式拨至 VDD
14	CKLink/JLink 连接器	JTAG 调试接口

表 1：PHY622X EVB 开发板资源清单

注意事项：

1. 开发板上电前，请确认 VDD/UART TXD/UART RXD 跳帽在位。
2. TM 拨码开关拨至 GND，复位开发板，进入工作模式。
3. TM 拨码开关拨至 VDD，同时需要确认 P24 和 P25 为低电平状态（默认为低电平），复位开发板，进入烧录模式。

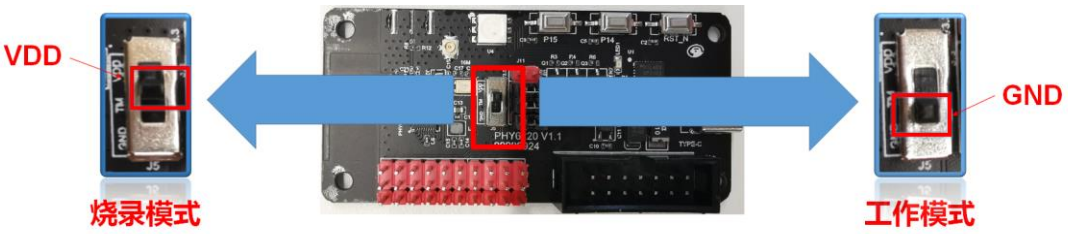
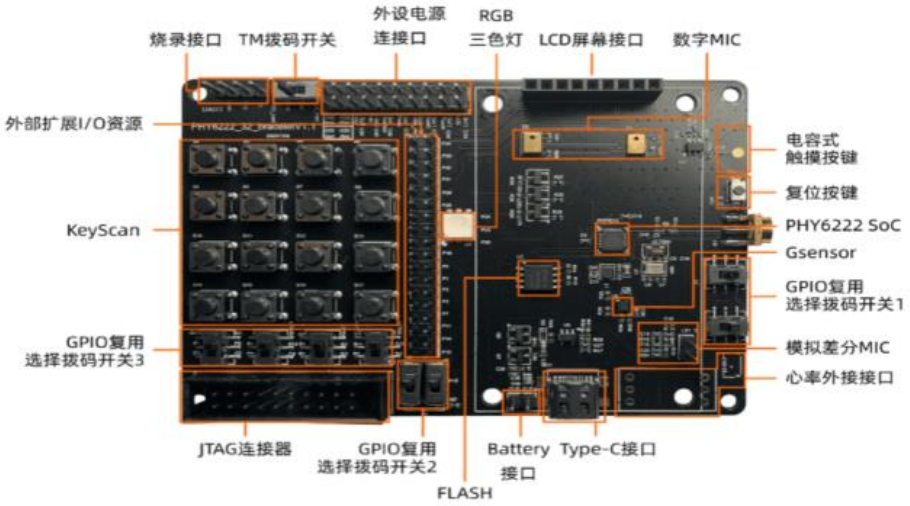


图 3：工作模式选择

- **PHY6222 手环板资源介绍如下：**



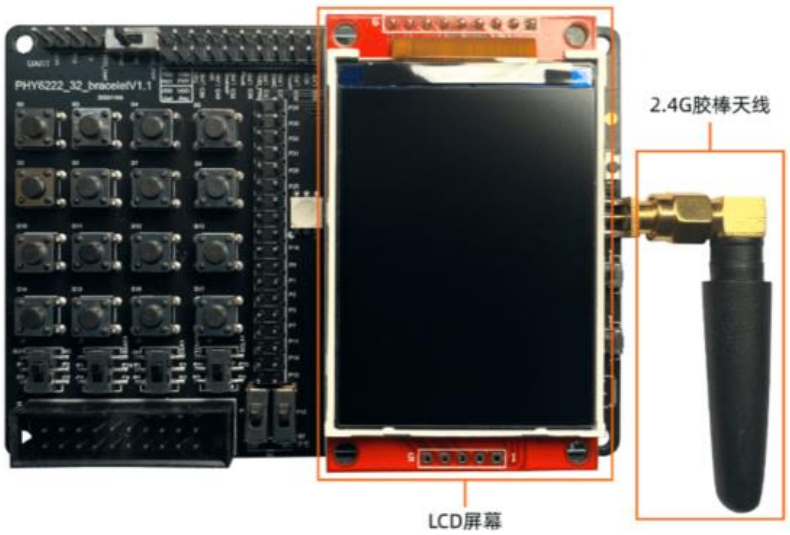


图 4：PHY6222 手环板

序号	资源	说明
1	Type-C 接口	Type C 接口，可用于供电和电池充电，不支持通信
2	Battery 接口	可外接锂电池，给锂电池供电，同时锂电池可以给开发板供电
3	GPIO 复用选择拨码开关 2	用于 GPIO 功能复用
4	Jtag 连接器	支持 SWD 在线调试
5	GPIO 复用选择拨码开关 3	用于 GPIO 功能复用
6	KeyScan	4X4 矩阵键盘
7	外部扩展 IO 资源	用于 SockGPIO 和外设连接
8	烧录接口	用于 PHY6222Sock 烧录
9	TM 拨码开关	用于 PHY6222Sock 烧录和运行选择
10	外设电源连接口	用于给板上外设供电，默认跳帽接上
11	RGB 三色灯	RGB 三色灯，控制脚 P15/P16/P17（需要通过拨码开关和跳帽选择）
12	LCD 屏幕接口	用于和 LCD 屏幕连接，屏幕已配。
13	数字 MIC	支持 DMic 输入
14	电容式触摸按键	电容式触摸按键，可用于中断
15	复位按键	用于 PHY6222 芯片复位
16	PHY6222SoC	低功耗蓝牙 SoC
17	Gsensor	SC7A20 三轴加速度 sensor
18	GPIO 复用选择拨码开关 1	用于 GPIO 功能复用
19	模拟差分 MIC	支持 AMIC 差分输入
20	心率外接接口	用于支持 I2C 的设备，不仅仅是心率

表 2：PHY6222 手环板资源清单

注意事项：

1. 开发板使用前，请确认 VDD/UARTTXD/UARTRXD 跳帽在位。
2. TM 拨码开关状态

- **PHY6252 EVB 板资源介绍如下：**

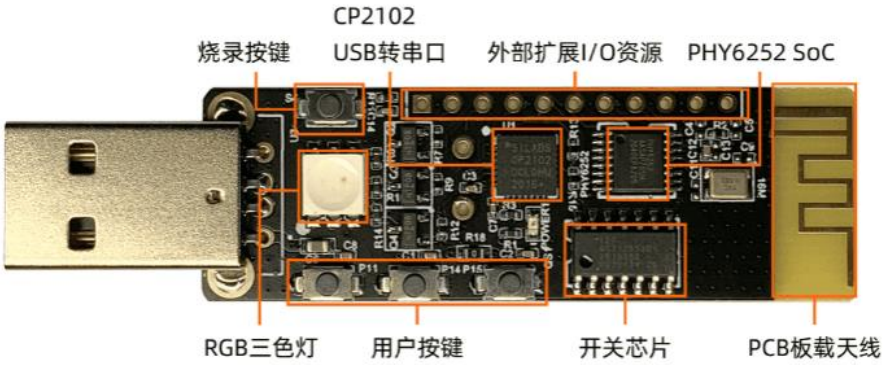


图 5：PHY6252 EVB 板

序号	资源	说明
1	USB 公头	USB 接口，用于供电和串口输入输出
2	CP2102USB-UART	USB-UART 转换芯片
3	用户按键	普通 GPIO 按键（P11/P14/P15）开发板已经并联一个 1uF 电容
4	烧录按键	按键用来使芯片进入烧录模式
5	外部扩展 I/O 资源	GPIO 引脚，引脚定义见开发板背部
6	PHY6222SoC	PHY6222 低功耗蓝牙 SoC
7	RGB 三色灯	RGB 三色灯,控制脚 P2/P3/P7，默认已经通过 0 欧姆电阻接好
8	QS3125S1G8 开关芯片	PHY6252 进烧录模式 reset 的开关
9	板载天线	默认接板载天线

表 3：PHY6252 EVB 开发板资源清单

注意事项：

1. 复位按键需要长按一段时间 1S 左右，否则可能无法进入烧录模式，或者烧录好程序后无法正常工作。

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 PlyPlusKit 工具删除开发板已经烧录的固件（PlyPlusKit 工具使用方法请参考工具使用指南：<PhyPlusKit User's Guide.pdf>）
- 从浏览器的 SDK 安装目录>example>ble_peripheral 选择一个样例，比如 simpleBlePeripheral，打开 MDK 项目文件

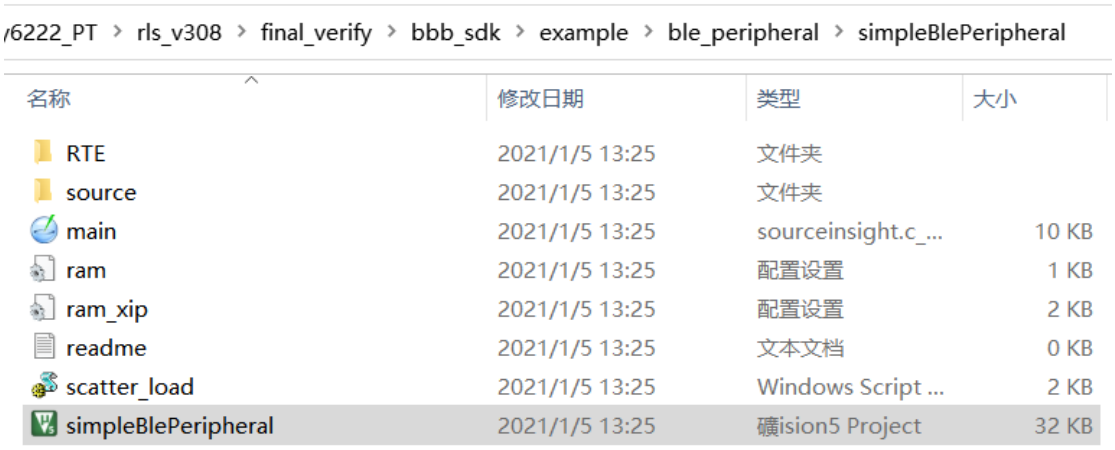


图 6：示例工程目录结构

- 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 7：工具栏上的 option 对话框

- 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：
  1. 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
  2. CFG_SLEEP_MODE=PWR_MODE_NO_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。
  3. DEBUG_INFO=1：使能调试信息，默认通过串口输出：P9(Tx),P10(Rx)
  4. DEBUG_INFO=0：关闭调试信息。

- 调试代码：

  根据在运行时是否有部分工程代码在 flash 中直接被调用执行（在被执行前无需事先将其从 flash 拷贝到 RAM），JTAG 调试方式可以分为 XIP 和 SRAM 两种方法。其中 XIP 方法对应的时有部分代码会在 flash 中直接被调用执行的情况；而 SRAM 方法对应的则为所有代码都会在被拷贝到 SRAM 后再执行的情况。用户可以根据在工程根目录下的 scatter_load.sct 文件的内容判断需要选择哪种方法。如果在 scatter_load.sct 文件的 LR_ROM_XIP region 中包含了有效的目标文件的话，用户需要采用 XIP 调试方法；否则的话，就应该采用 SRAM 调试方法。

  调试环境的搭建过程如下：

  1. 编译工程，获得固件文件。对于 XIP 调试方法，请使用 PhyPlusKit 工具先烧录好需要调试的固件，对于 SRAM 调试方法则跳过烧录固件这一步；
  2. 连接 JTAG 仿真器和电源线；

图 8：连接仿真器

  3. 用 keil 打开工程，打开工程的 Options -> Debug. 选择正确的仿真器类型后点击”Settings”按钮后确认仿真器能正确检测到如下：

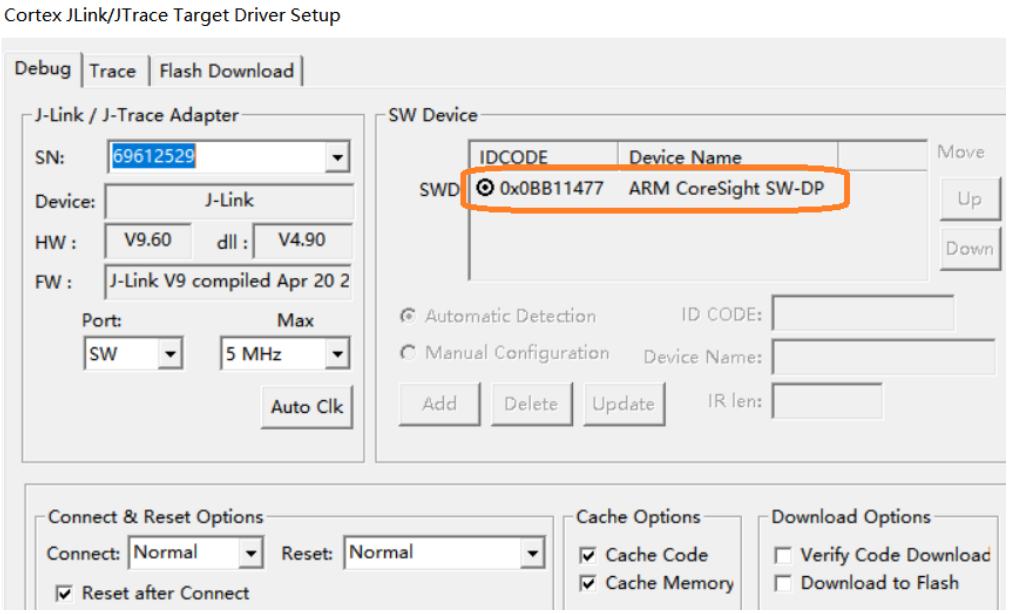


图 9：检测仿真器

4. 返回到 Debug 界面，根据实际情况选择初始化文件。SRAM 方法的选择 ram,ini 文件；XIP 方法的选择 ram_xip.ini 文件，这两个文件都在工程的根目录下。

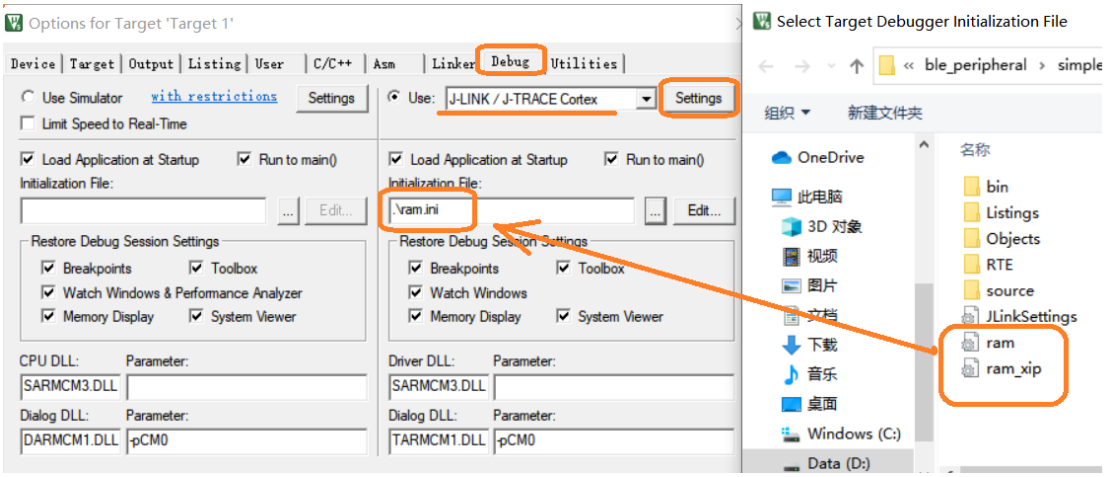


图 10：选择 ram 初始化文件

5. 在菜单栏或工具栏选择调试后就可开启调试功能

- 固件烧写：

PHY6222 烧录工具，支持通过串口进行镜像下载：

- 打开烧写工具 PhyPlusKit.exe；
- 勾选 UART Setting ，选择开发板串口，串口配置为波特率：115200，停止位：1，校验：NO；

- 点击 Connect ,连接串口；
- 选择 Flash_writer 标签页；
- 选择 HEX 烧入方式标签页；
- 双击选择 bin\目录下的 hex 文件；
- 下方选择 Single 标签，TYPE 选择 MAC，VALUE 填写 MAC 地址；
- 将开发板拨码开关拨到 VDD；
- 按开发板上的 RESET 按键，重启开发板，串口打印 UART RX : cmd>>；
- 点击 Erase 擦除；
- 点击 Write 烧写；
- LOG 区域显示烧录过程；
- 镜像烧录完成后，将开发板拨码开关拨到 GND，重启开发板即可看到打印的调试信息；

图 11：PhyPlusKit 使用窗口

注意：

1. Erase Size 下拉选项选择 512K 时，需要先点击 Erase 擦除 Flash 后，才能进行烧写。
2. Erase Size 下拉选项选择 HEXF 时，只需点击 Write 按钮，此时烧录工具会根据镜像文件内容完成部分 Flash 的擦除和烧录动作。

若需获悉更多细节，请参考 PhyPlusKit 使用文档。

3 平台和驱动

3.1 平台简介

PHY62XX 的固件架构分为三大部分：ROM，OTA Bootloader 和应用固件。其中 OTA Boot loader 为可选部分，两种启动模式的大致流程如下

OTA 启动模式：

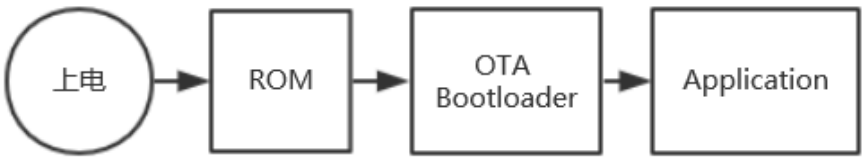


图 12：OTA 模式启动流程

非 OTA 启动模式：

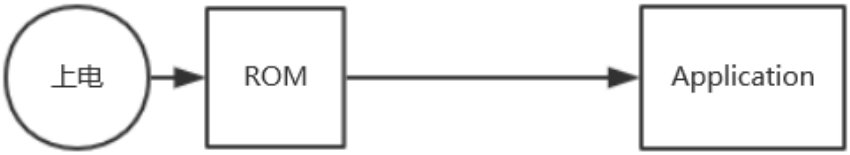


图 13：非 OTA 模式启动流程

- ROM：启动并引导 OTA 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

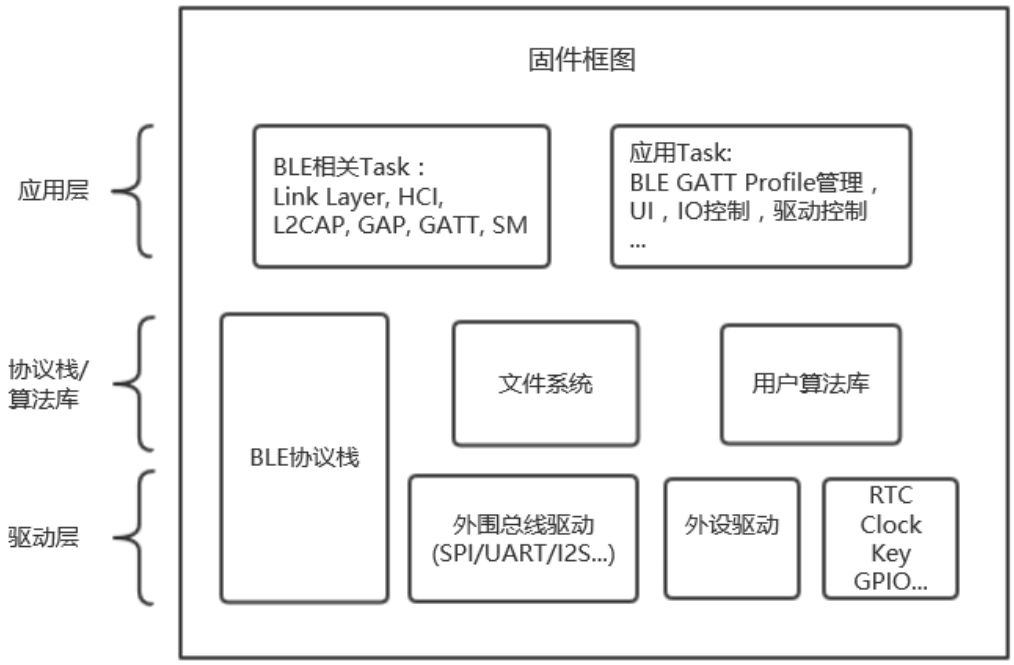


图 14：SDK 软件框图

PHY SDK 使用 OSAL 实现 OS 的部分功能，并在应用层引入了 task 的概念。每个 Task 分别包括一个初始化函数和一个事件处理函数。对于 BLE 应用，以下 Task 是必须的：

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

表 4：BLE 应用必须的 task 列表

如果需要支持 SMP 那么还需要加入 Bond Manager task：

Tasks (Task 初始化&Task 事件响应函数)	说明
GAPBondMgr_Init() GAPBondMgr_ProcessEvent(uint8, uint16)	用于对 SMP 的支持

表 5：BLE 应用支持 SMP 的 task

3.2.1 OSAL

PHY62XX SDK 中的 OSAL 实现了 OS 的部分功能，用户可以调用相应的 API 实现消息的收发，定时器，和堆管理等。在 components\osal\include\osal.h 中申明了支持的 API，其中常用的有：

1. uint8 osal_init_system(void)

用于初始化 Task system。

- 参数

无。

- 返回值

SUCCESS	成功

2. int8 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. 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>

4. 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>

5. 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

6. 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

7. 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

8. 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）

9. 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。

10. void *osal_memcpy(void *dst, const void GENERIC *src, unsigned int len)

将 len 个字节的数据从 src 拷贝到 dst 所在的地址处。

- 参数

类型	参数名	说明
void *	dst	目的字符串地址
const void *	src	源字符串地址
unsigned int	len	要拷贝的数据长度

- 返回值

无。

11. uint8 osal_memcmp(const void GENERIC *src1, const void GENERIC *src2, unsigned int len);

比较字符串 src1 和 src2 的前 len 个字符内容是否相同。

- 参数

类型	参数名	说明
const void GENERIC *	src1	目的字符串地址
const void GENERIC *	src2	源字符串地址
unsigned int	len	要比较的数据长度

- 返回值

TRUE	src1 和 src2 的前 len 个字符串内容相同
FALSE	src1 和 src2 的前 len 个字符串内容不相同

12. void *osal_memset(void *dest, uint8 value, int len)

将字符串 dest 的前 len 个字节内容设置为 value。

- 参数

类型	参数名	说明
void *	dest	字符串地址
uint8	value	要设置的值
int	len	要设置的字节数

- 返回值

无。

3.3 CACHE

为了提高 flash 的读写速度，PHY62XX 在 Bus Matrix 和 SPI Flash 之间挂接了一个 CACHE 模块，如下图：

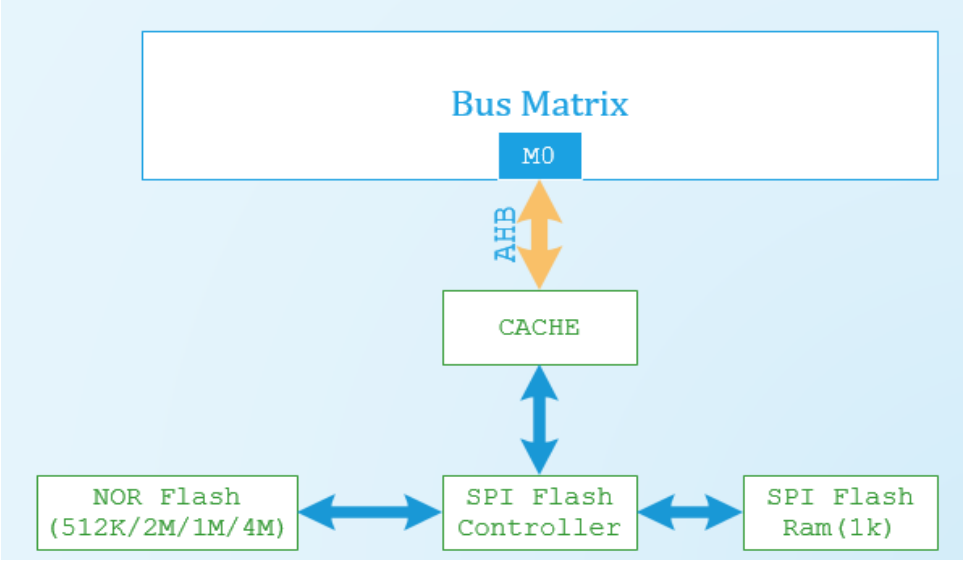


图 15：CACHE 模块

应用程序可以根据需要设置是否使用 CACHE(通过 AP_PCR->CACHE_BYPASS 来设置)。在使用 CACHE 的情况下，应用程序在读写 Flash 时需要调用相应的接口或宏定义，以确保读写数据的正确性和一致性。

3.3.1 APIs

PHY62XX SDK 在 components\driver\flash 目录下提供了跟 CACHE 相关的 API 和宏定义如下：

1. int hal_spif_cache_init(xflash_Ctx_t cfg);

初始化 cache 相关的变量和设置。

- 参数

类型	参数名	说明
xflash_Ctx_t	cfg	包括 spif 时钟频率和 flash 读指令

- 返回值

PPlus_SUCCESS	配置成功

2. void hal_cache_tag_flush(void);

清空 CACHE 中的数据，通常在初始化或 cache 使用模式切换时使用。

- 参数

无。

- 返回值

无。

3. HAL_CACHE_ENTER_BYPASS_SECTION()

这个宏定义用于切换为不使用 CACHE 模式时使用。

4. HAL_CACHE_EXIT_BYPASS_SECTION ()

这个宏定义用于切换为使用 CACHE 模式时使用。

常用的示例代码如下：

```
xflash_Ctx_t cfg =
{
    .spif_ref_clk    =    SYS_CLK_DLL_64M,
    .rd_instr        =    XFRD_FCMD_READ_DUAL
};
hal_spif_cache_init(cfg);
LOG_INIT();
hal_gpio_init();
hal_fs_init(0x1103C000,2);
```

3.4 硬件驱动

本章节介绍硬件模块驱动和外部总线驱动。主要包括 ADC, GPIO, UART, SPI, I2C, PWM, TIMER,KSCAN, DMA, clock 和 watchdog 等。在 PHY62XX SDK 的 example\peripheral 目录下有相应的示例工程可以参考。

3.4.1 模块 ID

PHY62XX SDK 定义的模块 ID 如下：

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。

表 6：模块 ID 列表

3.4.2 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，系统时钟源选择，和 32K 时钟源选择等操作。

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。

系统时钟源选择

SYS_CLK_RC_32M	选择内部 RC_32M 作为系统时钟源。
SYS_CLK_DLL_32M	选择内部 DLL_32M 作为系统时钟源。
SYS_CLK_XTAL_16M	选择外部 XTAL_16M 作为系统时钟源。
SYS_CLK_DLL_48M	选择内部 DLL_48M 作为系统时钟源。
SYS_CLK_DLL_96M	选择内部 DLL_96M 作为系统时钟源。
SYS_CLK_8M	选择内部 8M 分频作为系统时钟源。
SYS_CLK_4M	选择内部 4M 分频作为系统时钟源。

常用的 API 如下：

1. void hal_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。

2. void hal_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. int hal_clk_gate_get (MODULE_e module);

获取硬件模块的当前时钟设置。

- 参数

类型	参数名	说明
MODULE_e	module	硬件模块 ID

- 返回值

int	对应模块的时钟设置值

4. void hal_clk_reset(MODULE_e module);

复位硬件模块的时钟。

- 参数

类型	参数名	说明
MODULE_e	module	硬件模块 ID

- 返回值

无。

5. void hal_rtc_clock_config(CLK32K_e clk32Mode);

设置 RTC 的时钟源。

- 参数

类型	参数名	说明
CLK32K_e	clk32Mode	可选的值为 CLK_32K_XTAL 或 CLK_32K_RCOSC

- 返回值

无。

6. int clk_init(sysclk_t h_system_clk_sel);

系统时钟和 SPIF 时钟初始化。

- 参数

类型	参数名	说明
sysclk_t	h_system_clk_sel	可选的值为 SYS_CLK_XTAL_16M 等

- 返回值

PPlus_SUCCESS	设置完成

3.4.3 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_low_power_io_init()函数进行配置，请参考 components\driver\pwrmgr\pwrmgr.c 文件中的如下 API：hal_pwrmgr_RAM_retention(RET_SRAM0|RET_SRAM1|RET_SRAM2);

hal_pwrmgr_RAM_retention_set(void)

PHY62XX 总共有 3 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	16K Byte	1FFF_8000~1FFF_BFFF	SRAM1 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	16K Byte	1FFF_C000~1FFF_FFFF	SRAM2 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

表 7：SRAM 列表

3.4.4 GPIO

关于 GPIO 模块的配置和使用，请参考《PHY62XX_GPIO_Application_Note.pdf》文档。相应的示例工程在 example\peripheral\gpio 目录下。

3.4.5 ADC

关于 ADC 模块的配置和使用，请参考《PHY62XX_ADC_Application_Note.pdf》文档。相应的示例工程在 example\peripheral\adc 目录下。

3.4.6 其它 peripharal

除了上面的 GPIO 和 ADC, PHY62XX SDK 还提供了一些其它的外设驱动和示例工程，请参考《PHY62XX_Peripheral_Application_Note.pdf》文档。相应的示例工程在 example\peripheral 目录下。

3.5 低功耗管理

为了方便应用实现低功耗管理，PHY62XX 支持四类功耗模式：普通模式，CPU 休眠模式，深度休眠模式，和关机模式。关于低功耗相关的配置和使用，请参考《PHY62XX PWR MGR 应用指南.pdf》文档。

3.6 文件系统 FS

为了方便应用存取内部 flash 中的数据，PHY62XX 实现了一个轻量级文件系统，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。跟 FS 相关的 API 接口函数在 components\osal\snv\osal_snv.c 文件和 components\libraries\fs\fs.c 中实现。

常用的 API 如下：

1. int hal_fs_init(uint32_t fs_start_address,uint8_t sector_num);

用于初始化文件系统，该函数需要在系统初始化时候设置。

- 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>

2. int hal_fs_item_read(uint16_t id,uint8_t* buf,uint16_t buf_len,uint16_t* len);

从 FS 文件读取数据。

- 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. int hal_fs_item_write(uint16_t id,uint8_t* buf,uint16_t len);

写入数据到 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	参考<error.h>

4. uint32_t hal_fs_get_free_size(void);

获取文件系统中剩余字节空间大小。

- 参数

无。

- 返回值

uint32_t	FS 可用存储文件空间，单位为字节

5. int hal_fs_get_garbage_size(uint32_t* garbage_file_num);

获取文件系统中文件所占空间大小，以字节为单位。

- 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 中已删除文件所占空间大小，单位为字节。

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	初始化成功
其他	参考<error.h>

7. int hal_fs_garbage_collect(void);

实现文件系统的垃圾回收机制，将 FS 中已经删除的文件所占用的空间释放。

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	参考<error.h>

8. int hal_fs_format (uint32_t fs_start_address,uint8_t sector_num);

格式化 FS，文件系统中所有文件会被擦除，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>

9. bool hal_fs_initialized(void);

查询 FS 是否已经初始化。

- 参数

无。

- 返回值

true	已初始化，可以使用。
false	未初始化，不能使用。

10. uint8 osal_snv_read(osalSnvId_t id, osalSnvLen_t len, void* pBuf);

从文件系统中读取文件内容。

- 参数

类型	参数名	说明
osalSnvId	id	文件 ID。
osalSnvLen_t	len	文件中有效数据长度。
void*	pBuf	用于保存文件中读取的数据的地址

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

11. uint8 osal_snv_write(osalSnvId_t id, osalSnvLen_t len, void* pBuf);

写入数据到文件系统的文件中。该函数内部实现了垃圾回收机制，建议应用程序采用osal_开头的 read/write 函数来实现对 flash 中文件的读写。

- 参数

类型	参数名	说明
osalSnvId	id	文件 ID。
osalSnvLen_t	len	文件中有效数据长度。
void*	pBuf	要写入的数据的地址

- 返回值

PPlus_SUCCESS	成功。
其他数值	参考<error.h>

3.7 日期时间

为了方便应用获取精确的时间，PHY62XX 实现万年历功能. datetime_t 为精确到秒的时间，datetime 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 用于时间获取和设置。其实现代码在 components\libraries\datetime 目录下。

3.7.1 USE_SYS_TICK 宏

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。

3.7.2 datetime_t 数据结构

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。

3.7.3 相应的 APIs

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。

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（默认为 30S）。不需要精确的调用。

- 参数

无。

- 返回值

无。

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>

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>

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，那么返回值为负数。

4 BLE

4.1 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 层总是作为下面四种角色之一：

- 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定义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 bleUart_AT 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。

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 的客户端特性配置描述符(Client Characteristic Configuration Descriptor)。

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
    {
        { ATT_BT_UUID_SIZE, characterUUID },
        GATT_PERMIT_READ,
        0,
        &bleuart_PTCharProps
    },

    // 特征值 1：值
    {
        { ATT_BT_UUID_SIZE, bleuart_PTCharUUID },
        GATT_PERMIT_WRITE,
        0,
        &bleuart_PTCharValue[0]
    },

    // 特征值 2：客户端特性配置描述符
    {
        { ATT_BT_UUID_SIZE, clientCharCfgUUID },
        GATT_PERMIT_READ|GATT_PERMIT_WRITE,
        0,
        (uint8*)& bleuart_TxCCCD
    },

};
```

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。

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。

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。

static void bleuart_HandleConnStatusCB (uint16 connHandle, uint8 changeType)

回调函数，响应 link status 的改变。

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

函数，用于发送 Notify 给 Central。

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 bleUart_AT 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，PHY62XX 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 运行模式

对于支持 OTA 的 PHY62XX 设备，会有以下三种运行模式：

- **应用模式**

通常情况下，设备会运行在应用模式下。

- **OTA 模式**

OTA 模式下，手机能够通过无线进行应用升级

- **OTA resource 模式**

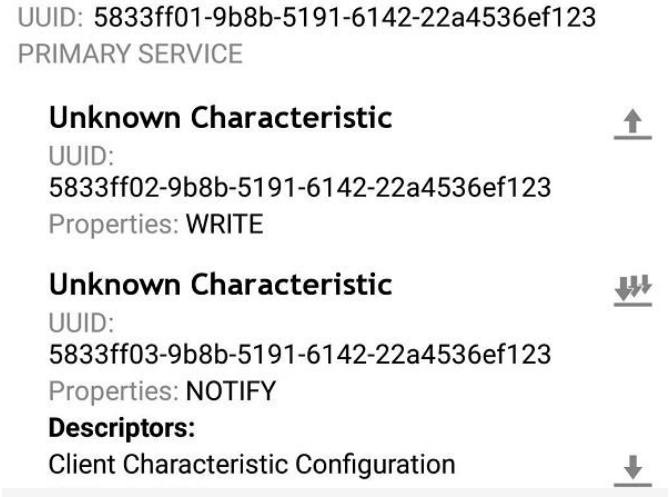
OTA resource 模式下，手机能够通过无线进行资源文件的升级。

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。



OTA App Service 代码位于 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，对于不同 OTA 模式，应用固件不需要做任何调整：

SLB OTA						
	256KB Flash			512KB Flash		
Reserved By PhyPlus	0	1FFF	8	0	1FFF	8
1st Boot info	2000	2FFF	4	2000	2FFF	4
2nd Boot info	3000	3FFF	4	3000	3FFF	4
FCDS	4000	4FFF	4	4000	4FFF	4
SLB Bootloader	5000	7FFF	12	5000	7FFF	12
App (Map to Sram)	8000	16FFF	60	8000	16FFF	60
XIP	17000	22FFF	48	17000	32FFF	112
FS(UCDS)	23000	24FFF	8	33000	36FFF	16
Resource	25000	24FFF	0	37000	54FFF	120
FW Storage	25000	3FFFF	108	55000	7FFFF	172

表 8：SLB OTA

Single Bank OTA							
	256KB Flash			512KB Flash			
Reserved By PhyPlus	0	1FFF	8	0	1FFF	8	
1st Boot info	2000	2FFF	4	2000	2FFF	4	Boot loader Info
2nd Boot info	3000	3FFF	4	3000	3FFF	4	App Info
FCDS	4000	4FFF	4	4000	4FFF	4	FCDS
OTA Bootloader	5000	10FFF	48	5000	10FFF	48	Bootloader
App Bank	11000	1FFFF	60	11000	1FFFF	60	App Sram
XIP	20000	3BFFF	112	20000	33FFF	80	
FS(UCDS)	3C000	3DFFF	8	34000	35FFF	8	FlashNVM
Resource	3E000	3FFFF	8	36000	7FFFF	296	
FW Storage	40000	3FFFF	0	80000	7FFFF	0	FW Storage

表 9：Single Bank OTA

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。

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，请参考示例工程：example\OTA\OTA_internal_flash 目录

4.3.7 烧写应用固件和 OTA bootloader

通过 PhyPlusKit 工具或者 Phyplus 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 PhyPlusKit 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 PhyPlusKit 或者 PlyPlus 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 PHYApp 进行应用固件或者资源文件（比如字库文件）的升级更新。