Application Note:
Lenze 17H25 SDK 开发指南

AN-SDK-UGE1

Ver 2.0

2013/09/22

Keyword:

本文档为 Lenze 17H25 BLE SDK 3.2.0 的开发指南。

Brief:

This application note is the user guide for Lenze 17H25 SDK.

Lenze technology co., LTD

Lenze technology co., LTD

Published by
Lenze technology co., LTD

Room C,26th FIoor,West Speclal Zone PressBuilding,NO,6008Shennan Road,Futian District Shenzhen

© Lenze technology co., LTD
All Right Reserved

Legal Disclaimer

Lenze technology co., LTD reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Lenze technology co., LTD disclaims any and all liability for any errors, inaccuracies or incompleteness contained herein or in any other disclosure relating to any product.

Lenze technology co., LTD does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights, nor the rights of others

The products shown herein are not designed for use in medical, life-saving, or life-sustaining applications. Customers using or selling Lenze technology co., LTD products not expressly indicated for use in such applications do so entirely at their own risk and agree to fully indemnify Lenze technology co., LTD for any damages arising or resulting from such use or sale.

Information:

For further information on the technology, product and business term, please contact Lenze technology co., LTD Company (www.Lenze tech.com).

For sales or technical support, please send email to the address of:

Lenze Technology Co.,LTD

Revision History

Version	Major Changes	Date	Author
1.0	Initial version	2013/9	Z.J.Q. ; Z.X.D;Caroline

Lenze Technology Co.,LTD

目录

目录.....4
1 SDK 介绍.....14
1.1 软件组织架构.....14
1.1.1 main.c.....15
1.1.2 app_config.h.....16
1.1.3 application file.....16
1.1.4 BLE stack entry.....16
1.2 IC 介绍.....17
1.3 Driver 介绍.....17
1.4 bootloader 介绍.....19
1.5 library 介绍.....20
1.5.1 根据功能分类.....20
1.5.2 根据 system clock 分类.....20
1.5.3 其他特殊的 library.....20
1.6 Demo 介绍.....20
1.6.1 BLE slave demo.....22
1.6.2 BLE master demo.....22
1.6.3 feature demo and driver demo.....22
2 MCU 基础模块.....22
2.1 MCU 地址空间.....22
2.1.1 MCU 地址空间分配.....22
2.1.2 Sram 空间分配.....23
2.1.2.1 Sram 和 Firmware 空间.....23
2.1.2.2 list 文件分析 demo.....28
2.1.3 MCU 地址空间访问.....31
2.1.3.1 外设空间的读写操作.....31
2.1.3.2 flash 空间的操作.....32
2.1.4 SDK FLASH 空间的分配.....34
2.1.4.1 512K Flash 的地址分配.....34
2.2 时钟模块.....36
2.2.1 系统时钟配置方法.....37
2.2.2 system tick 的使用.....38
2.3 GPIO 模块.....39
2.3.1 GPIO 定义.....39
2.3.2 GPIO 状态控制.....39
2.3.3 GPIO 的初始化.....41
2.3.4 配置 SWS 上拉防止死机.....42

Lenze Technology Co.,LTD

3 BLE 模块.....42
3.1 BLE SDK 软件架构.....42
3.1.1 标准 BLE SDK 软件架构.....42
3.1.2 Lenze BLE SDK 软件架构.....44
3.1.2.1 Lenze BLE controller.....44
3.1.2.2 Lenze BLE slave.....45
3.2 BLE controller.....46
3.2.1 BLE controller 介绍.....46
3.2.2 Link Layer 状态机.....47
3.2.3 Link Layer 状态机组合应用.....49
3.2.3.1 Link Layer 状态机初始化.....49
3.2.3.2 Idle + Advertising.....49
3.2.3.3 Idle + Scannning.....50
3.2.3.4 Idle + Advtersing + ConnSlaveRole.....51
3.2.4 Link Layer 时序.....52
3.2.4.1 Idle state 时序.....52
3.2.4.2 Advertising state 时序.....53
3.2.4.3 Scanning state 时序.....53
3.2.4.4 Initiating state 时序.....54
3.2.4.5 Conn state Slave role 时序.....54
3.2.4.6 Conn state Slave role 时序保护.....55
3.2.5 Link Layer 状态机扩展.....56
3.2.5.1 Scanning in Advertising state.....56
3.2.5.2 Scanning in ConnSlaveRole.....57
3.2.5.3 Advertising in ConnSlaveRole.....57
3.2.5.4 Advertising and Scanning in ConnSlaveRole.....58
3.2.6 Link Layer TX fifo & RX fifo.....58
3.2.6.1 Slave role fifo.....59
3.2.7 Lenze defined event.....59
3.2.7.1 BLT_EV_FLAG_ADV.....62
3.2.7.2 BLT_EV_FLAG_ADV_DURATION_TIMEOUT.....62
3.2.7.3 BLT_EV_FLAG_SCAN_RSP.....62
3.2.7.4 BLT_EV_FLAG_CONNECT.....62
3.2.7.5 BLT_EV_FLAG_TERMINATE.....63
3.2.7.6 BLT_EV_FLAG_ENCRYPTION_CONN_DONE.....63
3.2.7.7 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE.....64
3.2.7.8 BLT_EV_FLAG_GPIO_EARLY_WAKEUP.....64
3.2.7.9 BLT_EV_FLAG_CHN_MAP_REQ.....65
3.2.7.10 BLT_EV_FLAG_CHN_MAP_UPDATE.....65
3.2.7.11 BLT_EV_FLAG_CONN_PARA_REQ.....65
3.2.7.12 BLT_EV_FLAG_CONN_PARA_UPDATE.....66
3.2.7.13 BLT_EV_FLAG_SUSPEND_ENETR.....66
3.2.7.14 BLT_EV_FLAG_SUSPEND_EXIT.....66
3.2.7.15 BLT_EV_FLAG_READ_P256_KEY.....66

Lenze Technology Co.,LTD

3.2.7.16 BLT_EV_FLAG_GENERATE_DHKEY 66
3.2.8 Controller API 66
3.2.8.1 API 返回类型 ble_sts_t 66
3.2.8.2 MAC address 初始化 67
3.2.8.3 Link Layer 状态机初始化 67
3.2.8.4 bls_ll_setAdvData 67
3.2.8.5 bls_ll_setScanRspData 68
3.2.8.6 bls_ll_setAdvParam 69
3.2.8.7 bls_ll_setAdvEnable 73
3.2.8.8 bls_ll_setAdvDuration 73
3.2.8.9 blc_ll_setAdvCustomedChannel 74
3.2.8.10 rf_set_power_level_index 74
3.2.8.11 blc_ll_setScanParameter 75
3.2.8.12 blc_ll_setScanEnable 76
3.2.8.13 blc_ll_createConnection 76
3.2.8.14 blc_ll_setCreateConnectionTimeout 77
3.2.8.15 blm_ll_updateConnection 78
3.2.8.16 bls_ll_terminateConnection 79
3.2.8.17 blm_ll_disconnect 80
3.2.8.18 Get Connection Parameters 80
3.2.8.19 blc_ll_getCurrentState 80
3.2.8.20 blc_ll_getLatestAvgRSSI 80
3.2.8.21 Whitelist & Resolvinglist 81
3.3 L2CAP 82
3.3.1 注册 L2CAP 数据处理函数 83
3.3.2 更新连接参数 83
3.3.2.1 slave 请求更新连接参数 83
3.3.2.2 master 回应更新申请 84
3.3.2.3 master 在 Link Layer 上更新连接参数 86
3.4 ATT & GATT 86
3.4.1 GATT 基本单位 Attribute 86
3.4.2 Attribute and ATT Table 88
3.4.2.1 attNum 89
3.4.2.2 perm 89
3.4.2.3 uuid、uuidLen 89
3.4.2.4 pAttrValue、attrLen 90
3.4.2.5 回调函数 w 91
3.4.2.6 回调函数 r 93
3.4.2.7 Attribute Table 结构 94
3.4.2.8 ATT table Initialization 95
3.4.3 Attribute PDU & GATT API 95
3.4.3.1 Read by Group Type Request、Read by Group Type Response 96
3.4.3.2 Find by Type Value Request、Find by Type Value Response 97
3.4.3.3 Read by Type Request、Read by Type Response 97

Lenze Technology Co.,LTD

3.4.3.4 Find information Request、Find information Response..... 98
3.4.3.5 Read Request、Read Response.....98
3.4.3.6 Read Blob Request、Read Blob Response..... 99
3.4.3.7 Exchange MTU Request、Exchange MTU Response..... 99
3.4.3.8 Write Request、Write Response.....100
3.4.3.9 Write Command.....102
3.4.3.10 Handle Value Notification.....102
3.4.3.11 Handle Value Indication.....102
3.4.3.12 Handle Value Confirmation.....103
3.4.4 17H25 master GATT..... 104
3.5 SMP.....106
3.5.1 SMP 参数配置..... 106
3.5.1.1 设备绑定.....106
3.5.1.2 设备带外数据验证(OOB)..... 106
3.5.2 SMP 使能.....107
3.5.3 SMP event.....108
3.5.3.1 BLT_EV_FLAG_PAIRING_BEGIN.....108
3.5.3.2 BLT_EV_FLAG_PAIRING_END..... 108
3.5.4 SMP 绑定信息说明..... 109
3.6 Flash 存储方法设计.....110
3.6.1 Slave mac table..... 111
3.6.2 API..... 112
3.6.2.1 user_tbl_slave_mac_add..... 112
3.6.2.2 user_tbl_slave_mac_search.....112
3.6.2.3 user_tbl_slave_mac_delete_by_adr.....112
3.6.2.4 user_tbl_slave_mac_delete_by_index..... 112
3.6.2.5 user_tbl_slave_mac_delete_all..... 112
3.6.2.6 user_tbl_salve_mac_unpair_proc..... 113
3.6.3 连接和配对..... 113
3.6.4 解配对.....114
4 低功耗管理（PM）..... 115
4.1 低功耗 driver..... 115
4.1.1 低功耗模式..... 115
4.1.2 低功耗硬件唤醒源..... 115
4.1.3 低功耗模式的进入和唤醒..... 117
4.2 BLE 低功耗管理..... 118
4.2.1 PM in Idle state..... 118
4.2.2 PM in BLE Adv state & Conn state..... 119
4.3 BLE 低功耗管理配置方法.....119
4.3.1 PM 模块初始化.....119
4.3.2 设置低功耗模式 bls_pm_setSuspendMask..... 120
4.3.3 设置低功耗唤醒源 bls_pm_setWakeupSource..... 120

Lenze Technology Co.,LTD

4.3.4 低功耗管理工作机制 121
4.4 Latency use 的配置与计算 123
4.5 其他 API 123
4.5.1 bls_pm_getSystemWakeupTick 124
4.5.2 bls_pm_enableAdvMcuStall 125
4.6 GPIO 唤醒的注意事项 125
4.6.1 唤醒电平有效时无法进入 suspend/deepsleep 125
4.7 BLE 系统低功耗管理参考 126
4.8 应用层定时唤醒 128
5 OTA 129
5.1 17H25 Flash 架构设计和 OTA 流程 129
5.1.1 17H25 FLASH 存储架构 129
5.1.2 17H25 OTA 更新流程 130
5.1.3 cstartup.S、reset、reboot、代码搬移等细节问题 131
5.1.3.1 启动代码 cstartup.S 对 boot_flag 检测的处理 131
5.1.3.2 Firmware size 132
5.1.3.3 reset 和 reboot 133
5.1.4 修改 FLash 存储架构 133
5.1.4.1 修改 Firmware size 和 OTA 程序存放地址 134
5.1.4.2 修改 OTA boot bin 存放地址 134
5.1.4.3 修改 OTA boot flag 地址 134
5.1.5 修改 FLash 存储架构 135
5.1.5.1 修改 Firmware size 和 OTA 程序存放地址 135
5.1.5.2 修改 OTA boot flag 地址 135
5.2 OTA 模式 RF 数据处理 136
5.2.1 Slave 端 Attribute Table 中 OTA 的处理 136
5.2.2 OTA 数据 packet 格式 136
5.2.3 master 端 RF transform 处理方法 137
5.2.4 slave 端 RF receive 处理方法 140
6 按键扫描 142
6.1 Key matrix 142
6.2 Keyscan、keymap 和 keycode 144
6.2.1 Keyscan 144
6.2.2 Keymap &kb_event 145
6.3 Keycode 147
6.4 Keyscan flow 148
6.4.1 基本的 keyscan flow 148
6.4.2 Keyscan flow 时序优化 150
6.5 Deepsleep wakeup fast keyscan 151
6.6 卡键处理 153

Lenze Technology Co.,LTD

6.7 长按键功耗优化.....154
7 LED 管理.....154
7.1 LED 任务相关调用函数.....154
7.2 LED 任务的配置和管理.....155
7.2.1 定义 led event.....155
7.2.2 Led event 的优先级.....156
8 blt software timer.....156
8.1 timer 初始化.....157
8.2 Timer 的查询处理.....157
8.3 添加定时器任务.....159
8.4 删除定时器任务.....159
8.5 Demo.....160
9 Drivers in BLE SDK.....162
9.1 12M/16M 晶体外部电容.....162
9.2 外部 32k 晶振.....162
9.3 PA.....162
9.4 PWM.....163
9.5 UART.....163
9.5.1 UART GPIO.....163
9.5.2 UART 配置.....164
9.5.2.1 UART 通用配置.....164
9.5.2.2 UART 专有配置.....165
9.5.3 DMA 模式下 UART 收发数据.....167
9.5.3.1 DMA 模式下 UART 数据的接收.....167
9.5.3.2 DMA 模式下 UART 数据的发送.....167
9.5.4 非 DMA 模式下 UART 收发数据.....168
9.5.4.1 非 DMA 模式下 UART 数据的接收.....168
9.5.4.2 非 DMA 模式下 UART 数据的发送.....168
9.5.5 UART 波特率计算工具.....168
9.6 ADC.....171
9.6.1 ADC Clock.....171
9.6.2 ADC 配置.....171
9.6.2.1 17H25 ADC 初始化.....171
9.6.3 ADC 转化值的获取.....173
9.6.3.1 17H25 实际电压值计算.....174
9.7 低电压检测.....174
9.7.1 低电压检测的实现.....174
9.7.1.1 17H25 低电压检测.....174
9.7.2 低电压检测 demo.....175
9.8 IIC.....176
9.8.1 IIC Pin.....176

Lenze Technology Co.,LTD

9.8.2 IIC 配置 177
9.8.2.1 IIC Master 初始化 177
9.8.2.2 IIC Slave 初始化 178
9.8.3 IIC 数据的传输 178
9.8.3.1 IIC Master 写数据 178
9.8.3.2 IIC Master 读数据 179
9.8.3.3 IIC Slave 收发数据 180
9.8.4 IIC interrupt 180
9.9 SPI 180
9.9.1 SPI Pin 180
9.9.2 SPI 配置 182
9.9.2.1 SPI Master 初始化 182
9.9.2.2 SPI Slave 初始化 182
9.9.3 SPI 数据的传输 183
9.9.3.1 SPI Master 写数据 183
9.9.3.2 SPI Master 读数据 183
9.9.3.3 SPI Slave 收发数据 184
9.9.4 SPI interrupt 184
9.10 EMI 184
9.10.1 EMI Test 184
9.10.1.1 Emi 初始化设置 185
9.10.1.2 Power level and Channel 186
9.10.1.3 Emi Carrier Only 187
9.10.1.4 Emi Carrier Data 187
9.10.1.5 Emi TX 187
9.10.1.6 EMI RX 187
9.10.1.7 配置参数设置 189
9.10.2 EMI Test Tool 190
9.11 PHY test 194
10 红外 194
10.1 PWM Driver 194
10.1.1 PWM id and pin 194
10.1.2 PWM clock 195
10.1.3 PWM cycle and duty 195
10.1.4 PWM revert 196
10.1.5 PWM mode 196
10.1.6 PWM start and stop 197
10.1.7 PWM pulse number 198
10.1.8 PWM phase 198
10.1.9 PWM interrupt 198
10.2 IR 实现方法 199
10.3 IR Demo 详解 200

Lenze Technology Co.,LTD

10.3.1 NEC IR....200
10.3.2 设置载波....201
10.3.3 设置 logic1 和 logic0 的时间....201
10.3.4 配置一个完整的 NEC IR....202
10.3.5 添加时序信号....203
10.3.6 添加数据....204
10.3.7 NEC IR send....204
10.3.8 NEC IR repeat....205
10.3.9 中断的处理....205
10.3.10 应用层判断 IR busy....206
11 附录....206

Lenze Technology Co.,LTD

图目录

图 1-1 SDK 文件结构....15

图 1-2 pm 和 rf driver.....17
图 1-3 adc/gpio driver 和 bootloader.....18
图 1-4 i2c/spi/uart 等 driver.....18
图 1-5 i2c driver 中区分 17H25 和其他.....19
图 1-6 选择 bootloader.....19
图 2-1 MCU 地址空间分配.....23
图 2-2 Sram 空间和 Firmware 空间.....24
图 2-3 list 文件 Section 统计.....28
图 2-4 list 文件 section 地址.....29
图 2-5 512K FLASH 地址分配.....34
图 2-7 修改 lib 库方法.....37
图 3-1 BLE SDK 标准架构.....43
图 3-2 Host 和 Controller 的 HCI 数据交互.....44
图 3-3 17H25 hci 架构.....45
图 3-4 Lenze BLE slave 架构.....46
图 3-5 Lenze BLE master 架构.....错误！未定义书签。
图 3-6 State diagram of the Link Layer state machine in BLE Spec.....47
图 3-7 Lenze Link Layer state machine.....48
图 3-8 Idle + Advertising.....49
图 3-9 Idle + Scanning.....50
图 3-10 BLE slave LL state.....51
图 3-12 Advertising State 时序.....53
图 3-13 Scanning state 时序.....53
图 3-14 Initiating state 时序.....54
图 3-15 Conn state Slave role 时序.....54
图 3-17 Scanning in Advertising state 时序.....57
图 3-18 Scanning in ConnSlaveRole 时序.....57
图 3-19 Advertising in ConnSlaveRole 时序.....58
图 3-20 Advertising and Scanning in ConnSlaveRole 时序.....58
图 3-27 Lenze defined event 架构.....60
图 3-28 connect request PDU.....62
图 3-29 BLE 协议栈 LL_CONNECTION_UPDATE_REQ 格式.....65
图 3-30 BLE 协议栈广播包格式.....68
图 3-31 BLE 协议栈里 Advertising Event.....69
图 3-32 BLE 协议栈四种广播事件.....70
图 3-33 Whitelist/Resolvinglist 地址过滤.....82
图 3-34 BLE 协议栈中 Connection Para update Req 格式.....83
图 3-35 抓包显示 conn para update reqeust 和 response.....84
图 3-36 BLE 协议栈中 conn para update rsp 格式.....84
图 3-37 抓包显示 ll conn update req.....86
图 3-38 Attribute 构成 GATT service.....87
图 3-39 17H25 BLE SDK Attribute Table 截图.....88
图 3-40 master 读 hidInformation 的 BLE 抓包.....91
图 3-41 BLE 协议栈中 Write Request.....92
图 3-42 BLE 协议栈中 Write Command.....92

Lenze Technology Co.,LTD

图 3-43 Service/Attribute Layout....95
图 3-44 Read by Group Type Request/Read by Group Type Response....96
图 3-45 Find by Type Value Request/Find by Type Value Response....97
图 3-46 Read by Type Request/Read by Type Response....98
图 3-47 Find information request/Find information response....98
图 3-48 Read Request/Read Response....98
图 3-49 Read Blob Request/Read Blob Response....99
图 3-50 Exchange MTU Request/Exchange MTU Response....99
图 3-51 Write Request/Write Response....101
图 3-52 BLE Spec 中 Handle Value Notification....102
图 3-53 BLE Spec 中 Handle Value Indication....103
图 3-54 BLE Spec 中 Handle Value Confirmation....103
图 3-55 抓包显示 Pairing Disable....107
图 3-56 抓包显示 Pairing Conn Trigger....107
图 3-57 抓包显示 Pairing Peer Trigger....108
图 3-58 master 发起 Pairing_Req....108
图 4-1 17H25 MCU 硬件唤醒源....116
图 4-2 Link Layer Idle state 低功耗管理....119
图 4-3 17H25remote 低功耗管理参考 code 截图....127
图 4-4 app wakup tick 提前触发....129
图 6-2 17H25 Flash 存储结构....129
图 6-4 firmware size 信息....133
图 6-5 firmware 结尾....133
图 6-8 BLE 协议栈 Write Command 格式....137
图 6-9 OTA 命令和数据的格式....137
图 6-10 master 通过 Read By Type Request 获取 OTA 的 Attribute Handle....138
图 6-11 firmware 示例--开头部分....138
图 6-12 firmware 示例--结尾部分....138
图 6-13 master 发 OTA start....139
图 6-14 master OTA 数据....140
图 6-1 行列式键盘结构....143
图 6-2 keycode 处理函数....147
图 6-3 keyscan 时间优化....151
图 9-1 Tscript 启动界面....169
图 9-2 UART_BaudRate_cal.lua....169
图 9-3 entry the baudrate....170
图 9-4 entry system clock....170
图 9-5 get result....171
图 9-7 17H25 ADC channel....174
图 9-8 17H25 低电压检测硬件示意图....175
图 9-12 EMI test tool....190
图 9-13 选择数据总线....190
图 9-14 Swire 同步操作....191
图 9-15 set channel....191
图 9-16 选择 RF 模式....192

Lenze Technology Co.,LTD

图 9-17 set RF 模式显示界面 192
图 9-18 选择测试 mode 193
图 9-19 设置 TX packet number 193
图 9-20 TX mode 显示界面 193
图 9-21 RX packet number 和 RSSI 194
图 10-1 PWM timing and set 197
图 10-2 NEC IR 协议 200
图 10-3 IR 结尾 203
图 10-4 IR 尾部添加 end ir task 203
图 10-5 ir_nec_send 204
图 10-6 ir_nec_send_repeat 205

1 SDK 介绍

Lenze 17H25 BLE SDK 提供 BLE slave/master 开发 demo code，用户可以在这些 demo code 基础上开发自己的应用程序。

目前适用的 IC：17H25 。

1.1 软件组织架构

Lenze BLE 17H25 SDK 软件架构包括 APP 应用层和 BLE stack 协议栈部分。

在 Lenze IDE 中导入 sdk 工程后，显示的文件组织结构如下图所示。有 3 个主要的顶层文件夹：proj，proj_lib，vendor。

Lenze Technology Co.,LTD

图 1-1 SDK 文件结构

- proj：提供 MCU 相关的外设驱动程序，如 flash，i2c，usb，gpio, uart 等。
- proj_lib: 提供 MCU 运行所必需的库文件包括 BLE 协议栈、RF 驱动、PM 驱动等，这部分是以库文件形式提供的，user 无法看到源文件，如 liblt_17H25.a 等。
- vendor：用于存放用户应用层代码，17H25 ble remote 示例应用在这个文件夹中。用户新建一个文件夹后，需要最基本的几个文件如下所示：

1.1.1 main.c

包括 main 函数入口，系统初始化的相关函数，以及无限循环 while(1)的写法，建议不要对此文件进行任何修改，直接使用固有写法。

```
int main (void) {

    blc_pm_select_internal_32k_crystal(); //选择内部 32k rc 作为 32k counter 时钟源
    cpu_wakeup_init(CRYSTAL_TYPE);//MCU 最基本的硬件初始化，user 不用关注
    clock_init();          //时钟初始化，user 在 app_config.h 中配置相关参数即可
    set_tick_per_us(CLOCK_SYS_CLOCK_HZ/1000000);//时钟初始化
    gpio_init();           //gpio 初始化，user 在 app_config.h 中配置相关参数即可
    rf_drv_init(CRYSTAL_TYPE);//RF 初始化，user 不用关注
    user_init ();    //ble 初始化，整个系统初始化，user 进行设定
    irq_enable();          //开全局中断
    while (1) {
        #if (MODULE_WATCHDOG_ENABLE)
```


Lenze Technology LTD

```
                wd_clear(); //clear watch dog
            #endif
            main_loop (); //包括 ble 收发处理、低功耗管理和 user 的任务
        }
    }
```

1.1.2 app_config.h

用户配置文件，用于对整个系统的相关参数进行配置，包括 BLE 相关参数、GPIO 的配置、PM 低功耗管理的相关配置等。

后面介绍各个模块时会对 app_config.h 中的各个参数的含义进行详细说明。

1.1.3 application file

如 app.c：用户文件，用于完成系统的初始化和添加用户的 task UI。

BLE slave 工程的 app_att.c：service 和 profile 的配置文件，有 Lenze 定义的 Attribute 结构，根据该结构，已提供 GATT、标准 HID 和私有的 OTA、MIC 等相关 Attribute，用户可以参考这些添加自己的 service 和 profile。

1.1.4 BLE stack entry

Lenze BLE SDK 中 BLE stack 部分 code 的入口函数有两个：

1) main.c 文件 irq_handler 函数中 BLE 相关中断的处理入口 irq_blt_sdk_handler。

```
_attribute_ram_code_ void irq_handler(void)
{
    ......
    irq_blt_sdk_handler ();
    ......
}
```

2) application file mainloop 中 BLE 逻辑和数据处理的函数入口 blt_sdk_main_loop。

```
void main_loop (void)
{
    tick_loop ++;

    //////////////////// BLE entry ////////////////////////////
    blt_sdk_main_loop();

    ////////////////////// UI entry ////////////////////////////
```


Lenze Technology Co.,LTD

```
    ......
}
```

1.2 IC 介绍

17H25 硬件各个模块 audio/Flash Size/Sram Size 配置如下表所示。

IC	IP core	Audio	Flash size	Sram size
17H25	IP1	×	512 kB	16 kB

根据以上 IC 的特点，SDK 中文件架构会有一些特点，下面详细介绍。

1.3 Driver 介绍

SDK 中 driver code 绝大部分以源码形式提供（除了 rf 和 pm driver），这些文件主要分布在以下 3 个地方：

1) proj_lib

如下图所示，pm_17H25 不开放源码，相关接口封装到 lib 中，这里只提供头文件供用户参考。

图 1- 2 pm 和 rf driver

2) proj/mcu_spec

对于如 adc 和 gpio，将相应的源文件和头文件放到 proj/mcu_spec 文件夹下面，如下图所示的 adc/gpio/register 所有相关文件。

图 1-3 adc/gpio driver 和 bootloader

3) proj/drivers

对于如 uart/i2c，统一由 proj/driver 下的对应文件来实现，如下面两图所示。

图 1-4 i2c/spi/uart 等 driver

Lenze Technology Co.,LTD

```
/*****
 * brief: the function initial the relevant gpio as i2c.
 *        when enable one group of pins as i2c, the other two groups of pin should be disable the i2c function.
 *        in addition, when disable pin as i2c, we can enable the pin as gpio.
 * param[in] gpio_sda -- the pin as data line of i2c.
 * param[in] gpio_scl -- the pin as clock line of i2c.
 * return none
 */
void i2c_pin_init(I2C_GPIO_GroupTypeDef i2c_pin_group){
    u32 gpio_sda,  gpio_scl;
    gpio_sda = GPIO_PE7;
    gpio_scl = GPIO_PF1;

    gpio_set_func(gpio_sda,AS_I2C);  //disable gpio function
    gpio_set_func(gpio_scl,AS_I2C);  //disable gpio function

    gpio_setup_up_down_resistor(gpio_sda, PM_PIN_PULLUP_10K);
    gpio_setup_up_down_resistor(gpio_scl, PM_PIN_PULLUP_10K);

    gpio_set_input_en(gpio_sda,1);
    gpio_set_input_en(gpio_scl,1);


}
```

图 1-5 i2c driver 中区分 17H25 和其他

1.4 bootloader 介绍

IC 的 bootloader : cstartup_ram.S 文件在 SDK 的位置见 图 1-3adc/gpio driver 和 bootloader 中所示。

图 1-6 选择 bootloader

1.5 library 介绍

下图所示为 project 对应的 libH25-sdk.a library。

图 1-7 BLE SDK 提供的 library

Lenze BLE SDK 的 library 有以下几个特点：

1.5.1 根据功能分类

目前 SDK 已经提供了一种基本的 BLE 功能对应的 library 是 libH25-sdk.a 。
作为 BLE slave，Lenze 17H25 作为主控 MCU，对应的 library 包括 BLE slave，Lenze 17H25 作为 BLE 透传模块，通过 UART/SPI 等接口与其他主控 MCU 完成数据通信；

1.5.2 根据 system clock 分类

BLE Slave 相关的 library 会根据应用程序的 system clock 不同而有所区别。
对于 BLE slave 相关的 library， 默认为 16M 时钟 。

1.5.3 其他特殊的 library

由于实际应用中一些特殊的情况，SDK 会提供一些特殊的 library，比如做 IR 应用时，提供的 16m 和 32m system clock 对应的 library。
注意所有的 IR 应用都需要提供特殊的 library 来实现，若有需要实现 IR 应用，请 user 联系 Lenze 技术支持人员，申请相关的 library。

1.6 Demo 介绍

Lenze BLE SDK 提供了多个 BLE demo 给 user，每一个 demo code 都有对应的 hardware，user 通过软硬件 demo 的运行，可以观察到直观的效果，在 demo code 上修改，完成自己的应用开发。

Lenze Technology Co.,LTD

图 1-8 BLE SDK 提供的 demo code

图 1-9 编译时根据 BLE SDK 提供的 demo code 选择

1.6.1 BLE slave demo

BLE slave 的 demo 及区别如下表所示。

Demo	Stack	Application	MCU function
17H25 module	BLE controller + host	Application 在主控 MCU 上	BLE 透传模组
17H25 ble_remote	BLE controller + host	遥控器应用	主控 MCU

17H25 ble_remote/17H25 module 都是 Lenze 提供完整的 BLE slave 的 stack。17H25 module 只作为 BLE 透传模组，与主控 MCU 通过 UART 接口通信，一般应用代码写在对方主控 MCU；17H25 ble_remote 是 BLE remote controller 的 demo，实现 remote 的基本功能，可以和标准的 IOS/android 设备或者 Lenze 自己的 17H2x master kma dongle 连接，并控制对方。

17H25 ota boot 是 17H25 所有 BLE slave 工程 OTA 必须的一段 code，文档后面 OTA 部分会详细介绍。

1.6.2 BLE master demo

17H25 remote 对应的 libary 提供了标准的 BLE stack，包含了 BLE controller + BLE host，user 只需要在 app 层添加自己的应用代码，不用再去处理 BLE host 的东西，完全依赖于 controller 和 host 的 API 即可。

1.6.3 feature demo and driver demo

17H25 feature test 对一些常用的 BLE 相关的 feature，给出 demo code，user 参考这些 demo 完成自己的功能实现，详情见 code，该文档 BLE 部分会介绍所有的 feature。

17H25 driver test 对基本的 driver 给出 sample code，user 参考并实现自己的 driver 功能。本文档 driver 部分会详细介绍各个 driver。

2 MCU 基础模块

2.1 MCU 地址空间

2.1.1 MCU 地址空间分配

Lenze 17H25 MCU 的最大寻址空间为 16M bytes，从 0 到 0x7fffff 的 8M 空间为程序空间，即最大程序容量为 8M bytes；0x800000 到 0xffffff 的 8M 为外部设备空间（例如：SRAM、寄存器空间）。

Lenze Technology Co.,LTD

图 2-1 MCU 地址空间分配

17H25 MCU 物理寻址时，地址线 BIT（23）用于区别程序空间/外设空间：该地址线为 0 时，访问程序空间；该地址为 1 时，访问外设空间。

寻址空间为外设空间（BIT(23)为 1）时，地址线 BIT（15）用于区别 Register 和 Sram：该地址线为 0 时，访问 Register；该地址为 1 时，访问 Sram。

2.1.2 Sram 空间分配

2.1.2.1 Sram 和 Firmware 空间

对 MCU 地址空间中 Sram 空间的分配做进一步说明。

16kB Sram 地址空间范围为 0x808000～0x80C000，32kB Sram 地址空间范围为 0x808000～0x810000。

以 16kB Sram 为例，下图所示为 Sram 和 Firmware 空间分配。

Lenze Technology Co.,LTD

图 2-2 Sram 空间和 Firmware 空间

SDK 中 Sram 空间分配相关的文件有 boot.link 和 cstartup.S（如果是其他 MCU，这个 S 文件根据 IC 名字对应查看）。

Flash 中的 Firmware 包括 vector、ramcode、text、Rodata 和 Data initial value。Sram 中包括 vector、ramcode、Cache、data、bss、stack 和 unused area。Sram 中的 vector/ramcode 是 Flash 中 vector/ramcode 的拷贝。

1) vectors、ram_code

Vectors 是 Flash Firmware（SDK 中编译程序产生的可执行 bin 文件）中的一段 code，对应汇编文件 cstartup.S，即启动代码（bootloader）。

Ramcode 是 Flash Firmware 中需要常驻内存的 code，对应 SDK 中所有加了关键字“_attribute_ram_code_”的函数，比如 flash erase 函数：

```
_attribute_ram_code_ void flash_erase_sector(u32 addr);
```

函数常驻内存有两个原因：一是某些函数由于涉及到和 Flash MSPI 四根管脚的时序复用，必须常驻内存，如果放到 flash 中就会出现时序冲突，造成死机，如 flash 操作相关所有函数；二是放到 ram 中执行的函数每次被调用时不需要从 flash 重新读取，可以节省时间，所以对

Lenze Technology Co., LTD

于一些执行时间有要求的函数可以放到常驻内存，提高执行效率。SDK 中将 BLE 时序相关的一些经常要执行的函数常驻到内存，大大降低执行时间，最终节省功耗。

User 如果需要将某个函数常驻内存，可以仿照上面 flash_erase_sector，在自己的函数上添加关键字“_attribute_ram_code_”，编译之后就能在 list 文件中看到该函数在 ramcode 段了。

Firmware 中的 vector 和 ramcode 都需要在 MCU 上电时全部搬到 ram 上，编译之后，这两部分加起来的 size 为_ramcode_size_，_ramcode_size_是一个编译器能够认识的变量值，它的计算实现在 boot.link 中，如下所示，编译的结果_ramcode_size_等于 vector 和 ramcode 所有 code 的 size。

```
. = 0x0;
    .vectors :
    {
    *(.vectors)
    *(.vectors.*)
    }
    .ram_code :
    {
    *(.ram_code)
    *(.ram_code.*)
    }
PROVIDE(_ramcode_size_ = . );//计算实际的ramcode size(vector + ramcode)
PROVIDE(_ramcode_size_div_16_ = (. + 15 ) / 16);
PROVIDE(_ramcode_size_div_256_ = (. + 255) / 256);
PROVIDE(_ramcode_size_div_16_align_256_ = ( (. + 255) / 256) * 16);
```

2) Cache

Cache 是 MCU 的指令高速缓存，必须被配置为 Sram 中的一段才可以正常运行。Cache size 是固定的，包括 256 字节的 tag 和 2048 字节的 Instructions cache，总共 0x900 = 2.25K。

常驻内存的 code 可以直接从内存中读取并执行，但 firmware 中可以常驻内存的 code 只是很小的一部分，剩下绝大部分都还在 Flash 中。根据程序的局部性原理，可以将一部分 Flash code 存储到 Cache 中，如果当前需要执行的 code 在 Cache 里，直接从 Cache 读取指令并执行；如果不在 Cache 中，则从 Flash 读取 code 并覆盖之前的 Cache 中的 code，再从 Cache 中读取指令执行。

图 2-2 中 firmware 的 text 段是没有放到 Sram 中的 Flash code，这部分 code 符合程序局部性原理，需要一直被 load 到 Cache 中才能被执行。

Cache 大小是固定的 2.25K，但它在 Sram 中的起始地址为可配置的，为了保证 vector 和 ramcode 能全部存储到 Flash 上，这个起始地址必须大于 0x808000+_ramcode_size_。17H25 MCU 硬件上限定了 Cache 起始地址必须是 256 byte 对齐，所以 real_ramcode_size 为_ramcode_size_ 256 字节对齐后的 size，Cache 起始地址为：

0x808000 + real_ramcode_size

= 0x808000 + ((_ramcode_size_+255)/256)* 256

= 0x808000 + _ramcode_size_div_256_* 0x100

Lenze Technology Co.,LTD

Cache 的起始地址为 256 字节对齐的“0x808000 + _ramcode_size_div_256_* 0x100”，而_ramcode_size_一般不是 256 字节对齐，最终 MCU 上电时从 Flash load 到 ram 中的实际 size 为_ramcode_size_div_256_* 256，有一部分 Sram 空间是浪费的。

比如：假设_ramcode_size_为 0x780，load 到 Sram 中的 code 大小为 0x800，那么 Flash 中 0x00000 ~ 0x007ff 的 code 都属于常驻内存 code 了，Sram 上 0x808780 ~0x8087ff 共 128 byte 是浪费的，属于非 ramcode 常驻到了 Sram 上。

如果_ramcode_size_为 0x701，会浪费 255byte；如果_ramcode_size_为 0x800，一个 byte 都不浪费。最大可能浪费的 Sram 空间为 255 byte，所以 user 在进行程序设计时，需要检查 list 文件查看 ramcode 的占用情况，尽量避免浪费严重的情况。

由于 Cache size 固定为 0x900 = 2.25K，Cache 的结束地址为

0x808000 + real_ramcode_size + 0x900

= 0x808900 + real_ramcode_size

Lenze Technology Co.,LTD

3) data / bss

Data 段是 Sram 中存放程序已经初始化的全局变量，即初始非 0 的全局变量。Bss 段是 Sram 中存放程序未初始化的全局变量，即初始为 0 的全局变量。这两部分是连在一起的，data 段后紧跟 bss 段，所以这里作为一个整体介绍。

Data + bss 段紧跟 Cache，起始地址即 Cache 的结束地址 0x808900 + _ramcode_size_div_256_* 0x100。下面为 boot.link 中的代码，直接定义 data 段开始的地址：

```
. = 0x808900 + _ramcode_size_div_256_ * 0x100;
          .data :
```

Data 段是被初始化的全局变量，其初始即图 2- 2 中所示 firmware 中的 Data init value。

4) stack / unused area

Stack 是 Sram 中从最低地址 0x80C000（默认 16kB Sram）（32kB Sram 的话，则为 0x810000）开始的，其方向为从下往上延伸，即 stack 指针 SP 每次入栈时自减，出栈时自加。

默认情况下，SDK library 使用了 stack 的 size 不超过 256 byte，但由于 stack 的使用 size 取决于 stack 最深位置的地址，所以最终 stack 的使用量跟 user 上层的程序设计是有关的。如果 user 使用了比较麻烦的递归函数调用，或者在函数里使用了比较大的局部数组变量，或者其他有可能造成 stack 比较深的情况，都会导致最终 stack 的使用 size 变大。

当 user 的 Sram 使用较多时，需要明确知道自己的程序使用了多少 stack，这个无法通过 list 文件来分析，只能让实际产品应用运行起来，确保其运行了程序中所有的可能用 stack 比较深的 code 后，将 MCU reset，读取 Sram 空间去确定 stack 的使用量。

Unused area 即 bss 段结束地址与 stack 最深地址之间剩余的空间。只有当这个空间存在时，才说明 stack 没有和 bss 冲突，Sram 使用没有问题。如果 stack 最深的地方和 bss 段重合了，则说明 Sram 不够用了。

通过 list 文件可以查出 bss 段结束的地址，也就确定了留给 stack 的最大空间，user 需要分析这个空间是否足够，结合上面说的查看 stack 最深地址，可以知道 Sram 的使用是否超出。下面 demo 中会给出分析方法。

Lenze Technology Co.,LTD

5) text

Text 段是 Flash firmware 中的空间。程序中的函数如果加了“_attribute_ram_code_”，会被编译为 ram_code 段，其他没有加这个关键字的函数就全部编译到了 Text 段。Text 段是 Firmware 中最大的空间，一般远远大于 Sram 的 size，所以需要通过 Cache 的缓存功能，将需要执行的 code 先 load 到 Cache 中再执行。

6) rodata/data init value

Firmware 中除了 vector、ram_code 和 text，剩余的数据为 Rodata 和 Data initial value。

Rodata 是程序中定义的只可读、不能改写的数据，也就是用关键字“const”定义的变量。

比如 Slave 中的 ATT table：

```
const attribute_t my_Attributes[] = ……
```

User 可以在对应的 list 文件中看到“my_Attributes”是在 Rodata 段之内。

前面介绍的 Data 段是程序中已初始化的全局变量，比如定义全局变量如下：

```
int     testValue = 0x1234;
```

那么编译器就会将初值 0x1234 存放到 Data initial value 中，在运行 bootloader（cstartup.S）的时候，会将该初值 copy 到 testValue 对应的内存地址。

2.1.2.2 list 文件分析 demo

这里以 BLE slave 最简单的 demo 17H25 hid sample 为例，结合“Sram 和 Flash 地址空间”图来详细解释 Sram 和 Flash 地址空间的分配。User 只要理解这个 demo 的具体介绍，就可以去分析自己的应用程序上 Sram 和 Flash 地址空间的分配。

17H25 hid sample 的 bin 文件和 list 文件见目录“SDK”->“Demo”->“list file analyze”。重点关注 ble_hid_sample.list 文件，这个文件可以分析出 Sram 空间分配的所有信息。

下面开始详细分析，文中会出现多处截图，均来自 boot.link、cstartup.S、ble_hid_sample.bin 和 ble_hid_sample.list，请 user 自己查找对应文件找到截图对应位置。

list 文件中将不同功能的 code 称为“section”，可以参见 ble_hid_sample.list 中 section 的分布情况，如下图所示：

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .vectors      00000100  00000000  00000000  00008000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .ram_code     000012c8  00000100  00000100  00008100  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text         00005124  000013d0  000013d0  000093d0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00000894  000064f4  000064f4  0000e4f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000124  00809d00  00006d88  00011d00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000be0  00809e30  00006eb8  00011e24  2**4
                  ALLOC
```

图 2-3 list 文件 Section 统计

根据 Section 统计先将需要了解的信息列出来，和后面具体的 section 介绍都会一一对应。

1) Vectors：从 Flash 0 开始，Size 为 0x100；
2) ram_code：从 Flash 0x100 开始，Size 为 0x12c8；
3) text: 从 Flash 0x13d0 开始，Size 为 0x5124；
4) rodata: 从 Flash 0x64f4 开始，Size 为 0x894；
5) data: 从 Sram 0x809d00 开始，Size 为 0x124；
6) bss：从 Sram 0x809e30 开始，Size 为 0xbe0，这里可以计算出 bss 结束的地址为 0x809e30 + 0xbe0 = 0x80aa10。结合前面介绍可知，bss 后面还剩余空间为 0x80c000 – 0x80aa10 = 0x15F0 = 5616 byte，减去 stack 需要使用的 256 byte，还剩 5360 byte 没有使用。

```
Disassembly of section .ram_code:

00000100 <irq_handler>:
irq_handler():
     100:   6500        tpush   {lr}
     102:   9000 9b6d   tjl 7e0 <irq_blt_sdk_h
    13c4:   0080a70c    taddeq  sl, r0, ip, lsl

Disassembly of section .text:

000013d0 <__modsi3>:
__modsi3():
    64f0:   04c4b400    tstorerbeq  fp, [r4],

Disassembly of section .rodata:

000064f4 <tbl_advData>:
    64f4:   48740905    tloadmdami  r4!, {r0,
    6d84:   19867909    tstormmibne r6, {r0, r

Disassembly of section .data:

00809d00 <_start_data_>:
_start_data_():
  809d00:   00000304    tandeq  r0, r0, r4, ls

00809e08 <smpResSignalPkt>:
  809e08:   00111502    tandseq r1, r1, r2, ls
  809e0c:   00000006    tandeq  r0, r0, r6
    ...

Disassembly of section .bss:

00809e30 <_start_bss_>:

0080a9d0 <smp_param_peer>:
    ...

Disassembly of section .TC32.attributes:
```

图 2-4 list 文件 section 地址

上图为在 list 文件中搜索“section”查到各 section 的起始/结束地址后的拼图，结合该图和上面“list 文件 Section 统计”图，分析如下：

1) vector

vector 即 startup.S 汇编文件对应的 bootloader，list 文件可以看到，Flash 中起始地址为 0，结束地址为 0xff，size 为 256 byte。上电搬移到 Sram 后，在 Sram 上的地址为 0x808000 ~ 0x8080ff。

2) ram_code

ramcode section 起始地址为 0x100，结束地址为 0x13c8，size 为 0x12c8。那么_ramcode_size_ = 0x13c8，256 byte 对齐后 real_ramcode_size 为 0x1400，实际有 0x38 = 56 byte 的 Sram 浪费。

3) Cache

Cache 开始和结束地址为：

0x808000 + real_ramcode_size ~ 0x808900 + real_ramcode_size

0x809400 ~ 0x809d00

Cache 的相关信息在 list 文件中不会体现出来。

4) text

Text 段起始地址为 ram_code 段的结束地址 0x13d0，结束地址为 0x64f4，Size 为 0x64f4 – 0x13d0 = 0x5124，和前面 Section 统计中数据一致。

5) rodata

Rodata 的起始地址为 text 的结束地址 0x64f4，结束地址为 0x6d88。

打开 ble_hid_sample.bin 可以看到实际 bin size 为 0x6eac。所以根据前面的分析，剩下的 firmware 空间 0x6d88 ~ 0x6eac 实际是 data init value，是程序中已初始化的全局变量的初值。data init value 在 list 文件中没有专门作为一个 section 来统计，可以搜索“_dstored_”，找到该值为下面的 0x6d88，表示 data init value 的起始地址。

```
00006d88 g        *ABS*   00000000 _dstored_
```

_dstored_的定义在 boot.link 中，如下。这里是告诉编译器：data 段已初始化的全局变量的初值全部存放到 Firmware 中_dstored_段。

```
. = 0x808900 + _ramcode_size_div_256_ * 0x100;
   .data :
    AT ( _dstored_ )
    {
. = (((. + 3) / 4)*4);
    PROVIDE(_start_data_ = . );
    *(.data);
    *(.data.*);
. = (((. + 3) / 4)*4);
    PROVIDE(_end_data_ = . );
      }
```

6) data

Lenze Technology Co.,LTD

data 起始地址为 Cache 结束地址 0x809d00，list 文件 Section 统计部分给出的 size 为 0x124。data 段最后一个变量为 smpResSignalPkt，地址为 0x809e08，这是一个 SDK 底层的结构体变量，其大小为 28 = 0x1c，所以 data 结束地址为 0x809e24，data 段 size 为 0x809e24 - 0x809d00 = 0x124，和上面的一致。

7) bss

bss 紧跟 data 段，由于第一个数组“_start_bss_”需要 16 byte 对齐，起始地址为 0x9e30，list 文件 Section 统计部分给出的 size 为 0xbe0。

bss 段最后一个变量为 smp_param_peer，地址为 0x80a9d0，这是一个 SDK 底层的结构体变量，其大小为 64 = 0x40，所以 bss 结束地址为 0x80aa10，bss 段 size 为 0x80aa10 - 0x809e30 = 0xbe0，和上面的一致。

通过这里的分析，也能计算出剩余的 Sram 空间为 0x80c000 – 0x80aa10 = 0x15F0 = 5616 byte，减去 stack 需要使用的 256 byte，还剩 5360 byte 没有使用。

2.1.3 MCU 地址空间访问

程序中对 0x000000 - 0xffffff 地址空间的访问分以下两种情况：

2.1.3.1 外设空间的读写操作

外设空间（register 和 sram）的读写操作直接用指针访问实现：

```
u8   x = *(volatile u8*)0x800066;   //读 register 0x66 的值
 *(volatile u8*)0x800066 = 0x26;   //给 register 0x66 赋值
u32 y = *(volatile u32*)0x808000;          //读 sram 0x8000-0x8003 地址的值
*(volatile u32*)0x808000 = 0x12345678;   //给 sram 0x8000-0x8003 地址赋值
```

程序中使用函数 write_reg8、write_reg16、write_reg32、read_reg8、read_reg16、read_reg32 对外设空间进行读写，其实质也是操作指针，详情请查看 proj/common/compatibility.h 和 proj/common/utility.h。

注意程序中类似 write_reg8(0x8000)/ read_reg16(0x8000)的操作，其定义如下所示，可以看到是自动加上了 0x800000 的（地址线 BIT(23)为 1），所以能够确保访问的是 Register/Sram 空间，而不会去访问 flash 空间。

```
#define REG_BASE_ADDR              0x800000
#define write_reg8(addr,v)   U8_SET((addr + REG_BASE_ADDR),v)
#define write_reg16(addr,v) U16_SET((addr + REG_BASE_ADDR),v)
#define write_reg32(addr,v) U32_SET((addr + REG_BASE_ADDR),v)
#define read_reg8(addr)      U8_GET((addr + REG_BASE_ADDR))
#define read_reg16(addr) U16_GET((addr + REG_BASE_ADDR))
#define read_reg32(addr) U32_GET((addr + REG_BASE_ADDR))
```

这里注意一个内存对齐的问题：如果使用指向 2 字节/4 字节的指针来读写外设空间，一定要确保地址是 2 字节/4 字节对齐的，如果不对齐的话，会发生数据读写错误。如下两种是错误的：

```
u16   x = *(volatile u16*)0x808001;        //0x808001 没有 2 字节对齐
```


Lenze Technology Co.,LTD

(volatile u32)0x808005 = 0x12345678; //0x808005 没有 4 字节对齐

修改为正确的读写操作：

u16 x = *(volatile u16*)0x808000; //0x808000 2 字节对齐

(volatile u32)0x808004 = 0x12345678; //0x808004 4 字节对齐

2.1.3.2 flash 空间的操作

flash 空间的读写操作使用 flash_read_page 和 flash_write_page 函数实现，flash 的擦除使用，代码见 proj/drivers/flash.c 和 flash.h。

1) flash 读写操作

flash 读写操作使用 flash_read_page 和 flash_write_page 函数来实现。

```
void flash_read_page(u32 addr, u32 len, u8 *buf);
void flash_write_page(u32 addr, u32 len, u8 *buf)
```

flash_read_page 函数读取 flash 上的内容:

```
void flash_read_page(u32 addr, u32 len, u8 *buf);
u8 data[6] = {0 };
flash_read_page(0x11000, 6, data); //读 flash 0x11000 开始的 6 个 byte 到 data 数组。
```

flash_write_page 函数对 flash 进行写操作：

```
flash_write_page(u32 addr, u32 len, u8 *buf);
u8 data[6] = {0x11,0x22,0x33,0x44,0x55,0x66 };
flash_write_page(0x12000, 6, data); // 向 flash 0x12000 开始的 6 个 byte 写入 0x665544332211。
```

flash_write_page 函数是对 page 的操作，flash 里面一个 page 为 256 byte，这个函数操作的地址大小最大为 256 byte，不能跨越两个不同 page 范围。

当被操作的地址是一个 page 的首地址时，最大地址为 256 byte，flash_write_page(0x12000, 256 , data)操作正确，而 flash_write_page(0x12000, 257 , data)错误，因为最后一个地址不属于 0x12000 所在的 page 了，写操作会失败。

当被操作的地址不是一个 page 的首地址时，更要注意不能出现跨 page 的问题，如 flash_write_page(0x120f0, 20 , data)就错了，前 16 个地址在 0x12000 这个 page，而后 4 个地址在 0x12100 这个 page。

flash_read_page 不存在上面说的跨 page 的问题，可以一次性读取超过 256 byte 的数据。

2) flash 擦除操作

使用 flash_erase_sector 函数来擦除 flash。

```
void flash_erase_sector(u32 addr);
```

一个 sector 为 4096 byte，如 0x13000 ~0x13fff 是一个完整的 sector。

addr 必须是一个 sector 的首地址，该函数每次擦除整个 sector。

擦除一个 sector 的时间会比较长，16M 系统时钟时，大约需要 30~100ms 甚至更长时间。

3) flash 读写和擦除操作对系统中断的影响

上面介绍的三个 flash 操作函数 flash_read_page、flash_write_page、flash_erase_sector 在执行时，都必须先将系统中断关掉 irq_disable()，操作结束后再恢复中断 irq_restore()，其目的是为了保证 flash MSPI 时序操作的完整性和连续性，同时防止中断里又有 flash 操作调用 MSPI

Lenze Technology Co.,LTD

总线而造成硬件资源的重入。

这个 BLE SDK RF 收发包的时序全部由中断来控制的，flash 操作关闭系统中断造成的后果是 BLE 收发包的时序会被破坏，得不到及时响应。

其中 flash_read_page、flash_write_page 函数的执行时间不是太长，对 BLE 的中断影响很小，但连续读写的地址越长（最大 256 byte），时间就越长。所以强烈建议 user 在 mainloop 里 BLE 连接状态时，不要连续读写太长的地址。

flash_erase_sector 函数的执行时间为几十到几百个 ms，所以在主程序的 mainloop 里，一旦进入 BLE 连接状态，尽量不要去调用 flash_erase_sector 函数，否则会破坏 BLE 收发包的时间点，造成连接断开。如果是无法避免在 BLE 连接的时候需要擦除 flash，请根据本文档后面介绍的 Conn state Slave role 时序保护实现方法来操作。

4) 读 flash 可以使用指针访问来实现

17H25 BLE SDK 的 firmware 存储在 flash 上，程序运行时，只是将 flash 前一部分的代码作为常驻内存代码放在 ram 上执行，剩余的绝大部分代码根据程序的局部性原理，在需要的时候从 flash 读到 ram 高速缓存 cache 区域（简称 cache）。MCU 通过自动控制内部 MSPI 硬件模块，读取 flash 上的内容。

可以使用指针的形式读取 flash 上的内容，指针形式读 flash 的原理是 MCU 系统总线访问数据时，当发现数据地址不在常驻内存 ramcode 上，系统总线就会自动切换到 MSPI，通过 MSCN、MCLK、MSDI 和 MSDO 四根线去操作 spi 的时序来获得读取 flash 数据。

以下列出 3 种示例：

```
u16    x = *(volatile u16*)0x10000; //读 flash 0x10000 两个 byte
u8 data[16];
memcpy(data, 0x20000, 16);          //读 flash 0x20000   16 个 byte copy 到 data
 if(!memcmp(data, 0x30000, 16)){ //读 flash 0x30000   16 个 byte 和 data 比较
            //......
 }
```

user_init 里面读取 flash 上的校准值并设定到对应的 register 时，都是采用指针访问 flash 的方式实现的，请参考 SDK 里的函数

```
static inline void blc_app_loadCustomizedParameters(void);
```

读 flash 可以使用指针形式，但是不能使用指针写 flash（写 flash 只能通过 flash_write_page 来实现）。

需要注意指针读 flash 存在一个容易出问题的地方：只要是通过 MCU 系统总线读到的数据，MCU 都会将这些数据缓存在 cache 里，如果 cache 里这个数据没有被其他内容覆盖时，又有新的访问该数据的请求发生，此时 MCU 会直接用 cache 里缓存的内容作为结果。如果 user 的代码出现下面这种情况：

```
u8    result;
result = *(volatile u16*)0x40000;    //指针读取 flash
u8 data = 0x5A;
flash_write_page(0x40000, 1, &data );
result = *(volatile u16*)0x40000;  //指针读取 flash
if(result != 0x5A){    ..... }
```


Lenze Technology Co.,LTD

flash 地址 0x40000 处本来是 0xff，第 1 次读到的 result 为 0xff，然后写入 0x5A，理论上第 2 次读到的值是 0x5A，但实际程序给出的结果还是 0xff，是从 cache 里拿到的第一次缓存的结果。

所以 user 要注意，如果出现这种多次读同一个地址且这个地址的值会被改写时，千万不要用指针的形式，使用 API flash_read_page 来实现最安全，这个函数读到的结果不会从 cache 里拿之前缓存的值。改成如下实现才正确：

```
u8   result;
flash_read_page(0x40000, 1, &result );   //API 读取 flash
u8 data = 0x5A;
flash_write_page(0x40000, 1, &data );
flash_read_page(0x40000, 1, &result );   //API 读取 flash
if(result != 0x5A){   ..... }
```

2.1.4 SDK FLASH 空间的分配

FLASH 存储信息以一个 sector 的大小（4K byte）为基本的单位，因为 flash 的擦除是以 sector 为单位的（擦除函数为 flash_erase_sector），理论上同一种类的信息需要存储在一个 sector 里面，不同种类的信息需要在不同的 sector（防止擦除信息时将其他类的信息误擦除）。所以建议 user 在使用 FLASH 存储定制信息时遵循“不同类信息放在不同 sector”的原则。

Flash 地址空间的分配，根据实际的 512K flash size。

2.1.4.1 512K Flash 的地址分配

图 2-5 512K FLASH 地址分配

17H25 使用的 Flash 为 512K，上图所示为这颗 IC 默认的 FLASH 地址分配，实际所有的地址分配都给 user 提供了对应的修改接口，user 可以根据自己需要去规划地址分配。下面介绍默认的地址分配方法以及对应的修改地址的接口。

1. 0x76000~0x76FFF 这个 sector 存储 MAC 地址，实际上 MAC address 6 个 bytes 存储在 0x76000~0x76005，高 byte 的地址存放在 0x76005，低 byte 地址存放在 0x76000，比如 FLASH 0x76000 到 0x76005 的内容依次为 0x11 0x22 0x33 0x44 0x55 0x66，那么 MAC address 为 0x665544332211。

   伦茨的量产治具系统会将实际产品的 MAC 地址烧写到 0x76000 这个地址，和 SDK 相对应。如果 user 需要修改这个地址，请确保治具系统烧写的地址也作了相应的修改。SDK 中在 user_init 函数里会从 FLASH 的 CFG_ADR_MAC 读取 MAC 地址，这个宏在 proj_lib/ble/blt_config.h 里面修改即可。

```
#ifndef         CFG_ADR_MAC
#define         CFG_ADR_MAC                     0x76000
#endif
```

2. 0x77000~0x77fff 这个 sector 存储 Lenze MCU 需要校准定制的信息。只有这部分的信息不遵循“不同类信息放在不同 sector”的原则，将这个 sector 4096bytes 按照每 64 bytes 划分为不同的单元，每个单元存储一类校准信息。校准信息可以放在同一个 sector，是因为校准信息在治具烧录的过程中烧到对应地址，实际 firmware 在运行时只能读这些校准信息，而不允许去写，更不允许去擦除。具体的分配为：
   1) 第一个 64bytes 存储频偏校准信息，实际校准值只有一个 byte，存储在 0x77000。
   2) 第二个 64bytes 存储 TP 值校准值，实际只有两个 bytes(TP0、TP1)分别存储在 0x77040 和 0x77041。
   3) 第三个 64tytes 用来存储外部 32k crystal 的电容校准值，后面的第四个、第五个等留着做其他可能会校准的值。

   Lenze 的量产治具系统会将实际的校准值烧写到以上几个地址，和 SDK 相对应。如果 user 需要修改这个地址，请确保治具系统烧写的地址也作了相应的修改。SDK 中在 user_init 函数里 rf_customized_param_load() 函数会从这几个地址读相应的值，这几个宏在 proj_lib/ble/blt_config.h 里面修改即可。

```
#ifndef         CUST_CAP_INFO_ADDR
#define         CUST_CAP_INFO_ADDR          0x77000
#endif

#ifndef         CUST_TP_INFO_ADDR
#define         CUST_TP_INFO_ADDR           0x77040
#endif

#ifndef         CUST_32KPAD_CAP_INFO_ADDR
#define         CUST_32KPAD_CAP_INFO_ADDR       0x77080
#endif
```

3. 0x74000 ~ 0x75FFF 这两个 sector 被 BLE 协议栈系统占用，用来存储配对和加密信息。user 也可以修改这两个 sector 的位置，size 固定为两个 sector 8K，无法修改，可以调用下面函数

Lenze Technology Co., LTD

修改配对加密信息存储的起始地址：
proj_lib/ble/blt_smp_nv.h
void bls_smp_configParingSecurityInfoStorageAddr (int addr);

4. 17H25 的 OTA 需要 ota flag，0x73000 ~ 0x73FFF 这个 sector 被 BLE 协议栈系统占用，作为 OTA 程序更新时的标记 ota_flag。user 可以对这个地址进行修改，修改方法见后面 OTA 章节的详细说明。

5. 17H25 的 OTA 需要 ota boot，0x72000 ~ 0x72FFF 这个 sector 被 BLE 协议栈系统占用，存储 ota_boot.bin 用来实现新程序的搬移。user 可以对这个地址进行修改，修改方法见后面 OTA 章节的详细说明。

6. 0x00000 ~0x3FFFF 256K 空间默认作为程序空间。
   0x00000 ~ 0x1FFFF 共 128K 为 Firmware 存储空间；0x20000 ~ 0x3FFFF 128K 为 OTA 更新时存储新 Firmware 的空间，即支持最大 Firmware 空间为 128K。
   若默认的 128K 程序空间对 user 来说太大，而 user 希望在 0x00000~0x3FFFF 区域里腾出一些空间来作为数据存储空间，协议栈提供了相应的接口，修改方法见后面 OTA 章节的详细说明。

7. 剩余的 FLASH 空间全部作为 user 的数据存储空间。

2.2 时钟模块

系统时钟 system clock 是 MCU 执行程序的时钟，虽然 17H25 的系统时钟有多种时钟源（PLL、内部 OSC、内部 RC），但是在 17H25 ble SDK 中，只用 PLL 时钟，因为 PLL 时钟是最精准的。16M 或 12M 的外部晶振经过 MCU 内部的 PLL 硬件模块处理后，获得 192M 的 PLL 时钟（这部分硬件自动处理），通过软件配置相关寄存器进行分频获得低频的系统时钟。

17H25 BLE SDK 可以使用两种外部晶振，16M 和 12M，目前默认是使用 12M 晶振，在 app_config.h 中可以配置：

```
/////////////////Extern Crystal Type//////////////////////
#define CRYSTAL_TYPE        XTAL_12M      //  extern 12M crystal
```

可选的外部晶振类型为如下所示，在 main 的 rf 初始化时，代入 user 配置的晶体类型。XTAL_12M 默认就是 12M 晶体 BLE 1M 模式。

```
enum{
    XTAL_12M_RF_1m_MODE = 1,
    XTAL_12M_RF_2m_MODE = 2,
    XTAL_16M_RF_1m_MODE = 4,
    XTAL_16M_RF_2m_MODE = 8,

    XTAL_12M   = XTAL_12M_RF_1m_MODE,
    XTAL_16M    = XTAL_16M_RF_1m_MODE,
};
rf_drv_init(CRYSTAL_TYPE);
```


Lenze Technology Co., LTD

需要注意外部晶振的类型（12M、16M）与最终程序运行的系统时钟（16M）是两个不同的概念：前者是硬件晶体的具体规格，后者是 MCU 运行时的机器周期。不管外部晶振选哪个，MCU 都会通过内部的 PLL 电路倍频处理后获得 192M 的一个基础时钟。根据 user 在 app_config 中配置的系统时钟的频率，在 main 函数的 clock_init()阶段对 192M 基础时钟进行分频，获得 user 需要的频率。

2.2.1 系统时钟配置方法

main.c 中调用 clock_init 函数（详情见 proj/mcu/clock.c），对时钟源和分频系数相关寄存器进行配置，用户只需要在 app_config.h 中配置以下两个参数即可。

```
//////////////////// Clock  ////////////////////////////////
#define CLOCK_SYS_TYPE       CLOCK_TYPE_PLL   //时钟源选择PLL
#define CLOCK_SYS_CLOCK_HZ   16000000         //system clock 16M
```

目前 17H25 BLE SDK 推荐时钟频率为 16M，平时所有的调试都是基于该频率。16M 的时钟足以保证 BLE 的时序，并且功耗不会太大。

17H25 BLE SDK 支持多种系统时钟：默认的 16M、32M、48M 等。若 user 需要使用 32M 或 48M，按照上面将系统时钟配置好后，还需要选择对应的 lib 库，因为不同系统时钟有不同的 lib 库。

图 2-7 修改 lib 库方法

2.2.2 system tick 的使用

在配置好了系统时钟，并经过 clock_init 初始化后，16M 的 system clock 开始运行。基于这个时钟，可以不断读取系统时钟计数器的值，即 system clock tick，简称 system tick。该计数器每一个时钟周期加一，长度为 32bit，即每 1/16 us 加 1，最小值 0x00000000，最大值 0xffffffff。系统时钟启动的时候，system tick 值为 0，到最大值 0xffffffff 需要的时间为：(1/16) us * (2^32) 约等于 268 S，每过 268 S system clock tick 转一圈。

同理，如果系统时钟为 32M，那么 system clock tick 每 1/32 us 加 1，转一圈需要(1/32) us * (2^32) 约等于 134 S。

MCU 在运行程序过程中 system tick 不会停。

system tick 的值可以通过 clock_time()函数获得，比如需要记录当前的 system tick：

u32 current_tick = clock_time();

clock_time()函数实质是读出 system clock 计数的值。

17H25 BLE SDK 整个 BLE 时序都是基于 system tick 设计的，程序中也大量使用这个 system tick 来完成各种计时和超时判断，也强烈推荐 user 使用这个 system tick 来实现一些简单的定时和超时判断。

比如要实现一个简单的软件定时。软件定时器的实现基于查询机制，由于是通过查询来实现，不能保证非常好的实时性和准备性，一般用于对误差要求不是特别苛刻的应用。实现方法：

1) 启动计时：设置一个 u32 的变量，读取并记录当前 system clock tick。

   u32 start_tick = clock_time(); // clock_time()返回 system clock tick 值。

2) 在程序的某处不断查询当前 system clock tick 和 start_tick 的差值是否超过需要定时的时间值，若超过，认为定时器触发，执行相应的操作，并根据实际的需求清除计时器或启动新一轮的定时。假设需要定时的时间为 100 ms，那么对于 16M 的系统时钟，查询计时是否到达的写法为：

   if((u32) (clock_time() - start_tick) > 100 * 1000 * 16)

   由于将差值转化为 u32 型，解决了 system clock tick 从 0xffffffff 到 0 这个极限情况。32M 系统时钟查询方法为：

   if((u32) (clock_time() - start_tick) > 100 * 1000 * 32)

   实际上 SDK 中为了解决系统时钟不同导致和 u32 转换的问题，提供了统一的调用函数，不管系统时钟多少，都可以下面函数进行查询判断：

   if(clock_time_exceed(start_tick,100 * 1000)) //第二个参数单位为 us，不再需要考虑 16 //和 32 的问题

   需要注意的是：由于 16M 时钟转一圈为 268 S，这个查询函数只适用于 268 S 以内的定时，需要超过 268 S，需要在软件上加计数器累计实现（这里不介绍），同理 32M 时钟最大计时 134 S。

   应用举例：A 条件触发（只会触发一次）的 2 S 后，程序进行 B()操作。

```
u32    a_trig_tick;
int    a_trig_flg = 0;
while(1)
{
    if(A){
        a_trig_tick = clock_time();
        a_trig_flg = 1;
    }
    if(a_trig_flg &&clock_time_exceed(a_trig_tick,2 *1000 * 1000)){
```


Lenze Technology Co., LTD

```
            a_trig_flg = 0;
            B();
        }
    }
```

2.3 GPIO 模块

GPIO 模块的说明请 user 对照 proj/mcu_spec/gpio.h、gpio_default.h、gpio.c 来理解，所有代码都以源码形式提供。

代码中涉及到对寄存器的操作，请参考文档《17H25_gpio_lookuptable》、《ble_gpio_lookuptable》来理解，请根据《Getting Started with Lenze BLE SDK》的详细说明来找到这两个 gpio lookup table。

2.3.1 GPIO 定义

17H25 系列芯片共有 6 组 42 个 GPIO，分别为：

GPIO_PA0 - GPIO_PA7、GPIO_PB0 - GPIO_PB7、GPIO_PC0 - GPIO_PC7

GPIO_PD0 - GPIO_PD7、GPIO_PE0 - GPIO_PE7、GPIO_PF0-GPIO_PF1

注意：IC 的 core 部分都是有这 42 个 GPIO。

程序中需要使用 GPIO 时，必须按照上面的写法定义，详情见 proj/mcu_spec/gpio.h。

这 42 个 GPIO 中有 7 个比较特殊，需要注意：

1) MSPI 的 4 个 GPIO。这 4 个 GPIO 是 MCU 系统总线中主 SPI 总线，用于读写 flash 操作，上电默认为 spi 状态，user 永远不能操作它们，程序中不能使用。在 17H25 中，这个 4 个 GPIO 为 PA2、PA3、PB2、PB3，
2) SWS(Single Wire Slave)，用于 debug 和烧写 firmware，上电默认为 SWS 状态，程序中一般不使用。17H25 的 SWS 管脚为 PA0。
3) DM 和 DP，上电默认为 USB 状态，当需要 USB 功能时，DM 和 DP 需要使用，当不需要 USB 时，可以作为 GPIO 使用。17H25 的 DM、DP 管脚为 PB5、PB6。

2.3.2 GPIO 状态控制

这里只列举用户需要了解的最基本的 GPIO 状态。

42 个 GPIO 都包括以下状态：

1) func(功能配置：特殊功能/一般 GPIO)，如需要使用输入输出功能，需配置为一般 GPIO。

```
void    gpio_set_func(u32 pin, u32 func);
```

pin 为 GPIO 定义，以下一样。func 可选择 AS_GPIO 或其他特殊功能，这些功能定义如下：

```
#define AS_GPIO      0
#define AS_MSPI      1
#define AS_SWIRE  2
#define AS_UART      3
#define AS_PWM       4
#define AS_I2C       5
```


Lenze Technology Co.,LTD

```
#define AS_SPI        6
#define AS_I2S        8
#define AS_SDM        9
#define AS_DMIC       10
#define AS_USB        11
#define AS_SWS        12
#define AS_SWM        13
#define AS_ADC        15
```

2) ie(input enable)：输入使能

```
void gpio_set_input_en(u32 pin, u32 value);
```

value: 1 和 0 分别表示 enable 和 disable。

3) datai (data input)：当输入使能打开时，该值反应当前该 GPIO 管脚的电平，用于读取外部电压。

```
u32 gpio_read(u32 pin);
```

读到低电压返回值为 0，读到高电压，返回非 0 的值。这里要非常注意，当读到高电平时，返回值不一定是 1，是一个非 0 的值。详情见下面的 code。

```
static inline u32 gpio_read(u32 pin)
{
    return BM_IS_SET(reg_gpio_in(pin), pin & 0xff);
}
```

所以程序中，不能使用类似 if(gpio_read(GPIO_PA0) == 1)的写法，推荐使用方法是将读到的值取反处理，取反后只有 1 和 0 两种情况：

```
if( !gpio_read(GPIO_PA0) )   //判断高低电平
```

4) oe(output enable)：输出使能

```
void gpio_set_output_en(u32 pin, u32 value);
```

value 1 和 0 分别表示 enable 和 disable。

5) dataO (data output)：当输出使能打开时，该值为 1 输出高电平，为 0 输出低电平。

```
void gpio_write(u32 pin, u32 value)
```

6) 内部模拟上下拉电阻配置：有 1M 上拉、10K 上拉、100K 下拉 3 种模拟电阻，可配置的状态有 4 种：1M 上拉、10K 上拉、100K 下拉和 float 状态。

```
void gpio_setup_up_down_resistor(u32 gpio, u32 up_down);
```

up_down 的四种配置为：

PM_PIN_PULLUP_1M
PM_PIN_PULLUP_10K
PM_PIN_PULLDOWN_100K
PM_PIN_UP_DOWN_FLOAT

模拟电阻有一个特点：在 deepsleep 状态下，MCU 数字模块的状态都失效，失效的包括输入输出等状态（deepsleep 时无法输出电平）；但是只要之前配置了 3 种模拟，进入 deepsleep 后它们仍是生效的，依然保持着上下拉的作用。

GPIO 配置应用举例：

Lenze Technology Co., LTD

1) 将 GPIO_PA4 配置为输出态，并输出高电平。

```
gpio_set_func(GPIO_PA4, AS_GPIO) ;   // PA4 默认为 GPIO 功能，可以不设置
gpio_set_input_en(GPIO_PA4, 0);
gpio_set_output_en(GPIO_PA4, 1);
gpio_write(GPIO_PA4,1)
```

2) 将 GPIO_PC6 配置为输入态，判断是否读到低电平，需要开启上拉，防止 float 电平的影响。

```
gpio_set_func(GPIO_PC6, AS_GPIO) ;   // PC6 默认为 GPIO 功能，可以不设置
gpio_setup_up_down_resistor(GPIO_PC6, PM_PIN_PULLUP_10K);
gpio_set_input_en(GPIO_PC6, 1)
gpio_set_output_en(GPIO_PC6, 0);
if(!gpio_read(GPIO_PC6)){   //是否低电平
    ......
}
```

2.3.3 GPIO 的初始化

main.c 中调用 gpio_init 函数，会将 17H25 所有的 GPIO 的状态都初始化一遍。当用户的 app_config.h 中没有配置 GPIO 参数时，这个函数会将每个 IO 初始化为默认状态。42 个 GPIO 默认状态为：

1) func
   除了前面介绍的 7 个为特殊功能，其他均为一般 GPIO 状态。
2) ie
   7 个特殊的 GPIO（flash MSPI 4 个、USB 2 个、SWS）默认 ie 为 1，其他所有的一般 GPIO 默认 ie 为 0。这样的话，user 不用去将自己不用的 GPIO 的 ie 设为 0，但是要将自己需要使用 input 功能的 GPIO 的 ie 打开。需要使用 input 的情况包括：按键扫描中的 scan pin gpio、core/pad wakeup gpio、irq gpio 等。
3) oe
   全部为 0
4) dataO
   全部为 0
5) 内部上下拉电阻配置
   全部为 float。
   以上详情见 proj/mcu_spec/gpio.h、proj/mcu_spec/gpio_default.h。

如果在 app_config.h 中有配置到某个或某几个 GPIO 的状态，那么 gpio_init 时不再使用默认状态，而是使用用户在 app_config.h 配置的状态。原因是 gpio 的默认状态是使用宏来表示的，这些宏的写法为（以 PA7 的 ie 为例）：

```
#ifndef PA7_INPUT_ENABLE
#define PA7_INPUT_ENABLE     1
#endif
```

当在 app_config 中可以提前定义这些宏，这些宏就不再使用以上这种默认值。

在 app_config.h 中配置 GPIO 状态方法为（以 PA7 为例）：

1) 配置 func: #define PA7_FUNC AS_GPIO
2) 配置 ie: #define PA7_INPUT_ENABLE 1

Lenze Technology Co.,LTD

3) 配置 oe: #define PA7_OUTPUT_ENABLE 0
4) 配置 dataO：#define PA7_DATA_OUT 0
5) 配置内部上下拉电阻：
#define PULL_WAKEUP_SRC_PA7 PM_PIN_UP_DOWN_FLOAT

GPIO 的初始化总结：可以提前在 app_config.h 中定义 GPIO 的初始状态，在 gpio_init 中得以设定；可以在 user_init 函数中通过 2.3.2 节中 GPIO 状态控制函数加以设定；也可以使用以上两种方式混用，在 app_config.h 中提前定义一些，gpio_init 加以执行，在 user_init 中设定另外一些。需要注意的是：在 app_config.h 中定义和 user_init 中设定同一个 GPIO 的某个状态为不同的值时，根据程序的先后执行顺序，最终以 user_init 中设定为准。

2.3.4 配置 SWS 上拉防止死机

Lenze 所有的 MCU 都使用 SWS（single wire slave）来 debug 和烧录程序。在最终的应用代码上，SWS 这个 pin 的状态为：

1) function 上设为 SWS，非 GPIO。
2) ie =1，只有 input enable 时，才可以收到 EVK 发的各种命令，用来操作 MCU。
3) 其他的配置：oe，dataO 都为 0。

设为以上状态后，可以随时接收 EVK 的操作命令，但同时也带来一个风险：当整个系统的电源抖动很厉害的时候（如发送红外时，瞬间电流可能会冲到接近 100mA），由于 SWS 处于 float 状态，可能会读到一个错误的数据，误以为是 EVK 发来的命令，这个错误的命令可能会导致程序挂掉。

解决上面问题的方法是，将 SWS 的 float 状态修改为输入上拉。通过模拟上拉 1M 电阻来解决。

17H25 的 SWS 和 GPIO_PA0 复用，在 proj/mcu_spec/gpio_default.h 中将 PA0 的 1M 上拉开启即可：

```
#ifndef PULL_WAKEUP_SRC_PA0
#define PULL_WAKEUP_SRC_PA0     PM_PIN_PULLUP_1M  //sws pullup
#endif
```

3 BLE 模块

3.1 BLE SDK 软件架构

3.1.1 标准 BLE SDK 软件架构

根据 BLE spec，一个比较标准的 BLE SDK 架构如下图所示。

Lenze Technology Co.,LTD

图 3-1 BLE SDK 标准架构

在上图所示的架构中，BLE 协议栈分为 Host 和 Controller 两部分。

Controller 作为 BLE 底层协议，包括 Physical Layer（PHY）和 Link Layer（LL）。Host Controller Interface（HCI）是 Controller 与 Host 的唯一通信接口，Controller 与 Host 所有的数据交互都通过该接口完成。

Host 作为 BLE 上层协议，协议上有 Logic Link Control and Adaption Protocol（L2CAP）、Attribute Protocol（ATT）、Security Manager Protocol（SMP），Profile 包括 Generic Access Profile（GAP）、Generic Attribute Profile（GATT）。

应用层（APP）包含 user 自己相关应用代码和各种 service 对应的 Profile，user 通过 GAP 去控制访问 Host。

Host 通过 HCI 与 Controller 完成数据交互，如下图所示。

Lenze Technology Co.,LTD

图 3-2 Host 和 Controller 的 HCI 数据交互

1) BLE Host 通过 HCI cmd 去操作设置 Controller，这些 HCI cmd 对应本章后面将要介绍的 controller API。
2) Controller 通过 HCI 向 host 上报各种 HCI event，本章也会具体介绍。
3) Host 将需要发送给对方设备的数据通过 HCI 传送到 Controller，Controller 将数据直接丢到 Physical Layer 进行发送。
4) Controller 在 Physical Layer 收到的 RF 数据，先判断是属于 Link Layer 的数据还是 Host 的数据：如果是 Link Layer 的数据，直接处理数据；如果是 Host 的数据，则通过 HCI 将数据传到 Host。

3.1.2 Lenze BLE SDK 软件架构

3.1.2.1 Lenze BLE controller

Lenze BLE SDK 支持标准的 BLE controller，包括 HCI、PHY（Physical Layer）和 LL（link layer）。

Lenze BLE SDK 包含 Link Layer 的五种标准状态（standby、advertising、scanning、initiating、connection），conneciton 状态下的 Slave role 和 Master role 也都支持。目前 Slave role 和 Master role 都只是 single connection，即 Link Layer 只能维持一个连接，无法同时多个 Slave/Master 或者 Slave 和 Master 同时存在。

SDK 上的 17H25 hci 是一个 BLE slave 的 controller，需要和另外一个运行 BLE Host 的 MCU 协调工作形成一个标准的 BLE Slave 系统，架构图如下。

Lenze Technology Co.,LTD

图 3-3 17H25 hci 架构

Link Layer connection 状态同时支持 single connection 的 Slave 和 Master，那么 17H25 hci 实际也可以作为 BLE master controller 使用，但实际对于一个 BLE host 运行在较复杂的系统（如 linux/android）来说，单一连接的 Master controller 只能连接一个设备，几乎是没有意义的，所以 SDK 在 17H25 hci 上并没有将 master role 的初始化放进去。

3.1.2.2 Lenze BLE slave

Lenze BLE SDK 在 BLE host 上，只对 Slave 部分的 stack 完全支持；对于 Master 无法做到完全支持，因为 SDP（service discovery）太复杂。

当 user 只需要使用标准的 BLE slave 时，Lenze BLE SDK 运行 Host（slave 部分） + 标准的 Controller，实际的协议栈架构会对上面标准的结构做一些简化处理，使得整个 SDK 的系统资源开销（包括 sram、运行时间、功耗等）最小，其架构如下图所示。SDK 中 17H25 hid sample、17H25 remote、17H25 module 都是基于该架构。

Lenze Technology Co.,LTD

图 3-4 Lenze BLE slave 架构

图中实心箭头所示的数据交互是 user 可以通过各种接口来操作控制的，会提供 user API。空心箭头是协议栈内部完成的数据交互，user 无法参与。

Controller 部分 HCI 仍然是与 Host 的数据通信接口（和 L2CAP 层对接），但不是唯一的接口，APP 应用层也可以直接与 Link Layer 进行数据交互。Power Management（PM）低功耗管理单元被内嵌到 Link layer，应用层可以调用 PM 相关接口进行功耗管理的设置。

将 Host 层的 Generic Access Profile 具体实现部分去掉，只保留应用层中该 profile 的 service 声明。应用层与 Host 的数据交互不再是通过 GAP 来访问控制，协议栈在 ATT、SMP 和 L2CAP 都提供了相关接口，可以和应用层直接交互。

Host 层以 Attribute Protocol 为基础，实现了 Generic Attribute Profile（GATT）。应用层基于 GATT，定义 user 自己需要的各种 profile 和 service。17H25 BLE SDK demo code 提供几个基本的 profile，包括 HIDS、BAS、OTA 等。

下面基于这个架构对 17H25 BLE 协议栈各部分做一些基本的介绍，并给出各层的 user API。

其中 Physical Layer 完全由 Link Layer 控制，且不需要应用层任何的参与，这部分不介绍。

虽然 Host 与 Controller 的部分数据交互还是靠 HCI 来完成，但基本都是 Host 和 Controller 协议栈完成，应用层几乎不参与，只需要在 L2CAP 层注册 HCI 数据回调处理函数就行了，所以对 HCI 部分也不做介绍。

3.2 BLE controller

3.2.1 BLE controller 介绍

BLE controller 包括 Physical Layer、Link Layer 和 Power Management。

Lenze BLE SDK 对 Physical Layer（对应 driver 文件中的 rf_drv 的 c 文件）完全封装到 library

Lenze Technology Co.,LTD

中，user 不需要了解。Power Management 将在本文档后面的章节中详细介绍，本章只稍微提一下，不做过多介绍。

本章只介绍 Link Layer，并在介绍 Link Layer 和 API 的过程中对 HCI 相关 interface 一一描述。

3.2.2 Link Layer 状态机

下图为 BLE spec 中 Link Layer 状态机。参考《Core_v4.2》Page2574 1.1 LINK LAYER STATES。

图 3-6 State diagram of the Link Layer state machine in BLE Spec

Lenze BLE SDK Link Layer 状态机如下图所示。

Lenze Technology Co.,LTD

图 3-7 Lenze Link Layer state machine

Lenze BLE SDK Link Layer 状态机和 BLE spec 定义的一致，拥有 5 种基本状态：Idle（Standby）、Scanning、Advertising、Initiating、Connection。将 Conneciton 状态区分为 Slave Role 和 Master Role。

由文档前面对 libary 的介绍，目前的 Slave Role 和 Master Role 都是基于 single connection 的设计，Slave Role 默认为 single connection 的；Master Role 目前提供的是 single connection 的，但由于将来会提供 multi connection 的，所以这里名为 Master role single conection，和将来的 Master Role multi connection 作区分。

本文档中将 Slave Role 后面称为 Conn state Slave role 或 ConnSlaveRole/Connection Slave Role 或简写为 ConnSlaveRole。

本文档中将 Master Role 后面称为 Conn state Master role 或 ConnMasterRole/Connection Master Role 或简写为 ConnMasterRole。

图中 Power Management 并不是 LL 的一种状态，而是功能模块，表示的是 SDK 只对 Advertising 和 Connection Slave Role 进行了低功耗处理；Idle state 如果需要低功耗，user 在应用层调用相关 API 可以完成；其他几个 state，SDK 没有做低功耗管理，user 也无法在应用层实现低功耗。

基于上面 5 种状态，proj_lib/ble/ll/ll.h 中定义了状态机的命名。其中 ConnSlaveRole 和 ConnMasterRole 时状态名都是 BLS_LINK_STATE_CONN。

```
//ble link layer state
#define          BLS_LINK_STATE_IDLE          0
#define          BLS_LINK_STATE_ADV           BIT(0)
#define          BLS_LINK_STATE_SCAN          BIT(1)
#define          BLS_LINK_STATE_INIT          BIT(2)
#define          BLS_LINK_STATE_CONN          BIT(3)
```


Lenze Technology Co.,LTD

Link Layer 状态机的切换都在 BLE stack 底层自动完成，所以 user 在应用层无法去修改状态，只能获取当前状态，调用下面 API 即可，返回值为上面 5 种状态中的 1 种。

```
u8          blc_ll_getCurrentState(void);
```

3.2.3 Link Layer 状态机组合应用

3.2.3.1 Link Layer 状态机初始化

Lenze BLE SDK Link Layer 完整的支持所有状态，但在设计上很灵活，每一个状态都封装成为一个模块，默认情况下只有最基本的 Idle 模块，user 通过添加模块的方式去搭建自己的状态机组合，从而实现不同的应用。比如 user 需要的应用是 BLE slave，那么只需要添加 Advertising 模块和 ConnSlaveRole 模块就可以了，剩下的 Scanning/Initiating 等模块没有被配置进来。这样的设计的目的是节省 code size 和 ramcode，不需要使用的状态的相关代码不会被编译进来。

Idle 模块的添加 API 如下，这个是必须的，所有的 BLE 应用都需要初始化。

```
void        blc_ll_initBasicMCU (u8 *public_adr);
```

其他几个状态（Scanning、Advertising、Initiating、Slave Role、Master Role Single Connection）对应模块的初始化 API 分别如下：

```
void        blc_ll_initAdvertising_module(u8 *public_adr);
void        blc_ll_initScanning_module(u8 *public_adr);
void        blc_ll_initInitiating_module(void);
void        blc_ll_initSlaveRole_module(void);
void        blc_ll_initMasterRoleSingleConn_module(void);
```

上面 API 中实参 public_adr 都是 BLE public mac address 的指针。

User 通过以上几个 API 去配合组合 Link Layer 状态机，下面给出几个常用的组合方式和对应的应用场景，但不仅限于以下几种组合，user 可以自行配置组合。

3.2.3.2 Idle + Advertising

图 3-8 Idle + Advertising

上图所示，只初始化 Idle 和 Advertising 模块，使用最基本的 Advertising 功能实现的应用如 beacon 等，单方向广播产品信息。

Link Layer 状态机模块初始化代码为：

```
u8  tbl_mac [6] = {……};
blc_ll_initBasicMCU(tbl_mac);
 blc_ll_initAdvertising_module(tbl_mac);
```


Lenze Technology Co.,LTD

Idle 和 Advertising 状态的切换通过 bls_ll_setAdvEnable 来实现。

3.2.3.3 Idle + Scannning

图 3-9 Idle + Scanning

上图所示，只初始化 Idle 和 Scanning 模块，使用最基本的 Scanning 功能实现 beacon 等产品广播信息的扫描发现。

Link Layer 状态机模块初始化代码为：

```
 u8  tbl_mac [6] = {……};
blc_ll_initBasicMCU(tbl_mac);
blc_ll_initScanning_module( tbl_mac);
```

Idle和Scanning状态的切换通过blc_ll_setScanEnable来实现。

Lenze Technology Co.,LTD

3.2.3.4 Idle + Advtersing + ConnSlaveRole

图 3-10 BLE slave LL state

上图所示为一个基本的 BLE slave 应用的 Link Layer 状态机组合，SDK 中 17H25 remote/17H25 module 都是基于该状态机组合。

Link Layer 状态机模块初始化代码为：

```
u8  tbl_mac [6] = {……};
blc_ll_initBasicMCU(tbl_mac);
blc_ll_initAdvertising_module(tbl_mac);
blc_ll_initSlaveRole_module();
```

该状态机组合下，状态变化情况描述如下。

1) 17H25 MCU 上电后，进入 Idle state。Idle state 时将 adv Enable，Link Layer 切换到 Advertising State；adv Disable 时，回到 Idle state。
   Adv Enable 和 Disable 通过下面 API bls_ll_setAdvEnable 来控制。
   上电后，Link layer 默认处于 Idle 状态，一般需要在 user_init 里面将 Adv Enable，进入 Advertising state。
2) Link Layer 处于 Idle state 时，Physical Layer 不进行任何 RF 相关动作，不收包也不发包。
3) Link Layer 处于 Advertising state 时，在广播 channel 上发送广播包。若 master 收到广播包，并发送 conneciton request，Link Layer 收到这个 connection request 后，响应并建立连接，Link Layer 进入 ConnSlaveRole。

4) Link Layer 处于 ConnSlaveRole 时，有三种情况会回到 Idle State 或者 Advertising state：
   a) master 向 slave 发送 terminate 命令，断开连接。slave 收到 terminate 命令，退出 ConnSlaveRole。
   b) slave 向 master 发送 terminate 命令，主动断开连接，退出 ConnSlaveRole。

Lenze Technology Co.,LTD

c) slave 的 RF 收包异常或 master 发包异常，导致 slave 长时间收不到包，触发 BLE 的 connection supervision timeout，退出 ConnSlaveRole。

Link layer 的 ConnSlaveRole 退出该 state 时，根据 Adv 是否被 Enable 切换到不同 state：若 Adv 被 enable，Link Layer 重新回到 Advertising state；若 Adv 被 Disable，Link Layer 回到 Idle state。Adv 处于 Enable 或者 Disable 取决于 user 在应用层最后一次调用 bls_ll_setAdvEnable 时设置的值。

3.2.4 Link Layer 时序

结合 17H25 BLE SDK 的 irq_handler 和 mainloop 来说明 Link layer 在各状态的工作时序。

```
_attribute_ram_code_ void irq_handler(void)
{
    ......
    irq_blt_sdk_handler ();
    ......
}

void main_loop (void)
{
    ////////////////////// BLE entry ////////////////////////////
    blt_sdk_main_loop();
    ////////////////////// UI entry ////////////////////////////
    ......
}
```

BLE entry 部分 blt_sdk_main_loop 函数负责处理 BLE 协议栈相关的数据和事件。UI entry 是 user 写自己的应用代码的地方。

3.2.4.1 Idle state 时序

当 Link Layer 处于 Idle state 时，Link Layer 和 Physical Layer 没有任何任务要处理，blt_sdk_main_loop 函数完全不起作用，也不会产生任何中断。可以认为 UI entry 占据了整个 mainloop 的时序。

Lenze Technology Co., LTD

3.2.4.2 Advertising state 时序

图 3-12 Advertising State 时序

Advertising state 时序图如上所示。每一个 adv interval 时间内 Link Layer 触发一个 Adv event。一个典型的 3 个 adv channel 都发包的 Adv event 会在 channel 37、38、39 上都发送一个广播包。每发一个广播包之后都进入收包状态，等待 master 可能的回包，回包有两种：一个是 scan request，收到这个包后再发送一个 scan response 作为回复；另一个是 connect reqeust，收这个包后和 master 建立 ble 连接，进入 Connection state Slave Role。

对于 adv event 的 code 执行，不同版本的 SDK 在设计上有些不一致：

1) BLE SDK 3.0 和 3.1 版本上，整个 adv event 的 code 执行都是在 interrupt irq 中进行的，每隔一个 adv interval，MCU 都会进入中断去在 3 个 channel 上发包和收包。adv event 定时进入 irq 处理，是以本文档前面介绍的 system tick 的时间为基准，由 system tick irq 触发的。
2) 从 BLE SDK 3.2 版本开始，adv event 的 code 执行从 interrupt irq 中挪出来，不再占用 irq 时间，在 mainloop 的 blt_sdk_main_loop 函数中处理。这样的改进可以使得 user 不用再担心 adv event 占用太多 irq 时间导致自己的一些 irq 无法实时响应。

user 在 mainloop 的 UI entry 部分的 code 执行会在图中所示的 UI task/suspend 部分执行，这部分时间全部可以用来做 UI task，如果需要低功耗的话，可以将多余的时间用来 suspend 以降低功耗。

在 Advertising state，blt_sdk_main_loop 函数没有太多要处理的任务，只是触发几个跟 Adv 相关的几个回调事件，包括 BLT_EV_FLAG_ADV_DURATION_TIMEOUT、BLT_EV_FLAG_SCAN_RSP、BLT_EV_FLAG_CONNECT 等。

3.2.4.3 Scanning state 时序

图 3-13 Scanning state 时序

Scanning state 时序图如上所示，Scan interval 是由 API blc_ll_setScanParameter 设置。整个 Scan interval 对一个 channel 进行收包，没有将 Scan window 设计到 SDK 里，所以

Lenze Technology Co.,LTD

blc_ll_setScanParameter 中对 Scan window 的设置 SDK 是不处理的。

每个 Scan interval 结束后，切换到下一个听包的 Channel，开始新一轮的 Scan interval。切换 channel 的动作是 interrupt 触发的，在 irq 中完成该动作，其执行时间非常短。

Scanning interval 上，Scan 状态 PHY 层一直处于 RX 状态，靠 MCU 硬件去实现收包操作，所以软件上的 timing 都留给了 UI task。

在 Scan interval 上收到正确的 BLE packet 后，将收包数据先缓存在软件 RX fifo 中（对应 code 中的 my_fifo_t blt_rxfifo），blt_sdk_main_loop 函数会检查软件 RX fifo 中是否有数据，如果发现正确的 adv 数据会通过 event“HCI_SUB_EVT_LE_ADVERTISING_REPORT”将其 report 给 BLE host。

3.2.4.4 Initiating state 时序

图 3-14 Initiating state 时序

Initiating state 时序图如上所示，和 Scanning state 时序是一样的，不同点是 Scan interval 由 API blc_ll_createConnection 设置。整个 Scan interval 对一个 channel 进行收包，没有将 Scan window 设计到 SDK 里，所以 blc_ll_createConnection 中对 Scan window 的设置 SDK 是不处理的。

每个 Scan interval 结束后，切换到下一个听包的 Channel，开始新一轮的 Scan interval。切换 channel 的动作是 interrupt 触发的，在 irq 中完成该动作，其执行时间非常短。

Scanning state 中 BLE controller 将收到的 adv packet report 给 BLE host，而 Initiating state 不会 report adv 给 BLE host，它只是 Scan 到由 blc_ll_createConnection 指定的设备后，发送 connection_request 并建立连接，然后 Link Layer 进入 ConnMasterRole。

3.2.4.5 Conn state Slave role 时序

图 3-15 Conn state Slave role 时序

ConnSlaveRole 时序图如上所示。每一个 conn interval 开始的时候，Link Layer 进行一次 BLE 的 RF 包收发过程：先让 PHY 进入收包状态，收到 master 的包后发送一个 ack 包回复，若有 more data，则继续收 master 包并回复，这个过程简称为 brx event。

在 17H25 BLE SDK 中，根据软硬件的工作分配，每个 brx 过程分为 3 个阶段：

1) brx start 阶段

当 master 发包的时间快要到来时，会由 system tick irq 触发进入 brx start 阶段，在这个中断里 MCU 设置 PHY 的 BLE 状态机进入 brx 状态，底层硬件做好收发包的相关准备，然后退出

Lenze Technology Co., LTD

中断 irq。

2) brx working 阶段

brx start 结束后，MCU 退出 irq 后，底层硬件进入收包状态，并自动完成收发包的所有工作，不需要软件任何的参与，这个过程称为 brx working 阶段。

3) brx post 阶段

收发包完成后，brx working 结束，同样由 system tick irq 触发进入中断执行 brx post 阶段。这个阶段主要是协议栈根据 brx working 阶段收发包的情况对 BLE 的一些数据和时序进行相关的处理。

上面三个阶段中 brx start 和 brx post 都是中断完成，而 brx working 阶段不需要软件的参与，此时 UI task 可以正常执行。brx working 时间内，硬件需要进行收发包，所以不能进 suspend。

整个 conn interval 内除去 brx 过程的时间，可以用来做 UI task，如果需要低功耗的话，可以将多余的时间用来 suspend 以降低功耗。

在 ConnSlaveRole，blt_sdk_main_loop 需要处理 brx 过程收到的数据。brx working 过程中，实际是在 RX 接收中断 irq 处理中将硬件收到的 master 数据包拷贝出来，这些数据并不会立刻实时处理，而是将数据缓存到软件 RX fifo 中（对应 code 中的 my_fifo_t blt_rxfifo）。blt_sdk_main_loop 函数会检查软件 RX fifo 中是否有数据，只要有数据就去处理。

blt_sdk_main_loop 对数据包的处理包括：

1) 数据包的解密
2) 数据包的解析

解析的数据若发现是属于 master 发给 Link Layer 的控制命令，立即执行该命令，若是 master 发给 Host 层的数据，则会通过 HCI 接口将数据丢到 L2CAP 层处理。

3.2.4.6 Conn state Slave role 时序保护

ConnSlaveRole，每个 interval 需要一个收发包事件，也就是上面的 Brx Event。17H25 SDK 中，Brx Event 完全是由中断触发的，所以 MCU 系统总中断需要一直被打开。如果 user 在 Conn state 的一些任务时间较长且必须把系统总中断关闭（比如擦除 Flash），就会造成 Brx Event 被停掉，BLE 的时序很快就乱掉，最终连接断开。

对于这种情况 SDK 中提供了一套保护机制，让 user 去停掉 Brx Event 却不破坏 BLE 的时序，user 需要严格根据这个机制来操作。相关 API 如下：

```
int        bls_ll_requestConnBrxEventDisable(void);
void       bls_ll_disableConnBrxEvent(void);
void       bls_ll_restoreConnBrxEvent(void);
```

调用 bls_ll_requestConnBrxEventDisable 来申请关掉 Brx Event。

1) 该 API 返回值若为 0，表示当前不接受用户的申请，即此时不能停掉 Brx Event。在 Conn state 时的 Brx working 阶段，不能接受申请，返回值为 0，一定要等到一个完整的 Brx Event 结束后，在剩余的 UI task/suspend 时间内才会接受申请。
2) 该 API 返回非 0 值表示可以接受申请，返回的值是允许停掉 Brx Event 的时间，单位为 ms。该事件值有三种情况：
   A. 若当前 Link Layer 为 Advertising state 或 Idle state，返回值为 0xffff，即没有 Brx Event，用户关闭系统中断的时间随便多长都可以。
   B. 若当前为 Conn state，收到了 master 的 update map 或 update connection parameter 且还没有到更新的时间点时，返回时间为更新的时间点减去当前时间。即停掉 Brx Event

Lenze Technology Co.,LTD

的时间不能超过更新的时间点，否则会造成后面所有包收不到，最终断开连接。

C. 若当前为 Conn state，且没有 master 的更新请求，返回值为当前 connection supervision timeout 值的一半。比如当前 timeout 为 1S，返回值为 500ms。

user 调用上面的 API 申请停掉 Brx Event，若返回值对应的时间(ms)，足够自己的任务运行时间，即可进行该任务。在该任务执行之前，调用 API bls_ll_disableConnBrxEvent 停掉 Brx Event。任务结束后，调用 API bls_ll_restoreConnBrxEvent 重新开启 Brx Event 并修复 BLE 时序。

参考使用方法如下。其中具体时间的判断，以测试到的实际任务时间为准。

```
        if(bls_ll_requestConnBrxEventDisable() > 300)
        {

            bls_ll_disableConnBrxEvent();

#if 0  //test 1
            irq_disable();
            DBG_CHN3_HIGH;
            sleep_us(287*1000);
            DBG_CHN3_LOW;
            irq_enable();
#else  //test 2
            DBG_CHN3_HIGH;
            flash_erase_sector(0x40000);
            DBG_CHN3_LOW;
#endif

            bls_ll_restoreConnBrxEvent();


        }
```

3.2.5 Link Layer 状态机扩展

上面 BLE Link Layer 状态机和工作时序介绍了最基本的几种状态，能够满足 BLE slave/master 等基本应用。

但是考虑到 user 可能会有的一些特殊的应用（比如在 Conn sate Slave role 时还要能够 advertising），Lenze BLE SDK 对 Link Layer 的状态机添加了一些特殊的扩展的功能，下面详细描述。

3.2.5.1 Scanning in Advertising state

在 Link Layer 处于 Advertising state 时，可以添加 Scanning feature。

添加 Scanning feature 的 API 为：

```
ble_sts_t    blc_ll_addScanningInAdvState(void);
```

去掉 Scanning feature 的 API 为：

```
ble_sts_t    blc_ll_removeScanningFromAdvState(void);
```

以上两个 API ble_sts_t 类型的返回值都是 BLE_SUCCESS。

结合 Advertising state 和 Scanning state 的时序图，当加入 Scanning feature 到 Advertising state 后，时序图如下。

图 3-17 Scanning in Advertising state 时序

当前 Link Layer 还是处于 Advertising state（BLS_LINK_STATE_ADV），在每个 Adv interval 内，除去 Adv event 剩余的时间全部用来做 Scanning。

在每个 Set Scan 的时候，会判断一下当前时间距离上次 Set Scan 时间点是否超过一个 Scan interval（来自 blc_ll_setScanParameter 的设定），若超过则切换 Scan 的 channel（channel 37/38/39）。

Scanning in Advertising state 的使用请参考 17H25 feature test 中 TEST_SCANNING_IN_ADV_AND_CONN_SLAVE_ROLE。

3.2.5.2 Scanning in ConnSlaveRole

在 Link Layer 处于 ConnSlaveRole 时，可以添加 Scanning feature。

添加 Scanning feature 的 API 为：

```
ble_sts_t    blc_ll_addScanningInConnSlaveRole(void);
```

去掉 Scanning feature 的 API 为：

```
ble_sts_t    blc_ll_removeScanningFromConnSLaveRole(void);
```

以上两个 API ble_sts_t 类型的返回值都是 BLE_SUCCESS。

结合 Scanning state 和 ConnSlaveRole 的时序图，当加入 Scanning feature 到 ConnSlaveRole 后，时序图如下。

图 3-18 Scanning in ConnSlaveRole 时序

当前 Link Layer 还是处于 ConnSlaveRole（BLS_LINK_STATE_CONN），在每个 Conn interval 内，除去 brx event 剩余的时间全部用来做 Scanning。

在每个 Set Scan 的时候，会判断一下当前时间距离上次 Set Scan 时间点是否超过一个 Scan interval（来自 blc_ll_setScanParameter 的设定），若超过则切换 Scan 的 channel（channel 37/38/39）。

Scanning in ConnSlaveRole 的使用请参考 17H25 feature test 中 TEST_SCANNING_IN_ADV_AND_CONN_SLAVE_ROLE。

3.2.5.3 Advertising in ConnSlaveRole

在 Link Layer 处于 ConnSlaveRole 时，可以添加 Advertising feature。

添加 Advertising feature 的 API 为：

```
ble_sts_t    blc_ll_addAdvertisingInConnSlaveRole(void);
```

去掉 Advertising feature 的 API 为：

Lenze Technology Co.,LTD

```
ble_sts_t   blc_ll_removeAdvertisingFromConnSLaveRole(void);
```

以上两个 API ble_sts_t 类型的返回值都是 BLE_SUCCESS。

结合 Advertising 和 ConnSlaveRole 的时序图，当加入 Advertising feature 到 ConnSlaveRole 后，时序图如下。

图 3-19 Advertising in ConnSlaveRole 时序

当前 Link Layer 还是处于 ConnSlaveRole(BLS_LINK_STATE_CONN)，在每个 Conn interval 内 brx event 结束后，立刻执行一次 adv event，然后剩余的时间留给 UI task 或进入 suspend 节省功耗。

Advertising in ConnSlaveRole 的使用请参考 17H25 feature test 中 TEST_ADVERTISING_IN_CONN_SLAVE_ROLE。

3.2.5.4 Advertising and Scanning in ConnSlaveRole

结合上面 Scanning in ConnSlaveRole 和 Advertising in ConnSlaveRole 的使用，可以在 ConnSlaveRole 中同时加入 Scanning 和 Advertising。时序图如下：

图 3-20 Advertising and Scanning in ConnSlaveRole 时序

当前 Link Layer 还是处于 ConnSlaveRole(BLS_LINK_STATE_CONN)，在每个 Conn interval 内 brx event 结束后，立刻执行一次 adv event，然后剩余的时间用来做 Scanning。

在每个 Set Scan 的时候，会判断一下当前时间距离上次 Set Scan 时间点是否超过一个 Scan interval(来自 blc_ll_setScanParameter 的设定)，若超过则切换 Scan 的 channel(channel 37/38/39)。

Advertising and Scanning in ConnSlaveRole 的使用请参考 17H25 feature test 中 TEST_ADVERTISING_SCANNING_IN_CONN_SLAVE_ROLE。

3.2.6 Link Layer TX fifo & RX fifo

应用层和 BLE Host 所有的数据最终都需要通过 Controller 的 Link Layer 完成 RF 数据的发送，在 Link Layer 中设计了一个 BLE TX fifo，可以用来缓存传过来的数据，并在 brx/btx 开始后

Lenze Technology Co.,LTD

进行数据发送。

Link Layer brx/btx 时收到的所有 peer device 的数据都会先存放在一个 BLE RX fifo 中，然后才上传给 BLE Host 或应用层处理。

Slave role 和 Master role 的 BLE TX fifo 和 BLE RX fifo 的处理有些区别，下面分别说明。

3.2.6.1 Slave role fifo

Slave role 中 BLE TX fifo 和 BLE RX fifo 都在应用层定义：

```
MYFIFO_INIT(blt_rxfifo, 64, 8);
MYFIFO_INIT(blt_txfifo, 40, 16);
```

其中 RX fifo size 默认为 64，TX fifo size 默认为 40，除非需要使用 core 4.2 的 data length extension，否则不允许修改这两个 size。

不管是 TX fifo number 还是 RX fifo number，必须设置为 2 的幂，即 2、4、8、16 等值。User 可以根据自己的需要稍作修改。

RX fifo number 默认为 8，这是一个比较合理的值，能够确保 Link Layer 底层最多缓存 8 个数据包。如果设置的太大，会占用过多的 Sram，如果设置的太小，可能出现数据覆盖的风险：在 brx event 时，Link Layer 很可能在一个 interval 上出现 more data 模式，连续收多个包，如果设置 4 的话，很可能在一个 interval 上出现五六个包（比如 OTA、播放 master 语音数据等情况），而上层对这些数据的响应由于解密时间较长来不及处理，那么就有可能有一些数据被 overflow。同理如果可能出现一个 interval 多于 8 个有效数据包的话，默认的 8 也就不够用了。

TX fifo number 默认为 16，能够处理数据量较大的语音遥控器功能。User 如果用不到这么大的 fifo，可以修改为 8。

如果设置太大（如 32）会占用过多的 Sram。

TX fifo 中，SDK 底层 stack 需要用到 2 个，剩下的才完全由应用层使用，TX fifo 为 16 时，应用层只能用 14 个；为 8 时应用层只能用 6 个。

User 在应用层发送数据时（比如调用 bls_att_pushNotifyData），应该先检查一下当前 Link Layer 还有多少 TX fifo 可用。

下面 API 用于判断当前 TX fifo 被占用了多少个，注意不是剩余多少个可用。

```
u8          bls_ll_getTxFifoNumber (void);
```

比如 TX fifo number 默认为 16 时，user 可用为 14 个，所以该 API 返回的值只要小于 14，就是可用的：返回 13 表示还有 1 个可用，返回 0 则表示还有 14 个可用。

在 17H25 remote 的语音处理中，由于已知每笔语音数据会被拆成 5 个包，需要 5 个 TX fifo，所以实现方法如下，被占用的 fifo 不能超过 9 个。

```
if (blc_ll_getTxFifoNumber() < 10)
{
    ......
}
```

3.2.7 Lenze defined event

SDK 提供了 Lenze defined event。架构如下图所示。

其特点是以 user 的 application 的角度来看的，event 来自 Host 和 Controller 两部分（相当于整个 BLE stack），其中绝大部分来自 Controller 部分，所以在本节介绍，少数的几个来自 Host

Lenze Technology Co.,LTD

的 event 将会在 Host 部分对应介绍。

图 3- 27 Lenze defined event 架构

Lenze defined event 最多支持 20 个，在 proj_lib/ble/ll/ll.h 中用宏来定义。

目前最新版本的 SDK 中支持以下回调事件。其中 BLT_EV_FLAG_CONNECT/BLT_EV_FLAG_TERMINATE 可理解为 HCI_SUB_EVT_LE_CONNECTION_COMPLETE /HCI_EVT_DISCONNECTION_COMPLETE 功能上是重复的，只是 event 数据部分定义不同。

```
#define        BLT_EV_FLAG_ADV                  0
#define        BLT_EV_FLAG_ADV_DURATION_TIMEOUT 1
#define        BLT_EV_FLAG_SCAN_RSP             2
#define        BLT_EV_FLAG_CONNECT              3
#define        BLT_EV_FLAG_TERMINATE            4
#define        BLT_EV_FLAG_PAIRING_BEGIN        5
#define        BLT_EV_FLAG_PAIRING_END          6
#define        BLT_EV_FLAG_ENCRYPTION_CONN_DONE  7
#define        BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 8
#define        BLT_EV_FLAG_GPIO_EARLY_WAKEUP    9
#define        BLT_EV_FLAG_CHN_MAP_REQ          10
#define        BLT_EV_FLAG_CONN_PARA_REQ        11
#define        BLT_EV_FLAG_CHN_MAP_UPDATE       12
#define        BLT_EV_FLAG_CONN_PARA_UPDATE     13
#define        BLT_EV_FLAG_SUSPEND_ENTER        14
#define        BLT_EV_FLAG_SUSPEND_EXIT         15
#define        BLT_EV_FLAG_READ_P256_KEY        16
#define        BLT_EV_FLAG_GENERATE_DHKEY       17
```

Lenze defined event 只在 BLE slave 相关应用中（remote/module 等）有意义，在 BLE master 中不会被触发。本文档前面对 library 的介绍，BLE slave 类的 library 从功能上分为主控类（控制核心 + 蓝牙）和透传类（17H25 module，仅蓝牙），对于这两类不同功能的 library，SDK 中 App 层对这些 event 回调的处理也不同。

对于主控类 library，如 lib_17H26 等，回调函数的原型说明为：

typedef void (*blt_event_callback_t)(u8 e, u8 *p, int n);

Lenze Technology Co.,LTD

其中 e 是 event number；p 为回调函数执行时，底层传上来相关数据的指针，不同回调函数传上来的指针数据都不一样；n 是回传指针所指的有效数据长度。

注册回调函数的 API 为：

```
void bls_app_registerEventCallback (u8 e, blt_event_callback_t p);
```

每一个event是否响应，取决于应用层是否有对应的回调函数被注册。

以 BLT_EV_FLAG_CONNECT 来举例说明注册函数的用法，该 event 表示的是：广播设备在发广播包时，Master 发送 connection request 请求连接，广播设备收到该 connection request 并处理，进入 Conn state Slave role，此时 stack 检查一下 BLT_EV_FLAG_CONNECT 事件是否注册了回调函数，若被注册了，就调用注册的函数进行相关的操作。这个操作就是 user 自己定义的，user 可以在设备进入连接的时候进行一些记录或是相关的设置。

```
void task_connect (u8 e, u8 *p,   int n)
{
    // add your code
}
bls_app_registerEventCallback (BLT_EV_FLAG_CONNECT,   &task_connect);
```

根据以上设定，BLE slave 每次收到 connection request 并进入连接状态时都会调用一次 task_connect 函数。

对于透传类 library，如 libH25-sdk.a ，函数原型和注册 callback 函数都跟 HCI event 一致：

```
typedef int (*hci_event_handler_t) (u32 h, u8 *para, int n);
void    blc_hci_registerControllerEventHandler   (hci_event_handler_t
                handler);
```

不同点是u32 h的实际值为：

```
h = HCI_FLAG_EVENT_TLK_MODULE  |  e;
#define          HCI_FLAG_EVENT_TLK_MODULE          (1<<24)
```

e 是 Lenze defined event 的 event number。

类似于 HCI event 的 mask 方法，每一个 event 是否被响应，由下面 API 来设置其 mask：

```
ble_sts_t  bls_hci_mod_setEventMask_cmd(u32 evtMask)
```

evtMask 的值和 event number 的对应关系为：

```
evtMask = BIT(e);
```

User 可参考 17H25 module 中对 Lenze defined event 的处理来理解。

Lenze Technology Co.,LTD

下面对 Controller 所有的事件、事件触发条件、对应的回调函数的参数进行详细的说明。BLT_EV_FLAG_PAIRING_BEGIN 和 BLT_EV_FLAG_PAIRING_END 事件不属于 Controller，会在 Host SMP 中具体介绍。

3.2.7.1 BLT_EV_FLAG_ADV

该事件目前没有使用。

3.2.7.2 BLT_EV_FLAG_ADV_DURATION_TIMEOUT

事件触发条件：如果 user 调用 API bls_ll_setAdvDuration 设置了广播时间限制，BLE 协议栈底层启动一个计时，在这个限时达到后，停止广播，同时触发该事件，user 可以在该事件回调函数中进行修改广播事件类型、重新打开广播使能、再次设置广播时间限制等操作。

回传指针 p：空指针 NULL。

数据长度 n： 0。

注意：该 event 在 Link Layer 扩展状态中的 advertising in ConnSlaveRole 不触发。

3.2.7.3 BLT_EV_FLAG_SCAN_RSP

事件触发条件：slave 处于广播状态，收到 master 的 scan request 并响应，回 scan response 时触发该事件。

回传指针 p：空指针 NULL。

数据长度 n： 0。

3.2.7.4 BLT_EV_FLAG_CONNECT

事件触发条件：Link Layer 处于广播状态时收到 master 的 connect reqeust，响应这个请求，进入 Conn state Slave role，触发该事件。

数据长度 n： 34。

回传指针 p:p 指向长度为 34 byte 的一片 ram 区域,对应如下图所示的 connect request PDU。

Payload		
InitA (6 octets)	AdvA (6 octets)	LLData (22 octets)

Figure 2.10: CONNECT_REQ PDU payload

The format of the LLData field is shown in Figure 2.11.

LLData									
AA (4 octets)	CRCInit (3 octets)	WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	ChM (5 octets)	Hop (5 bits)	SCA (3 bits)

Figure 2.11: LLData field structure in CONNECT_REQ PDU's payload

图 3- 28 connect request PDU

可以参考 ble_common.h 中 rf_packet_connect_t 的定义，connect request PDU 从该结构体的 scanA[6]（对应上图中的 InitA）开始，到 hop 结束。

```
typedef struct{
```

```
        u32 dma_len;
        u8  type;
        u8 rf_len;
        u8  scanA[6];
        u8  advA[6];
        u8  accessCode[4];
        u8  crcinit[3];
        u8  winSize;
        u16 winOffset;
        u16 interval;
        u16 latency;
        u16 timeout;
        u8  chm[5];
        u8  hop;
    }rf_packet_connect_t;
```

3.2.7.5 BLT_EV_FLAG_TERMINATE

事件触发条件：Link Layer 状态机中从 Conn state Slave role 退出时触发。

回传指针 p：p 指向一个 u8 类型的变量 terminate_reason，该变量表明当前是什么 reason 导致的 Link Layer 断开连接。

数据长度 n： 1。

Conn state Slave role 退出的三种情况对应的 reason 分别如下：

1) slave 的 RF 通信出现问题（RF 不好或者 master 断电等），slave 连续一段时间收不到 master 的包，连接超时（connection supervision timeout）时触发该事件，连接断开，回到 None Conn state。
2) Master 发送 terminate 主动断开连接，slave 收到 terminate 命令，对这个命令进行 ack 后，触发该事件，连接断开，回到 None Conn state。 terminate reason 是 slave 在 Link Layer 上收到的 LL_TERMINATE_IND 控制包中的 Error Code，该 Error Code 是由 master 决定的。常见的 Error code 有 HCI_ERR_REMOTE_USER_TERM_CONN(0x13)、HCI_ERR_CONN_TERM_MIC_FAILURE（0x3D）等。
3) Slave 端调用了 API bls_ll_terminateConnection(u8 reason)，主动断开连接。terminate reason 是该 API 的实参 reason。

3.2.7.6 BLT_EV_FLAG_ENCRYPTION_CONN_DONE

事件触发条件：Link Layer 加密完成时（Link Layer 收到 Master 发的 start encryption response）触发。

回传指针 p：指向一个 u8 类型的变量 smp_flag，用于指示当前 Link Layer 的加密是在第一次配对还是自动回连时触发。smp_flag 为 0 时，表示第一次配对；smp_flag 为 1 时，表示自动回连。

```
#define SMP_STANDARD_PAIR    0
#define SMP_FAST_CONNECT     1
```

数据长度 n： 1。

Lenze Technology Co.,LTD

3.2.7.7 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE

事件触发条件：Slave 和 Master 交互 Link Layer 最大数据长度时触发，即一方发送 ll_length_req，另一方回复 ll_length_rsp。如果 Slave 主动发送 ll_length_req，需要等收到 ll_length_rsp 时触发；如果 Master 发起 ll_length_req，Slave 回复 ll_length_rsp 后立刻触发。

数据长度 n： 12。

回传指针 p：指向一片内存数据，对应如下结构体的前 6 个 u16 的变量。

```
typedef struct {
    u16     connEffectiveMaxRxOctets;
    u16     connEffectiveMaxTxOctets;
    u16     connMaxRxOctets;
    u16     connMaxTxOctets;
    u16     connRemoteMaxRxOctets;
    u16     connRemoteMaxTxOctets;
    u16     supportedMaxRxOctets;
    u16     supportedMaxTxOctets;

    u16     connInitialMaxTxOctets;
    u8      connMaxTxRxOctets_req;
    }ll_data_extension_t;
```

connEffectiveMaxRxOctets 和 connEffectiveMaxTxOctets 是当前连接上最终允许的 RX 和 TX 最大数据长度；connMaxRxOctets 和 connMaxTxOctets 是设备自己 RX 和 TX 最大数据长度；connRemoteMaxRxOctets 和 connRemoteMaxTxOctets 是对方设备 RX 和 TX 最大数据长度。

connEffectiveMaxRxOctets = min(supportedMaxRxOctets,connRemoteMaxTxOctets);

connEffectiveMaxTxOctets = min(supportedMaxTxOctets, connRemoteMaxRxOctets);

3.2.7.8 BLT_EV_FLAG_GPIO_EARLY_WAKEUP

事件触发条件：slave 进入 suspend 之前，计算好下一次醒来的时间点，到该时间点后醒来（这是由 suspend 状态下的 timer 计时实现的），那么会存在一个问题：如果 suspend 的时间过长，user 的任务必须到 suspend 醒来后才能处理，对于一些实时性要求比较严的应用，可能会出问题，如对于键盘扫描，使用者按键的动作可能很快，为了保证按键不丢同时又要处理按键去抖动，按键扫描的时间间隔在 10~20ms 比较好，此时如果 suspend 时间较大，如 400ms、1s 等（suspend 时间在启用 latency 的时候会达到这些值），按键就会丢掉。那么需要在 MCU 进入 suspend 之前判断当前 suspend 的时间，如果时间过长，设置 suspend 可以被使用者的按键动作提前唤醒（后面 PM 模块详细介绍）。

当 suspend 在 timer wakeup 时间点之前被 GPIO 提前唤醒时，触发 BLT_EV_FLAG_GPIO_EARLY_WAKEUP 事件。

数据长度 n： 1。

回传指针 p：指向一个 u8 型变量 wakeup_status，wakeup_status 变量记录了当前 suspend 哪些唤醒源状态生效了，由 proj_lib/pm.h 可看到如下唤醒状态（SDK 中没有用到 WAKEUP_STATUS_COMP，只关注后面几个即可）。

```
enum {
    WAKEUP_STATUS_COMP   = BIT(0),
    WAKEUP_STATUS_TIMER  = BIT(1),
    WAKEUP_STATUS_CORE   = BIT(2),
```


Lenze Technology Co.,LTD

```
        WAKEUP_STATUS_PAD     = BIT(3),
        STATUS_GPIO_ERR_NO_ENTER_PM  = BIT(7),
    };
```

以上参数具体含义请参考“低功耗管理”部分下面 API 中返回值 int 的详细说明：

int cpu_sleep_wakeup (int deepsleep, int wakeup_src, u32 wakeup_tick);

3.2.7.9 BLT_EV_FLAG_CHN_MAP_REQ

事件触发条件：slave 在 Conn state，master 需要更新当前连接的频点列表，发送一个 LL_CHANNEL_MAP_REQ，slave 收到这个请求后触发该事件，注意是 slave 收到该请求的时候，还并没有处理。

数据长度 n：5。

回传指针 p：p 指向下面频点列表数组的首地址。

unsigned char 类型的 bltc.conn_chn_map[5]

注意回调函数执行时 p 指向的 bltc.conn_chn_map 是还没有更新的老 channel map。

conn_chn_map 用 5 个 byte 表示当前频点列表，采用映射的方法，每个 bit 代表一个 channel：

conn_chn_map[0]的 bit0-bit7 分别表示 channel0-channel7

conn_chn_map[1]的 bit0-bit7 分别表示 channel8-channel15

conn_chn_map[2]的 bit0-bit7 分别表示 channel16-channel23

conn_chn_map[3]的 bit0-bit7 分别表示 channel24-channel31

conn_chn_map[4]的 bit0-bit4 分别表示 channel32-channel36

3.2.7.10 BLT_EV_FLAG_CHN_MAP_UPDATE

事件触发条件：slave 在连接状态，收到 LL_CHANNEL_MAP_REQ 命令后到了更新的时间点，将 channel map 进行更新，触发该事件。

回传指针 p：p 指向 conn_chn_map[5]的首地址，此时的 conn_chn_map 是更新以后新的 map。

数据长度 n：5。

3.2.7.11 BLT_EV_FLAG_CONN_PARA_REQ

事件触发条件：Slave 设备处于连接态（Conn state Slave role），master 设备需要更新当前连接的参数，发送 LL_CONNECTION_UPDATE_REQ 命令，Slave 设备收到这个请求后触发该事件，此时还没有处理这个请求。

数据长度 n：11。

回传指针 p：p 指向 LL_CONNECTION_UPDATE_REQ 的 PDU，如下图所示的 11 个 byte。

CtrData					
WinSize (1 octet)	WinOffset (2 octets)	Interval (2 octets)	Latency (2 octets)	Timeout (2 octets)	Instant (2 octets)

Figure 2.15: CtrData field of the LL_CONNECTION_UPDATE_REQ PDU

图 3- 29 BLE 协议栈 LL_CONNECTION_UPDATE_REQ 格式

Lenze Technology Co.,LTD

3.2.7.12 BLT_EV_FLAG_CONN_PARA_UPDATE

事件触发条件：slave 在连接状态，收到 LL_CONNECTION_UPDATE_REQ 后到了该更新的时间点，对连接参数进行更新，触发该事件。

数据长度 n：6。

回传指针 p：p 指向连接参数更新后的新参数，如下

p[0] | p[1]<<8: new connection interval，以 1.25ms 为 unit

p[2] | p[3]<<8: new connection latecny

p[4] | p[5]<<8: new connection timeout，以 10ms 为 unit

3.2.7.13 BLT_EV_FLAG_SUSPEND_ENETR

事件触发条件：slave 执行 blt_sdk_main_loop 函数时，进入 suspend 之前触发该事件。

回传指针 p：空指针 NULL。

数据长度 n： 0。

3.2.7.14 BLT_EV_FLAG_SUSPEND_EXIT

事件触发条件：slave 执行 blt_sdk_main_loop 函数时， suspend 唤醒之后触发该事件。

回传指针 p：空指针 NULL。

数据长度 n： 0。

注意：实际 SDK 底层执行 cpu_sleep_wakeup 唤醒后执行该回调，且不管是被 gpio 唤醒还是被 timer 唤醒，都会无条件触发该事件。如果同时发生了 BLT_EV_FLAG_GPIO_EARLY_WAKEUP 事件，这两个事件的先后执行顺序请参考“低功耗管理—低功耗管理工作机制”部分伪代码的描述。

3.2.7.15 BLT_EV_FLAG_READ_P256_KEY

待补充。

3.2.7.16 BLT_EV_FLAG_GENERATE_DHKEY

待补充。

3.2.8 Controller API

3.2.8.1 API 返回类型 ble_sts_t

在 proj_lib/ble/ble_common.h 中定义了枚举类型 ble_sts_t，该类型作为 SDK 中大多数 API 的返回值类型，只有调用 API 的设置参数都正确且被协议栈接受时，才会返回 BLE_SUCCESS，值为 0；返回的其他非 0 值都表示设置错误，且每一个不同的值都对应一种错误类型。后面的 API 具体说明中，会列举每一个 API 所有可能的返回值，并解释各个错误返回值的具体错误原因。

这个返回值类型 ble_sts_t 并不只用于 Link Layer 的 API，而是对所有层的 API 都适用。

Lenze Technology Co.,LTD

3.2.8.2 MAC address 初始化

本文档中的 BLE MAC address 默认指的是 public address。

BLE MAC address 的长度为 6 个 byte，本文档前面已经介绍了 MAC address 在 Flash 中的存储地址。最终的产品在烧录时，Lenze 会通过自己的治具系统将 MAC address 烧在指定的 Flash 区域，具体的 MAC address 需要 user 从蓝牙联盟获取。

以 17H25 512K Flash 为例，目前在调试阶段，SDK 对 MAC address 处理的方法为：slave 设备第一次上电时，读到 flash 的 0x76000 处的 MAC address 为空，将 MAC address 设为 0xC7E4E3E2E1xx（前 5 个 byte 固定，最后一个 byte 随机生成），然后将这 6 个 byte 写到 flash 的 0x76000~0x76005。以后每次上电时，读到 flash 的 0x76000 处的 MAC address 已经存在，直接使用该地址，从而确保 slave device 的 MAC address 保持一致。相关代码如下，参考本文档之前对 flash 访问的介绍，加以理解。用户也可以根据自己设计的这个方法进行修改。

```
u8  tbl_mac [] = {0xe1, 0xe1, 0xe2, 0xe3, 0xe4, 0xc7};
u32 *pmac = (u32 *) CFG_ADR_MAC;
if (*pmac != 0xffffffff)
{
    memcpy (tbl_mac, pmac, 6);
}
else{
    tbl_mac[0] = (u8)rand();
    flash_write_page (CFG_ADR_MAC, 6, tbl_mac);
}
```

上面 MAC address 成功获取后，调用 Link Layer 初始化的 API，将 MAC address 传入 BLE 协议栈：

```
blc_ll_initBasicMCU(tbl_mac);   //mandatory
```

如果用到 Link Layer 的状态机中的 Advertising state 或 Scanning state 也需要将 MAC address 传入：

```
blc_ll_initAdvertising_module（tbl_mac）;
blc_ll_initScanning_module（tbl_mac）;
```

3.2.8.3 Link Layer 状态机初始化

结合前面对 Link Layer 状态机的详细介绍，以下几个 API 用于配置搭建 BLE 状态机时各个模块的初始化。

```
void        blc_ll_initBasicMCU (u8 *public_adr)
void        blc_ll_initAdvertising_module(u8 *public_adr);
void        blc_ll_initScanning_module(u8 *public_adr);
void        blc_ll_initInitiating_module(void);
void        blc_ll_initSlaveRole_module(void);
void        blc_ll_initMasterRoleSingleConn_module(void);
```

3.2.8.4 bls_ll_setAdvData

参考《Core_v4.2》Page1282 LE Set Advertising Data Command。

Lenze Technology Co.,LTD

图 3-30 BLE 协议栈广播包格式

BLE 协议栈里，广播包的格式如上图所示，前两个 byte 是 head，后面是 Payload（PDU），最多 31 byte。

下面的 API 用于设置 PDU 部分的数据：

```
ble_sts_t bls_ll_setAdvData(u8 *data, u8 len);
```

data 指针指向 PDU 的首地址，len 为数据长度。返回类型 ble_sts_t 可能返回的结果如下表所示。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	Len 超过最大长度 31

user可以在初始化的时候调用该API设置广播数据，也可以程序运行时在mainloop里随时调用来修改广播数据。

17H25 BLE SDK中17H25 ble remote工程中定义的Adv PDU如下，其中各个字段的含义请参考文档BLE Spec《CSS v6》（Core Specification Supplement v6.0）中Data Type Specifcation的具体说明。

```
u8 tbl_advData[] = {
    0x05, 0x09, 't', 'h', 'i', 'd',
    0x02, 0x01, 0x05,
    0x03, 0x19, 0x80, 0x01,
    0x05, 0x02, 0x12, 0x18, 0x0F, 0x18,
};
```

上面广播数据里，设置广播设备名为"thid"。

3.2.8.5 bls_ll_setScanRspData

参考《Core_v4.2》Page1283 LE Set Scan response Data Command。

类似于上面广播包 PDU 的设置，scan response PDU 的设置使用 API：

```
ble_sts_t bls_ll_setScanRspData(u8 *data, u8 len);
```

data 指针指向 PDU 的首地址，len 为数据长度。返回类型 ble_sts_t 可能返回的结果如下表所示。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	Len 超过最大长度 31

user 可以在初始化的时候调用该 API 设置 Scan response data，也可以程序运行时在 mainloop 里随时调用来修改 Scan response data。

Lenze Technology Co., LTD

17H25 BLE SDK 中 17H25 ble remote 工程中定义 scan response data 如下，scan 设备名为 "tRemote" 。各个字段的含义请参考文档 BLE Spec《CSS v6》(Core Specification Supplement v6.0) 中 Data Type Specifcation 的具体说明。

```
u8  tbl_scanRsp [] = {
     0x08, 0x09, 't', 'R', 'e', 'm', 'o', 't', 'e',
};
```

上面在 advertising data 和 scan response data 中都设置了设备名称且不一样，那么在手机或 IOS 系统上扫描蓝牙设备时，看到的设备名称可能会不一样：

1) 一些设备只看广播包，那么显示的设备名称为"thid"
2) 一些设备看到广播后，发送 scan request，并读取回包 scan response，那么显示的设备名称可能就会是"tRemote"

user 也可以在这两个包中将设备名称写为一样，被扫描时就不会显示两个不同的名字了。

实际上设备被 master 连接后，master 在读设备的 Attribute Table 时，会获取设备的 gap device name，连上设备后也可能会根据那里的设置来显示设备名称。

3.2.8.6 bls_ll_setAdvParam

参考《Core_v4.2》Page1277 LE Set Advertising Parameters Command。

图 3-31 BLE 协议栈里 Advertising Event

BLE 协议栈里 Advertising Event(简称 Adv Event)如上图所示，指的是在每一个 T_advEvent，slave 进行一轮广播，在三个广播 channel（channel 37、channel 38、channel 39）上各发一个包。下面 API 对 Adv Event 相关的参数进行设置。

```
ble_sts_t bls_ll_setAdvParam( u16 intervalMin, u16 intervalMax,
                              u8  advType,   u8 ownAddrType,
                              u8 peerAddrType, u8 *peerAddr,
                              u8 adv_channelMap, u8 advFilterPolicy);
```

1) intervalMin 和 intervalMax：

设置广播时间间隔 adv interval 的范围，以 0.625ms 为基本单位，范围在 20ms ~10.24S 之间，并且 intervalMin 小于等于 intervalMax。

BLE spec 要求 adv interval 不要设一个固定的值，需要有一些随机的变化。Lenze BLE SDK 通过不同值的 intervalMin 和 intervalMax 来实现，最终的 adv interval 是在 intervalMin~intervalMax 之间随机变化。若设置的 intervalMin 和 intervalMax 相等，adv interval 会等于固定的 intervalMin，不会变化。

根据不同广播包的类型，intervalMin 和 intervalMax 的值有一些限定，参考《Core_v4.2》

Page2609 4.4.2.2 Advertising Interval。

2) advType

《Core_v4.2》Page2607 Advertising State 部分介绍了四种基本的广播事件类型，如下:

Advertising Event Type	PDU used in this advertising event type	Allowable response PDUs for advertising event	
		SCAN_REQ	CONNECT_REQ
Connectable Undirected Event	ADV_IND	YES	YES
Connectable Directed Event	ADV_DIRECT_IND	NO	YES*
Non-connectable Undirected Event	ADV_NONCONN_IND	NO	NO
Scannable Undirected Event	ADV_SCAN_IND	YES	NO

Table 4.1: Advertising event types, PDUs used and allowable response PDUs

图 3-32 BLE 协议栈四种广播事件

上图中 Allowable response PDUs for advertising event 部分用 YES 和 NO 说明了各种类型广播事件是否对其他设备的 Scan request 和 Connect Request 进行响应，如：第一个 Connectable Undirected Event 对 Scan request 和 Connect Request 都能响应，而 Non-connectable Undireted Event 对它们都不响应。

注意第二个 Connectable Directed Event 对 Connect Request 响应那个"YES"右上角加了“*”号，表示它只要收到匹配的 Connect Request，就一定会响应，而不会被 whitelist 过滤。剩下的 3 个"YES"表示可以响应相应的请求，但实际需要依赖于 whitelist 的设置，根据 whitelist 的过滤条件来决定最终是否响应，后面的 whitelist 中会详细介绍。

以上四种广播事件中，Connectable Directed Event 又分为 Low Duty Cycle Directed Advertising 和 High Duty Cycle Directed Advertising，这样一共能够得到五种广播事件类型，如下定义（proj_lib/ble/ble_common.h），参考《Core_v4.2》Page2609 ~ Page2615。

```
/* Advertisement Type */
typedef enum{
    ADV_TYPE_CONNECTABLE_UNDIRECTED          = 0x00,  // ADV_IND
    ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY  = 0x01,
                                  //ADV_INDIRECT_IND (high duty cycle)
    ADV_TYPE_SCANNABLE_UNDIRECTED          = 0x02 //ADV_SCAN_IND
    ADV_TYPE_NONCONNECTABLE_UNDIRECTED     = 0x03, //ADV_NONCONN_IND
   ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY            =            0x04,
                                  //ADV_INDIRECT_IND (low duty cycle)
}advertising_type;
```

默认最常用的广播类型为 *ADV_TYPE_CONNECTABLE_UNDIRECTED*。

3) ownAddrType

指定广播包中 MAC address 类型。

基本的地址类型有以下两种：public 和 random。

```
/* Device Address Type */
#define BLE_ADDR_PUBLIC                 0
#define BLE_ADDR_RANDOM                 1
```

指定广播地址类型时，ownAddrType 4 个可选的值如下，具体地址如何生成参考《Core_v4.2》Page1279 的详细说明。

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

这里只介绍最基本的 OWN_ADDRESS_PUBLIC，实际最终的地址来自 MAC address 初始化时 API blc_ll_initAdvertising_module(u8 *public_adr)的设置。

Lenze Technology Co.,LTD

4) peerAddrType 和*peerAddr

当 advType 被设置为直接广播包类型 directed adv（*ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY* 和 *ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY*）时，peerAddrType 和*peerAddr 用于指定 peer device MAC Address 的类型和地址。

当 advType 为其他类型时，peerAddrType 和*peerAddr 的值都无效，可以设定为 0 和 NULL。

5) adv_channelMap

设定广播 channel，可以选择 channel 37、38、39 中任意一个或多个。adv_channelMap 的值可设置如下 3 个或它们中任意或组合。

```
#define        BLT_ENABLE_ADV_37        BIT(0)
#define        BLT_ENABLE_ADV_38        BIT(1)
#define        BLT_ENABLE_ADV_39        BIT(2)
#define        BLT_ENABLE_ADV_ALL
(BLT_ENABLE_ADV_37 | BLT_ENABLE_ADV_38 | BLT_ENABLE_ADV_39)
```

6) advFilterPolicy

用于设定发送广播包时，对其他设备的 scan request 和 connect request 采取的过滤策略。过滤的地址需要提前存储到 whitelist 中。在后面 whitelist 介绍中详细解释。

可设置的 4 种过滤类型如下，若不需要 whitelist 过滤功能，选择 ADV_FP_NONE。

```
#define    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY      0x00
#define    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_ANY       0x01
#define    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_WL       0x02
#define    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL        0x03
#define    ADV_FP_NONE  ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY
```

返回值 ble_sts_t 可能出现的值和原因如下表所示：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	intervalMin 或 intervalMax 的值不符合 BLE spec 的规定

按照 BLE spec 的设计，Set Advertising parameters 同时设置了上面的 8 个参数。同时设置的思路也是合理的，因为一些不同的参数之间是有耦合关系的，比如 advType 和 advInterval，在不同的 advType 下，对 intervalMin 和 intervalMax 的范围限定会不一样，所以会有不同的范围检查，如果将 set advType 和 set advInterval 拆成两个不同的 API，彼此间的范围检查就无法控制。

但是考虑到 user 对一些常用的参数可能会经常性的去修改，而又不希望每次都要调用 bls_ll_setAdvParam 同时设置 8 个参数，SDK 对其中 4 个不会跟其他参数有耦合关系的参数，单独封装了 API，以方便 user 的使用。单独封装 3 个 API 如下：

```
ble_sts_t  bls_ll_setAdvInterval(u16 intervalMin, u16 intervalMax);
ble_sts_t  bls_ll_setAdvChannelMap(u8 adv_channelMap);
ble_sts_t  bls_ll_setAdvFilterPolicy(u8 advFilterPolicy);
```

这 3 个 API 的参数与 bls_ll_setAdvParam 中一致。

Lenze Technology Co.,LTD

返回值 ble_sts_t:

1) bls_ll_setAdvChannelMap 和 bls_ll_setAdvFilterPolicy 会无条件返回 BLE_SUCCESS。
2) bls_ll_setAdvInterval 返回 BLE_SUCCESS 或 HCI_ERR_INVALID_HCI_CMD_PARAMS。

3.2.8.7 bls_ll_setAdvEnable

参考《Core_v4.2》Page1284 LE Set Advertising Enable Command。

```
ble_sts_t  bls_ll_setAdvEnable(u8 en);
```

en 为 1 时，Enable Advertising；en 为 0 时，Disable Advertising。

1) 在 Idle state 时，Enable Advertising，Link Layer 进入 Advertising state。
2) 在 Advertising state 时，Disable Advertising，Link layer 进入 Idle state。
3) 在其他 state，Enable Advertising 或 Disable Advertising 都不影响 Link Layer 的 state。
4) ble_sts_t 无条件返回 BLE_SUCCESS。

3.2.8.8 bls_ll_setAdvDuration

```
ble_sts_t  bls_ll_setAdvDuration (u32 duration_us, u8 duration_en);
```

使用 bls_ll_setAdvParam 对广播所有参数设置成功后，使用 bls_ll_setAdvEnable（1）开始广播。若希望对设置好的广播事件进行时间限定，让它持续发送一段时间后就自动关闭，可以调用上面的 API。

duration_en 设为 1 表示开启计时功能，设为 0 表示关闭计时功能。只有在计时功能开启的情况下，duration_us（单位:us）的设置才有意义，程序从设置的时间点开始计时，一旦超出预设的时间时，停止广播，广播使能（AdvEnable）失效，在 None Conn state 下，会切换到 Idle State。此时会触发 Link Layer 事件 BLT_EV_FLAG_ADV_DURATION_TIMEOUT。

BLE Spec 中规定 ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY 广播类型，Duration Time 固定为 1.28s，到 1.28s 之后将停止广播。所以对于这种广播类型，调用 bls_ll_setAdvDuration 设置将无效。返回值 ble_sts_t 见下表。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY 广播类型不能被设置 Duration Time

如果 user 需要在 Adv Duratrion Time 到达广播停止后，重新设置广播参数（AdvType、AdvInterval、AdvChannelMap 等），需要在 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 事件的回调函数里设置，并且需要再次调用 bls_ll_setAdvEnable（1）开始新的广播。

触发 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 需要注意一种特殊的情况：假设设定了 duration_us 为 2000000，即 2s。如果 Slave 一直在广播，那么广播时间到达 2s 时会触发 timeout，执行 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调。如果广播时间不到 2s 的情况下（假设在 0.5s 的时候），Slave 和 Master 连接上了，这个 timeout 的计时在底层并没有被清除掉，而是被缓存起来。在进入连接状态 1.5s 的时候，也就是到达实际设定的 timeout 时间点正好 2s 时，由于此时已经处于连接状态，不会去检查广播事件是否超时，也就不会触发

Lenze Technology Co., LTD

BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调。当进入连接状态一段时间（如 10s）后断开连接重新回到广播（adv）状态，在发第一个广播包前，协议栈认为此时的时间超过了之前设定的 2s timeout，触发 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调。在这种情况下触发的 BLT_EV_FLAG_ADV_DURATION_TIMEOUT 回调已经远远超过了实际设定的 timeout 时间点，所以需要特别注意。

3.2.8.9 blc_ll_setAdvCustomedChannel

下面 API 用于定制特殊的 advertising channel/scanning channel，只对一些非常特殊的应用有意义，如 BLE mesh，其他常规 BLE 应用不要使用该 API。

```
void blc_ll_setAdvCustomedChannel (u8 chn0, u8 chn1, u8 chn2);
```

chn0/chn1/chn2 填需要定制的频点，默认的标准频点是 37/38/39，比如设置 3 个 advertising channel 分别为 2420MHz、2430MHz、2450MHz，可如下调用：

blc_ll_setAdvCustomedChannel (8, 12, 22);

3.2.8.10 rf_set_power_level_index

17H25 BLE SDK 提供了 BLE RF packet 能量设定的 API:

void rf_set_power_level_index (int level);

level 值的选取在 proj_lib/rf_drv.h 中给出了枚举变量，从名字可以看出设定的能量值大小，

```
enum {
    RF_POWER_8dBm = 0,
    RF_POWER_4dBm = 1,
    RF_POWER_0dBm = 2,
    RF_POWER_m4dBm   = 3,
    RF_POWER_m10dBm  = 4,
    RF_POWER_m14dBm  = 5,
    RF_POWER_m20dBm  = 6,
    RF_POWER_m24dBm  = 8,
    RF_POWER_m28dBm  = 9,
    RF_POWER_m30dBm  = 10,
    RF_POWER_m37dBm  = 11,
    RF_POWER_OFF  = 16,
};
```

如设定当前 RF 发包能量为最大的 8dbm：

```
rf_set_power_level_index(RF_POWER_8dBm);
```

该 API 设定的 RF 发包能量，对广播包和连接包同时有效，且在程序的任意位置都可以设置，实际发包时的能量以时间上最近一次的设置为准。

Lenze Technology Co.,LTD

3.2.8.11 blc_ll_setScanParameter

参考《Core_v4.2》Page1286 LE Set Scan Parameters Command。

```
ble_sts_t  blc_ll_setScanParameter (u8 scan_type,
                                    u16 scan_interval, u16 scan_window,
                                u8  ownAddrType, u8 filter_policy);
```

参数解析：

1) scan_type

可选择 passive scan 和 active scan，区别是 active scan 会在收到 adv packet 基础上发 scan_req 以获取设备 scan_rsp 的更多信息，scan rsp 包也会通过 adv report event 传给 BLE Host；passive scan 不发 scan req。

```
typedef enum {
    SCAN_TYPE_PASSIVE = 0x00,
    SCAN_TYPE_ACTIVE,
  } scan_type_t;
```

2) scan_inetrval/scan window

scan_interval 设置 Scanning state 时频点切换时间，单位为 0.625ms，scan_window 在 Lenze BLE SDK 中暂时没有处理，实际的 scan window 设置为 scan_interval。

3) ownAddrType

指定广播包中 MAC address 类型。

```
/* Device Address Type */
#define BLE_ADDR_PUBLIC                 0
#define BLE_ADDR_RANDOM                 1
```

指定 scan req 包地址类型时，ownAddrType 4 个可选的值如下，具体地址如何生成参考《Core_v4.2》Page1287 的详细说明。

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

这里只介绍最基本的 OWN_ADDRESS_PUBLIC，实际最终的地址来自 MAC address 初始化时 API blc_ll_initScanning_module(u8 *public_adr)的设置。

4) filter_policy

目前支持的 scan filter policy 为下面两个：

```
#define    SCAN_FP_ALLOW_ADV_ANY              0x00
#define    SCAN_FP_ALLOW_ADV_WL               0x01
```

SCAN_FP_ALLOW_ADV_ANY 表示 Link Layer 对 scan 到的 adv packet 不做过滤，直接 report 到 BLE Host，SCAN_FP_ALLOW_ADV_WL 则要求 scan 到的 adv packet 必须是在 whitelist 里面的，

Lenze Technology Co.,LTD

才 report 到 BLE Host。

返回值 ble_sts_t 只有 BLE_SUCCESS，API 不会进行参数合理性检查，user 需要注意设置参数的合理性。

3.2.8.12 blc_ll_setScanEnable

参考《Core_v4.2》Page1289 LE Set Scan Enable Command。

```
ble_sts_t blc_ll_setScanEnable (u8 scan_enable, u8 filter_duplicate);
```

scan_enable 为 1 时，Enable Scanning；scan_enable 为 0 时，Disable Scanning。

1) 在 Idle state 时，Enable Scanning，Link Layer 进入 Scanning state。
2) 在 Scanning state 时，Disable Scanning，Link layer 进入 Idle state。

filter_duplicate 为 1 时，表示开启重复包过滤，此时对每个不同的 adv packet，Controller 只向 Host 上报一次 adv report event；filter_duplicate 为 0 时，不开启重复包过滤，对 scan 到的 adv packet 会一直上报给 Host。

返回值 ble_sts_t 见下表。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_CONTROLLER_BUSY	0x3A	Link Layer 处于 BLS_LINK_STATE_ADV /BLS_LINK_STATE_CONN 状态

当设置了scan_type为active scan，Enable Scanning后，对每个device，只读一次scan_rsp并上报给Host。因为每次Enable Scanning后，Controller会对不同设备的scan_rsp进行记录，将它们存储到scan_rsp列表里，确保后面不会再次去scan_req该设备。

若user需要多次上报同一个device的scan_rsp，可以通过blc_ll_setScanEnable重复设置Enable Scanning实现，因为每次Enable/Disable Scanning时，设备的scan_rsp列表都会清0。

3.2.8.13 blc_ll_createConnection

参考《Core_v4.2》Page1291 LE Create Connection Command。

```
ble_sts_t blc_ll_createConnection (u16 scan_interval, u16 scan_window, u8
                                   initiator_filter_policy,
                                   u8 adr_type, u8 *mac,
                                   u8 own_adr_type,
          u16 conn_min, u16 conn_max, u16 conn_latency, u16 timeout,
                        u16 ce_min, u16 ce_max );
```

1) scan_inetrval/scan window

Lenze Technology Co.,LTD

scan_interval 设置 Initiating state 中 Scan 频点切换时间，单位为 0.625ms，scan_window 在 Lenze BLE SDK 中暂时没有处理，实际的 scan window 设置为 scan_interval。

2) initiator_filter_policy

指定当前连接设备的策略，可选如下两种：

```
#define    INITIATE_FP_ADV_SPECIFY   0x00  //adv specified by host
#define    INITIATE_FP_ADV_WL        0x01  //adv in whitelist
```

INITIATE_FP_ADV_SPECIFY 表示连接的设备地址是后面的 adr_type/mac；INITIATE_FP_ADV_WL 表示根据 whitelist 里面的设备来连接，此时 adr_type/mac 无意义。

3) adr_type/ mac

initiator_filter_policy 为 INITIATE_FP_ADV_SPECIFY 时，连接地址类型为 adr_type（BLE_ADDR_PUBLIC 或者 BLE_ADDR_RANDOM）、地址为 mac[5…0]的设备。

4) own_adr_type

指定建立连接的 Master role 使用的 MAC address 类型。ownAddrType 4 个可选的值如下。

```
typedef enum{
    OWN_ADDRESS_PUBLIC = 0,
    OWN_ADDRESS_RANDOM = 1,
    OWN_ADDRESS_RESOLVE_PRIVATE_PUBLIC = 2,
    OWN_ADDRESS_RESOLVE_PRIVATE_RANDOM = 3,
}own_addr_type_t;
```

这里只介绍最基本的 OWN_ADDRESS_PUBLIC，实际最终的地址来自 MAC address 初始化时 API blc_ll_initBasicMCU(u8 *public_adr)的设置。

5) conn_min/ conn_max/ conn_latency/ timeout

这 4 个参数规定了建立连接后 Master role 的连接参数，同时这些参数也会通过 connection request 发给 Slave，Slave 也会是同样的连接参数。

conn_min/conn_max 指定 conn interval 的范围，Lenze BLE SDK 中 Master role Single Connection 直接使用 conn_min 的值。单位为 0.625ms。

conn_latency 指定 connection latency，一般设为 0。

timeout 指定 connection supervision timeout，单位为 10ms。

6) ce_min/ ce_max

SDK 暂未处理 ce_min/ ce_max。

3.2.8.14 blc_ll_setCreateConnectionTimeout

```
ble_sts_t  blc_ll_setCreateConnectionTimeout (u32 timeout_ms);
```

返回值为BLE_SUCCESS，timeout_ms单位为ms。

Link Layer状态机中介绍，当blc_ll_createConnection触发Idle state/Scanning state进入Initiating

Lenze Technology Co., LTD

state后，如果长时间无法建立连接，会触发Initiate timeout，退出Initiating state。

每次调用blc_ll_createConnection时，SDK默认设置当前的Initiate timeout时间为connection supervision timeout *2。如果User不希望使用SDK默认的这个时间，可以在紧接着blc_ll_createConnection之后调用blc_ll_setCreateConnectionTimeout设置自己需要的Initiate timeout。

3.2.8.15 blm_ll_updateConnection

参考《Core_v4.2》Page1302 LE Connection Update Command。

```
ble_sts_t blm_ll_updateConnection (u16 connHandle,
          u16 conn_min, u16 conn_max, u16 conn_latency, u16 timeout,
                        u16 ce_min, u16 ce_max );
```

1) connection handle
   指定需要更新连接参数的connection。

2) conn_min/ conn_max/ conn_latency/ timeout
   指定需要更新的连接参数，Master role single connection 目前直接使用 conn_min 作为新的interval。

3) ce_min/ce_max
   目前不处理。

返回值 ble_sts_t 只有 BLE_SUCCESS，API 不会进行参数合理性检查，user 需要注意设置参数的合理性。

Lenze Technology Co.,LTD

3.2.8.16 bls_ll_terminateConnection

参考《Core_v4.2》Page2593 LL_TERMINATE_IND。

```
ble_sts_t  bls_ll_terminateConnection (u8 reason);
```

该 API 使用于 BLE Slave 设备，只对 Connection state Slave role 适用。

应用层可以调用此 API 在 Link Layer 上发送一个 Terminate 给 master，主动断开连接，reason 为需要指定的断开原因，reason 列表参考《Core_v4.2》Page680 Error Code Descriptions，对应 ble_common.h 中 ble_sts_t 的定义。

若不是系统运行异常导致的 terminate，应用层一般指定 reason 为

HCI_ERR_REMOTE_USER_TERM_CONN = 0x13

```
bls_ll_terminateConnection(HCI_ERR_REMOTE_USER_TERM_CONN);
```

Lenze BLE SDK 底层 stack 中，只有一个地方会调用该 API 主动 termiante：解密对方设备的数据包，如发现认证数据 MIC 错误，调用 bls_ll_terminateConnection(HCI_ERR_CONN_TERM_MIC_FAILURE)，告知对方解密错误，断开连接。

Slave 调用该 API 主动发起断开连接后，一定会触发 BLT_EV_FLAG_TERMINATE 事件，在该事件的回调函数里可以看到对应的 terminate reason 和这个手动设置的 reason 是一样的。

在 Connection state Slave role 时，一般情况下直接调用该 API 可以成功发送 terminate 并断连，但也存在一些特殊情况导致该 API 调用失败，根据返回值 `ble_sts_t` 可以了解对应的错误原因。建议应用层调用该 API 时，检查一下返回值是否为 BLE_SUCCESS。返回值列表如下。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer 处于非 Connection state Slave role
HCI_ERR_CONTROLLER_BUSY	0x3A	Controller busy（有大量数据正在发送）暂时无法接受该命令。

Lenze Technology Co., LTD

3.2.8.17 blm_ll_disconnect

参考《Core_v4.2》Page2593 LL_TERMINATE_IND。

```
ble_sts_t  blm_ll_disconnect (u16 handle, u8 reason);
```

该 API 使用于 BLE Master 设备，只对 Connection Master role 适用。跟 Slave role 的 API bls_ll_terminateConnection 功能一样，区别是多一个 conn handle 参数，这是因为 Lenze BLE SDK 对 Slave role 的设计上最多只维持 single connection。

API 返回值如下表。

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_UNKNOWN_CONN_ID	0x02	Handle 错误，找不到对应的 connection
HCI_ERR_CONTROLLER_BUSY	0x3A	Controller busy（有大量数据正在发送）暂时无法接受该命令。

3.2.8.18 Get Connection Parameters

获取当前连接参数的 Connection Interval、Connection Latency、Connection Timeout 的 API 如下（只对 Slave role 适用）：

```
u16        bls_ll_getConnectionInterval(void);
u16        bls_ll_getConnectionLatency(void);
u16        bls_ll_getConnectionTimeout(void);
```

1) 若返回值为 0，表示当前 Link Layer 处于 None Conn state，也就没有连接参数。
2) 若返回为非 0 值表示参数值：
   bls_ll_getConnectionInterval 返回的是实际 conn interval 除以 1.25ms 单位值，比如当前 conn interval 为 10ms，则返回值为 8。
   bls_ll_getConnectionLatency 返回实际 Latency 值。
   bls_ll_getConnectionTimeout 返回的是实际 conn timeout 除以 10ms 的单位值，比如当前 conn timeout 为 1000ms，则返回值为 100。

3.2.8.19 blc_ll_getCurrentState

下面 API 用于获取当前 Link Layer 所处的状态。

```
u8  blc_ll_getCurrentState(void);
```

user 在应用层判断当前状态，如：

```
if(  bls_ll_getCurrentState() == BLS_LINK_STATE_ADV)
if(  bls_ll_getCurrentState() == BLS_LINK_STATE_CONN )
```

3.2.8.20 blc_ll_getLatestAvgRSSI

当 Link Layer 进入 Slave role 后，通过下面 API 能得到跟自己连接的 peer device 过去一段时间的平均 RSSI。

```
u8          blc_ll_getLatestAvgRSSI(void)
```


Lenze Technology Co.,LTD

返回值u8的rssi_raw要做一定的转换，rssi_real = rssi_raw- 110，假设返回值为50，则rssi = -60 db。

3.2.8.21 Whitelist & Resolvinglist

前面介绍过，Advertising/Scanning/Initiating state 的 filter_policy 中都涉及到 Whitelist，会根据 Whitelist 中的设备进行相应的操作。实际 Whitelist 概念中包含 Whitelist 和 Resolvinglist 两部分。

通过 peer_addr_type 和 peer_addr 可以判断 peer device 地址类型是否 RPA（resolvable private address）。使用下面的宏判断即可。

```
#define IS_NON_RESOLVABLE_PRIVATE_ADDR(type, addr)
            ( (type)==BLE_ADDR_RANDOM && (addr[5] & 0xC0) == 0x00 )
```

非 RPA 地址才可以存储到 whitelist 中，目前 SDK whitelist 最多存储 4 个设备：

```
#define    MAX_WHITE_LIST_SIZE                4
```

相关接口：

```
ble_sts_t ll_whiteList_reset(void);
```

Reset whitelist，返回值为 BLE_SUCCESS。

```
ble_sts_t ll_whiteList_add(u8 type, u8 *addr);
```

添加一个设备到 whitelist，返回值列表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	添加成功
HCI_ERR_MEM_CAP_EXCEEDED	0x07	whitelist 已满，添加失败

```
ble_sts_t ll_whiteList_delete(u8 type, u8 *addr);
```

从 whitelist 删除之前添加的设备，返回值为 BLE_SUCCESS。

RPA（resolvable private address）设备，需要使用 Resolvinglist。为了节省 ram 使用，目前 SDK Resolvinglist 最多存储 2 个设备：

```
#define    MAX_WHITE_IRK_LIST_SIZE        2
```

相关 API 如下：

```
ble_sts_t  ll_resolvingList_reset(void);
```

Reset Resolvinglist。返回值 BLE_SUCCESS。

```
ble_sts_t  ll_resolvingList_setAddrResolutionEnable (u8 resolutionEn);
```

设备地址解析使用，如果要使用 Resolvinglist 解析地址，一定要打开使能。不需要解析的时候，可以关闭。

```
ble_sts_t  ll_resolvingList_add(u8 peerIdAddrType, u8 *peerIdAddr,
```


Lenze Technology Co.,LTD

```
u8 *peer_irk, u8 *local_irk);
```

添加使用 RPA 地址的设备，peerIdAddrType/ peerIdAddr 和 peer-irk 填 peer device 宣称的 identity address 和 irk，这些信息会在配对加密过程中存储到 Flash 中，user 可以在文档 SMP 部分找到获取这些信息的接口。对于 local_irk SDk 暂时没有处理，填 NULL 即可。

返回值列表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x0D	Link Layer 已经处于 Initiating state，不再接收新的 create connection
HCI_ERR_CONTROLLER_BUSY	0x3A	Link Layer 处于 Advertising state 或 Connection state

```
ble_sts_t  ll_resolvingList_delete(u8 peerIdAddrType, u8 *peerIdAddr);
```

删除之前添加的设备。

Whitelist/Resolvinglist 实现地址过滤的使用，请参考 17H25 feature test 中的 TEST_WHITELIST：

```
smp_param_save_t  bondInfo;
u8 bond_number = blc_smp_param_getCurrentBondingDeviceNumber();  //get bonded device number
if(bond_number)   //get latest device info
{
    blc_smp_param_loadByIndex( bond_number - 1, &bondInfo);  //get the latest bonding device (index: bond_number-1 )
}


ll_whiteList_reset();     //clear whitelist
ll_resolvingList_reset(); //clear resolving list


if(bond_number)  //use whitelist to filter master device
{
    app_whilteList_enable = 1;

    //if master device use RPA(resolvable private address), must add irk to resolving list
    if( IS_RESOLVABLE_PRIVATE_ADDR(bondInfo.peer_addr_type, bondInfo.peer_addr) ){
        //resolvable private address, should add peer irk to resolving list
        ll_resolvingList_add(bondInfo.peer_id_adrType, bondInfo.peer_id_addr, bondInfo.peer_irk, NULL);  //no local IRK
        ll_resolvingList_setAddrResolutionEnable(1);
    }
    else{
        //if not resolvable random address, add peer address to whitelist
        ll_whiteList_add(bondInfo.peer_addr_type, bondInfo.peer_addr);
    }


    bls_ll_setAdvParam( ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, \
                        ADV_TYPE_CONNECTABLE_UNDIRECTED, OWN_ADDRESS_PUBLIC, \
                        0,  NULL, BLT_ENABLE_ADV_37, ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL);

}
else{

    bls_ll_setAdvParam( ADV_INTERVAL_30MS, ADV_INTERVAL_30MS,
                        ADV_TYPE_CONNECTABLE_UNDIRECTED, OWN_ADDRESS_PUBLIC,
                        0,  NULL, BLT_ENABLE_ADV_37, ADV_FP_NONE);
}
```

图 3-33 Whitelist/Resolvinglist 地址过滤

3.3 L2CAP

L2CAP 根据 BLE Spec 设计，主要功能是完成 Controller 和 Host 的数据对接，绝大部分都

在协议栈底层完成，需要 user 参与的地方很少。user 根据以下几个 API 进行设置即可。

3.3.1 注册 L2CAP 数据处理函数

在 BLE SDK 架构中，Controller 的数据通过 HCI 与 Host 对接，从 HCI 到 Host 数据，首先会在 L2CAP 层处理，使用下面 API 注册该处理函数：

```
void        blc_l2cap_register_handler (void *p);
```

在 17H25 remote/17H25 module 等 BLE Slave 应用中，SDK L2CAP 层处理 Controller 数据的函数为：

```
int         blc_l2cap_packet_receive (u16 connHandle, u8 * p);
```

该函数已经在协议栈中实现，它会对接收到的数据进行解析后向上传输给 ATT 或 SMP。

初始化：

```
blc_l2cap_register_handler (blc_l2cap_packet_receive);
```

3.3.2 更新连接参数

3.3.2.1 slave 请求更新连接参数

在 BLE 协议栈中，slave 通过 l2cap 层的 CONNECTION PARAMETER UPDATE REQUEST 命令向 master 申请一组新的连接参数，该命令格式如下所示，详细说明参考《Core_v4.2》Page 1775 CONNECTION PARAMETER UPDATE REQUEST。

Figure 4.22: Connection Parameters Update Request Packet

图 3-34 BLE 协议栈中 Connection Para update Req 格式

17H25 BLE SDK 在 L2CAP 层上提供了 slave 主动申请更新连接参数的 API，用来向 master 发送上面这个 CONNECTION PARAMETER UPDATE REQUEST 命令。

```
void  bls_l2cap_requestConnParamUpdate (u16 min_interval,
                                        u16 max_interval,
                                        u16 latency, u16 timeout);
```

以上四个参数跟 CONNECTION PARAMETER UPDATE REQUEST 的 data 区域中四个参数对应。注意 min_interval 和 max_interval 的值是实际 interval 时间值除以 1.25 ms（如申请 7.5ms

的连接，该值为 6），timeout 的值为实际 supervision timeout 时间值除以 10ms（如 1 s 的 timeout 该值为 100）。

应用举例：在 connection 建立的时候，申请更新连接参数。

```
void task_connect (u8 e, u8 *p)
{
    bls_l2cap_requestConnParamUpdate (6, 6, 99, 400);  //interval=7.5ms latency=99 timeout=4s
}
```

图 3-35 抓包显示 conn para update reqeust 和 response

3.3.2.2 master 回应更新申请

slave 申请新的连接参数后，master 收到该命令，回 CONNECTION PARAMETER UPDATE RESPONSE 命令，参考《Core_v4.2》Page 1776 CONNECTION PARAMETER UPDATE RESPONSE。

下图所示为该命令格式及 result 含义，result 为 0x0000 时表示接受该命令，result 为 0x0001 时表示拒绝该命令。实际的 Android、iOS 设备是否接受 user 所申请的连接参数，跟各个厂家 BLE master 的做法有关，基本上每一家都是不同的，这里没法提供一个统一的标准，只能靠 user 在平时的 master 兼容性测试中去慢慢总结归纳。

图 3-35 所示的抓包显示 master 接受了申请。

Figure 4.23: Connection Parameters Update Response Packet

The data field is:

- *Result (2 octets)*

  The result field indicates the response to the Connection Parameter Update Request. The result value of 0x0000 indicates that the LE master Host has accepted the connection parameters while 0x0001 indicates that the LE master Host has rejected the connection parameters.

Result	Description
0x0000	Connection Parameters accepted
0x0001	Connection Parameters rejected
Other	Reserved

图 3-36 BLE 协议栈中 conn para update rsp 格式

Lenze 的 17H25 master kma dongle 中处理 slave 的连接参数更新 demo code 如下：

```
else if(ptrL2cap->chanId == L2CAP_CID_SIG_CHANNEL)  //signal
{
    if(ptrL2cap->opcode == L2CAP_CMD_CONN_UPD_PARA_REQ)  //slave send conn param update req on l2cap
    {
        rf_packet_l2cap_connParaUpReq_t  * req = (rf_packet_l2cap_connParaUpReq_t *)ptrL2cap;

        u32 interval_us = req->min_interval*1250;  //1.25ms unit
        u32 timeout_us = req->timeout*10000; //10ms unit
        u32 long_suspend_us = interval_us * (req->latency+1);

        //interval < 200ms
        //long suspend < 11S
        // interval * (latency +1)*2 <= timeout
        if( interval_us < 200000 && long_suspend_us < 20000000 && (long_suspend_us*2<=timeout_us) )
        {
            //when master host accept slave's conn param update req, should send a conn param update re
            //with CONN_PARAM_UPDATE_ACCEPT; if not accpet,should send  CONN_PARAM_UPDATE_REJECT
            blc_l2cap_SendConnParamUpdateResponse(current_connHandle, CONN_PARAM_UPDATE_ACCEPT);  //se


            //if accept, master host should mark this, add will send  update conn param req on link la
            //set a flag here, then send update conn param req in mainloop
            host_update_conn_param_req = clock_time() | 1 ; //in case zero value
            host_update_conn_min = req->min_interval;  //backup update param
            host_update_conn_latency = req->latency;
            host_update_conn_timeout = req->timeout;
        }
        else
        {
            blc_l2cap_SendConnParamUpdateResponse(current_connHandle, CONN_PARAM_UPDATE_REJECT);  //se
        }
    }
}
```

在 L2CAP_CID_SIG_CHANNEL 上收到 L2CAP_CMD_CONN_UPD_PARA_REQ 后，首先读取 interval_min（将其作为最终 ineterval）、supervision timeout 和 long suspend time(interval * (latency +1))，对这些数据做一些合理性的判断：要求 interval < 200ms；long suspend time<20S; supervision timeout >= 2* long suspend time，符合这些条件就接收该参数申请，不符合则拒绝。这是个简单的 demo 设计，user 可以根据需要进行相关的修改。

不管是否接收 Slave 的参数申请，都使用下面 API 对该申请进行答复：

```
void  blc_l2cap_SendConnParamUpdateResponse( u16 connHandle,
int result);
```

connHandle 指定当前 connection ID，result 如下两个选择表示接收和拒绝。

```
typedef enum{
        CONN_PARAM_UPDATE_ACCEPT = 0x0000,
        CONN_PARAM_UPDATE_REJECT = 0x0001,
}conn_para_up_rsp;
```

如果 17H25 master 接收了 Slave 的申请，必须在一定时间内通过 API blm_ll_updateConnection 向 Controller 发一个更新的 cmd，demo code 上使用 host_update_conn_param_req 作为标记，并在 mainloop 中延时 50ms 后发起这个 update：

```
//proc master update
//at least 50ms later and make sure smp/sdp is finished
if( host_update_conn_param_req && clock_time_exceed(host_update_conn_param_req, 50000) && !app_host_smp_sdp_pe
{
    host_update_conn_param_req = 0;

    if(blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){  //still in connection state
        blm_ll_updateConnection (current_connHandle,
                host_update_conn_min, host_update_conn_min, host_update_conn_latency,  host_update_conn_timeou
                                    0, 0 );
    }
}
```

3.3.2.3 master 在 Link Layer 上更新连接参数

Slave 发送 conn para update req，并且 master 回 conn para update rsp 接受申请后，master 会发送 link layer 层的 LL_CONNECTION_UPDATE_REQ 命令，如下图所示。

图 3- 37 抓包显示 ll conn update req

slave 收到此更新请求后，记下最后一个参数为 master 端的 instant 的值，当 slave 端的 instant 值到达这个值的时候，更新到新的连接参数，并触发回调事件 BLT_EV_FLAG_CONN_PARA_UPDATE。

instant 是 master 和 slave 端各自都维护的连接事件计数值，范围为 0x0000~0xffff，在一个连接中，它们的值一直都是相等的。当 master 发送 conn_req 申请和 slave 连接后，master 开始切换自己的状态（从扫描状态到连接状态），并将 master 端的 instant 清 0。slave 收到 conn_req，从广播状态切换到连接状态，将 slave 端的 instant 清 0。master 和 slave 的每一个连接包都是一个连接事件，两端在 conn_req 后的第一个连接事件，instant 值为 1，第二个连接事件 instant 值为 2，依次往后递增。

当 master 发送 LL_CONNECTION_UPDATE_REQ 时，最后一个参数 instant 是指在标号为 instant 的连接事件时，master 将使用 LL_CONNECTION_UPDATE_REQ 包中前几个连接参数对应值。由于 slave 和 master 的 instant 值始终是相等的，它收到 LL_CONNECTION_UPDATE_REQ 时，在自己的 instant 等于 master 所声明的那个 instant 的连接事件时，使用新的连接参数。这样就可以保证两端在同一个时间点完成连接参数的切换。

3.4 ATT & GATT

3.4.1 GATT 基本单位 Attribute

GATT 定义了两种角色：Server 和 Client。17H25 BLE SDK 中，Slave 是 Server，对应的 Android、iOS 设备是 Client。Server 需要提供多个 service 供 Client 访问。

GATT 的 service 实质是由多个 Attribute 构成，每个 Attribute 都具有一定的信息量，当多个不同种类的 Attribute 组合在一起时，就能够反映出一个基本的 service。

Lenze Technology Co.,LTD

图 3-38 Attribute 构成 GATT service

一个 Attribute 的基本内容和属性包括以下:

1) Attribute Type：UUID

UUID 用来区分每一个 attribute 的类型，其全长为 16 个 bytes。BLE 标准协议中 UUID 长度定义为 2 个 bytes，这是因为 master 设备都遵循同一套转换方法，将 2 个 bytes 的 UUID 转换成 16 bytes。

user 直接使用蓝牙标准的 2 byte 的 UUID 时，master 设备都知道这些 UUID 代表的设备类型。17H25 BLE stack 中已经定义了一些标准的 UUID，分布在以下文件中：proj_lib/ble_l2cap/hids.h、proj_lib/ble_l2cap/gatt_uuid.h、proj_lib/ble_l2cap/service.h。

Lenze 私有的一些 profile（OTA、MIC、SPEAKER 等），标准蓝牙里面不支持，在 proj_lib/ble_l2cap/service.h 中定义这些私有设备的 UUID，长度为 16 bytes。

2) Attribute Handle

slave 拥有多个 Attribute，这些 Attribute 组成一个 Attribute Table。在 Attribute Table 中，每一个 Attribute 都有一个 Attribute Handle 值，用来区分每一个不同的 Attribute。slave 和 master 建立连接后，master 通过 Service Discovery 过程解析读取到 slave 的 Attribute Table，并根据 Attribute Handle 的值来对应每一个不同的 Attribute，这样它们后面的数据通信只要带上 Attribute Handle，对方就知道是哪个 Attribute 的数据了。

3) Attribute Value

每个 Attribute 都有对应的 Attribute Value，用来作为 request、response、notification 和 indication 的数据。在 17H25 BLE stack 中，Attribute Value 用指针和指针所指区域的长度来描述。

3.4.2 Attribute and ATT Table

为了实现 slave 端的 GATT service，17H25 BLE SDK 设计了一个 Attribute Table，该 Table 由多个基本的 Attribute 组成。Attribute 的定义为：

```
typedef struct attribute
{
    u16 attNum;
    u8   perm;
    u8   uuidLen;
    u32  attrLen;    //4 bytes aligned
    u8*  uuid;
    u8*  pAttrValue;
    att_readwrite_callback_t  w;
    att_readwrite_callback_t  r;
} attribute_t;
```

结合目前 17H25 BLE SDK 给出的参考 Attribute Table 来说明以上各项的含义。Attribute Table 代码见 app_att.c，如下截图所示：

```
const attribute_t my_Attributes[] = {
    {50,0,0,0,0,0}, // total num of attribute

    // 0001 - 0007  gap
    {7,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID),  (u8*)(&my_gapServiceUUID), 0},
    {0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characteristicUUID),      (u8*)(&my_devNameCharacter), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_devName), (u8*)(&my_devNameUUID), (u8*)(my_devName), 0},
    {0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characteristicUUID),      (u8*)(&my_appearanceCharacter),
    {0,ATT_PERMISSIONS_READ,2,sizeof (my_appearance), (u8*)(&my_appearanceUIID),    (u8*)(&my_ap
    {0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characteristicUUID),      (u8*)(&my_periConnParamChar), 0
    {0,ATT_PERMISSIONS_READ,2,sizeof (my_periConnParameters),(u8*)(&my_periConnParamUUID),  (u8


    // 0008 - 000a  device Information Service
    {3,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID),  (u8*)(&my_devServiceUUID), 0},
    {0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characteristicUUID),      (u8*)(&my_PnPCharacter), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof (my_PnPtrs),(u8*)(&my_PnPUUID), (u8*)(my_PnPtrs), 0},


    ////////////////////////////////// 4. HID Service ////////////////////////////////////////
    // 000b
    {27,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID),     (u8*)(&my_hidServiceUUID),

    // 000c - 000e  include battery service
    {0,ATT_PERMISSIONS_READ,2,sizeof(include),(u8*)(&hidIncludeUUID),   (u8*)(include), 0},

    // 000d - 000e  protocol mode
    {0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characteristicUUID),      (u8*)(&protocolModeProp), 0},
    {0,ATT_PERMISSIONS_RDWR,2, sizeof(protocolMode),(u8*)(&hidProtocolModeUUID),    (u8*)(&prot
```

图 3- 39 17H25 BLE SDK Attribute Table 截图

首先请注意，Attribute Table 的定义前面加了 const：

const attribute_t my_Attributes[] = { ... };

const 关键字会让编译器将这个数组的数据最终都存储到 flash，以节省 ram 空间。这个 Attribute Table 定义在 Flash 上的所有内容是只读的，不能改写。

3.4.2.1 attNum

attNum 有两个作用。

attNum 第一个作用是表示当前 Attribute Table 中有效 Attribute 数目，即 Attribute Handle 的最大值，该数目只在 Attribute Table 数组的第 0 项无效 Attribute 中使用：

{50,0,0,0,0,0},

attNum = 50 表示当前 Attribute Table 中共有 50 个 Attribute。

在 BLE 里，Attribute Handle 值从 0x0001 开始，往后加一递增，而数组的下标从 0 开始，在 Attribute Table 里加上上面这个虚拟的 Attribute，正好使得后面每个 Attribute 在数据里的下标号等于其 Attribute Handle 的值。当定义好了 Attribute Table 后，数 Attribute 在当前 Attribute Table 数组中的下标号，就能知道该 Attribute 当前的 Attribute Handle 值。

将 Attribute Table 中所有的 Attribute 数完，数到最后一个的编号就是当前 Attribute Table 中有效 Attribute 的数目 attNum，目前 SDK 中为 50，user 如果添加或删除了 Attribute，需要对此 attNum 进行修改。

attNum 第二个作用是用于指定当前的 service 由哪几个 Attribute 构成。

每一个 service 的第一个 Attribute 的 UUID 都必须是 GATT_UUID_PRIMARY_SERVICE(0x2800)，在这个 Attribute 上的 attNum 指定从当前 Attribute 开始往后数总共有 attNum 个 Attribute 属于该 service 的组成部分。

如上面截图所示，gap service UUID 为 GATT_UUID_PRIMARY_SERVICE 的那个 Attribute 上 attNum 为 7，则 Attribute Handle 1~ Attribute Handle 7 这 7 个 Attribute 是属于 gap service 的描述。

同样，上图中的 HID service 的首个 Attribute 的 attNum 设为 27 后，从这个 Attribute 开始往后连续 27 个 Attribute 都属于 HID service。

除了第 0 项 Attribute 和每一个 service 首个 Attribute 外，其他所有的 Attribute 的 attNum 的值都必须设为 0。

3.4.2.2 perm

perm 是 permission 的简写。

perm 用于指定当前 Attribute 被 Client 访问的权限。

权限有以下 8 种，每个 Attribute 的权限都必须为下面的值或它们的组合。

```
#define ATT_PERMISSIONS_READ              0x01
#define ATT_PERMISSIONS_WRITE             0x02
#define ATT_PERMISSIONS_AUTHEN_READ       0x04
#define ATT_PERMISSIONS_AUTHEN_WRITE      0x08
#define ATT_PERMISSIONS_AUTHOR_READ       0x10
#define ATT_PERMISSIONS_AUTHOR_WRITE      0x20
#define ATT_PERMISSIONS_ENCRYPT_READ      0x40
#define ATT_PERMISSIONS_ENCRYPT_WRITE     0x80
```

3.4.2.3 uuid、uuidLen

按照之前所述，UUID 分两种：BLE 标准的 2 bytes UUID 和 Lenze 私有的 16 bytes UUID。通过 uuid 和 uuidLen 可以同时描述这两种 UUID。

uuid 是一个 u8 型指针，uuidLen 表示从指针开始的地方连续 uuidLen 个 byte 的内容为当前

Lenze Technology Co., LTD

UUID。Attribute Table 是存在 flash 上的，所有的 UUID 也是存在 flash 上的，所以 uuid 是指向 flash 的一个指针。

1) BLE 标准的 2 bytes UUID：

如 Attribute Handle = 2 的 devNameCharacter 那个 Attribute，相关代码如下：

```
#define GATT_UUID_CHARACTER                    0x2803
static const u16 my_characterUUID = GATT_UUID_CHARACTER;
{0,2,1,1,(u8*)(&my_characterUUID),        (u8*)(&my_devNameCharacter), 0},
```

UUID=0x2803 在 BLE 中表示 character，uuid 指向 my_devNameCharacter 在 flash 中的地址，uuidLen 为 2，master 来读这个 Attribute 时，UUID 会是 0x2803。

2) Lenze 私有的 16 bytes UUID：

如 audio 中 MIC 的 Attribute，相关代码：

```
#define Lenze _MIC_DATA
{0x18,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00}
const u8 my_MicUUID[16]        = Lenze _MIC_DATA;
{0,16,1,1,(u8*)(&my_MicUUID),  (u8*)(&my_MicData), 0},
```

uuid 指向 my_MicData 在 flash 中的地址，uuidLen 为 16，master 来读这个 Attribute 时，UUID 会是 0x000102030405060708090a0b0c0d2b18。

3.4.2.4 pAttrValue、attrLen

每一个 Attribute 都会有对应的 Attribute Value。pAttrValue 是一个 u8 型指针，指向 Attribute Value 所在 RAM/Flash 的地址，attrLen 用来反映该数据在 RAM/Flash 上的长度。当 master 读取 slave 某个 Attribute 的 Attribute Value 时，17H25 BLE SDK 从 Attribute 的 pAttrValue 指针指向的区域（RAM/Flash）开始，取 attrLen 个数据回给 master。

UUID 是只读的，所以 uuid 是指向 flash 的指针；而 Attribute Value 可能会涉及到写操作，如果有写操作必须放在 RAM 上，所以 pAttrValue 可能指向 RAM，也可能指向 Flash。

Attribute Handle=35 hid Information 的 Attribute，相关代码：

```
const u8 hidInformation[] =
{
    U16_LO(0x0111), U16_HI(0x0111),    // bcdHID (USB HID version)，0x11,0x01
    0x00,                              // bCountryCode
    0x01                               // Flags
};
{0,2, sizeof(hidInformation), sizeof(hidInformation),(u8*)(&hidinformationUUID),
                                                    (u8*)(hidInformation), 0},
```

在实际应用中，hid information 4 个 byte 0x01 0x00 0x01 0x11 是只读的，不会涉及到写操作，所以定义时可以使用 const 关键字存储在 flash 上。 pAttrValue 指向 hidInformation 在 flash 上的地址，此时 attrlen 以 hidInformation 实际的长度取值。当 master 读该 Attribute 时，会根据 pAttrValue 和 attrLen 返回 0x01000111 给 master。

master 读该 Attribute 时 BLE 抓包如下图，master 使用 ATT_Read_Req 命令，设置要读的 AttHandle = 0x23 = 35，对应 SDK 中 Attribute Table 中的 hid information。

Lenze Technology Co.,LTD

	Data Type	Data Header					Security Enabled	L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length		L2CAP-Length	ChanId	Opcode	AttHandle			
	L2CAP-S	2	1	0	0	11	Yes	0x0003	0x0004	0x0A	0x0023	0x65CCC5	0	OK

	Data Type	Data Header					Security Enabled	CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length				
	Empty PDU	1	1	1	0	0	Yes	0x2A576A	0	OK

	Data Type	Data Header					Security Enabled	CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length				
	Empty PDU	1	0	1	0	0	Yes	0x2A51B9	0	OK

	Data Type	Data Header					Security Enabled	L2CAP Header		ATT_Read_Rsp		CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length		L2CAP-Length	ChanId	Opcode	AttValue			
	L2CAP-S	2	0	0	0	13	Yes	0x0005	0x0004	0x0B	11 01 00 01	0x9BF6A0	0	OK

图 3-40 master 读 hidInformation 的 BLE 抓包

Attribute Handle=40 battery value 的 Attribute，相关代码：

u8 my_batVal[1] = {99};

{0,2,1,1,(u8*)(&my_batCharUUID), (u8*)(my_batVal), 0},

实际应用中，反应当前电池电量的 my_batVal 值会根据 ADC 采样到的电量而改变，然后通过 slave 主动 notify 或者 master 主动读的方式传输给 master，所以 my_batVal 应该放在内存上，此时 pAttrValue 指向 my_batVal 在 RAM 上的地址。

3.4.2.5 回调函数 w

回调函数 w 是写函数。函数原型：

typedef int (*att_readwrite_callback_t)(void* p);

user 如果需要定义回调写函数，须遵循上面格式。回调函数 w 是 optional 的，对某一个具体的 Attribute 来说，user 可以设置回调写函数，也可以不设置回调（不设置回调的时候用空指针 0 表示）。

回调函数 w 触发条件为：当 slave 收到的 Attribute PDU 的 Attribute Opcode 为以下两个时，slave 会检查回调函数 w 是否被设置：

1) opcode = 0x12，Write Request，参考《Core_v4.2》Page2191。
2) opcode = 0x52，Write Command，参考《Core_v4.2》Page2193。

slave 收到以上写命令后，如果没有设置回调函数 w，slave 会自动向 pAttrValue 指针所指向的区域写 master 传过来的值，写入的长度为 master 数据包格式中的 l2capLen-3；如果 user 设置了回调函数 w，slave 收到以上写命令后执行 user 的回调函数 w，此时不再向 pAttrValue 指针所指区域写数据。这两个写操作是互斥的，只能有一个生效。

user 设置回调函数 w 是为了处理 master 在 ATT 层的 Write request 和 Write Command 命令，如果没有设置回调函数 w，需要评估 pAttrValue 所指向的区域是否能够完成对以上命令的处理（如 pAttrValue 指向 flash 无法完成写操作；或者 attrLen 长度不够，master 的写操作会越界，导致其他数据被错误的改写）。

Lenze Technology Co., LTD

3.4.5.1 Write Request

The *Write Request* is used to request the server to write the value of an attribute and acknowledge that this has been achieved in a *Write Response*.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x12 = Write Request
Attribute Handle	2	The handle of the attribute to be written
Attribute Value	0 to (ATT_MTU-3)	The value to be written to the attribute

Table 3.26: Format of Write Request

图 3- 41 BLE 协议栈中 Write Request

3.4.5.3 Write Command

The *Write Command* is used to request the server to write the value of an attribute, typically into a control-point attribute.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x52 = Write Command
Attribute Handle	2	The handle of the attribute to be set
Attribute Value	0 to (ATT_MTU-3)	The value of be written to the attribute

Table 3.28: Format of Write Command

图 3- 42 BLE 协议栈中 Write Command

Lenze Technology Co.,LTD

回调函数 w 的 void 型 p 指针指向 master 写命令的具体数值。实际 p 指向一片内存，内存上的值如下面结构体所示。

```
typedef struct{
    u32 dma_len;
    u8  type;
    u8 rf_len;
    u16 l2cap;      //l2cap_length
    u16 chanid;

    u8  att;        //opcode
    u8  hl;         //low byte of Atthandle
    u8  hh;         //high byte of Atthandle
    u8  dat[20];
    }rf_packet_att_data_t;
```

p 指向 dma_len。写过来的数据有效长度为 l2cap - 3，第一个有效数据为 pw->dat[0]。

```
int     my_WriteCallback (void *p)
{
    rf_packet_att_data_t *pw = (rf_packet_att_data_t *)p;
    int len = pw->l2cap - 3;

    //add your code
    //valid data is pw->dat[0] ~ pw->dat[len-1]

    return 1;
}
```

上面这个结构体 rf_packet_att_data_t 所在位置为 proj_lib/ble/ble_common.h。

3.4.2.6 回调函数 r

回调函数 r 是读函数。函数原型：

typedef int (*att_readwrite_callback_t)(void* p);

user 如果需要定义回调读函数，须遵循上面格式。回调函数 r 是 optional 的，对某一个具体的 Attribute 来说，user 可以设置回调读函数，也可以不设置回调（不设置回调的时候用空指针 0 表示）。

Lenze Technology Co.,LTD

回调函数 r 触发条件为：当 slave 收到的 Attribute PDU 的 Attribute Opcode 为以下两个时，slave 会检查回调函数 r 是否被设置：

1) opcode = 0x0A，Read Request，参考《Core_v4.2》Page2184。
2) opcode = 0x0C，Read Blob Request，参考《Core_v4.2》Page2185。

slave 收到以上读命令后，

1) 如果 user 设置了回调读函数，执行该函数，根据该函数的返回值决定是否回复 Read Response/Read Blob Response：
   A. 若返回值为 1，slave 不回复 Read Response/Read Blob Response 给 master。
   B. 若返回值为其他值，slave 从 pAttrValue 指针所指向的区域读 attrLen 个值用 Read Response/Read Blob Response 回复给 master。
2) 如果 user 没有设置回调读函数，slave 从 pAttrValue 指针所指向的区域读 attrLen 个值用 Read Response/Read Blob Response 回复给 master。

如果 user 想在收到 master 的 Read Request/Read Blob Request 后修改即将回复的 Read Response/Read Blob Response 的内容，就可以注册对应的回调函数 r，在回调函数里修改 pAttrValue 指针所指 ram 的内容，并且 return 的值只能是 0。

3.4.2.7 Attribute Table 结构

根据以上对 Attribute 的详细说明，使用 Attribute Table 构造 Service 结构如下图所示。第一个 Attribute 的 attnum 用于指示当前 ATT Table Attribute 的数量，剩余的 Attribute 首先按 Service 分组，每一组的头一条 Attribute 是该 Service 的 declaration，并且使用 attnum 指定后面紧跟的多少条 Attribute 属于该 Service 的具体描述。实际每组 Service 的第一条是一个 Primary Service。

```
const u16 my_primaryServiceUUID = GATT_UUID_PRIMARY_SERVICE;
#define GATT_UUID_PRIMARY_SERVICE          0x2800        //!< Primary Service
```

Lenze Technology Co.,LTD

图 3- 43 Service/Attribute Layout

3.4.2.8 ATT table Initialization

GATT & ATT 初始化只需要将应用层的 Attribute Table 的指针传到协议栈即可，提供的 API：

```
void        bls_att_setAttributeTable (u8 *p);
```

p 为 Attribute Table 的指针。

3.4.3 Attribute PDU & GATT API

根据 BLE Spec，17H25 BLE SDK 目前支持的 Attribute PDU 有以下几类：

1) Requests：client 发送给 server 的数据请求。

2) Responses：server 收到 client 的 request 后发送的数据回应。
3) Commands：client 发送给 server 的命令。
4) Notifications：server 发送给 client 的数据。
5) Indications：server 发送给 client 的数据。
6) Confirmations：client 对 server 数据的确认。

下面结合之前介绍的 Attribute 结构和 Attribute Table 结构，对 ATT 层所有的 ATT PDU 进行分析。

3.4.3.1 Read by Group Type Request、Read by Group Type Response

master 发送 Read by Group Type Request，在该命令中指定起始和结束的 attHandle，指定 attGroupType。slave 收到该 Request 后，遍历当前 Attribute table，在指定的起始和结束的 attHandle 中找到符合 attGroupType 的 Attribute Group，通过 Read by Group Type Response 回复 Attribute Group 信息。

图 3- 44 Read by Group Type Request/Read by Group Type Response

上图所示，master 查询 slave 的 UUID 为 0x2800 的 primaryServiceUUID 的 Attribute Group 信息：

```
#define GATT_UUID_PRIMARY_SERVICE          0x2800
const u16 my_primaryServiceUUID = GATT_UUID_PRIMARY_SERVICE;
```

参考当前 demo code，slave 的 Attribute table 中有以下几组符合该要求：

1) attHandle 从 0x0001 ~ 0x0007 的 Attribute Group，Attribute Value 为 SERVICE_UUID_GENERIC_ACCESS（0x1800）。
2) attHandle 从 0x0008 ~ 0x000a 的 Attribute Group，Attribute Value 为 SERVICE_UUID_DEVICE_INFORMATION（0x180A）。

3) attHandle 从 0x000B ~ 0x0025 的 Attribute Group，Attribute Value 为 SERVICE_UUID_HUMAN_INTERFACE_DEVICE（0x1812）。
4) attHandle 从 0x0026 ~ 0x0028 的 Attribute Group，Attribute Value 为 SERVICE_UUID_BATTERY（0x180F）。
5) attHandle 从 0x0029 ~ 0x0032 的 Attribute Group，Attribute Value 为 Lenze _AUDIO_UUID_SERVICE
（0x11,0x19,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00）。

slave 将以上 5 个 GROUP 的 attHandle 和 attValue 的信息通过 Read by Group Type Response 回复给 master，最后一个 ATT_Error_Response 表明所有的 Attribute Group 都已回复完毕，Response 结束，master 看到这个包也会停止发送 Read by Group Type Request。Read by Group Type Request 和 Read by Group Type Response 的命令详细说明参考《Core_v4.2》Page2188。

3.4.3.2 Find by Type Value Request、Find by Type Value Response

master 发送 Find by Type Value Request，在该命令中指定起始和结束的 attHandle，指定 AttributeType 和 Attribute Value。slave 收到该 Request 后，遍历当前 Attribute table，在指定的起始和结束的 attHandle 中找到 AttributeType 和 Attribute Value 相匹配的 Attribute，通过 Find by Type Value Response 回复 Attribute。

Find by Type Value Request 和 Find by Type Value Response 的命令详细说明参考《Core_v4.2》Page2179。

图 3- 45 Find by Type Value Request/Find by Type Value Response

3.4.3.3 Read by Type Request、Read by Type Response

master 发送 Read by Type Request，在该命令中指定起始和结束的 attHandle，指定 AttributeType。slave 收到该 Request 后，遍历当前 Attribute table，在指定的起始和结束的 attHandle 中找到符合 AttributeType 的 Attribute，通过 Read by Type Response 回复 Attribute。

图 3-46 Read by Type Request/Read by Type Response

上图所示，master 读 attType 为 0x2A00 的 Attribute，slave 中 Attribute Handle 为 00 03 的 Attribute：

```
const u8  my_devName [] = {'t', 'S', 'e', 'l', 'f', 'i'};
#define GATT_UUID_DEVICE_NAME                    0x2a00
const u16 my_devNameUUID = GATT_UUID_DEVICE_NAME;
{0,2,sizeof (my_devName), sizeof (my_devName),(u8*)(&my_devNameUUID),
                                                (u8*)(my_devName), 0},
```

Read by Type response 中 length 为 8，attData 中前两个 byte 为当前的 attHandle 0003，后面 6 个 bytes 为对应的 Attribute Value。

Read by Type Request 和 Read by Type Response 的命令详细说明《Core_v4.2》Page2181。

3.4.3.4 Find information Request、Find information Response

master 发送 Find information request，指定起始和结束的 attHandle。slave 收到该命令后，将起始和结束的所有 attHandle 对应 Attribute 的 UUID 通过 Find information response 回复给 master。如下图所示，master 要求获得 attHandle 0x0016 ~0x0018 三个 Attribute 的 information，slave 回复这三个 Attribute 的 UUID。

图 3-47 Find information request/Find information response

Find information request 和 Find information response 的命令详细说明《Core_v4.2》Page2177。

3.4.3.5 Read Request、Read Response

master 发送 Read Request，指定某一个 attHandle，slave 收到后通过 Read Response 回复指定的 Attribute 的 Attribute Value（若设置了回调函数 r，执行该函数），如下图所示。

Read Request 和 Read Response 的命令详细说明见《Core_v4.2》Page2184。

图 3-48 Read Request/Read Response

3.4.3.6 Read Blob Request、Read Blob Response

当 slave 某个 Attribute 的 Attribute Value 值的长度超过 MTU_SIZE（目前 SDK 中为 23）时，master 需要启用 Read Blob Request 来读取该 Attribute Value，从而使得 Attribute Value 可以分包发送。master 在 Read Blob Request 指定 attHandle 和 ValueOffset，slave 收到该命令后，找到对应的 Attribute，根据 ValueOffset 值通过 Read Blob Response 回复 Attribute Value（若设置了回调函数 r，执行该函数）。

如下图所示，master 读 slave 的 HID report map（report map 很大，远远超过 23）时，首先发送 Read Request，slave 回 Read response，将 report map 前一部分数据回给 master。之后 master 使用 Read Blob Request，slave 通过 Read Blob Response 回数据给 master。

Read Blob Request 和 Read Blob Response 的命令详细说明参考《Core_v4.2》Page2185。

图 3- 49 Read Blob Request/Read Blob Response

3.4.3.7 Exchange MTU Request、Exchange MTU Response

如下面所示，master 和 slave 通过 Exchange MTU Request 和 Exchange MTU Response 获知对方的 MTU size。

Exchange MTU Request 和 Exchange MTU Response 的命令详细说明参考《Core_v4.2》Page2175。

图 3- 50 Exchange MTU Request/Exchange MTU Response

当 Lenze BLE Slave GATT 层的的数据访问过程中出现超过一个 RF 包长度的数据，涉及到 GATT 层分包和拼包时，需要提前和 master 交互双方的 RX MTU size，也就是 MTU size exchange 的过程。MTU size exchange 的目的是为了实现 GATT 层长包数据的收发。

1) MTU size exchange 的回调函数

函数原型：

```
typedef int (*att_mtuSizeExchange_callback_t)(u16, u16);
```

第一个 u16 为当前 connection handle，在 Slave 应用中都为 BLS_CONN_HANDLE，第二个 u16 为对方 Master 端的 ClientRxMTU，根据该 ClientRxMTU Slave 可确定发送的数据最大长度。

注册回调函数接口：

```
void blc_att_registerMtuSizeExchangeCb
(att_mtuSizeExchange_callback_t cb);
```

2) 17H25 Slave GATT 层收长包数据的处理。

17H25 Slave ServerRxMTU 默认为 23，实际最大 ServerRxMTU 可以支持到 241，即 master 端 241 个 byte 的分包数据在 Slave 端可以正确的完成数据包重拼。当应用中需要使用到 master 的分包重拼时，使用下面 API 先修改 Slave 端的 RX size：

```
ble_sts_t blc_att_setRxMtuSize(u16 mtu_size);
```

返回值列表：

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
ATT_ERR_INVALID_PARAMETER	0x12	mtu_size 大于最大值 241

如果 Master GATT 层有长包数据需要发送给 Slave，Master 端会主动发起 ATT_Exchange_MTU_req，此时 Slave 回复 ATT_Exchange_MTU_rsp，其中 ServerRxMTU 为上面 API blc_att_setRxMtuSize 设置的值，且此时触发 user 通过 blc_att_registerMtuSizeExchangeCb 注册的回调函数，在回调函数第 2 个参数为 Master 端的 ClientRxMTU。

3) 17H25 Slave GATT 层发长包数据的处理。

当 17H25 Slave 需要在 GATT 层发送长包数据时，需要先获取到 Master 的 Client RxMTU，最终的数据长度不能大于 ClientRxMTU。

Slave 先使用 API blc_att_setRxMtuSize 设置自己的 ServerRxMTU，假设设为 158。

blc_att_setRxMtuSize（158）；

再调用下面 API 主动发起一个 ATT_Exchange_MTU_req：

```
ble_sts_t blc_att_requestMtuSizeExchange (
u16 connHandle, u16 mtu_size);
```

connHandle 为 Slave conection 的 ID，为 BLS_CONN_HANDLE，mtu_size 为 ServerRxMTU。

blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, 158);

Master 收到 ATT_Exchange_MTU_req 后，回复 ATT_Exchange_MTU_rsp，SDK 收到后触发 blc_att_registerMtuSizeExchangeCb 注册的回调函数，在回调函数第 2 个参数为 Master 端的 ClientRxMTU。

3.4.3.8 Write Request、Write Response

master 发送 Write Request，指定某个 attHandle，并附带相关数据。slave 收到后，找到指定的 Attribute，根据 user 是否设置了回调函数 w 决定数据是使用回调函数 w 来处理还是直接写入对应的 Attribute Value，并回复 Write Response。

下图所示为 master 向 attHandle 为 0x0016 的 Attribute 写入 Attribute Value 为 0x0001，slave 收到后执行该写操作，并回 Write Response。

Lenze Technology Co., LTD

Write Request 和 Write Response 的命令详细说明参考《Core_v4.2》Page2191。

Data Type	Data Header					L2CAP Header		ATT_Write_Req			CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
L2CAP-S	2	0	1	0	9	0x0005	0x0004	0x12	0x0016	01 00	0xDC8476	-38	OK

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
Empty PDU	1	0	0	0	0	0xAE00D5	-38	OK

Data Type	Data Header					CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length			
Empty PDU	1	1	0	0	0	0xAE0606	-38	OK

Data Type	Data Header					L2CAP Header		ATT_Write_Rsp	CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode			
L2CAP-S	2	1	1	0	5	0x0001	0x0004	0x13	0xFBDB12	-38	OK

图 3- 51 Write Request/Write Response

Lenze Technology Co.,LTD

3.4.3.9 Write Command

master 发送 Write Command，指定某个 attHandle，并附带相关数据。slave 收到后，找到指定的 Attribute，根据 user 是否设置了回调函数 w 决定数据是使用回调函数 w 来处理还是直接写入对应的 Attribute Value，不回复任何信息。Write Command 的命令详细说明见参考《Core_v4.2》Page2193。

3.4.3.10 Handle Value Notification

参考《Core_v4.2》Page2199。

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1B = Handle Value Notification
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.34: Format of Handle Value Notification

图 3- 52 BLE Spec 中 Handle Value Notification

上图所示为 BLE Spec 中 Handle Value Notification 的格式。

17H25 BLE SDK 提供 API，用于某个 Attribute 的 Handle Value Notification。user 调用这个 API 以将自己需要 notify 的数据 push 到底层的 BLE 软件 fifo，协议栈会在最近的收发包 interval 时将软件 fifo 的数据 push 到硬件 fifo，最终通过 RF 发送出去。

```
ble_sts_t bls_att_pushNotifyData (u16 handle, u8 *p, int len);
```

handle 为对应 Attribute 的 attHandle，p 为要发送的连续内存数据的头指针，len 指定发送的数据的字节数。该 API 支持自动拆包功能，可将一个很长的数据拆成多个 BLE RF packet 发送出去，所以 len 可以支持很大。

Link Layer 在 Conn state 时，一般情况下直接调用该 API 可以成功 push 数据到底层软件 fifo，但也存在一些特殊情况会导致该 API 调用失败，根据返回值 ble_sts_t 可以了解对应的错误原因。

建议应用层调用该 API 时，检查一下返回值是否为 BLE_SUCCESS，若不为 BLE_SUCCESS，则需要等待一段时间后再次 push。

返回值列表如下。

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer 处于 None Conn state
SMP_EER_PAIRING_IS_GOING_ON	0x8F	处于配对阶段，不能发送数据
HCI_ERR_CONTROLLER_TX_FIFO_NOT_ENOUGH	0x3A	有大数据量任务在运行，软件 Tx fifo 不够用

3.4.3.11 Handle Value Indication

参考《Core_v4.2》Page2199。

Lenze Technology LTD

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1D = Handle Value Indication
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.35: Format of Handle Value Indication

图 3-53 BLE Spec 中 Handle Value Indication

上图所示为 BLE Spec 中 Handle Value Indication 的格式。

17H25 BLE SDK 提供 API，用于某个 Attribute 的 Handle Value Indication。user 调用这个 API 以将自己需要 indicate 的数据 push 到底层的 BLE 软件 fifo，协议栈会在最近的收发包 interval 时将软件 fifo 的数据 push 到硬件 fifo，最终通过 RF 发送出去。

```
ble_sts_t bls_att_pushIndicateData (u16 handle, u8 *p, int len);
```

handle 为对应 Attribute 的 attHandle，p 为要发送的连续内存数据的头指针，len 指定发送的数据的字节数。该 API 支持自动拆包功能，可将一个很长的数据拆成多个 BLE RF packet 发送出去，所以 len 可以支持很大。

BLE Spec 里规定了每一个 indicate 的数据，都要等到 Master 的 confirm 才能认为 indicate 成功，未成功时不能发送下一个 indicate 数据。

Link Layer 在 Conn state 时，一般情况下直接调用该 API 可以成功 push 数据到底层软件 fifo，但也存在一些特殊情况会导致该 API 调用失败，根据返回值 ble_sts_t 可以了解对应的错误原因。建议应用层调用该 API 时，检查一下返回值是否为 BLE_SUCCESS，若不为 BLE_SUCCESS，则需要等待一段时间后再次 push。返回值列表如下。

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer 处于 None Conn state
SMP_EER_PAIRING_IS_GOING_ON	0x8F	处于配对阶段，不能发送数据
HCI_ERR_CONTROLLER_TX_FIFO_NOT_ENOUGH	0x3A	有大数据量任务在运行，软件 Tx fifo 不够用
ATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	0x6B	前一个 indicate 的数据还没有被 master confirm

3.4.3.12 Handle Value Confirmation

参考《Core_v4.2》Page2200。

应用层每调用一次 bls_att_pushIndicateData，向 master 发送 indicate 数据后，master 会回复一个 confirm，表示对这个数据的确认，然后 slave 才可以继续发送下一个 indicate 数据。

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1E = Handle Value Confirmation

Table 3.36: Format of Handle Value Confirmation

图 3-54 BLE Spec 中 Handle Value Confirmation

Lenze Technology Co.,LTD

从上图中可以看出，Confirmation 并不指定是对哪一个具体 handle 的确认，对所有不同 handle 上的 indicate 数据都统一回复一个 Confirmation。

为了让应用层了解发送出去的 indicate data 是否已经被 Confirm，SDK 提供 Confirmation 的注册回调函数，只要注册回调函数，每收到一个 Handle Value Confirmation，就执行一次回调函数。

定义回调函数类型：

```
typede fint (*att_handleValueConfirm_callback_t)(void);
```

注册回调函数的API：

```
void   bls_att_registerHandleValueConfirmCb
                    (att_handleValueConfirm_callback_t cb);
```

3.4.4 17H25 master GATT

在 17H25 master kma dongle 中，提供了以下 GATT API，用于做简单的 service discovery 或其他数据访问功能。

```
void   att_req_find_info(u8 *dat, u16 start_attHandle, u16 end_attHandle);
```

dat 实际长度（byte）：11 。

```
void   att_req_find_by_type   (u8   *dat,   u16   start_attHandle,   u16
end_attHandle, u8 *uuid, u8* attr_value, int len);
```

dat 实际长度（byte）： 13 + attr_value 长度。

```
void   att_req_read_by_type   (u8   *dat,   u16   start_attHandle,   u16
end_attHandle, u8 *uuid, int uuid_len);
```

dat 实际长度（byte）： 11 + uuid 长度。

```
void   att_req_read (u8 *dat, u16 attHandle);
```

dat 实际长度（byte）：9 。

```
void   att_req_read_blob (u8 *dat, u16 attHandle, u16 offset);
```

dat 实际长度（byte）：11 。

```
void   att_req_read_by_group_type  (u8  *dat,  u16  start_attHandle,  u16
end_attHandle, u8 *uuid, int uuid_len);
```

dat 实际长度（byte）： 11 + uuid 长度。

```
void   att_req_write (u8 *dat, u16 attHandle, u8 *buf, int len);
```

dat 实际长度（byte）：9 + buf 数据长度。

```
void   att_req_write_cmd (u8 *dat, u16 attHandle, u8 *buf, int len);
```

dat 实际长度（byte）：9 + buf 数据长度。

Lenze Technology Co.,LTD

以上 API，需要预先定义内存空间*dat，然后调用 API 进行数据组装，最后调用 blm_push_fifo 将 dat 送到 Controller 发送，并且注意需要判断返回值是否为 TRUE。以 att_req_find_info 为例如下，其他接口都可使用类似方法。

```
u8   cmd[12];
att_req_find_info(cmd,   0x0001,   0x0003);
if( blm_push_fifo (BLM_CONN_HANDLE, cmd) ){
    //cmd send OK
}
```

使用上面参考的方法向 Slave 发送了对应的 find info req、read req 等 cmd 后，很快会收到 Slave 回复的 find info rsp、read rsp 等对应的 response 信息，在 int app_l2cap_handler (u16 conn_handle, u8 *raw_pkt)中按照如下框架处理即可：

```
if(ptrL2cap->chanId == L2CAP_CID_ATTR_PROTOCOL)  //att data
{
    if(pAtt->opcode == ATT_OP_EXCHANGE_MTU_RSP){
        //add your code
    }
    if(pAtt->opcode == ATT_OP_FIND_INFO_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_FIND_BY_TYPE_VALUE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BY_TYPE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BLOB_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BY_GROUP_TYPE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_WRITE_RSP){
        //add your code
    }
}
```

Lenze Technology Co.,LTD

3.5 SMP

3.5.1 SMP 参数配置

SMP 初始化相关参数的配置包括：是否需要进行绑定(bonding)，是否需要进行带外数据验证。具体介绍如下：

3.5.1.1 设备绑定

当设备需要在配对完毕之后绑定对方设备信息时，设备需要将当前的设备绑定请求使能。相关的函数为

int blc_smp_enableBonding (int en);

参数解析：当 en == 0 时，当前设备不会进行绑定操作。

当 en == 1 时，当前设备会进行绑定操作。

如果设备不进行此项设置，当前设备默认为绑定使能。

3.5.1.2 设备带外数据验证(OOB)

带外数据验证，相关函数如下:

void blc_smp_enableOobFlag (int en, u8 *oobData);

参数解析： en -- 表示是否使能带外数据验证。

en == 0, 不使能带外数据验证。

en == 1, 使能带外数据验证。

oobData -- 带外数据验证值，为一组 16byte 数据指针。

如果设备不进行此项设置，则默认设备不进行带外数据验证。

Lenze Technology Co.,LTD

3.5.2 SMP 使能

```
int bls_smp_enableParing (smp_paringTrriger_t encrypt_en);
```

枚举类型 smp_paringTrriger_t 具体定义以及每个参数的意义介绍如下:

```
typedef enum{
    SMP_PARING_DISABLE_TRRIGER = 0,
    SMP_PARING_CONN_TRRIGER ,
    SMP_PARING_PEER_TRRIGER,
}smp_paringTrriger_t;
```

1) `encrypt_en = SMP_PARING_DISABLE_TRRIGER;`

表示设备端不会对当前连接进行配对加密过程，即使对方请求配对加密时，设备端也会拒绝配对加密。一般用于当前设备不支持设备加密配对的过程。

如下图，master 发起配对请求，slave 回复 SM_Pairing_Failed。

图3- 55抓包显示Pairing Disable

2) `encrypt_en = SMP_PARING_CONN_TRRIGER;`

表示当前设备一经连接便会主动发起配对请求加密请求，并且如果在当前设备发送配对请求之前，对方设备已经发送配对请求，当前设备依旧会发送配对请求，并且也会响应对方设备的请求。

如下图，slave主动发送SM_Security_Req 配对连接请求：

图3- 56抓包显示Pairing Conn Trigger

3) `encrypt_en = SMP_PARING_PEER_TRRIGER;`

表示当前设备不会主动发起配对请求，只有在对方发起配对请求时响应对方的配对请求。如果对方不发送配对请求，则不会进行加密配对。

如下图，在master发送配对请求SM_Pairing_Req时，slave会响应，但是不会主动发起配对请求。

Lenze Technology Co.,LTD

图3- 57抓包显示Pairing Peer Trigger

注意：当前函数只能在连接之前调用。建议在初始化的时候调用。

3.5.3 SMP event

前面在 Controller 里介绍了 Lenze defined event，其中有一些 event 不属于 Controller，BLT_EV_FLAG_PAIRING_BEGIN 和 BLT_EV_FLAG_PAIRING_END 属于 SMP，下面具体介绍。

3.5.3.1 BLT_EV_FLAG_PAIRING_BEGIN

事件触发条件：当 slave 和 master 刚刚连接进入 connection state，slave 发送 SM_Security_Req 命令后，master 发送 SM_Pairing_Req 请求开始配对，slave 收到这个配对请求命令时，触发该事件，表示配对开始。

图 3- 58 master 发起 Pairing_Req

回传指针 p：空指针 NULL。
数据长度 n：0。

3.5.3.2 BLT_EV_FLAG_PAIRING_END

事件触发条件：配对结束，包括配对正确完成和错误结束。错误结束时由于 slave/master 其中一个不符合标准的配对流程，或者通信中出现错报等异常原因导致。
数据长度 n：1。
回传指针 p：指向一个标志变量，有以下两种可能：

1) 0
表示正确结束，配对成功。
2) 其他
表示配对失败。

Lenze Technology Co.,LTD

3.5.4 SMP 绑定信息说明

参考 17H25 remote 中初始化中设置 direct adv 的代码：

```
u8 bond_number = blc_smp_param_getCurrentBondingDeviceNumber();  //get bonded device number
smp_param_save_t  bondInfo;
if(bond_number)   //at least 1 bonding device exist
{
    blc_smp_param_loadByIndex( bond_number - 1, &bondInfo);  //get the latest bonding device (inde

}

if(bond_number)   //set direct adv
{
    //set direct adv
    u8 status = bls_ll_setAdvParam( MY_ADV_INTERVAL_MIN, MY_ADV_INTERVAL_MAX,
                                    ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY, OWN_ADDRESS_PUBLIC,
                                    bondInfo.peer_addr_type,  bondInfo.peer_addr,
                                    MY_APP_ADV_CHANNEL,
                                    ADV_FP_NONE);
    if(status != BLE_SUCCESS) { write_reg8(0x8000, 0x11);   while(1); }  //debug: adv setting err

    //it is recommended that direct adv only last for several seconds, then switch to indirect adv
    bls_ll_setAdvDuration(MY_DIRECT_ADV_TMIE, 1);
    bls_app_registerEventCallback (BLT_EV_FLAG_ADV_DURATION_TIMEOUT, &app_switch_to_indirect_adv);

}
```

Slave 最多可以同时存储 4 个 master 的配对信息，这 4 个设备都可以回连成功。下面接口用于设定当前存储的最多设备数，不超过 4（SMP_BONDING_DEVICE_MAX_NUM），如果用户不设置的话，默认值也为 4.

```
#define        SMP_BONDING_DEVICE_MAX_NUM                    4
ble_sts_t     blc_smp_param_setBondingDeviceMaxNumber( int device_num);
```

如果设置了 blc_smp_param_setBondingDeviceMaxNumber（4），配对 4 个设备后，一旦配对第 5 个，就会自动将最老的那个设备的配对信息删除，然后存储第 5 个设备的配对信息。

如果设置了 blc_smp_param_setBondingDeviceMaxNumber（2），配对 2 个设备后，一旦配对第 3 个，就会自动将最老的那个（第 1 个）设备的配对信息删除，然后存储第 3 个设备的配对信息。

下面 API 用于获取当前 slave 在 flash 上存储的配对成功的 master 设备数量。

```
u8          blc_smp_param_getCurrentBondingDeviceNumber(void);
```

假设返回值为 3，就说明 flash 上目前存储了 3 个配对成功的设备，这 3 个设备都可以回连成功。

和 BondingDeviceNumber 相关的一个概念叫 index。如果当前 BondingDeviceNumber 为 1，那么只有 1 个 bonding 设备，它的 index 为 0；

如果 BondingDeviceNumber 为 2，两个设备的 index 分别为 0 和 1。index 顺序是以最近成功的一次连接为准的，并不是最近一个配对为准：假设 slave 先和 masterA 配对成功，再和 masterB 配对成功，此时 slave 的 flash 存储上 masterA 为 index 0, masterB 为 index1，因为 masterB 是最近的设备。 接着让 slave 和 masterA 回连一次成功，这时候 masterA 就成为最近的一个设备，此时 index 0 设备变为 masterB，index1 设备变为 masterA。

如果 BondingDeviceNumber 为 3，三个设备的 index 分别为 0、1、2，0 是最老连接的，2 是最新连接的。

如果 BondingDeviceNumber 为 4，四个设备的 index 分别为 0、1、2、3，此时 3 是最新连接

Lenze Technology LTD

的设备。根据上面的描述，若 slave 连续配对 masterA、B、C、D，此时 masterD 是 index3 设备，若 slave 和 masterB 回连一次，则 master B 成为最新的 index3 设备。

由于 index 是以最近连接时间为顺序的。所以也要注意设备配对超过 4 的情况：若连续配对 masterA、B、C、D，再配对 master E，则 slave 会删除最老的 masterA；但如果在配对 masterA、B、C、D 后，先和 masterA 回连一次，此时顺序变为 B、C、D、A，再配对 masterE 的话 slave 就会删除 masterB 的配对信息。

Master 设备绑定信息存储在 flash 上，其格式为：

```
typedef struct {
    u8      flag;
    u8      peer_addr_type;  //address used in link layer connection
    u8      peer_addr[6];

    u8      peer_key_size;
    u8      peer_id_adrType; //peer identity address information in key distribution, used to identify
    u8      peer_id_addr[6];

    u8      own_ltk[16];      //own_ltk[16]
    u8      peer_irk[16];
    u8      peer_csrk[16];

}smp_param_save_t;
```

绑定信息共 64byte。peer_addr_type 和 peer_addr 是 link layer 上 master 的连接地址，设备 direct adv 时使用这个地址。

peer_id_adrType/peer_id_addr 和 peer_irk 是 master 在 key distribution 阶段宣称的 identity address 和 irk，只有当 peer_addr_type 和 peer_addr 是可解析的私有地址(resolvable private addr，简称 RPA)，且用户需要使用地址过滤时，才需要将相关的信息添加到 resolving list 中，以便 slave 可以解析出（参考 17H25feature test 中 TEST_WHITELIST 的用法）。

其他参数 user 不用关注。

下面 API 使用 index 从 flash 上获取设备的信息。

```
u32  blc_smp_param_loadByIndex(u8 index,
                               smp_param_save_t* smp_param_load);
```

返回值为 0 表示获取信息失败，非 0 的值为该信息区域 Flash 首地址。比如当前 bonding 设备为 3 时，获取最近的那个连接设备的相关信息：

blc_smp_param_loadByIndex(2， …)

下面接口使用 master 的地址（link layer 上的连接地址）从 flash 上获取 bonding 的设备的信息。

```
u32  blc_smp_param_loadByAddr(u8 addr_type,
                              u8* addr, smp_param_save_t* smp_param_load);
```

返回值为 0 表示获取信息失败，非 0 的值为该信息区域 Flash 首地址。

3.6 Flash 存储方法设计

默认使用的 flash 数据区 sector 为 0x78000 - 0x78FFF，在 app_config.h 中可以修改：

```
#define FLASH_ADR_PARING   0x78000
```

将 flash 0x78000 开始每 8 个 bytes 划分为一个 area，称 8 bytes area。每个 area 可以存储一

Lenze Technology Co.,LTD

个 Slave 的 mac address，其中第一个 byte 是标志位，第二个 byte 为地址类型，后面 6 个为 6 bytes 的 mac address。

```
typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;
```

flash 存储过程中使用依次往后推 8 bytes area 的方法，第一个有效 slave mac 存储在 0x78000-0x78007，将 0x78000 的第一 byte 标志位写为 0x5A，表示当前地址有效；当存储第二个有效 mac address 时存储在 0x78008-0x7800f，将 0x78008 打上标记 0x5A；当存储第三个有效 mac address 时存储在 0x78010-0x78017，将 0x78010 打上标记 0x5A。

如果要某个 slave 设备解配对，dongle 端需要擦掉这个设备的 mac address，只需要将之前存储该 mac address 8 bytes area 的标志位写为 0x00 即可；如擦掉上面三个 device 中的第一个 device，将 0x78000 写为 0x00 即可。

采用上面这种 8bytes 顺延方法的原因是，程序在运行过程中不能调用 flash_erase_sector 这个函数擦 flash，因为该操作擦一个 sector 4K 的 flash 耗时在 20-200ms 之间，这个时间会引起 BLE 时序的错误。

将所有的 slave mac 的配对存储和解配对擦除使用 0x5A 和 0x00 标志位来表示，当 8 bytes area 越来越多，可能会占满整个 sector 4K flash 导致出错时，在初始化的时候加了特别处理：从 0x78000 开始读取 8 bytes area 信息，将所有的有效 mac address 读到 ram 中的 slave mac table。这过程中检查是否 8 bytes area 太多，太多的话，就擦掉整个 sector，然后将 ram 中维护的 slave mac table 重新写回 0x78000 开始的 8 bytes area。

3.6.1 Slave mac table

```
/* define pair slave max num,
   if exceed this max num, two methods to process new slave pairing
   method 1: overwrite the oldest one(telink use this method)
   method 2: not allow paring unness unpair happend  */
#define USER_PAIR_SLAVE_MAX_NUM       1  //telink use max 1


typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;


typedef struct {
    u32 bond_flash_idx[USER_PAIR_SLAVE_MAX_NUM];  //mark paired slave mac address in flash
    macAddr_t bond_device[USER_PAIR_SLAVE_MAX_NUM];  //macAddr_t alreay defined in ble stack
    u8 curNum;
} user_salveMac_t;
user_salveMac_t user_tbl_slaveMac;
```

用上面结构在 ram 中使用 slave mac table 维护所有的配对设备，改变宏 USER_PAIR_SLAVE_MAX_NUM 即可定义自己想要的最多允许几只配对，Lenze 默认为 1，指维护 1 个设备的配对，user 可以修改这个值。

假设 user 将最多维护 3 个设备，将 USER_PAIR_SLAVE_MAX_NUM 改为 3 后，user_tbl_slaveMac 中 curNum 表示当前 flash 上记录了几个有效的 slave 设备，bond_flash_idx 数组记录有效地址在 flash 上的 8 bytes area 起始地址相对于 0x78000 的偏移量（当解配对这个设备时，可以通过这个偏移量找到 8 bytes area 的标志位，将其写为 0x00），bond_device 数组记录 mac address。

3.6.2 API

基于上面 flash 存储设计和 ram 中 slave mac table 的设计，分别有以下几个 API 可以调用。

3.6.2.1 user_tbl_slave_mac_add

```
int user_tbl_slave_mac_add(u8 adr_type, u8 *adr);
```

添加一个 slave mac， return 1 表示成功， 0 表示失败。当有新的设备配对上时，需要调用此函数。

函数先判断当前 flash 和 slave mac table 中设备是否已经到达最大值。若没有到最大值，无条件添加到 slave mac table，并在 flash 的一个 8 bytes area 上存储。

若已经到最大值。涉及到处理的策略问题：是不允许配对还是直接覆盖最老的，Lenze demo 的方法是直接覆盖最老的，由于 Lenze 最大配对个数为 1，覆盖最老的也就是抢占当前配对设备，先使用 user_tbl_slave_mac_delete_by_index(0)删掉当前设备，再往 slave mac table 里面加入新的。

User 可以根据自己的策略去修改这个函数的实现。

3.6.2.2 user_tbl_slave_mac_search

```
int user_tbl_slave_mac_search(u8 adr_type, u8 * adr)
```

根据 adv report 的设备地址搜索该设备是否已经在 slave mac table 中，即判断当前发广播包的设备是否之前已经和 master 配对上，若是已经配对过的设备可以直接连接。

3.6.2.3 user_tbl_slave_mac_delete_by_adr

```
int user_tbl_slave_mac_delete_by_adr(u8 adr_type, u8 *adr)
```

通过指定地址删除一个配对的设备。

3.6.2.4 user_tbl_slave_mac_delete_by_index

```
void user_tbl_slave_mac_delete_by_index(int index)
```

通过指定 index 删除配对设备。Index 值反映的是设备配对的顺序。如果最大配对个数为 1，配上的那个设备 index 永远为 0；如果如果最大配对个数为 2，第一个配上的设备 index 为 0，第二个配上的设备 index 为 1；依次类推。

3.6.2.5 user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

删除所有配对设备。

Lenze Technology Co.,LTD

3.6.2.6 user_tbl_salve_mac_unpair_proc

```
void user_tbl_salve_mac_unpair_proc(void)
```

处理解配对命令，参考代码中删除所有配对设备，是默认最大配对个数为 1 时的处理方法。User 可以修改该函数实现。

3.6.3 连接和配对

master 收到 Controller 上报的广播包时，有以下两种情况会和 Slave 进行连接：

1) 调用函数 user_tbl_slave_mac_search 来检查当前设备是否已经跟 master 配对过并且没有被解配，如果已经配对过，可以自动连接。

   master_auto_connect = user_tbl_slave_mac_search(pa->adr_type, pa->mac);

   if(master_auto_connect) { create connection }

2) 若当前广播设备不在 slave mac table 里面，不符合自动连接，检查是否满足手动配对条件。SDK 中默认设置了两个手动配对方案，在当前广播设备距离足够近的前提下，一是 master dongle 上配对键被按下；二是当前广播数据是 Lenze 定义的配对广播包数据。代码：

```
//user design manual paring methods
user_manual_paring = dongle_pairing_enable && (rssi > -56);  //button trigger pairing(rssi threshold, short distance)
if(!user_manual_paring){  //special adv pair data can also trigger pairing
    user_manual_paring = (memcmp(pa->data, telink_adv_trigger_paring, sizeof(telink_adv_trigger_paring)) == 0) && (rssi > -56
}
```

   if(user_manual_paring) { create connection }

若是手动配对触发的建立连接，在连接成功建立后，即 HCI LE CONECTION ESTABLISHED EVENT 上报时，将当前设备添加到 slave mac table 中：

```
// if this connection establish is a new device manual paring, should add this device to slave table
if(user_manual_paring && !master_auto_connect){
        user_tbl_slave_mac_add(pc->peer_adr_type, pc->mac);
}
```

Lenze Technology Co.,LTD

3.6.4 解配对

```
void host_unpair_proc(void)
{
    //terminate and unpair proc
    static int master_disconnect_flag;
    if(dongle_unpair_enable){
        if(!master_disconnect_flag && blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){
            if( blm_ll_disconnect(current_connHandle, HCI_ERR_REMOTE_USER_TERM_CONN) == BLE_SUCCESS){
                master_disconnect_flag = 1;
                dongle_unpair_enable = 0;

                #if (BLE_HOST_SMP_ENABLE)
                    tbl_bond_slave_unpair_proc(current_conn_adr_type, current_conn_address); //by telin
                #else
                    user_tbl_salve_mac_unpair_proc();
                #endif
            }
        }
    }
    if(master_disconnect_flag && blc_ll_getCurrentState() != BLS_LINK_STATE_CONN){
        master_disconnect_flag = 0;
    }
}
```

参考上面 code，当解配对条件生效时，Master 先调用 blm_ll_disconnect 断开连接，然后调用 user_tbl_salve_mac_unpair_proc 函数处理解配对，Demo code 直接删掉所有的配对设备，由于默认的最大配对个数是 1，所以也就只删掉了一个。如果 user 设置了比较复杂的配对多个设备，此时应该调用 user_tbl_slave_mac_delete_by_adr 或 user_tbl_slave_mac_delete_by_index 去删除某个设备。

解配对条件的生效，Demo code 中给出了两种情况，一是 master dongle 上解配对按键被按下，二是在 HID keyboard report service 上收到解配对键值 0xFF。

User 也可以按照自己的需要去修改解配对的触发条件。

Lenze Technology Co.,LTD

4 低功耗管理（PM）

低功耗管理（Power Management），本文档中后面会简称为 PM。

4.1 低功耗 driver

低功耗 driver 相关的文件为 proj_lib/pm.h、proj/mcu_spec/gpio.h、proj/mcu_spec/gpio.c。

4.1.1 低功耗模式

17H25 MCU 具有三种基本的运行模式：

1) working mode：此时 MCU 正常执行程序，硬件数字模块正常工作，根据程序需要打开相应的模拟模块和 BLE RF 收发器模块。工作电流在 10~30mA 之间。
2) suspend mode：低功耗模式 1，此时程序停止运行，类似一个暂停功能。IC 硬件上大部分的硬件模块都断电，PM 模块维持正常工作。数字和模拟寄存器、内存上所有的数据和状态都 hold 住，不会丢任何数据，此时纯粹的 IC 电流在 7~8 uA 之间。当 suspend 被唤醒后，程序继续执行。
3) deepsleep mode：低功耗模式 2，此时程序停止运行，IC 上绝大部分的硬件都断电，PM 硬件模块维持工作。数字寄存器和内存掉电，所有数据不再保存，只有模拟寄存器少数几个寄存器不掉电，可以保存一些必要的信息（pm.h 中的 DEEP_ANA_REG）。当 deepsleep 被唤醒时，MCU 将重新开始工作，跟重新上电是一样的效果，程序重新开始进行初始化和执行，可以在 deepsleep 前在 DEEP_ANA_REG 寄存器里存入一些信息，那么在初始化的时候读这些寄存器是否存入了预先设置的信息用来判断程序是一个纯粹的重新上电还是 deepsleep 醒来。在 deepsleep 下纯粹的 IC 电流为 0.7 uA 左右，如果加上内置 flash 的 1 uA 左右，总体电流在 1.7 uA 左右。

在前面介绍的 Link Layer 工作时序中，可以看到在每个 Adv Interval 和 Connection Interval 上 MCU 处于 working mode 的时间占很小的比例，处理完相应的任务后，即可进入 suspend 状态，由于 suspend 的电流非常低且 suspend 占的时间比例大，MCU 平均电流就很低。

当不需要 MCU 工作时，可以设置 MCU 进入 deepsleep 模式，使得功耗极低，通过一些特殊的操作将其唤醒并重新开始跑程序。

4.1.2 低功耗硬件唤醒源

17H25 MCU 的低功耗唤醒源示意图如下，suspend 可被 CORE 和 timer 唤醒，deepsleep 可被 PAD 和 timer 唤醒。17H25 BLE SDK 中，只关注三种唤醒源：

```
enum {
    PM_WAKEUP_PAD   = BIT(4),
    PM_WAKEUP_CORE  = BIT(5),
    PM_WAKEUP_TIMER = BIT(6),
};
```


Lenze Technology Co.,LTD

图 4-1 17H25 MCU 硬件唤醒源

如上图所示，MCU 的 suspend mode 和 deepsleep mode 在硬件上有三个唤醒源：TIMER、CORE 和 PAD。

唤醒源 PM_WAKEUP_TIMER 来自硬件 32k RC timer。32k RC timer 在 SDK 中已经被正确初始化，user 在使用时不需要任何配置，只需要在 cpu_sleep_wakeup()中设置该唤醒源即可。

唤醒源 PM_WAKEUP_CORE 和 PM_WAKEUP_PAD 都来自 GPIO，所有的 GPIO 的高低电平都可以通过 CORE 和 PAD 模块去唤醒 suspend 和 deepsleep。CORE 模块只能唤醒 suspend，PAD 模块可以同时唤醒 suspend 和 deepsleep，但是 17H25 BLE SDK 中，我们约定：

用 GPIO CORE 去唤醒 suspend；

用 GPIO PAD 去唤醒 deepsleep。

由于所有的 GPIO 的电平 level 都可以通过 CORE 和 PAD 去唤醒相应的低功耗模式，当 user 需要指定某些 GPIO 的某个电平 level 作唤醒源时，使用如下 API 进行设定：

1) 配置 GPIO CORE 唤醒 suspend 的 API：

void gpio_set_wakeup(u32 pin, u32 level, int en);

pin 为 GPIO 定义，level： 1 表示高电平唤醒，0 表示低电平唤醒；en: 1 表示 enable，0 表示 disable。举例说明：

```
gpio_set_wakeup(GPIO_PC2,  1,  1);  //GPIO_PC2 CORE 唤醒打开，高电平唤醒
gpio_set_wakeup(GPIO_PC2,  1,  0);  //GPIO_PC2 CORE 唤醒关闭
gpio_set_wakeup(GPIO_PB5,  0,  1);  //GPIO_PB5 CORE 唤醒打开，低电平唤醒
gpio_set_wakeup(GPIO_PB5,  0,  0);  //GPIO_PB5 CORE 唤醒关闭
```

2) 配置 GPIO PAD 唤醒 deepsleep 的 API：

void cpu_set_gpio_wakeup (int pin, int pol, int en);

pin 为 GPIO 定义，pol： 1 表示高电平唤醒，0 表示低电平唤醒；en: 1 表示 enable，0 表示 disable。举例说明：

```
cpu_set_gpio_wakeup (GPIO_PC2,  1,  1);  //GPIO_PC2 PAD 唤醒打开，高电平唤醒
cpu_set_gpio_wakeup (GPIO_PC2,  1,  0);  //GPIO_PC2 PAD 唤醒关闭
```


Lenze Technology Co.,LTD

cpu_set_gpio_wakeup (GPIO_PB5, 0, 1); //GPIO_PB5 PAD 唤醒打开，低电平唤醒
cpu_set_gpio_wakeup (GPIO_PB5, 0, 0); //GPIO_PB5 PAD 唤醒关闭

4.1.3 低功耗模式的进入和唤醒

调用低功耗进入和唤醒的 API“cpu_sleep_wakeup”（proj_lb/pm.h 中），可以让 MCU 进入低功耗模式，并设置相应的唤醒源。

int cpu_sleep_wakeup (int deepsleep, int wakeup_src, u32 wakeup_tick);

第一个参数 deepsleep： 0 表示进入 suspend，1 表示进入 deepsleep。

第二个参数 wakeup_src：设置当前的 suspend/deepsleep 的唤醒源，参数只能是 PM_WAKEUP_PAD、PM_WAKEUP_CORE、PM_WAKEUP_TIMER 中的一个或者多个：suspend 的唤醒源为 PM_WAKEUP_TIMER 和 PM_WAKEUP_CORE；deepsleep 的唤醒源为 PM_WAKEUP_TIMER 和 PM_WAKEUP_PAD。如果 wakeup_src 为 0，那么进入低功耗后，无法被唤醒。

第三个参数 wakeup_tick：当 wakeup_src 中设置了 PM_WAKEUP_TIMER 时，需要设置 wakeup_tick 来决定 timer 在何时将 MCU 唤醒，如果没有设置 PM_WAKEUP_TIMER 唤醒，该参数无意义。wakeup_tick 的值是一个绝对值，按照前面所说的 system tick 来设置，当 system tick 的值达到这个设定的 wakeup_tick 后，低功耗模式被唤醒。wakeup_tick 的值需要根据当前的 system tick 的值，加上需要睡眠的时间换算成的 tick 值，才可以有效的控制睡眠时间。如果没有考虑 system tick，直接对 wakeup_tick 进行设置，唤醒的时间就无法控制。

由于 wakeup_tick 是绝对时间，必须在 32bit 的 system tick 能表示的范围之内，这个 API 能表示的最大睡眠时间是有限的。目前的设计时最大睡眠时间为 32bit 能表示的最大 system tick 对应时间的 3/4。如 16M clock 时，system tick 能表示大概 268S，最大 suspend/deepsleep 时间为 268*3/4=201 S。

int 返回值为以下几个中的一个或多个“或逻辑”的结果。

```
enum {
    WAKEUP_STATUS_COMP   = BIT(0),
    WAKEUP_STATUS_TIMER  = BIT(1),
    WAKEUP_STATUS_CORE   = BIT(2),
    WAKEUP_STATUS_PAD    = BIT(3),

    STATUS_GPIO_ERR_NO_ENTER_PM  = BIT(7),
};
```

上面 5 个值中：

1) WAKEUP_STATUS_COMP 在 BLE SDK 中从不使用，user 不需要了解；
2) *WAKEUP_STATUS_TIMER/ WAKEUP_STATUS_CORE/ WAKEUP_STATUS_PAD* 则和 3 个唤醒源 *PM_WAKEUP_TIMER/ PM_WAKEUP_CORE/ PM_WAKEUP_PAD* 对应，表示当前是由这些唤醒源触发唤醒的；
3) *STATUS_GPIO_ERR_NO_ENTER_PM* 是一个比较特殊的状态，表示当前发生了 GPIO 唤醒错误：比如当设置了某个 GPIO CORE 高电平唤醒，而在这个 GPIO 为高电平的时候尝试调用 cpu_sleep_wakeup 进入 suspend，且设置了 PM_WAKEUP_CORE 唤醒源。此时会出现无法进入 suspend，MCU 立刻退出 cpu_sleep_wakeup 函数，给出返回值 *STATUS_GPIO_ERR_NO_ENTER_PM*。
4) 另外需要注意，返回值可能出现(*WAKEUP_STATUS_TIMER | WAKEUP_STATUS_CORE)* 的情

Lenze Technology Co.,LTD

况，表示两个唤醒源同时生效了。

一般采用如下的形式来控制睡眠时间：

cpu_sleep_wakeup (0, PM_WAKEUP_TIMER, clock_time() + delta_Tick);

delta_Tick 是一个相对的时间（比如 100*CLOCK_SYS_CLOCK_1MS），加上当前的 clock_time()就变成了绝对时间。

举例说明 cpu_sleep_wakeup 的用法：

1) cpu_sleep_wakeup (0, PM_WAKEUP_CORE, 0);
程序执行该函数时进入 suspend 模式，只能被 GPIO CORE 唤醒。
2) cpu_sleep_wakeup (0, PM_WAKEUP_TIMER, clock_time() + 10*CLOCK_SYS_CLOCK_1MS);
程序执行该函数时进入 suspend 模式，只能被 TIMER 唤醒，唤醒时间为执行该函数时的时间加上 10 ms，所以 suspend 时间为 10 ms。
3) cpu_sleep_wakeup (0, PM_WAKEUP_CORE | PM_WAKEUP_TIMER,clock_time() + 50*CLOCK_SYS_CLOCK_1MS);
程序执行该函数时进入 suspend 模式，可被 GPIO CORE 和 TIMER 唤醒，Timer 唤醒的时间设置为 50ms，那么如果在 50ms 之前触发了 GPIO 的唤醒动作，MCU 会被 GPIO 唤醒，如果 50ms 之前无 GPIO 动作，MCU 会被 timer 唤醒。
4) cpu_sleep_wakeup (1, PM_WAKEUP_PAD, 0);
程序执行该函数时进入 deepsleep 模式，可被 GPIO PAD 唤醒。
5) cpu_sleep_wakeup (1, PM_WAKEUP_TIMER, clock_time() + 8* CLOCK_SYS_CLOCK_1S);
程序执行该函数时进入 deepsleep 模式，可被 Timer 唤醒，唤醒时间为执行该函数的 8 s 后。
6) cpu_sleep_wakeup (1, PM_WAKEUP_PAD | PM_WAKEUP_TIMER,clock_time() + 10*CLOCK_SYS_CLOCK_1S);
程序执行该函数时进入 deepsleep 模式，可被 GPIO PAD 和 Timer 唤醒，Timer 唤醒时间为执行该函数的 10 s 后。如果在 10 s 之前触发了 GPIO 动作，被 GPIO 唤醒，否则被 Timer 唤醒。

4.2 BLE 低功耗管理

17H25 BLE SDK 低功耗管理是对 Link Layer 功耗的管理，参考本文档前面对 Link Layer 的介绍。

在 Lenze BLE SDK 中，stack 底层暂时只对 Advertising state 和 Connection state Slave role 做了低功耗管理，并且提供了一套 API 供 user 配置调用，其他状态没有提供低功耗管理或者需要 user 自己调用 PM driver 去实现，如下 PM in Idle state。

4.2.1 PM in Idle state

由前面介绍可知，当 Link Layer 处于 Idle state 时，blt_sdk_main_loop 不执行任何操作，所以在这个状态下，SDK 不提供任何低功耗管理，需要 user 自己来完成低功耗的相关配置，user 可以调用 API cpu_sleep_wakeup()，设置 MCU 进入 suspend 模式或者 deepsleep 模式，并设置相应的唤醒源，来完成低功耗的管理。

Lenze Technology Co.,LTD

```
void main_loop ()
{
    tick_loop ++;

    ////////////////////////////////////// BLE entry ////////////////////////////////////
    blt_slave_main_loop ();


    ////////////////////////////////////// UI entry ////////////////////////////////////

    //add user task


    if(bls_ll_getCurrentState() == BLS_LINK_STATE_IDLE){  //Idle state
        cpu_sleep_wakeup(0, PM_WAKEUP_TIMER, clock_time() + 10 *CLOCK_SYS_CLOCK_1MS);
    }
    else{
        blt_pm_proc();  //BLE Adv & Conn state
    }

}
```

图 4- 2 Link Layer Idle state 低功耗管理

上图所示为一个简单的参考 code，当 Link Layer 处于 Idle state 时，每个 mainloop suspend 10ms。

在 Idle state 时，也可以直接进入 deepsleep 模式。

4.2.2 PM in BLE Adv state & Conn state

当 Link Layer 处于 Advertising state 或 Conn state Slave role 时，根据前面时序图可知：

1) 处于 Advertising state 时，每个 Adv Interval 里，除了 Adv Event 占用的时间，剩余的时间可以用来做 UI task。或者进入 suspend 模式。
2) 处于 Conn state Slave role 时，每个 Conn interval 内，除了 brx Event（brx start+brx working+brx post）占用的时间，剩余的时间可以用来做 UI task，或者进入 suspend。

BLE 状态的低功耗管理实际就是对上面两个 UI task/suspend 时间进行管理，user 自己来规划这些时间是运行自己的任务，还是进入 suspend 模式以节省功耗。

BLE 状态的低功耗管理不包括对 deepsleep 的管理，user 如果需要进入 deepsleep，在 UI 层直接调用 cpu_sleep_wakeup 进入 deepsleep 即可。

BLE 状态的低功耗管理不需要 user 直接调用 PM driver 层的 API cpu_sleep_wakeup。17H25 BLE SDK 根据 Link Layer 的状态和低功耗模式，在 BLE 协议栈部分加了一套低功耗管理机制（对应的代码在 blt_sdk_main_loop 中实现），user 只需要调用相应的 API 即可对低功耗进行配置管理。

4.3 BLE 低功耗管理配置方法

4.3.1 PM 模块初始化

类似于 Link Layer 各状态机的设计方法，PM 模块也需要在初始化的时候将该模块打开了，调用下面 API 即可。若不需要 PM 的应用，不调用此 API，则 PM 相关的代码和变量都不会被

Lenze Technology Co.,LTD

编译到程序中，可以节省很多资源。

```
void        blc_ll_initPowerManagement_module(void);
```

4.3.2 设置低功耗模式 bls_pm_setSuspendMask

用于配置 17H25 BLE SDK Link Layer Advertising state 和 Conn state 低功耗管理的 API 为：

```
void        bls_pm_setSuspendMask (u8 mask);
u8          bls_pm_getSuspendMask (void);
```

使用 bls_pm_setSuspendMask 设置一个底层变量（称之为 SuspendMask，上电默认为 SUSPEND_DISABLE，代码中实际变量为 bls_pm.suspend_mask），配置低功耗模式。

bls_pm_getSuspendMask 用于获取当前 SuspendMask 的值，该值等于 user 最近一次调用 bls_pm_setSuspendMask 时设置的值，若 user 没有设置则等于默认的 SUSPEND_DISABLE。

SuspendMask 可取如下几个值。

```
/////////////////// Power Management //////////////////////
#define            SUSPEND_DISABLE          0
#define            SUSPEND_ADV              BIT(0)
#define            SUSPEND_CONN             BIT(1)
#define            MCU_STALL                BIT(6)
```

以上 mask 中 MCU_STALL 是一个特殊的 mode，后面会介绍。

bls_pm_setSuspendMask 的设置方法，请结合 Link Layer 工作时序和后面的低功耗管理工作机制的说明来理解。

SuspendMask 的设置，可以选择上面几个值中的一个，或者选择多个值的“或操作”，但是“或”操作只限于 Advertising state 和 Conn state 的组合值。SuspendMask 所有可以设置的值如下：

```
bls_pm_setSuspendMask(SUSPEND_ADV);
bls_pm_setSuspendMask(SUSPEND_CONN);
bls_pm_setSuspendMask(MCU_STALL);
bls_pm_setSuspendMask(SUSPEND_DISABLE);
bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN);
```

4.3.3 设置低功耗唤醒源 bls_pm_setWakeupSource

user 通过上面的 bls_pm_setSuspendMask 设置 MCU 进入 suspend 或 deepsleep，通过下面的 API 设置 suspend 和 deepsleep 的唤醒源。

```
void        bls_pm_setWakeupSource(u8 source);
```

该API设置底层一个变量，称之为WakeupSource（代码中实际变量为bls_pm.wakeup_src）。

WakeupSource 可以选择三个唤醒源 PM_WAKEUP_PAD、PM_WAKEUP_CORE、PM_WAKEUP_TIMER，或者它们的“或”操作。

Lenze Technology Co.,LTD

MCU 在 Advertising state 或 Conn state Slave role 进入 suspend mode，系统实际的唤醒源为：

WakeupSource | PM_WAKEUP_TIMER

即 PM_WAKEUP_TIMER 是一定会有的，不依赖于 user 的设定，这是为了保证 MCU 一定要唤醒去继续处理 Adv Event 或 Brx Event。

每次调用 bls_pm_setWakeupSource 设置唤醒源后，一旦 MCU 进入 suspend 或 deepsleep 被唤醒后， WakeupSource 失效，在底层会被清 0。需要重新设置下一个 suspend 或 deepsleep 的唤醒源。

4.3.4 低功耗管理工作机制

为了让 user 更好的理解 SuspendMask 和 WakeupSource 的设置使用方法,这里对低功耗管理机制的工作原理做详细的说明。

SDK 中 mainloop 是一个 while(1)的结构：

```
while(1)
{
    blt_sdk_main_loop();
    //UI   task
}
```

只要 UI task 的时间不是特别长，blt_sdk_main_loop 就会一直被执行。BLE 低功耗管理机制的 code 是在 blt_sdk_main_loop 中执行的，所以低功耗管理的那一段 code 会一直被运行。

结合前面的 BLE Link Layer 工作时序的说明，定义两个时间参数：当前状态为 Advertising state 时，Link Layer Adv Event 开始的时间称为 T_advertising；当前状态为 Conn state Slave role 时，Link Layer Brx Event 开始的时间称为 T_brx。

blt_sdk_main_loop 中对低功耗管理的伪代码如下。

```
int   blt_sdk_main_loop (void)
{
    ......
    if(SuspendMask == SUSPEND_DISABLE)   // SUSPEND_DISABLE 不进低功耗
    {
        return   0;
    }

    if(Link Layer State 为 Advertising state 或 Conn state Slave role)
    {
        if(Link Layer 处于 Adv Event 或 Brx Event)  //BLE 收发包正在工作，不进低功耗
        {
            return   0;
        }
        else
        {
            blt_brx_sleep ();//suspend & wakeup 处理函数

        }
```


Lenze Technology Co., LTD

```
    }
    return 1;
}

void    blt_brx_sleep (void)
{
    if( (Link Layer state == Adv state)&& (SuspendMask&SUSPEND_ADV) )
    {   //当前广播状态，进 suspend
        执行 BLT_EV_FLAG_SUSPEND_ENTER 事件回调函数
        cpu_sleep_wakeup (0, PM_WAKEUP_TIMER | WakeupSource,
                T_advertising + advInterval); //suspend
        执行 BLT_EV_FLAG_SUSPEND_EXIT 事件回调函数

        if(当前 suspend 是被 GPIO CORE 提前唤醒)
        {
            执行 BLT_EV_FLAG_GPIO_EARLY_WAKEUP 事件回调函数
            再次进入 suspend，直到 T_advertising + advInterval 醒来
        }
    }
    else if((Link Layer state == Conn state Slave role)&&
    (SuspendMask&SUSPEND_CONN) )//当前 Conn state，进 suspend
    {
        u32 wakeup_tick;
        if(conn_latency 非 0) //conn_latency != 0
        {
            //latency_use 的计算在后面详细介绍，没有特殊设置时
            u16 latency_use = bls_calculateLatency();
            wakeup_tick = T_brx + (latency_use+1) * conn_interval;
        }
        else  //conn_latency == 0
        {
            wakeup_tick = T_brx + conn_interval;
        }

        执行 BLT_EV_FLAG_SUSPEND_ENTER 事件回调函数
        cpu_sleep_wakeup(0,PM_WAKEUP_TIMER|WakeupSource, wakeup_tick);
        执行 BLT_EV_FLAG_SUSPEND_EXIT 事件回调函数

        if(当前 suspend 是被 GPIO CORE 提前唤醒)
        {
            执行 BLT_EV_FLAG_GPIO_EARLY_WAKEUP 事件回调函数
            调整 BLE 时序相关处理
        }
    }
```


Lenze Technology Co.,LTD

```
    //清除跟 user 相关的一些低功耗配置参数
    WakeupSource= 0;//清除之前设置的唤醒源
    user_latecny = 0xffff;
}
```

4.4 Latency use 的配置与计算

由上面低功耗管理工作机制的介绍可知，在 Conn state Slave role 时，若设置了 suspendMask 为 SUSPEND_CONN，实际睡眠唤醒时间为

wakeup_tick = T_brx + (latency_use+1) * conn_interval;

T_brx 为当前 interval 上 Brx Event 收包时间点。

若 latency_use 为 0，在下一个 interval 上必须唤醒去听包；若 latency_use 非 0，则可以跳过 latency_use 个 interval，从而节省功耗。

latency_use 的值：

latency_use = bls_calculateLatency();

在 latency_use 计算中，涉及到一个 user_latency，这个是 user 可以设置的值，调用的 API 及其源码为：

```
void bls_pm_setManualLatency(u16 latency)
{
    user_latency = latency;
}
```

Latency_use 的计算过程如下。

首先计算 system latency：

1) 若当前连接参数中 connection latency 为 0，system latency 为 0。
2) 若当前连接参数中 connection latency 非 0：
   A. 若当前系统还有一些任务没有处理完，必须在下一个 interval 醒来收发包继续处理（比如还有数据没有发送完、收到 master 的数据还没处理完等等），system latency 为 0。
   B. 若当前系统已经没有任务需要处理了，则 system latency 等于 connection latency。但是有一个例外，如果收到了 master 的 update map request 或 update connection parameter request 且实际的更新时间点在(connection latency+1)个 interval 之前，则实际的 system latency 会强制 MCU 在实际更新时间点之前那个 interval 醒来，确保 BLE 时序的正确。

然后实际最终的 Latency_use = min(system latency, user_latency)，取 system latency 和 user_latency 中较小值。

那么 user 在 UI entry 部分调用 bls_pm_setManualLatency 手动设置的 latency 若比 system latency 小，则可以被作为最终的 Latency_use。只有当 system latency 非 0 时才有意义。

注意每次 blt_sdk_main_loop 执行的最后一句都会将 user_latency 置为 0xffff。所以 user 每次调用 bls_pm_setManualLatency 设置的 user latency 只对最近要进入的一次 suspend 有效。

4.5 其他 API

除了上面已经介绍过的 API，本节对其他的一些 API 进行详细说明。

Lenze Technology Co.,LTD

4.5.1 bls_pm_getSystemWakeupTick

下面的 API 用于获取低功耗管理计算的 suspend 醒来的时间点（system tick 值）。

u32 bls_pm_getSystemWakeupTick(void);

结合前面低功耗管理工作机制的说明，这个 API 只有在 BLT_EV_FLAG_SUSPEND_ENTER 事件回调函数里才能调用，其他地方不允许调用。

低功耗管理模块在执行 blt_brx_sleep 函数时，根据当前 Link layer 状态和应用层设置的 SuspendMask，才能计算出 suspend 醒来的时间点，此时应用层只有通过 BLT_EV_FLAG_SUSPEND_ENTER 回调函数才能读到这个值。比如在 Conn state 要进入 suspend 且 conn latency 不为 0 时：

```
u16 latency_use = bls_calculateLatency();
wakeup_tick = T_brx + (latency_use+1) * conn_interval;
cpu_sleep_wakeup(0,PM_WAKEUP_TIMER|WakeupSource, wakeup_tick);
```

应用层无法提前预知 bls_calculateLatency 计算的 latency_use 是多少，也就不知道实际的 wakeup_tick 大概在什么时间点，此时只能在 BLT_EV_FLAG_SUSPEND_ENTER 回调函数里调用 bls_pm_getSystemWakeupTick 来获取醒来的时间点。

下面以带按键扫描的应用为例，说明 BLT_EV_FLAG_SUSPEND_ENTER 事件回调函数和 bls_pm_getSystemWakeupTick 的用法。

```
bls_app_registerEventCallback(  BLT_EV_FLAG_SUSPEND_ENTER,
                                &ble_remote_set_sleep_wakeup);
```

```
voidble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( bls_ll_getCurrentState() == BLS_LINK_STATE_CONN &&
    ((u32)(bls_pm_getSystemWakeupTick() - clock_time())) >
              80 * CLOCK_SYS_CLOCK_1MS)
    {
        //gpio CORE wakeup suspend

        bls_pm_setWakeupSource(PM_WAKEUP_CORE);
    }
}
```

以上这个回调函数要实现的作用是防止按键丢失。一个正常的人为机械按键动作大概会持续几百毫秒，按的快的时候也会有一两百毫秒。当 user 通过 bls_pm_setSuspendMask 设置了 Advertising state 和 Conn state 都要进入 suspend，在 BLE 连接参数 Conn Latency 没有开启，为 0 时，只要 Adv interval 和 Conn interval 的值不是特别大（一般设置在 100ms 以内），suspend 的时间不会超过 Adv interval 和 Conn interval，能够确保按键扫描的频率，就不会丢键。此时不设置 GPIO 唤醒，不让按键动作唤醒 MCU。

但是当 latency 启用后（比如 conn_interval 为 10ms，latency 为 99），可能某次 Conn state 时的 suspend 会持续 1s。这个过程中按键可能会丢掉。在 BLT_EV_FLAG_SUSPEND_ENTER 回调里判断如果当前状态为 Conn state，并且当前要进入的 suspend 的唤醒时间点距离当前时间大于 80 ms，那么将 GPIO CORE 的唤醒添加进去，如果 timer 的唤醒时间点还没到，有按键按下导致 GPIO 上电平发生变化，触发 MCU 提前唤醒，去处理按键的扫描任务，确保按键不会丢失。

Lenze Technology Co.,LTD

4.5.2 bls_pm_enableAdvMcuStall

调用下面 API 用来降低广播时的电流峰值。

```
void        bls_pm_enableAdvMcuStall(u8 en);
```

en 为 1 时打开 MCU stall，en 为 0 时关闭 MCU stall。

注意 stack 底层使用了 Timer0 来实现广播时 MCU stall，所以如果加了这个功耗优化，应用层就不能使用 Timer0，可用 Timer1/Timer2 来替代。

4.6 GPIO 唤醒的注意事项

4.6.1 唤醒电平有效时无法进入 suspend/deepsleep

由于 17H25 的 CORE/PAD 唤醒是靠高低电平唤醒的，而不是上升沿下降沿唤醒，所以当配置了 GPIO CORE 或者 PAD 唤醒时，比如设置了某个 GPIO CORE 高电平唤醒 suspend，要确保 MCU 在调用 cpu_wakeup_sleep 进入 suspend 时，当前的这个 GPIO 读到的电平不能是高电平。若当前已经是高电平了，实际进入 cpu_wakeup_sleep 函数里面，触发 suspend 时是无效的，会立刻退出来，即完全没有进入 suspend。同样 GPIO PAD 唤醒也是如此。

如果出现以上情况，可能会造成意想不到的问题，比如本来想进入 deepsleep 后被唤醒，程序重新执行，结果 MCU 无法进入 deepsleep，导致 code 继续运行，不是我们预想的状态，整个程序的 flow 可能会乱掉。

针对以上问题，在 17H25 ble remote 代码中，做了相应的处理。

在 BLT_EV_FLAG_SUSPEND_ENTER 中设置了只有当 suspend 时间超过某个时间时，才会开启 GPIO CORE 唤醒。

```
void ble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( bls_ll_getCurrentState() == BLS_LINK_STATE_CONN &&
    ((u32)(bls_pm_getSystemWakeupTick() - clock_time())) >
                80 * CLOCK_SYS_CLOCK_1MS)
    {
        bls_pm_setWakeupSource(PM_WAKEUP_CORE);
    }
}
```

而当按键没有释放时，通过手动设置 latency 为 0 或者一个较小值使得 suspend 时间较短，可以保证 suspend 时间不会超过 80ms，那么就不会发生按键按着的时候（drive pin 上有高电平）开启了 GPIO CORE 高电平唤醒，如下代码所示。

Lenze Technology Co.,LTD

```
        user_task_flg = scan_pin_need || key_not_released || DEVICE_LED_BUSY;

        if(user_task_flg){
#if (LONG_PRESS_KEY_POWER_OPTIMIZE)
            extern int key_matrix_same_as_last_cnt;
            if(key_matrix_same_as_last_cnt > 5){
                bls_pm_setManualLatency( ui_manual_latency_when_key_press() );
            }
            else{
                bls_pm_setManualLatency(0);  //latency off: 0
            }
#else
            bls_pm_setManualLatency(0);
#endif
        }
```

进入 deepsleep 的两种情况，一是连续 60s 没有任何任务会进入 deepsleep，这里的任务包括按键被按下，所以此时不会有 drive pin 高电平导致 deepsleep 无法进入；二是卡键 60s 后进入 deepsleep，这时候虽然有 drive pin 上的高电平，但是我们会将卡键的 drive pin 唤醒电平极性取反，设为低电平唤醒，同样避免了这个问题（详情参考按键扫描章节的卡键处理）。

user 在使用 Lenze 的 GPIO CORE/PAD 唤醒时，要注意避免这个问题。

4.7 BLE 系统低功耗管理参考

根据以上 Suspend entry 的工作机制和各个接口函数的详细介绍，user 可以在 UI entry 部分比较灵活的配置自己的低功耗管理。下面给出目前 17H25 remote（语音遥控器）低功耗管理的参考 code，同时做一些解释。如下截图所示。

Lenze Technology Co.,LTD

```
void blt_pm_proc(void)
{
    if(ui_mic_enable){
        bls_pm_setSuspendMask (MCU_STALL);
    }
    else
    {
        bls_pm_setSuspendMask (SUSPEND_ADV | SUSPEND_CONN);

        user_task_flg = ota_is_working || scan_pin_need || key_not_released || DEVICE_LED_BUSY;
        if(user_task_flg){
            #if (LONG_PRESS_KEY_POWER_OPTIMIZE)
                extern int key_matrix_same_as_last_cnt;
                if(!ota_is_working && key_matrix_same_as_last_cnt > 5){  //key matrix stable can optize
                    bls_pm_setManualLatency(3);
                }
                else{
                    bls_pm_setManualLatency(0);  //latency off: 0
                }
            #else
                bls_pm_setManualLatency(0);
            #endif
        }

        if(sendTerminate_before_enterDeep == 1){ //sending Terminate and wait for ack before enter deepsleep
            if(user_task_flg){  //detect key Press again,  can not enter deep now
                sendTerminate_before_enterDeep = 0;
                bls_ll_setAdvEnable(1);   //enable adv again
            }
        }
        else if(sendTerminate_before_enterDeep == 2){  //Terminate OK
            analog_write(DEEP_ANA_REG0, CONN_DEEP_FLG);
            cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);  //deepsleep
        }

        //adv 60s, deepsleep
        if( blc_ll_getCurrentState() == BLS_LINK_STATE_ADV && !sendTerminate_before_enterDeep && \
            clock_time_exceed(advertise_begin_tick , ADV_IDLE_ENTER_DEEP_TIME * 1000000))
        {
            cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);  //deepsleep
        }
        //conn 60s no event(key/voice/led), enter deepsleep
        else if( device_in_connection_state && !user_task_flg && \
                clock_time_exceed(latest_user_event_tick, CONN_IDLE_ENTER_DEEP_TIME * 1000000) )
        {

            bls_ll_terminateConnection(HCI_ERR_REMOTE_USER_TERM_CONN); //push terminate cmd into ble TX buffer
            bls_ll_setAdvEnable(0);   //disable adv
            sendTerminate_before_enterDeep = 1;
        }
    }
}
```

图 4-3 17H25remote 低功耗管理参考 code 截图

在 main_loop 的 UI entry 部分加入 blt_pm_proc()函数，注意当 UI entry 有很多任务需要处理的时候，blt_pm_proc()要放在最后面，运行时间上最接近 blt_sdk_main_loop 的地方，因为这个函数里面会根据 UI entry 其他任务的处理情况来做相应设置。

以上低功耗管理几个要点总结如下：

1) 某些任务需要关闭 suspend 时，如语音(ui_mic_enable)、红外等任务运行时，设置 SuspendMask 为 SUSPEND_DISABLE。

2) 如果 slave 在 Advertising state，且连续广播时间达到 60s，设置当前 main_loop 进入 deepsleep，设置 deepsleep 的唤醒源为 GPIO PAD（需要提前设置按键可以唤醒）。判断是否广播超过 60s 的方法是用软件定时器，用变量 advertise_begin_tick 记录广播开始的 system tick。
设置连续 60s 无广播进入 deepsleep 的目的是为了省功耗，防止 slave 设备没有被 master 响应时还一直在广播。实际最后 user 根据自己的需求，对功耗进行评估后，决定如何处理广播态的时间问题。

3) 如果 slave 在连接状态、所有的按键都已经释放、没有音频任务、LED 任务等，超过最近一

次有效的任务时间 60s 以上，设置当前 main_loop 进入 deepsleep，设置唤醒源为 GPIO PAD(需要提前设置按键可以唤醒)，并且在 deepsleep 记忆寄存器 DEEP_ANA_REG0 标记当前是在连接状态下进入 deepsleep（deep 唤醒后，可以设置快速广播包尽快跟 master 连上）。

设置连续 60s 无有效任务进入 deepsleep 的目的也是为了省功耗。实际只要将维持连接的功耗调到很小，也可以不进入 deepsleep。最终需要 user 根据自己的需求和功耗状况决定如何实现。

连接状态进入 deepsleep 时，先调用 bls_ll_terminateConnection 向 master 发送一个 TERMINATE 命令，然后等到这个命令被 ack 或者超时了触发 BLT_EV_FLAG_TERMINATE 事件回调函数后再进入 deepsleep。

4) 当有一些任务不能被长睡眠（suspend 时间过长）破坏时，user 需要手动设置 latency 为 0。如上面的 key_not_released、DEVICE_LED_BUSY 时需要 latency 为 0（LONG_PRESS_KEY_POWER_OPTIMIZE 为 0）。

5) 在上面第 4 步的基础上，手动关闭 latency 后，每个 conn_interval 都要醒来，功耗会稍微有点高，且检测按键变化和 LED 任务的处理并不需要每个 conn_interval 都做一次，可以手动设置 latency 为其他值，继续进行功耗优化。LONG_PRESS_KEY_POWER_OPTIMIZE 为 1 时，当按键已经稳定后（key_matrix_same_as_last_cnt > 5），可以手动设置 latency 的值，设置 bls_pm_setManualLatency (4)后，suspend 会持续 5 个 conn_interval。 conn_interval 为 10 ms 时，每 10*(4+1) = 50 ms 醒来一次处理 LED 和按键变化的检测。实际 user 在使用这个优化时，需要自己根据 conn_interval 的值和任务响应时间，来评估任何在不影响功能的情况下，最大化的对功耗优化。

4.8 应用层定时唤醒

在 Advertising state 和 Conn state Slave role，一旦进入 suspend，只能被 stack 在特定的时间点唤醒，user 很难在某一个特定的时间点将 suspend 提前唤醒。为了增加 PM 的灵活性，SDK 增加了应用层定时唤醒的接口和它的回调函数。

应用层定时唤醒的 API：

```
void  bls_pm_setAppWakeupLowPower(u32 wakeup_tick, u8 enable);
```

wakeup_tick 为定时唤醒的绝对 system tick 时间点，enable 为 1 时打开该唤醒功能，enable 为 0 时关闭。

应用层定时唤醒发生时，执行 bls_pm_registerAppWakeupLowPowerCb 注册的回调函数，其原型和 API 如下：

```
typedef    void (*pm_appWakeupLowPower_callback_t)(int);
void       bls_pm_registerAppWakeupLowPowerCb(
                                pm_appWakeupLowPower_callback_t cb);
```

当 user 使用了 bls_pm_setAppWakeupLowPower 设置了应用层定时唤醒的 app wakeup_tick，SDK 底层在进入 suspend 前，会检查这个 app wakeup tick 是否在当前 suspend 的时间内，如果在这个时间之内，如下图所示，就会在 app wakeup_tick 触发 suspend 提前唤醒；如果 app wakeup tick 不在当前 suspend 时间内，底层不用管这个时间，还是按照 BLE 的时序去唤醒。

Lenze Technology Co.,LTD

图 4-4 app wakup tick 提前触发

5 OTA

17H25 不支持 Flash 多地址启动，只能从 0 地址启动，所以它的 OTA 的流程相同； Flash 为 512K，它的 FLash 架构设计如下所示：

为了实现 17H25 slave 的空中下载，我们需要一个设备作为 BLE OTA master。OTA master 可以是实际与 slave 配套使用的蓝牙设备（须在 APP 里实现 OTA），也可以使用 Lenze 的简易 BLE master dongle，下面以 Lenze 的 kma dongle 作为 ota master 示例来说明 17H25 实现 OTA 的方法。

5.1 17H25 Flash 架构设计和 OTA 流程

5.1.1 17H25 FLASH 存储架构

图 6-2 17H25 Flash 存储结构

Lenze Technology Co.,LTD

所有的 firmware 应不大于 128K，即 flash 的 0~0x20000 之间的区域可以存储。

Slave 端 0~0x20000 128K 区域存储当前老的 firmware；0x72000~0x72600 1.5K 区域存储 ota_boot.bin（该 bin 大小控制在 1.5K 以内）；0x20000~0x40000 128K 区域作为 new bin 存放区，存储从 ota_master 获得的 New_firmware.bin；0x73000 地址设置一个 byte 的 boot_flag，用作 boot 时检测的标志。

ota_master 端 0~0x20000 128K 区域存储 ota_master.bin；0x20000~0x40000 128K 区域作为 slave 新 firmware 的存放区域。

5.1.2 17H25 OTA 更新流程

根据 Slave 端和 ota_master 端 flash 架构，结合各个 bin 文件的实际功能，详细说明整个 OTA 实现的过程：

1) Slave 正常上电
   Slave 端 flash 上各个 bin 文件烧写完毕后，flash 0x73000 上的 boot_flag 值为 0xff。正常上电启动时，MCU 将 flash 0x00000 地址处的 Old_firmware.bin 中前一部分指令搬到 SRAM 从 0x808000 开始的地方，运行 Old_firmware.bin 中 cstartup.S 对应的启动代码，该启动代码对 boot_flag 的值做检测，发现该值不是 0xa5，直接运行正常的 Old_firmware.bin C 语言部分的代码，实现正常 Slave 功能。
2) ota_master 正常上电
   ota_master 上电和 Slave 上电顺序不固定，可先可后。两个设备都正常运行后，建立 BLE 连接，确保可以正常通信。
3) OTA 模式触发，两设备都进入 OTA 模式
   触发 ota_matser 进入 OTA 模式（可以是按键、PC 工具写内存等），Master 发送 OTA start 命令让 Slave 进入 OTA 模式。
4) ota_master 通过 RF 将 New_firmware.bin 空运给 Slave
   二者都进入 OTA 模式后，ota_master 发送带有 New_firmware 数据的 OTA data 包，Slave 收到包并解析，将数据烧写到 flash 的 0x20000~0x40000 新 firmware 存储区。
5) OTA data 发送完毕后，ota_master 发送 OTA end 命令，Slave reboot。
   当整个 OTA 过程顺利完成后，此时 New_firmware.bin 已经存储在 Slave 的 0x20000~0x40000。Slave 将 flash 0x73000 上的 boot_flag 的值设置为特定的 0xa5，然后 reboot MCU。
6) slave 运行 ota_boot.bin
   slave reboot 后，MCU 将 flash 0x00000 地址处的 Old_firmware.bin 中前一部分指令搬到 SRAM 从 0x808000 开始的地方，运行 Old_firmware.bin 中 cstartup.S 对应的启动代码，该启动代码对 flash 0x73000 上的 boot_flag 的值做检测，发现该值是 0xa5，这时候不再运行正常的 Old_firmware.bin 对应的代码，而是将 flash 0x72000~0x72600 区域 1.5K 的 ota_boot.bin 搬到 SRAM 0x808000~0x808600 的地方，搬移完成后，reset MCU（reset 只是让 MCU 从 SRAM 0x808000 地址开始运行，不会重新从 flash 搬代码到 SRAM 中）。此时 MCU 从 0x808000 处开始重新运行，相当于运行了 ota_boot.bin 的功能。
7) ota_boot 更新代码，reboot
   ota_boot.bin 运行后，从 flash 0x20000 开始的地方逐页读取 New_firmware.bin 的内容，并写到 flash 0x00000 开始的对应地址处，相当于将 New_firmware.bin 完全更新到 flash 0 地址处。更新完成后，将 flash 0x73000 上的 boot_flag 的值设定为 0x00，reboot MCU。
8) New_firmware.bin 正常运行
   MCU 再次 reboot 后，从 flash 0 地址搬代码到 SRAM 0x808000 开始的地方，并且检测到

Lenze Technology Co.,LTD

boot_flag 的值不是 0xa5，启动正常的 slave 功能，该 New_firmware.bin 类似于之前的 Old_firmware.bin，也具有 OTA 功能，可以再次启动 OTA 模式更新代码（最新的代码要重新下载到 ota_master 的 flash 0x20000 地址处）。

5.1.3 cstartup.S、reset、reboot、代码搬移等细节问题

5.1.3.1 启动代码 cstartup.S 对 boot_flag 检测的处理

四个 bin 文件中 ota_master.bin 运行在 ota_master 设备上，这个 bin 的启动文件没有特殊要求。Old_firmware.bin、New_firmware.bin 和 ota_boot.bin 都运行在 Slave 硬件上，这三个 bin 文件对应工程的 cstartup.S 必须一样，且 iCache 和 iTag 的具体位置也必须一致，实际上 New_firmware.bin 是 Old_firmware.bin 的更新程序，这两个 bin 文件的启动代码肯定是一样的，这里只介绍 Old_firmware.bin 和 ota_boot.bin 启动代码的区别：

1) Old_firmware 设置 iCache 和 iTag；ota_boot 运行的时候，MCU 并没有断电，可以沿用之前 Old_firmware 的设置，所以不用再设置。
2) ota_boot 不检测 boot_flag 的值，Old_firmware 检测 boot_flag 的值并处理：若不为 0xa5，正常运行，若为 0xa5，从 flash 0x72000 拷贝 1.5K 的 ota_boot.bin 到 SRAM 0x808000~0x808600，reset MCU。

根据以上这两点，cstartup.S 中从 SETIC 到 COPY_DATA 修改如下：

```
SETIC:
    tloadr      r1, DAT0 + 24
    tloadr      r0, DAT0 + 36              @ IC tag start
#ifdefMCU_CORE_17H25_OTA_BOOT
    tloadr      r0, DAT0 + 36 @这三句无意义，只是为了对齐
    tloadr      r0, DAT0 + 36
    tloadr      r0, DAT0 + 36
#else
    tstorerb    r0, [r1, #0]
    tadd        r0, #1                 @ IC tag end
    tstorerb    r0, [r1, #1]
#endif
    @tmov       r0, #0;
    @tstorerb r0, [r1, #2]

COPY_CODE_INIT:
     tmov       r3, #115        @ OTA FW ready flag at 0x73000;
    (0x73000 = #115<< 12)
     tshftl     r3, r3, #12     @ 0x73<<12 = 0x73000
     tloadr     r3, [r3, #0]    @flash 0x73000的值读到R3中
#ifdefMCU_CORE_17H25_OTA_BOOT
tcmp      r3, #0 @ota boot启动时R3跟0 比较（只是为了和下面的对齐）
#else
     tcmp       r3, #165 @应用程序启动时R3和0xa5比较
```


Lenze Technology Co.,LTD

```
#endif
    tjne     COPY_DATA_INIT @若上面的比较不相等，直接跳到COPY_DATA_INIT，
运行正常的程序启动，若相等，运行下面的语句，将flash 0x72000~0x72600的内容（即
ota_boot.bin）搬移到SRAM 0x8000~0x8600.
    tmov     r2, #114@ OTA boot code at: 0x72000
    tloadr       r3, COPY_CODE_DAT
    tloadr       r0, COPY_CODE_DAT + 4
    tshftl       r2, r2, #12      @ 0x72<<12
COPY_CODE_START:
    tloadm       r2!, {r1}
    tstorem  r3!, {r1}
    tcmp     r3, r0
    tjne     COPY_CODE_START
```

搬移完成后，将core_602寄存器写入0x88，reset MCU，程序重新运行，此时运行的是ota_boot.bin

```
    tloadr       r3, COPY_CODE_DAT + 8
    tmov     r2, #136     @0x88
    tstorerb r2, [r3, #0]

COPY_CODE_END:
    tj    COPY_CODE_END
   .balign   4
COPY_CODE_DAT:
   .word  (0x808000)
   .word  (0x808600)
   .word  (0x800602)

COPY_DATA_INIT:
   tloadr    r1, DATA_I
   tloadr    r2, DATA_I+4
   tloadr    r3, DATA_I+8
COPY_DATA:
```

5.1.3.2 Firmware size

ota_master 进入 OTA 模式后，从 flash 0x20000 开始读取 New_firmware.bin 时，涉及到该 bin 文件大小的问题，在 cstartup.S 0x18~0x1b 处存放生成 bin 文件 size 值：

```
cstartup.S:
  .org 0x18
  .word  (_bin_size_)
 boot.link:
PROVIDE(_bin_size_ = _code_size_ + _end_data_ - _start_data_);
```

这样设置的话，firmware 的 0x18 处 4 个字节就是 firmware 的大小，而 New_firmware.bin 存放在 0x20000 处，那么它的 size 就在 0x20018 的地方，ota_master 读 New_firmware.bin 的时候，

Lenze Technology Co.,LTD

首先读这个 size。

随便打开一个 17H25 ble SDK 编译出来的 bin 文件，如下所示，size 大小应该为 0x5570，实际 bin 文件内容范围为 0x0000～0x556f。

```
00000000 0e 80 f5 08 00 00 00 00 4b 4e 4c 54 20 01 88 00 .€
00000010 76 80 00 00 00 00 00 00 70 55 00 00 00 00 00 00 v€
```

图 6-4 firmware size 信息

```
00005540 00 00 00 00 00 00 00 00 00 00 00 00 02 12 00 00 ..
00005550 bf 01 fb 9d 4e f3 bc 36 d8 74 f5 39 41 38 68 4c ??
00005560 90 78 56 34 12 ef cd ab 74 24 00 00 00 00 00 00 .x
Hex Edit View        nb char : 21872        Ln : 8   Col : 13
```

图 6-5 firmware 结尾

5.1.3.3 reset 和 reboot

1) 将 MCU reset：向 core_0x602 写 0x88。此时 MCU 只是 PC 指针从 0 地址开始重新运行程序（运行的程序是之前留在 SRAM 上的内容）。
2) 将 MCU reboot：MCU 进入 deep，5 ms 后 timer 唤醒，唤醒后 MCU 会重新从 flash load 程序，然后 PC 指针从 0 开始运行程序。
3) slave 在 OTA 模式下通过 RF 获取并将 New_firmware.bin 存储到 flash 0x20000 地址处后，设置 boot_flag 为 0xa5，reboot MCU:

```
flash_erase_sector (0x73000);
u32 flag = 0xa5;
flash_write_page (0x73000, 4, &flag);   // set boot_flag 0xa5
start_reboot();

void start_reboot(void)
{
    irq_disable ();
    cpu_sleep_wakeup (1, PM_WAKEUP_TIMER, clock_time() +
            5*CLOCK_SYS_CLOCK_1MS);
}
```

4) ota_boot 将 0x20000 上的 New_firmware.bin 搬移到 0 地址后，清除 boot_flag，并 reboot MCU:

```
buff[0] = 0;
flash_write_page (0x73000, 1, buff);
REG_ADDR8(0x6f) = 0x20;//core_6f写0x20也是一种reboot，和
start_reboot()实现的效果是一样的，需要重新load flash。
while(1);
```

17H25 的 cstartup.S 中已经添加了上面介绍的所有内容，不需要再修改，17H25 ota_boot 可以使用 SDK 上 17H25_ota_boot 编译选项编出来的程序。

5.1.4 修改 FLash 存储架构

结合上面 FLash 架构和 OTA 更新流程来说明 Flash 架构的修改。

Lenze Technology Co.,LTD

SDK 默认的 Firmware size 为 128K，Firmware 运行地址为 0x00000~0x20000 128K 空间，OTA 程序地址为 0x20000~0x40000。 0x72000 sector 存放 17H25_ota_boot.bin，0x73000 sector 存放 ota boot flag。以上这些地址都是可配置修改的。

5.1.4.1 修改 Firmware size 和 OTA 程序存放地址

提供如下 API 用来修改 Firmware size 和 OTA 程序存放地址。

void bls_ota_setFirmwareSizeAndOffset(int firmware_size_k, u32 ota_offset);

API 声明在 proj_lib/ble/service/ble_ll_ota.h。

ota_offset 是 OTA 程序存放的地址。

firmware_size_k 为 user 最大的 Firmware size 值，单位为 K byte，一定要 4K byte 对齐，比如最大 size 可能是 57K，此时这个值一定要设为 60。这个 API 只能在 main.c 的第一句就调用，否则无效。

假设实际的 Firmware size 最大不会超过 64K，user 希望使用 0x00000~0x10000 64K 空间来存放 Firmware，而用 0x10000 ~0x20000 存放 OTA 程序，设置如下：

```
int main (void) {

    bls_ota_setFirmwareSizeAndOffset(64, 0x10000);
    cpu_wakeup_init();

    clock_init();
    set_tick_per_us(CLOCK_SYS_CLOCK_HZ/1000000);
```

按照这样设置的话，省下 0x20000 ~ 0x40000 128K 的空间可以用来做 user 的数据空间。

5.1.4.2 修改 OTA boot bin 存放地址

SDK 中默认将 17H25_ota_boot.bin 存放在 0x72000 这个 sector，大小为不超过 1.5K。如果需要修改这个地址，只需要修改 cstartup.S 即可。

假设需要将 17H25_ota_boot.bin 存放在 0x40000 这个 sector，在 proj/mcu_spec/cstartup.S 文件中找到下面这段 code，其中的 114 改为 64 即可（64=0x40，0x40<<12=0x40000）。

```
    tmov    r2, #114        @ OTA boot code at: 0x72000
    tloadr  r3, COPY_CODE_DAT
    tloadr  r0, COPY_CODE_DAT + 4
    tshftl  r2, r2, #12     @ 0x72<<12
```

5.1.4.3 修改 OTA boot flag 地址

SDK 中默认将 ota boot flag 存放在 0x73000。如果需要修改这个地址（假设修改 ota boot flag 地址为 0x41000），一共有三个地方要修改。

1) 在初始化的时候调用下面的 API：

   void bls_ota_setBootFlagAddr(u32 bootFlag_addr);

   API 声明在 proj_lib/ble/service/ble_ll_ota.h。

   bls_ota_setBootFlagAddr(0x41000);

2) 修改 proj/mcu_spec/cstartup.S：

   找到下面这段 code，将 115 改为 65（65=0x41, 0x41<<12=0x41000）。

Lenze Technology Co.,LTD

```
COPY_CODE_INIT:
    tmov        r3, #115            @ OTA FW ready flag at 0x73000; (0x73000 = #115 << 12)
    tshftl      r3, r3, #12     @ 0x73<<12
    tloadr      r3, [r3, #0]
#ifdef MCU_CORE_OTA_BOOT
    tcmp        r3, #0          @ 0x0
```

5.1.5 修改 FLash 存储架构

结合上面 FLash 架构和 OTA 更新流程来说明 Flash 架构的修改。

SDK 默认的 Firmware size 为 40K，Firmware 运行地址为 0x00000~0x0A000 40K 空间，OTA 程序地址为 0x10000~0x1A000 40K 空间，0x1A000 sector 存放 ota_boot.bin，0x1B000 sector 存放 ota boot flag，剩余的 24K 空间（0x0A000~0x10000）作为 user 的数据空间。

0x00000 ~0x1A000 这 104K 可用空间采用的 40+24+40 的结构设计的。如果需要修改这个架构设计，可用调用 SDK 提供的相关 API。

5.1.5.1 修改 Firmware size 和 OTA 程序存放地址

提供如下 API 用来修改 Firmware size 和 OTA 程序存放地址。

void bls_ota_setFirmwareSizeAndOffset(int firmware_size_k,
u32 ota_offset);

API 声明在 proj_lib/ble/service/ble_ll_ota.h。

ota_offset 是 OTA 程序存放的地址。

firmware_size_k 为 user 最大的 Firmware size 值，单位为 K byte，一定要 4K byte 对齐，比如最大 size 可能是 25K，此时这个值一定要设为 28。这个 API 只能在 main.c 的第一句就调用，否则无效。

假设实际的 Firmware size 最大不会超过 25K，user 希望使用 0x00000~0x07000 28K 空间来存放 Firmware，而用 0x07000 ~0x0E000 28K 存放 OTA 程序，将剩下的 0x0E000 ~0x1A000 共 48K 空间作为数据空间，即 28+28+48 的结构，设置如下：

```
int main (void) {

    blc_pm_select_internal_32k_crystal();

    bls_ota_setFirmwareSizeAndOffset(28, 0x7000);

    cpu_wakeup_init(CRYSTAL_TYPE);

    set_tick_per_us (CLOCK_SYS_CLOCK_HZ/1000000);
    clock_init();
```

5.1.5.2 修改 OTA boot flag 地址

SDK 中默认将 ota boot flag 存放在 0x1B000。如果需要修改这个地址（假设修改 ota boot flag 地址为 0x19000），一共有三个地方要修改。

1) 在初始化的时候调用下面的 API：

void bls_ota_setBootFlagAddr(u32 bootFlag_addr);

API 声明在 proj_lib/ble/service/ble_ll_ota.h。

bls_ota_setBootFlagAddr(0x19000);

2) 修改 proj/mcu_spec/cstartup .S：

Lenze Technology Co.,LTD

找到下面这段 code，将 27 改为 25（25=0x19, 0x19<<12=0x19000）。

```
COPY_CODE_INIT:
        tmov        r3, #27          @ OTA FW ready flag at 0x1B000;
        tshftl      r3, r3, #12      @ 0x1B<<12
        tloadr      r3, [r3, #0]
#ifdef MCU_CORE_8261_OTA_BOOT
        tcmp        r3, #0           @ 0x0
```

5.2 OTA 模式 RF 数据处理

5.2.1 Slave 端 Attribute Table 中 OTA 的处理

首先需要在 Attribute Table 所在的 app_att.c 中添加库里面 ota 的引用

#include "../../proj_lib/ble/service/ble_ll_ota.h"。

其次，在 Attribute Table 中添加 OTA 的相关内容，其中 OTA 数据 Attribute 的 att_readwrite_callback_t r 和 att_readwrite_callback_t w 分别设为 otaRead 和 otaWrite，将属性设为 Read 和 Write_without_Rsp（Master 通过 Write Command 发数据，不需要 slave 回 ack，速度会更快）。

static u8 my_OtaProp= CHAR_PROP_READ | CHAR_PROP_WRITE_WITHOUT_RSP;

{0,2,1,1,(u8*)(&my_characterUUID), (u8*)(&my_OtaProp), 0},
{0,2,1,1,(u8*)(&my_OtaUUID), (&my_OtaData), &otaWrite, &otaRead},
{0,2,sizeof (my_OtaName), sizeof (my_OtaName),(u8*)(&userdesc_UUID),
(u8*)(my_OtaName), 0},

master 向 slave 发送 OTA 数据时，实际是向上面第二个 Attribute 写数据，master 需要知道这个 Attribute 在整个 Attribute Table 中的 Attribute Handle。如果 user 使用 master 和 slave 事先约定好 Attribute Handle 值的方法，需要数出这个 Attribute Handle 值，然后直接在 master 端定义好。

5.2.2 OTA 数据 packet 格式

Master 端通过 L2CAP 层的 Write Command 向 slave 发命令和数据。

3.4.5.3 Write Command

The *Write Command* is used to request the server to write the value of an attribute, typically into a control-point attribute.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x52 = Write Command
Attribute Handle	2	The handle of the attribute to be set
Attribute Value	0 to (ATT_MTU-3)	The value of be written to the attribute

Lenze Technology Co.,LTD

图 6-8 BLE 协议栈 Write Command 格式

Attribute Handle 的值为 slave 端 OTA 数据的 handle_value。Attribute Value 长度设为 20，格式见下图。

OTA_cmd

0	1 … 19
OTA_CMD	invalid data

OTA_data

0	1	2 … 17	18	19
adr_index		firmware data:adr_index*16 - adr_index*16+15	CRC	

图 6-9 OTA 命令和数据的格式

前两个 byte 的范围为 0xff00 ~0xff10 时表示一个 OTA 的命令，命令类型由这两个 byte 的值决定：

1) 0xff00 为 OTA_FW_VERSION，要求获得 slave 当前 firmware 版本号。此命令作为一个预留的命令，user 可以选择使用，如果使用了这个命令，slave 端留出了相应的回调函数供 user 来完成 firmware 版本号的传送。
2) 0xff01 为 OTA_Start 命令，master 发这个命令给 slave，用来正式启动 OTA 更新。
3) 0xff02 为 OTA_end 命令，当 master 确定所有的 OTA 数据都被 slave 正确接收后，发送 OTA end 命令。为了让 slave 再次确定已经完全收到了 master 所有数据（double check，加一层保险），OTA end 命令后面带 4 个有效的 bytes，后面详细介绍。
4) 0xff03 ~0xff0f，待补充。

前两个 byte 的范围在 0~0x1000 之内表示一个 OTA 数据。由于 firmware size 不超过 128K（0x20000），OTA data packet 中每次传送 16 byte 的 firmware 数据，使用的 adr_index 为实际 firmware 地址除以 16 的值，adr_index=0 表示 OTA 数据是 firmware 地址 0x0~0xf 的值，adr_index=1，表示 OTA 数据是 firmware 地址 0x10~0x1f 的值。最后两个 byte 是将前 18 个 byte 进行一个 CRC_16 计算得到第一个 CRC 的值，slave 收到 OTA data 后，会对前 18 个 byte 进行同样的 CRC 计算，只有当计算结果与第 19、20 byte 的 CRC 吻合时，才认为这是一个有效的数据。

5.2.3 master 端 RF transform 处理方法

基于 BLE link layer RF 数据自动 ack 确保所有数据包不丢的前提， OTA 的数据 transform 不 check 每一个 OTA 数据是否被 ack，即 master 通过 write command 发一个 ota 数据后，不在软件上检查对方是否有 ack 信息回复，只要 master 端硬件 TX buffer 缓存的待发送数据未达到一定数量，直接将下一笔数据丢进 TX buffer。

ota master 软件上对 RF transform 的处理流程为：

1) 检测查询是否有触发进入 OTA 模式的行为，一旦检测到该行为，进入 OTA 模式。
2) master 向 slave 传送 OTA 命令和数据，需要知道 slave 端当前 OTA 数据的 Attribute 的 Attribute Handle 值，若 user 采用事先约定好的方式，直接定义该值；若没有事先约定好，采用 Read By Type Request 的方式获得这个 Attribute Handle 值。Lenze 所有 BLE SDK 的 OTA data 的 UUID 都是 16bytes，且永远都是下面这个值：

```
#define Lenze _SPP_DATA_OTA
```


Lenze Technology Co.,LTD

```
{0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,
0x03,0x02,0x01,0x00}      //!< Lenze _SPP data for ota
```

在 master 的 Read By Type Request 中将 Type 设置为这 16 个 bytes 的 UUID，slave 端回复的 Read By Type Rsp 中可以查到 OTA UUID 所在的这个 Attribute Handle，如下图所示，master 可以查到 Attribute Handle 的值为 0x0031。

图 6-10 master 通过 Read By Type Request 获取 OTA 的 Attribute Handle

3) 获取 slave 当前 firmware 版本号，决定是否要继续做 OTA 更新（若版本已经最新，不需要更新）。这一步为 user 选择性是否要做。17H25 BLE SDK 不提供具体的版本号获取办法，user 可以自行发挥。

   虽然这里预留了 OTA version 命令，目前的 17H25 BLE SDK 并没有实现版本号的传送。user 可以使用 write cmd 的形式通过 OTA version cmd 向 slave 传送一个获取 OTA version 的请求，但是 slave 那端在收到 OTA version 请求的时候只提供一个回调函数，user 自己在回调函数里想办法将 slave 端的版本号传送给 master（如手动送一个 NOTIFY/INDICATE 的数据）。

4) 启动 OTA 开始的一个计时，后面要不断检测该计时是否超过 15 秒（这只是个参考时间，实际根据 user 测试的正常 OTA 需要多少时间后再做评估），如果超过 15 秒认为 OTA 超时失败，因为 slave 端收到 OTA 数据后会 check CRC，一旦 CRC 错误或者出现其他错误（如烧写 flash 错误），就认为 OTA 失败，直接程序重启，此时 link layer 无法 ack master，master 端的数据一直发不出去导致超时。

5) 读取 Master flash 0x20018~0x2001b 四个字节，确定 firmware 的 size。这个 size 是由我们的编译器实现的，假设 firmware 的 size 为 20k = 0x5000，那么 firmware 的 0x18~0x1b 的值为 0x00005000，所以在 20018~0x2001b 可以读到 firmware 的大小。如下图所示的 ble_remote.bin，0x18 ~0x1b 内容为 0x00005a98，所以大小为 0x5a98 = 23192 bytes，从 0x0000 到 0x5a97。

8267_remote.bin

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000	0e	80	01	03	00	00	00	00	4b	4e	4c	54	80	01	88	00
00000010	5e	80	00	00	00	00	00	00	98	5a	00	00	00	00	00	00
00000020	25	08	26	09	26	0a	91	02	02	ca	08	50	04	b1	fa	87
00000030	14	08	c0	6b	15	08	85	06	13	08	c0	6b	14	08	85	06

图 6-11 firmware 示例--开头部分

00005a40	02	03	04	05	00	01	02	03	04	05	00	00	e1	77	ad	92
00005a50	24	ab	dc	ba	13	02	f1	e0	df	ce	bd	ac	02	01	00	00
00005a60	04	01	00	00	08	01	00	00	40	01	00	00	10	03	00	00
00005a70	20	03	00	00	40	03	00	00	80	03	00	00	01	04	00	00
00005a80	02	04	00	00	5c	58	00	00	2c	58	00	00	44	58	00	00
00005a90	44	58	00	00	01	00	00	00								

图 6-12 firmware 示例--结尾部分

Lenze Technology Co.,LTD

6) 向 slave 发一个 OTA start 命令 0xff01，通知 slave 进入 OTA 模式，等待 master 端的 OTA 数据，如下图所示。

Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
L2CAP-S	2	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF	0x61875B	0	OK

图 6- 13 master 发 OTA start

7) 从 Master flash 0x20000 区域开始每次读 16 个 byte 的 firmware，填入 OTA data packet，设置对应的 adr_index，并计算 CRC 值，将 packet push 到 TX fifo，一直到 firmware size 最后一个 16 byte 为止，将 firmware 所有的数据全部发送给 slave。数据发送方法如前面介绍，使用 OTA data 的格式，有效数据为 20 bytes，前两个 bytes 放 adr_index，紧跟 16 个有效的 firmware 数据，最后两个是前 18 个数据的 CRC 计算值。

注意，如果 firmware 最后一笔数据不是 16 字节对齐，需要将剩余的部分按 0xff 补对齐，计算 CRC 的时候需要将补充的数据计算进去。

结合上图所示的 ble_remote.bin 来详细介绍 OTA 数据如何拼装。

第一笔数据：adr_index 为 0x00 00，16 个数据为 0x0000 ~0x000f 地址的值，然后这 18 个数据计算 CRC，假设 CRC 结果为 0xXYZW，那么 20bytes 排列为:

0x00 0x00 0x0e 0x80 ....省略 12 个 bytes..... 0x88 0x00 0xZW 0xXY

第二笔数据：

0x01 0x00 0x5e 0x80 ....省略 12 个 bytes..... 0x00 0x00 0xJK 0xHI

第三笔数据：

0x02 0x00 0x25 0x08 ....省略 12 个 bytes..... 0xfa 0x87 0xNO 0xLM

.....

倒数第二笔数据：

0xa8 0x05 0x02 0x04 ....省略 12 个 bytes..... 0x00 0x00 0xST 0xPQ

最后一笔数据：

0xa9 0x05 0x44 0x58 0x00 0x00 0x01 0x00 0x00 0x00

0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xWX 0xUV

8 个 0xff 为补齐的数据。

0xa9 ~0xff 共 18 个 bytes 的 CRC 计算结果为 0xUVWX。

上面的数据如下图所示：

Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
L2CAP-S	2	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF	0x61875B	0	OK
Empty PDU	1	1	0	0	0						0x8FE27A	0	OK
L2CAP-S	2	1	1	1	27	0x0017	0x0004	0x52	0x0031	00 00 0E 80 01 03 00 00 00 00 4B 4E 4C 54 80 01 88 00 BA A5			
Empty PDU	1	0	1	0	0						0x8FE90F	0	OK
L2CAP-S	2	0	0	1	27	0x0017	0x0004	0x52	0x0031	01 00 5E 80 00 00 00 00 00 00 98 5A 00 00 00 00 00 00 EA EF			
Empty PDU	1	1	0	0	0						0x8FE27A	0	OK
L2CAP-S	2	1	1	0	27	0x0017	0x0004	0x52	0x0031	02 00 25 08 26 09 26 0A 91 02 02 CA 08 50 04 B1 FA 87 A7 0D			

Lenze Technology Co.,LTD

| Data Type | Data Header | | | | | L2CAP Header | | ATT_Write_Command | | | | |

user 需要注册这个回调，以便在 OTA start 的时候做一些操作，比如配置 LED 灯的特殊闪烁方式来指示 OTA 正在进行。另外 slave 这端一旦收到 OTA start 开始 OTA 后，也会启动一个计时，目前 SDK 中默认是 30s，如果 30s 之内 OTA 还没有完成，就认为超时失败，实际 user 最后需要根据自己的 firmware 大小（越大越耗时）和 master 端 BLE 数据带宽（太窄的话会影响 OTA 速度）来修改这个默认的 30s，SDK 提供修改的接口为：

```
void bls_ota_setTimeout(u32 timeout_us);//单位: us
```

3) 收到有效的 OTA 数据（前两 bytes 为 0~0x1000）：

这个范围的值表示具体的 OTA data。每次 slave 收到一个 20 byte 的 OTA data packet，先看 adr_index 是否等于 slave_adr_index 的值加 1，若不等，说明丢包，OTA 失败，若相等，更新 slave_adr_index 的值。然后对前 18 byte 的内容进行 CRC_16 的 check，若不匹配，OTA 失败，若匹配，则将 16 byte 的有效数据写到 flash 对应位置 ota_program_offset+adr_index*16 ~ ota_program_offset+adr_index*16 + 15。在写 flash 的过程中，如果出错 OTA 也失败。

4) 收到 OTA end（前两 bytes 为 0xff02）：

检查 OTA end 包中的 adr_max 和其取反校验值是否正确，若正确，则 adr_max 可以用来做 double check。double check 的时候，判断 slave 之前收到的 master 的数据 index 最大值与该包中的 adr_max 是否相等，若相等，认为 OTA 成功，若不等，认为丢掉了最后一笔或几笔数据，OTA 不完整。

当 OTA 成功的时候，slave 将老的 firmware 所在地址的 flash 启动标志设为 0，将新的 firmware 所在地址的 flash 启动标志设为 0x4b，将 MCU reboot。

5) slave 提供 OTA 状态的回调函数：

slave 端一旦启动 OTA，不管 OTA 是成功还是失败，最会将 MCU reboot：若成功，会在 reboot 前设置 flag 告诉 MCU 再次启动后执行新的程序；若 OTA 失败，会将错误的新程序擦掉后重新启动，还会执行老的程序。在 MCU reboot 前，根据 user 是否注册了 OTA 状态回调函数，来决定是否触发该函数。以下是相关 code：

```
typedef void (*ota_resIndicateCb_t)(int result);

enum{
    OTA_SUCCESS = 0,      //success
    OTA_PACKET_LOSS,      //lost one or more OTA PDU
    OTA_DATA_CRC_ERR,      //data CRC err
    OTA_WRITE_FLASH_ERR,  //write OTA data to flash ERR
    OTA_DATA_UNCOMPLETE,  //lost last one or more OTA PDU
    OTA_TIMEOUT,       //
};
    void bls_ota_registerResultIndicateCb
                                (ota_resIndicateCb_t cb);
```

设置了回调函数后，回调函数的参数 result 的 6 个值如上面 enum 描述，第一个是 OTA 成功，剩下的 5 个是 5 种不同原因的 OTA 失败。

由于回调函数执行完后，一定会触发 MCU reboot，实际代码中看到这个状态指示的结果时，并没有太多功能性的用途。user 主要是用来 debug，在 OTA 不成功时，可以读到上面的 result

Lenze Technology Co.,LTD

后，将 MCU 用 while(1)停住，来了解当前是何种原因导致的 OTA 失败。

另外可以在 OTA 成功时，加一个 LED 效果来指示当前 OTA 成功，如下所示：

```
void LED_show_ota_result(int result)
{
    irq_disable();
    WATCHDOG_DISABLE;

    gpio_set_output_en(GPIO_LED, 1);

    if(result == OTA_SUCCESS){  //OTA success
        gpio_write(GPIO_LED, 1);
        sleep_us(2000000);  //led on for 2s
        gpio_write(GPIO_LED, 0);
    }
    else{  //OTA fail

    }

    gpio_set_output_en(GPIO_LED, 0);
}

bls_ota_registerResultIndicateCb (LED_show_ota_result);
```

slave 端 otaWrite 函数被封装在 lib 中，其他相关接口见 SDK 中 proj_lib/ble/service/ble_ll_ota.h。

6 按键扫描

Lenze 提供了一套基于行列式扫描的 keyscan 架构，用于按键扫描处理，user 可以直接使用这部分的 code，也可以自己去实现。

6.1 Key matrix

以 Lenze 的 demo 板为例，这是一个 4*6 的矩阵，最多支持 24 个按键，如下图所示。Row0 - Row3 是 4 个 drive pin，用来输出驱动电平，CoL0 - CoL5 是 6 个 scan pin，用来扫描当前列上是否有按键被按下。

Lenze Technology Co.,LTD

图 6-1 行列式键盘结构

结合上图，对 app_config.h 中 keyscan 相关的配置进行详细说明如下。

根据实际的硬件电路，demo 板上 Row0 - Row3 为 PB1、PB2、PB3、PB6。CoL0-CoL5 为 PD4、PD5、PD6、PD7、PE0、PE1。

定义 drive pin 数组和 scan pin 数组：

```
#define  KB_DRIVE_PINS  {GPIO_PB1, GPIO_PB2, GPIO_PB3, GPIO_PB6}
#define  KB_SCAN_PINS   {GPIO_PD4, GPIO_PD5, GPIO_PD6, GPIO_PD7,
                         GPIO_PE0, GPIO_PE1}
```

keyscan 使用的上下拉电阻都使用 17H25IC 中的模拟电阻，drive pin 选取下拉 100K，scan pin 选取上拉 10K。那么当没有按键按下时，scan pin 作为输入 GPIO 读到的是被 10K 上拉的高电平。当扫描开始时，在 drive pin 上输出低电平，scan pin 读到低电平就表示当前列上有按键按下（注意此时 drive pin 不是 float 态，若 output 没打开，scan pin 读到的是 100K 和 10K 的分压电平，还是高）。

定义行列式扫描中，drive pin 输出低电平时 scan pin 扫描到的有效电平。

```
#define    KB_LINE_HIGH_VALID              0
```

定义 Row 和 COL 的上下拉：

```
#define    MATRIX_ROW_PULL                 PM_PIN_PULLDOWN_100K
#define    MATRIX_COL_PULL                 PM_PIN_PULLUP_10K

#define    PULL_WAKEUP_SRC_PB1      MATRIX_ROW_PULL
#define    PULL_WAKEUP_SRC_PB2      MATRIX_ROW_PULL
```


Lenze Technology Co.,LTD

```
#define    PULL_WAKEUP_SRC_PB3      MATRIX_ROW_PULL
#define    PULL_WAKEUP_SRC_PB6      MATRIX_ROW_PULL

#define    PULL_WAKEUP_SRC_PD4      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PD5      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PD6      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PD7      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PE0      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PE1      MATRIX_COL_PULL
```

由于在 gpio_init 时将 ie 的状态会默认设为 0，scan pin 需要读电平，打开 ie：

```
#define PD4_INPUT_ENABLE    1
#define PD5_INPUT_ENABLE    1
#define PD6_INPUT_ENABLE    1
#define PD7_INPUT_ENABLE    1
#define PE0_INPUT_ENABLE    1
#define PE1_INPUT_ENABLE    1
```

当 MCU 进入 suspend 或 deepsleep 时，需要设置 CORE/PAD GPIO 唤醒。设置 drive pin 高电平唤醒，按下按键时，drive pin 读到 100K 和 10K 的分压电平，为 10/11 VCC 的高电平。需要打开 drive pin 的 ie 读取其电平状态：

```
#define PB1_INPUT_ENABLE    1
#define PB2_INPUT_ENABLE    1
#define PB3_INPUT_ENABLE    1
#define PB6_INPUT_ENABLE    1
```

6.2 Keyscan、keymap 和 keycode

6.2.1 Keyscan

按照上面的配置完成后，在 mainloop 中调用下面函数完成 keyscan。

```
u32 kb_scan_key (int numlock_status, int read_key)
```

第一个参数 numlock_status 在 mainloop 中调用时设为 0 即可，只有在 deepsleep 醒来的快速扫描按键时才会将其设为 KB_NUMLOCK_STATUS_POWERON，后面的快速扫键中介绍（对应 DEEPBACK_FAST_KEYSCAN_ENABLE）。

第二个参数 read_key 是 keyscan 函数按键的缓存处理，这个一般用不到，一直设为 1 即可（为 0 时会将按键值缓存在 buffer 里，不报告给上层）。

返回值用于通知 user 当前的按键扫描是否发现矩阵键盘有变化：有变化时，返回 1；无变化时返回 0。

kb_scan_key 这个函数是在 mainloop 中调用的，根据 BLE 时序可知，mainloop 的运行时间为 adv_interval 或 conn_interval。广播状态时(假设 adv_interval 为 30ms)，每 30ms 做一次 key scan；连接状态时（假设 conn_interval = 10ms），每 10ms 做一次 key scan。

理论上，当前 key scan 发现矩阵上按键的状态和上次 key scan 的状态不一样时，就认为有变化。实际代码中开启了一个防抖动滤波处理，只有发现连续两次 key scan 的按键状态一样，

Lenze Technology Co.,LTD

且和上一次存储的最新矩阵按键状态不一样时，才认为是一个有效的按键变化，这时返回 1 表示按键有变化，并将矩阵按键的状态通过 kb_event 结构体反映出来，同时将更新当前的按键状态为最新的矩阵按键状态。这部分对应的代码为 keyboard.c 中的：

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
```

上面所说的最新按键状态指的是矩阵上 24 个按键的按下或松开的状态的集合。上电时，默认的第一次矩阵按键状态为 24 个按键都是松开的。只要经过防抖动滤波处理后的矩阵按键的状态发生任何变化，返回值都会为 1，否则返回 0。如：按下一个按键返回一个变化；松开一个按键返回一个变化；按下一个键时再按下第二个键返回一个变化；按下两个键时再按下第三个键返回一个变化；按下两个键时松开其中一个键返回一个变化……

6.2.2 Keymap &kb_event

user 在调用 kb_scan_key 看到一个按键变化时，通过一个全局的结构体变量 kb_event 来获取当前的按键状态。

```
#define KB_RETURN_KEY_MAX    6

typedef struct{
    u8 cnt;
    u8 ctrl_key;
    u8 keycode[KB_RETURN_KEY_MAX];
}kb_data_t;
kb_data_t  kb_event;
```

kb_event 由 8 个 byte 构成，第一个 cnt 用于指示当前有几个有效的按键被按下，第二个 ctrl_key 一般不会用到，只有在做标准的 USB HID keyboard 时才会用到（keymap 中的 keycode 设为 0xe0-0xe7 时会触发，所以 user 千万不要设这 8 个值）。

keycode[6]用于最多存储当前 6 个被按下按键的 keycode（如果实际按下的键超过 6 个，只有前 6 个能反应出来）。

24 个按键对应的 keycode 在 app_config.h 中定义：

```
#define        KB_MAP_NORMAL {\
            {CR_VOL_MUTE, VK_3,      VK_1,      VK_MEDIA,  }, \
            {VK_2,        VK_5,      VK_M,      VK_4,      }, \
            {CR_RIGHT,    VK_NONE,   CR_SEL,        CR_LEFT,   }, \
            {CR_REWIND,       VK_NONE,   CR_DN,     CR_HOME,   }, \
            {CR_VOL_UP,       VK_NONE,   VK_MMODE,  CR_VOL_DN, }, \
            {VK_WEB,      VK_NONE,   CR_UP,     CR_POWER,  }, }
```

这个 keymap 的格式和 4*6 矩阵结构一致，可以对应设置按键按下后的 keycode，如按下 Row0 和 CoL0 两条线交叉的按键，出来的 keycode 为 CR_VOL_MUTE。

在 kb_scan_key 函数内部，每次扫描前会将 kb_event.cnt 清 0，而 kb_event.keycode[]这个数组是不清除的。所以每次返回 1 表示有变化时，用 kb_event.cnt 判断当前矩阵按键上有几个有效的按键。

Lenze Technology Co.,LTD

1) kb_event.cnt = 0 时，上一次有效矩阵状态 kb_event.cnt 肯定是不等于 0 的，但不确定是 1、2 还是 3，这个变化一定是按键释放，不确定是一个键释放还是同时好几个键释放。
此时 kb_event.keycode[]里面即使有数据，也是无效的，忽略不看。

2) kb_event.cnt = 1，可能上次 kb_event.cnt =0，那么按键变化是一个键被按下；可能上次 kb_event.cnt = 2，那么按键变化是两个键中一个被释放；也还有其他可能性，如三个键中两个被同时释放。
此时 kb_event.keycode[0]表示当前被按下的这个键的键值，后面的 keycode 忽略不看。

3) kb_event.cnt = 2，可能上次 kb_event.cnt = 0，变化是两个键同时按下；可能上次 kb_event.cnt = 1，一个键被按下时另一个键被按下；可能上次 kb_event.cnt = 3，三个键被按下时，其中一个被释放；其他可能性等等。
此时 kb_event.keycode[0]和 kb_event.keycode[1]表示当前被按下的两个键的键值，后面的 keycode 忽略不看。

user 可以每次在 key scan 前自己将 kb_event.keycode 清 0，这时就可以根据 kb_event.keycode 来判断是否有按键变化发生，如下所示。这个示例只是简单的处理单个按键按下的情况，所以当 kb_event.keycode[0]非 0 时，就认为是一个按键被按下，并不去判断是否两个键同时按下或者两个键中的一个释放等复杂的情况。

```
kb_event.keycode[0] = 0;//手动清除keycode[0]
int det_key = kb_scan_key (0, 1);

if (det_key)
{
    key_not_released = 1;

    u8 key = kb_event.keycode[0];
    if (key)          //key press
    {
        key_buf[2] = key;
        //send key press
        blt_push_notify_data (HID_HANDLE_KEYBOARD_REPORT, key_buf,
                                                          8);
    }
    else
    {
        key_not_released = 0;
        key_buf[2] = 0;
        //send key release
        blt_push_notify_data (HID_HANDLE_KEYBOARD_REPORT, key_buf,
                                                          8);
    }
}
```


Lenze Technology Co.,LTD

6.3 Keycode

上面介绍了在 app_config.h 定义 keymap，将 keycode 填入 KB_MAP_NORMAL。由于这个 key scan 是实现标准的 USB HID keyboard 的软件处理，有一些特殊的 keycode 需要处理，所以 user 在定义 keycode 时要注意一些细节。

keyboard.c中的kb_remap_key_row函数用于keycode的处理。

```
void kb_remap_key_row(int drv_ind, u32 m, int key_max, kb_data_t
                                                        *kb_data)
```

```
static inline void kb_remap_key_row(int drv_ind, u32 m, int key_max, kb_data_t *kb_data){
    foreach_arr(i, scan_pins){
        if(m & 0x01){
            u8 kc = kb_k_mp[i][drv_ind];
#if(KB_HAS_CTRL_KEYS)

            if(kc >= VK_CTRL && kc <= VK_RWIN)
                kb_data->ctrl_key |= BIT(kc - VK_CTRL);
            //else if(kc == VK_MEDIA_END)
                //lock_button_pressed = 1;
            else if(VK_ZOOM_IN == kc || VK_ZOOM_OUT == kc){
                kb_data->ctrl_key |= VK_MSK_LCTRL;
                kb_data->keycode[kb_data->cnt++] = (VK_ZOOM_IN == kc)? VK_EQUAL : VK_MINUS,
            }
            else if(kc != VK_FN)//fix fn ghost bug
                kb_data->keycode[kb_data->cnt++] = kc;

#else
            kb_data->keycode[kb_data->cnt++] = kc;
#endif
            if(kb_data->cnt >= key_max){
                break;
            }
        }
        m = m >> 1;
        if(!m){
            break;
        }
    }
}
```

图 6-2 keycode 处理函数

前面介绍的 kb_event.ctrl_key 会获得对应的 CTRL KEY，其 keycode 范围为 0xe0 - 0xe7，user 不能使用。

proj/drivers/usbkeycode.h 中：

```
#define    VK_CTRL        0xe0
#define    VK_SHIFT       0xe1
#define    VK_ALT         0xe2
#define    VK_WIN         0xe3
#define    VK_RCTRL       0xe4
#define    VK_RSHIFT      0xe5
#define    VK_RALT        0xe6
#define    VK_RWIN        0xe7
```

下面所示的键值，slave 传送到 Lenze 的 master dongle 后会做特殊的转化处理，通过 USB 给上位机实现相应功能，这个需要依赖于 master dongle 端的 USB 报表描述符的处理。

所以 user 需要注意，这些键值直接传送给 ble master 时，并不一定有相应的功能，需要依赖

于 app_att.c 中的 ble HID 报表描述符的配置。

```
enum{
VK_EXT_START =          0xa0,

VK_SYS_START =          VK_EXT_START, //0xa0
VK_SLEEP =              VK_SYS_START, //0xa0,    sleep
VK_POWER,                             //0xa1,    power
VK_WAKEUP,                            //0xa2, wake-up
VK_SYS_END,                               //0xa3
    VK_SYS_CNT =        (VK_SYS_END - VK_SYS_START),//0xa3-0xa0=0x03

VK_MEDIA_START =     VK_SYS_END,          //0xa3
VK_W_SRCH =          VK_MEDIA_START,      //0xa3
VK_WEB,                                   //0xa4
VK_W_BACK,
VK_W_FORWRD,
VK_W_STOP,
VK_W_REFRESH,
VK_W_FAV,                             //0xa9
VK_MEDIA,
VK_MAIL,
VK_CAL,
VK_MY_COMP,
VK_NEXT_TRK,
VK_PREV_TRK,
VK_STOP,       //b0
VK_PLAY_PAUSE,
VK_W_MUTE,
VK_VOL_UP,
VK_VOL_DN,

VK_MEDIA_END,
VK_EXT_END =      VK_MEDIA_END,
    VK_MEDIA_CNT =    (VK_MEDIA_END - VK_MEDIA_START),//0xb5-0xa3=0x12

    VK_ZOOM_IN =        (VK_MEDIA_END + 1),//0xb6
    VK_ZOOM_OUT ,                         //0xb7
}
```

6.4 Keyscan flow

6.4.1 基本的 keyscan flow

调用 kb_scan_key 时，一个最基本的 keyscan 的 flow 如下：

Lenze Technology Co.,LTD

1) 第一次全矩阵通扫。

将 drive pin 全部输出 drive 电平（0），同时读取所有的 scan pin，检查是否能读到有效的电平，并记录哪一列上读到了有效电平（用 scan_pin_need 用于标记有效的列号）。

若不使用第一次全矩阵通扫，直接逐行扫的话，至少要进行所有行的（当前 demo 程序 4 行）扫描，即使没有按键按下也要每次都逐行扫描，比较耗时间。加入了第一次全矩阵通扫后，若没发现任何列上有按键按下，就可以直接退出 keyscan，在没有按键按下时会节省很多时间。

第一次全矩阵通扫的 code 对应如下：

```
scan_pin_need = kb_key_pressed (gpio);
```

在 kb_key_pressed 函数中将所有的行输出低电平，延时 20us 后（延时是为了等待电平稳定后才读 scan pin）。设置了一个 release_cnt 为 6，当检测到矩阵上的按键按下并全部释放后，并不是立刻就认为没有按键而不去逐行扫描了，而是最终缓冲 6 帧，直到发现连续 6 次都是检测到按键全部释放后不再去逐行扫描。实现了一个 key debouce 防抖动的处理。

2) 根据全矩阵通扫的结果，逐行扫描。

全矩阵通扫发现有按键按下时，开始逐行扫描，从 ROW0 - ROW3 逐行输出有效 drive 电平，读取列上的电平值，找出按键按下的位置。

对应的代码为：

```
u32 pressed_matrix[ARRAY_SIZE(drive_pins)] = {0};

kb_scan_row (0, gpio);
for (int i=0; i<=ARRAY_SIZE(drive_pins); i++) {
    u32 r = kb_scan_row (i < ARRAY_SIZE(drive_pins) ? i : 0, gpio);
    if (i) {
        pressed_matrix[i - 1] = r;
    }
}
```

在做逐行扫描时使用了一些方法来优化代码执行时间：

一是当某行 drive 时，并不需要读取全部的列 CoL0 - CoL5，根据之前通扫的 scan_pin_need 可以知道哪些列上能够读到有效电平，此时只读取已经被标记的列即可。

二是每一行 drive 时，需要 20 us 左右的等待稳定时间，做了一个缓冲处理，把 20us 的等待时间转化到执行 code 中，节省了这个时间。具体怎么实现不介绍，请 user 自行理解。

最终的矩阵按键状态使用 u32 pressed_matrix[4]（可以看出最多支持 32 列）来存储，pressed_matrix[0] 的 bit0 - bit5 是否为 1 标记 Row0 上 CoL0 -CoL5 是否有按键 ..... pressed_matrix[3]的 bit0 - bit5 是否为 1 标记 Row3 上 CoL0 -CoL5 是否有按键。

3) 对 pressed_matrix[]进行防抖动滤波处理

对应代码为：

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );

u32 key_changed = key_debounce_filter( pressed_matrix, \
            (numlock_status & KB_NUMLOCK_STATUS_POWERON) ? 0 : 1);
```

当 deepsleep 醒来后快速按键检测时，numlock_status = KB_NUMLOCK_STATUS_POWERON，

Lenze Technology Co.,LTD

此时 filt_en = 0，不进行滤波，是为了最快速的获取键值。其他情况下，filt_en = 1，需要滤波处理。滤波处理的思路是：最近的连续两次 pressed_matrix[]一致，且和上一次有效的 pressed_matrix[]不一样，才认为是按键矩阵发生了有效的变化，key_changed = 1。

4) 对 pressed_matrix[]进行缓存处理
将 pressed_matrix[]存入到缓冲区，当 **kb_scan_key** (int numlock_status, int read_key) 中的 read_key 为 1 时，立刻读出缓冲区的数据，当 read_key 为 0 时，缓冲区数据保存起来，不通知上层，只有等到 read_key 为 1 时，才能读出之前缓存的数据。
由于我们的 read_key 永远为 1，这部分可以忽略不计，相当于缓冲区没有起到作用。具体代码不介绍。

5) 根据 pressed_matrix[]，查表 KB_MAP_NORMAL，返回键值。
对应的函数为 kb_remap_key_code 和 kb_remap_key_row，不具体介绍，user 自行理解。

6.4.2 Keyscan flow 时序优化

根据上面的最基本的 keyscan 的 flow 可以知道，即使没有任何键按下，每个 mainloop 都必须至少会执行第一次全矩阵通扫，执行时间大概会有接近 100 us。

使用 GPIO 中断状态位的查询可以优化掉没有按键按下时的全矩阵通扫的时间。

由前面的低功耗管理部分可以知道，drive pin 的所有 GPIO 都被设置了高电平 CORE 唤醒 suspend，对应的设置 code 在 user_init 里面：

```
u32 pin[] = KB_DRIVE_PINS;
for (int i=0; i<(sizeof (pin)/sizeof(*pin)); i++)
{
    gpio_set_wakeup(pin[i],1,1);    //drive pin core(gpio) high
                                    wakeup suspend
}
```

上面的 gpio_set_wakeup(pin[i],1,1)设置了 drive pin 的唤醒极性（高电平）和唤醒使能。

实际 GPIO 的中断和唤醒前端的处理在硬件上是同一个模块完成的，也就是说 GPIO 中断的电平极性与中断使能和唤醒是用相同的寄存器设置的。gpio_set_wakeup(pin[i],1,1)同时将 GPIO 的中断使能打开并且设置了中断电平为高电平。

按照上面设置后，GPIO 上的高电平可使得 GPIO 的中断服务标志位（core_648 的 BIT(18)）置位，只要查询这个位就知道在之前的一段时间里是否有按键曾经按下过（按键按下时，drive pin 上读到一个 100K 和 10K 分压的 10/11 VCC 高电平）。

```
#define reg_irq_mask        REG_ADDR32(0x640)
#define reg_irq_src         REG_ADDR32(0x648)

FLD_IRQ_GPIO_EN =       BIT(18),
```

只要不打开 GPIO 中断的 mask 位（core_640 的 BIT(18)）就不会触发中断（我们需要的效果就是不响应中断），只是会让中断标志位置位。

app_config.h 中 KEYSCAN_IRQ_TRIGGER_MODE 的定义用于打开这个 keyscan flow 的时间优化，user 可以查询相关代码。

Lenze Technology Co.,LTD

```
#define KEYSCAN_IRQ_TRIGGER_MODE        1
```

初始化：

```
gpio_core_irq_enable_all(1);
reg_irq_src = FLD_IRQ_GPIO_EN;
```

```
static inline u32 kb_scan_key (int numlock_status, int read_key) {
    u8 gpio[8];

#if(KEYSCAN_IRQ_TRIGGER_MODE)
    static u8 key_not_released = 0;
    if(numlock_status & KB_NUMLOCK_STATUS_POWERON){
        key_not_released = 1;
    }

    if(reg_irq_src & FLD_IRQ_GPIO_EN){  //FLD_IRQ_GPIO_RISC2_EN
        key_not_released = 1;
        reg_irq_src = FLD_IRQ_GPIO_EN;  //FLD_IRQ_GPIO_RISC2_EN
    }
    else{  //no key press
        if(!key_not_released && !(numlock_status & KB_NUMLOCK_STATUS_POWERON)){
            return 0;
        }
    }
#endif

    scan_pin_need = kb_key_pressed (gpio);
    if(scan_pin_need){
        return  kb_scan_key_value(numlock_status,read_key,gpio);
    }
    else{
#if (KB_REPEAT_KEY_ENABLE)
        repeat_key.key_change_flg = KEY_NONE;
#endif
#if (KEYSCAN_IRQ_TRIGGER_MODE)
        key_not_released = 0;
#endif
        return 0;
    }
}
```

图 7-3 keyscan 时间优化

keyscan 部分代码如上图，每次首先检查上次 keyscan 结束后到现在的时间段中是否有中断标志置位，有的话，表示这期间发生了按键按下的动作，由于按键是人为触发的，至少会持续 200ms 时间，此时按键还没有被释放，去执行后面的 basic keyscan flow（包括全矩阵通扫、逐行扫等）。

按键松开后，kb_key_pressed 中的 debouce 生效，至少会连续 6 个 loop 的 keyscan，6 次按键检测全部为松开状态时，keyscan 的 flow 才开始不执行。

6.5 Deepsleep wakeup fast keyscan

当 Slave 设备在连接状态时进入 deepsleep 后，被按键唤醒。唤醒后，程序从头开始跑，要经过 user_init 后在 mainloop 中先发广播包等到连上后才能将按键的值发送给 ble master。

17H25 BLE SDK 中已经做了相关的处理让 deep back 的回来速度尽可能的快，但这个时间还是可能会达到 100 ms 级别（如 300ms）。为了防止唤醒的这个按键动作丢掉，在 SDK 里做了快速扫键并缓存数据的处理。快速扫键是因为按键唤醒后 MCU 重新从 flash load 程序重新初

始化会耗掉一些时间，在 mainloop 中 keyscan 的时间由于防抖动滤波处理也会多一些时间，可能会导致这个按键的丢失。缓存数据是因为如果在广播态扫描到了有效的按键数据，push 到 BLE TX fifo 后，进入连接态数据会被重新清掉。

相关的 code 在 app_config.h 中的宏 DEEPBACK_FAST_KEYSCAN_ENABLE 控制。

```
#define DEEPBACK_FAST_KEYSCAN_ENABLE   1

void deep_wakeup_proc(void)
{
    #if(DEEPBACK_FAST_KEYSCAN_ENABLE)
    if(analog_read(DEEP_ANA_REG0) == CONN_DEEP_FLG){
        if(kb_scan_key (KB_NUMLOCK_STATUS_POWERON,1) && kb_event.cnt){
            deepback_key_state = DEEPBACK_KEY_CACHE;
            key_not_released = 1;
            memcpy(&kb_event_cache,&kb_event,sizeof(kb_event));
        }
    }
    #endif
}
```

初始化的时候在 user_init 前就进行按键扫描，读取 deep 不掉电模拟寄存器检测到是连接状态进入 deep 唤醒后，调用 kb_scan_key，这时不启动防抖动滤波处理，直接获取当前读到的整个矩阵的按键状态。若扫描发现有键按下(返回了按键变化并且 kb_event.cnt 非 0)，则将 kb_event 变量 copy 到缓存变量 kb_event_cache。

mainloop 的 keyscan 中添加 deepback_pre_proc 和 deepback_post_proc 处理：

```
void proc_keyboard (u8 e, u8 *p)
{
    kb_event.keycode[0] = 0;
    int det_key = kb_scan_key (0, 1);

#if(DEEPBACK_FAST_KEYSCAN_ENABLE)
    if(deepback_key_state != DEEPBACK_KEY_IDLE){
        deepback_pre_proc(&det_key);
    }
#endif

    if (det_key){
        key_change_proc();
    }

#if(DEEPBACK_FAST_KEYSCAN_ENABLE)
    if(deepback_key_state != DEEPBACK_KEY_IDLE){
        deepback_post_proc();
    }
#endif
```


Lenze Technology Co.,LTD

```
}
```

deepback_pre_proc 处理是等到 slave 和 master 连接上以后，在某一次 kb_key_scan 没有按键状态变化时，将之前缓存的 kb_event_cache 的值作为当前最新的按键变化，实现了快速扫键值的缓存处理。

需要注意一下按键释放的处理，手动给这个按键值时，判断一下当前矩阵按键的状态是否还有按键按着，若有键按着，不用后面加手动的 release，因为实际的按键释放时会产生一个释放动作；若当前按键已经释放了，需要标记一下后面需要给一个手动的 release，否则有可能会出现缓存的按键事件一直有效，无法释放。

deepback_post_proc 处理就是根据 deepback_pre_proc 中是否留下手动 release 事件，来决定要不要往 ble TX fifo 里放一个按键 release 事件。

6.6 卡键处理

卡键处理（Stuck Key process）指的是当一个遥控器/键盘在不用的时候，用户不小心用一些东西把其中一个键或多个键压住了，如家里的茶杯/烟灰缸等压住了遥控器。此时正常的 keyscan 会发现一直有一些按键被按着没有释放，code 上若不做相应的卡键处理，就会一直认为是按键按着没有释放，永远进不了 deepsleep 或其他低功耗状态。

app_config.h 中相关的两个宏为

```
//stuck key
#define STUCK_KEY_PROCESS_ENABLE        0
#define STUCK_KEY_ENTERDEEP_TIME        60//in s
```

默认卡键处理是关着的，将 STUCK_KEY_PROCESS_ENABLE 设为 1 即打开卡键处理。STUCK_KEY_ENTERDEEP_TIME 定义卡键的时间，设为 60 S 表示当一个或多个按键被按着，状态一直没有改变的连续时间超过 60 S，就认为是卡键发生了，此时我们会让 MCU 进入 deepsleep。

搜索 STUCK_KEY_PROCESS_ENABLE 宏，可以在 keyboard.c 中找到相关代码如下：

```
#if (STUCK_KEY_PROCESS_ENABLE)
    u8 stuckKeyPress[ARRAY_SIZE(drive_pins)];
#endif
```

定义一个 u8 的数组 stuckKeyPress[4]，用于记录当前按键矩阵上哪一行或多行上有卡键。该值的获取是在 key_debounce_filter 函数中实现的，请 user 自行理解。

上层的相关处理为：

```
kb_event.keycode[0] = 0;
int det_key = kb_scan_key (0, 1);

if (det_key){
    #if (STUCK_KEY_PROCESS_ENABLE)
    if(kb_event.cnt){  //key press
        stuckKey_keyPressTime = clock_time();
    }
    #endif

    .......
}
```


Lenze Technology Co.,LTD

对于每一个按键状态的变化，当发现是按键按下时（kb_event.cnt 非 0）,记录这个最近的按键按下状态的时间 stuckKey_keyPressTime。

然后在blt_pm_proc中处理如下：

```
#if (STUCK_KEY_PROCESS_ENABLE)
    if(key_not_released  &&  clock_time_exceed(stuckKey_keyPressTime,
STUCK_KEY_ENTERDEEP_TIME*1000000)){
        u32 pin[] = KB_DRIVE_PINS;
        for (int i=0; i<(sizeof (pin)/sizeof(*pin)); i++)
        {
            extern u8 stuckKeyPress[];
            if(stuckKeyPress[i]){
                cpu_set_gpio_wakeup (pin[i],0,1);  //reverse stuck
                                                   key pad wakeup level
            }
        }
        cpu_sleep_wakeup(1, PM_WAKEUP_PAD, 0);  //deepsleep
    }
#endif
```

判断最近的一次按键被按下的时间是否已经连续超过 60s，若超过，则认为是发生了卡键处理，根据底层的 stuckKeyPress[]获取发生卡键的所有行号，将这些行原来高电平 PAD 唤醒 deepsleep 改为低电平 PAD 唤醒 deepsleep。修改的原因是本来按键按下时，对应的行上 drive pin 读到的是 10/11 VCC 高电平，此时是无法进入 deepsleep 的，因为已经是高电平了，只要进入 deepsleep 就会立刻被这个高电平唤醒，修改为低电平唤醒后，可以正常进入 deepsleep，且按键被释放时，行上的 drive pin 的电平变为 100K 下拉的低电平，释放可以唤醒整个 MCU。

6.7 长按键功耗优化

LONG_PRESS_KEY_POWER_OPTIMIZE 宏打开时，可以对长按键的功耗进行优化，详情见低功耗管理部分的详细说明。

7 LED 管理

7.1 LED 任务相关调用函数

17H25 BLE SDK 提供了一个 led 管理的参考代码，以源码提供，user 可以直接使用这部分的 code 或参考其实现方法自己设计。代码在 vendor/common/blt_led.c，user 在自己的 C file 中 include vendor/common/blt_led.h 即可。

user 需要调用的三个函数为：

```
void device_led_init(u32 gpio,u8 polarity);
int device_led_setup(led_cfg_t led_cfg);
```


Lenze Technology Co., LTD

```
static inline void device_led_process(void);
```

在初始化的时候使用 device_led_init(u32 gpio,u8 polarity)设置当前 LED 对应的 GPIO 和极性，极性设为 1 表示 gpio 输出高电平点亮 LED，极性设为 0 表示低电平点亮 LED。

在 mainloop 的 UI Entry 部分添加 device_led_process 函数，该函数每次检查是否有 LED 任务没有完成(DEVICE_LED_BUSY)，若有任务，去执行相应的操作。

7.2 LED 任务的配置和管理

7.2.1 定义 led event

使用如下结构体定义一个 led event

```
typedef struct{
    unsigned short onTime_ms;
    unsigned short offTime_ms;
    unsigned char repeatCount;  //0xff special for long
                                on(offTime_ms=0)/long off(onTime_ms=0)
    unsigned char priority;     //0x00 < 0x01 < 0x02 < 0x04 < 0x08 <
                                0x10 < 0x20 < 0x40 < 0x80
} led_cfg_t;
```

onTime_ms 和 offTime_ms 表示当前的 led event 保持亮起的时间(ms)和熄灭的时间(ms)，注意它们是用 unsigned short int 定义的，最大 65535。repeatCount 表示 onTime_ms 和 offTime_ms 定义的一亮一灭的动作持续 repeat 多少次，注意它是用 unsigned char 定义的，最大 255。priority 表示当前 led event 的优先级。

当我们要定义一个长亮或长灭的 led event 时(没有时间限制，也就是 repeatCount 不起作用)，将 repeatCount 的值设为 255(0xff)，此时 onTime_ms 和 offTime_ms 里面必须一个是 0，一个非 0，根据非 0 来判断是长亮还是长灭。

以下为几个 led event 的示例：

1) 1 Hz 的频率闪烁 3 秒：亮 500ms，灭 500ms，repeat 3 次。
   led_cfg_t led_event1 = {500, 500 , 3, 0x00, };
2) 4 Hz 的频率闪烁 50 秒：亮 125ms，灭 125ms，repeat 200 次
   led_cfg_t led_event2 = {125, 125 , 200, 0x00, };
3) 长亮：onTime_ms 非 0， offTime_ms 为 0，repeatCount 为 0xff
   led_cfg_t led_event3 = {100, 0 , 0xff, 0x00, };
4) 长灭：onTime_ms 为 0， offTime_ms 非 0，repeatCount 为 0xff
   led_cfg_t led_event4 = {0, 100, 0xff, 0x00, };
5) 亮 3 秒后熄灭：onTime_ms 为 1000， offTime_ms 为 0，repeatCount 为 0x3
   led_cfg_t led_event5 = {1000, 0, 3, 0x00, };

调用 device_led_setup 将一个 led_event 送给 led 任务管理：

```
device_led_setup(led_event1);
```


Lenze Technology Co.,LTD

7.2.2 Led event 的优先级

user 可以在 SDK 里定义多个 led event，LED 在一个时间点只能执行一个 led event。这个简单的 led 管理没有设置任务列表，当 led 空闲时，led 接受 user 调用 device_led_setup 建立的任何 led event；当 led busy 时（前一个 old led event 还没有结束），对于 new led event，对两个 led event 的优先级进行比较，若 new led event 的优先级高于 old led event 的优先级，将 old led event 抛弃，开始执行 new led event，若 new led event 的优先级低于或等于 old led event 的优先级，继续执行 old led event，将 new led event 抛弃（注意：是彻底抛弃，并不会将这个 led event 缓存起来后面再处理）。

user 可以根据以上的 led event 优先级的原则，在自己的应用里定义不同优先级的 led event，实现自己的 led 指示效果。

另外，由于 led 的管理采用了查询的机制，当 DEVICE_LED_BUSY 时，不能进入 latency 生效时的 long suspend，如果进入了一个 long suspend（比如 10ms * 50 = 500ms），会导致 onTime_ms 较小的值（如 250ms）无法得到及时响应，从而影响了 LED 闪烁的效果。

```
#define   DEVICE_LED_BUSY   (device_led.repeatCount)
```

针对以上问题，需要在 blt_pm_proc 中作相应的处理：

```
user_task_flg = scan_pin_need || key_not_released || DEVICE_LED_BUSY;
if(user_task_flg){
    bls_pm_setManualLatency(0);   //手动关闭latency
}
```

user 可以参考当前 17H25 ble remote 工程里代码 LED 相关的处理方法。

8 blt software timer

为了方便 user 一些简单的定时器任务，Lenze BLE SDK 提供了 blt software timer demo，并且全部源码提供，user 可以在理解了该 timer 的设计思路后直接使用，也可以自己做一些修改设计。

源代码全部在 vendor/common/blt_soft_timer.c 和 blt_soft_timer.h 文件中，若需要使用，先将下面宏改为 1：

```
#define   BLT_SOFTWARE_TIMER_ENABLE      0   //enable or disable
```

blt soft timer 是基于 system tick 设计的查询式 timer，其准确度无法达到硬件 timer 那么准，且需要保证在 mainloop 中一直被查询。我们预定，blt soft timer 的使用场景为定时时间大于 5ms，且对于时间误差要求不是特别高的情况。

blt soft timer 的最大特点是不仅在 mainloop 中会被查询，也能确保在进入 suspend 后能够及时唤醒并执行 timer 的任务，该设计是基于低功耗唤醒部分介绍的“应用层定时唤醒”实现的。

目前设计上最多同时支持 4 个 timer 运行，实际 user 可以修改下面的宏来实现更多的或者更少的 timer：

```
#define   MAX_TIMER_NUM     4   //timer max number
```


Lenze Technology Co.,LTD

8.1 timer 初始化

调用下面 API 进行初始化：

```
void blt_soft_timer_init(void);
```

可以看到源码上初始化只是将 blt_soft_timer_process 注册为应用层提前唤醒的回调函数。

```
void blt_soft_timer_init(void)
{
    bls_pm_registerAppWakeupLowPowerCb(blt_soft_timer_process);
}
```

8.2 Timer 的查询处理

blt soft timer 的查询处理使用 blt_soft_timer_process 函数来实现：

```
void blt_soft_timer_process(int type);
```

一方面需要在 mainloop 中如下图所示位置确保一直被调用，另一方面被注册为应用层提前唤醒的回调函数，那么每次在 suspend 中发生定时提前唤醒时，也会快速执行该函数，去处理各 timer 任务。

```
void main_loop (void)
{
    static u32 tick_loop;

    tick_loop ++;

    blt_soft_timer_process(MAINLOOP_ENTRY);

    blt_sdk_main_loop();
}
```

blt_soft_timer_process 的参数中 type 如下两种情况，0 表示在 mainloop 中查询进入，1 表示发生了 timer 提前唤醒时进入该函数。

```
#define    MAINLOOP_ENTRY                    0
#define    CALLBACK_ENTRY                    1
```

blt_soft_timer_process 的具体实现比较复杂，基本思路如下：

1) 首先检查当前 timer table 中是否还有 user 定义的 timer，若没有则直接退出，并关掉应用层定时唤醒；若有 timer 任务，继续往下运行。

```
if(!blt_timer.currentNum){
    bls_pm_setAppWakeupLowPower(0, 0);  //disable
    return;
}
```

2) 检查时间上最近的一个 timer 任务是否到达：若没有达到，则退出，否则继续往下运行。设计上会保证 timer 在任何时候都是按照时间排序的，所有这里只要看时间上最近的 timer 即可。

```
if( !blt_is_timer_expired(blt_timer.timer[0].t, now) ){
```


Lenze Technology Co.,LTD

```
        return;
    }
```

3) 轮询当前所有的 timer 任务，只要时间达到了就执行 timer 对应的任务。

```
for(int i=0; i<blt_timer.currentNum; i++){
    if(blt_is_timer_expired(blt_timer.timer[i].t ,now) ){ //timer trigger

        if(blt_timer.timer[i].cb == NULL){
            write_reg32(0x8000, 0x11111122); while(1); //debug ERR
        }
        else{
            result = blt_timer.timer[i].cb();

            if(result < 0){
                blt_soft_timer_delete_by_index(i);
            }
            else if(result == 0){
                change_flg = 1;
                blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
            }
            else{  //set new timer interval
                change_flg = 1;
                blt_timer.timer[i].interval = result * CLOCK_SYS_CLOCK_1US;
                blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
            }
        }
    }
}
```

这里面可以看到对 timer 任务函数的处理，若该函数返回值小于 0，这个 timer 任务会被删掉，后面不再响应；若返回值为 0，保持上一次的定时值；若返回值大于 0，则以该返回值为新的定时周期（单位 us）。

4) 在上面的第 3 步中，如果 timer 任务表中的任务发生了变化，则之前的时间顺序可能会被破坏，这里再重新排序。

```
if(change_flg){
        blt_soft_timer_sort();
}
```

5) 若最近的 timer 任务的响应时间距离现在只剩 3 秒（3S 可以再改大一些）不到，则将该时间设为应用层提前唤醒的时间，否则关闭应用层提前唤醒。

```
if( (u32)(blt_timer.timer[0].t - now) < 3000 *  CLOCK_SYS_CLOCK_1MS){
        bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t,  1);
}
else{
        bls_pm_setAppWakeupLowPower(0, 0);  //disable
}
```

Lenze Technology Co.,LTD

8.3 添加定时器任务

使用如下 API 添加定时器任务：

```
typedef int (*blt_timer_callback_t)(void);

int    blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us);
```

func 为定期执行的任务函数，interval_us 为定时时间，单位为 us。定时任务 func 的 int 返回值三种处理方法为：

1) 返回-小于 0，则该任务执行后被自动删除，可使用这个实现定时器任务的定量执行。
2) 返回 0，则一直使用之前的 interval_us 来定时。
3) 返回值大于 0，则使用该返回值做为新的定时周期，单位 us。

```
int blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us)
{
    int i;
    u32 now = clock_time();

    if(blt_timer.currentNum >= MAX_TIMER_NUM){  //timer full
        return  0;
    }
    else{
        blt_timer.timer[blt_timer.currentNum].cb = func;
        blt_timer.timer[blt_timer.currentNum].interval = interval_us * CLOCK_SYS_CLOCK_1US
        blt_timer.timer[blt_timer.currentNum].t = now + blt_timer.timer[blt_timer.currentN
        blt_timer.currentNum ++;

        blt_soft_timer_sort();
        return  1;
    }
}
```

代码实现中，可以看到当定时器数量超过最大值时，添加失败。每添加一个新的 timer 任务，必须重新做一下排序，以确保定时器任务在任何时候都是按照时间排序的，时间上最近的那个 timer 任务对应的 index 为 0。

8.4 删除定时器任务

除了使用上面返回值小于 0 来自动删除定时器任务，还可以使用下面 API 来指定要删除的定时器任务。

```
int    blt_soft_timer_delete(blt_timer_callback_t func);
```

8.5 Demo

blt soft timer 的 Demo code 请参考 17H25 feature 中 TEST_USER_BLT_SOFT_TIMER。

```
int gpio_test0(void)
{
    DBG_CHN0_TOGGLE;        //gpio 0 toggle to see the effect
    return 0;
}


int gpio_test1(void)
{
    DBG_CHN1_TOGGLE;        //gpio 1 toggle to see the effect

    static u8 flg = 0;
    flg = !flg;
    if(flg){
        return 7000;
    }
    else{
        return 17000;
    }
}

int gpio_test2(void)
{
    DBG_CHN2_TOGGLE;        //gpio 2 toggle to see the effect
    //timer last for 5 second
    if(clock_time_exceed(0, 5000000)){
        //return  -1;
        blt_soft_timer_delete(&gpio_test2);
    }

    return 0;
}

int gpio_test3(void)
{
    //gpio 3 toggle to see the effect
    DBG_CHN3_TOGGLE;
    return 0;
}
```

Lenze Technology Co.,LTD

初始化：

```
blt_soft_timer_init();
blt_soft_timer_add(&gpio_test0, 23000);
blt_soft_timer_add(&gpio_test1, 7000);
blt_soft_timer_add(&gpio_test2, 13000);
blt_soft_timer_add(&gpio_test3, 27000);
```

定义了 4 个任务，这 4 个定时任务各有特点：

1) gpio_test0 每 23ms toggle 一次
2) gpio_test1 使用了 7ms/17ms 两个时间的切换定时
3) gpio_test2 在 5S 后将自己删掉，代码中有两种方式都可以实现这个功能：一是调用 blt_soft_timer_delete(&gpio_test2)；二是 return -1。
4) gpio_test3 每 27ms toggle 一次。

Lenze Technology Co.,LTD

9 Drivers in BLE SDK

9.1 12M/16M 晶体外部电容

SDK 默认使用 17H25 MCU 内部电容（即 ana_81<4:0>对应的 cap）作为 12M/16M 晶振的匹配电容。使用该方案的优势是：在 Lenze 治具上可以测量并调节该电容，使得最终应用产品的频点值达到最佳。

如果需要使用外部焊接电容作为 12M/16M 晶振的匹配电容，则只要在 main 函数开始的地方（一定要在 cpu_wakeup_init 函数之前）调用下面 API 即可：

```
static inline void blc_app_setExternalCrystalCapEnable(u8  en)
{
    blt_miscParam.ext_cap_en = en;
}
```

只要在 cpu_wakeup_init 之前调用该 API，SDK 会自动处理所有的东西，包括关掉内部匹配电容、不再读取频偏校正值等。

9.2 外部 32k 晶振

SDK 默认使用 17H25 MCU 内部 32k 晶体，简称 32k RC。该晶体的最大误差在 500ppm，所以对于 suspend 时间较长的应用，其时间准确性会差一些，目前 32k RC 默认支持的最大长连接不能超过 1.5 S，一旦超过这个时间，ble timing 会出错，造成收包时间点不准确，容易出现收发包 retry，功耗增大，甚至出现断连。

如果 user 需要实现 suspend 时间较大时依然保证时间准确性，需要使用外部 32K 晶体，简称 32k pad。SDK 目前支持这种模式。

在 main 函数开始的地方（一定要在 cpu_wakeup_init 函数之前）调用下面两个 API 中的一个：

```
void blc_pm_select_internal_32k_crystal(void);
void blc_pm_select_external_32k_crystal(void);
```

它们分别是选择 32k RC 和 32k pad 的 API，SDK 默认调用 blc_pm_select_internal_32k_crystal 选择的 32k RC，如果需要使用 32k pad，将其改成 blc_pm_select_external_32k_crystal 即可。

注意：IC 方面，17H25 全部支持；请 user 务必与 Lenze 技术支持人员确认清楚，防止使用了错误的 IC 导致的 32k pad 功能失败。

9.3 PA

如果需要使用 RF PA 的话，请参考 proj/drivers/rf_pa.c 和 rf_pa.h。

首先打开下面的宏，默认是关闭的。

```
#ifndef PA_ENABLE
#define PA_ENABLE                    0
#endif
```


Lenze Technology Co.,LTD

在系统初始化的时候，调用 PA 的初始化。

```
void rf_pa_init(void);
```

参考代码实现，该初始化里面，将 PA_TXEN_PIN 和 PA_RXEN_PIN 设为 GPIO 输出模式，初始状态为输出 0。需要 user 定义 TX 和 RX PA 对应的 GPIO：

```
#ifndef PA_TXEN_PIN
#define PA_TXEN_PIN                    GPIO_PB2
#endif

#ifndef PA_RXEN_PIN
#define PA_RXEN_PIN                    GPIO_PB3
#endif
```

另外将void app_rf_pa_handler(int type)注册为PA的回调处理函数，参考该函数的实现，实际它处理了下面3种PA状态：PA关、开TX PA、开RX PA。

```
#define PA_TYPE_OFF                    0
#define PA_TYPE_TX_ON                  1
#define PA_TYPE_RX_ON                  2
```

User只需要调用上面的rf_pa_init，app_rf_pa_handler被注册到底层的回调，BLE在各种状态时，都会自动调用app_rf_pa_handler的处理。

9.4 PWM

PWM driver 的说明请参考红外那一章的介绍。

9.5 UART

9.5.1 UART GPIO

Lenze 的 17H25 芯片集成了一个 UART 模块，17H25 只有一组 GPIO 用于该 UART。用于 UART 的 GPIO 口具体组合如下表：

UART GPIO 映射

芯片类型	UART ID	GPIO Pin	UART Pin
17H25	1	PC6	Tx
		PC7	Rx
		PD0	RTS
		PD1	CTS

为了使用芯片上的 UART 模块首先需要配置 UART 所使用的 GPIO 口。Lenze 芯片的 GPIO 口是复用的，默认情况下芯片的大部分管脚都工作在 GPIO 模式，为了方便用户将 GPIO 口用于其他复用功能，SDK 提供了 GPIO 功能设置函数，其原型如下：

```
void  gpio_set_func(u32 pin, u32 func)
```


Lenze Technology Co.,LTD

参数	说明
pin	要设置为复用功能的 GPIO
func	要设置的 GPIO 功能

例：

```
gpio_set_func(GPIO_PA6,AS_UART);
gpio_set_func(GPIO_PA7,AS_UART);
```

为了方便使用 UART，SDK 在 uart.h 中以宏定义的形式给出了所有串口的 GPIO 配置，如下：

```
#define    UART_GPIO_CFG_PA6_PA7()
#define    UART_GPIO_CFG_PB2_PB3()
#define    UART_GPIO_CFG_PC2_PC3()
#define    UART_GPIO_CFG_PC6_PC7()
```

用户只需调用相应的宏即可完成 UART GPIO 的配置。

9.5.2 UART 配置

9.5.2.1 UART 通用配置

对于所有的 UART 来说需要配置的基本参数有波特率、数据位、奇偶校验位、停止位等，Lenze 的 MCU UART 的模块也大致一样。用户可以调用 UART 初始化函数来配置上述参数，函数原型如下：

```
unsigned char uart_Init( unsigned short uartCLKdiv,
                         unsigned char bwpc,
                         UART_ParityTypeDef Parity,
                         UART_StopBitTypeDef StopBit)
```

参数	说明
uartCLKdiv	这两个参数是用来确定 Baudrate 的（参见芯片 datasheet），在下述的表中给出了一些常用的 Baudrate 所对应的这两个参数的值。
bwpc	
Parity	奇偶校验位设置，它是个枚举类型的值。
StopBit	停止位设置，它是个枚举类型的值。

例：

```
uart_Init（9,13,PARITY_NONE,STOP_BIT_ONE）;
```


Lenze Technology Co.,LTD

奇偶校验位的枚举类型定义：

```
typedef enum {
    PARITY_NONE = 0,
    PARITY_EVEN,
    PARITY_ODD,
} UART_ParityTypeDef;
```

停止位枚举类型的定义：

```
typedef enum{
    STOP_BIT_ONE = 0,
    STOP_BIT_ONE_DOT_FIVE = BIT(12),
    STOP_BIT_TWO = BIT(13),
} UART_StopBitTypeDef;
```

常用 BaudRate 所对应的 clk_div 和 bwpc 值如下表：

常用 BaudRate

Baudrate	clk_div		bwpc	
	sys_clk 16MHz	sys_clk 32MHz	sys_clk 16MHz	sys_clk 32MHz
4800	302	605	10	10
9600	118	302	13	10
19200	118	118	6	13
38400	25	118	15	6
57600	30	36	8	14
115200	9	30	13	8

对于其他 Baudrate 所对应的的 clk_div 和 bwpc 值可以通过 Lenze 提供的工具软件 TScript 获得。TScript 软件的使用方法放在本节最后介绍。

9.5.2.2 UART 专有配置

Lenze MCU 上集成的 UART 模块除了需要通用配置以外，还需要进行一些 Lenze MCU 专有的配置。Lenze MCU 的 UART 模块有两种工作模式：DMA 模式和非 DMA 模式（Normal 模式），UART 一次只能工作在一种模式下，不同的模式要进行相应的配置。

1) UART 工作在 DMA 模式时所需的配置

UART 工作在 DMA 模式时，需要配置与 UART 相关的 DMA 通道，用户可使用 DMA 初始化函数来配置 UART 的 DMA 通道，其原型如下：

```
void uart_DmaModeInit(    unsigned char dmaTxIrqEn,
                          unsigned char dmaRxIrqEn)
```

参数	说明
dmaTxIrqEn	UART 发送中断使能，1：enable； 0：disable
dmaRxIrqEn	UART 接收中断使能，1：enable； 0：disable

例：

```
uart_DmaModeInit(1,0);
```

在 DMA 模式下，UART 还需要设置一个 buffer 用于 DMA 存放接收的 UART 数据，用户可以调用 uart_RecBufInit()函数去设置接收缓冲区，其原型如下：

```
void uart_RecBuffInit(      unsigned char *recAddr,
                            unsigned short recBuffLen)
```

参数	说明
recAddr	指向 Rx buffer 的指针
recBuffLen	Rx buffer 的长度

例：

```
unsigned char rxBuf[100];

uart_RecBuffInit(rxBuf,100);
```

2) UART 工作在非 DMA 模式下所需的配置

在非 DMA 模式下，用户除了要调用 uart_Init()函数初始化 UART 外，还需要调用 uart_notDmaModeInit()函数来设置 UART，其原型如下：

```
void uart_notDmaModeInit(   unsigned char rx_level,
                            unsigned char tx_level,
                            unsigned char rx_irq_en,
                            unsigned char tx_irq_en)
```

参数	说明
rx_level	用于设置 UART 接收 rx_level 个数据后才产生中断（如果 Rx 中断使能），最大可取 8。
tx_level	用于设置 UART 发送 tx_level 个数据后才产生中断（如果 Tx 中断使能），最大可取 8。
rx_irq_en	UART 接收中断使能；1：enable，0：disable
tx_irq_en	UART 发送中断使能；1：enable，0: disable

注意：对于 UART 发送数据，建议不要采用中断的方式实现，推荐采用查询发送完成标志的方式实现（UART 现有驱动是采用的这种方式实现的），如果一定要采用中断的方式，请联系 Lenze 技术支持。

例：

```
uart_notDmaModeInit(1,0,1,0);
```


Lenze Technology Co.,LTD

为了用户的方便，SDK 还提供了一些宏函数在 uart.h 中，用户只需调用这些宏函数即可完成 UART 的所有配置（只提供了 DMA 模式的 UART 宏函数），现有的宏函数如下：

```
#define CLK32M_UART9600  //sys_clk = 32MHz,Baudrate = 9600
#define CLK32M_UART115200//sys_clk=32MHz,Baudrate = 115200
#define CLK16M_UART115200//sys_clk=16MHz,Baudrate = 115200
#define CLK16M_UART9600  //sys_clk = 16MHz,Baudrate = 9600
```

9.5.3 DMA 模式下 UART 收发数据

UART 接收数据采用中断的方式实现。需要注意的是：上述所述的 DMA 模式 UART 配置只是使能了与 UART 相关的 DMA 中断，要想 CPU 能检测 UART 中断必须开启全局中断，用户只需调用 irq_enable()函数即可实现开启全局中断。

9.5.3.1 DMA 模式下 UART 数据的接收

在 DMA 模式下，UART 接收数据是不需要 CPU 干预的，DMA 模块会自动将 UART 接收到的数据存放到用户初始化 UART 时给定的 Rx buffer 中，只在数据接收完成时产生一个接收中断，告诉 MCU 数据接收完成，之后用户可以从 Rx buffer 中读取数据。为了获取中断用户可以调用 uart_IRQSourceGet()函数，其原型如下：

```
enum UARTIRQSOURCE uart_IRQSourceGet(void)
```

该函数返回一个枚举类型的值，该枚举类型的值指定了中断的类型，其定义如下：

```
enum UARTIRQSOURCE
{
    UARTNONEIRQ = 0,
    UARTRXIRQ = BIT(0),
    UARTTXIRQ = BIT(1),
};
```

UART 产生中断后不需要用户清除中断标志，中断标志由硬件自动清除。

9.5.3.2 DMA 模式下 UART 数据的发送

用户可以调用 uart_Send（）函数发送数据，其原型如下：

```
unsigned char uart_Send(unsigned char* addr)
```

参数	说明
addr	指向用户数据 buffer
返回值	0：DMA 忙；1：发送成功

例：

```
unsigned char txBuf[] = {0x02,0x00,0x00,0x00,0xAA,0xBB};
uart_Send(txBuf);
```

需要注意的是：

1) DMA 模式下 UART 发送数据时，存放发送数据的 buffer 中前 4bytes（低字节在前）用于指定发送数据的长度，这之后才是用户要发送的数据，发送数据的格式如下：

Lenze Technology Co.,LTD

0	1	2	3	4	5	6	…
dataLen	0x00	0x00	0x00	data0	data1	data2	…

2) DMA 模式下的 UART 一次所能发送数据的最大长度为 512-5=507bytes，对于多于 507bytes 的数据应该分批发送。

9.5.4 非 DMA 模式下 UART 收发数据

UART 接收数据采用中断的方式实现。需要注意的是：上述所述的非 DMA 模式 UART 配置只是使能了与 UART 相关的中断，要想 CPU 能检测 UART 中断必须开启全局中断，用户只需调用 irq_enable()函数即可实现开启全局中断。

9.5.4.1 非 DMA 模式下 UART 数据的接收

非 DMA 模式下，UART 接收数据是在中断中实现的，当串口接收到数据后会产生中断，用户可以调用 GET_UART_NOT_DMA_IRQ()函数（这个函数是用宏定义的形式给出的，其返回值：1：表示接收到数据；0：表示未接收到数据）来获取接收中断，然后调用 uart_notDmaModeRevData（）获取接收的数据，其原型如下：

```
unsigned char uart_notDmaModeRevData(void)
```

参数	说明
返回值	返回一个字节接收的数据，对于多个字节循环读取即可。

注意：接收中断的产生取决于用户在 uart_notDmaModeInit（）函数中对 rx_level 参数的设置，例如：rx_level = 1 时，则 UART 接收到一个字节数据后产生接收中断；rx_level = 2 时，则 UART 接收到两个字节数据才产生接收中断；以此类推。

9.5.4.2 非 DMA 模式下 UART 数据的发送

用户可以调用 uart_notDmaModeSendByte（）函数发送数据，其原型如下：

```
unsigned char uart_notDmaModeSendByte(unsigned char uartData)
```

参数	说明
uartData	待发送的数据
返回值	未用

9.5.5 UART 波特率计算工具

1) 打开 TScript 工具软件，其界面如下图所示。

Lenze Technology Co.,LTD

图 9-1 Tscript 启动界面

2) 单击 TScript 窗体左上角的“UART_Baudrate_calculate”选项，在窗体左下角会出现“UART_BaudRate_cal.lua”选项，如下图所示。

图 9-2 UART_BaudRate_cal.lua

3) 双击 TScript 软件窗体左下角的“UART_BaudRate_cal.lua”选项，在 TScript 软件右边的显示区会出现一些提示：“please entry the baudrate”，提示输入波特率，如下图所示。

图 9-3 entry the baudrate

4) 在文本框中输入用户想要的波特率，然后回车，在显示区会出现下一个提示：“please entry system clock”，提示输入系统时钟，如下图所示。

图 9-4 entry system clock

5) 按照提示输入系统时钟，然后回车，出现 baudrate 所对应的 clk_div 和 bwpc 值，如下图所示。

6)

图 9- 5 get result

9.6 ADC

9.6.1 ADC Clock

ADC 模块工作的时钟来源于 FHS（高速时钟）并通过分频得到。Lenze MCU 的 ADC 模块最高的工作频率为 4MHz，具体如何配置请参见芯片 datasheet。ADC 要工作必须开启其时钟，ADC 的驱动中已经为用户配置了 ADC 的时钟，用户不需要额外再配置 ADC 的时钟。

9.6.2 ADC 配置

Lenze MCU 的 ADC 支持 7 种不同的分辨率，2 种工作模式（单端和差分模式），12 个单端输入通道和多组差分输入通道，2 种参考电压（17H25：1.3V 和 3.3V；），电池电压检测和温度检测（基于内部温度传感器）。用户可以调用 adc_Init()函数来初始化 ADC。

9.6.2.1 17H25 ADC 初始化

对于 17H25 芯片来说，adc_Init 函数原型如下：

```
void adc_Init(ADC_CLK_t adc_clock,
              ADC_INPUTCHN_t chn,
              ADC_INPUTMODE_t mode,
              ADC_REFVOL_t ref_vol,
              ADC_RESOLUTION_t resolution,
              ADC_SAMPCYC_t sample_cycle)
```


Lenze Technology LTD

参数	说明
adc_clock	设置 ADC 时钟，它是一个枚举类型的值，只能设置为 ADC_CLK_4M
chn	设置 ADC 通道，它是一个枚举类型的值
mode	设置 ADC 工作模式，它是一个枚举类型的值
ref_vol	设置 ADC 的参考电压，它是一个枚举类型的值
resolution	设置 ADC 的分辨率，它是一个枚举类型的值
sample_cycle	设置 ADC 的采样时间，它是一个枚举类型的值

例：

```
adc_Init(ADC_CLK_4M,            ADC_CHN_D2,SINGLEEND,ADC_REF_VOL_AVDD,
         ADC_SAMPLING_RES_14BIT, ADC_SAMPLING_CYCLE_6);
```

ADC 时钟选择枚举类型定义：

```
typedef enum{
        ADC_CLK_4M = 4,
        ADC_CLK_5M = 5,
} ADC_CLK_t;
```

ADC 输入通道枚举类型定义：

```
typedef enum{
    ADC_CHN_D0          = 0x01,
    ADC_CHN_D1          = 0x02,
    ADC_CHN_D2          = 0x03,
    ADC_CHN_D3          = 0x04,
    ADC_CHN_D4          = 0x05,
    ADC_CHN_D5          = 0x06,
    ADC_CHN_C2          = 0x07,
    ADC_CHN_C3          = 0x08,
    ADC_CHN_C4          = 0x09,
    ADC_CHN_C5          = 0x0a,
    ADC_CHN_C6          = 0x0b,
    ADC_CHN_C7          = 0x0c,
    ADC_CHN_PGA_R       = 0x0d,
    ADC_CHN_PGA_L       = 0x0e,
    ADC_CHN_TEMP_POS    = 0x0f,
    ADC_CHN_TEMP_NEG    = 0x10,
    ADC_CHN_VBUS            = 0x11,
    ADC_CHN_GND             = 0x12,
} ADC_INPUTCHN_t;
```

ADC 工作模式枚举类型定义：

```
typedef enum{
        SINGLEEND,
        INVERTD_5,
        INVERTC_3,
```


Lenze Technology Co.,LTD

```
        CHN_PGA_L,
} ADC_INPUTMODE_t;
```

ADC 参考电压枚举类型定义：

```
typedef enum{
    ADC_REF_VOL_1V3  =0x00, //!< ADC Reference:1.3v
    ADC_REF_VOL_AVDD = 0x01, //!< ADC Reference:AVDD
} ADC_REFVOL_t;
```

ADC 分辨率枚举类型定义：

```
typedef enum{
      ADC_SAMPLING_RES_7BIT  =0,
      ADC_SAMPLING_RES_9BIT  =1,
      ADC_SAMPLING_RES_10BIT = 2,
      ADC_SAMPLING_RES_11BIT = 3,
      ADC_SAMPLING_RES_12BIT = 4,
      ADC_SAMPLING_RES_13BIT = 5,
      ADC_SAMPLING_RES_14BIT = 7,
} ADC_RESOLUTION_t;
```

ADC 采样间隔枚举类型定义：

```
typedef enum{
        ADC_SAMPLING_CYCLE_3   = 0,
        ADC_SAMPLING_CYCLE_6   = 1,
        ADC_SAMPLING_CYCLE_9   = 2,
        ADC_SAMPLING_CYCLE_12  =3,
        ADC_SAMPLING_CYCLE_18  =4,
        ADC_SAMPLING_CYCLE_24  =5,
        ADC_SAMPLING_CYCLE_48  =6,
        ADC_SAMPLING_CYCLE_144 = 7,
  } ADC_SAMPCYC_t;
```

9.6.3 ADC 转化值的获取

ADC 驱动中提供了 ADC 转化值的获取函数，用户可以在自己的程序中调用此函数来获取 ADC 的转化值，函数原型如下：

```
unsigned short adc_SampleValueGet(void)
```

参数	说明
返回值	返回 ADC 转化值

adc_SampleValueGet（）函数获取的是 ADC 的转化值，若要得到实际电压值需要通过计算。

Lenze Technology Co.,LTD

9.6.3.1 17H25 实际电压值计算

对于 17H25，实际电压的计算方法：

$$Vol = V_{ref} * (adc_value) / (2^{resolution});$$

参数的意义参看上表。需要注意的是：17H25 芯片 ADC 的参考电压有两种： AVDD 和内部 1.3V，AVDD 即当前 IC AVDD 管脚上的电压；内部 1.3V 对应 datasheet 上描述的 1.428V 那个参考源（datasheet 数据不够准确，程序中以 1.3V 来计算）。

9.7 低电压检测

Low battery check 是使用 MCU 的 ADC 功能检测当前电压是否偏低。

对于锂电池、纽扣电池等应用， user 必须加入低电压检测功能。因为 17H25 在低于正常门限值的低电压时，可能会出现工作不稳定，尤其对 Flash 写操作和擦除操作上，可能会引起写 Flash 错误，造成难以估计的风险。比如在低压环境下 OTA 更新程序时，可能出现 firmware 写入错误，造成新程序无法执行，产品失效。

应用层一旦检测到低电压，必须将整个 MCU cut off，不再做任何操作。

9.7.1 低电压检测的实现

9.7.1.1 17H25 低电压检测

图 9- 7 17H25 ADC channel

17H25 的低电压检测只能通过 ADC channel 实现，上图为 17H25 的 ADC channel 硬件示意图，通过 adc_Init 初始化 ADC 配置。

Lenze Technology Co.,LTD

图 9- 8 17H25 低电压检测硬件示意图

上图所示为 17H25 低电压检测示意图。由于被检测的电压是电源电压，所以不能用它自己来做参考源，只能选择内部 1.3V 电压，而使用 1.3V 参考电压时，实际能测量的最大电压值不会超过 1.3V，为了满足 VCC 测试范围，必须在硬件电路上设计外部分压网络，图上以 1/3 分压为例，能测到最大电压为 3.9V，如果 user 的测量电压更大，可以设置 1/4、1/5 分压等。VCC 1/3 分压后，选择一个 ADC channel 即可。

如果硬件上有 DCDC 电压转化后给 MCU 供电，需要注意测量的电压必须是 DCDC 转换前的电压，如图上所示。

9.7.2 低电压检测 demo

在 17H25 remote/17H25 module 工程中，给出了低电压检测的 demo，user 可以参考该方法或者加以改进优化去实现自己的低电压检测。

下面以 17H25 remote 的 demo 来说明。

需要在 app_config.h 中打开低电压检测的宏。

```
#define BATT_CHECK_ENABLE        1   //enable or disable battery
```

根据当前 MCU 类型进行低电压检测初始化，17H25 remote 在硬件上增加了 1/3 分压网络，分压后的电压连接 PC4 管脚，code 中使用 adc_init 初始化；17H25remote 没有 DCDC，使用了 VCC 内部 1/3 分压网络分压后进入 ADC，adc_BatteryCheckInit 进行初始化。

```
#if(MCU_CORE_TYPE == MCU_CORE_17H25)
      adc_Init(ADC_CLK_4M,    ADC_CHN_C4,    SINGLEEND,    ADC_REF_VOL_1V3,
                  ADC_SAMPLING_RES_14BIT, ADC_SAMPLING_CYCLE_6);
#endif
```

在 mainloop 中对电压进行检测，一旦发现电压低于 2.0V 门限值，cut off MCU，调用

Lenze Technology Co.,LTD

cpu_sleep_wakeup 直接进入 deepsleep，允许 GPIO PAD 唤醒。

code 如下所示：

```
void battery_power_check(void)
{
    static u32 battCheckTick = 0;
    if(clock_time_exceed(battCheckTick, 100000)){
        battCheckTick = clock_time();
    }
    else{
        return;
    }



    int adc_idx = 0;
    unsigned short adcValue[BATT_CHECK_CNT] = {0};

    for(adc_idx=0;adc_idx<BATT_CHECK_CNT;adc_idx++){
        adcValue[adc_idx] = adc_SampleValueGet();
    }

    unsigned short average_data;
    average_data = filter_data(adcValue,BATT_CHECK_CNT);


    unsigned int tem_batteryVol;          //2^14 - 1 = 16383;
#if((MCU_CORE_TYPE == MCU_CORE_8261)||(MCU_CORE_TYPE == MCU_CORE_8267)||(MCU_CORE_TYP
    tem_batteryVol = 3*(1428*(average_data-128)/(16383-256)); //2^14 - 1 = 16383;
#elif(MCU_CORE_TYPE == MCU_CORE_8266)
    tem_batteryVol = 3*((1300*average_data)>>14);
#endif

    if(tem_batteryVol < 2000){  //when battery voltage is lower than 2.0v, chip will
        cpu_sleep_wakeup(1, PM_WAKEUP_PAD, 0);
    }
}
```

代码中有一些软件上的平均和滤波的处理，是为了让测量到的电压值更加准确。具体实现方法请 user 自行理解，user 可以参考 SDK 给出的方法，也可以使用其他的更好的软件算法去处理。

9.8 IIC

9.8.1 IIC Pin

Lenze MCU 集成了一个 IIC 模块，该 IIC 模块的 SDA 和 SCK 线可以映射到多组 GPIO 上，其映射如表所示。17H25 芯片只有一组 GPIO 映射到 IIC 的 SDA 和 SCK 线上。

表：IIC 的管脚映射

芯片类型	IIC 编号	SDA	SCL
17H25	1	PE7	PF1

Lenze MCU 的 GPIO 是复用的，使用 IIC 功能时需要对相应的 GPIO 口进行相应的配置，用户可以调用 i2c_pin_init（）函数来完成 GPIO 口的配置，其原型如下：

```
void i2c_pin_init(I2C_GPIO_GroupTypeDef i2c_pin_group)
```

参数	说明
i2c_pin_group	要配置的 GPIO Pin 组，是个枚举类型的值

例：

```
i2c_pin_init(I2C_GPIO_GROUP_C0C1);
```

I2C_GPIO_GroupTypeDef 枚举类型定义如下：

```
typedef enum {
#if (MCU_CORE_TYPE == MCU_CORE_17H25)
    I2C_GPIO_GROUP_E7F1,
#endif
    }I2C_GPIO_GroupTypeDef;
```

9.8.2 IIC 配置

Lenze MCU 的 IIC 模块支持 Master/Slave 模式。

9.8.2.1 IIC Master 初始化

为了满足不同用户的使用需求，在驱动中提供了 i2c_master_init_div()和 i2c_master_init_khz（）两个函数用于 IIC Master 初始化，用户可以根据自己的需求调用相应的函数初始化 IIC Master，i2c_master_init_div（）函数的原型如下：

```
void i2c_master_init_div( unsigned char slave_id,
                          unsigned char div_clock)
```

参数	说明
slave_id	设置要访问的从机设备地址
div_clock	设置 IIC 的频率，$f_{iic} = f_{sys_clk}/(4 * div_clk)$

例：

```
//sys_clk=16MHz 时，fiic=200KHz
i2c_master_init_div(0xA0, 0x14);
```

i2c_master_init_khz（）函数原型：

```
void i2c_master_init_khz(  unsigned char slave_id,
                           unsigned int i2c_speed)
```

参数	说明
slave_id	设置要访问的从机设备地址
i2c_speed	直接设置 IIC 的频率

例：

```
//sys_clk=16MHz 时，fiic=200KHz
i2c_master_init_khz(0xA0, 200);
```


Lenze Technology Co.,LTD

注意：Lenze MCU 的 IIC 模块最大的工作频率为 $f_{max_iic}=f_{sys}/10$，建议不要使用最大频率。

9.8.2.2 IIC Slave 初始化

Lenze MCU 的 IIC 模块做 Slave 使用时有两种工作模式：DMA 模式和 MAP 模式，详细细节参见相应芯片 datasheet。

对于 IIC Slave 的初始化，SDK 提供了 Slave 初始化函数，其原型如下：

```
void i2c_slave_init(   unsigned char device_id,
                       enum I2C_SLAVE_MODE i2c_mode,
                       unsigned char* pbuf)
```

参数	说明
device_id	设置设备地址
i2c_mode	设置 Slave Mode，枚举类型的值
pbuf	指向 slave 设备缓冲区。Slave 在 MAP 模式时该参数有效，在 DMA 模式时应该设置为 NULL

例：

```
i2c_slave_init(0xA0, I2C_SLAVE_DMA,NULL);
```

I2C_SLAVE_MODE 枚举类型的定义：

```
enum I2C_SLAVE_MODE{
    I2C_SLAVE_DMA = 0,
    I2C_SLAVE_MAP,
};
```

如果 Lenze MCU 作 Slave 使用，则当 Slave 工作在 DMA 模式下时，IIC Master（不一定是 Lenze 的 MCU 作 Master）访问的是 Lenze MCU 的寄存器和 SRAM 地址空间，访问时需要给定寄存器或者 SRAM 中存储单元的地址；当 Slave 工作在 MAP 模式下时，IIC Master（不一定是 Lenze 的 MCU 作 Master）访问的是用户在 i2c_slave_init（）函数中给定的 pbuf 所指向的存储空间，且访问时不需要给定存储单元的地址信息，直接操作，具体内容参见芯片 datasheet。

9.8.3 IIC 数据的传输

当 Lenze MCU 的 IIC 工作在 Master 模式时可以访问具有 IIC 接口的 Slave 设备（可以是其他厂商生产的具有 IIC 接口的 Slave 设备，也可以是 Lenze MCU IIC 作 Slave 设备）；当 Lenze MCU 的 IIC 作 Slave 设备使用时，SPI Master 可以访问 Lenze MCU 内部的寄存器和 SRAM 地址空间（Slave 设备工作在 DMA 模式时）或者访问用户指定的内存区地址空间（Slave 设备工作在 MAP 模式时）。

9.8.3.1 IIC Master 写数据

Lenze MCU IIC 模块工作在主机时支持 8bit 和 16bit 地址宽度 slave 设备的访问，SDK 提供了 IIC 写数据函数，其原型如下：

```
void i2c_write_dma(      unsigned short addr,
                         unsigned char addr_len,
                         unsigned char* pbuf,
```


Lenze Technology Co.,LTD

```
                                int len)
```

参数	说明
addr	设置要访问的从机地址
addr_len	从机地址长度 1:8bit addr; 2:16bit addr
pbuf	指向待写数据的缓冲区
len	待写数据的长度

例：

```
unsigned char dataBuf[] = {0x00,0x11,0x22,0x33,0x44};

i2c_write_dma(0x0000,2,dataBuf,sizeof(dataBuf));
```

如果使用 Lenze MCU 的 IIC 作 Slave 设备且 Slave 设备工作在 MAP 模式时，用户需要在 IIC Master 中调用 i2c_write_mapping（）向 Slave 写数据，写数据时不需要给定内存地址，该函数的原型如下：

```
void i2c_write_mapping(unsigned char* pbuf, int len)
```

参数	说明
pbuf	指向待写数据的缓冲区
len	数据的长度

例：

```
unsigned char writeBuf[]={0x00,0x01,0x02,0x03};

i2c_write_mapping(writeBuf, sizeof(writeBuf));
```

9.8.3.2 IIC Master 读数据

用户可以调用 i2c_read_dma（）函数来读取 slave 设备的数据，其原型如下：

```
void i2c_read_dma(    unsigned short addr,
                      unsigned char addr_len,
                      unsigned char* pbuf,
                      int len)
```

参数	说明
addr	设置要访问的从机地址
addr_len	从机地址长度；1:8bit addr; 2:16bit addr
pbuf	指向存放读取数据缓冲区
len	读数据的长度

例：

```
unsigned char dataBuf[] = {0x00,0x11,0x22,0x33,0x44};
i2c_read_dma(0x00ff,2,dataBuf,sizeof(dataBuf));
```

如果使用 Lenze MCU 的 IIC 作 Slave 设备且 Slave 设备工作在 MAP 模式时，用户需要在 IIC Master 中调用 i2c_read_mapping（）从 Slave 读数据，读数据时不需要给定内存地址，该函数的原型如下：

```
void i2c_read_mapping(unsigned char* pbuf, int len)
```


Lenze Technology Co.,LTD

参数	说明
pbuf	指向读数据的缓冲区
len	数据的长度

例：

```
unsigned char readBuf[]={0x00,0x01,0x02,0x03};
i2c_read_mapping(readBuf, sizeof(readBuf));
```

9.8.3.3 IIC Slave 收发数据

当 IIC 工作在 Slave 模式并且正确初始化以后，数据的发送和接收完全由硬件处理。如果 Lenze MCU 作 Slave 使用且工作在 MAP 模式时，IIC Master 在访问这个 Slave 时不必给定内存地址，直接读写数据就行。

9.8.4 IIC interrupt

Lenze MCU 工作在 Slave 模式时，如果 IIC Master 向 Slave 设备写数据或者从 Slave 设备中读取数据时，Slave 设备会检测 IIC Master 的 Start 信号和 Stop 信号，然后产生中断（如果中断使能）。调用 I2C_IRQ_EN()宏函数可以使能 IIC 中断，注意别忘了还要调用 irq_enable()使能全局中断。

使能 IIC 中断以后，用户可以在中断服务函数中调用 I2C_SlaveIrqGet（）来获取中断标志位，其原型如下：

```
I2C_I2CIrqSrcTypeDef I2C_SlaveIrqGet(void)
```

参数	说明
返回值	返回中断标志，是个枚举值

I2C_I2CIrqSrcTypeDef 枚举类型定义：

```
typedef enum {
    I2C_IRQ_NONE = 0,
    I2C_IRQ_HOST_WRITE_ONLY,
    I2C_IRQ_HOST_READ_ONLY,
}I2C_I2CIrqSrcTypeDef;
```

9.9 SPI

9.9.1 SPI Pin

Lenze MCU 集成了一个 SPI 模块，该 SPI 模块支持 Master/Slave 模式并且 SPI 的 MISO，MOSI，CS，CK 可以映射到不同的 GPIO 上。其中 17H25 芯片 SPI 的控制线只能映射到一组 GPIO 上。SPI 模块控制线的映射如下表所示。

表：SPI 控制线 GPIO 映射

芯片类型	SPI 编号	GPIO Pins	SPI Ctrl Lines

Lenze Technology Co.,LTD

17H25	1	PE6	CS
		PE7	DI
		PF0	DO
		PF1	CK

Lenze MCU 的 GPIO Pin 是复用的，因此，作 SPI 功能使用时需要配置 GPIO Pin 为 SPI 功能。用户可以调用 SPI 管脚初始化函数来配置 GPIO，对于 17H25 芯片来说，其 SPI Pin 初始化函数的原型如下：

```
void spi_master_pin_init (unsigned int cs_pin)
```

参数	说明
cs_pin	用于设置 SPI 的片选管脚

例：

```
spi_master_pin_init(GPIO_PC0);
```

注意：如果 Lenze MCU 的 SPI 作 Master 使用时，其片选管脚既可以是 SPI 模块自带的片选管脚，也可以设置其他 GPIO 作 SPI 片选管脚，用户只需在上述函数的形参中给定自己想要的 GPIO Pin 即可；如果 Lenze SPI 作 Slave 使用时，SPI 的片选管脚只能选择 SPI 模块自带的片选管脚，正因如此，所以驱动中将 Slave Pin 初始化封装在 slave_init（）函数 (在本节之后描述) 中，不再单独给出 Slave Pin 初始化函数定义。

Lenze Technology Co.,LTD

9.9.2 SPI 配置

Lenze MCU 的 SPI 模块支持 Master 和 Slave，并支持 4 种工作模式，详情见芯片的 datasheet。

9.9.2.1 SPI Master 初始化

当 SPI 模块要工作在 Master 时，用户需要调用 spi_master_init（）函数来完成 SPI 模块的配置，其原型如下：

```
void spi_master_init(     unsigned char div_clk,
                          enum spi_mode_t spi_mode)
```

参数	说明
div_clk	设置 SPI 工作时钟。$f_{spi} = f_{sys} / (2 *(div_clk + 1))$
spi_mode	设置 SPI 工作模式，是枚举值。

例：

```
spi_master_init(0x0f, SPI_MODE0);
```

spi_mode_t 的定义如下：

```
enum spi_mode_t{
        SPI_MODE0 = 0,
        SPI_MODE1 = 2,
        SPI_MODE2 = 1,
        SPI_MODE3 = 3,
};
```

需要注意的是：$f_{spi} < f_{sys} / 5$;

9.9.2.2 SPI Slave 初始化

当 SPI 模块要工作在 Slave 时,用户需要调用 spi_slave_init()函数来完成 SPI 的配置。对于 17H25 芯片，该函数的原型如下：

```
void spi_slave_init(enum spi_mode_t spi_mode)
```

参数	说明
spi_mode	设置 SPI 工作模式，是枚举值。

例：

```
spi_slave_init(SPI_MODE0);
```


Lenze Technology Co.,LTD

9.9.3 SPI 数据的传输

当 Lenze MCU 的 SPI 作 Master 时可以访问其他具有 SPI 接口的 Slave 设备（可以是其他厂商生产的具有 SPI 接口的 Slave 设备，也可以是 Lenze MCU 的 SPI 作 Slave 设备）；当 Lenze MCU 的 SPI 工作在 Slave 时，SPI Master 可以访问 Lenze MCU 的寄存器地址空间和 SRAM 地址空间。

9.9.3.1 SPI Master 写数据

当 SPI 工作在 Master 模式且初始化完成以后就可以调用 spi_write（）函数来完成数据的发送，其原型如下：

```
void spi_write(   unsigned char* addr_cmd,
                  unsigned char addr_cmd_len,
                  unsigned char* pbuf,
                  int buf_len,
                  unsigned int cs_pin)
```

参数	说明
addr_cmd	指向 SPI 待写的寄存器的地址和写命令的缓冲区
addr_cmd_len	地址和命令的长度
pbuf	指向待写数据的缓冲区
buf_len	待写数据的长度
cs_pin	片选管脚

例：

```
#define SLAVE_REG_ADD_H 0x80
#define SLAVE_REG_ADD_L 0x00
#define SPI_WRITE_CMD   0x00//Lenze SPI write cmd=0x00
unsigned char slaveRegAddr_WriteCMD[]= {SLAVE_REG_ADD_H,
                                        SLAVE_REG_ADD_L,
                                         SPI_WRITE_CMD};
unsigned char spi_write_buff[]= {0x00,0x11,0x22};
spi_write(slaveRegAddr_WriteCMD, 3,
          spi_write_buff,  sizeof (spi_write_buff),
          GPIO_PC0
          );
```

9.9.3.2 SPI Master 读数据

用户可以调用 spi_read()函数来读取数据，其原型如下：

Lenze Technology Co.,LTD

```
void spi_read(  unsigned char* addr_cmd,
                unsigned char addr_cmd_len,
                unsigned char* pbuf,
                int buf_len,
                unsigned int cs_pin)
```

参数	说明
addr_cmd	指向 SPI 待读的寄存器的地址和读命令的缓冲区
addr_cmd_len	地址和命令的长度
pbuf	指向存储数据的缓冲区
buf_len	读数据的长度
cs_pin	片选管脚

例：

```
#define SLAVE_REG_ADD_H 0x80
#define SLAVE_REG_ADD_L 0x00
#define SPI_READ_CMD    0x80//Lenze SPI read cmd=0x80
unsigned char slaveRegAddr_ReadCMD[]= {SLAVE_REG_ADD_H,
                                       SLAVE_REG_ADD_L,
                                       SPI_READ_CMD};
unsigned char spi_read_buff[10];
spi_read(slaveRegAddr_ReadCMD, 3,
         spi_read_buff,   sizeof (spi_read_buff),
         GPIO_PC0
         );
```

9.9.3.3 SPI Slave 收发数据

当 SPI 工作在 Slave 模式并且正确初始化以后，数据的发送和接收完全由硬件处理。

9.9.4 SPI interrupt

当 SPI 工作在 Slave 模式时，如果 SPI Master 向 Slave 设备写数据或者从 Slave 设备中读取数据时，会产生中断（如果中断使能）。用户可以调用宏函数 SPI_IRQ_EN()来完成 SPI 中断的使能，但别忘了还需要调用 irq_enable()函数使能全局中断。使能 SPI 中断以后，用户可以在中断服务函数中调用宏函数 SPI_IRQ_GET()来获取中断标志位，调用 SPI_IRQ_CLK()宏函数来清除中断标志。

9.10 EMI

9.10.1 EMI Test

EMI Test 在测试时，需要调用 rfdrv 相关的接口，比如 rf_drv_init()、rf_drv_1m()、rf_drv_2m()等，这些操作接口都封装到 library 中，在 rf_drv.h 中可以看到 API 声明。

Lenze Technology Co.,LTD

EMI Test 有四种测试模式：

```
Struct test_list_sate_list[] = {
        {0x01,EmiCarrierOnly},
        {0x02,EmiCarrierData},
        {0x03,EmiRx},
        {0x04,EmiTxPrbs9},
        {0x05,EmiTx55},
        {0x06,EmiTxff},
    };
```

其中： Carrier 模式（载波发送模式）、CD 模式（载波上有数据的发送模式）、RX 模式、三种 TX 模式（发送的数据包类型不同）。

9.10.1.1 Emi 初始化设置

1) 在进行 EMI 测试前，首先需要先调用 rf_drv_init()函数来完成 rf 初始化：
   void rf_drv_init (int xtal_type);
   其中参数 xtal_type 用来选择 external crystal，XTAL_12M/ XTAL_16M。
2) 设置完 rf 后，需调用 Rf_Emi_Init()函数，Rf_Emi_Init()会记录测试之前的一些数据：
   int Rf_EmiInit(void);
3) 如果需要打开 usbprint 功能，需要进行如下配置：
   WriteAnalogReg (0x88, 0x0f);
   WriteAnalogReg (0x05, 0x60);
   write_reg8(0x80013c,0x10);
4) 接下来需要设置 emi 初始状态：
   write_reg8(RUN_STATUE_ADDR,run); //run,0
   write_reg8(TEST_COMMAND_ADDR,cmd_now); //cmd,1
   write_reg8(POWER_ADDR,power_level); //power,0
   write_reg8(CHANNEL_ADDR,chn); //chn,2
   write_reg8(RF_MODE_ADDR,mode); //mode,1
   write_reg8(TX_PACKET_MODE_ADDR,tx_mode); //tx_mode,0
5) 调用 PhyTest_PRBS9()函数，完成 tx buffer 中的数据初始化设置。
   void PhyTest_PRBS9 (unsigned char *p, int n);
   PhyTest_PRBS9()函数是往首地址为 p，长度为 n 的 buffer 中写 prbs9 数据。
6) 最后在测试之前，调用 irq_disable()函数将所有中断关闭。

Lenze Technology Co.,LTD

9.10.1.2 Power level and Channel

在测试时，可以通过配置 rf power level 和 rf channel 来设置发包的能量以及发包的 channel。

Power level ：发射功率选择，枚举数据类型

RF_POWER_8dBm：0，实际为 7dbm 发射功率
RF_POWER_4dBm：1，实际为 5dbm 发射功率
RF_POWER_0dBm：2，实际为-0.6dbm 发射功率
RF_POWER_m4dBm ：3，实际为-4.3dbm 发射功率
RF_POWER_m10dBm ：4，实际为-9.5dbm 发射功率
RF_POWER_m14dBm ：5，实际为-13.6dbm 发射功率
RF_POWER_m20dBm ：6，实际为-18.8dbm 发射功率
RF_POWER_m24dBm：8，实际为-23.3dbm 发射功率
RF_POWER_m28dBm ：9，实际为-27.5dbm 发射功率
RF_POWER_m30dBm ：10，-30dbm 发射功率
RF_POWER_m37dBm ：11，-37dbm 发射功率
RF_POWER_OFF ：16，关闭 PA

RF Channel：设置的频率值等于（2400+chn）MHz
(0≤chn≤100)

其中，设置 power level 时，需要注意一下，其发射功率以实际值为准。比如如果设置 power level 为 RF_POWER_8dBm，实际上其发射功率为 7dbm。用户可以通过调用下面的函数来实现 power 设置：

```
void    rf_set_power_level_index (int level);
```

其中参数 level 就按照 power level 设置即可。

设置 chn 时，chn 范围为 0~100。比如用户想设置 2405MHz 的 channel，将 chn 设置为 5 即可。用户可以调用如下函数：

```
void    SetRxMode (signed char chn, unsigned short set);
```

其中，参数 chn 可参照 RF_channel 设置，参数 set 设置为 0。

Lenze Technology Co.,LTD

9.10.1.3 Emi Carrier Only

Carrier 模式为 EMI Test 载波发送模式，用户直接调用 EmiCarrierOnly()函数即可，无需其它设置。

```
void   EmiCarrierOnly(int power_level, signed char rf_chn);
```

其中，参数 powerlevel 和参数 rf_chn 按照前面介绍的设置方法来设置即可。

在设置 Carrier 模式时，EmiCarrierOnly()函数会调用 Rf_EmiCarrierRecovery()函数来恢复 emi 的一些寄存器默认设置。

```
int          Rf_EmiCarrierRecovery(void);
```

9.10.1.4 Emi Carrier Data

CD 模式为 EMI Test 载波上有数据发送模式，其中载波上的数据由 Rf_EmiDataUpdate()函数来更新，以此来保证载波上的数据是一系列随机数。用户直接调用 EmiCarrierData()函数即可进入 CD 模式，无需其它设置。

在设置 CD 模式时，EmiCarrierData()函数会调用 Rf_EmiCarrierDataTest()函数来完成 CD 模式的设置，如 power level，chn 等。也会调用 Rf_EmiDataUpdate()函数来更新载波上的数据。

```
void       EmiCarrierData(int power_level,signed char rf_chn);
```

其中，参数 power level 和参数 rf_chn 参照前面的介绍来设置即可。

```
void       Rf_EmiCarrierDataTest(int power_level,signed char rf_chn);
void       Rf_EmiDataUpdate(void);
```

Rf_EmiCarrierDataTest()函数中的参数设置与 EmiCarrierData()函数一致。

9.10.1.5 Emi TX

Tx 模式为可以发送三种类型的数据包，PRBS9 packet payload、00001111 packet payload、10101010 packet payload，用户可以通过 cmd 来选择不同 TX 模式。

用户直接调用 EmiTXff()、EmiTx55()、EmiTxPrbs9()其中一个函数即可进入 TX 模式，无需其它设置。

```
void       EmiTxPrbs9(int power_level,    signed char rf_chn);
void       EmiTx55(int power_level,    signed char rf_chn);
void       EmiTxff(int power_level,    signed char rf_chn);
```

其中，参数 power level 和参数 rf_chn 参照之前的介绍来设置即可。

EmiTXff()、EmiTx55()、EmiTxPrbs9()函数都会调用 Rf_EmiTxInit()函数来完成 EMI TX 初始化设置，做完 TX 初始化之后，结合 Rf_EmiSingleTx()函数开始发包。

```
void   Rf_EmiTxInit(int power_level, signed char rf_chn);
void   Rf_EmiSingleTx(unsigned char *addr, int power_level);
```

其中，参数 power level 和参数 rf_chn 参照之前的介绍设置即可，参数 addr 设置为 tx buffer 首地址。

9.10.1.6 EMI RX

Rx 模式采用轮询的方式来接收数据。Rx 模式下，通过 EmiRx()函数来完成状态寄存器及标志位的初始化设置，其中通过标志位可以判断是否有数据进来，然后通过 EmiRxProc()函数进行数据处理。用户直接调用 EmiRx()和 EmiRxProc()函数即可进入 Rx 模式，无需其它设置。

Lenze Technology Co.,LTD

```
void    EmiRx(intpower_level, signed char rf_chn);
void    EmiRxProc(void);
```

其中，参数 power level 和参数 rf_chn 参照之前的介绍来设置即可。

函数 EmiRx()会调用 Rf_EmiRxTest()函数来设置接收 buffer 的首地址、size、chn 等。

```
void Rf_EmiRxTest (    unsigned char *addr,
                       signed char rf_chn,
                       int buffer_size,
                       unsigned char    pingpong_en)
```

addr：接收 buffer 在 ram 中的指针（一般是一个数组的起始地址）；

buffer_size： buffer 长度（16 的整数倍，主要用在 pingpong buffer 的情况下）

rf_chn:设置 tx channel（参照之前 rf_chn 的设置）；

pingpong_en：是否使能 pingpong buffer，1：使能；0：关闭。

Lenze Technology Co.,LTD

9.10.1.7 配置参数设置

Run：

0	Default	1	Start test

Cmd：

1	Carrier	2	CD	3	RX
4	TX(PRBS9)	5	TX(0x55)	6	TX(0x0f)

Power 、channel 前面已经介绍。

Mode:

0	Ble_2M	1	Ble_1M

这些参数的默认上电状态为（mode=1;power=0;channel=2;cmd=1），即在 ble_1M 模式、2402MHz 下以 7dbm 的发射功率发射单载波。

用户可以通过 flash 设置相应配置参数（如需修改 flash 中的定制参数，需先将该区域擦掉，再重新写入新值），也可以通过 Ram address 配置相应参数（若断电，则会恢复默认值）。

注意 Flash 地址是 user 可修改的，具体的值请根据 Flash 地址空间分配情况去决定，下面以 3f000 sector 为例来说明。

```
#define EMI_TEST_TX_MODE          0x3f005
#define EMI_TEST_RUN              0x3f006
#define EMI_TEST_CMD              0x3f007
#define EMI_TEST_POWER_LEVEL      0x3f008
#define EMI_TEST_CHANNEL          0x3f009
#define EMI_TEST_MODE             0x3f00a
```

通过修改 flash 中 0x3f007~0x3f00a 四个地址来改变测试状态，可以根据需要，将对应参数写到相应的 flash 地址，然后重新上电即可得到想要的状态。

通过修改 Ram_address 中 0x8007~0x800a（四个地址来改变测试状态，根据需要，将相应的参数写入到相应的 RAM 地址，最后将 0x8006 写 1，则可以输出想要的测试状态。

Ram_address	Flash_address	function
0x8004		Rssi
0x8005	0x3f005	对于 cmd(4/5/6)可以配置为如下两种模式: 1:发送 1000 个包停止 0：一直发包
0x8006		Run
0x8007	0x3f007	Cmd
0x8008	0x3f008	Power
0x8009	0x3f009	Channel
0x800a	0x3f00a	Mode
0x800c-0x800f		RX_packet_num(处于 RX mode 时，该地址存储接收到的包数，四个字节)

Lenze Technology Co.,LTD

	0x1E000/0x77000	0x81_Cap_value(0xbf<cap< 0xe0)
	0x1E040/0x77040	Tp0(1M: 0x13<Tp0<0x27; 2M: 0x36<Tp0<0x4a)
	0x1E080/0x77041	Tp1(1M: 0x0f<Tp0<0x23; 2M: 0x2f<Tp1<0x43)

9.10.2 EMI Test Tool

为了方便测试，用户可以结合 EMI Test Tool 工具进行 EMI 测试，工具界面如下图所示：

图 9- 12 EMI test tool

第一步：用户可以选择不同方式与硬件连接。当选择 Swire 时，如果系统时钟在 16MHz 或以下，需要通过 WTCDB 工具 SWB SPEED 一下，保证能正常通信。

图 9- 13 选择数据总线

Lenze Technology Co.,LTD

图 9- 14 Swire 同步操作

第二步：设置 chn，可以在输入框内直接输入，然后点击 Set_Channel 即可。如果正常通信的话，会显示 Swire ok，如下图中所示。

图 9- 15 set channel

第三步：通过下拉框可以选择不同的 power level 及 ble mode，选择完后，点击右侧 set 按钮（“Set_Power”/“Set_RF_Mode”）即可完成设置。

图 9-16 选择 RF 模式

图 9-17 set RF 模式显示界面

第四步：点击 Carrier、CarrierData、RXTest、PRBS9、0x55、0x0f 即可进入不同的模式。

图 9-18 选择测试 mode

第五步：TX 模式，可以选择发 1000 个包或无限发包。

图 9-19 设置 TX packet number

图 9-20 TX mode 显示界面

第六步：RX 模式，可以通过点击 Read_Rx_Cnt 读取收包个数，点击 ReadRssi 可以获取当前 RSSI，如下图所示。

Lenze Technology Co.,LTD

图 9-21 RX packet number 和 RSSI

9.11 PHY test

待补充。

10 红外

10.1 PWM Driver

Lenze 的 17H25 spec 上有一章详细介绍 PWM 的章节，请 user 先认真阅读那一章，本文档在那个基础上介绍 PWM 驱动的实现。

由于 PWM 相关的硬件配置都非常简单，基本都是直接操作寄存器来实现，BLE SDK 不使用专门的驱动文件，而是将操作接口都定义在 proj/mcu_spec/register.h/register.h 中，使用 static inline function 来实现，这样可以提高运行效率，也节省 code size。这部分的设置请结合 17H25 spec 中 PWM Register Table 来理解。

10.1.1 PWM id and pin

17H25 总共 12 路 PWM，分别为 PWM0 ~ PWM5 和 PWM0_N ~ PWM5_N。驱动上定义了 6 路 PWM 为：

```
typedef enum {
    PWM0_ID = 0,
    PWM1_ID,
```


Lenze Technology, LTD

```
        PWM2_ID,
        PWM3_ID,
        PWM4_ID,
        PWM5_ID,
    }pwm_id;
```

软件上只设置 6 路 PWM0~PWM5，另外 6 路 PWM0_N~ PWM5_N 是对应 PWM 的波形取反。PWM0_N 和 PWM0 相反，当 PWM0 为高时 PWM0_N 为低，而 PWM0 为低时 PWM0_N 为高。所以只要设置了 PWM0，PWM0_N 就被设置了。同理其他几路也一样。

17H25 的 12 路 PWM 对应的 IC 管脚如下表所示：

Pin	PWM	Pin	PWM
PC0	PWM0	PB7	PWM0_N
PC3	PWM1	PC1/PC2	PWM1_N
PC4	PWM2	PC5	PWM2_N
PA1/PD2	PWM3	PA4	PWM3_N
PA5/PD3	PWM4	PA6	PWM4_N
PB0	PWM5	PB1	PWM5_N

使用 void gpio_set_func(u32 pin, u32 func)来设置对应 pin 的 PWM 功能，比如要使用 PA0 作为 PWM0 来用：

gpio_set_func(GPIO_PA0, AS_PWM)

10.1.2 PWM clock

使用 pwm_set_clk(int system_clock_hz, int pwm_clk)来设置 PWM 的 clock，system_clock_hz 填当前的系统时钟 CLOCK_SYS_CLOCK_HZ，pwm_clk 为要设置的 clock， system_clock_hz 一定要能被 pwm_clk 整除，这样才能分频得到这个 clock。为了让 PWM 时间尽量精准，推荐使用的 pwm_clk 等于 system_clock_hz，即：

pwm_set_clk(CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_HZ);

system_clock_hz 和 pwm_clk 都是 system clock 对应的 tick 值，即每秒钟 system clock 增加的数值。

10.1.3 PWM cycle and duty

在设置了 pwm clock 后，对于每个 pwm 的波形，需要设置周期 cycle 和占空比 duty。

static inline pwm_set_cycle(pwm_id id, u16 cycle_tick)用于设置对应的 PWM 的周期 cycle，cycle_tick 单位为 pwm clock 的 tick 值。

static inline pwm_set_cmp(pwm_id id, u16 cmp_tick)用于设置对应的PWM周期中高电平的时间，cmp_tick 单位为 pwm clock 的 tick 值。占空比等于 cmp_tick/ cycle_tick。对于 PWM0 ~ PWM5，硬件上自动会将高电平放前面，低电平放后面。

如果想要低电平在前的话，有以下几种方法：

Lenze Technology Co.,LTD

1. 使用对应的 PWM0_N ~ PWM5_N，和 PWM0 ~ PWM5 相反
2. 使用 void pwm_revert(pwm_id id)接口将 PWM0 ~ PWM5 的波形直接取反。

比如当前的 pwm clock 为 16MHz，需要设置 PWM 周期为 1ms，占空比为 40%的 PWM0 一个 frame 方法为：

```
pwm_set_cycle(PWM0_ID , 16000);     //16*1000
pwm_set_cmp (PWM0_ID , 6400);       //16*400
```

static inline pwm_set_cycle_and_duty(pwm_id id, u16 cycle_tick, u16 cmp_tick)接口是将上面两个接口合二为一，可以使用这个接口提高设置的效率。

10.1.4 PWM revert

static inline void pwm_revert(pwm_id id)用于将 PWM0 ~ PWM5 波形取反。
static inline pwm_n_revert(pwm_id id) 用于将 PWM0_N ~ PWM5_N 波形取反。

10.1.5 PWM mode

PWM 共 3 种 mode，如下定义，其中 PWM0 ~ PWM5 都具有 normal mode，而只有 PWM0~PWM1 才具有 count mode 和 IR mode。

```
typedef enum{
    PWM_NORMAL_MODE = 0x00,
    PWM_COUNT_MODE  = 0x01,
    PWM_IR_MODE     = 0x03,
   }pwm_mode;
```

1. normol mode
   普通的 PWM 模式，pwm_set_clk/ pwm_set_cycle_and_duty 配置好的 PWM 时序称之为一个 frame。用 pwm start 开启这个 PWM 后，会按照这个 frame 时序一直运行，一个 frame 结束后，会立刻继续运行这个 frame，不会自动停止。只能用 pwm stop 来停止该 PWM。
2. count mode
   用 pwm_set_clk/ pwm_set_cycle_and_duty 配置好一个 PWM frame 后，使用 pwm_set_pulse_num 设定 PWM 持续运行的 frame 数量。假设设置的 pulse number 为 n，pwm start 后，连续运行 n 个 frame 后，PWM 自动停止，不需要再使用 pwm stop 来停止。
   用 pwm_set_clk/pwm cycle and dut/pwm_set_pulse_num 设定一组 PWM frame 的时序，pwm start 运行到自动结束后，如果想要继续运行这一组 PWM 时序，需要再次 pwm start。在一组 PWM 运行过程中还没有结束的时候，如果用 pwm cycle and duty 去修改当前正在运行的时序，会立刻起作用，比如设置了 1ms 周期占空比为 1/2 的 PWM frame，pwm_set_pulse_num 设置 frame number 为 10，当运行到第 6 个 frame 时使用 pwm cycle and duty 修改 frame 的时序为周期 2ms 占空比 1/3，那么剩余的 4 个 frame 就会变为最新设置的周期 2ms 占空比 1/3。

Lenze Technology Co.,LTD

3. IR mode

类似于上面的 count 模式，用 pwm_set_clk/pwm cycle and dut/pwm_set_pulse_num 设定一组 PWM frame 的时序（后面称之为一个 ir task，会用 ir_task0、ir_task1…ir_taskn 甚至更简单的 ir0、ir1…irn 来表示）。使用 pwm start 后，硬件开始运行这一个 ir_task，这个 ir_task 结束后，PWM 不会自动停止，会继续运行这个 ir_task，且一直这样循环下去，只能用 pwm stop 来停止。

IR mode 和 count mode 的第一个区别是不会自动停止，连续不断的运行。

第二个区别是，count mode 正在运行时，如果修改了周期和占空比，PWM frame 时序会立刻改变，而 IR mode 不会立刻改变：IR mode 下，设定第一个 ir_task0 后，在运行 ir_task0 过程中还没有结束的时候如果修改 cycle/duty/pulse number 得到 ir_task1，此时 PWM 的时序不会立刻改变（还是运行之前的 ir_task0），而是要等到 ir_task0 完全结束后才会运行新的 ir_task1。

下面举例具体描述 IR mode 的时序。

图 10- 1 PWM timing and set

结合上图所示的 PWM timing and PWM set 来说明。ts0 是第一次设置 pwm frame/pulse number 的时间点，假设此时设置了 ir_task0。t1、t2、t3…是一个 IR task 运行结束的时间点，同时也是下一组 IR task 开始的时间点。ts1、ts2、ts3、ts4 是可设置新的 IR task 的时间点。t0 是用 pwm start 启动 pwm IR mode 的时间点，pwm start 后，t0-t1 时间内运行 ir_task0。那么以下几种设置及对应的运行状态为：

1) ts0 设置 ir task0 后不再修改 PWM 的 setting，即不设定新的 IR task
那么 t0-t1、t1-t2、t2-t3、t3-t4、t4-t5 时间段内运行的都是 ir_task0，会一直运行下去，除非用 pwm stop 来停止
2) ts0 设置 ir task0，ts1 设置 ir_task1
此时 t0-t1 时间段内的 IR 是不受影响的，还是运行 ts0 设定的 ir_task0；等到 t1 时刻，运行 ir_task1；t2 时刻 ir_task1 结束，还是会运行最新的 ir_task1，后面 t3-t4、t4-t5 也是运行 ir_task1
3) ts0 设置 ir task0，ts1 设置 ir_task1，ts2 设置 ir_task2
t0-t1 运行 ir_task0，t1-t2 运行 ir_task1，t2-t3 运行 ir_task2，t3-t4、t4-t5 运行 ir_task2。

10.1.6 PWM start and stop

下面两个接口用于开启和停止某一路 PWM

```
static inline void pwm_start(pwm_id id)
static inline void pwm_stop(pwm_id id)
```


Lenze Technology Co.,LTD

10.1.7 PWM pulse number

static inline void pwm_set_pulse_num(pwm_id id, u16 pulse_num)用于设置指定的 pwm count mode 和 IR mode 时 pwm frame 持续的个数，对于 normal mode 无意义。由于只有 PWM0 和 PWM1 支持 count mode 和 ir mode，该接口对于 PWM2 ~ PWM5 无意义。

10.1.8 PWM phase

static inline pwm_set_phase(pwm_id id, u16 phase)用于设置 PWM 开始前的一个延时时间，一般不需要延时，设为 0。

10.1.9 PWM interrupt

PWM 所有的中断共 8 种，如下定义，我们简称为 IRQ_PWMn_PNUM（n= 0,1）、IRQ_PWMn_FRAME（n=0,1,2,3,4,5）

```
typedef enum{
    FLD_IRQ_PWM0_PNUM =      BIT(0),
    FLD_IRQ_PWM1_PNUM =      BIT(1),
    FLD_IRQ_PWM0_FRAME =     BIT(2),
    FLD_IRQ_PWM1_FRAME =     BIT(3),
    FLD_IRQ_PWM2_FRAME =     BIT(4),
    FLD_IRQ_PWM3_FRAME =     BIT(5),
    FLD_IRQ_PWM4_FRAME =     BIT(6),
    FLD_IRQ_PWM5_FRAME =     BIT(7),
}PWM_IRQ;
```

pwm_set_clk/ pwm_set_cycle_and_duty 配置好的一个 PWM 周期的时序称之为一个 frame，每个 frame 结束后会有产生一个中断 FLD_IRQ_PWMn_FRAME，PWM0 ~ PWM5 的 normal mode、count mode、IR mode 都会有这一类中断。

在 pwm frame 基础上，使用 pwm_set_pulse_num 设定 PWM 持续运行的 frame 的数量为一组 pwm frame（在 ir mode 里成为 IR task），每一组 pwm frame 结束后产生一个中断 FLD_IRQ_PWMn_PNUM，只有 PWM0 和 PWM1 的 count mode 和 ir mode 才会有这一类中断。

假设设定了一个 count mode 的 PWM：周期 1ms，占空比 1/2，pulse number 为 50：

pwm_set_clk(16000000, 16000000)

pwm_set_cycle_and_duty(PWM0_ID , 16000, 8000);

pwm_set_pulse_num(IR_PWM_ID, 50);

那么每隔 1ms 会产生一个 frame done 对应 FLD_IRQ_PWM0_FRAME，共 50 个；50 个 frame 都结束后会产生一个 FLD_IRQ_PWM0_PNUM。

上面所举例说明的产生的中断仅仅是硬件上产生的 irq status 或 irq flag，如果需要在 SDK 的中断入口 irq_hander 里处理的话，软件上还需要对中断的 mask 和 status 进行的相应的处理，下面以 IRQ_PWM0_PNUM 为例来说明。

Lenze Technology Co.,LTD

1. 打开 FLD_IRQ_PWM0_PNUM 的 mask
   reg_pwm_irq_mask |= FLD_IRQ_PWM0_PNUM;
   我们习惯在打开 mask 之前先清掉之前的 status，防止这个 mask 一打开就被之前的 status 触发进了 irq_handler：
   reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;
2. 打开 MCU 系统中断上的 PWM 总中断的 mask
   reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
3. 确保 MCU 总中断是打开的，即 irq_enable（）
4. 以上几个 setting 可以保证这个中断能够触发 MCU 进入 irq_handler()，在 irq_handler() 中的处理建议如下：

```
u8 pwm_sta = reg_pwm_irq_sta;
if(pwm_sta & FLD_IRQ_PWM0_PNUM){
    func();
    reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;
}
```

10.2 IR 实现方法

IR 的发送需要在特定的时间切换 PWM 的输出，而切换的时间的准确性要求比较高，误差稍微大一点就会引起 IR 的错误。

根据本文档 BLE 部分对 Link Layer 时序的介绍，Link Layer 使用了系统中断来处理 brx event（最新的 SDK 已经将 adv event 放到 mainloop 中处理，不再占用系统中断的时间）。如果 IR 某个切换 PWM 输出的时间点快要到来时，brx event 相关的中断先来了，这个中断会占用 MCU 的时间，可能会导致 PWM 输出的切换时间被延迟，IR 发生错误。

为了避免上述问题，IR 的实现使用 PWM 的 IR mode。参考本文档前面对 IR mode 的详细说明，BLE SDK 将一个 IR 数据切分成多个 ir task：ir_task0、ir_task1、ir_task2…….ir_task n-1 、ir_task n。

建立好 ir task 后，首先设置 ir_task0，pwm_start，此时 PWM 输出为 ir_task0，然后立刻设置 ir_task1；等到 ir_task0 结束时的 FLD_IRQ_PWMn_PNUM 中断发生后，在 irq_handler()里面设置 ir_task2，此时 PWM 输出的是 ir_task1；然后等到 ir_task1 结束时的 FLD_IRQ_PWMn_PNUM 中断发生后，在 irq_handler()里面设置 ir_task3，此时 PWM 输出的是 ir_task2…….。

如此依次设置，总是在提前一个 IR task 的时间前就设置好下一个 IR task，只要 MCU 系统中断运行的最大时间不超过最短的 IR task 的运行时间，就能保证 IR 的时序不会被延时。BLE SDK 里 brx event 里面的中断处理最大时间比较小，比一般的 IR task 的时间都短。

Lenze Technology Co.,LTD

10.3 IR Demo 详解

10.3.1 NEC IR

BLE SDK 上以 NEC IR 编号为 demo 来实现的。NEC IR 协议如下图所示。

图 10-2 NEC IR 协议

根据 NEC IR 具体的时序，使用 PWM 的 IR mode 来实现，IR mode 将完整的时序切割成多个 IR task，实际每个 IR task 只有 3 个参数：cycle、duty 和 pulse number，可以通过下面两个接口来完成：

static inline pwm_set_cycle_and_duty(pwm_id id, u16 cycle_tick, u16 cmp_tick)

static inline void pwm_set_pulse_num(pwm_id id, u16 pulse_num)

但在这个 BLE SDK 里，我们并不需要 user 自己去切割每个 IR task，而是用一套统一的处理方式，通过提供一些配置接口让用户来调用，然后根据 user 的配置，自动去完成一个 IR 的实现。

下面将详细介绍这套机制如何实现，user 只需要理解这套机制，然后根据这个方法去配置自己的 IR 即可。介绍过程中会具体说明哪些接口是客户只能调用不能修改的，哪些函数是需要客户自己去修改的。

IR 管理机制的核心在于下面这个结构体，后面会结合这个结构体的数据进行说明。

```
typedef struct{
    ir_send_ctrl_data_t  data[IR_GROUP_MAX];
    u8 group_index;
    u8 group_cnt;
    u8 is_sending;
    u8 repeat_timer_enable;
```

```
u8 ir_send_irq_idx;
u8 ir_send_start_high;
u8 last_cmd;
u8 rsvd;

u16 carrier_cycle;
u16 carrier_high;
u32 sending_start_time;
u32 repeat_time;
}ir_send_ctrl_t;
```

10.3.2 设置载波

NEC IR 载波频率为 38k，占空比为 1/3。

user 调用下面接口用来设置载波的周期和占空比，该接口只能调用而不能修改里面的实现方法。

void ir_config_carrier(u16 cycle_tick, u16 high_tick)

系统时钟为 CLOCK_SYS_CLOCK_HZ 时，

cycle_tick = CLOCK_SYS_CLOCK_HZ/38000

占空比为 1/3，只需要将 high_tick 设为 cycle_tick 的 1/3 即可。

从 code 上可以看出，user 设置的载波周期和占空比被存储到 carrier_cycle 和 carrier_high 两个变量。

10.3.3 设置 logic1 和 logic0 的时间

通过下面接口设置 IR 数据 bit 1 和 bit 0 的 PWM 持续时间。该接口只能调用而不能修改里面的实现方法。

void ir_config_byte_timing(u32 tick_logic_1_carr, u32 tick_logic_1_none,
u32 tick_logic_0_carr, u32 tick_logic_0_none)

tick_logic_1_carr 和 tick_logic_1_none 表示逻辑电平 1 的载波时间、无载波时间。tick_logic_0_carr 和 tick_logic_0_none 表示逻辑电平 0 的载波时间、无载波时间。

根据 NEC IR 协议，这四个值被设为：

```
ir_config_byte_timing( 560 * CLOCK_SYS_CLOCK_1US,
                       1690 * CLOCK_SYS_CLOCK_1US,
                       560 * CLOCK_SYS_CLOCK_1US,
                       560 * CLOCK_SYS_CLOCK_1US);
```

每个 IR task 只有 3 个参数：cycle、duty 和 pulse number，在 SDK 里使用如下的结构体表示，这个结构体用来描述一个 IR task。

```
typedef struct{
    u32 cycle;
```


Lenze Technology Co.,LTD

```
        u16 hich;
        u16 cnt;
    }ir_ctrl_t;
```

为 logic1 和 logic0 的有载波和无载波的时序分别创建 IR task 的数据：

```
ir_ctrl_t  ir_bit_1_controll[2];
ir_ctrl_t  ir_bit_0_controll[2];
```

ir_config_byte_timing(…)接口设置的 4 个参数实际被存储到上面的数组中。IR 发一个 byte 时（比如 0x55），发送的 8 个 bit 01010101，每个 bit 0 和 1 都会按照 ir_bit_1_controll 和 ir_bit_0_controll 预先算好的 IR task 的 3 个值来设置。user 只需要填入要发送的数据 0x55，SDK 就会根据自动将它拆分为 8 个对应的 IR task。

结合 ir_config_byte_timing(…)的具体实现 code，可以看出载波部分的时序，是根据 user 配置的载波参数，然后通过 tick_logic_1_carr 的持续时间除以载波周期得到 pulse number：

```
ir_bit_1_controll[0].cycle = ir_send_ctrl.carrier_cycle;
ir_bit_1_controll[0].hich = ir_send_ctrl.carrier_high;
ir_bit_1_controll[0].cnt=( tick_logic_1_carr)/
                            ir_send_ctrl.carrier_cycle;
```

PWM 无载波的时序也需要转换为一个 IR task，实际是一个周期等于持续时间，占空比为 0，pulse number 为 1 的 IR task。如果是周期比较大，需要考虑数据溢出问题的话，可以减小周期，加大 pulse number，比如将其等效为周期为持续时间除以 2，占空比为 0，pulse number 为 2 的 IR task。

```
ir_bit_1_controll[1].cycle = tick_logic_1_none;
ir_bit_1_controll[1].hich = 0;
ir_bit_1_controll[1].cnt = 1;
```

10.3.4 配置一个完整的 NEC IR

根据 NEC IR 的协议，如果要发送一个 cmd 比如 7，先要有 start（9ms 载波+4.5ms 无载波），然后是 address+ ~address+ cmd + ~cmd，demo code 中我们取 address 为 0x88。

当发送最后一个 ~cmd 的最后一个 bit 时，不管是 0 或 1 都有一段无载波的时间，如果~cmd 后面不再有任何数据，接收端可能会出现一个问题：由于没有载波的边界作为区分，不知道最后一个 bit 的无载波时间是 560us 还是 190us，导致不识别这是一个 0 还是 1。为了解决这个问题，我们自定义添加一段载波作为结束，取 563us 的载波时间。

另外一个问题是，我们采用了 PWM 的 IR mode，都是在提前一个周期就设好了下一个 IR task 的时序，如下图所示为 NEC IR 最后的 3 个 IR task，ir_task n-2 和 ir_task n-1 为 ~cmd 的最后一个 bit 的载波和无载波时间，ir_task n 是我们自定义的 563us 的载波时间作为结束。

Lenze Technology Co.,LTD

图 10-3 IR 结尾

实际软件上在 Tn-3 时刻(ir_task n-3 结束的 FLD_IRQ_PWM0_PNUM 中断)设置 ir_task n-1，在 Tn-2 时刻（ir_task n-2 结束的 FLD_IRQ_PWM0_PNUM 中断）设置 ir_task n，而到了 Tn-1 时刻（ir_task n-1 结束的 FLD_IRQ_PWM0_PNUM 中断）没有可以设置的 IR task 了。这时候如果等到 ir_task n 结束后的 FLD_IRQ_PWM0_PNUM 中断能够立刻响应的话，可以在用 pwm stop 来停止，但如果这个中断被 BLE 的中断延迟了一段时间，无法立刻响应的话，Tn 时刻 MCU 就会继续新一轮的 ir_task n，这个载波时间就不是我们预期的 563us，可能会变成 600us 或其他错误值。这里的 563us 是我们自定义的，时间有变化看起来也不会有太大的问题，但如果某个 IR 无载波在前载波在后或者其他特殊的格式，使最后一个 IR task 是有载波的，不需要我们加一个自定义的载波区分的话，这种情况下就有可能出现错误。

为了解决上述问题，我们约定，只要一个 IR 开始，不管最后一个 IR task 是否我们自定义的，都无条件的在结尾处再添加一个无载波的 IR task 作为结尾。还是以上图为例，添加一个 500us 的无载波 IR task 做结尾（时间可以随便设，不一定是 500us），即下图中的 ir_task n+1，这样在 Tn-1 时刻，就可以设置 ir_task n+1，到了 Tn 时刻，即使 FLD_IRQ_PWM0_PNUM 中断没有被及时响应，PWM 发送的也是一个无载波的信号，不影响任何时序，在 FLD_IRQ_PWM0_PNUM 中断里将 pwm stop 即可。

图 10-4 IR 尾部添加 end ir task

根据上面的描述，NEC 一个 IR 数据主要分为 3 部分：

1. start 信号，9ms 载波+4.5ms 无载波
2. 有效数据：address+ ~address+ cmd + ~cmd
3. stop 信号，自定义的 563us 载波 + 作为 end 的 560us 无载波

10.3.5 添加时序信号

NEC IR 中 start 和 stop 信号都属于时序信号，使用如下接口来添加，该接口只能调用而不能修改里面的实现方法。

void ir_send_add_series_item(u32 *time_series, u8 series_cnt, ir_ctrl_t *ir_control, u8 start_high)

time_series 和 series_cnt 为时序的描述，需要 user 定义，可以定义为 const 变量（存储在 flash 上）或者局部变量，用以节省 ram 空间。

NEC IR start 信号为：

```
const u32 ir_lead_times[] = {   9000 * CLOCK_SYS_CLOCK_1US,
                                4500 * CLOCK_SYS_CLOCK_1US};
```

ir_control 为 ir_ctrl_t 结构体，用来存储对应时序转化为 IR task 后的 cycle/duty/pulse number 等信号，需要 user 定义，必须定义为全局变量：

ir_ctrl_t nec_start[ARRAY_SIZE(ir_lead_times)];

start_high 为 1 表示 IR 发送时载波在前，为 0 表示载波在后。

Lenze Technology Co.,LTD

将 NEC IR start 信号配置为 IR task：
ir_send_add_series_item(ir_lead_times, ARRAY_SIZE(ir_lead_times), &nec_start, 1);
NEC IR stop 信号跟 start 信号的配置方法相同，不再详细介绍。

10.3.6 添加数据

使用下面接口添加以 byte 为单位的数据，该接口只能调用不能修改里面的实现方法。
void ir_send_add_byte_item(u8 code, u8 start_high)
code 为数据，start_high 为 1 表示 IR 发送时载波在前，为 0 表示载波在后。
NEC IR 数据为：address+ ~address+ cmd + ~cmd
若 address 为 0x88，cmd 为 7:

```
ir_send_add_byte_item(0x88, 1);
ir_send_add_byte_item(~0x88, 1);
ir_send_add_byte_item(0x07, 1);
ir_send_add_byte_item(~0x07, 1);
```

10.3.7 NEC IR send

如下图所示为 ir_nec_send 的实现：

```
void ir_nec_send(u8 addr1, u8 addr2, u8 cmd)
{

    if(ir_send_ctrl.last_cmd != cmd)
    {
        if(ir_sending_check())
        {
            return;
        }


        ir_send_ctrl.last_cmd = cmd;


        ir_send_ctrl_clear();

        ir_send_add_series_item(ir_lead_times, ARRAY_SIZE(ir_lead_times), &nec_start, 1);

        ir_send_add_byte_item(addr1, 1);
        ir_send_add_byte_item(addr2, 1);
        ir_send_add_byte_item(cmd, 1);
        ir_send_add_byte_item(~cmd, 1);

        ir_send_add_series_item(ir_stop_bit_times, ARRAY_SIZE(ir_stop_bit_times), &nec_stop, 1);

        ir_send_ctrl_start(1);
    }
}
```

图 10- 5 ir_nec_send

这个函数是 NEC IR 的参考实现，user 需要调用接口进行相关的配置和修改，以实现自己的 IR send。

int ir_sending_check(void)检查 PWM 是否还在发送之前的 IR，若还在发送，新的 IR 不处理。该接口只能调用而不能修改里面的实现方法。

void ir_send_ctrl_clear(void)接口在配置一个新的 IR 前必须调用。该接口只能调用而不能修改里面的实现方法。

添加时序信号和数据的接口上面已经介绍过。

时序和数据配置完成后，调用下面接口开始一个 IR 的发送。该接口只能调用而不能修改里面的实现方法。

void ir_send_ctrl_start(int need_repeat)

need_start 为 1 表示需要 repeat 信号。

10.3.8 NEC IR repeat

上面介绍了 NEC IR 数据的配置和发送，ir_send_ctrl_start（1）启动了 repeat 信号。repeat 信号的配置和发送类似于上面介绍的红外数据信号的配置/发送。

根据 NEC 协议，repeat 信号为 9ms 载波+2.25ms 无载波+560us 载波，加上 500us 的 end 信号，最终 repeat 信号定义为：

```
const u32 ir_repeat_times[]   = {   9000 * CLOCK_SYS_CLOCK_1US,
                                    2250 * CLOCK_SYS_CLOCK_1US,
                                     560 * CLOCK_SYS_CLOCK_1US,
                                     500 * CLOCK_SYS_CLOCK_1US};
```

下图所示的 repeat 信号发送函数跟上面数据的发送类似分析即可。user 需要调用接口进行相关的配置和修改，以实现自己的 IR send repeat。

```
void ir_nec_send_repeat(void)
{
    if(ir_sending_check()){
        return;
    }


    ir_send_ctrl_clear();
    ir_send_add_series_item((u32 *)ir_repeat_times, ARRAY_SIZE(ir_repeat_times), &nec_repeat, 1);
    ir_send_ctrl_start(1);
}
```

图 10-6 ir_nec_send_repeat

NEC IR repeat 信号每隔 110ms 发送一次，SDK 使用了硬件 Timer2 定时 108ms 实现，每到定时时间就添加一次 repeat 信号的发送即可。但按键松开需要结束红外时，关掉 Timer2。

10.3.9 中断的处理

demo code 上在 irq_handler 对 Timer2 和 FLD_IRQ_PWMn_PNUM 中断的处理，不需要修改，按照目前方式即可。

FLD_IRQ_PWMn_PNUM 的处理要注意硬件上选择了 PWM0 还是 PWM1，code 上通过如下宏来定义。

```
#define PWM0_IR_MODE                0
#define PWM1_IR_MODE                1


#define IR_PWM_SELECT                   PWM0_IR_MODE
```

void ir_irq_send(void)函数是根据用户对 IR 信号和数据的配置，逐个取出 ir_task0、ir_task1……ir_task n，完成所有的 IR task 后自动关闭 PWM。这个函数不能修改，user 可以根据本文档的介绍自行理解实现的方法。

void ir_repeat_handle(void)是每隔 108 ms 再次启动一个 IR repeat 信号。

10.3.10 应用层判断 IR busy

user 在应用层通过两个变量的检查来判断当前 IR 是否在发送数据或 repeat 信号：

ir_send_ctrl.is_sending

ir_send_ctrl.repeat_timer_enable

如下所示为 demo code 功耗管理中对 IR 是否 busy 的判断。当 IR busy 时，MCU 不能进 suspend。

```
if( ir_send_ctrl.is_sending ||ir_send_ctrl.repeat_timer_enable)
{
    bls_pm_setSuspendMask(SUSPEND_DISABLE);
}
```

11 附录

附录 1：crc16 算法

```
unsigned shortcrc16 (unsigned char *pD, int len)
{
    static unsigned short poly[2]={0, 0xa001};
    unsigned short crc = 0xffff;
    unsigned char ds;
    int i,j;

    for(j=len; j>0; j--)
    {
        unsigned char ds = *pD++;
        for(i=0; i<8; i++)
        {
            crc = (crc >> 1) ^ poly[(crc ^ ds ) & 1];
            ds = ds >> 1;
        }
    }

    return crc;
}
```


Lenze Technology Co., LTD

Lenze Technology Co.,LTD

Lenze Technology Co.,LTD