
应用文档:

泰凌 Kite BLE SDK 开发指南

AN-19011501-C5

Ver1.4.0

2019/9/26

简介:

本文档为泰凌微电子 Kite BLE SDK 3.4.0 的开发指南，适用于 8x5x 系列。



TELINK SEMICONDUCTOR

Published by

Telink Semiconductor

**Bldg 3, 1500 Zuchongzhi Rd,
Zhangjiang Hi-Tech Park, Shanghai, China**

© Telink Semiconductor

All Right Reserved

Legal Disclaimer

This document is provided as-is. Telink Semiconductor reserves the right to make improvements without further notice to this document or any products herein. This document may contain technical inaccuracies or typographical errors. Telink Semiconductor disclaims any and all liability for any errors, inaccuracies or incompleteness contained herein.

Copyright (c) 2019 Telink Semiconductor (Shanghai) Ltd, Co.

Information:

For further information on the technology, product and business term, please contact Telink Semiconductor Company (www.telink-semi.com).

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

telinknsales@telink-semi.com

telinknsupport@telink-semi.com

版本历史

版本	主要改动	日期	作者 & 标准化
1.0.0	初始版本	2019/1	WSH, FQH & Cynthia
1.1.0	2.1.4 SDK FLASH 空间的分配 , 2.3 GPIO 模块 , 3.5.4 SMP 绑定信息说明 , 4.1.1 低功耗模式 , 8.3 Keyscan flow , 8.6 卡键处理	2019/2	WSH, TYF, Cynthia
1.2.0	修订章节 : 1.4.2 BLE master demo , 1.4.3 feature demo 和 driver demo , 2.1.3.2 Flash 空间的操作 , 2.3.4 GPIO 数字状态在 deepsleep retention mode 失效 , 3.1.2.2 Telink BLE slave , 3.2.4.5 Conn state Slave role 时序 , 3.2.6 Link Layer TX fifo & RX fifo , 3.2.7 Controller Event , 3.2.8 Data Length Extension , 3.2.9.7 bls_ll_setAdvParam , 3.2.9.12 blc_ll_setScanParameter , 3.2.9.13 blc_ll_setScanEnable , 3.2.9.14 blc_ll_createConnection , 3.2.9.20 blc_ll_getCurrentState , 3.3.2 L2CAP , 3.3.3 ATT & GATT , 4.1.1 低功耗模式 , 新增章节 : 3.3.1 BLE host 介绍 , 3.3.4 SMP , 3.3.5 GAP , 12.2 32k 时钟源选择。	2019/5	TYF, WSH, FQH, Cynthia
1.3.0	根据 SDK 3.3.1 , 修订章节 : 1.1 软件组织架构 , 2.1.2 Sram 空间分配 , 4.2.6 API blc_pm_setDeepsleepRetentionType	2019/5	TYF, WSH, Cynthia

版本	主要改动	日期	作者 & 标准化
	新增章节： 1.2 适用 IC 介绍， 1.3 software bootloader 介绍		
1.4.0	修订章节： 1.3 software bootloader 介绍 2.1.2.1 Sram 和 Firmware 空间 3.2.3.1 Link Layer 状态机初始化 3.2.9.13 blc_ll_setScanEnable 3.3.3.3 Attribute PDU & GATT API 3.3.5.2 GAP event 7.1.1 FLASH 存储架构 7.1.3 修改 Firmware size 和 boot address 根据 SDK 3.4.0，新增章节： 3.2.10 Coded PHY/2M PHY 3.2.11 Channel Selection Algorithm#2 3.2.12 Extended Advertising 14 FreeRTOS SDK 说明及注意事项	2019/8	FQH,TYF, WSH, JF

目录

1	SDK 介绍.....	17
1.1	软件组织架构.....	17
1.1.1	main.c.....	18
1.1.2	app_config.h	19
1.1.3	application file.....	19
1.1.4	BLE stack entry	19
1.2	适用 IC 介绍	20
1.3	software bootloader 介绍	20
1.4	Demo 介绍.....	22
1.4.1	BLE slave demo	24
1.4.2	BLE master demo	24
1.4.3	feature demo 和 driver demo	25
2	MCU 基础模块	26
2.1	MCU 地址空间	26
2.1.1	MCU 地址空间分配	26
2.1.2	Sram 空间分配	27
2.1.2.1	Sram 和 Firmware 空间	27
2.1.2.2	list 文件分析 demo.....	34
2.1.3	MCU 地址空间访问	38
2.1.3.1	外设空间的读写操作	38
2.1.3.2	Flash 空间的操作	39
2.1.4	SDK FLASH 空间的分配	42
2.2	时钟模块.....	45
2.2.1	System clock & System Timer	45
2.2.2	System Timer 的使用	47
2.3	GPIO 模块	48
2.3.1	GPIO 定义	48

2.3.2	GPIO 状态控制	49
2.3.3	GPIO 的初始化	51
2.3.4	GPIO 数字状态在 deepsleep retention mode 失效	53
2.3.5	配置 SWS 上拉防止死机	53
3	BLE 模块	55
3.1	BLE SDK 软件架构	55
3.1.1	标准 BLE SDK 软件架构	55
3.1.2	Telink BLE SDK 软件架构	56
3.1.2.1	Telink BLE controller	56
3.1.2.2	Telink BLE slave	57
3.1.2.3	Telink BLE master	59
3.2	BLE controller	60
3.2.1	BLE controller 介绍	60
3.2.2	Link Layer 状态机	60
3.2.3	Link Layer 状态机组合应用	62
3.2.3.1	Link Layer 状态机初始化	62
3.2.3.2	Idle + Advtersing	63
3.2.3.3	Idle + Scannning	64
3.2.3.4	Idle + Advtersing + ConnSlaveRole	65
3.2.3.5	Idle + Scannning + Initiating + ConnMasterRole	66
3.2.4	Link Layer 时序	68
3.2.4.1	Idle state 时序	68
3.2.4.2	Advertising state 时序	69
3.2.4.3	Scanning state 时序	69
3.2.4.4	Initiating state 时序	70
3.2.4.5	Conn state Slave role 时序	71
3.2.4.6	Conn state Master role 时序	72
3.2.4.7	Conn state Slave role 时序保护	73
3.2.5	Link Layer 状态机扩展	74
3.2.5.1	Scanning in Advertising state	75

3.2.5.2	Scanning in ConnSlaveRole.....	75
3.2.5.3	Advertising in ConnSlaveRole.....	76
3.2.5.4	Advertising and Scanning in ConnSlaveRole.....	77
3.2.6	Link Layer TX fifo & RX fifo	77
3.2.7	Controller Event	81
3.2.7.1	Controller HCI Event	81
3.2.7.2	Telink defined event	88
3.2.8	Data Length Extension	97
3.2.9	Controller API.....	100
3.2.9.1	Controller API 说明	100
3.2.9.2	API 返回类型 ble_sts_t.....	101
3.2.9.3	BLE MAC address 初始化.....	101
3.2.9.4	Link Layer 状态机初始化.....	102
3.2.9.5	bls_ll_setAdvData.....	102
3.2.9.6	bls_ll_setScanRspData.....	103
3.2.9.7	bls_ll_setAdvParam	104
3.2.9.8	bls_ll_setAdvEnable	108
3.2.9.9	bls_ll_setAdvDuration	109
3.2.9.10	blc_ll_setAdvCustomedChannel	110
3.2.9.11	rf_set_power_level_index	110
3.2.9.12	blc_ll_setScanParameter.....	111
3.2.9.13	blc_ll_setScanEnable	113
3.2.9.14	blc_ll_createConnection	114
3.2.9.15	blc_ll_setCreateConnectionTimeout	116
3.2.9.16	blm_ll_updateConnection	116
3.2.9.17	bls_ll_terminateConnection	117
3.2.9.18	blm_ll_disconnect.....	117
3.2.9.19	Get Connection Parameters.....	118
3.2.9.20	blc_ll_getCurrentState	118
3.2.9.21	blc_ll_getLatestAvgRSSI	119
3.2.9.22	Whitelist & Resolvinglist	119

3.2.10	Coded PHY/2M PHY	120
3.2.10.1	Coded PHY/2M PHY 介绍	120
3.2.10.2	Coded PHY/2M PHY Demo 介绍	120
3.2.10.3	Coded PHY/2M PHY API 介绍:	121
3.2.11	Channel Selection Algorithm #2	122
3.2.12	Extended Advertising	122
3.2.12.1	Extended Advertising 介绍	122
3.2.12.2	Extended Advertising Demo 搭建	123
3.2.12.3	Extended Advertising 相关的 API 介绍	124
3.3	BLE host	127
3.3.1	BLE host 介绍	127
3.3.2	L2CAP	127
3.3.2.1	注册 L2CAP 数据处理函数	128
3.3.2.2	更新连接参数	129
3.3.3	ATT & GATT	134
3.3.3.1	GATT 基本单位 Attribute	134
3.3.3.2	Attribute and ATT Table	135
3.3.3.3	Attribute PDU & GATT API	145
3.3.3.4	GATT Service Security	157
3.3.3.5	8258 master GATT	159
3.3.4	SMP	161
3.3.4.1	SMP 安全等级	161
3.3.4.2	SMP 参数配置	162
3.3.4.3	SMP 安全请求配置	168
3.3.4.4	SMP 绑定信息说明	170
3.3.4.5	master SMP	174
3.3.5	GAP	180
3.3.5.1	GAP 初始化	180
3.3.5.2	GAP event	181
4	低功耗管理 (PM)	188

4.1	低功耗 driver.....	188
4.1.1	低功耗模式.....	188
4.1.2	低功耗唤醒源.....	191
4.1.3	低功耗模式的进入和唤醒.....	192
4.1.4	低功耗唤醒后运行流程.....	195
4.1.5	API pm_is_MCU_deepRetentionWakeup.....	198
4.2	BLE 低功耗管理.....	198
4.2.1	BLE PM 初始化	198
4.2.2	BLE PM for Link Layer.....	198
4.2.3	相关变量.....	201
4.2.4	API bls_pm_setSuspendMask.....	201
4.2.5	API bls_pm_setWakeupSource.....	203
4.2.6	API blc_pm_setDeepsleepRetentionType	204
4.2.7	PM 软件处理流程	204
	4.2.7.1 blt_sdk_main_loop.....	205
	4.2.7.2 blt_brx_sleep.....	206
4.2.8	deepsleep retention 的详细分析.....	208
	4.2.8.1 API blc_pm_setDeepsleepRetentionThreshold.....	209
	4.2.8.2 blc_pm_setDeepsleepRetentionEarlyWakeupTiming	213
	4.2.8.3 T_init 的优化和测量.....	214
4.2.9	Connection Latency	220
	4.2.9.1 Connection latency 生效时的 Sleep 时序	220
	4.2.9.2 latency_use 的计算.....	221
4.2.10	API bls_pm_getSystemWakeupTick.....	222
4.3	GPIO 唤醒的注意事项	223
4.3.1	唤醒电平有效时无法进入 sleep mode.....	223
4.4	BLE 系统低功耗管理参考.....	225
4.5	应用层定时唤醒.....	227
5	低电检测	228

5.1	低电检测的重要性.....	228
5.2	低电检测的实现.....	229
5.2.1	低电检测的注意事项.....	229
5.2.1.1	必须使用 GPIO 输入通道	229
5.2.1.2	只能使用差分模式	231
5.2.1.3	必须使用 Dfifo 模式获得 ADC 采样值.....	231
5.2.1.4	不同的 ADC 任务需要切换	232
5.2.2	低电检测单独使用.....	232
5.2.2.1	低电检测初始化	232
5.2.2.2	低电检测处理	235
5.2.2.3	低压报警	236
5.2.2.4	低电检测 debug 模式	238
5.2.3	低电检测和 Amic Audio	238
6	Audio.....	240
6.1	Audio 初始化	240
6.1.1	Amic 和低电检测	240
6.1.2	Amic 初始化设置	240
6.1.3	Dmic 初始化设置	241
6.2	Audio 数据处理	242
6.2.1	Audio 数据量和 RF 传送方法	242
6.2.2	Audio 数据压缩	243
6.3	压缩与解压缩算法.....	245
7	OTA	247
7.1	Flash 架构设计和 OTA 流程	247
7.1.1	FLASH 存储架构	247
7.1.2	OTA 更新流程.....	248
7.1.3	修改 Firmware size 和 boot address.....	250
7.2	OTA 模式 RF 数据处理.....	251
7.2.1	Slave 端 Attribute Table 中 OTA 的处理.....	251

7.2.2	OTA 数据 packet 格式.....	251
7.2.3	master 端 RF transform 处理方法	253
7.2.4	slave 端 RF receive 处理方法.....	256
8	按键扫描	260
8.1	键盘矩阵.....	260
8.2	Keyscan、keymap.....	262
8.2.1	Keyscan	262
8.2.2	Keymap & kb_event.....	263
8.3	Keyscan flow	265
8.4	Deepsleep 唤醒快速扫键（wake_up fast keyscan）	267
8.5	Repeat Key 处理	269
8.6	卡键处理.....	270
9	LED 管理	273
9.1	LED 任务相关调用函数	273
9.2	LED 任务的配置和管理	273
9.2.1	定义 led event	273
9.2.2	Led event 的优先级.....	274
10	Blt 软件定时器（Software Timer）	276
10.1	timer 初始化.....	276
10.2	Timer 的查询处理	277
10.3	添加定时器任务.....	279
10.4	删除定时器任务.....	279
10.5	Demo.....	280
11	IR.....	282
11.1	PWM Driver.....	282
11.1.1	PWM id 和管脚.....	282
11.1.2	PWM 时钟	283
11.1.3	PWM 周期（cycle）和占空比（duty）	284

11.1.4	PWM 波形取反	285
11.1.5	PWM 开启和停止	285
11.1.6	PWM 模式	285
11.1.7	PWM 脉冲数 (pulse number)	286
11.1.8	PWM 中断	286
11.1.9	PWM phase	289
11.1.10	API for IR DMA FIFO mode	289
11.1.10.1	DMA FIFO 的配置	289
11.1.10.2	设置 DMA FIFO buffer	290
11.1.10.3	IR DMA FIFO mode 的开启与停止	290
11.2	IR Demo	291
11.2.1	PWM 模式的选择	291
11.2.2	Demo IR 协议	292
11.2.3	IR 时序设计	292
11.2.4	IR 初始化	295
11.2.4.1	rc_ir_init	295
11.2.4.2	IR 硬件配置	295
11.2.4.3	IR 变量初始化	296
11.2.5	FifoTask 的配置	296
11.2.5.1	FifoTask_data	296
11.2.5.2	FifoTask_idle	298
11.2.5.3	FifoTask_repeat	298
11.2.5.4	FifoTask_repeat*n & FifoTask_idle_repeat*n	299
11.2.6	应用层判断 IR busy	299
12	其他模块	300
12.1	24M 晶体外部电容	300
12.2	32k 时钟源选择	301
12.3	PA	301
12.4	PhyTest	302

12.4.1	PhyTest API	302
12.4.2	PhyTest demo	303
12.4.2.1	Demo1: 8258_feature_test.....	303
12.4.2.2	Demo2: 8258_ble_remote.....	304
12.4.2.3	PhyTest 参数调整	305
12.5	EMI	305
12.5.1	EMI Test	305
12.5.1.1	Emi 初始化设置	306
12.5.1.2	Power level（功率）和 Channel（频点）	306
12.5.1.3	Emi Carrier Only（单载波）	307
12.5.1.4	emi_con_prbs9	307
12.5.1.5	Emi TX Burst	308
12.5.1.6	EMI RX.....	308
12.5.1.7	上位机配置参数设置	309
12.5.2	EMI Test Tool	310
13	附录	315
13.1	附录 1: crc16 算法.....	315
14	FreeRTOS SDK 说明及注意事项	316
14.1	freeRTOS 主要参数	316
	系统时钟	316
	Tick Rate	316
14.2	系统初始化.....	316
14.3	创建任务.....	316

图目录

图 1-1 SDK 文件结构	17
图 1-2 不同 IC 对应的 bootloader 以及 boot.link 路径.....	20
图 1-3 software bootloader 设置	22
图 1-4 BLE SDK 提供的 demo code	23
图 2-1 MCU 地址空间分配	26
图 2-2 各 IC 在 16k 和 32k retention 对应的 Sram 空间分配	27
图 2-3 Sram 空间分配& Firmware 空间分配	28
图 2-4 list 文件 section 统计	35
图 2-5 list 文件 section 地址	36
图 2-6 512K FLASH 地址分配	43
图 2-7 system clock & System Timer.....	45
图 3-1 BLE SDK 标准架构	55
图 3-2 Host 和 Controller 的 HCI 数据交互	56
图 3-3 8258 hci 架构.....	57
图 3-4 Telink BLE slave 架构	58
图 3-5 Telink BLE master 架构	59
图 3-6 State diagram of the Link Layer state machine in BLE Spec.....	61
图 3-7 Telink Link Layer state machine	61
图 3-8 Idle + Advertising	63
图 3-9 Idle + Scanning	64
图 3-10 BLE slave LL state	65
图 3-11 BLE master LL state	67
图 3-12 Advertising State 时序.....	69
图 3-13 Scanning state 时序	69
图 3-14 Initiating state 时序	70
图 3-15 Conn state Slave role 时序	71
图 3-16 ConnMasterRole 时序	72
图 3-17 Scanning in Advertising state 时序.....	75
图 3-18 Scanning in ConnSlaveRole 时序	76

图 3-19 Advertising in ConnSlaveRole 时序	76
图 3-20 Advertising and Scanning in ConnSlaveRole 时序	77
图 3-21 RX overflow 图示 1	78
图 3-22 RX overflow 图示 2	79
图 3-23 BLE SDK event 架构	81
图 3-24 HCI event	82
图 3-25 Disconnection Complete Event	83
图 3-26 Read Remote Version Information Complete Event	84
图 3-27 LE Connection Complete Event	85
图 3-28 LE Advertising Report Event	85
图 3-29 LE Connection Update Complete Event	86
图 3-30 连接请求包单元 PDU	92
图 3-31 BLE 协议栈 LL_CONNECTION_UPDATE_REQ 格式	96
图 3-32 BLE 协议栈广播包格式	102
图 3-33 BLE 协议栈里 Advertising Event	104
图 3-34 BLE 协议栈四种广播事件	105
图 3-35 BLE L2CAP 结构以及 ATT 组包模型	128
图 3-36 BLE 协议栈中 Connection Para update Req 格式	129
图 3-37 抓包显示 conn para update request 和 response	130
图 3-38 BLE 协议栈中 conn para update rsp 格式	131
图 3-39 抓包显示 ll conn update req	133
图 3-40 Attribute 构成 GATT service	134
图 3-41 该 BLE SDK Attribute Table 截图	136
图 3-42 master 读 hidInformation 的 BLE 抓包	140
图 3-43 BLE 协议栈中 Write Request	141
图 3-44 BLE 协议栈中 Write Command	141
图 3-45 BLE 协议栈中 Execute Write Request	141
图 3-46 Service/Attribute Layout	144
图 3-47 Read by Group Type Request/Read by Group Type Response	146
图 3-48 Find by Type Value Request/Find by Type Value Response	147
图 3-49 Read by Type Request/Read by Type Response	147

图 3-50 Find information request/Find information response.....	148
图 3-51 Read Request/Read Response	149
图 3-52 Read Blob Request/Read Blob Response.....	149
图 3-53 Exchange MTU Request/Exchange MTU Response	150
图 3-54 Write Request/Write Response	152
图 3-55 Write Long Characteristic Values 示例	153
图 3-56 BLE Spec 中 Handle Value Notification.....	153
图 3-57 BLE Spec 中 Handle Value Indication.....	155
图 3-58 BLE Spec 中 Handle Value Confirmation	156
图 3-59 服务请求响应映射关系	157
图 3-60 ATT Permission 定义.....	158
图 3-61 本地设备配对状态	161
图 3-62 抓包显示 Pairing Disable	162
图 3-63 传统配对模式下 MITM、OOB flag 使用规则	165
图 3-64 根据不同 IO 能力映射 KEY 产生方法.....	165
图 3-65 抓包显示 Pairing Peer Trigger.....	170
图 3-66 抓包显示 Pairing Conn Trigger.....	170
图 3-67 master 发起 Pairing_Req.....	182
图 4-1 8x5x MCU 硬件唤醒源.....	191
图 4-2 sleep mode wakeup work flow	195
图 4-3 sleep timing for Advertising state & Conn state Slave role.....	199
图 4-4 suspend & deepsleep retention timing & power	211
图 4-5 T_init timing.....	215
图 4-6 sleep timing for conn_latency valid	221
图 4-7 Early wake_up at app_wakup_tick	227
图 6-1 audio 数据抓包	242
图 6-2 MIC service in Attribute Table	243
图 6-3 数据压缩处理	244
图 6-4 压缩算法对应数据	246
图 7-1 Flash 存储结构	247
图 7-2 BLE 协议栈 Write Command 格式.....	251

图 7-3 OTA 命令和数据的格式	252
图 7-4 master 通过 Read By Type Request 获取 OTA 的 Attribute Handle	253
图 7-5 firmware 示例--开头部分	254
图 7-6 firmware 示例--结尾部分	254
图 7-7 master 发 OTA start	254
图 7-8 master OTA 数据	256
图 8-1 行列式键盘结构	260
图 11-1 PWM cycle & duty	284
图 11-2 PWM interrupt	287
图 11-3 DMA FIFO buffer for IR DMA FIFO mode	289
图 11-4 demo IR 协议	292
图 11-5 IR timing 1	293
图 11-6 IR timing 2	294
图 12-1 24M 晶体电路	300
图 12-2 EMI test tool	310
图 12-3 选择芯片型号	310
图 12-4 选择数据总线	311
图 12-5 Swire 同步操作	311
图 12-6 set channel	312
图 12-7 选择 RF 模式	312
图 12-8 set RF 模式显示界面	313
图 12-9 选择测试模式	313
图 12-10 设置 TX packet number	314
图 12-11 RX packet number 和 RSSI	314

1 SDK 介绍

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

1.1 软件组织架构

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

在 Telink IDE 中导入 sdk 工程后，显示的文件组织结构如下图所示。顶层文件夹有 7 个：application，boot，common，drivers，proj_lib，stack，vendor。

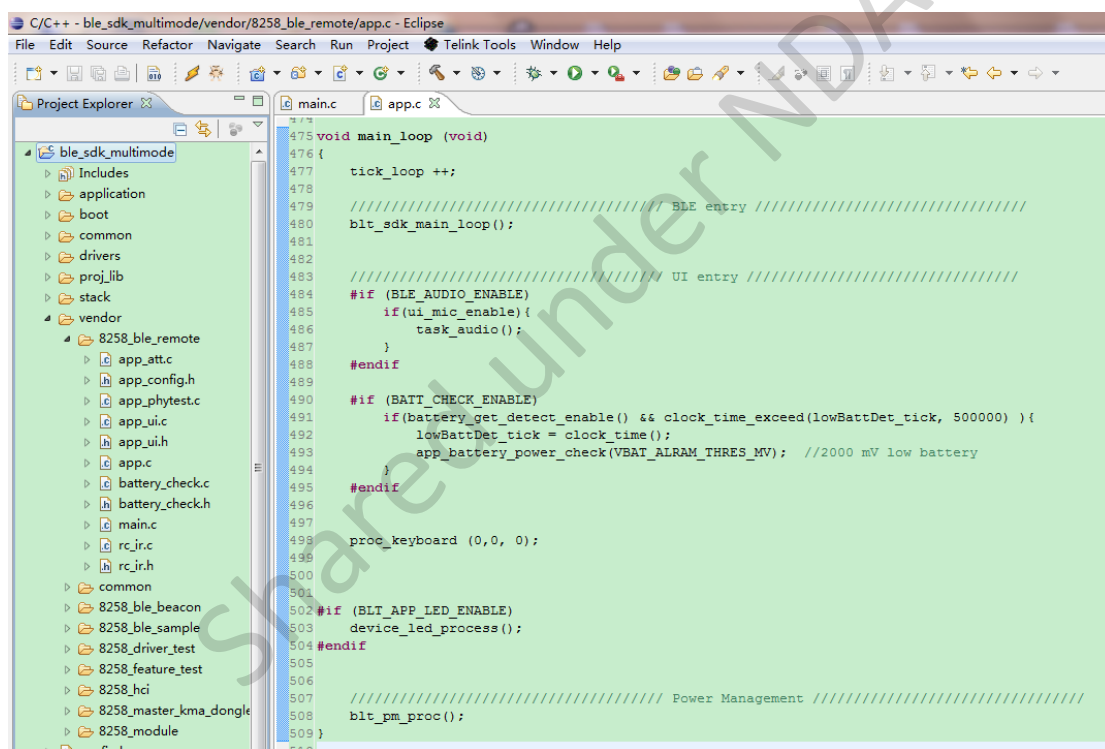


图 1-1 SDK 文件结构

- ✧ application: 提供一些通用的应用处理程序，如 print、keyboard 等。
- ✧ boot: 提供芯片的 software bootloader，即 MCU 上电启动或 deepsleep 唤醒后的汇编处理过程，为后面 C 语言程序的运行搭建好环境。
- ✧ common: 提供一些通用的跨平台的处理函数，如内存处理函数、字符串处理函数等。
- ✧ drivers: 提供与 MCU 紧密相关的硬件设置和外设驱动程序，如 clock、flash、i2c、usb、gpio、uart 等。

- ✧ **proj_lib**: 存放 SDK 运行所必需的库文件（如 liblt_8258.a）。BLE 协议栈、RF 驱动、PM 驱动等文件，被封装在库文件里，用户无法看到源文件。
- ✧ **stack**: 存放 BLE 协议栈相关的头文件。源文件被编译到库文件里面，对于用户是不可见的。
- ✧ **vendor**: 用于存放用户应用层代码。

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(); //MCU 最基本的硬件初始化
    int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();
                                     //判断是否从 deep retention 唤醒
    rf_drv_init(RF_MODE_BLE_1M); //RF 初始化
    gpio_init(!deepRetWakeUp); //gpio 初始化，user 在 app_config.h 中配置相关参数

    if( deepRetWakeUp ){
        user_init_deepRetn (); //deep retention 醒来的快速初始化
    }
    else{
        user_init_normal (); //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)
            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

app.c: 用户主文件，用于完成 BLE 系统初始化、数据处理、低功耗处理等。

BLE slave 工程的 app_att.c: service 和 profile 的配置文件，有 Telink 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。

其他 UI 文件：如 IR（红外）、battery detect（电池检测）等用户任务的处理文件。

1.1.4 BLE stack entry

Telink 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 main_loop 中 BLE 逻辑和数据处理的函数入口

blt_sdk_main_loop。

```
void main_loop (void)
{
    //////////////////////////////////// BLE entry ////////////////////////////////////
    blt_sdk_main_loop();

    //////////////////////////////////// UI entry ////////////////////////////////////
    .....

    //////////////////////////////////// PM configuration ////////////////////////////////////
    .....
}
```

1.2 适用 IC 介绍

适用如下几种 IC 型号，它们均属于 8x5x 系列，8251/8253/8258 核是同一个，硬件模块基本一致，只是在 SRAM size 方面略有差异。

IC	Flash size	SRAM size
8251	512 kB	32 kB
8253	512 kB	48 kB
8258	512 kB	64 kB

因为上面三颗 IC 的差异主要是 SRAM size，其他部分是一致的，SDK 文件架构除了 SDK/boot/启动脚本（即 software bootloader 文件）和 boot.link 文件有差异外，其他部分完全共用。

1.3 software bootloader 介绍

三种类型 IC 的 software bootloader 并不共用，文件存放在 SDK/boot/目录下：

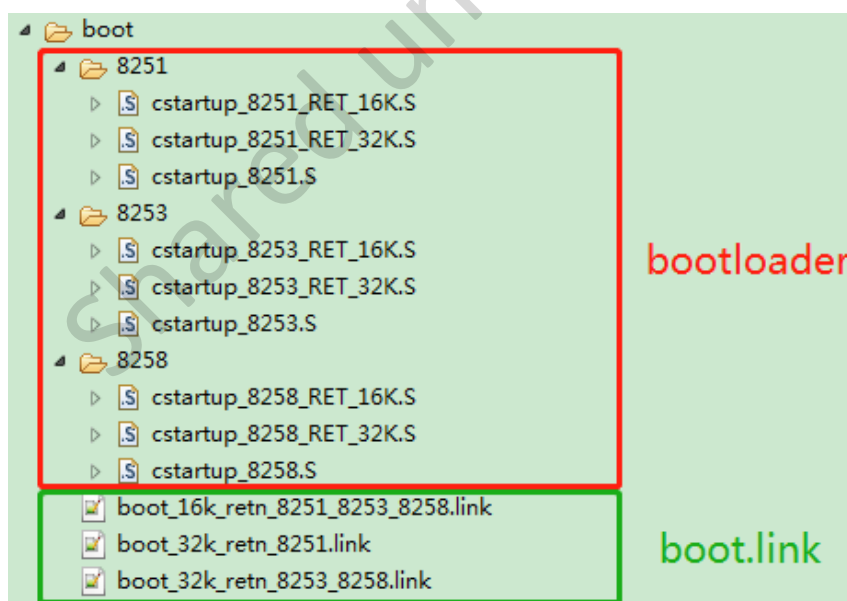


图 1-2 不同 IC 对应的 bootloader 以及 boot.link 路径

每种 IC 对应 3 个 software bootloader 文件，分别对应启用 16k deep retention、32k deep retention 和不启用 deep retention 功能（deep retention 的介绍可以参考“低功耗管理”章节）。

以 `cstartup_8258_RET_16K.S` 为例，第一句 `#ifdef`
`MCU_STARTUP_8258_RET_16K` 说明了只有当 `user` 定义了
`MCU_STARTUP_8258_RET_16K` 时，该 `bootloader` 才会生效。

用户可以根据实际使用的 IC 以及是否使用 `deep retention`（16K 或者 32k）功能选择不同的 `software bootloader`。

Kite BLE SDK 里的工程默认使用的配置是 Sram size 64K 的 8258 , `deepsleep retention 16K sram`,即对应使用的 `software bootloader` 和 `link` 文件分别为 `cstartup_8258_RET_16K.S` 和 `boot_16k_retn_8251_8253_8258.link`，用户需要根据自己使用的芯片类型，Retention 大小的评估情况手动修改其配置（详细分析方法可参考小节“Sram 空间”）。

以 `8258_ble_remote` 为例说明如何选择 8258 的 `software bootloader` 改为 `deepsleep retention 32K sram`。

1. `user` 可以在 `8258_ble_remote` 工程设置中定义 `-DMCU_STARTUP_8258_RET_32K` 即可，如下图所示。

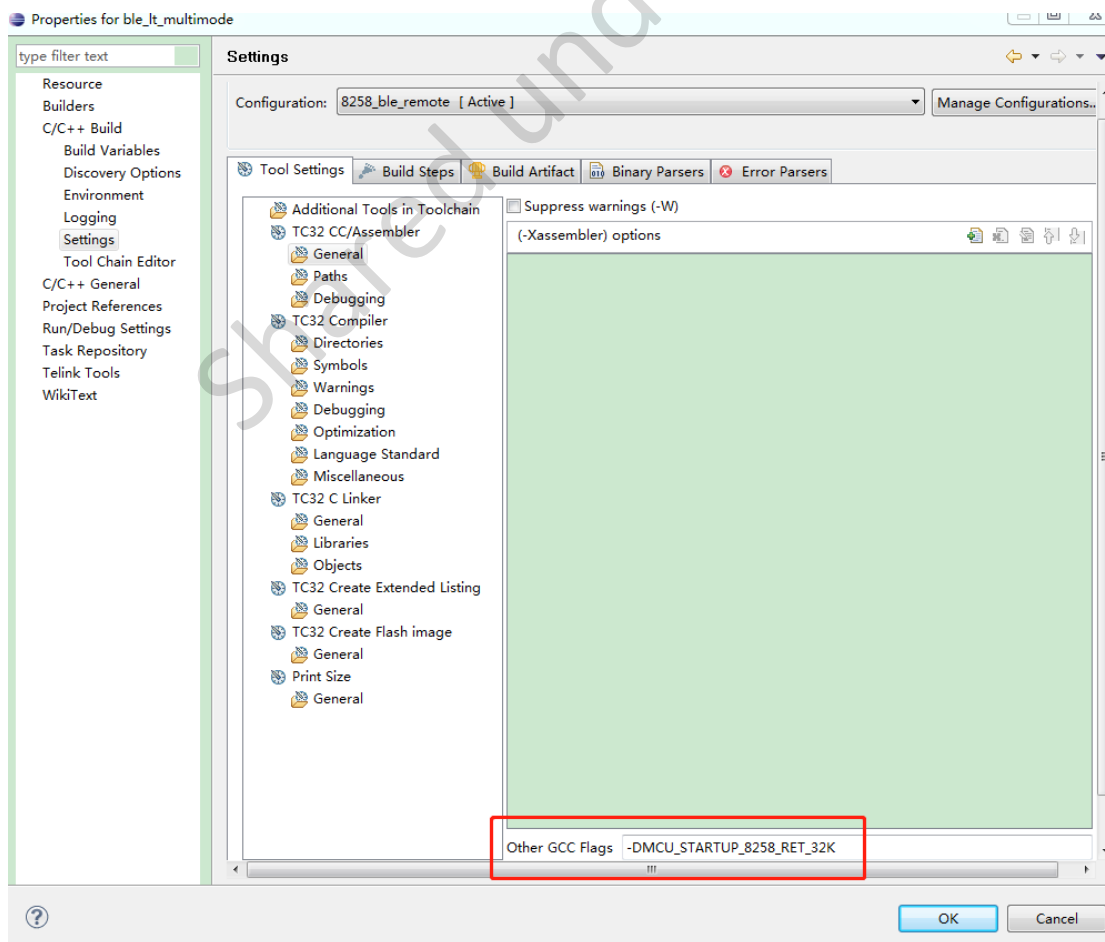


图 1-3software bootloader 设置

注意：根据前面介绍我们知道 8251、8253 和 8258 的 Sram size 是不一样的，所以，实际用户在选择不同的 software bootloader 文件后还需要修改 SDK 根目录下的 boot.link 文件（根据下表对应关系将其中的 link 文件中的内容替换到 SDK 根目录下的 boot.link 中），不同 IC 的 software bootloader 以及 boot.link 选用关系见下表。

Retention type IC	16kB retention	32kB retention
8251	boot_16k_retn_8251_8253_8258.link cstartup_8251_RET_16K.S	boot_32k_retn_8251.link cstartup_8251_RET_32K.S
8253	boot_16k_retn_8251_8253_8258.link cstartup_8253_RET_16K.S	boot_32k_retn_8253_8258.link cstartup_8253_RET_32K.S
8258	boot_16k_retn_8251_8253_8258.link cstartup_8258_RET_16K.S	boot_32k_retn_8253_8258.link cstartup_8258_RET_32K.S

- 根据上面的例子以及映射表，我们知道 software bootloader 文件为 cstartup_8258_RET_32K.S，需要选择将 SDK/boot/boot_32k_retn_8253_8258.link 文件内容替换到 SDK 根目录下的 boot.link 文件中。
- 在 API use_init() 中 blc_ll_initPowerManagement_module() 后调用 API blc_pm_setDeepSleepRetentionType(DEEPSLEEP_MODE_RET_SRAM_LOW32K) 用于设置硬件的 Retention 区域。

1.4 Demo 介绍

TeLink BLE SDK 给用户提供了多个 BLE demo。

用户通过软硬件 demo 的运行，可以观察到直观的效果。用户也可以在 demo code 上进行修改，完成自己的应用开发。

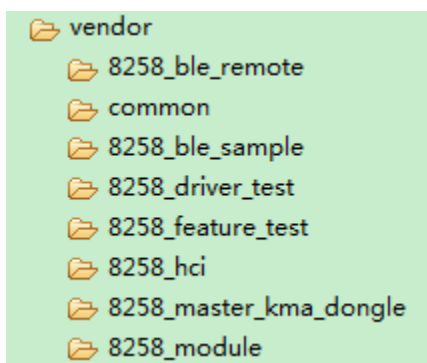


图 1-4 BLE SDK 提供的 demo code

1.4.1 BLE slave demo

BLE slave 的 demo 及区别如下表所示。

Demo	Stack	Application	MCU function
8258 hci	BLE controller	No	Controller, HCI 接口与其他 MCU host 通信
8258 module	BLE controller + host	Application 在主控 MCU 上	BLE 透传模组
8258 ble remote	BLE controller + host	遥控器应用	主控 MCU
8258 ble sample	BLE controller + host	最简单的 slave demo, 只有广播和连接功能	主控 MCU

8258 hci 是一个 BLE slave controller, 提供了基于 USB/UART 的 HCI, 和其他 MCU 的 host 通信, 形成一个完整的 BLE slave 系统。

8258 ble remote/8258 module 都是 Telink 提供的完整的 BLE slave stack。8258 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。

8258 ble sample 是对 8258_ble_remote 的简化, 可以和标准的 IOS/android 设备配对连接。

1.4.2 BLE master demo

8258 master kma dongle 是 BLE master single connection 的 demo, 可以和 8258 ble sample/8258 ble remote/8258 module 连接并通信。

8258 ble remote/8258 ble sample 对应的 library 提供了标准的 BLE stack(master 和 slave 共用一个 library), 包含了 BLE controller + BLE host, 用户只需要在 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。

新 SDK 的 library 将 slave 和 master 库合二为一了, 8258 master kma dongle 编译 code 的时候只会调用库中标准的 BLE controller 功能部分, library 中并没有提供 master 的标准 host 功能。8258 master kma dongle demo code 在 app 层上给出了参考的 BLE Host 的实现方法, 包括 ATT、简单的 SDP (service discovery protocol) 和最常用的 SMP (security management protocol) 等。

BLE master 最复杂的功能在于对 slave server 的 service discovery 和所有 service 的识别，一般是在 android/linux 系统中才能实现。Telink 8258 IC 由于 flash size 和 Sram size 的限制，无法提供完整的 service discovery。但是 SDK 中提供了所有 service discovery 需要用到的 ATT 接口，用户可以参考 8258 master kma dongle 对 8258 ble remote 的 service discovery 过程，去实现自己的特定 service 的遍历。

1.4.3 feature demo 和 driver demo

8258_feature_test 针对一些常用的 BLE 相关的 feature 给出了 demo code，用户可参考这些 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。

在 8258_feature_test 工程里面 app_config.h 中对宏 “FEATURE_TEST_MODE” 进行选择性的定义，即可切换到不同 feature test 的 Demo。

8258 driver test 对基本的 driver 给出了 sample code，供用户参考并实现自己的 driver 功能。本文档 driver 部分会详细介绍各个 driver。

在 8258 driver test 工程里面 app_config.h 中对宏 “DRIVER_TEST_MODE” 进行选择性的定义，即可切换到不同 driver test 的 Demo。

2 MCU 基础模块

2.1 MCU 地址空间

2.1.1 MCU 地址空间分配

以典型的 64K Sram 版本为例，介绍 MCU 地址空间分配。如下图所示。

Telink 8x5x MCU 的最大寻址空间为 16M bytes:

- ✧ 从 0 到 0x7FFFFFFF 的 8M 空间为程序空间，即最大程序容量为 8M bytes。
- ✧ 0x800000 到 0xFFFFFFFF 为外部设备空间：0x800000~0x80FFFF 为寄存器空间；0x840000~0x84FFFF 为 64K Sram 空间。

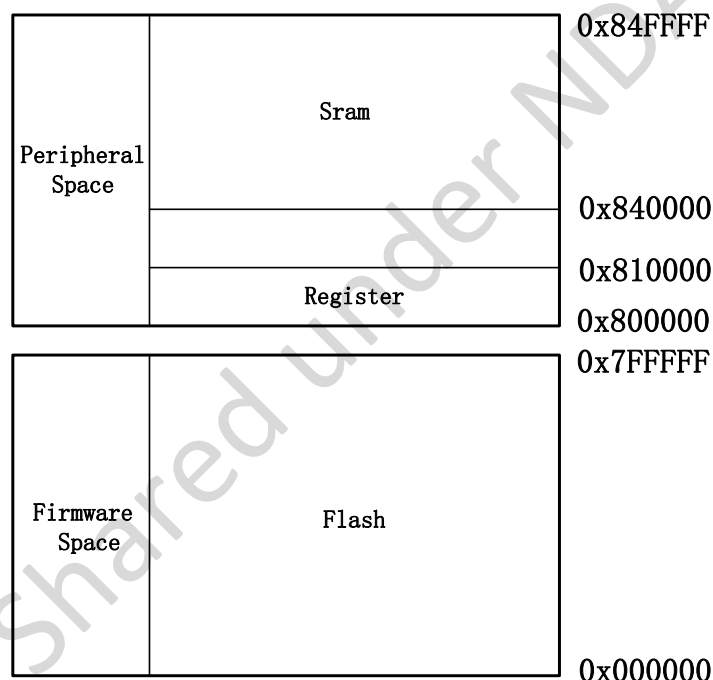


图 2-1 MCU 地址空间分配

8x5x 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（18）用于区别 Register 和 Sram：

- ✧ 该地址为 0 时，访问 Register；
- ✧ 该地址为 1 时，访问 Sram。

2.1.2 Sram 空间分配

8x5x Sram 空间分配和低功耗管理部分的 deepsleep retention 功能密切相关，请用户先掌握了解 deepsleep retention 相关知识。

如果不使用 deepsleep retention 功能，只用到 suspend 和普通的 deepsleep 功能，8x5x Sram 空间分配可以和 Telink 上一代 BLE IC 826x 系列一样。用过 826x BLE SDK 的用户可先参考《826x BLE SDK handbook》中 Sram 空间分配的介绍，再和本节要介绍的 8x5x Sram 空间分配做一些对比，这样可以加深对这一块的理解。

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。

32kB Sram 地址空间范围为 0x840000 ~ 0x848000，48kB Sram 地址空间范围为 0x840000 ~ 0x84C000，64kB Sram 地址空间范围为 0x840000 ~ 0x850000。

下图是 8258、8253、8251 在 16k retention 和 32k retention 模式下对应的 SRAM 空间分配说明。需要注意的是 IC 为 8251 且使用 deepsleep retention 32K sram 模式时，其 sram 空间分配的各个段是动态调整的，具体可以参考对应的 software bootloader 和 link 文件。

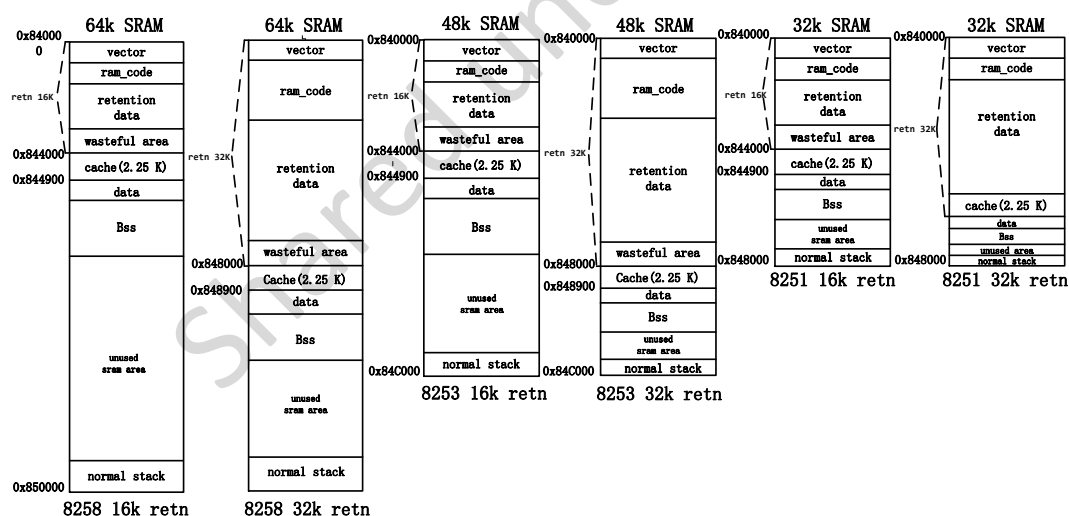


图 2-2 各 IC 在 16k 和 32k retention 对应的 Sram 空间分配

下面以 Sram size 64K 的 IC: 8258、SDK 中默认的 deepsleep retention 16K sram 模式为例详细介绍 Sram 区域各个部分。如果 Sram size 是其他值，或者 deepsleep retention 32k sram 模式，用户可以类推一下即可。

64k Sram 对应的 Sram 和 Firmware 空间分配如下图所示。

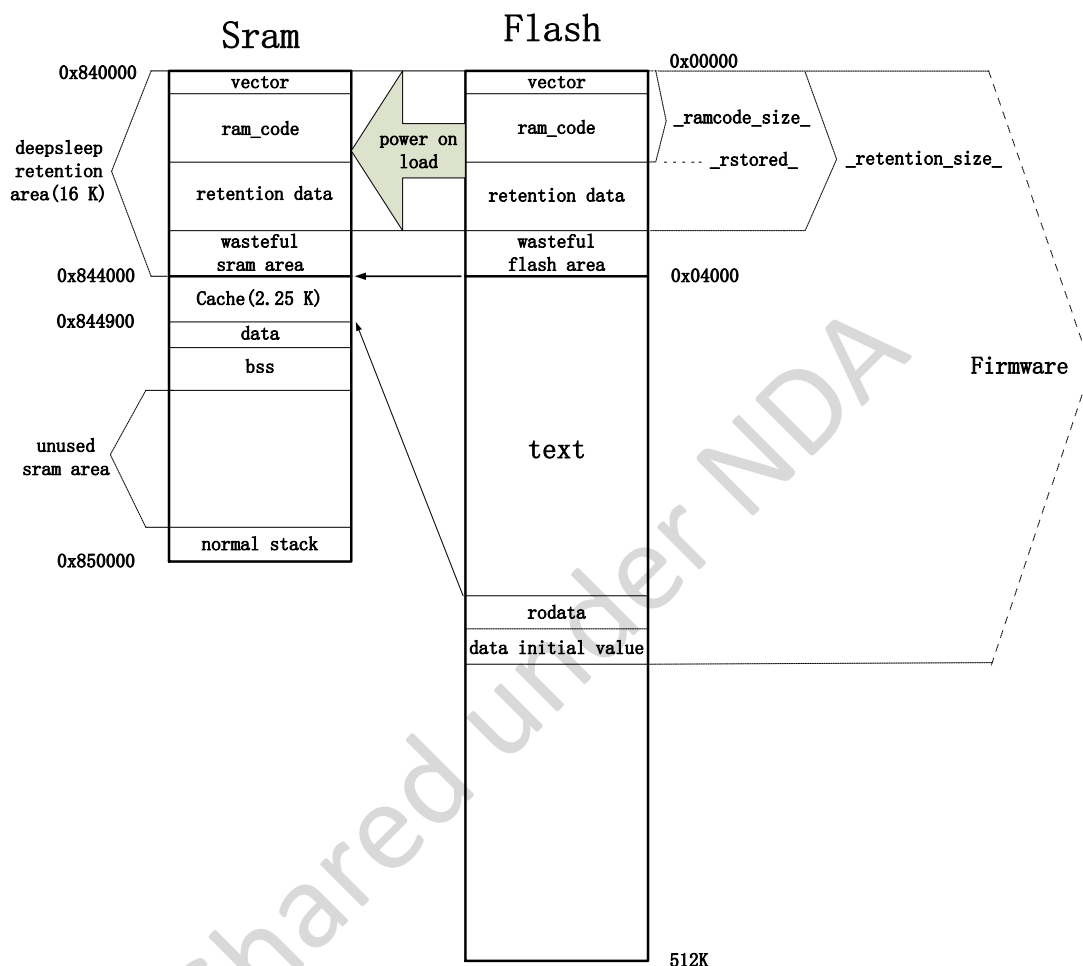


图 2-3 Sram 空间分配& Firmware 空间分配

SDK 中 Sram 空间分配相关的文件有 boot.link (由“software bootloader 介绍”小节我们可知这里的 boot.link 内容同 boot_16k_retn_8251_8253_8258.link 文件) 和 cstartup_8258_RET_16K.S。 (如果使用 deepsleep retention 32K Sram，则 bootloader 对应 cstartup_8258_RET_32K.S，link 文件对应 boot_32k_retn_8253_8258.link。)

Flash 中 Firmware 包括 vector、ramcode、retention_data、text、Rodata 和 Data initial value。

Sram 中包括 vector、ramcode、retention_data、Cache、data、bss、stack 和 unused sram area。

Sram 中的 vector/ramcode/ retention_data 是 Flash 中 vector/ramcode/ retention_data 的拷贝。

1) vectors、ram_code

“vectors”段是汇编文件 cstartup_8258_RET_16K.S 对应的程序，是软件启动代码（software bootloader）。

“ramcode”段是 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_sector 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，最终达到节省功耗。

用户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)
```

2) retention_data

8258 的 deepsleep retention mode 支持 MCU 进入 deepsleep 后, Sram 的前 16K/32K 可以不掉电保持住 Sram 上的数据不丢失。

程序中的全局变量如果直接编译的话, 会分配在"data"段或者"bss"段, 这两段的内容不在前 16K 的 retention 区域, 进入 deepsleep 会掉电丢失。

如果希望一些特定的变量在 deepsleep (deepsleep retention mode) 期间能够不掉电保存, 只要将它们分配到"retention_data"段即可, 方法是在定义变量时添加关键字"_attribute_data_retention_". 以下为几个示例:

```
_attribute_data_retention_ int AA;
_attribute_data_retention_ unsigned int BB = 0x05;
_attribute_data_retention_ int CC[4];
_attribute_data_retention_ unsigned int DD[4] = {0,1,2,3};
```

参考下面即将要介绍的 "data/bss" 段可知, data 段的全局变量对应的 initial value 需要提前存放在 flash 上; bss 段的变量 initial value 为 0, 无需提前准备, bootloader 运行时直接在 sram 上设为 0 即可。

但"retention_data"段的全局变量不管 initial value 是否为 0, 会无条件准备好它们的 initial value, 存放在 flash 的 retention_data area 上。上电 (或 normal deepsleep 唤醒) 后会整体拷贝到 sram 的 retention_data area 上。

"retention_data"段是紧跟着"ram_code"段的, 即"vector + ramcode + retention_data" 3 段按顺序排布在 flash 的前面, 它们的总大小为 "_retention_size_". MCU 上电 (deepsleep wake_up) 后"vector + ramcode + retention_data"作为一个整体拷贝到 sram 的前面, 此后程序执行过程中只要不进 deepsleep (只有 suspend/deepsleep retention), 这一整块的内容就一直保持在 sram 上, MCU 无须再从 flash 上读取。

boot.link 文件中 retention_data 段相关的配置如下。

```
. = (0x840000 + (_rstored_));
.retention_data :
    AT ( _rstored_ )
    {
        . = (((. + 3) / 4)*4);
        PROVIDE(_retention_data_start_ = . );
        *(.retention_data)
        *(.retention_data.*)
        PROVIDE(_retention_data_end_ = . );
    }
```

以上配置的含义为：编译的时候看到含有“`retention_data`”关键字的变量，在 flash firmware 中分布的起始地址为“`_rstored_`”，在 Sram 上对应的地址为 $0x840000 + (_rstored_)$ 。而“`_rstored_`”这个值就是“ram_code” section 的结尾。

如果用户选择的配置使用的是 deepsleep retention 16K Sram mode，但定义的“`_retention_size_`”超过所定义的 16K，编译时会出现如下图所示的错误。

```
C:\TelinkSDK1.3\opt\tc32\bin\tc32-elf-ld.exe: section .text loaded at [00004000,0000ab8b] overlaps section .retention_data loaded at [000037a0,00004fd7]
make: *** [ble_1t_multimode.elf] Error 1
```

用户可以通过下面的一种方式修改错误：

1. 减少所定义的“`_attribute_data_retention_`”属性的数据。
2. 选择切换为 deepsleep retention 32K Sram,详细配置方法参考[小节 1.3](#)

当“`_retention_size_`”不超过 16K 时（假设为 12K），flash 上存在一片 4K 的“wasteful flash area”（无效 flash 区域），对应的 firmware binary file 上可以看到 12K~16K 的内容全部是无效的“0”，拷贝到 Sram 后，Sram 上也有 4K 的“wasteful sram area”（无效 SRAM 区域）。

如果用户不希望浪费太多的 flash/sram，可以适当增加自己的 ram_code 和 retention_data，将之前不在 ram_code/retention_data 的 function（函数）/variable（变量）通过添加对应的关键字切换到 ram_code/retention_data 中。放在 ram_code 中的 function 可以节省运行时间以降低功耗，放在 retention_data 中的 variable 也可以节省初始化时间以降低功耗（具体原因请参考低功耗管理部分的介绍）。

3) Cache

Cache 是 MCU 的指令高速缓存，且必须被配置为 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 Cache 中之前的 code，再从 Cache 中读取指令执行。

图 2-3 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，它在 Sram 中的起始地址为可配置的，这里配置到 Sram 16K retention area 后面，即起始地址为 0x844000，结束地址为 0x844900。

Shared under NDA

4) data / bss

“data”段是 Sram 中存放程序已经初始化的全局变量，即 initial value 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 initial value 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 0x844900。下面为 boot.link 中的代码，直接定义 Sram 上 data 段开始的地址：

```
. = 0x844900;

.data :
```

“data”段是被初始化的全局变量，其初值需要提前存储在 flash 中，即图 2-3 所示 Firmware 中的“data initial value”。

5) stack / unused area

对于默认 64K 的 Sram，“stack”是从最高地址 0x850000（48K Sram 对应地址为 0x84C000，32K Sram 对应地址为 0x848000）开始的，其方向为从下往上延伸，即 stack 指针 SP 在数据入栈时自减，数据出栈时自加。

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 的使用量跟用户上层的程序设计是有关的。如果用户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。

当用户的 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，只能让应用程序运行起来，确保其运行了程序中所有的可能使用 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，用户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。

6) text

“text”段是 Flash firmware 中所有非 ram_code 函数的集合。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。一般情况下，“text”段是 Firmware

中最大的空间，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 中再可以被执行。

7) rodata/data init value

Firmware 中除了 vector、ram_code 和 text，剩余的数据为“rodata”段和“data initial value”。

“rodata”段是程序中定义的可读、不能改写的的数据，是用关键字“const”定义的变量。比如 Slave 中的 ATT table:

```
const attribute_t my_Attributes[] = .....
```

用户可以在对应的 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 时，会将该初值拷贝到 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 8258 ble sample 为例，结合“Sram 空间分配& Firmware 空间分配”图来分析。

8258 ble sample 的 bin 文件和 list 文件见目录“SDK”->“Demo”->“list file analyze”。

以下分析中，会出现多处截图，均来自 boot.link、cstartup_8258_RET_16K.S、8258 ble sample.bin 和 8258 ble sample.list，请用户自行查找文件找到截图对应位置。

list 文件中各个 section 的分布情况如下图所示（注意 Algn 字节对齐）:

Sections:					
Idx	Name	Size	VMA	LMA	File off
0	.vectors	00000170	00000000	00000000	00008000
	CONTENTS, ALLOC, LOAD, READONLY, CODE				
1	.ram_code	000023f0	00000170	00000170	00008170
	CONTENTS, ALLOC, LOAD, READONLY, CODE				
2	.text	0000614c	00004000	00004000	0000c000
	CONTENTS, ALLOC, LOAD, READONLY, CODE				
3	.rodata	000008ec	0000a14c	0000a14c	0001214c
	CONTENTS, ALLOC, LOAD, READONLY, DATA				
4	.retention_data	00000ce8	00842560	00002560	0000a560
	CONTENTS, ALLOC, LOAD, DATA				
5	.data	0000002c	00844900	0000aa38	00014900
	CONTENTS, ALLOC, LOAD, DATA				
6	.bss	00000259	00844930	0000aa68	0001492c
	ALLOC				
7	.TC32.attributes	00000010	00000000	00000000	0001492c
	CONTENTS, READONLY				
8	.comment	0000001a	00000000	00000000	0001493c
	CONTENTS, READONLY				

图 2-4 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 0x170, 计算出结束地址为 0x170;
- 2) ram_code: 从 Flash 0x170 开始, Size 为 0x23f0, 计算出结束地址为 0x2560;
- 3) retention_data: 从 Flash 0x2560 开始, Size 为 0xce8, 计算出结束地址为 0x3248;
- 4) text: 从 Flash 0x4000 开始, Size 为 0x614c, 计算出结束地址为 0xa14c;
- 5) rodata: 从 Flash 0xa14c 开始, Size 为 0x8ec, 计算出结束地址为 0xaa38;
- 6) data: 从 Sram 0x844900 开始, Size 为 0x2c, 计算出结束地址为 0x84492c;
- 7) bss: 从 Sram 0x844930 开始, Size 为 0x259, 计算出结束地址为 0x844b89。

结合前面介绍可知, 剩余 Sram 空间为 $0x850000 - 0x844b89 = 0xb477 = 46199$ byte, 减去 stack 需要使用的 256 byte, 还剩 45943 byte。

```

Disassembly of section .vectors:

00000000 <__start>:
-----
Disassembly of section .ram_code:

00000170 <blt_packet_crc24_opt>:
-----
Disassembly of section .text:

00004000 <__modsi3>:
-----
Disassembly of section .rodata:

0000a14c <C.1.4443-0x8>:
-----
Disassembly of section .retention_data:

00842560 <_retention_data_start_>:
-----
Disassembly of section .data:

00844900 <_start_data_>:
-----
Disassembly of section .bss:

00844930 <_start_bss_>:
-----
00844b88 <blt_dma_tx_rptr>:
...
-----
00002560 g      *ABS*  00000000 _rstored_

```

图 2-5 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”段在 flash firmware 中起始地址为 0，结束地址为 0x170（最后一笔数据地址为 0x16e~0x16f），size 为 0x170。上电搬移到 Sram 后，在 Sram 上的地址为 0x840000 ~ 0x840170。

2) ram_code

“ram_code” section 起始地址为 0x170，结束地址为 0x2560（最后一笔数据地址为 0x255c~0x255f）。上电搬移到 Sram 后，在 Sram 上的地址为 0x840170 ~ 0x842560。

3) retention_data

“retention_data”在 flash 中起始地址 “_rstored_” 为 0x2560，也是“ram_code”的结尾。

“retention_data”在 Sram 中起始地址为 0x842560，结束地址为 0x843248（最后一笔数据地址为 0x843244~0x843247）。

“vector+ram_code+retention_data”总的 size 值“_retention_size_”为 0x3248，那么 flash firmware 中前 16K 只有 0x3248 byte 有效数据，从 0x3248 到 0x4000 大约 3.43K 的空间属于“wasteful flash area（无效 flash 区域）”（用户可以打开 8258_ble_sample.bin 看到这段空间中全部是无效的 0）；Sram 中 0x843248~0x844000 大约 3.43K 的空间属于“wasteful sram area”（无效 SRAM 区域）。

4) Cache

Cache 在 Sram 中地址范围为：0x844000~0x844900。

Cache 的相关信息在 list 文件中不会体现出来。

5) text

“text”段在 flash firmware 中起始地址为 0x4000，结束地址为 0xa14c（最后一笔数据地址为 0xa148~0xa14b），Size 为 0xa14c- 0x4000 = 0x614c，和前面 Section 统计中数据一致。

6) rodata

“rodata”段起始地址为 text 的结束地址 0xa14c，结束地址为 0xaa38（最后一笔数据地址为 0xaa34~0xaa37）。

7) data

“data”段在 Sram 上起始地址为 Cache 的结束地址 0x844900，list 文件 Section 统计部分给出的 size 为 0x2c。

“data”段在 Sram 上的结束地址为 0x84492c（最后一笔数据地址为 0x844928~0x84492b）。

8) bss

“bss”段在 Sram 上起始地址为“data”段的结束地址 0x844930(16 字节对齐)，list 文件 Section 统计部分给出的 size 为 0x259。

“bss”段在 Sram 上的结束地址为 0x844b89（最后一笔数据地址为 0x844b84~0x844b88）。

剩余 Sram 空间为 $0x850000 - 0x844b89 = 0xb477 = 46199$ byte，减去 stack 需要使用的 256 byte，还剩 45943 byte。

2.1.3 MCU 地址空间访问

程序中对 0x000000 - 0xFFFFFFFF 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*)0x840000; //读 sram 0x40000-0x40003 地址的值
*(volatile u32*)0x840000 = 0x12345678; //给 sram 0x40000-0x40003 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，其实是指针操作。更多信息，请参照 drivers/8258/bsp.h。

注意程序中类似 write_reg8(0x40000)/ read_reg16(0x40000)的操作，其定义如下所示，可以看到是自动加上了 0x800000 的偏移（地址线 BIT(23)为 1），所以 MCU 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*)0x840001; //0x840001 没有 2 字节对齐
```

```
*(volatile u32*)0x840005 = 0x12345678; //0x840005 没有 4 字节对齐
```

修改为正确的读写操作：

```
u16 x = *(volatile u16*)0x840000; //0x840000 2 字节对齐
```

```
*(volatile u32*)0x840004 = 0x12345678; //0x840004 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，flash 的擦除使用 flash_erase_sector 函数。

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 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，时间就越长。所以强烈建议用户在 `main_loop` 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 `main_loop` 里，一旦进入 BLE 连接状态，不允许去调用 `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

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){ ..... }
```

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。

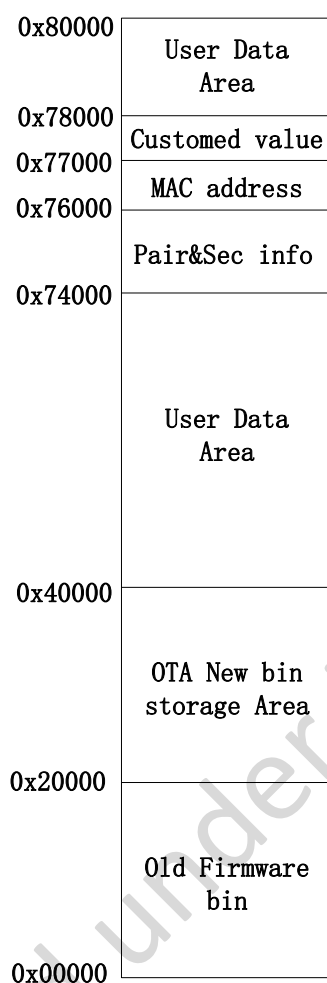


图 2-6 512K FLASH 地址分配

8x5x 使用的 Flash 为 512K，上图所示为 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。

泰凌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 stack/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 telink 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 4096 bytes 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：因为 Telink 上一代 IC RF 需要做 TP 值校准，所以这个位置被用来存储 TP 值校准值。而 8x5x IC 上 RF 采用新一代的设计后，不再需要 TP 校准这个步骤，所以完全不用再考虑 TP 的问题，但这个位置仍然沿用了上一代 IC 的设计。
 - 3) 第三个 64bytes 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。
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，可以调用下面函数修改配对加密信息存储的起始地址：

```
void bls_smp_configParingSecurityInfoStorageAddr (int addr);
```
4. 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，协议栈也提供了相应的 API，修改方法见后面 OTA 章节的详细说明。
5. 剩余的 FLASH 空间全部作为 user 的数据存储空间。

2.2 时钟模块

2.2.1 System clock & System Timer

系统时钟（system clock）是 MCU 执行程序的时钟。

系统定时器（System Timer）是一个只读的定时器，为 BLE 的时序控制提供时间基准，同时也可以提供给用户使用。

在 Telink 上一代 IC (826x 系列) 上，System Timer 的时钟来源于 system clock，而 8x5x IC 上，System Timer 和 system clock 是独立分开的。如下图所示，System Timer 是由外部 24M Crystal Oscillator 经 3/2 分频得到的 16M。

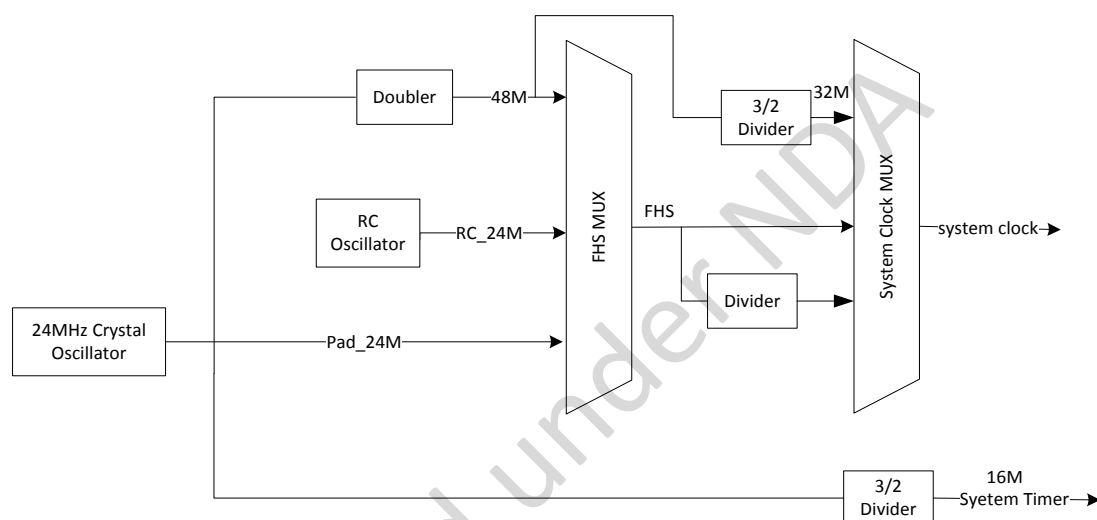


图 2-7 system clock & System Timer

图上可以看到，system clock 可以由外部 24M 晶体振荡器经“doubler”电路倍频到 48M 后再分频得到 16M、24M、32M、48M 等，这一类 clock 我们称为 crystal clock（如 16M crystal system clock、24M crystal system clock）；也可以由 IC 内部 24M RC Oscillator 处理后得到 24M RC clock、32M RC clock、48M RC clock 等。这一类我们称为 RC clock（BLE SDK 不支持 RC clock）。

在 BLE SDK 中，我们推荐使用 crystal clock。

初始化时调用下面的 API 配置 system clock，在枚举变量 SYS_CLK_TYPEDEF 定义中选择时钟对应的 clock 即可。

```
void clock_init(SYS_CLK_TYPEDEF SYS_CLK)
```

由于 8x5x System Timer 与 system clock 不一样，用户需要了解 MCU 上各硬件模块的 clock 是来源于 system clock 还是 System Timer。我们以 system clock 为 24M crystal 的情况来进行说明，此时 system clock 为 24M，而 System Timer 是 16M。

在文件 app_config.h 中 system clock 以及对应的 S、mS、uS 的定义如下：

```
#define CLOCK_SYS_CLOCK_HZ 24000000

enum{
    CLOCK_SYS_CLOCK_1S = CLOCK_SYS_CLOCK_HZ,
    CLOCK_SYS_CLOCK_1MS = (CLOCK_SYS_CLOCK_1S / 1000),
    CLOCK_SYS_CLOCK_1US = (CLOCK_SYS_CLOCK_1S / 1000000),
};
```

所有时钟源为 system clock 的硬件模块，在设置模块的 clock 时，只能使用上面 CLOCK_SYS_CLOCK_HZ、CLOCK_SYS_CLOCK_1S 等；换言之，如果用户看到模块中 clock 的设置使用的是以上几个定义，说明该模块的时钟源为 system clock。

如 PWM 驱动中 PWM 周期和占空的设置如下，说明 PWM 的时钟源是 system clock。

```
pwm_set_cycle_and_duty(PWM0_ID, (u16) (1000 * CLOCK_SYS_CLOCK_1US),
(u16) (500 * CLOCK_SYS_CLOCK_1US) );
```

System Timer 是固定的 16M，所以对于这个 timer，SDK code 中使用如下的数值来表示 S、mS 和 uS。

```
//system timer clock source is constant 16M, never change
enum{
    CLOCK_16M_SYS_TIMER_CLK_1S = 16000000,
    CLOCK_16M_SYS_TIMER_CLK_1MS = 16000,
    CLOCK_16M_SYS_TIMER_CLK_1US = 16,
};
```

SDK 中以下几个 API 都是跟 System Timer 相关的一些操作，所以涉及到这些 API 操作时，都使用上面类似“CLOCK_16M_SYS_TIMER_CLK_xxx”的方式来表示时间。

```
void sleep_us (unsigned long microsec);
unsigned int clock_time(void);

int clock_time_exceed(unsigned int ref, unsigned int span_us);

#define ClockTime          clock_time
#define WaitUs              sleep_us
#define WaitMs(t)           sleep_us((t)*1000)
```

由于 System Timer 是 BLE 计时的基准，SDK 中所有 BLE 时间相关的参数和变量，在涉及到时间的表达时，都是用“CLOCK_16M_SYS_TIMER_CLK_xxx”的方式。

2.2.2 System Timer 的使用

Main 函数中 `cpu_wakeup_init` 初始化完成后，System Timer 就开始工作，用户可以读取 System Timer 计数器的值（简称 System Timer tick）。

System Timer tick 每一个时钟周期加一，其长度为 32bit，即每 1/16 us 加 1，最小值 0x00000000，最大值 0xffffffff。System Timer 刚启动的时候，tick 值为 0，到最大值 0xffffffff 需要的时间为： $(1/16) \text{ us} * (2^{32})$ 约等于 268 S，每过 268 S System Timer tick 转一圈。

MCU 在运行程序过程中 system tick 不会停止。

System Timer tick 的读取可以通过 `clock_time()` 函数获得：

```
u32 current_tick = clock_time();
```

该 BLE SDK 整个 BLE 时序都是基于 System Timer tick 设计的，程序中也大量使用这个 System Timer tick 来完成各种计时和超时判断，强烈推荐 user 使用这个 System Timer 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 Timer tick。

```
u32 start_tick = clock_time(); // clock_time()返回 System Timer tick 值。
```

- 2) 在程序的某处不断查询当前 System Timer 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，查询计时是否到达的写法为：

```
if( (u32)( clock_time() - start_tick) > 100 * CLOCK\_16M\_SYS\_TIMER\_CLK\_1MS )
```

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。

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
```

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。如果超过 268 S，需要在软件上加计数器累计实现（这里不介绍）。

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

2.3 GPIO 模块

GPIO 模块的说明请 user 对照 drivers/8258/gpio_8258.h、gpio_default_8258.h、gpio_8258.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《gpio_lookuptable》来理解。

2.3.1 GPIO 定义

8258 系列芯片共有 5 组 36 个 GPIO，分别为：

GPIO_PA0 - GPIO_PA7、GPIO_PB0 - GPIO_PB7、GPIO_PC0 - GPIO_PC7

GPIO_PD0 - GPIO_PD7、GPIO_PE0 - GPIO_PE3

注意：IC 的 core 部分都是有这 36 个 GPIO，但具体到每一颗 IC 的不同封装上，可能有些 GPIO 并没有封出来。所以使用 GPIO 时请以 IC 实际封装出来的 GPIO 管脚为准。

程序中需要使用 GPIO 时，必须按照上面的写法定义，详情见 drivers/8258/gpio_8258.h。

7 个 GPIO 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。这个 4 个 GPIO 为 PE0、PE1、PE2、PE3。
- 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。8x5x 的 SWS 管脚为 PA7。
- 3) DM 和 DP，上电默认为 GPIO 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。8x5x 的 DM、DP 管脚为 PA5、PA6。

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。

- 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(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func);
```

pin 为 GPIO 定义，以下一样。func 可选择 AS_GPIO 或其他特殊功能。

- 2) ie(input enable): 输入使能

```
void gpio_set_input_en(GPIO_PinTypeDef pin, unsigned int value);
```

value: 1 和 0 分别表示 enable 和 disable。

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

```
unsigned int gpio_read(GPIO_PinTypeDef 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。

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(GPIO_PinTypeDef pin, unsigned int 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(GPIO_PinTypeDef pin, unsigned int 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( GPIO_PinTypeDef gpio,
                                  GPIO_PullTypeDef up_down);
```

up_down 的四种配置为:

```
typedef enum {
    PM_PIN_UP_DOWN_FLOAT    = 0,
    PM_PIN_PULLUP_1M        = 1,
    PM_PIN_PULLDOWN_100K    = 2,
    PM_PIN_PULLUP_10K       = 3,
}GPIO_PullTypeDef;
```

在 deepsleep 和 deepsleep retention 状态下, GPIO 的输入输出状态全部失效, 但是模拟上下拉电阻仍然有效。

GPIO 配置应用举例:

- 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
    . . . . .
}
```

- 3) 将 PA5、PA6 脚配置成 USB 功能。

```
gpio_set_func(GPIO_PA5, AS_USB );
gpio_set_func(GPIO_PA6, AS_USB );
gpio_set_input_en(GPIO_PA5, 1);
gpio_set_input_en(GPIO_PA6, 1);
```

2.3.3 GPIO 的初始化

main.c 中调用 `gpio_init` 函数，会将除了 MSPI 4 个 GPIO 以外的其他 32 个 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。32 个 GPIO 默认状态为：

- 1) `func`

除了 SWS，其他均为一般 GPIO 状态。

- 2) `ie`

除了 SWS 默认 `ie` 为 1，其他所有的一般 GPIO 默认 `ie` 为 0。

- 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`。

更多详情请参照 `drivers/8258/gpio_8258.h`、`drivers/8258/gpio_default_8258.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 PA0 的 ie 为例）：

```
#ifndef PA0_INPUT_ENABLE
#define PA0_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 PA0 为例）：

- 1) 配置 func: `#define PA0_FUNC AS_GPIO`
- 2) 配置 ie: `#define PA0_INPUT_ENABLE 1`
- 3) 配置 oe: `#define PA0_OUTPUT_ENABLE 0`
- 4) 配置 dataO: `#define PA0_DATA_OUT 0`
- 5) 配置内部上下拉电阻:
`#define PULL_WAKEUP_SRC_PA0 PM_PIN_UP_DOWN_FLOAT`

GPIO 的初始化总结：

- 1) 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；
- 2) 可以在 `user_init` 函数中通过 GPIO 状态控制函数(`gpio_set_input_en` 等)加以设定；
- 3) 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。

注意：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。

`gpio_init` 函数的实现如下，`anaRes_init_en` 的值决定模拟上下拉电阻是否被设置。

```
void gpio_init(int anaRes_init_en)
{
    // gpio digital status setting
    if(anaRes_init_en){
        gpio_analog_resistance_init();
    }
}
```

参考文档后面低功耗管理的介绍可知，控制 GPIO 模拟上下拉电阻的寄存器在 deepsleep retention 期间是可以保持不掉电的，所以 GPIO 模拟上下拉电阻的状态能在 deepsleep retention mode 下被维持住。

为了确保 deepsleep retention 唤醒之后 GPIO 模拟上下拉电阻的状态不被改变，在 gpio_init 前要先判断当前是否被 deepsleep retention wake_up，根据这个状态去设置 anaRes_init_en 的值，如下面的 code 所示：

```
int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();  
  
gpio_init( !deepRetWakeUp );
```

2.3.4 GPIO 数字状态在 deepsleep retention mode 失效

上面介绍的 GPIO 状态控制中，除了模拟上下拉电阻是由模拟寄存器（analog register）控制，其他所有的状态（func、ie、oe、dataO 等）都是被数字寄存器（digital register）控制的。

参考文档后面低功耗管理的介绍可知，deepsleep retention 期间所有 digital register 的状态掉电丢失。

在 Telink 上一代 826x 系列 IC 上，suspend 期间可以用 gpio output 来控制一些外围设备，但到了 8x5x 上如果 suspend 被切换为 deepsleep retention mode 后，gpio output 状态失效，无法在 sleep 期间准确的控制外围设备。此时可以使用 GPIO 模拟上下拉电阻的状态来代替实现：上拉 10K 代替 gpio output high，下拉 100K 代替 gpio output low。

注意：deepsleep retention 期间的 GPIO 状态控制不要使用上拉 1M（上拉电压可能会比供电电压 VCC 低一些）。另外，上拉 10K 的控制中不要使用 PC0~PC7 的上拉 10K（在 deepsleep retention wake_up 时会有短时间的抖动，产生毛刺），其他 GPIO 上拉 10K 都是可以的。

2.3.5 配置 SWS 上拉防止死机

Telink 所有的 MCU 都使用 SWS（single wire slave）来调试和烧录程序。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。

8x5x 的 SWS 和 GPIO_PA7 复用，在 drivers/8258/gpio_default_8258.h 中将 PA7 的 1M 上拉开启即可：

```
#ifndef PULL_WAKEUP_SRC_PA7
#define PULL_WAKEUP_SRC_PA7      PM_PIN_PULLUP_1M  //sws pullup
#endif
```

Shared under NDA

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。

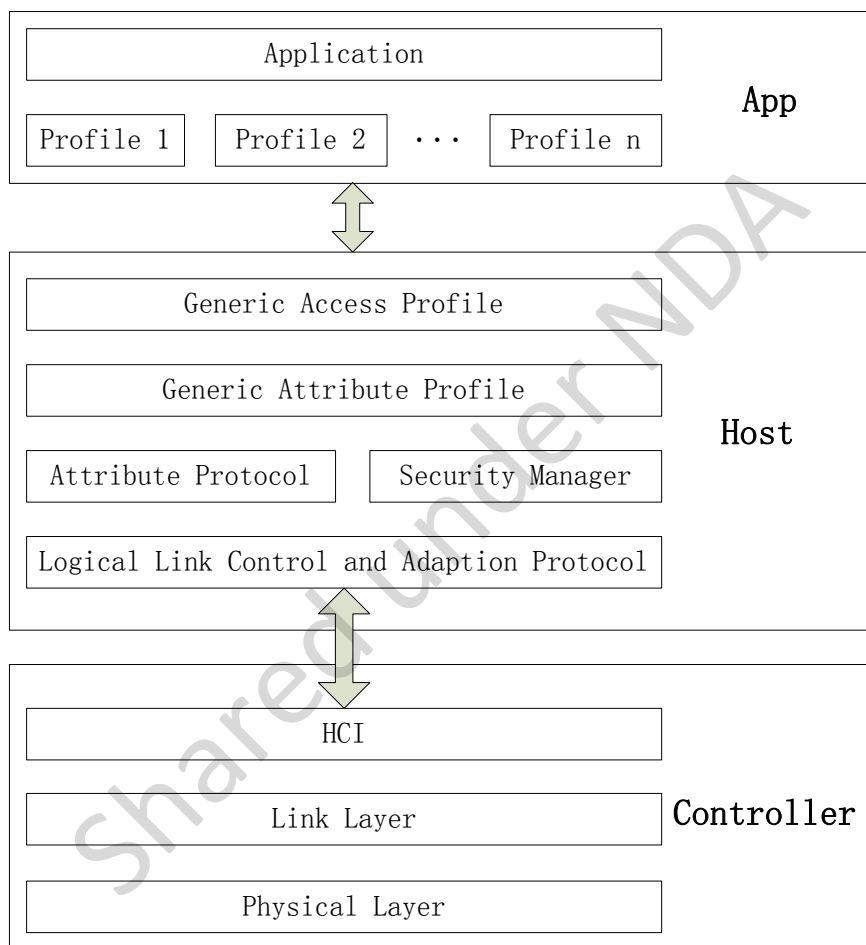


图 3-1 BLE SDK 标准架构

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。

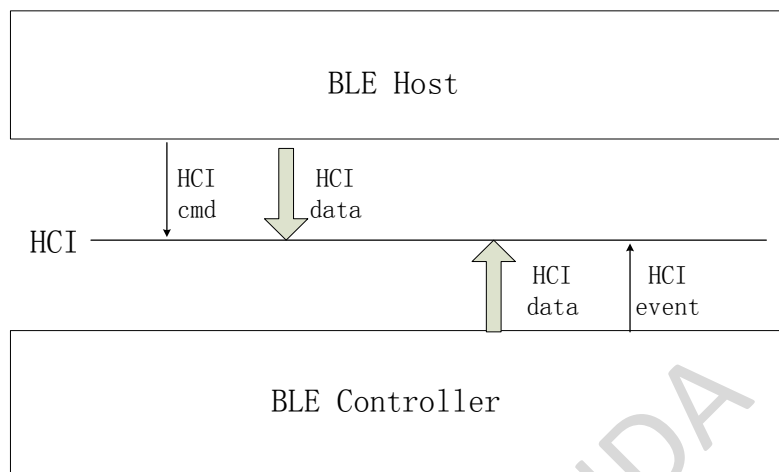


图 3-2 Host 和 Controller 的 HCI 数据交互

- 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 Telink BLE SDK 软件架构

3.1.2.1 Telink BLE controller

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

Telink BLE SDK 包含 Link Layer 的五种标准状态（standby、advertising、scanning、initiating、connection），connection 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 8258 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。

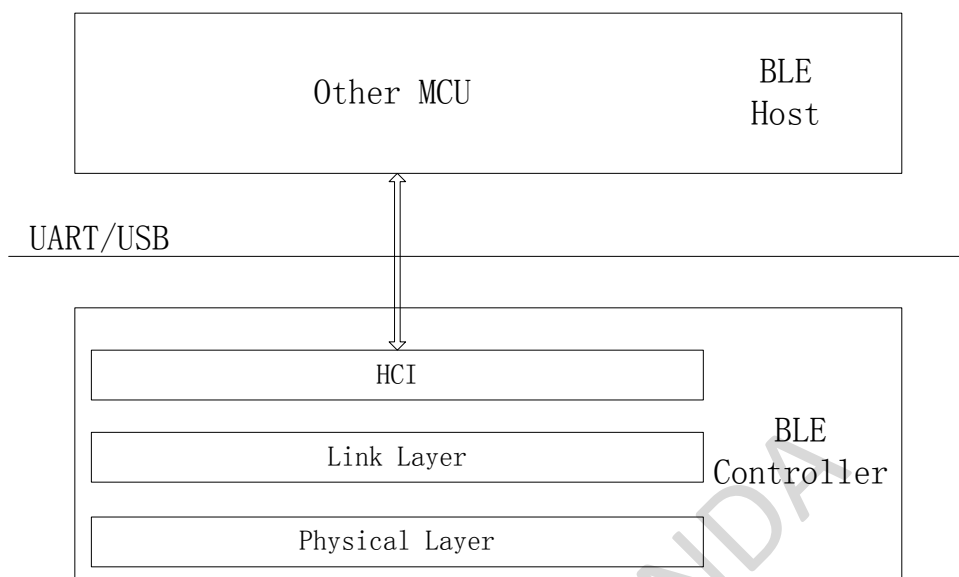


图 3-3 8258 hci 架构

Link Layer connection 状态同时支持 single connection 的 Slave 和 Master，那么 8258 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，几乎是没有什么意义的，所以 SDK 在 8258 hci 上并没有将 master role 的初始化放进去。

3.1.2.2 Telink BLE slave

Telink 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，Telink 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 8258 ble sample、8258 remote、8258 module 都是基于该架构。

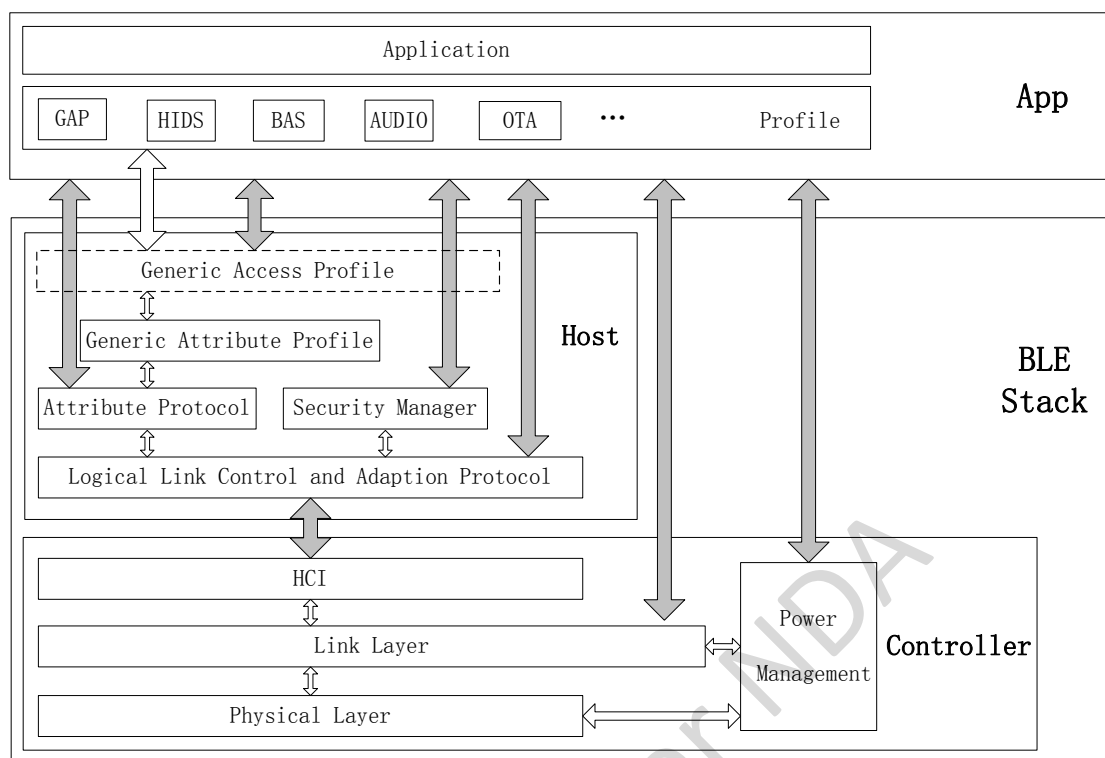


图 3-4 Telink BLE slave 架构

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。

考虑到效率，应用层与 Host 的数据交互不通过 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。但是 Host 的事件需要通过 GAP 层和应用层交互。

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。该 BLE SDK demo code 提供几个基本的 profile，包括 HIDS、BAS、AUDIO、OTA 等。

下面基于这个架构对 8x5x 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。

3.1.2.3 Telink BLE master

Telink BLE master 的实现方式不同于 Slave, SDK 上提供标准的 controller 封装到 library 里面, 但在 app 层实现 host 和 user 自己的 application, 如下图所示。

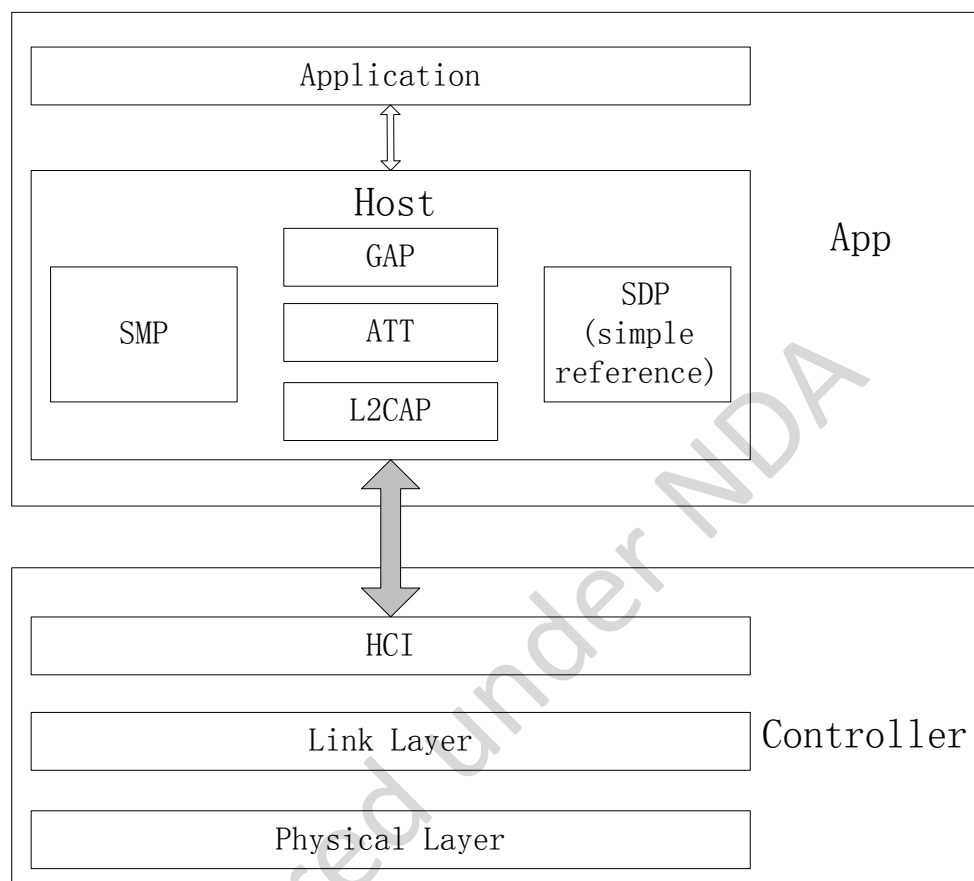


图 3-5 Telink BLE master 架构

SDK 中 8258 master kma dongle 工程 demo code 是基于该架构实现的, host 层的代码几乎都在 app 中实现了, SDK 提供各种标准的 interface 供 user 去完成这些功能。

App 层中实现了标准的 l2cap、att 等的处理, 在 SMP 部分只提供了最基本的 just work 方式, 8258 master kma dongle 默认 SMP 是 disable 的, 需要 user 自己打开这个宏才能 enable SMP。由于 SMP 的实现比较复杂, 具体的 code 实现还是封装在 library 里面, app 层只需要调用相关的 interface 即可, user 搜索 BLE_HOST_SMP_ENABLE 即可找到所有的 code 处理。

```

#define BLE_HOST_SMP_ENABLE 0

//1 for standard security management,
// 0 for telink referenced pairing&bonding(no security)

```

SDP 是最复杂的部分，Telink BLE master 不提供标准的 SDP，只给出了一个简单的参考，是对 8258 remote 的 service discovery。8258 master kma dongle 默认该 simple reference SDP 是打开的。

```
#define BLE_HOST_SIMPLE_SDP_ENABLE    1    //simple service discovery
```

SDK 提供所有的 service discovery 相关 ATT 操作的标准 interface，user 可以参考 8258 remote 的 service discovery 去实现自己的 service discovery，或者将 BLE_HOST_SIMPLE_SDP_ENABLE disable，使用 slave 约定好的 service ATT handle 来实现数据访问。

Telink BLE master 不支持 Power Management，因为对 Link Layer 的 scanning 和 connection master role 没有做 suspend 处理。

3.2 BLE controller

3.2.1 BLE controller 介绍

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

Telink BLE SDK 对 Physical Layer（对应 driver 文件中的 rf_drv.h 的 c 文件）完全封装到 library 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 和 HCI，且由于 HCI 主要是 interface，其实现的功能也都是为了操作 Link Layer 和获取 Link Layer 的数据，所以重点详细介绍 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。

更多信息请参照《Core_v5.0》(Vol 6/Part B/1.1 “LINK LAYER STATES”)。

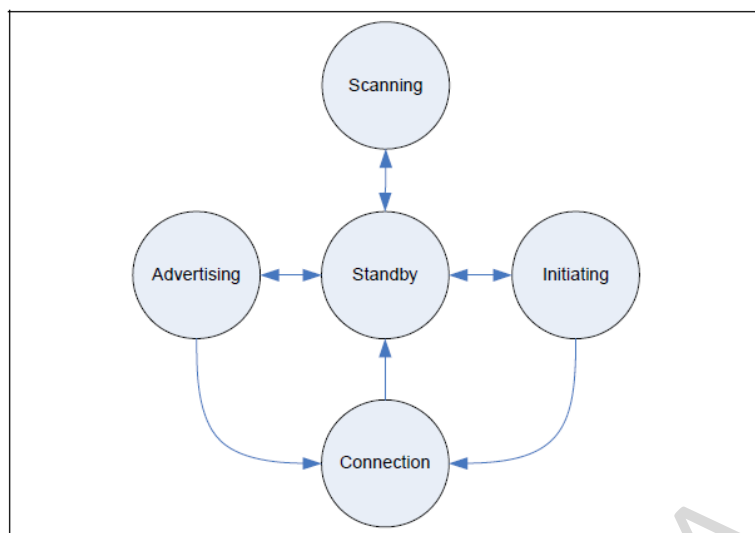


图 3-6 State diagram of the Link Layer state machine in BLE Spec

Telink BLE SDK Link Layer 状态机如下图所示。

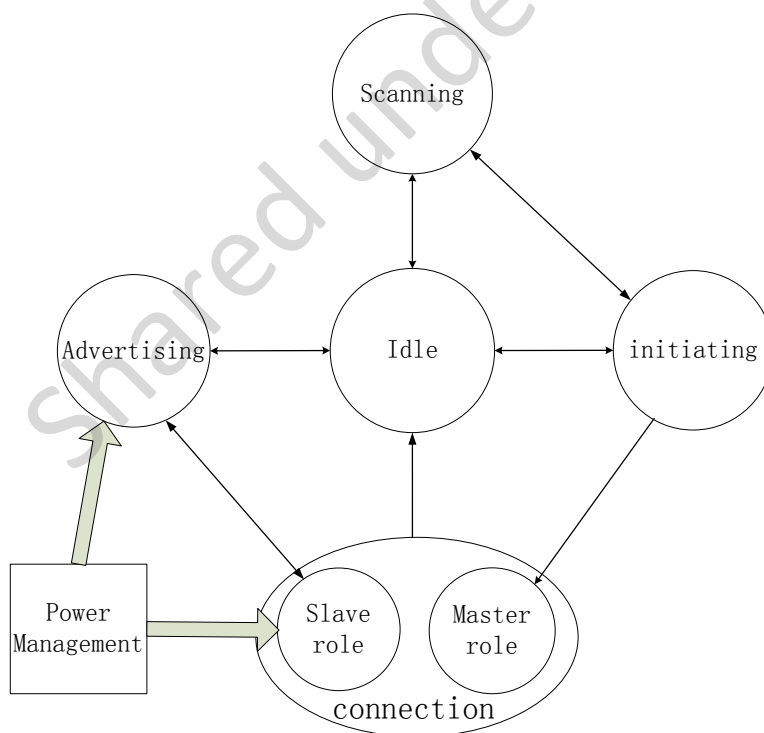


图 3-7 Telink Link Layer state machine

Telink 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 Connection 状态区分为 Slave Role 和 Master Role。

由文档前面对 library 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 connection, 和将来的 Mater 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, stack/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)
```

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

Telink 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。

MCU 的初始化是必须的, API 如下:

```
void blc_ll_initBasicMCU(void);
```

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_initStandby_module (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_initConnection_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。

```
void      blc_ll_initConnection_module(void);
```

用于初始化master和slave共用的module

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 + Advtersing

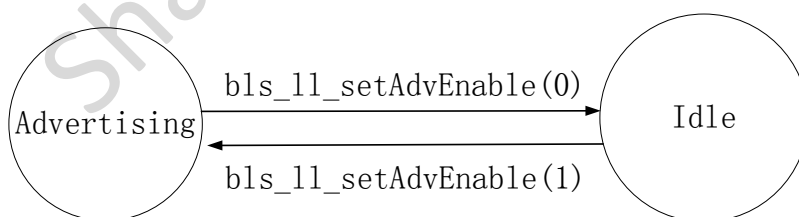


图 3-8 Idle + Advertising

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();
blc_ll_initStandby_module(tbl_mac);
```

```
blc_ll_initAdvertising_module(tbl_mac);
```

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

3.2.3.3 Idle + Scanning

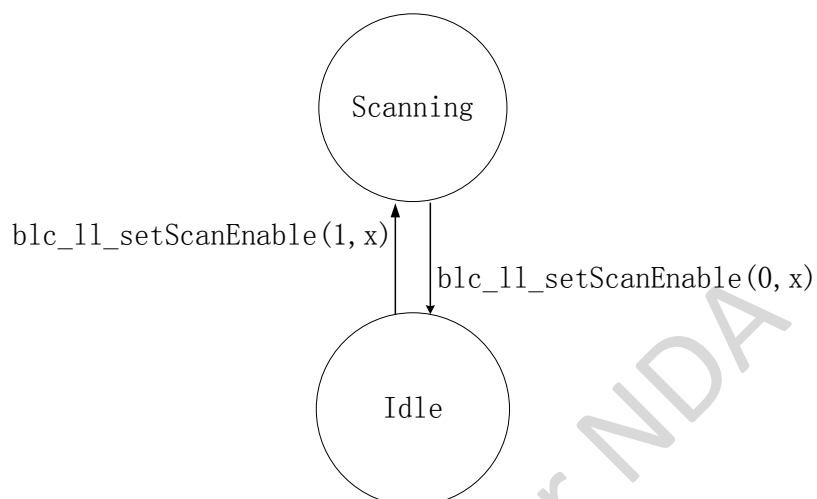


图 3-9 Idle + Scanning

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();
blc_ll_initStandby_module(tbl_mac);
blc_ll_initScanning_module(tbl_mac);
```

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

3.2.3.4 Idle + Advtersing + ConnSlaveRole

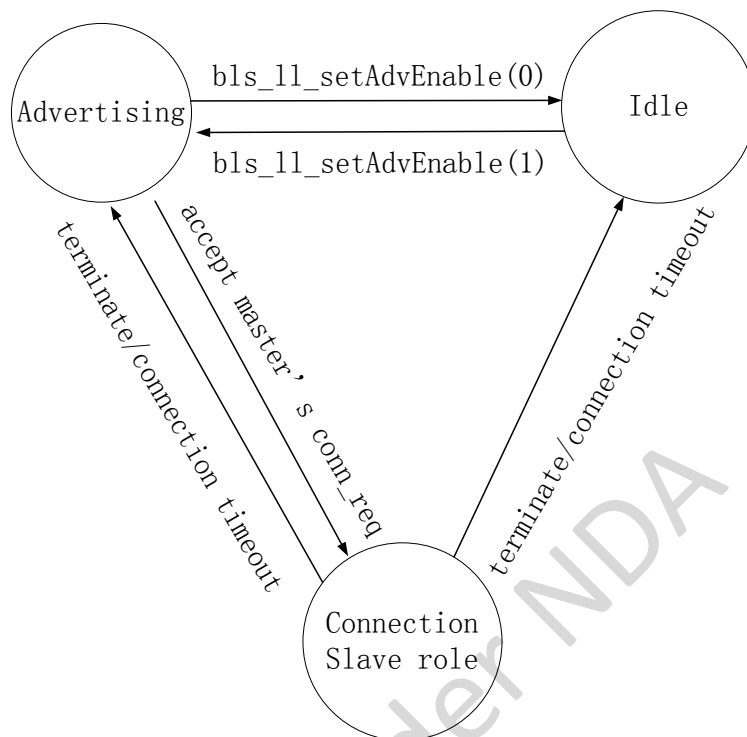


图 3-10 BLE slave LL state

上图所示为一个基本的 BLE slave 应用的 Link Layer 状态机组合，SDK 中 8258 hci/8258 ble sample/8258 remote/8258 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();
blc_ll_initStandby_module(tbl_mac);
blc_ll_initAdvertising_module(tbl_mac);
blc_ll_initConnection_module();
blc_ll_initSlaveRole_module();

```

该状态机组合下，状态变化情况描述如下。

- 1) 8x5x 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 connecton 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。
 - 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。

3.2.3.5 Idle + Scanning + Initiating + ConnMasterRole

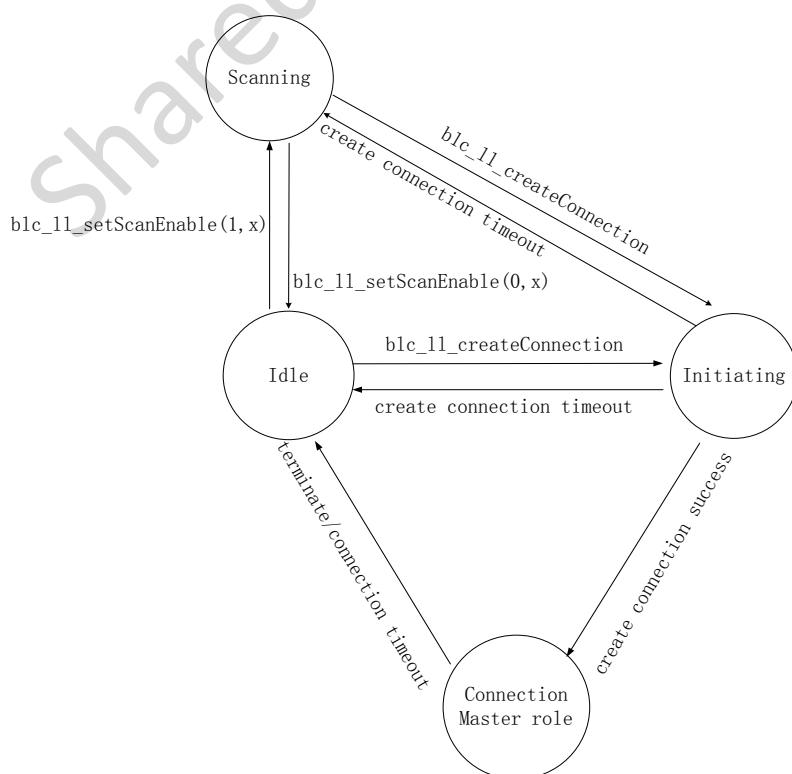


图 3-11 BLE master LL state

上图所示为一个基本的 BLE master 应用的 Link Layer 状态机组合, SDK 中 8258 master kma dongle 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();  
blc_ll_initStandby_module(tbl_mac);  
blc_ll_initScanning_module( tbl_mac);  
blc_ll_initInitiating_module();  
blc_ll_initConnection_module();  
blc_ll_initMasterRoleSingleConn_module();
```

该状态机组合下, 状态变化情况描述如下。

- 1) 8x5x MCU 上电后, 进入 Idle state。Idle state 时将 Scan Enable, Link Layer 切换到 Scanning State; 在 Scanning State 时将 Scan Disable 时, 回到 Idle state。

Scan Enable 和 Disable 通过 API blc_ll_setScanEnable 来控制。

上电后, Link layer 默认处于 Idle state, 一般需要在 user_init 里面将 Scan Enable, 进入 Scanning state。

Link Layer 处于 Scanning state 时, 会将 Scan 到的 adv packet 通过 event “HCI_SUB_EVT_LE_ADVERTISING_REPORT” report 给 BLE host。

- 2) Link Layer 可以在 Idle state 和 Scanning state 通过 API blc_ll_createConnection 触发进入 Initiating state。

blc_ll_createConnection 指定了需要连接的一个或多个 BLE 设备的 mac address。Link Layer 进入 Initiating state 后不断 Scan 指定的 BLE 设备, 在收到某个正确的可连接的 adv packet 后发送 connection request 并进入 ConnMasterRole。若在一定时间内 Initiating state 没有 Scan 到指定的 BLE 设备, 无法发起连接, 将会触发 create connection timeout, 重新回到 Idle State 或者 Scanning state。

上面要注意的是 Initiating state 可以从 Idle state 和 Scanning state 进入 (8258 master kma dongle 就是从 Scanning state 直接进入), create connection timeout 后回到 create connection 之前的 Idle State 或者 Scanning state。

- 3) Link Layer 处于 ConnMasterRole 时, 有三种情况会回到 Idle State:

- a) slave 向 master 发送 terminate 命令, 断开连接。master 收到 terminate 命令, 退出 ConnMasterRole。
- b) master 向 slave 发送 terminate 命令, 主动断开连接, 退出 ConnMasterRole。

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

Link layer 的 ConnMasterRole 退出该 state 时，只能回到 Idle state。若需要继续 Scan 的话必须使用 API `blc_ll_setScanEnable` 设置 Link Layer 再次进入 Scanning state。

3.2.4 Link Layer 时序

结合该 BLE SDK 的 `irq_handler` 和 `main_loop` 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 `main_loop` 的时序。

3.2.4.2 Advertising state 时序

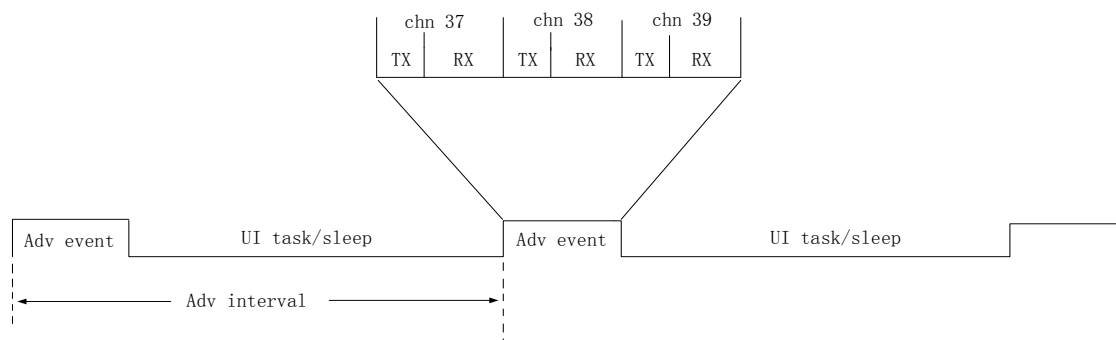


图 3-12 Advertising State 时序

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 request**，收这个包后和 master 建立 ble 连接，进入 **Connection state Slave Role**。

user 在 **main_loop** 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 **sleep(suspend/deepsleep retention)** 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

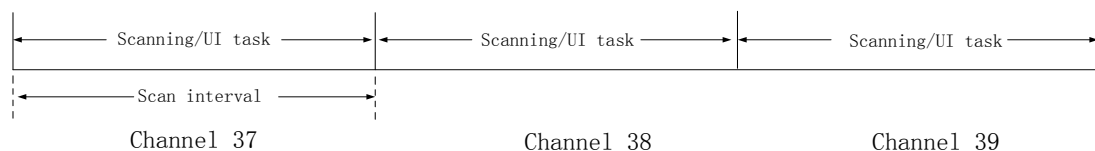


图 3-13 Scanning state 时序

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 `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

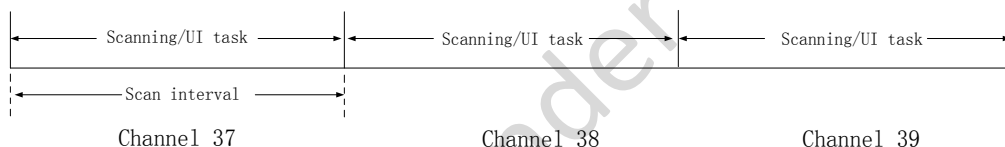


图 3-14 Initiating state 时序

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

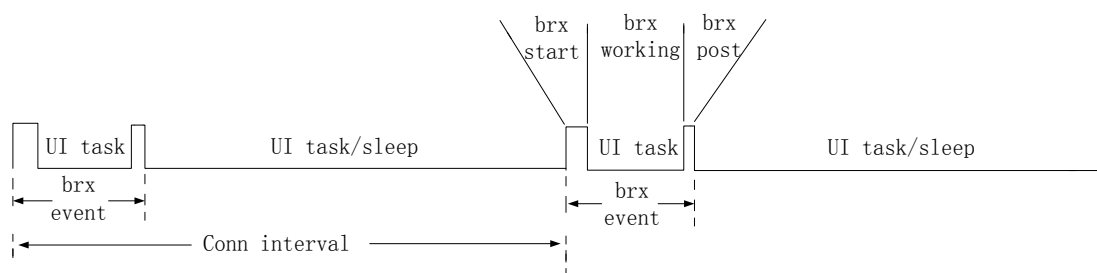


图 3-15 Conn state Slave role 时序

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。

在该 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准备，然后退出中断 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（注意 brx working 阶段 RX、TX、系统定时器中断等处理函数以外的时隙是可以跑 UI task 的）。brx working 时间内，硬件需要进行收发包，所以不能进 sleep(suspend/deepsleep retention)。

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 sleep(suspend/deepsleep retention)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 Master role 时序

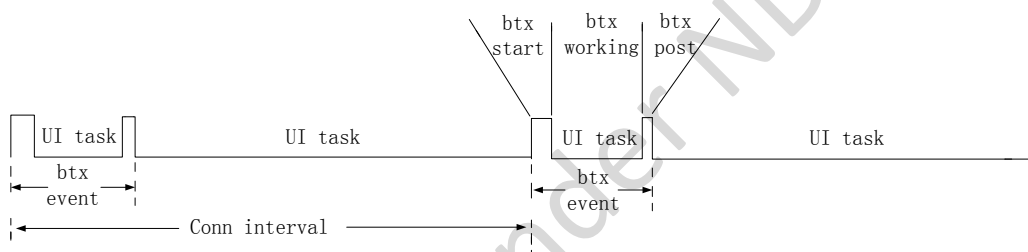


图 3-16 ConnMasterRole 时序

ConnMasterRole 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 发收包过程：先让 PHY 进入发包状态，向 slave 发送一个包后等待对方的 ack 包，若有 more data，则继续向 slave 发包，这个过程简称为 btx event。

在该 BLE SDK 中，根据软硬件的工作分配，每个 btx 过程分为 3 个阶段：

- 1) btx start 阶段

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

- 2) btx working 阶段

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

- 3) btx post 阶段

收发包完成后，btx working 结束，同样由 system tick irq 触发进入中断执行 btx post 阶段。这个阶段主要是协议栈根据 btx 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 `btx start` 和 `btx post` 都是中断完成，而 `btx working` 阶段不需要软件的参与，此时 `UI task` 可以正常执行。

在 `ConnMasterRole`，`blt_sdk_main_loop` 需要处理 `btx` 过程收到的数据。`btx 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。

`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 `slave` 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。

3.2.4.7 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`。`8258 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 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), Telink 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。

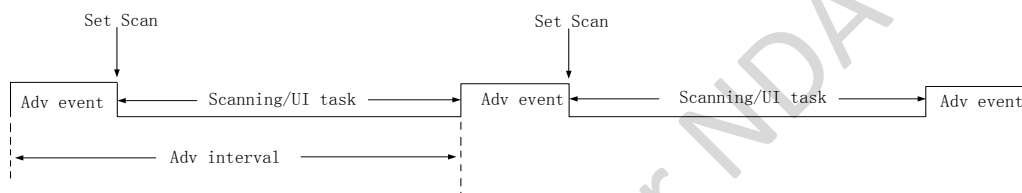


图 3-17 Scanning in Advertising state 时序

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 8258_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。

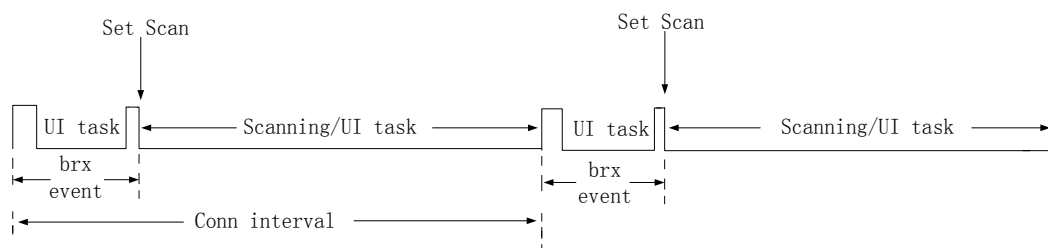


图 3-18 Scanning in ConnSlaveRole 时序

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 8258_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：

```
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。

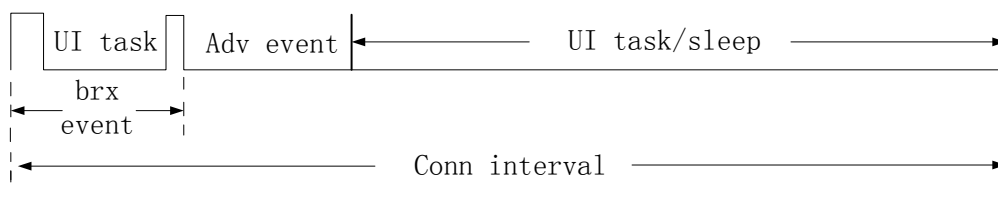


图 3-19 Advertising in ConnSlaveRole 时序

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 sleep(suspend/deepsleep retention)节省功耗。

Advertising in ConnSlaveRole 的使用请参考 8258_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：

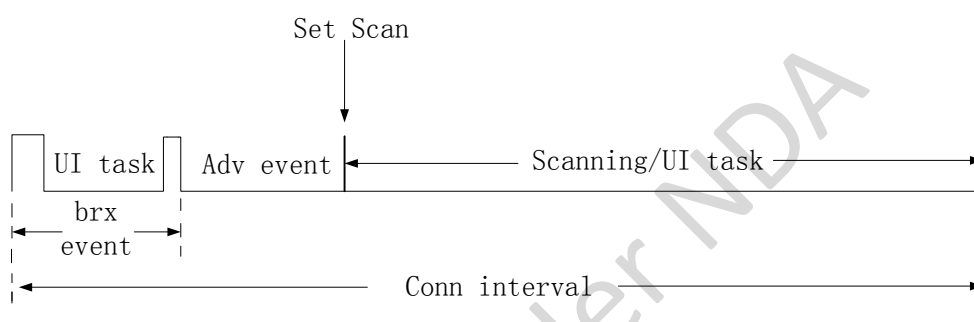


图 3-20 Advertising and Scanning in ConnSlaveRole 时序

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 8258_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 开始后数据进行数据发送。

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 的处理方式一致，BLE TX fifo 和 BLE RX fifo 都在应用层定义：

```
MYFIFO_INIT(blt_rxfifo, 64, 8);
```

```
MYFIFO_INIT(blk_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 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(MD)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。

下面举一个 RX overflow 的示例，我们有如下假设：

- 1) RX fifo 数量为 8；
- 2) 在 brx_event(n)开启前 RX fifo 的读、写指针分别为 0 和 2；
- 3) 在 brx_event(n)和 brx_event(n+1)阶段 main_loop 存在任务阻塞，没有及时去取 RX fifo；
- 4) 两个 brx_event 阶段都是多包情况。

由上文“Conn state Slave role 时序”小节描述我们知道，在 brx_working 阶段收到的 BLE 数据包只会拷贝到 RX fifo 中（RX fifo 写指针++），真正取出 RX fifo 数据进行处理操作是放在 main_loop 阶段的（RX fifo 读指针++），我们可以看到第 6 笔数据会覆盖读指针 0 区域。这里需要注意的是在 brx working 阶段的 UI task 时隙是在 RX、TX、系统定时器等中断处理除外的时间。

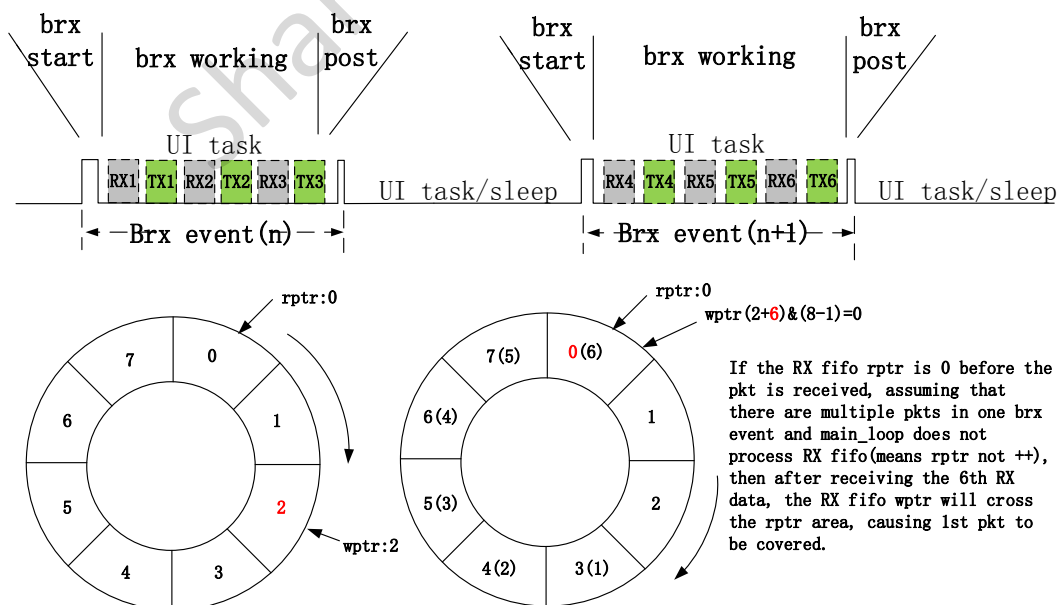


图 3-21 RX overflow 图示 1

上面的例子由于间隔了 1 个连接间隔，任务阻塞时间得要足够长，有点极端，下面这个 RX overflow 情形则相对而言出现的概率更高：在一个 brx_event 期间，master 向 slave 写入多笔数据，比如多包数量 7、8 个，这种情况下由于 master 一下子发送了很多数据，slave 来不及进行处理。如下图，读指针只移动了 2 笔，但是写指针移动了 8 笔也会造成数据溢出。

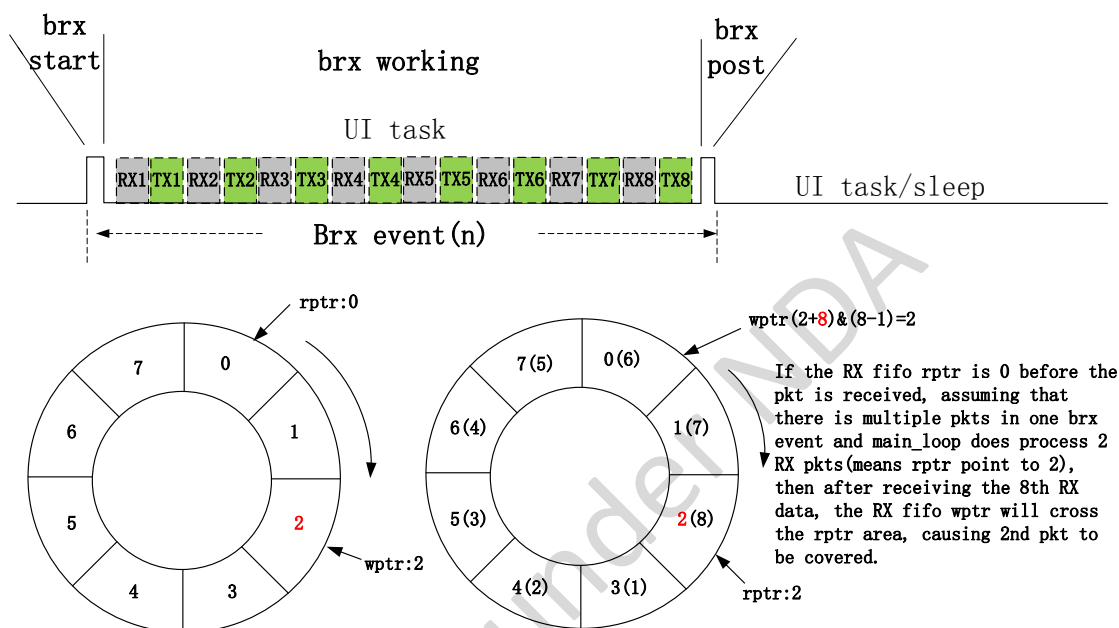


图 3-22 RX overflow 图示 2

一旦出现 overflow 导致的数据丢失问题，对加密系统而言，则会出现 MIC failure 断连问题。（旧 SDK 由于 brx event Rx IRQ 将往 Rx fifo 填数据，但是不做数据溢出检查，如果 main_loop 处理 RX fifo 太慢就会导致溢出问题，所以在使用旧 SDK 时，用户需要更多的注意这个风险，避免 master 在一个连接间隔上发太多的数据，注意用户 UI task 处理时间尽量短，避免阻塞问题。）

目前 SDK 新增了 Rx overflow 校验：在 brx/btx event Rx IRQ 中检查当前的 RX fifo 写指针和读指针差值是否大于 Rx fifo 数量，一旦发现 Rx fifo 被占满则让 RF 不去 ACK 对方，BLE 协议会确保数据重传，此外 SDK 还提供了 Rx overflow 回调函数以便通知用户，文档后面章节“Telink defined event”会介绍这个回调。

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      blc_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。

TX fifo 使用上, 如果客户先看多少个剩余, 再决定是否直接 push 数据时, 要留一个 fifo, 防止发生各种边界问题。

在 8258 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, 被占用的 fifo 不能超过 9 个。为了避免 TX fifo 使用时的一些边界条件导致的异常（比如正好赶上 BLE stack 要回复 master 的 command, 往 TX fifo 里插入了一个数据），最终 code 写法如下: 当已被占用的 TX fifo 不超过 8 个时, 才将语音数据 push 到 TX fifo。

```
if (blc_ll_getTxFifoNumber() < 9)
{
    .....
}
```

上面讨论过数据溢出问题, SDK 底层除了提供数据快要 overflow 自动处理机制外, 还提供了下面的 API 用于限定一个 interval 上 more data 接收数量（如果客户希望 RX fifo 足够用的情况下也对数据进行限制, 则可以使用）。

```
void      blc_ll_init_max_md_nums (u8 num);
```

其中参数 num 可以设置的 more data 数量最大不要超过 RX fifo number。

需要注意的是, 只有在应用层调用该 API（参数 num 大于 0）才开启限定一个连接事件上 more data 数据的功能。

3.2.7 Controller Event

为了满足 user 在应用层对 BLE stack 底层一些关键动作的记录和处理，Telink BLE SDK 提供了三种类型的 event：一是 BLE Controller 定义的标准的 HCI event；二是 Telink 自己定义的一套 event，称为 Telink defined event（前两种类型均属于 Controller event）；三是 BLE host 定义的一些协议栈流程交互的事件通知型 GAP event（也可以认为是 host event，这部分具体介绍请参考本文档“GAP event”章节）。

BLE SDK event 架构说明如下图所示，可以看到 HCI event 和 Telink defined event 均属于 Controller event，而 GAP event 属于 BLE host event。下面两小节内容主要介绍 Controller event。

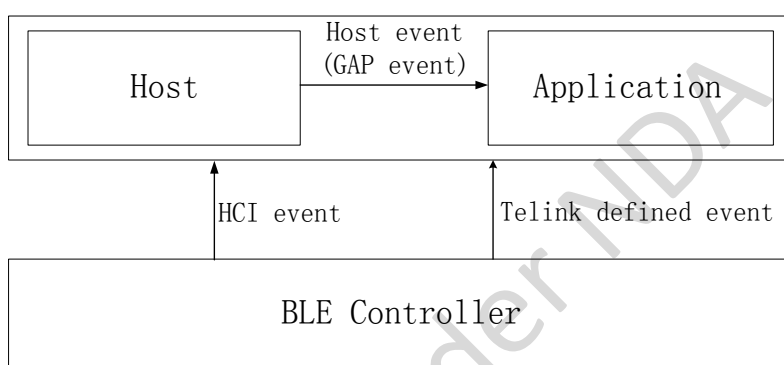


图 3-23 BLE SDK event 架构

3.2.7.1 Controller HCI Event

HCI event 是按 BLE Spec 标准设计的，而 Telink defined event 只在 BLE slave（8258 remote/8258 module 等）上有效，即：

- ✧ 对于 BLE master，只有 HCI event；
- ✧ 对于 BLE slave，HCI event 和 Telink defined event 同时可用。

在 BLE slave 上，这两套 event 基本相互独立，只有两个 event 重复定义了：Link Layer 的 connect 和 disconnct event，后面会具体介绍。

User 可以根据自己的需要，在这两套 event 挑选一套使用，也可以两套同时使用。Telink BLE SDK 中，8258 remote/8258 module 等使用了 Telink defined event，8258 hci/8258 master kma dongle 使用了 Controller HCI event。

如下图 Host + Controller 架构所示，Controller HCI event 是通过 HCI 将 Controller 所有的 event 报告给 Host。

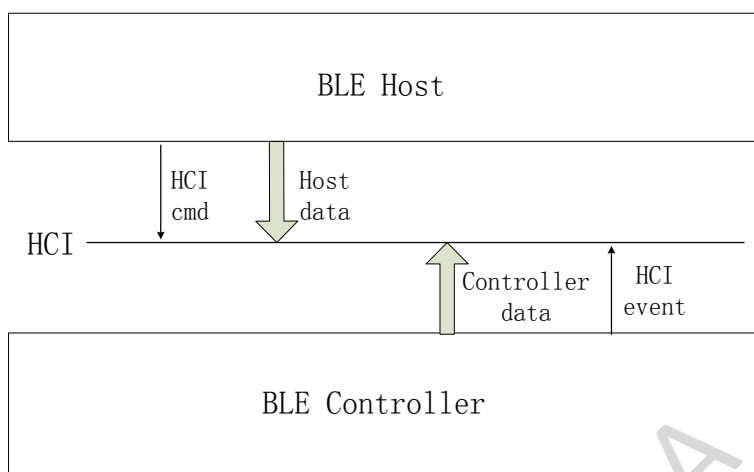


图 3-24 HCI event

Controller HCI event 的定义，详情请参照《Core_v5.0》(Vol 2/Part E/7.7 “Events”)。其中 7.7.65 “LE Meta Event” 指 HCI LE(low energy) Event，其他的都是普通的 HCI event。根据 Spec 的定义，Telink BLE SDK 也将 Controller HCI event 分为两类：HCI Event 和 HCI LE event。由于 Telink BLE SDK 主打低功耗蓝牙，所以对 HCI event 只支持了最基本的几个，而 HCI LE event 绝大多数都支持。

Controller HCI event 相关的宏定义、接口定义等请参考 stack/ble/hci 目录下的头文件。

如果 user 需要在 Host 或 App 层接收 Controller HCI event，首先需要注册 Controller HCI event 的 callback 函数，其次需要将对应 event 的 mask 打开。

Controller HCI event 的 callback 函数原型和注册接口分别为：

```

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

callback函数原型中的u32 h是一个标记，底层协议栈多处会用到，user只需要知道以下两个即可：

```

#define HCI_FLAG_EVENT_TLK_MODULE (1<<24)
#define HCI_FLAG_EVENT_BT_STD (1<<25)
  
```

HCI_FLAG_EVENT_TLK_MODULE 在后面的 Telink defined event 会再介绍，HCI_FLAG_EVENT_BT_STD 这个标志表示当前 event 为 Controller HCI event。

callback 函数原型中 para 和 n 表示 event 的数据和数据长度，该数据和 BLE spec 中定义的一致。User 可参考 8258 master kma dongle 中如下用法以及 controller_event_callback 函数的具体实现。

```
blc_hci_registerControllerEventHandler(controller_event_callback);
```

1. HCI event

Telink BLE SDK 中支持了少部分的 HCI event，下面列出了几个 user 可能希望了解的 event。

```
#define HCI_EVT_DISCONNECTION_COMPLETE 0x05
#define HCI_EVT_ENCRYPTION_CHANGE 0x08
#define HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE 0x0C
#define HCI_EVT_ENCRYPTION_KEY_REFRESH 0x30
#define HCI_EVT_LE_META 0x3E
```

1) HCI_EVT_DISCONNECTION_COMPLETE

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.5 “Disconnection Complete Event”)。

该 event 的总体数据长度为 7，param len 为 4，如下所示，具体数据含义请直接参考 BLE spec。

hci event	event code	param len	status	connection handle	reason
0x04	0x05	4	0x00		

图 3-25 Disconnection Complete Event

2) HCI_EVT_ENCRYPTION_CHANGE 和 HCI_EVT_ENCRYPTION_KEY_REFRESH

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.8 & 7.7.39)。

跟 Controller 加密相关,这两个 event 可以在 8258 master kma dongle 中看到,具体的处理封装到 library 中完成了,这里不介绍细节。

3) HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.12)。

当 Host 使用了 HCI_CMD_READ_REMOTE_VER_INFO 命令 Controller 和 BLE peer device 交互 version 信息,并且收到了 peer device 的 version 后,向 Host 上报该 event。

该 event 的总体数据长度为 11，param len 为 8，如下所示，具体数据含义请直接参考 BLE spec。

hci event	event code	param len	status	connection handle	version	manufacture name	subversion
0x04	0x0c	8	0x00				

图 3-26 Read Remote Version Information Complete Event

4) HCI_EVT_LE_META

表示当前是 HCI LE event, 根据后面的 sub event code 判断具体的 event 类型。

HCI event 中除了 HCI_EVT_LE_META, 其他都要通过下面 API 来打开 event mask。

```
ble_sts_t blc_hci_setEventMask_cmd(u32 evtMask); //eventMask: BT/EDR
```

event mask的定义如下所示:

```
#define HCI_EVT_MASK_DISCONNECTION_COMPLETE 0x0000000010
#define HCI_EVT_MASK_ENCRYPTION_CHANGE 0x0000000080
#define HCI_EVT_MASK_READ_REMOTE_VERSION_INFORMATION_COMPLETE 0x00000000800
```

若 user 未通过该 API 设置 HCI event mask, SDK 默认只打开 HCI_CMD_DISCONNECTION_COMPLETE 对应的 mask, 即保证 Controller disconnect event 的上报。

2. HCI LE event

当 HCI event 中 event code 为 HCI_EVT_LE_META, 就是 HCI LE event, subevent code 最常用的且 user 可能需要了解的如下, 其他的不做介绍。

```
#define HCI_SUB_EVT_LE_CONNECTION_COMPLETE 0x01
#define HCI_SUB_EVT_LE_ADVERTISING_REPORT 0x02
#define HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE 0x03
#define HCI_SUB_EVT_LE_CONNECTION_ESTABLISH 0x20 //telink private
```

1) HCI_SUB_EVT_LE_CONNECTION_COMPLETE

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.65.1 “LE Connection Complete Event”)。

当 controller Link Layer 和 peer device 建立 connection 后, 上报该 event。

该 event 的总体数据长度为 22, param len 为 19, 如下所示, 具体数据含义请直接参考 BLE spec。

0x04	0x3e	19	0x01				
hci event	event code	param len	subevent code	status	connection handle	Role	peerAddr type
peer addr					conn interval		
conn latecncy		supervision timeout		master clock accuracy			

图 3-27 LE Connection Complete Event

2) HCI_SUB_EVT_LE_ADVERTISING_REPORT

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.65.2 “LE Advertising Report Event”)。

当 controller 的 Link Layer scan 到正确的 adv packet 后，通过 HCI_SUB_EVT_LE_ADVERTISING_REPORT 上报给 Host。

该 event 的数据长度不定（取决于 adv packet 的 payload），如下所示，具体数据含义请直接参考 BLE spec。

0x04	0x3e		0x02			
hci event	event code	param len	subevent code	num report	event type	address type[1...i]
address[1...i]					length[1..i]	
data[1...i]					rssi[1..i]	

图 3-28 LE Advertising Report Event

注意: Teink BLE SDK 中的 LE Advertising Report Event 每次只报一个 adv packet，即上图中的 i 为 1。

3) HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE

详情请参照《Core_v5.0》(Vol 2/Part E/7.7.65.3 “LE Connection Update Complete Event”)。

当 Controller 上的 connection update 生效时，向 Host 上报 HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE。

该 event 的总体数据长度为 13，param len 为 10，如下所示，具体数据含义请直接参考 BLE spec。

0x04	0x3e	10	0x03		
hci event	event code	param len	subevent code	status	connection handle
conn interval		conn latency		supervision timeout	

图 3-29 LE Connection Update Complete Event

4) HCI_SUB_EVT_LE_CONNECTION_ESTABLISH

HCI_SUB_EVT_LE_CONNECTION_ESTABLISH 是对 HCI_SUB_EVT_LE_CONNECTION_COMPLETE 的补充，除了 subevent 不一致外，其他所有参数相同。SDK 中 8258 master kma dongle 使用了该 event。

该 event 是唯一的非 BLE spec 标准的 event，属于 Telink 私有定义，且只在 8258 master kma dongle 中使用。

下面详细说明 Telink 定义该 event 的原因。

BLE Controller 在 Initiating state 时，扫描到指定需要连接的 device adv packet 时，向它发送 connection request 包，此时不管对方是否有收到这个 connection request，都会无条件认为 Connection complete，向 Host 上报 LE Connection Complete Event，Link Layer 迅速进入 Master role。

由于这个包是不带 ack/retry 机制的，无法保证 Slave 一定会收到，如果 Slave 丢掉这个 connection request，就无法进入 Slave role，后面也不会进入 brx 模式收发包。

当这种情况发生时，Master Controller 这边的处理机制是：进入 Master role 后，检查一下前 6 ~ 10 个 conn interval 上有没有收到任何 slave 的包（此时不关心 CRC 是否正确）。

- ✧ 如果一个包都没收到，那么认为是 Slave 没有收到 connection request，在前面已经上报了 LE Connection Complete Event 的前提下，必须快速上报一个 Disconnection Complete Event，并指出 disconnect reason 是 0x3E（HCI_ERR_CONN_FAILED_TO_ESTABLISH）。
- ✧ 在前 6 ~ 10 个 conn interval 上有收到 Slave 的包，才能确定 Connection Established（连接已确立），Master 后面的流程才能继续往下进行。

根据上面的描述，BLE Host 的处理方法应该是：在收到 Controller 的 LE Connection Complete Event 后，不能认为 connection 已经 Established，必须根据 conn interval 启动一个 timer（时间设置大一些，10 个 interval 以上，覆盖住最长时间），在这个 timer 之内检查是否有 disconnect reason 为 0x3E

Disconnection Complete Event，如果没有的话才能认为 connection Established。

鉴于 BLE host 的这个处理非常的复杂，很容易出错，所以 SDK 在底层定义了 HCI_SUB_EVT_LE_CONNECTION_ESTABLISH，当 Host 收到这个 event 时，就表明 Controller 已经确定 Slave 端 connection OK 了，可以继续下面的流程。

HCI LE event 需要通过下面的 API 来打开 mask。

```
ble_sts_t blc_hci_le_setEventMask_cmd(u32 evtMask);  
//eventMask: LE
```

evtMask 的定义也对应上面给出一些，其他的 event 用户可以在 hci_const.h 中查到。

```
#define HCI_LE_EVT_MASK_CONNECTION_COMPLETE 0x00000001  
#define HCI_LE_EVT_MASK_ADVERTISING_REPORT 0x00000002  
#define HCI_LE_EVT_MASK_CONNECTION_UPDATE_COMPLETE 0x00000004  
#define HCI_LE_EVT_MASK_CONNECTION_ESTABLISH 0x80000000  
//telink private
```

若 user 未通过该 API 设置 HCI LE event mask，SDK 默认所有 HCI LE event 都不打开。

3.2.7.2 Telink defined event

除了标准的 Controller HCI event，SDK 提供了 Telink defined event。Telink defined event 最多支持 20 个，在 stack/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 可理解为和 HCI event 中的 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_LL_REJECT_IND 5
#define BLT_EV_FLAG_RX_DATA_ABANDON 6
#define BLT_EV_FLAG_PHY_UPDATE 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
```

前面已经介绍，Telink defined event 只在 BLE slave 相关应用中才会被触发。Telink defined event 在 BLE slave 应用上的回调实现，有两种方式。

- 1) 第一种方式是每个 event 的回调函数单独注册。这种方式我们称为“independent registration”。

回调函数的原型说明为：

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

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。没有被注册回调函数的 event，不会响应。

- 2) 第二种方式是：所有 event 的回调函数共用同一个入口，每个事件是否响应取决于该 event 对应的 event mask 是否被打开。这种方式我们称为“shared event entry”。

“shared event entry”方式注册 event 回调是使用和 HCI event 一样的 API：

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

void
blc_hci_registerControllerEventHandler(
    hci_event_handler_t handler);
```

虽然这里共用了 HCI event 的注册回调函数，但二者在实现上有一些区别。HCI event 回调函数里面

```
h = HCI_FLAG_EVENT_BT_STD | hci_event_code;
```

Telink defined event “shared event entry”方式里面

```
h = HCI_FLAG_EVENT_TLK_MODULE | e;
```

其中 e 是 Telink defined event 的 event number。

Telink defined event “shared event entry”方式，类似于 HCI event 的 mask 方法，每一个 event 是否被响应，由下面的 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);
```

Telink defined event 的两种实现方式，是互斥独立的，只能使用一种。推荐使用方式 1 “independent registration”，SDK 中绝大多数都是用的这种方式；只有 8258 module 使用了方式 2 “shared event entry”。

在 Telink defined event 的使用上，方式 1 “independent registration”请参考 project “8258_remote”的 demo code；方式 2 “shared event entry”请参考 project “8258_module”的 demo code。

下面以 connect 和 terminate 事件回调为例，描述这两种方式的 code 实现的方法。

1) 方式一 “independent registration”

```
void task_connect (u8 e, u8 *p, int n)
{
    // add connect callback code here
}

void task_terminate (u8 e, u8 *p, int n)
{
    // add terminate callback code here
}

bls_app_registerEventCallback (BLT_EV_FLAG_CONNECT,
                               &task_connect);
bls_app_registerEventCallback (BLT_EV_FLAG_TERMINATE,
                               &task_terminate);
```

2) 方式二 “shared event entry”

```
int event_handler(u32 h, u8 *para, int n)
{
    if( (h&HCI_FLAG_EVENT_TLK_MODULE) != 0 ) //module event
    {
        switch(event)
        {
            case BLT_EV_FLAG_CONNECT:
            {
                // add connect callback code here
            }
            break;

            case BLT_EV_FLAG_TERMINATE:
            {
                // add terminate callback code here
            }
            break;

            default:
            break;
        }
    }
}
```

```
    }  
    }  
}  
  
blc_hci_registerControllerEventHandler(event_handler);  
  
bls_hci_mod_setEventMask_cmd( BIT(BLT_EV_FLAG_CONNECT) |  
                               BIT(BLT_EV_FLAG_TERMINATE) );
```

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。

1) BLT_EV_FLAG_ADV

该事件目前没有使用。

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) 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。

4) BLT_EV_FLAG_CONNECT

事件触发条件：Link Layer 处于广播状态时收到 master 的连接请求包单元，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，对应如下图所示的连接请求包单元 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 3-30 连接请求包单元 PDU

可以参考 ble_common.h 中 rf_packet_connect_t 的定义，连接请求包单元 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;
```

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：

- A. slave 和 master 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。terminate reason 为 HCI_ERR_CONN_TIMEOUT（0x08）。
- B. 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。
- C. 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。

6) BLT_EV_FLAG_LL_REJECT_IND

事件触发条件：Master 在 Link Layer 发送 LL_ENC_REQ（encryption request），且声明了使用之前已经分配好的 LTK，slave 无法找到对应的 LTK，发送 LL_REJECT_IND（or LL_REJECT_EXT_IND），此时触发该事件。

回传指针 p：指向发送的 command(LL_REJECT_IND or LL_REJECT_EXT_IND)。

数据长度 n：1。

更多信息请参照《Core_v5.0》(Vol 6/Part B/2.4.2)。

7) BLT_EV_FLAG_RX_DATA_ABANDON

事件触发条件：当 BLE RX fifo overflow 时（参考前文“Link Layer TX fifo & RX fifo”小节），或者在一个连接间隔里收到多包数量>设定的多包数量阈值（用户需要调用 API：blc_ll_init_max_md_nums 且参数不为 0，SDK 底层才会做多包数量的检查），触发 BLT_EV_FLAG_RX_DATA_ABANDON 事件。

回传指针 p：空指针 NULL。

数据长度 n：0。

8) BLT_EV_FLAG_PHY_UPDATE

事件触发条件：当 slave 或者 master 主动发起 LL_PHY_REQ，更新成功或者失败后触发；或者当 slave 或者 master 被动收到 LL_PHY_REQ，并且 PHY 更新成功后触发。

数据长度 n：1

回传指针 p：指向一个 u8 类型的变量，指示当前连接的 PHY mode。

```
typedef enum {
    BLE_PHY_1M          = 0x01,
    BLE_PHY_2M          = 0x02,
    BLE_PHY_CODED       = 0x03,
} le_phy_type_t;
```

9) 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;
    .....
}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);

10) BLT_EV_FLAG_GPIO_EARLY_WAKEUP

事件触发条件：BLE slave 进入 sleep（suspend 或 deepsleep retention）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 sleep 状态下的 timer 计时实现的），那么会存在一个问题：如果 sleep 的时间过长，user 的任务必须到 sleep 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 sleep 时间较大，如 400ms、1s 等（sleep 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 sleep 之前判断当前睡眠的时间，如果时间过长，设置 suspend/deepsleep retention 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/deepsleep retention 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 drivers/8258/pm.h 可看到如下唤醒状态。

```
enum {
    WAKEUP_STATUS_TIMER = BIT(1),
    WAKEUP_STATUS_PAD    = BIT(3),
    STATUS_GPIO_ERR_NO_ENTER_PM = BIT(7),
    STATUS_ENTER_SUSPEND = BIT(30),
};
```

以上参数的含义请参考“低功耗管理”部分下的详细说明。

11) 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

12) 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。

13) 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 3-31 BLE 协议栈 LL_CONNECTION_UPDATE_REQ 格式

14) 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 latency

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

15) 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。

16) 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。

3.2.8 Data Length Extension

BLE Spec 从 core_4.2 开始增加了 data length extension (DLE)。

该 BLE SDK Link Layer 上支持 data length extension, 且 rf_len 长度支持到 BLE spec 上最大长度 251 bytes。

详情请参照《Core_v5.0》(Vol 6/Part B/2.4.2.21 “LL_LENGTH_REQ and LL_LENGTH_RSP”)。

User 如果需要使用 data length extension 功能, 按如下步骤设置。

1) 设置合适的 TX & RX fifo size

收发长包都需要更大的 TX &RX fifo size, 考虑到这些 fifo 会占据大量的 Sram 空间, 在设置 fifo size 时应选取最合适的值, 避免 Sram 的浪费。

发长包需要加大 TX fifo size。TX fifo size 至少比 TX rf_len 大 12, 且必须按 4 字节对齐。如:

TX rf_len = 56 bytes: MYFIFO_INIT(blt_txfifo, 68, 8);

TX rf_len = 141 bytes: MYFIFO_INIT(blt_txfifo, 156, 8);

```
TX rf_len = 191 bytes: MYFIFO_INIT(bl_txfifo, 204, 8);
```

收长包需要加大 RX fifo size。RX fifo size 至少比 RX rf_len 大 24，且必须按 16 字节对齐。如：

```
RX rf_len = 56 bytes: MYFIFO_INIT(bl_rxfifo, 80, 8);
```

```
RX rf_len = 141 bytes: MYFIFO_INIT(bl_rxfifo, 176, 8);
```

```
RX rf_len = 191 bytes: MYFIFO_INIT(bl_rxfifo, 224, 8);
```

比如，TX 和 RX 最大长度都需要支持到 200bytes，可按如下设置

```
MYFIFO_INIT(bl_txfifo, 212, 8);
```

```
MYFIFO_INIT(bl_rxfifo, 224, 8);
```

2) data length exchange

在收发长包前，一定要确保在 BLE connection 上 data length exchange 这个流程已经完成。

data length exchange 流程是 Link Layer 上 LL_LENGTH_REQ 和 LL_LENGTH_RSP 两个包的交互过程，slave 和 master 任何一方都可以主动发起 LL_LENGTH_REQ，另一方回应 LL_LENGTH_RSP。通过这两个包的交互后，master 和 slave 都能知道彼此 TX 和 RX 最大包长，然后取两个最大包长中的较小值，即可确定当前 connection 收发包最大包长。

不管是哪一端发起的 LL_LENGTH_REQ，data length exchange 流程结束时，SDK 都会产生 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 事件回调（前提是注册了该事件的回调），user 可参考“Telink defined event”部分的说明来了解该事件回调函数各个参数的含义。

在这个 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 事件回调函数里，可以获得最终的最大 TX 包长和 RX 包长。

在实际应用中，当 8x5x 作为 BLE slave 设备时，master 端可能会主动发起 LL_LENGTH_REQ，也可能不会主动发起。如果 master 端没有主动发起 LL_LENGTH_REQ，就需要由 slave 端主动发起。SDK 提供主动发起 LL_LENGTH_REQ 的 API 如下：

```
ble_sts_t blc_ll_exchangeDataLength (u8 opcode, u16 maxTxOct);
```

API 中 opcode 填“LL_LENGTH_REQ”，maxTxOct 填当前设备支持的最大 TX 包长，比如 TX 最大包长为 200 bytes 时，设置如下

```
blc_ll_exchangeDataLength(LL_LENGTH_REQ, 200);
```

由于 slave 设备并不知道 master 会不会主动发送 LL_LENGTH_REQ，推荐一个参考的方法：注册 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE 事件回调，当 connection 建立后开启一个软件定时器开始计时（比如 2S），如果在 2S 后这个回调一直没有触发，就说明 master 还没有主动发起 LL_LENGTH_REQ，此时 slave 调用 API blc_ll_exchangeDataLength 主动发起 LL_LENGTH_REQ。

Shared under NDA

3) MTU size exchange

除了上面 data length exchange 流程, MTU size exchange 的流程必须也要执行, 确保大的 MTU size 生效, 防止对方设备在 BLE L2cap 层无法处理长包。MTU size 的值需要大于等于 TX & RX 最大包长。

MTU size exchange 的实现, 请参考本文档" ATT & GATT"部分的详细说明, 也可以参考 8258_feature_test 里面 demo 的写法。

4) 收发长包的操作

请 user 先参考本文档"ATT & GATT"部分的一些说明, 包括 Handle Value Notification 和 Handle Value Indication, Write request 和 Write Command 等。

在以上 3 个步骤都正确完成的基础上, 可以开始收发长包。

发长包调用 ATT 层的 Handle Value Notification 和 Handle Value Indication 对应的 API 即可, 分别如下所示, 将要发送的数据地址和长度分别带入下面的形参"*p"和"len"即可。

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

收长包只要处理 Write request 和 Write Command 对应的回调函数"w"即可, 在回调函数里, 引用形参指针指向的数据。

3.2.9 Controller API

3.2.9.1 Controller API 说明

在图 3-1 所示的标准 BLE 协议栈架构中, 应用层是无法与 Controller 的 Link Layer 直接数据交互的, 必须通过 Host 把数据往下发, 最终由 Host 通过 HCI 把控制命令传送给 Link Layer。Host 通过 HCI 接口下发的所有 Controller 控制命令都在 BLE spec 《Core_v5.0》中严格定义了, 详情请参照《Core_v5.0》(Vol 2/Part E/ Host Controller Interface Functional Specification)。

Telink BLE SDK 的设计是按照标准的 BLE 架构设计, 可以作为一个单独的 Controller 与另外一个运行 Host 协议的系统级联工作, 所以所有的操作设置 Link Layer 的 API 都是严格按照 Spec 上 Host command 的数据格式来定义实现的。

虽然 Telink BLE SDK 最终采用了

图 3-4 的架构, 应用层跨越 Host 直接操作设置 Link Layer, 但使用的 API 还是标准的 HCI 部分的 API。下面的 API 具体介绍中会给出 Spec 上对应的 Host command, user 可以参考 Spec 上具体说明加以理解。

BLE spec 上所有的 HCI command，Controller 的处理都会有对应的 HCI command complete event 或 HCI command status event 作为对 Host 层的应答，但在 Telink BLE SDK 中，是分情况处理的：

- 1) 对于 8258_hci 类的应用，Telink 的 IC 只作为 BLE controller，需要和其他家的运行 BLE host 的 MCU 协同工作，每个 HCI command 都会有对应的 HCI command complete event 或 HCI command status event 产生。
- 2) 对于 8258 master kma dongle 应用，BLE Host 和 Controller 都运行在 Telink IC 上，Host 调用 interface 发送 HCI command 给 Controller 时，Controller 全部都能正确收到，不会有丢失的情况，所以 Controller 在处理 HCI command 时不再回复 HCI command complete event 或 HCI command status event。

Controller API 的声明在 stack/ble/ll 和 stack/ble/hci 目录下的头文件中，其中 ll 目录下根据 Link Layer 状态机功能的分类分为 ll.h、ll_adv.h、ll_scan.h、ll_init.h、ll_slave.h、ll_master.h，user 可以根据 Link Layer 的功能去寻找，比如跟 advertising 相关功能的 API 就应该都在 ll_adv.h 中声明。

3.2.9.2 API 返回类型 ble_sts_t

在 stack/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 不仅限于 Link Layer 的 API，对 Host 层一些 API 也适用。

3.2.9.3 BLE MAC address 初始化

本文档中的 BLE MAC address 最基本的类型包括 public address 和 random static address。

该 BLE SDK 中，调用如下接口获得 public address 和 random static address。

```
void blc_initMacAddress(int flash_addr, u8 *mac_public,  
                        u8 *mac_random_static);
```

flash_addr 填 flash 上存储 MAC address 的地址即可，参考文档前面的介绍，8x5x 512K flash 上对应的这个地址为 0x76000。如果不需要 random static address，上面的 mac_random_static 填 "NULL" 即可。

BLE public 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_initStandby_module (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);
```

3.2.9.4 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 (void)
void      blc_ll_initStandby_module (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);
```

3.2.9.5 bls_ll_setAdvData

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.7 “LE Set Advertising Data Command”)。

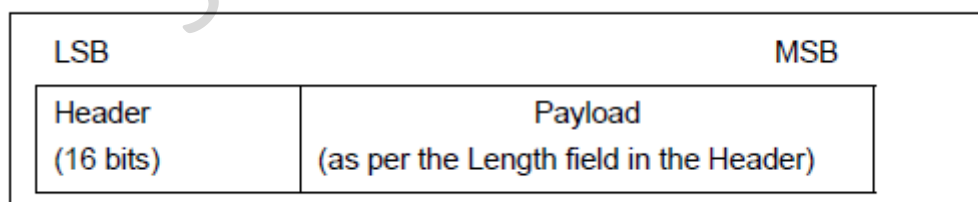


图 3-32 BLE 协议栈广播包格式

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，也可以于程序运行时在 main_loop里随时调用该API来修改广播数据。

该BLE SDK中8258 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 Specification的具体说明。

```
u8 tbl_advData[] = {
    0x05, 0x09, 'k', '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"khid"。

3.2.9.6 bls_ll_setScanRspData

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.8 “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, 也可以于程序运行时在 main_loop 里随时调用该 API 来修改 Scan response data。

该 BLE SDK 中 8258 ble remote 工程中定义 scan response data 如下, scan 设备名为"KRemote"。各个字段的含义请参考文档 BLE Spec 《CSS v6》 (Core Specification Supplement v6.0) 中 Data Type Specfication 的具体说明。

```
u8 tbl_scanRsp [] = {
    0x08, 0x09, 'K', '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"khid";
- 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"KRemote"。

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。

3.2.9.7 bls_ll_setAdvParam

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.5 "LE Set Advertising Parameters Command")。

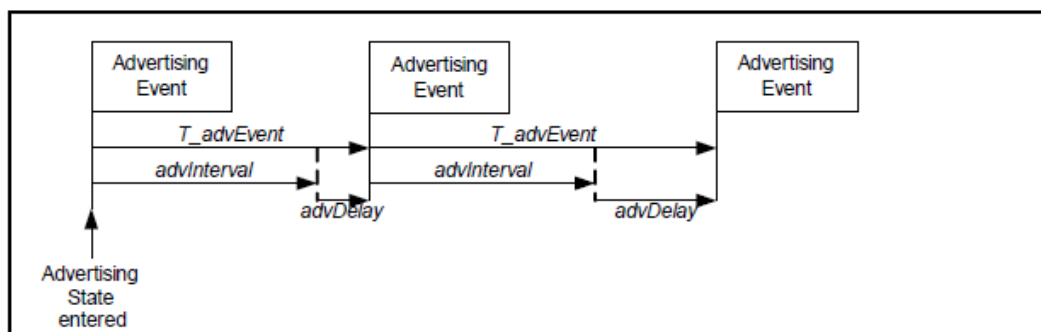


图 3-33 BLE 协议栈里 Advertising Event

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。

下面的 API 对 Adv Event 相关的参数进行设置。

```
ble_sts_t bls_11_setAdvParam( u16 intervalMin, u16 intervalMax,
                             adv_type_t advType, own_addr_type_t ownAddrType,
                             u8 peerAddrType, u8 *peerAddr, u8 adv_channelMap,
                             adv_fp_type_t 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。Telink BLE SDK 通过 intervalMin 和 intervalMax 设置为不同的值来实现, 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, 请参照 (Vol 6/Part B/ 4.4.2.2 “Advertising Events”)。

2) advType

参考 BLE Spec, 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 3-34 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 Undirected 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（stack/ble/ble_common.h）：

```
/* 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)
}adv_type_t;
```

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

3) ownAddrType

指定广播地址类型时，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;
```

这里只介绍前两个参数。

OWN_ADDRESS_PUBLIC 表示广播的时候使用 public MAC address，实际地址来自 MAC address 初始化时 API blc_ll_initAdvertising_module(u8 *public_adr)的设置。

OWN_ADDRESS_RANDOM 表示广播的时候使用 random static MAC address，

该地址来源于下面 API 设定的值：

```
ble_sts_t blc_ll_setRandomAddr(u8 *randomAddr);
```

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。

```
typedef enum {
    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY = 0x00,
    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_ANY = 0x01,
    ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_WL = 0x02,
    ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL = 0x03,
    ADV_FP_NONE = ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY
} adv_fp_type_t;
```

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 HCI 部分 Host command 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。

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。

3.2.9.8 bls_ll_setAdvEnable

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.9 “LE Set Advertising Enable Command”)。

```
ble_sts_t bls_ll_setAdvEnable(int 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。

3.2.9.9 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 Duration 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 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。

3.2.9.10 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);
```

3.2.9.11 rf_set_power_level_index

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

```
void rf_set_power_level_index (RF_PowerTypeDef level)
```

level 值的设置参考 drivers/8258/rf_drv.h 中定义的枚举变量

RF_PowerTypeDef。

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。

需要注意的是：rf_set_power_level_index 这个函数内部是对 MCU RF 相关的一些寄存器进行设置，而一旦 MCU 进入 sleep（包括 suspend/deepsleep retention）后，这些寄存器的值都会丢失。所以 user 需要注意，每次 sleep 唤醒后，这个函数必须得重新设置一遍。比如在 SDK demo 中使用了 BLT_EV_FLAG_SUSPEND_EXIT 事件回调，来确保每次 suspend 醒来 rf power 都被重新设置一遍，如下 code 所示。

```
_attribute_ram_code_ void user_set_rf_power (u8 e, u8 *p, int n)
{
    rf_set_power_level_index (MY_RF_POWER_INDEX);
}

user_set_rf_power(0, 0, 0);
bls_app_registerEventCallback (BLT_EV_FLAG_SUSPEND_EXIT,
                               &user_set_rf_power);
```

3.2.9.12 blc_ll_setScanParameter

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.10 “LE Set Scan Parameters Command”)。

```
ble_sts_t blc_ll_setScanParameter (u8 scan_type,
                                    u16 scan_interval, u16 scan_window,
                                    own_addr_type_t ownAddrType,
                                    scan_fp_type_t 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_interval/scan window

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

3) ownAddrType

指定 scan_req 包地址类型时，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;
```

OWN_ADDRESS_PUBLIC 表示 Scan 的时候使用 public MAC address，实际地址来自 MAC address 初始化时 API `blc_ll_initScanning_module(u8 *public_adr)` 的设置。

OWN_ADDRESS_RANDOM 表示 Scan 的时候使用 random static MAC address，该地址来源于下面 API 设定的值：

```
ble_sts_t blc_ll_setRandomAddr(u8 *randomAddr);
```

4) filter_policy

目前支持的 scan filter policy 为下面两个：

```
typedef enum {  
    SCAN_FP_ALLOW_ADV_ANY= 0x00, //except direct adv address not  
    match  
    SCAN_FP_ALLOW_ADV_WL=0x01, //except direct adv address not match  
    SCAN_FP_ALLOW_UNDIRECT_ADV=0x02, //and direct adv address match  
    initiator's resolvable private MAC  
    SCAN_FP_ALLOW_ADV_WL_DIRECT_ADV_MACTH= 0x03, //and direct adv  
    address match initiator's resolvable private MAC  
} scan_fp_type_t;
```

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，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。

3.2.9.13 blc_ll_setScanEnable

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.11 “LE Set Scan Enable Command”)。

```
ble_sts_t blc_ll_setScanEnable (scan_en_t scan_enable, dupFilter_en_t
filter_duplicate);
```

scan_enable 参数类型有如下 2 个可选值:

```
typedef enum {
    BLC_SCAN_DISABLE = 0x00,
    BLC_SCAN_ENABLE = 0x01,
} scan_en_t;
```

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 参数类型有如下 2 个可选值:

```
typedef enum {
    DUP_FILTER_DISABLE = 0x00,
    DUP_FILTER_ENABLE = 0x01,
} dupFilter_en_t;
```

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	
LL_ERR_CURRENT_STATE_NOT_SUPPORTED_THIS_CMD	见 SDK 中定义	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读该设备的 scan_req。

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。

3.2.9.14 blc_ll_createConnection

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.12 “LE Create Connection Command”)。

```
ble_sts_t blc_ll_createConnection (u16 scan_interval, u16 scan_window,
                                   init_fp_type_t 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

scan_interval 设置 Initiating state 中 Scan 频点切换时间, 单位为 0.625ms。

scan_window 在 Telink 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:

```
typedef enum {
    INITIATE_FP_ADV_SPECIFY = 0x00, //connect ADV specified by host
    INITIATE_FP_ADV_WL = 0x01, //connect ADV in whiteList
} init_fp_type_t;
```

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;
```

OWN_ADDRESS_PUBLIC 表示连接的时候使用 public MAC address, 实际地址来自 MAC address 初始化时 API `blc_ll_initStandby_module (u8 *public_adr)` 的设置。

OWN_ADDRESS_RANDOM 表示连接的时候使用 random static MAC address, 该地址来源于下面 API 设定的值:

```
ble_sts_t blc_ll_setRandomAddr (u8 *randomAddr);
```

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, Telink 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。

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

3.2.9.15 blc_ll_setCreateConnectionTimeout

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

返回值为 BLE_SUCCESS, timeout_ms 单位为 ms。

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

3.2.9.16 blm_ll_updateConnection

详情请参照《Core_v5.0》(Vol 2/Part E/ 7.8.18 “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。

3.2.9.17 bls_ll_terminateConnection

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

该 API 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 的设置详情请参照《Core_v5.0》(Vol 2/Part D/2 “Error Code Descriptions”)。

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);
```

Telink BLE SDK 底层 stack 中，只有一个地方会调用该 API 主动 terminate：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。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。

3.2.9.18 blm_ll_disconnect

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

该 API 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 Telink BLE SDK 对 Slave role 的设计上最多只维持 single

connection，而 Master role 的设计上有 multi connection，所以必须指定需要 disconnect 的 connection handle。

API 返回值如下表。

<code>ble_sts_t</code>	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。

3.2.9.19 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。

3.2.9.20 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( blc_ll_getCurrentState() == BLS_LINK_STATE_ADV)
if( blc_ll_getCurrentState() == BLS_LINK_STATE_CONN )
```

3.2.9.21 blc_ll_getLatestAvgRSSI

当 Link Layer 进入 Slave role 或 Master 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)
```

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。

3.2.9.22 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,  
                               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。

```
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 SDK feature test demo 中的 TEST_WHITELIST:

3.2.10 Coded PHY/2M PHY

3.2.10.1 Coded PHY/2M PHY 介绍

Coded PHY 和 2M PHY 是《Core_5.0》新增加的 Feature, 很大程度上扩展了 BLE 的应用场景, Coded PHY 包含 S2(500kbps)和 S8(125kbps)以适应更远距离的应用, 2M PHY(2Mbps)大大提高了 BLE 带宽。2M PHY/Coded PHY 可以使用在广播通道, 也可以用在连接状态下的数据通道。连接状态下的使用方法在本小节中介绍, 广播通道下使用方法在“Extended Advertising”章节中涉及到。

3.2.10.2 Coded PHY/2M PHY Demo 介绍

Telink 提供的 Kite BLE SDK 中, 为节省 Sram, Coded PHY/2M PHY 默认是关闭

的, 用户如果选择使用此 Feature, 可以手动打开, 打开方法可以参考 Kite BLE SDK 提供的 Demo。

◆ Slave 端可参考 Demo “8258_feature_test”

在 vendor/8258_feature_test/app_config.h 中定义宏

```
#define FEATURE_TEST_MODE TEST_2M_CODED_PHY_CONNECTION
```

◆ Master 端可参考 Demo “8258_master_kma_dongle”

用户也可以选择使用其他厂家的设备, 只要支持 Coded PHY/2M PHY 即可以和 Telink 的 Slave 设备兼容互联。

如果使用 Telink 提供的 SDK, Master 端的 Coded PHY 和 2M PHY 也是默认关闭的, 需要通过下面的方法打开。

在 vendor/8258_master_kma_dongle/app.c 函数 void user_init(void) 中添加 API(SDK 里默认是关闭的)

```
blc_ll_init2MPhyCodedPhy_feature();
```

3.2.10.3 Coded PHY/2M PHY API 介绍:

1. API blc_ll_init2MPhyCodedPhy_feature()

```
void blc_ll_init2MPhyCodedPhy_feature(void)
```

用于使能 2M PHY/Coded PHY。

2. 在“Telink defined event”中增加了一个“BLT_EV_FLAG_PHY_UPDATE”Event, 具体细节请参考前面“Controller Event”章节介绍。

3. API blc_ll_setPhy()

```
ble_sts_t blc_ll_setPhy( u16 connHandle,
    le_phy_prefer_mask_t all_phys, le_phy_prefer_type_t tx_phys,
    le_phy_prefer_type_t rx_phys, le_ci_prefer_t phy_options);
```

BLE Spec 标准接口, 详细请参考《Core_v5.0》(Vol 2/PartE/7.8.49 “LE Set PHY Command”)。

connHandle: slave 应用填 BLS_CONN_HANDLE; master 应该填 BLM_CONN_HANDLE。

其他参数请参考 Spec 定义、结合 SDK 上枚举类型定义和 demo 用法去理解。

4. API blc_ll_setDefaultConnCodingIndication()

```
ble_sts_t blc_ll_setDefaultConnCodingIndication(ble_ci_prefer_t
prefer_CI);
```

非BLE Spec标准接口,当Peer Device通过API blc_ll_setPhy ()主动发出PHY_Req 申请时,被申请方可以通过此API设置本地设备的preferred Encode Mode (S2/S8)。

3.2.11 Channel Selection Algorithm #2

Channel Selection Algorithm #2 是《Core_5.0》中新添加的 Feature,拥有更强的抗干扰能力,有关算法的具体说明请参考《Core_5.0》(Vol 6/Part B/4.5.8.3 “Channel Selection Algorithm #2”)

Kite BLE SDK 中相关 Demo 参考。

◆ Slave 端参考 Demo “8258_feature_test”

在 vendor/8258_feature_test/app_config.h 中定义宏如下

```
#define FEATURE_TEST_MODE TEST_CSA2
```

1. 如果使用《Core_4.2》API 定义的广播,用户可以选择使用或者不使用跳频算法#2, SDK 中默认是不使用的,如果想要使用跳频算法#2,需要通过下面的API 使能。

```
void blc_ll_initChannelSelectionAlgorithm_2_feature(void)
```

2. 如果使用《Core_5.0》API 定义的 Extended Advertising,并且需要通过 Extend Adv 发起连接,就必须通过上面的 API 使能跳频算法#2,这是 Spec 中的规定,必须要支持(因为通过 Extended Adv 发起的连接,连接成功后默认使用跳频算法#2),如果只是使用 Advertising 功能,为节省 Sram,建议不使能跳频算法#2。

◆ Master 端参考 Demo “8258_master_kma_dongle”

默认 master 端跳频算法#2 也是关闭的,如果需要同样也需手动要在 user_init() 中调用同样的 API 使能。

```
void blc_ll_initChannelSelectionAlgorithm_2_feature(void);
```

3.2.12 Extended Advertising

3.2.12.1 Extended Advertising 介绍

Extended Advertising 是《Core_5.0》新添加的 Feature。

由于《Core_5.0》对 Advertising 部分的扩展,SDK 中新添加了几个 API 可以发送《Core_4.2》中定义的广播功能和《Core_5.0》新添加的广播功能,这部分

API 我们会在以后的章节中描述为《Core_5.0》API（下面小节后引用这个名称），而《Core_4.2》API 指“Controller API”章节中介绍的 `bls_ll_setAdvData()`、`bls_ll_setScanRspData()`、`bls_ll_setAdvParam()`，这 3 个 API 只能实现《Core_4.2》定义的广播功能，不能够实现《Core_5.0》的广播功能。

Extended Advertising 主要特点如下：

1. 增加了 Advertising PDUs 的最大数据长度，从《core_4.2》的 Advertising PDU 长度 6—37 bytes 到《core_v5.0》Extended Advertising PDU 的最大长度 0—255 bytes（单个包 PDU 的长度），如果 Advertising Data 数据长度大于 Adv PDU，会自动将分包为 N 个 Advertising PDU 分发。
2. 可根据应用场景灵活选择不同的 PHYs（1Mbps, 2Mbps, 125kbps, 500kbps）。

3.2.12.2 Extended Advertising Demo 搭建

Extended Advertising Demo “8258_feature_test”使用方法：

Demo1：用于说明《Core_5.0》支持的所有的基础广播功能用法。

1. 在 `vendor/8258_feature_test/app_config.h` 定义宏

```
#define FEATURE_TEST_MODE TEST_EXTENDED_ADVERTISING
```

2. 根据想要发送包的类型使能对应的宏， Demo 可以遍历《Core_5.0》所支持的所有广播包类型，所支持类型如下图所示。

```
/* Advertising Event Properties */
typedef enum {
    ADV_EVT_PROP_LEGACY_CONNECTABLE_SCANNABLE_UNDIRECTED = 0x0013, // 0001 0011'b ADV_IND
    ADV_EVT_PROP_LEGACY_CONNECTABLE_DIRECTED_LOW_DUTY = 0x0015, // 0001 0101'b ADV_DIRECT_IND(low duty cycle)
    ADV_EVT_PROP_LEGACY_CONNECTABLE_DIRECTED_HIGH_DUTY = 0x001D, // 0001 1101'b ADV_DIRECT_IND(high duty cycle)
    ADV_EVT_PROP_LEGACY_SCANNABLE_UNDIRECTED = 0x0012, // 0001 0010'b ADV_SCAN_IND
    ADV_EVT_PROP_LEGACY_NON_CONNECTABLE_NON_SCANNABLE_UNDIRECTED = 0x0010, // 0001 0000'b ADV_NONCONN_IND

    ADV_EVT_PROP_EXTENDED_NON_CONNECTABLE_NON_SCANNABLE_UNDIRECTED = 0x0000, // 0000 0000'b ADV_EXT_IND + AUX_ADV_IND/AUX_CHAIN_IND
    ADV_EVT_PROP_EXTENDED_CONNECTABLE_UNDIRECTED = 0x0001, // 0000 0001'b ADV_EXT_IND + AUX_ADV_IND/AUX_CHAIN_IND
    ADV_EVT_PROP_EXTENDED_SCANNABLE_UNDIRECTED = 0x0002, // 0000 0010'b ADV_EXT_IND + AUX_ADV_IND/AUX_CHAIN_IND
    ADV_EVT_PROP_EXTENDED_NON_CONNECTABLE_NON_SCANNABLE_DIRECTED = 0x0004, // 0000 0100'b ADV_EXT_IND + AUX_ADV_IND/AUX_CHAIN_IND
    ADV_EVT_PROP_EXTENDED_CONNECTABLE_DIRECTED = 0x0005, // 0000 0101'b ADV_EXT_IND + AUX_ADV_IND/AUX_CHAIN_IND
    ADV_EVT_PROP_EXTENDED_SCANNABLE_DIRECTED = 0x0006, // 0000 0110'b ADV_EXT_IND + AUX_ADV_IND/AUX_CHAIN_IND

    ADV_EVT_PROP_EXTENDED_MASK_ANONYMOUS_ADV = 0x0020, //if this mask on(only extended ADV event can mask it), anonymous advertising
    ADV_EVT_PROP_EXTENDED_MASK_TX_POWER_INCLUDE = 0x0040, //if this mask on(only extended ADV event can mask it), TX power include
}adv_event_prop_t;
```

Demo2：在 Demo1 广播功能的基础上，添加了 Coded PHY/2M PHY 的选择。

1. 在 `vendor/8258_feature_test/app_config.h` 定义宏

```
#define FEATURE_TEST_MODE TEST_2M_CODED_PHY_EXT_ADV
```

2. 根据需要的包类型和 PHY mode 选择相应的宏使能相关功能。

注意：编译某个 Demo 时，如果出现下图所示的错误提示，可能是因为定义的“`_attribute_data_retention_`”属性的数据 size 超过了 16K，而 SDK 默认的是 `deepsleep retention 16K sram`。

```

C:\TelinkSDK1.3\opt\tc32\bin\tc32-elf-ld.exe: section .text loaded at [00004000,0000ab8b] overlaps section .retention_data loaded at [000037a0,00004fd7]
make: *** [ble_1t_multimode.elf] Error 1

```

可以通过下面的一种方法修改（详细分析请参考“Sram 和 Firmware 空间”章节的介绍）

1. 减少所定义的“_attribute_data_retention_”属性的数据。
2. 选择切换为 deepsleep retention 32K Sram，详细配置方法参考“software bootloader 介绍”章节。

3.2.12.3 Extended Advertising 相关的 API 介绍

Extended Advertising 部分采用模块化的设计，另外考虑到 adv data length/scan response data 长度的不确定性，需要支持最长 1000 多 bytes 的数据长度，而我们又不希望底层写死的 code 导致 Sram 的浪费（因为大部分客户可能使用的数据长度很短），所以将底层需要用到的 Sram 全部交给用户层去定义。

当前 SDK 只支持 1 个 Advertising set 不支持 Multiple Advertising Sets，但是为了方便以后扩展为 multiple adv sets，code 中的 API 都是兼容 Multiple Advertising Sets 而设计的，用户可暂时忽略 multiple adv sets 的参数设置。

根据以上的设计思想，设计了如下的 API。

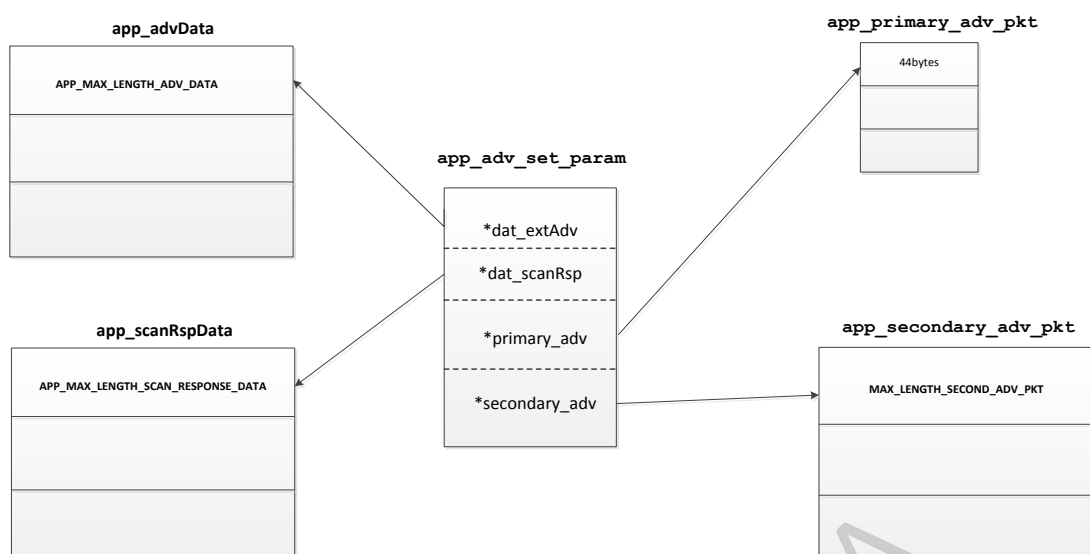
1. 初始化时需要调用如下 API 来分配 Extended Advertising 所使用的 Sram。

```

blc_ll_initExtendedAdvertising_module(app_adv_set_param,
                                     app_primary_adv_pkt, APP_ADV_SETS_NUMBER);
blc_ll_initExtSecondaryAdvPacketBuffer(app_secondary_adv_pkt,
                                       MAX_LENGTH_SECOND_ADV_PKT);
blc_ll_initExtAdvDataBuffer(app_advData, APP_MAX_LENGTH_ADV_DATA);
blc_ll_initExtScanRspDataBuffer(app_scanRspData,
                                APP_MAX_LENGTH_SCAN_RESPONSE_DATA);

```

根据上面 API 调用其内存分配如下所示：



- ◆ APP_MAX_LENGTH_ADV_DATA: 一个Advertising Set 的数据长度，用户需要根据实际应用，调节宏定义大小，以节省deepRetention空间。
- ◆ APP_MAX_LENGTH_SCAN_RESPONSE_DATA: 一个Advertising Set 的 scanResponse数据长度，用户需要根据实际应用，调节宏定义大小，以节省 deepRetention空间。
- ◆ app_primary_adv_pkt: 一个Primary Advertising PDU数据长度的大小，固定分配长度为44 bytes，用户层不能随意修改。
- ◆ app_Secondary_adv_pkt: 一个Secondary Advertising PDU数据长度的大小，固定分配长度为264 bytes，用户层不能随意修改。

在 SDK 提供的 demo“8258_feature_test”中 (vendor/8258_feature_test/feature_2m_coded_phy_adv.c)用户可以根据自己的需求来定义以下的宏来分配 sram，以达到节省 Sram 的目的。

```

#define APP_ADV_SETS_NUMBER 1 // Number of
Supported Advertising Sets (本版SDK只支持1个Adv Set，所以当前只能为1)
#define APP_MAX_LENGTH_ADV_DATA 1024 // Maximum
Advertising Data Length, (if legacy ADV, max length 31 bytes is enough)
#define APP_MAX_LENGTH_SCAN_RESPONSE_DATA 31 // Maximum Scan
Response Data Length, (if legacy ADV, max length 31 bytes is enough)

```

2. API blc_ll_setExtAdvParam()

```
ble_sts_t blc_ll_setExtAdvParam (.....);
```

BLE Spec标准接口，用于设置广播参数，详细请参考《Core_5.0》(Vol 2/Part E/7.8.53 “LE Set Extended Advertising Parameters Command”), 并结合SDK上枚举

类型定义和demo用法去理解。

注意：参数中Advertising_Tx_Power暂不支持选择发送power值，需要另外调用API `void rf_set_power_level_index (RF_PowerTypeDef level)`来配置发送power。

3. API `blc_ll_setExtScanRspData()`

```
ble_sts_t blc_ll_setExtScanRspData(u8 advHandle, data_oper_t
operation, data_fragm_t fragment_prefer, u8 scanRsp_dataLen, u8
*scanRspData);
```

BLE Spec标准接口，用于设置Scan Response Data，详细可参考《Core_5.0》(Vol 2/Part E/7.8.53 “LE Set Extended Scan Response Command”)，并结合SDK上枚举类型定义和demo用法去理解。

4. API `blc_ll_setExtAdvEnable_n()`

```
ble_sts_t blc_ll_setExtAdvEnable_n(u32 extAdv_en, u8 sets_num, u8
*pData);
```

BLE Spec标准接口，用于打开/关闭Extended Advertising，详细可参考《Core_5.0》(Vol 2/Part E/7.8.56 “LE Set Extended Advertising Enable Command”)，并结合SDK上枚举类型定义和demo用法去理解。

但目前SDK只支持1个Adv Sets，所以此API暂时不支持，只是为以后multipleAdv sets预留，不过Telink SDK根据此API功能写了一个简化的API，用来操作打开/关闭1个Adv Sets，执行效率更高。简化的API如下所示，输入参数和返回值和标准的API一样，但只用来设置1个Adv Set。

```
ble_sts_t blc_ll_setExtAdvEnable_1(u32 extAdv_en, u8 sets_num, u8
*pData);
```

5. API `blc_ll_setAdvRandomAddr()`

```
ble_sts_t blc_ll_setAdvRandomAddr(u8 advHandle, u8* rand_addr);
```

BLE Spec标准接口，用于设置设备的Random地址，详细请参考《Core_5.0》(Vol 2/Part E/7.8.4 “LE Set Random Address Command”)，并结合SDK上枚举类型定义和demo用法去理解

6. API `blc_ll_setDefaultExtAdvCodingIndication()`

```
void blc_ll_setDefaultExtAdvCodingIndication(u8 advHandle,
le_ci_prefer_t prefer_CI);
```

非BLE Spec标准接口，用BLE标准API `blc_ll_setExtAdvParam()`设置广播参数时，如果设置为Coded PHY（包含S2和S8）但并没有指出具体的哪种Encode mode。SDK默认为S2，为方便用户选择，定义了此API来选择preferenced Encode mode S2/S8。

7. API `blc_ll_setAuxAdvChnIdxByCustomers()`

```
void blc_ll_setAuxAdvChnIdxByCustomers(u8 aux_chn);
```

非BLE Spec标准接口，用户可以通过此函数设置Auxiliary Advertising channel的通道值，常用于debug用，如果用户没有调用此函数来定义，Auxiliary Advertising channel值会随机生成（随机数范围0--31）。

8. API `blc_ll_setMaxAdvDelay_for_AdvEvent()`

```
void blc_ll_setMaxAdvDelay_for_AdvEvent(u8 max_delay_ms);
```

非BLE Spec标准接口，用于设置在AdvInteval基础上的AdvDelay时间，输入参数范围0, 1, 2, 4, 8, 单位为ms。

```
advDelay(unit: uS) = Random() % (max_delay_ms*1000);  
T_advEvent = advInterval + advDelay
```

如果max_delay_ms = 0, T_advEvent的时间是精确的advInterval时间;
如果max_delay_ms = 8, T_advEvent的时间在advInterval基础上有0 -- 8ms的随机偏移量。

9. 下面的API，为Multiple Advertising Sets API所预留，本版SDK不支持，用户可以暂时忽略。

```
➤ ble_sts_t blc_ll_removeAdvSet(u8 advHandle);  
➤ ble_sts_t blc_ll_clearAdvSets(void);
```

3.3 BLE host

3.3.1 BLE host 介绍

BLE host 包括 L2CAP、ATT、SMP、GATT、GAP 等层，用户层的应用都是基于host 层实现的。

3.3.2 L2CAP

逻辑链路控制与适配协议通常简称为 L2CAP（Logical Link Control and Adaptation Protocol），它向上连接应用层，向下连接控制器层，发挥主机与控制器之间的适配器的作用，使上层应用操作无需关心控制器的数据处理细节。

BLE 的 L2CAP 层是经典蓝牙 L2CAP 层的简化版本，它在基础模式下，不执行分段和重组，不涉及流程控制和重传机制，仅使用固定信道进行通信。L2CAP 的简化结构如下图所示，简单说就是将应用层数据分包发给 BLE controller，将 BLE controller 收到的数据打包成不同 CID 数据上报给 host 层。

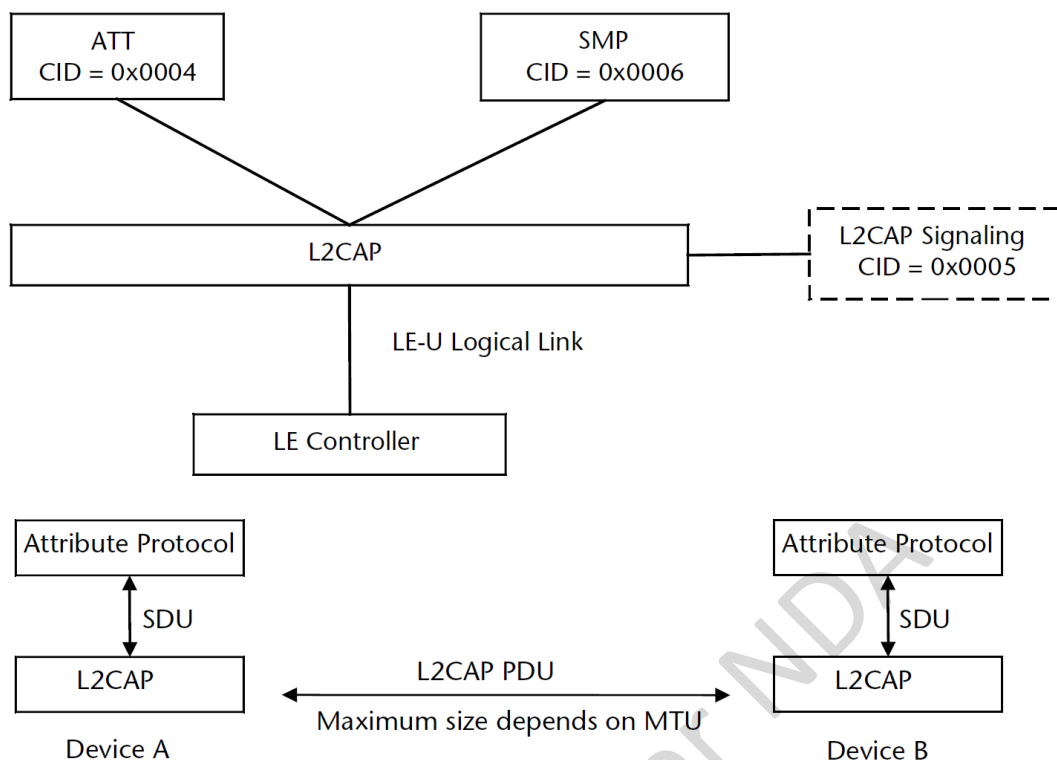


图 3-35 BLE L2CAP 结构以及 ATT 组包模型

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。

3.3.2.1 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 b1c_l2cap_register_handler (void *p);
```

在 8258 remote/8258 module 等 BLE Slave 应用中，SDK L2CAP 层处理 Controller 数据的函数为：

```
int b1c_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、SIG 或 SMP。

初始化：

```
b1c_l2cap_register_handler (b1c_l2cap_packet_receive);
```

在 8258 master kma dongle 中，应用层既包含了 BLE Host 功能，其处理函数如下，是源码提供给 user 参考的：

```
int app_l2cap_handler (u16 conn_handle, u8 *raw_pkt);
```

初始化：

```
btc_l2cap_register_handler (app_l2cap_handler);
```

在 8258 hci 中，只实现了 slave controller，btc_hci_sendACLData2Host 函数将 controller 数据通过 UART/USB 等硬件接口传送给 BLE Host 设备。

```
int btc_hci_sendACLData2Host (u16 handle, u8 *p)
```

初始化： btc_l2cap_register_handler (btc_hci_sendACLData2Host);

3.3.2.2 更新连接参数

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，详情请参照《Core_v5.0》(Vol 3/Part A/ 4.20 “CONNECTION PARAMETER UPDATE REQUEST”)。

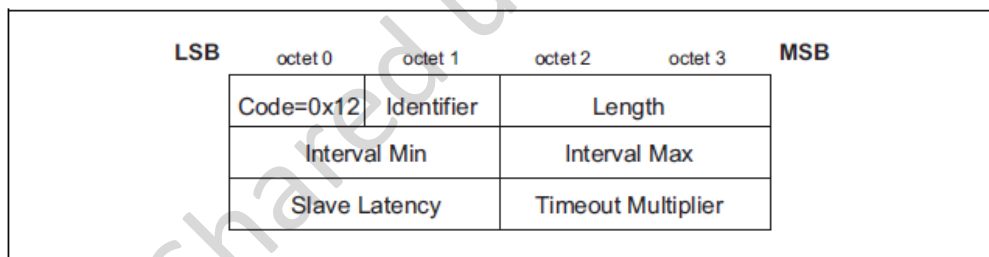


Figure 4.22: Connection Parameters Update Request Packet

图 3-36 BLE 协议栈中 Connection Para update Req 格式

该 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
    bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(1000
);
}
```

Data Type	Data Header					L2CAP Header		SIG Pkt Header			SIG_Connection_Param_Update_Req			
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Code	Id	Data-Length	IntervalMin	IntervalMax	SlaveLatency	TimeoutMultiplier
L2CAP-S	2	1	0	0	16	0x000C	0x0005	0x12	0x01	0x0008	0x0006	0x0006	0x0063	0x0190
Data Type	Data Header					L2CAP Header		SIG Pkt Header			SIG_Connection_Param_Update_Rsp			
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Code	Id	Data-Length	Result	CRC	RSSI (dBm)	FCS
L2CAP-S	2	1	1	0	10	0x0006	0x0005	0x13	0x01	0x0002	0x0000	0x2DE483	-38	OK

图 3-37 抓包显示 conn para update request 和 response

其中 API:

void bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(int time_ms) 用于设置在连接建立后，slave 设备等待 time_ms（单位：毫秒）后执行 API: **bls_l2cap_requestConnParamUpdate**，来更新连接参数。用户如果在连接建立后，只调用 **bls_l2cap_requestConnParamUpdate**，则 slave 设备在连接建立后 1s 再执行发送连接参数更新请求。

在 slave 应用中，SDK 提供了获取 Conn_UpdateRsp 结果的注册回调函数接口，用于通知用户 slave 申请的连接参数请求是否被 master 拒绝还是接受，如上图所示，master 接受了 slave 的 Connection_Param_Update_Req 参数。

```
void
blc_l2cap_registerConnUpdateRspCb (l2cap_conn_update_rsp_callback_t
cb);
```

参考 slave 初始化用例:

```
blc_l2cap_register_handler (blc_l2cap_packet_receive);
```

其中 **blc_l2cap_packet_receive** 函数参考如下:

```
int app_conn_param_update_response(u8 id, u16 result)
{
    if(result == CONN_PARAM_UPDATE_ACCEPT){
        //the LE master Host has accepted the connection parameters
    }
    else if(result == CONN_PARAM_UPDATE_REJECT)
    {
        //the LE master Host has rejected the connection parameter
    }
}
```

```

return 0;
}

```

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，详情请参照《Core_v5.0》(Vol 3/Part A/ 4.20 “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 3-37 所示的抓包显示 master 接受了申请。

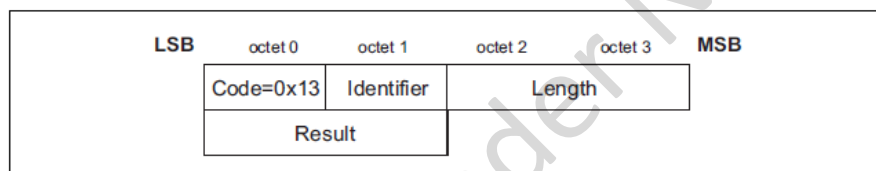


Figure 4.23: Connection Parameters Update Response Packet

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 3-38 BLE 协议栈中 conn para update rsp 格式

Telink 的 8258 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_PARAM_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 req
            //with CONN_PARAM_UPDATE_ACCEPT; if not accept,should send CONN_PARAM_UPDATE_REJECT
            blc_l2cap_SendConnParamUpdateResponse(current_connHandle, CONN_PARAM_UPDATE_ACCEPT); //send

            //if accept, master host should mark this, add will send update conn param req on link layer
            //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); //send
        }
    }
}

```

在 L2CAP_CID_SIG_CHANNEL 上收到 L2CAP_CMD_CONN_UPD_PARAM_REQ 后，首先读取 interval_min（将其作为最终 interval）、supervision timeout 和 long suspend time(interval * (latency+1))，并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。符合这些条件就接受该参数申请，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。

不管是否接受 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受和拒绝。

```

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

```

如果 8258 master kma dongle 接受了 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 main_loop 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_timeout,
                                0, 0 );
    }
}
```

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。

is	Data Type	Data Header					LL_Opcode		LL_Connect_Update_Req					
	Control	LLID	NESN	SN	MD	PDU-Length	Connection Update Req(0x00)		WinSize	WinOffset	Interval	Latency	Timeout	Instant
		3	1	1	0	12			0x02	0x001F	0x0006	0x0063	0x0190	0x006C
is	Data Type	Data Header					CRC	RSSI (dBm)	FCS					
	Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8FE90F	0	OK					
		1	0	1	0	0								

图 3-39 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。

3.3.3 ATT & GATT

3.3.3.1 GATT 基本单位 Attribute

GATT 定义了一种角色：Server 和 Client。该 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。

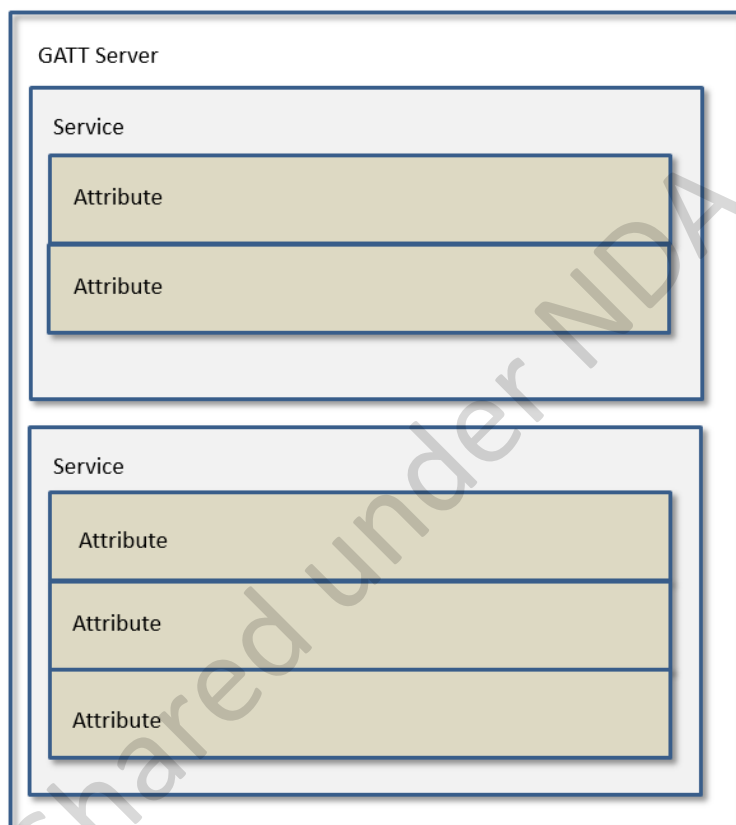


图 3-40 Attribute 构成 GATT service

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。该 BLE SDK 中已经定义了一些标准的 UUID，分布在以下文件中：stack/service/hids.h、stack/ble/uuid.h。

Telink 私有的一些 profile（OTA、MIC、SPEAKER 等），标准蓝牙里面不支持，在 stack/ble/uuid.h 中定义这些私有的 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。在该 BLE SDK 中, Attribute Value 用指针和指针所指区域的长度来描述。

3.3.3.2 Attribute and ATT Table

为了实现 slave 端的 GATT service, 该 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;
```

结合目前该 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:

```
static const attribute_t my_Attributes[] = {

    {ATT_END_H - 1, 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,sizeof(my_devNameCharVal),(u8*)(&my_characterUUID), (u8*)(my_devNameCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_devName), (u8*)(&my_devNameUUID), (u8*)(my_devName), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_appearanceCharVal),(u8*)(&my_characterUUID), (u8*)(my_appearanceCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_appearance), (u8*)(&my_appearanceUUID), (u8*)(my_appearance), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_periConnParamCharVal),(u8*)(&my_characterUUID), (u8*)(my_periConnParamCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_periConnParameters), (u8*)(&my_periConnParamUUID), (u8*)(my_periConnParameter

    // 0008 - 000b gatt
    {4,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID), (u8*)(&my_gattServiceUUID), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_serviceChangeCharVal),(u8*)(&my_characterUUID), (u8*)(my_serviceChangeC
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_serviceChangeVal), (u8*)(&my_serviceChangeUUID), (u8*)(my_serviceChangeC
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_serviceChangeCCC), (u8*)(&my_serviceChangeCCCUUID), (u8*)(my_serviceChangeCCC), 0},

    // 000c - 000e device Information Service
    {3,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID), (u8*)(&my_devServiceUUID), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_PnCharVal),(u8*)(&my_characterUUID), (u8*)(my_PnCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_PnPtrs), (u8*)(&my_PnPUUID), (u8*)(my_PnPtrs), 0},

    ////////////////////////////////// 4. HID Service //////////////////////////////////
    // 000f
    //{27, ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID), (u8*)(&my_hidServiceUUID), 0},
    {HID_CONTROL_POINT_DP_H - HID_PS_H + 1, ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID), (u8*)(&my_hidSe

    // 0010 include battery service
    {0,ATT_PERMISSIONS_READ,2,sizeof(include), (u8*)(&hidIncludeUUID), (u8*)(include), 0},
```

图 3-41 该 BLE SDK Attribute Table 截图

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 上的所有内容都是只读的，不能改写。

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：

```
{57,0,0,0,0,0}, // ATT_END_H - 1 = 57 in "8258_ble_remote"
```

attNum = 57 表示当前 Attribute Table 中共有 57 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 57，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。

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 10 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   0x61
#define ATT_PERMISSIONS_AUTHEN_WRITE  0x62
#define ATT_PERMISSIONS_SECURE_CONN_READ 0xE1
#define ATT_PERMISSIONS_SECURE_CONN_WRITE 0xE2
#define ATT_PERMISSIONS_AUTHOR_READ    0x11
#define ATT_PERMISSIONS_AUTHOR_WRITE   0x12
#define ATT_PERMISSIONS_ENCRYPT_READ    0x21
#define ATT_PERMISSIONS_ENCRYPT_WRITE   0x22
```

注意：目前 SDK 暂不支持授权读和授权写。

3) uuid、uuidLen

按照之前所述，UUID 分两种：BLE 标准的 2 bytes UUID 和 Telink 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 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。

a. 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;

static const u8 my_devNameCharVal[5] = {
    0x12, 0x03, 0x00, 0x00, 0x2A
};

{0,1,2,5,(u8*)&my_characterUUID), (u8*)(my_devNameCharVal), 0},
```

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

b. Telink 私有的 16 bytes UUID:

如 audio 中 MIC 的 Attribute，相关代码：

```
#define TELINK_MIC_DATA
{0x18,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x0}

const u8 my_MicUUID[16]      = TELINK_MIC_DATA;

static u8 my_MicData        = 0x80;

{0,1,16,1,(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。

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，该 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
};

{0,1,2, 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。

us	Data Type	Data Header	Security Enabled	L2CAP Header	ATT_Read_Req	CRC	RSSI	FCS
	L2CAP-S	LLID NESN SN MD PDU-Length	Yes	L2CAP-Length ChanId	Opcode AttHandle	0x65CCC5	0	OK
		2 1 0 0 11		0x0003 0x0004	0x0A 0x0023			
us	Data Type	Data Header	Security Enabled	CRC	RSSI	FCS		
	Empty PDU	LLID NESN SN MD PDU-Length	Yes	0x2A576A	0	OK		
		1 1 1 0 0						
us	Data Type	Data Header	Security Enabled	CRC	RSSI	FCS		
	Empty PDU	LLID NESN SN MD PDU-Length	Yes	0x2A51B9	0	OK		
		1 0 1 0 0						
us	Data Type	Data Header	Security Enabled	L2CAP Header	ATT_Read_Rsp	CRC	RSSI	FCS
	L2CAP-S	LLID NESN SN MD PDU-Length	Yes	L2CAP-Length ChanId	Opcode AttValue	0x9BF6A0	0	OK
		2 0 0 0 13		0x0005 0x0004	0x0B 11 01 00 01			

图 3-42 master 读 hidInformation 的 BLE 抓包

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

```
u8      my_batVal[1]   = {99};

{0,1,2,1,(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。

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 为以下三个时，slave 会检查回调函数 w 是否被设置：

- opcode = 0x12, Write Request
- opcode = 0x52, Write Command
- opcode = 0x18, Execute Write Request

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、Write Command 和 Execute Write Request 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）。

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

图 3-43 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

图 3-44 BLE 协议栈中 Write Command

3.4.6.3 Execute Write Request

The *Execute Write Request* is used to request the server to write or cancel the write of all the prepared values currently held in the prepare queue from this client. This request shall be handled by the server as an atomic operation.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x18 = Execute Write Request

图 3-45 BLE 协议栈中 Execute Write Request

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 `stack/ble/ble_common.h`。

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）。

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：

- A. opcode = 0x0A, Read Request
- B. opcode = 0x0C, Read Blob Request

`slave` 收到以上读命令后，

- A. 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`。
- B. 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。

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。

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

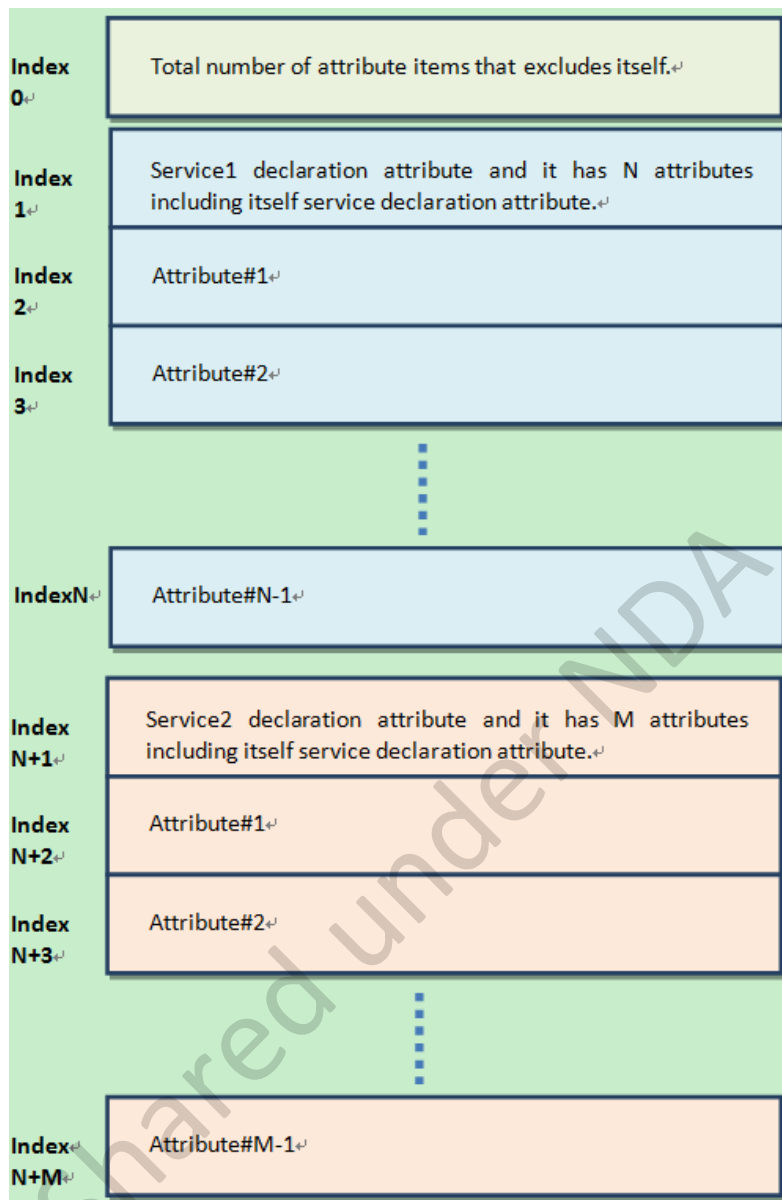


图 3-46 Service/Attribute Layout

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。

3.3.3.3 Attribute PDU & GATT API

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

- ✧ Requests: client 发送给 server 的数据请求。
- ✧ Responses: server 收到 client 的 request 后发送的数据回应。
- ✧ Commands: client 发送给 server 的命令。
- ✧ Notifications: server 发送给 client 的数据。
- ✧ Indications: server 发送给 client 的数据。
- ✧ 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。

1) Read by Group Type Request、Read by Group Type Response

Read by Group Type Request 和 Read by Group Type Response 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.4.9 and 3.4.4.10)。

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。

Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	1	0	11	0x0007	0x0004	0x0004	0x10	0x0001	0xFFFF	00 28	0x89867B	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
Empty PDU	1	0	0	0	0	0xAED0D5							0xAED0D5	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	Length	AttData		CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	0	0	24	0x0014	0x0004	0x0004	0x11	0x06	01 00 07 00 00 18 08 00 0A 00 0A 18 0B 00 25 00 12 18		0x58FC67	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x0004	0x10	0x0026	0xFFFF	00 28	0x5A6275	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
Empty PDU	1	1	1	0	0	0xAED0BA0							0xAED0BA0	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
Empty PDU	1	0	1	0	0	0xAED073							0xAED073	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	Length	AttData		CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	0	0	12	0x0008	0x0004	0x0004	0x11	0x06	26 00 28 00 0F 18		0x158866	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x0004	0x10	0x0029	0xFFFF	00 28	0x055C4D	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
Empty PDU	1	1	1	0	0	0xAED0BA0							0xAED0BA0	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
Empty PDU	1	0	1	0	0	0xAED073							0xAED073	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	Length	AttData		CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	0	0	26	0x0016	0x0004	0x0004	0x11	0x14	29 00 32 00 11 19 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00		0x898D99	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
L2CAP-S	2	1	0	0	11	0x0007	0x0004	0x0004	0x10	0x0033	0xFFFF	00 28	0x3C57D1	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
Empty PDU	1	1	1	0	0	0xAED0BA0							0xAED0BA0	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttGroupType	CRC	RSSI (dBm)	FCS
Empty PDU	1	0	1	0	0	0xAED073							0xAED073	-38	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	ReqOpCode	AttHandle	ErrorCode	CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	0	0	9	0x0005	0x0004	0x0004	0x01	0x10	0x0033	ATT_NOT_FOUND(0x001)	0x600F33	-38	OK

图 3-47 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：

- attHandle 从 0x0001 ~ 0x0007 的 Attribute Group，Attribute Value 为 SERVICE_UUID_GENERIC_ACCESS (0x1800)。
- attHandle 从 0x0008 ~ 0x000a 的 Attribute Group，Attribute Value 为 SERVICE_UUID_DEVICE_INFORMATION (0x180A)。
- attHandle 从 0x000B ~ 0x0025 的 Attribute Group，Attribute Value 为 SERVICE_UUID_HUMAN_INTERFACE_DEVICE (0x1812)。
- attHandle 从 0x0026 ~ 0x0028 的 Attribute Group，Attribute Value 为 SERVICE_UUID_BATTERY (0x180F)。
- attHandle 从 0x0029 ~ 0x0032 的 Attribute Group，Attribute Value 为 TELINK_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。

2) Find by Type Value Request、Find by Type Value Response

Find by Type Value Request 和 Find by Type Value Response 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.3.3 and 3.4.3.4)。

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。

Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Req					CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	AttValue	0x4CEA12	-54	OK
	2	1	1	0	13	0x0009	0x0004	0x06	0x0001	0xFFFF	0x2800	0A 18			
Data Type	Data Header					CRC	RSSI (dBm)	FCS							
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xC4C0E8	-54	OK							
	1	0	0	0	0										
Data Type	Data Header					L2CAP Header		ATT_Find_By_Type_Value_Rsp				CRC	RSSI (dBm)	FCS	
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	HandleInfo			0xF92ED9	-54	OK	
	2	1	0	0	9	0x0005	0x0004	0x07	0C 00 0E 00						

图 3-48 Find by Type Value Request/Find by Type Value Response

3) Read by Type Request、Read by Type Response

Read by Type Request 和 Read by Type Response 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.4.1 and 3.4.4.2)。

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。

Data Type	Data Header					L2CAP Header		ATT_Read_By_Type_Req				
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType	
L2CAP-S	2	0	0	1	11	0x0007	0x0004	0x08	0x0001	0xFFFF	00 2A	0x
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x898717	0	OK				
Empty PDU	1	1	0	0	0							
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x898AB1	0	OK				
Empty PDU	1	1	1	0	0							
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x898C62	0	OK				
Empty PDU	1	0	1	0	0							
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0x8981C4	0	OK				
Empty PDU	1	0	0	0	0							
Data Type	Data Header					L2CAP Header		ATT_Read_By_Type_Rsp				CRC
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData		
L2CAP-S	2	1	0	0	14	0x000A	0x0004	0x09	0x08	03 00 74 53 65 6C 66 69		

图 3-49 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;

{0,1,2, 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。

4) Find information Request、Find information Response

Find information request 和 Find information response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.3.1 and 3.4.3.2)。

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。

Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 1 0 9	L2CAP Header L2CAP-Length ChanId 0x0005 0x0004	ATT_Find_Info_Req Opcode StartingHandle EndingHandle 0x04 0x0016 0x0018	CRC 0x362A2F	RSSI (dBm) -38	FCS
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 0 0 0 0	CRC 0xAE00D5	RSSI (dBm) -38	FCS OK		
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 0 0 0	CRC 0xAE0606	RSSI (dBm) -38	FCS OK		
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 1 1 0 18	L2CAP Header L2CAP-Length ChanId 0x000E 0x0004	ATT_Find_Info_Rsp Opcode Format InfoData 0x05 0x01 16 00 02 29 17 00 08 29 18 00 03 28			

图 3-50 Find information request/Find information response

5) Read Request、Read Response

Read Request 和 Read Response 详请参照《Core_v5.0》(Vol 3/Part F/3.4.4.3 and 3.4.4.4)。

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。

Data Type	Data Header					L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0x99C5FD	-38	OK
L2CAP-S	2	0	1	0	7	0x0003	0x0004	0x0A	0x0017			
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK				
Empty PDU	1	0	0	0	0							
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0606	-38	OK				
Empty PDU	1	1	0	0	0							
Data Type	Data Header					L2CAP Header		ATT_Read_Rsp		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttValue	0x9082A7	-38	OK
L2CAP-S	2	1	1	0	7	0x0003	0x0004	0x0B	02 01			

图 3-51 Read Request/Read Response

6) Read Blob Request、Read Blob Response

Read Blob Request 和 Read Blob Response 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.4.5 and 3.4.4.6)。

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。

Data Type	Data Header					L2CAP Header		ATT_Read_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	0xF4DC27	-38	OK
L2CAP-S	2	0	1	0	7	0x0003	0x0004	0x0A	0x0020			
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK				
Empty PDU	1	0	0	0	0							
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0606	-38	OK				
Empty PDU	1	1	0	0	0							
Data Type	Data Header					L2CAP Header		ATT_Read_Rsp				
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttValue			
L2CAP-S	2	1	1	0	27	0x0017	0x0004	0x0B	05 01 09 02 A1 01 85 01 09 01 A1 00 05 09 19 01 29 03 15 00 25 01			
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle ValueOffset	0x8F3E95	-38	OK
L2CAP-S	2	0	1	0	9	0x0005	0x0004	0x0C	0x0020 0x0016			
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE00D5	-38	OK				
Empty PDU	1	0	0	0	0							
Data Type	Data Header					CRC	RSSI (dBm)	FCS				
Empty PDU	LLID	NESN	SN	MD	PDU-Length	0xAE0606	-38	OK				
Empty PDU	1	1	0	0	0							
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Rsp				
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	PartAttValue			
L2CAP-S	2	1	1	0	27	0x0017	0x0004	0x0D	75 01 95 03 81 02 75 05 95 01 81 01 05 01 09 30 09 31 09 38 15 81			
Data Type	Data Header					L2CAP Header		ATT_Read_Blob_Req		CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle ValueOffset	0x557D8E	-38	OK
L2CAP-S	2	0	1	0	9	0x0005	0x0004	0x0C	0x0020 0x002C			

图 3-52 Read Blob Request/Read Blob Response

7) Exchange MTU Request、Exchange MTU Response

Exchange MTU Request 和 Exchange MTU Response 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.2.1 and 3.4.2.2)。

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

Data Type	Data Header					L2CAP Header		ATT_Exchange_MTU_Req		CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ClientRxMTU			
L2CAP-S	2	0	1	0	7	0x0003	0x0004	0x02	0x009E	0xC70102	-38	OK

Data Type	Data Header					L2CAP Header		ATT_Exchange_MTU_Rsp		CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	ServerRxMTU			
L2CAP-S	2	0	0	0	7	0x0003	0x0004	0x03	0x0017	0x1D88E1	-38	OK

图 3-53 Exchange MTU Request/Exchange MTU Response

当 Telink BLE Slave GATT 层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。

A. 用户可以通过注册 GAP event 回调并开启 eventMask:

GAP_EVT_MASK_ATT_EXCHANGE_MTU 来获取 EffectiveRxMTU，其中：

EffectiveRxMTU=min(ClientRxMTU, ServerRxMTU)。

本文档 “GAP event” 小节会详细介绍 GAP event。

B. 8258 Slave GATT 层收长包数据的处理。

8258 Slave ServerRxMTU 默认为 23，实际最大 ServerRxMTU 可以支持到 250，即 master 端 250 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 ble_att_setRxMtuSize(u16 mtu_size);
```

返回值列表:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
GATT_ERR_INVALID_PARAMETER	见 SDK 中定义	mtu_size 大于最大值 250

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。如果用户注册了 GAP event，并且开启了 `eventMask: GAP_EVT_MASK_ATT_EXCHANGE_MTU`，用户可以在 GAP event 回调函数中获取 EffectiveRxMTU 和 Master 端的 ClientRxMTU。

C. 8258 Slave GATT 层发长包数据的处理。

当 8258 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 connection 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 rsp 后，计算 EffectiveRxMTU。如果用户注册了 GAP event 并且开启了 `eventMask: GAP_EVT_MASK_ATT_EXCHANGE_MTU`，则 EffectiveRxMTU 和 ClientRxMTU 会上报给用户。

8) Write Request、Write Response

Write Request 和 Write Response 详情请参照 《Core_v5.0》(Vol 3/Part F/3.4.5.1 and 3.4.5.2)。

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。

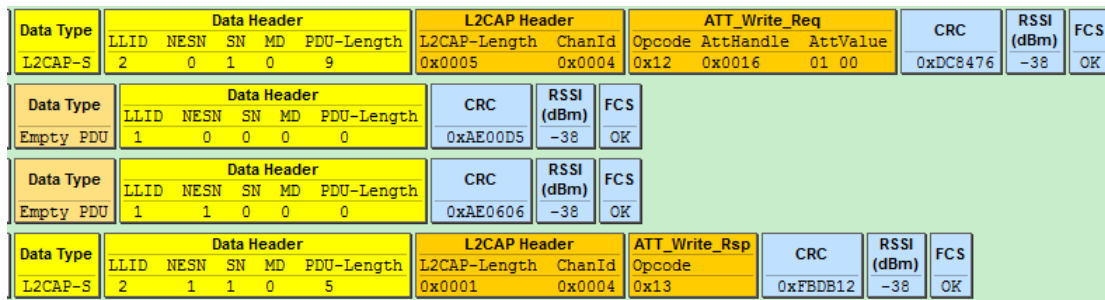


图 3-54 Write Request/Write Response

9) Write Command

Write Command 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.5.3)。

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。

10) Queued Writes

Queued Writes 包含 Prepare Write Request/Response 和 Execute Write Request/Response 等 ATT 协议，详细内容可以参照《Core_v5.0》(Vol 3/Part F/3.4.6/Queued Writes)。

Prepare Write Request 和 Execute Write Request 可以实现如下两种功能：

- 提供长属性值的写入功能。
- 允许在一个单独执行的原子操作中写入多个值。

Prepare Write Request 包含 AttHandle、ValueOffset 和 PartAttValue，这和 Read_Blob_Req/Rsp 类似。这说明 Client 既可以在队列中准备多个属性值，也可以准备一个长属性值的各个部分。这样，在真正执行准备队列之前，Client 可以确定某属性的所有部分都能写入 Server。

备注：当前 SDK 版本仅支持 A.长属性值写入功能，长属性值最大长度小于等于 244 字节。

如下图所示，master 向 slave 某个特性写很长的字符串：“I am not sure what a new song”（字节数远远超过 23，使用默认 MTU 情况下）时，首先发送 Prepare Write Request，偏移 0x0000，将“I am not sure what”部分数据写给 slave，slave 向 master 回 Prepare Write Response。之后 master 发送 Prepare Write Request，偏移 0x12，将“a new song”部分数据写给 slave，slave 向 master 回 Prepare Write Response。当 master 将长属性值全部写完成后，发送 Execute Write Request 给

slave, Flags 为 1: 表示写立即生效, slave 回复 Execute Write Response, 整个 Prepare write 过程结束。

这里我们可以看到 Prepare Write Response 也包含请求中的 AttHandle、ValueOffset 和 PartAttValue。这样做的目的是为了数据传递的可靠性。Client 可以对比 Response 和 Request 的字段值, 确保准备的数据被正确接收。

Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue	CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	1	0	27	0x0017	0x0004	0x0000	0x17	0x0015	0x0000	49 20 61 6D 20 6E 6F 74 20 73 75 72 65 20 77 68 61 74	0x99D4A6	-54	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue	CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	0	0	20	0x0010	0x0004	0x0012	0x16	0x0015	0x0012	20 61 20 6E 65 77 20 73 6F 6E 67	0x99D4A6	-54	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	CRC	RSSI (dBm)	FCS							
Empty PDU	1	1	0	0	0	0x071388	-54	OK							
Data Type	LLID	NESN	SN	MD	PDU-Length	CRC	RSSI (dBm)	FCS							
Empty PDU	1	1	1	0	0	0x071E2E	-54	OK							
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue	CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	1	0	20	0x0010	0x0004	0x0012	0x17	0x0015	0x0012	20 61 20 6E 65 77 20 73 6F 6E 67	0xFF79B4	-54	OK
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue	CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	0	0	6	0x0002	0x0004	0x0000	0x18	0x001	0x0000	0x24D166	-54	OK	
Data Type	LLID	NESN	SN	MD	PDU-Length	CRC	RSSI (dBm)	FCS							
Empty PDU	1	1	0	0	0	0x071388	-54	OK							
Data Type	LLID	NESN	SN	MD	PDU-Length	CRC	RSSI (dBm)	FCS							
Empty PDU	1	1	1	0	0	0x071E2E	-54	OK							
Data Type	LLID	NESN	SN	MD	PDU-Length	CRC	RSSI (dBm)	FCS							
Empty PDU	1	0	0	0	0	0x07155B	-54	OK							
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Header	L2CAP-Length	ChanId	Opcode	AttHandle	ValueOffset	PartAttValue	CRC	RSSI (dBm)	FCS
L2CAP-S	2	0	1	0	5	0x0001	0x0004	0x0000	0x19	0x0015	0x0000	0x49D57	-54	OK	

图 3-55 Write Long Characteristic Values 示例

11) Handle Value Notification

Handle Value Notification 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.7.1)。

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 3-56 BLE Spec 中 Handle Value Notification

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

该 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（根据 EffectiveMaxTxOctets 做分包处理，即链路层 RF TX 最大发送字节数，DLE 可能会修改该值，默认为 27，下文将介绍其替换 API，见备注），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再次 push。

返回值列表如下。

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	
LL_ERR_CONNECTION_NOT_ESTABLISHED	见 SDK 中定义	Link Layer 处于 None Conn state
LL_ERR_ENCRYPTION_BUSY	见 SDK 中定义	处于配对或加密阶段，不能发送数据
LL_ERR_TX_FIFO_NOT_ENOUGH	见 SDK 中定义	有大数据量任务在运行，软件 Tx fifo 不够用
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	见 SDK 中定义	处于遍历服务阶段，不能发数据

备注：SDK 新增了另一个替代 API（使用 min(EffectiveMaxTxOctets, EffectiveRxMTU) 做为分包最小单元，并且 master 和 slave 都可以调用，建议用户使用新 API）：

```
ble_sts_t blc_gatt_pushHandleValueNotify (u16 connHandle, u16
attHandle, u8 *p, int len);
```

调用该 API 时，建议用户检查返回值的是否为 BLE_SUCCESS，并且返回值和 bls_att_pushNotifyData 的返回值有一处差异：1. 处于配对阶段时，新 API 返回值：SMP_ERR_PAIRING_BUSY；2. 处于加密阶段时，新 API 返回值：LL_ERR_ENCRYPTION_BUSY；3. 当 len 大于 ATT_MTU-3 时（3 是 ATT 层的包格式长度 opcode 和 handle），说明要发送的数据长度超出了 ATT 层支持的最大数据长度 ATT_MTU，返回值为 GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE。

12) Handle Value Indication

Handle Value Indication 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.7.2)。

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 3-57 BLE Spec 中 Handle Value Indication

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

该 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（根据 EffectiveMaxTxOctets 做分包处理，即链路层 RF TX 最大发送字节数，DLE 可能会修改该值，默认为 27，下文将介绍其替换 API，见备注），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	
LL_ERR_CONNECTION_NOT_ESTABLISH	见 SDK 中定义	Link Layer 处于 None Conn state
LL_ERR_ENCRYPTION_BUSY	见 SDK 中定义	处于配对或加密阶段，不能发送数据

ble_sts_t	Value	ERR reason
LL_ERR_TX_FIFO_NOT_ENOUGH	见 SDK 中定义	有大数据量任务在运行，软件 Tx fifo 不够用
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	见 SDK 中定义	处于遍历服务阶段，不能发数据
GATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	见 SDK 中定义	前一个 indicate 数据还没有被 master 确认

备注：SDK 新增了另一个替代 API（使用 min(EffectiveMaxTxOctets, EffectiveRxMTU) 做为分包最小单元，并且 master 和 slave 都可以调用，建议用户使用新 API）：

```
ble_sts_t blc_gatt_pushHandleValueIndicate (u16 connHandle, u16
attHandle, u8 *p, int len);
```

调用该 API 时，建议用户检查返回值的是否为 BLE_SUCCESS，并且返回值和 bls_att_pushIndicateData 的返回值有一处差异：1. 处于配对阶段时，新 API 返回值：SMP_ERR_PAIRING_BUSY；2. 处于加密阶段时，新 API 返回值：LL_ERR_ENCRYPTION_BUSY；3. 当 len 大于 ATT_MTU-3 时（3 是 ATT 层的包格式长度 opcode 和 handle），说明要发送的数据长度 PDU 超出了 ATT 层支持的最大 PDU 长度 ATT_MTU，返回值为 GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE。。

13) Handle Value Confirmation

Handle Value Confirmation 详情请参照《Core_v5.0》(Vol 3/Part F/3.4.7.3)。

应用层每调用一次 bls_att_pushIndicateData（或者调用 blc_gatt_pushHandleValueIndicate），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 3-58 BLE Spec 中 Handle Value Confirmation

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，用户可以通过注册 GAP event 回调，并开启相应的 eventMask：

GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM 来获取 Confirm 事件，本文档“GAP

event”小节会详细介绍 GAP event。

3.3.3.4 GATT Service Security

在介绍 GATT Service Security 前，用户可以先了解一下 SMP 相关的内容。

请参考“SMP”章节相关的详细介绍，了解 LE 配对方式、加密等级等基础知识。

下图是 BLE spec 给出的 GATT 服务安全等级服务请求之间映射关系，详细可以参考《core5.0》(Vol3/Part C/10.3 AUTHENTICATION PROCEDURE)。

Link Encryption State	Local Device's Access Requirement for Service	Local Device Pairing Status			
		No LTK No STK	Unauthenticated LTK or Unauthenticated STK	Authenticated LTK or Authenticated STK	Authenticated LTK with Secure Connections
Unencrypted	None	Request succeeds	Request succeeds	Request succeeds	Request succeeds
	Encryption, No MITM Protection	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	Encryption, MITM Protection	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
	Encryption, MITM Protection, Secure Connections	Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption	Error Resp.: Insufficient Encryption
Encrypted	None	N/A (Not possible to be encrypted without LTK)	Request succeeds	Request succeeds	Request succeeds
	Encryption, No MITM Protection		Request succeeds	Request succeeds	Request succeeds
	Encryption, MITM Protection		Error Resp.: Insufficient Authentication	Request succeeds	Request succeeds
	Encryption, MITM Protection, Secure Connections		Error Resp.: Insufficient Authentication	Error Resp.: Insufficient Authentication	Request succeeds

Table 10.2: Local device responds to a service request

图3-59 服务请求响应映射关系

用户可以很清楚的看到：第一列跟当前连接的 slave 设备是否处于加密状态下有关，第二列（local Device's Access Requirement for service）则跟用户设置的 ATT 表中特性的权限（Permission Access）设置有关，如下图所示。而第三列又分为 4 个子列，这 4 个子列则对应当前 LE 安全模式 1 下四个级别（具体说就是当前的设备配对状态是否是如下 4 种中的一种：1、No authentication and no

encryption; 2、Unauthenticated pairing with encryption; 3、Authenticated pairing with encryption; 4、Authenticated LE Secure Connections)。

```

/** @defgroup ATT_PERMISSIONS_BITMAPS GAP ATT Attribute Access Permissions Bit Fields
 * @{
 * (See the Core_v5.0 (Vol 3/Part C/10.3.1/Table 10.2) for more information)
 */
#define ATT_PERMISSIONS_AUTHOR 0x10 //Attribute access(Read & Write) requires Authorization
#define ATT_PERMISSIONS_ENCRYPT 0x20 //Attribute access(Read & Write) requires Encryption
#define ATT_PERMISSIONS_AUTHEN 0x40 //Attribute access(Read & Write) requires Authentication(MITM protection)
#define ATT_PERMISSIONS_SECURE_CONN 0x80 //Attribute access(Read & Write) requires Secure Connection
#define ATT_PERMISSIONS_SECURITY (ATT_PERMISSIONS_AUTHOR | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN | ATT_PERMISSIONS_SECURE_CONN)

//user can choose permission below
#define ATT_PERMISSIONS_READ 0x01 //!< Attribute is Readable
#define ATT_PERMISSIONS_WRITE 0x02 //!< Attribute is Writable
#define ATT_PERMISSIONS_RDWR (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_WRITE) //!< Attribute is Readable & Writable

#define ATT_PERMISSIONS_ENCRYPT_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_ENCRYPT) //!< Read requires Encryption
#define ATT_PERMISSIONS_ENCRYPT_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT) //!< Write requires Encryption
#define ATT_PERMISSIONS_ENCRYPT_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_ENCRYPT) //!< Read & Write requires Encryption

#define ATT_PERMISSIONS_AUTHEN_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Read requires Encryption
#define ATT_PERMISSIONS_AUTHEN_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Write requires Encryption
#define ATT_PERMISSIONS_AUTHEN_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Read & Write requires Encryption

#define ATT_PERMISSIONS_SECURE_CONN_READ (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)
#define ATT_PERMISSIONS_SECURE_CONN_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)
#define ATT_PERMISSIONS_SECURE_CONN_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)

```

图3-60 ATT Permission定义

最终 GATT service security 的实现跟 SMP 初始化时的参数配置包括支持的最高安全级别设置、ATT 表中的特性权限设置等都有关系,而且跟 master 也有关系,比如我们 slave 设置的 SMP 能支持的最高等级是 Authenticated pairing with encryption,但是 master 具备的最高安全等级是 Unauthenticated pairing with encryption,此时如果 ATT 表中某个写特性的权限是 ATT_PERMISSIONS_AUTHEN_WRITE,那么 master 在写该特性时,我们会回复加密等级不够的错误。

用户可以设定 ATT 表中特性权限实现如下应用:

比如 slave 设备支持的最高安全级别是 Unauthenticated pairing with encryption,但是不想连接后使用发送 Security Request 这种方式去触发 master 开始配对,那么客户可以将某些具备 notify 属性的客户端特性配置 (Client Characteristic Configuration, 简称 CCC) 属性的权限设置为 ATT_PERMISSIONS_ENCRYPT_WRITE,那么 master 只有写该 CCC 后,slave 会回复其安全级别不够,这会触发 master 开启配对加密流程。

需要注意的是:用户设置的安全级别只表示设备能支持的最高安全级别,只要 ATT 表中特性的权限 (ATT Permission) 不超过实际生效的最高级别就可以通过 GATT service security 管控。对于 LE 安全模式 1 中的等级 4 来说,如果用户只设置 Authenticated LE Secure Connections 一种级别,则代表当前设置支持 LE Secure Connections only。

GATT 安全级别的示例用户可以参 8258_feature_test/feature_gatt_security.c。

3.3.3.5 8258 master GATT

在 8258 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。

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
    }
}
```

3.3.4 SMP

Security Manager (SM) 在 BLE 中的主要目的是为 LE 设备提供加密所需要的各种 Key，确保数据的机密性。加密链路可以确保避免第三方“攻击者”拦截、破译或者读取空中数据原始内容。SMP 详细内容请用户参考《Core_v5.0》(Vol 3/Part H/ Security Manager Specification)。

3.3.4.1 SMP 安全等级

BLE 4.2 Spec 新增了一种称作安全连接 (LE Secure Connections) 配对方式，新配对方式在安全性方面得到进一步增强，而 BLE4.2 以前的配对方式我们统称传统配对 (LE legacy pairing)。

回顾“GATT service Security”小节，可知本地设备的配对状态类型如下：

Local Device Pairing Status			
No LTK No STK	Unauthenticated LTK or Unauthenticated STK	Authenticated LTK or Authenticated STK	Authenticated LTK with Secure Connections

图 3-61 本地设备配对状态

这四个状态分别对应 LE 安全模式 1 的四个级别，详情可以参考《Core_v5.0》(Vol 3//Part C/10.2 LE SECURITY MODES)：

- A. No authentication and no encryption (LE security **mode1 level1**);
- B. Unauthenticated pairing with encryption (LE security **mode1 level2**);
- C. Authenticated pairing with encryption (LE security **mode1 level3**);
- D. Authenticated LE Secure Connections (LE security **mode1 level4**).

注意：本端设备设定的安全级别只表示本端设备可能达到的最高安全级别，想要达到设定的安全级别跟两个因素有关：1、master 对端设定能支持的最高安全级别>=slave 本端设定能支持的最高安全级别；2、本端和对端按照各自设定的 SMP 参数正确处理完配对整个流程（如果存在配对的话）。

举例来说，用户即便是设置 slave 端能够支持的最高安全等级是 mode1 level3，但是连接 slave 的 master 设置为不支持配对加密（最高只支持 mode1 level1），那么连接后 slave 和 master 不会进行配对流程，slave 实际使用的安全级别是 mode1 level1。

用户可以通过如下 API 设置 SM 能支持的最高安全等级：

```
void blc_smp_setSecurityLevel(le_security_mode_level_t mode_level);
```

枚举类型 le_security_mode_level_t 具体定义介绍如下：

```
typedef enum {
    LE_Security_Mode_1_Level_1 = BIT(0),
    No_Authentication_No_Encryption = BIT(0), No_Security = BIT(0),
    LE_Security_Mode_1_Level_2 = BIT(1),
    Unauthenticated_Paring_with_Encryption = BIT(1),
    LE_Security_Mode_1_Level_3 = BIT(2),
    Authenticated_Paring_with_Encryption = BIT(2),
    LE_Security_Mode_1_Level_4 = BIT(3),
    Authenticated_LE_Secure_Connection_Paring_with_Encryption = BIT(3),
    .....
}le_security_mode_level_t;
```

3.3.4.2 SMP 参数配置

Telink BLE SDK 中 SMP 参数配置介绍主要围绕 SMP 四个安全等级的配置展开，slave 的 SMP 功能目前能支持的最高级别是 LE security mode1 level4；master 的 SMP 功能目前支持传统配对方式下的最高级别是 LE security mode1 level2（传统配对 Just Works 方式）。

1) LE security mode1 level1

安全级别 1 表示设备不支持加密配对，如果需要禁用 SMP 功能，用户只需要在初始化地方需调用如下函数：

```
blc_smp_setSecurityLevel(No_Security);
```

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。

Data Type		Data Header					L2CAP Header		SM_Pairing_Req							CRC
L2CAP-S		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	IOCap	OOBDataFlag	AuthReq	MaxEncKeySize	InitKeyDist	RespKeyDist	
		2	1	1	0	11	0x0007	0x0006	0x01	0x04	0x00	0x05	0x10	0x07	0x07	0x000014

Data Type		Data Header					CRC	RSSI (dBm)	FCS
Empty PDU		LLID	NESN	SN	MD	PDU-Length	0x000014	-54	OK
		1	0	1	0	0			

Data Type		Data Header					CRC	RSSI (dBm)	FCS
Empty PDU		LLID	NESN	SN	MD	PDU-Length	0x000015	-62	OK
		1	0	0	0	0			

Data Type		Data Header					L2CAP Header		SM_Pairing_Failed		CRC	RSSI (dBm)	FCS
L2CAP-S		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Reason			
		2	1	0	0	6	0x0002	0x0006	0x05	0x05	0x00000E	-54	OK

图3-62 抓包显示Pairing Disable

2) LE security mode1 level2

安全级别 2 表示设备最高支持 Unauthenticated_Paring_with_Encryption, 如传统配对和安全连接配对方式下的 Just Works 配对模式。

- A. 通过前文 SMP 基本概念的描述, 我们知道 SM 配对方法包括传统加密和安全连接配对, SDK 提供了如下 API 用于设置是否支持 BLE4.2 新加密特性:

```
void blc_smp_setParingMethods (paring_methods_t method);
```

枚举类型 paring_methods_t 具体定义介绍如下:

```
typedef enum {
    LE_Legacy_Paring      = 0,    // BLE 4.0/4.2
    LE_Secure_Connection = 1,    // BLE 4.2/5.0/5.1
}paring_methods_t;
```

- B. 使用 LE security mode1 level1 以外的安全级别配置就必须要调用如下 API 用于初始化 SMP 各参数配置, 包括绑定区域 FLASH 的初始化配置:

```
int blc_smp_peripheral_init (void);
```

如果在初始化阶段只调用了该 API, 则 SDK 会使用默认参数去配置 SMP:

- ✧ 默认支持的最高安全等级: Unauthenticated_Paring_with_Encryption;
- ✧ 默认绑定模式: Bondable_Mode (存储配对加密后分发的 KEY 到 FLASH);
- ✧ 默认 IO 能力是 IO_CAPABILITY_NO_INPUT_NO_OUTPUT。

以上默认参数是按照传统配对 Just Works 模式配置的, 所以用户只调用该 API, 相当于配置了 LE security mode1 level2, 通过 A 和 B 我们知道 LE security mode1 level2 有两种配置:

- A. 设备具备在传统配对下 Just works 的初始化配置:

```
blc_smp_peripheral_init();
```

- B. 设备具备在安全连接下 Just works 的初始化配置:

```
blc_smp_setParingMethods(LE_Secure_Connection);
blc_smp_peripheral_init();//SMP 参数配置必须放在该 API 之前
```

3) LE security mode1 level3

安全级别 3 表示设备最高支持 Authenticated pairing with encryption，如传统配对模式下的 Passkey Entry、Out of Band 等。

该级别需要设备支持 Authentication，也就是需要通过某种方法确保配对双方身份的合法性，BLE 给出了如下三种 Authentication 方式：

- 1、有人参与的方式，如设备具备按键或者显示能力，通过一方显示 TK，另一方输入相同的 TK（比如 Passkey Entry）；
- 2、配对的双方通过非 BLE RF 传输方式交互一些信息，进行后续的配对操作（如 Out of Band，一般通过 NFC 传输 TK）；
- 3、设备自行协商 TK（如 Just Works，两端设备均使用 TK: 0）。需要注意的是第 3 种方式属于 Unauthenticated，所以 Just works 的安全级别对应 LE security mode1 level2。

Authentication 能确保配对双方身份的合法性，提供这种方式的保护又可以称为 MITM（Man in the Middle）中间人保护。

- A. 具备 Authentication 的设备需要设置其 MITM flag 或 OOB flag，SDK 提供如下两个 API 用于设置 MITM 和 OOB flag 的值：

```
void blc_smp_enableAuthMITM (int MITM_en);  
void blc_smp_enableOobAuthentication (int OOB_en);
```

其中参数 MITM_en、OOB_en 的值为 0 或者 1，0 对应失能；1 对应使能。

- B. 根据 Authentication 方式的介绍，SM 提供了三类鉴权方式，这三类方式的选择依赖于配对双方具备的 IO 能力，我们 SDK 提供了如下接口用于配置当前设备具备的 IO 能力：

```
void blc_smp_setIoCapability (io_capability_t ioCapablility);
```

枚举类型 io_capability_t 具体定义介绍如下：

```
typedef enum {  
    IO_CAPABILITY_UNKNOWN = 0xff,  
    IO_CAPABILITY_DISPLAY_ONLY = 0,  
    IO_CAPABILITY_DISPLAY_YES_NO = 1,  
    IO_CAPABILITY_KEYBOARD_ONLY = 2,  
    IO_CAPABILITY_NO_INPUT_NO_OUTPUT = 3,  
    IO_CAPABILITY_KEYBOARD_DISPLAY = 4,  
} io_capability_t;
```

C. 传统配对模式下 MITM、OOB flag 使用规则:

		Initiator			
		OOB Set	OOB Not Set	MITM Set	MITM Not Set
Responder	OOB Set	Use OOB	Check MITM		
	OOB Not Set	Check MITM	Check MITM		
	MITM Set			Use IO Capabilities	Use IO Capabilities
	MITM Not Set			Use IO Capabilities	Use Just Works

Table 2.6: Rules for using Out-of-Band and MITM flags for LE legacy pairing

图3-63 传统配对模式下MITM、OOB flag使用规则

设备会根据本地设备和对端设备的 OOB 以及 MITM flag 决定使用 OOB 方式还是根据 IO 能力决定选择什么样的 KEY 产生方式。下图是 SDK 根据 IO 能力映射关系选择不同的 KEY 产生方法(行、列参数类型 io_capability_t):

```
// H: Initiator Capabilities
// V: Responder Capabilities
// See the Core v5.0 (Vol 3/Part H/2.3.5.1) for more information.
static const stk_generationMethod_t gen_method_legacy[5 /*Responder*/][5 /*Initiator*/] = {
    { JustWorks, JustWorks, PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { JustWorks, JustWorks, PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsply_Resp_Input },
    { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_Resp_Dsply_Init_Input, JustWorks, PK_Init_Dsply_Resp_Input }
};

#if SECURE_CONNECTION_ENABLE
static const stk_generationMethod_t gen_method_sc[5 /*Responder*/][5 /*Initiator*/] = {
    { JustWorks, JustWorks, PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { JustWorks, Numric_Comparison, PK_Resp_Dsply_Init_Input, JustWorks, Numric_Comparison },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsply_Resp_Input },
    { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
    { PK_Init_Dsply_Resp_Input, Numric_Comparison, PK_Resp_Dsply_Init_Input, JustWorks, Numric_Comparison },
};
#endif
```

图3-64 根据不同IO能力映射KEY产生方法

这部分具体映射关系可以参考《core5.0》(Vol3/Part H/2.3.5.1 Selecting Key Generation Method), 文档不再展开介绍。

根据以上介绍, 我们知道 LE security mode1 level3 有如下几种初始值配置方式:

A. 设备具备传统配对下 OOB 的初始化配置:

```
blc_smp_enableOobAuthentication(1);
blc_smp_peripheral_init();//SMP 参数配置必须放在该 API 之前
```

这里因为涉及到 OOB 传输 TK 值, SDK 在应用层提供了相关的 GAP event 给用户, 请参考“GAP event”章节。提供给用户设置 OOB TK 值的 API 如下:

```
void blc_smp_setTK_by_OOB (u8 *oobData);
```

参数 oobData 表示需要设置的 16 位 TK 值数组的头指针。

- B. 设备具备传统配对下 Passkey Entry (PK_Resp_Dsply_Init_Input) 的初始化配置:

```
btc_smp_enableAuthMITM(1);
btc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_ONLY);
btc_smp_peripheral_init();//SMP 参数配置必须放在该 API 之前
```

- C. 设备具备传统配对下 Passkey Entry (PK_Init_Dsply_Resp_Input 或者 PK_BOTH_INPUT) 的初始化配置:

```
btc_smp_enableAuthMITM(1);
btc_smp_setIoCapability(IO_CAPABILITY_KEYBOARD_ONLY);
btc_smp_peripheral_init();//SMP 参数配置必须放在该 API 之前
```

这里因为涉及到用户输入 TK 值, SDK 在应用层提供了相关的 GAP event 给用户, 请参考“GAP event”章节。提供给用户设置 Passkey Entry 的 TK 值 API 如下:

```
void btc_smp_setTK_by_PasskeyEntry (u32 pinCodeInput);
```

参数 pinCodeInput 表示设置的 pincode 值, 范围在“0~999999”。在 Passkey Entry 方式下, master 显示 TK, slave 需要输入 TK 的情况下使用。

最终设备使用何种 Key 产生方式是基于配对连接的两端设备支持什么样的 SMP 安全等级的, 如果 master 只支持安全等级 LE security mode1 level1, 那么最终 slave 是不会启用 SMP 功能的, 因为 master 不支持配对加密。

4) LE security mode1 level4

安全级别 4 表示设备最高支持 Authenticated LE Secure Connections, 如安全连接配对模式下的 Numeric Comparison、Passkey Entry、Out of Band 等。

根据以上介绍, 我们知道 LE security mode1 level4 有如下几种初始值配置方式:

- A. 设备具备安全连接配对下 Numeric Comparison 的初始化配置:

```
btc_smp_setPairingMethods(LE_Secure_Connection);
btc_smp_enableAuthMITM(1);
btc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_YESNO);
```

这里因为涉及到向用户显示数值比较值, SDK 在应用层提供了相关的 GAP event 给用户, 请参考“GAP event”章节。提供给用户设置数值比较结果“YES”或“NO”值的 API 如下:

```
void btc_smp_setNumericComparisonResult (bool YES_or_NO);
```

参数 YES_or_NO: 在数值比较配对方式下, 用于给用户确认比较两端显示的数值是否一致。当用户确认显示的 6 位数值和对端一致时, 可以输入 1: “YES”, 不一致则输 0: “NO”。

B. 设备具备安全连接配对下 Passkey Entry 的初始化配置:

这部分用户初始化代码和 LE security mode1 level3 配置方式 B、C (传统配对 Passkey Entry) 基本一致, 唯一不同的是需要在初始化最开始的地方设置配对方式为“安全连接配对”:

```
btc_smp_setPairingMethods(LE_Secure_Connection);
.....//参考 LE security mode1 level3 配置方式 B、C
```

C. 设备具备安全连接配对下 Out of Band 的初始化配置:

该部分目前 SDK 并未实现, 所以这里就不再做具体介绍。

5) 下面再介绍几个额外的 SMP 参数配置相关的 API:

A. SDK 提供是否需要开启绑定功能的 API:

```
void btc_smp_setBondingMode(bonding_mode_t mode);
```

枚举类型 bonding_mode_t 具体定义介绍如下:

```
typedef enum {
    Non_Bondable_Mode = 0,
    Bondable_Mode = 1,
}bonding_mode_t;
```

配置安全级别非 mode1 level1 的设备, 必须要使能绑定功能, SDK 已经默认开启了, 所以一般情况下用户不需要再调用该 API。

B. SDK 提供是否需使能 Key Press 功能的 API:

```
void btc_smp_enableKeypress (int keyPress_en);
```

表示 Passkey Entry 期间是否支持为 KeyboardOnly 设备提供一些必要的输入状态信息, 因为目前 SDK 不支持该功能, 所以参数必须设置为 0。

C. 在安全连接方式下是否启用 Debug 椭圆加密密钥对:

```
void btc_smp_setEcdhDebugMode(ecdh_keys_mode_t mode);
```

枚举类型 ecdh_keys_mode_t 具体定义介绍如下:

```
typedef enum {
    non_debug_mode = 0, //ECDH distribute private/public key pairs
    debug_mode = 1, //ECDH use debug mode private/public key pairs
} ecdh_keys_mode_t;
```

该 API 仅在安全连接配对情况下使用，由于安全连接配对情况下使用了椭圆加密算法，可以有效避免窃听，这对调试开发就不那么友好了，用户无法通过 sniffer 工具抓取 BLE 空中包，进而进行数据分析调试，所以 BLE spec 也规定给出了一组用于 Debug 的椭圆加密私钥/公钥对，只要开启这个模式，BLE sniffer 工具就可以用已知的密钥去解密链路。

- D. 通过如下 API 设置 SM 是否绑定、是否开启 MITM flag、是否支持 OOB、是否支持 Keypress notification，以及支持的 IO 能力（文档前面给的都是单独的配置 API，为了用户设置方便，SDK 也提供了统一配置 API）：

```
void blc_smp_setSecurityParameters (bonding_mode_t mode, int MITM_en,
int OOB_en, int keyPress_en, io_capability_t ioCapablility);
```

前面已经介绍了每个参数含义，这里就不再重复了。

3.3.4.3 SMP 安全请求配置

SMP 安全请求（Security Request）只有 slave 可以发送，所以这部分只对 slave 设备来说。

我们知道配对流程阶段 1 有一个可选的安全请求包（Security Request），该包的目的是使 slave 可以主动触发配对流程的开始。SDK 提供了如下 API 用于灵活地配置 slave 是否在连接后或者回连后立即还是 pending_ms 毫秒后再向 master 发送 Security Request，亦或是不发送 Security Request 以实现不同的配对触发组合：

```
blc_smp_configSecurityRequestSending( secReq_cfg newConn_cfg,
secReq_cfg reConn_cfg, u16 pending_ms);
```

枚举类型 secReq_cfg 具体定义介绍如下：

```
typedef enum {
    SecReq_NOT_SEND = 0,
    SecReq_IMM_SEND = BIT(0),
    SecReq_PEND_SEND = BIT(1),
} secReq_cfg;
```

每个参数的意义介绍如下：

SecReq_NOT_SEND：连接建立后，slave 不会主动发送 Security Request；

SecReq_IMM_SEND：连接建立后，slave 会立即发送 Security Request；

SecReq_PEND_SEND：连接建立后，slave 等待 pending_ms（单位毫秒）后再决定是否发送 Security Request（1、首次连接，slave 在 pending_ms 毫秒前就收到 master 的 Pairing_request 包也不会再发送 Security Request；2、在回连阶段，pending_ms 毫秒之前如果 master 已经发送 LL_ENC_REQ 加密回连链路，则不再发送 Security Request）。

newConn_cfg: 用于配置新设备, **reConn_cfg**: 用于配置回连的设备。这里 SDK 在回连时也提供配置是否发配对请求的目的: 配对绑定过的设备, 下次再连接的时候 (即回连), **master** 有时候不一定会主动发起 **LL_ENC_REQ** 来加密链路, 此时如果 **slave** 发一下 **Security Request** 就会去触发 **master** 主动加密链路, 所以 SDK 提供了 **reConn_cfg** 配置, 客户可以根据实际需要配置。

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。

函数 **blc_smp_configSecurityRequestSending** 的输入参数有如下 9 种组合:

reConn_cfg newConn_cfg	SecReq_NOT_SEND	SecReq_IMM_SEND	SecReq_PEND_SEND
SecReq_NOT_SEND	第一次连接或者回连都不发 SecReq (参数 pending_ms 无效)	第一次连接不发 SecReq, 回连立即发 SecReq (参数 pending_ms 无效)	第一次连接不发 SecReq, 回连 pending_ms 毫秒后发 SecReq (*见前面参数说明)
SecReq_IMM_SEND	第一次连接立即发 SecReq, 回连不发 SecReq (参数 pending_ms 无效)	第一次连接或者回连都立即发 SecReq (参数 pending_ms 无效)	第一次连接立即发 SecReq, 回连 pending_ms 毫秒后发 SecReq (*见前面参数说明)
SecReq_PEND_SEND	第一次连接 pending_ms 毫秒后发 SecReq (*见前面参数说明), 回连不发 SecReq	第一次连接 pending_ms 毫秒后发 SecReq (*见前面参数说明), 回连立即发 SecReq	第一次连接或者回连都 pending_ms 毫秒后发 SecReq (*见前面参数说明)

我们挑其中两组做一下详细说明, 其他组合类似, 不做具体介绍:

- 1) **newConn_cfg**: **SecReq_NOT_SEND**,
reConn_cfg: **SecReq_NOT_SEND**,
pending_ms: 此时该参数不起作用。
newConn_cfg: **SecReq_NOT_SEND** 表示新设备 **slave** 不会主动发起 **Security Request**, 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** 发送配对请求包 **SM_Pairing_Req** 时, **slave** 会响应, 但是不会主动触发 **master** 发起配对请求。

Data Type						CRC		RSSI (dBm)		FCS																									
Empty PDU						1		1		0		0		0x00000D		-54		OK																	
Data Header						L2CAP Header						SM_Pairing_Req						CRC																	
L2CAP-S						2		1		1		0		11		L2CAP-Length		ChanId		Opcode		IOCap		OOBDataFlag		AuthReq		MaxEncKeySize		InitKeyDist		RespKeyDist		0x000008	
Data Header						CRC		RSSI (dBm)		FCS																									
Empty PDU						1		0		1		0		0		0x00001C		-54		OK															
Data Header						CRC		RSSI (dBm)		FCS																									
Empty PDU						1		0		0		0		0		0x00000C		-78		OK															
Data Type						L2CAP Header						SM_Pairing_Rsp						CRC																	
L2CAP-S						2		1		0		11		L2CAP-Length		ChanId		Opcode		IOCap		OOBDataFlag		AuthReq		MaxEncKeySize		InitKeyDist		RespKeyDist		0x000012			

图3-65 抓包显示Pairing Peer Trigger

reConn_cfg: SecReq_NOT_SEND表示设备已经配对，回连时slave设备不会发送Security Request。

- newConn_cfg: SecReq_IMM_SEND,
reConn_cfg: SecReq_NOT_SEND,
pending_ms: 此时该参数不起作用。

newConn_cfg: SecReq_IMM_SEND表示新设备slave一经连接便会主动向master发Security Request，以触发master开始配对流程。如下图，slave主动发送SM_Security_Req 触发master发送配对请求：

M->S	OK	Control	3	0	0	0	9	Feature_Req(0x08) 00 00 00 00 00 00 00 E1 0x000021 -54 OK									
Direction	ACK Status	Data Type	Data Header					L2CAP Header		SM_Security_Req		CRC	RSSI (dBm)	FCS			
			LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AuthReq						
S->M	OK	L2CAP-S	2	1	0	0	6	0x0002	0x0006	0x0B	01	0x000041	-54	OK			
Direction	ACK Status	Data Type	Data Header					L2CAP Header		SM_Pairing_Req		CRC	RSSI (dBm)	FCS			
			LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	IOCap	OOBDataFlag	AuthReq	MaxEncKeySize	InitKeyDist	RespKeyDist	
M->S	OK	L2CAP-S	2	1	1	0	11	0x0007	0x0006	0x01	0x04	0x00	0x0D	0x10	0x0F	0x0F	
Direction	ACK Status	Data Type	Data Header					LL Opcode		LL_Feature_Rsp		CRC	RSSI (dBm)	FCS			

图3-66 抓包显示Pairing Conn Trigger

reConn_cfg: SecReq_NOT_SEND表示回连的时候slave不会发送Security Request。

此外SDK还提供了一个单独发送Security Request包的API，用于特殊应用场合，应用层可以随时调用该API发送Security Request包：

```
int blc_smp_sendSecurityRequest (void);
```

这里需要注意的时，用户如果使用blc_smp_configSecurityRequestSending管控安全配对请求包的话，就不要再使用调用blc_smp_sendSecurityRequest函数。

3.3.4.4 SMP 绑定信息说明

这里讨论的 SMP 绑定信息是对 slave 设备来说的。用户可以参考 SDK demo "8258_ble_remote"初始化中设置 direct advertising 的代码。

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（第 1 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。

1) 绑定信息存储顺序

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。

SDK 提供了两种 index 更新顺序方式：1、根据设备最近连接时间为顺序；2、根据设备配对时间先后为顺序。可以通过如下 API 设置 index 更新方式：

```
void bls_smp_setIndexUpdateMethod(index_updateMethod_t method);
```

枚举类型 index_updateMethod_t 具体定义介绍如下：

```
typedef enum {
    Index_Update_by_Pairing_Order = 0,    //default value
    Index_Update_by_Connect_Order = 1,
} index_updateMethod_t;
```

注意：老版本 SDK 只有第 2 种方式（无法修改），新版 SDK 默认为第 1 种（可以不调用这个 API），如果需要配置成第 2 种方式，调用该 API 即可。

下面分别介绍两种 index 更新方式:

A. 根据设备最近连接时间为顺序 (Index_Update_by_Connect_Order)

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。

B. 根据设备配对时间先后为顺序 (Index_Update_by_Pairing_Order)

如果 BondingDeviceNumber 为 2, 两个设备的 index 分别为 0 和 1。Index 顺序是**以配对时间先后为准**: 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, 接着让 slave 和 masterA 回连一次成功, 此时 index 0 设备依然为 masterA, index1 设备为 masterB。

如果 BondingDeviceNumber 为 4, 四个设备的 index 分别为 0、1、2、3, 0 是最老配对的设备, 3 是最新配对的设备。根据上面的描述, 若 slave 连续配对 masterA、B、C、D, 此时 masterD 是 index3 设备, 期间不管 slave 和 master A、B、C、D 什么顺序回连, index 0、1、2、3 依然分别对应 masterA、B、C、D。

需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; 如果在配对 masterA、B、C、D 后, 先和 masterA 回连一次, 此时顺序依旧 A、B、C、D, 再配对 masterE 的话 slave 会删除 masterA 的配对信息。

2) 绑定信息格式及相关 API 说明

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_addrType; //peer identity address information in key distri
    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_addrType/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 8258_feature_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 bls_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：

```
bls_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 bls_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。

下面 API 用于 slave 设备清除本地 FLASH 上存储的所有配对绑定信息：

```
void bls_smp_eraseAllParingInformation(void);
```

需要注意的是，用户调用该 API 前需要确保设备处于非连接态。

下面 API 可以用于 slave 设备配置绑定信息存储在 FLASH 中的位置：

```
void bls_smp_configParingSecurityInfoStorageAddr (int addr);
```

其中参数 addr 可以根据实际需要修改，用户配置前可以参考文档中“SDK FLASH 空间的分配”章节，决定绑定信息放置在 FLASH 中合适区域。

3.3.4.5 master SMP

master 的 SMP 功能目前支持传统配对方式下的最高级别是 LE security mode1 level2（传统配对 Just Works 方式）。用户可以参考“8258 master kma dongle”，只需要修改“8258 master kma dongle/app_config.h”文件下的宏：

```
#define BLE_HOST_SMP_ENABLE 0
```

该宏配置为 1，则使用标准的 SMP：配置 master 支持的最高安全级别为 LE security mode1 level2，支持传统配对 Just Works 方式；若该宏配置为 0，则表示启用非标准的自定义配对管理功能。

1) master 使能 SMP（设置宏 BLE_HOST_SMP_ENABLE 为 1）

使用该安全级别配置就必须调用如下 API 用于初始化 SMP 各参数配置，包括绑定区域 FLASH 的初始化配置：

```
int blc_smp_central_init (void);
```

如果在初始化阶段只调用了该 API，则 SDK 会使用默认参数去配置 SMP：

- ✧ 默认支持的最高安全等级：Unauthenticated_Paring_with_Encryption；
- ✧ 默认绑定模式：Bondable_Mode（存储配对加密后分发的 KEY 到 FLASH）；
- ✧ 默认 IO 能力是 IO_CAPABILITY_NO_INPUT_NO_OUTPUT。

在配对设备支持 LE security mode1 level2 时，还需要用户配置如下三个 API：

```
void blm_smp_configParingSecurityInfoStorageAddr (int addr);
```

```
void blm_smp_registerSmpFinishCb (smp_finish_callback_t cb);
```

```
void blm_host_smp_setSecurityTrigger(u8 trigger);
```

下面分别介绍这三个 API:

A. blm_smp_configParingSecurityInfoStorageAddr

该 API 可以用于 master 设备配置绑定信息存储在 FLASH 中的位置，其中参数 addr 可以根据实际需要修改。

B. blm_smp_registerSmpFinishCb

该回调函数在配对第三阶段密钥分发完成后触发，用户可以在应用层注册以获取配对完成事件。

C. blm_host_smp_setSecurityTrigger

该 API 主要用于配置 master 是否主动发起加密、回连时主动加密链路。具体参数可以选择如下：

```
#define SLAVE_TRIGGER_SMP_FIRST_PAIRING 0
#define MASTER_TRIGGER_SMP_FIRST_PAIRING BIT(0)
#define SLAVE_TRIGGER_SMP_AUTO_CONNECT 0
#define MASTER_TRIGGER_SMP_AUTO_CONNECT BIT(1)
```

具体说就是：1、第一次配对时，是 master 选择 master 主动发起配对请求还是在收到 slave 发送的 Security Request 后再开始配对；2、已经配对过的设备回连时，是 master 主动发送 LL_ENC_REQ 加密链路还是等收到 slave 发送的 Security Request 后再开始加密链路。一般我们会配置 Master 第一次配对主动发起配对请求，回连时主动发 LL_ENC_REQ。

最终的用户初始化代码参考如下，用户可以参考“8258 master kma dongle”：

```
blm_smp_configParingSecurityInfoStorageAddr(0x78000);
blm_smp_registerSmpFinishCb(app_host_smp_finish);
blc_smp_central_init();
//SMP trigger by master
blm_host_smp_setSecurityTrigger(MASTER_TRIGGER_SMP_FIRST_PAIRING |
                                MASTER_TRIGGER_SMP_AUTO_CONNECT);
```

至于以下几个 master 端的绑定信息相关的 API，是供 master SMP 协议底层使用的，用户不需要具体了解。

```
int    tbl_bond_slave_search(u8 adr_type, u8 * addr);
int    tbl_bond_slave_delete_by_addr(u8 adr_type, u8 *addr);
void   tbl_bond_slave_unpair_proc(u8 adr_type, u8 *addr);
```

2) 非标准的自定义配对管理（设置宏 BLE_HOST_SMP_ENABLE 为 0）

在 8258 master kma dongle 中，若 Disable SMP，则无法由 SDK 自动完成配对解配对等操作，需要在应用层添加配对管理。

如果用户需要使用自定义的配对管理，初始化相关 API 如下：

```
btc_smp_setSecurityLevel(No_Security); //禁用 SMP 功能
user_master_host_pairing_flash_init(); //自定义方式
```

A. 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_PAIRING 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 的 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，因此在初始化的时候加了特别处理：从 0x78000 开始读取 8 bytes area 信息，将所有有效 mac address 读到 ram 中的 slave mac table。这过程中检查 8 bytes area 是否太多，如果太多的话，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。

B. 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 pairing unless 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
    macAddr_t bond_device[USER_PAIR_SLAVE_MAX_NUM]; //macAddr_t already defined
    u8 curNum;
} user_slaveMac_t;

user_slaveMac_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，Telink 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。

C. 相关 API 说明

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。

a. user_master_host_pairing_flash_init

```
void user_master_host_pairing_flash_init(void);
```

用户自定义配对管理 flash 初始化函数，启用自定义方式时需要调用该初始化函数。

b. 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，Telink demo 的方法是直接覆盖最老的，由于 telink 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。

c. 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。

d. 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。

e. 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。

f. user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

删除所有配对设备。

g. 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。

D. 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：

- a. 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对，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 }
```

- b. 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 Telink 定义的配对广播包数据。代码：

```
//user design manual paring methods
user_manual_paring = dongle_paring_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);
}
```

E. 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 telink
                #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。

3.3.5 GAP

3.3.5.1 GAP 初始化

GAP 初始化分主从角色，slave 使用如下 API 初始化 GAP：

```
void          blc_gap_peripheral_init(void);
```

Master 使用如下 API 初始化 GAP：

```
void          blc_gap_central_init(void);
```

由前文我们知道，应用层与 Host 的数据交互不通过 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。目前 SDK 的 GAP 层主要处理 host 层上的事件，GAP 初始化主要是注册 host 层事件处理函数入口。

3.3.5.2 GAP event

GAP event 则是 ATT、GATT、SMP、GAP 等 host 协议层交互过程中产生的事件。从前文我们可以知道，目前 SDK 事件主要分为两大类：Controller event 和 GAP(host) event，其中 controller event 又分为 HCI event 和 Telink defined event。

Telink BLE SDK 中新增了 GAP event 处理，主要是协议栈事件分层更加清晰，协议栈处理用户层交互事件更加便捷，特别是 SMP 相关的处理，如 Passkey 的输入，配对结果通知用户等。

如果 user 需要在 App 层接收 GAP event，首先需要注册 GAP event 的 callback 函数，其次需要将对应 event 的 mask 打开。

GAP event 的 callback 函数原型和注册接口分别为：

```
typedef int (*gap_event_handler_t) (u32 h, u8 *para, int n);  
void blc_gap_registerHostEventHandler (gap_event_handler_t handler);
```

callback 函数原型中的 u32 h 是 GAP event 标记，底层协议栈多处会用到，下面列出几个用户可能会用到的事件：

```
#define GAP_EVT_SMP_PAIRING_BEAGIN 0  
#define GAP_EVT_SMP_PAIRING_SUCCESS 1  
#define GAP_EVT_SMP_PAIRING_FAIL 2  
#define GAP_EVT_SMP_CONN_ENCRYPTION_DONE 3  
#define GAP_EVT_SMP_TK_DISPALY 4  
#define GAP_EVT_SMP_TK_REQUEST_PASSKEY 5  
#define GAP_EVT_SMP_TK_REQUEST_OOB 6  
#define GAP_EVT_SMP_TK_NUMERIC_COMPARE 7  
#define GAP_EVT_ATT_EXCHANGE_MTU 16  
#define GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM 17
```

callback 函数原型中 para 和 n 表示 event 的数据和数据长度，下文将详细说明以上列出的 GAP event。User 可参考 8258 featuretest/feature_security.c 中如下用法以及 app_host_event_callback 函数的具体实现。

```
blc_gap_registerHostEventHandler ( app_host_event_callback );
```

GAP event 需要通过下面的 API 来打开 mask。

```
void blc_gap_setEventMask (u32 evtMask);
```

eventMask的定义也对应上面给出一些，其他的event mask用户可以在ble/gap/gap_event.h中查到。

```
#define GAP_EVT_MASK_SMP_PARING_BEAGIN
    (1<<GAP_EVT_SMP_PARING_BEAGIN)
#define GAP_EVT_MASK_SMP_PARING_SUCCESS
    (1<<GAP_EVT_SMP_PARING_SUCCESS)
#define GAP_EVT_MASK_SMP_PARING_FAIL
    (1<<GAP_EVT_SMP_PARING_FAIL)
#define GAP_EVT_MASK_SMP_CONN_ENCRYPTION_DONE
    (1<<GAP_EVT_SMP_CONN_ENCRYPTION_DONE)
#define GAP_EVT_MASK_SMP_TK_DISPALY
    (1<<GAP_EVT_SMP_TK_DISPALY)
#define GAP_EVT_MASK_SMP_TK_REQUEST_PASSKEY
    (1<<GAP_EVT_SMP_TK_REQUEST_PASSKEY)
#define GAP_EVT_MASK_SMP_TK_REQUEST_OOB
    (1<<GAP_EVT_SMP_TK_REQUEST_OOB)
#define GAP_EVT_MASK_SMP_TK_NUMERIC_COMPARE
    (1<<GAP_EVT_SMP_TK_NUMERIC_COMPARE)
#define GAP_EVT_MASK_ATT_EXCHANGE_MTU
    (1<<GAP_EVT_ATT_EXCHANGE_MTU)
#define GAP_EVT_MASK_GATT_HANDLE_VLAUE_CONFIRM
    (1<<GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM)
```

若 user 未通过该 API 设置 GAP event mask，那么当 GAP 相应的 event 产生时将不会通知应用层。

注意：以下论述 GAP event 时，均设定注册了 GAP event 回调，且开启了对应的 eventMask。

1) GAP_EVT_SMP_PARING_BEAGIN

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。

Data Type	Data Header					L2CAP Header		SM_Security_Req											
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AuthReq										
	2	1	0	0	6	0x0002	0x0006	0x0B	01										

Data Type	Data Header					L2CAP Header		SM_Pairing_Req												
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	IOCap	OOBDataFlag	AuthReq	MaxEncKeySize	InitKeyDist	RespKeyDist						
	2	1	1	0	11	0x0007	0x0006	0x01	0x03	0x00	0x01	0x10	0x02	0x03						

图 3-67 master 发起 Pairing_Req

数据长度 n: 4。

回传指针 p: 指向一片内存数据，对应如下结构体：

```
typedef struct {  
    u16 connHandle;  
    u8 secure_conn;  
    u8 tk_method;  
} gap_smp_paringBeginEvt_t;
```

connHandle 表示当前连接句柄。

secure_conn 为 1 表示使用安全加密特性（LE Secure Connections），否则将使用 LE legacy pairing。

tk_method 表示接下来配对使用什么样的 TK 值方式：例如 JustWorks、PK_Init_Dsply_Resp_Input、PK_Resp_Dsply_Init_Input、Numric_Comparison 等。

2) GAP_EVT_SMP_PARING_SUCCESS

事件触发条件：配对整个流程正确完成时产生该事件，该阶段即为 LE 配对阶段之密钥分发阶段 3（Key Distribution, Phase 3），如果有密钥需要分发，则等待双方密钥分发完成后触发配对成功事件，否则直接触发配对成功事件。

数据长度 n: 4。

回传指针 p: 指向一片内存数据，对应如下结构体：

```
typedef struct {  
    u16 connHandle;  
    u8 bonding;  
    u8 bonding_result;  
} gap_smp_paringSuccessEvt_t;
```

connHandle 表示当前连接句柄。

bonding 为 1 表示启用 bonding 功能，否则不启用。

bonding_result 表示 bonding 的结果：如果没有开启 bonding 功能，则为 0，如果开启了 bonding 功能，则还需要检查加密 Key 是否被正确的存储在 FLASH 中，存储成功为 1，否则为 0。

3) GAP_EVT_SMP_PARING_FAIL

事件触发条件：由于 slave 或 master 其中一个不符合标准配对流程，或者通信中出现报错等异常原因导致配对流程终止。

数据长度 n: 2。

回传指针 p: 指向一片内存数据，对应如下结构体：

```
typedef struct {
    u16 connHandle;
    u8 reason;
} gap_smp_paringFailEvt_t;
```

connHandle 表示当前连接句柄。

reason 表示配对失败的原因，这里列出几个常见的配对失败原因值，其他配对失败原因值我们可以参考 SDK 目录下的“stack/ble/smp/smp_const.h”文件。

配对失败值具体含义可以参照《Core_v5.0》(Vol 3/Part H/3.5.5 “Pairing Failed”)。

```
#define PARING_FAIL_REASON_CONFIRM_FAILED          0x04
#define PARING_FAIL_REASON_PARING_NOT_SUPPORTED    0x05
#define PARING_FAIL_REASON_DHKEY_CHECK_FAIL        0x0B
#define PARING_FAIL_REASON_NUMUERIC_FAILED         0x0C
#define PARING_FAIL_REASON_PARING_TIEMOUT          0x80
#define PARING_FAIL_REASON_CONN_DISCONNECT         0x81
```

4) GAP_EVT_SMP_CONN_ENCRYPTION_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。

数据长度 n：3。

回传指针 p：指向一片内存数据，对应如下结构体：

```
typedef struct {
    u16 connHandle;
    u8 re_connect;    //1: re_connect, encrypt with previous
                     distributed LTK; 0: pairing , encrypt with STK
} gap_smp_connEncDoneEvt_t;
```

connHandle 表示当前连接句柄。

re_connect 为 1 表示快速回连（将使用之前分发的 LTK 加密链路），若该值为 0 则表示当前加密是第一次加密。

5) GAP_EVT_SMP_TK_DISPALY

事件触发条件：slave 收到 master 发送的 Pairing_Req 后，根据对端设备的配对参数和本地设备的配对参数配置，我们就可以知道接下来配对使用什么样的 TK（pincode）值方式。如果启用的是 PK_Resp_Dsply_Init_Input（即：slave 端显示 6 位 pincode 码，master 端负责输入 6 位 pincode 码）方式，则会立即触发。

数据长度 n：4。

回传指针 p：指向一个 u32 型变量 tk_set，该值即为 slave 需要通知应用层的 6 位 pincode 码，应用层需要显示该 6 位码值。

用户也可以不使用底层随机生成的 6 位 pincode 码，可以手动设置一个用户指定的 pincode 码例如“123456”。

```
case GAP_EVT_SMP_TK_DISPALY:
{
    char pc[7];
    #if 1 //手动设置pincode码
    u32 pinCode = 123456;
    memset(smp_param_own.paring_tk, 0, 16);
    memcpy(smp_param_own.paring_tk, &pinCode, 4);

    #else//使用底层随机生成的pincode码
        u32 pinCode = *(u32*)para;
    #endif
}

break;
```

用户将 slave 上看到的 6 位 pincode 码输入到 master 设备上（如手机），完成 TK 输入，配对流程得以继续执行。如果用户输入 pincode 错误或者点击取消，则配对流程失败。

关于 Passkey Entry 应用的实例，用户可以参考 SDK 提供的 demo “sdk/8258_feature_test/feature_security.c”。

6) GAP_EVT_SMP_TK_REQUEST_PASSKEY

事件触发条件：当 slave 设备启用 Passkey Entry 方式时，且使用的 PK_Init_Dsply_Resp_Input 或者 PK_BOTH_INPUT 配对方式时，会触发该事件，通知用户需要输入 TK 值。用户在收到该事件后就需要通过 IO 输入能力输入 TK 值（超时 30s 如果还未输入则配对失败），输入 TK 值的 API: blc_smp_setTK_by_PasskeyEntry 在“SMP 参数配置”章节有说明。

数据长度 n: 0。

回传指针 p: NULL。

7) GAP_EVT_SMP_TK_REQUEST_OOB

事件触发条件：当 slave 设备启用传统配对 OOB 方式时，会触发该事件，通知用户需要通过 OOB 方式输入 16 位 TK 值。用户在收到该事件后就需要通过 IO 输入能力输入 16 位 TK 值（超时 30s 如果还未输入则配对失败），输入 TK 值的 API: blc_smp_setTK_by_OOB 在“SMP 参数配置”章节有说明。

数据长度 n: 0。

回传指针 p: NULL。

8) GAP_EVT_SMP_TK_NUMERIC_COMPARE

事件触发条件: slave 收到 master 发送的 Pairing_Req 后, 根据对端设备的配对参数和本地设备的配对参数配置我们就可以知道接下来配对使用什么样的 TK (pincode) 值方式, 如果启用的是 Numeric_Comparison 方式, 则会立即触发。

(Numeric_Comparison 方式即数值比较, 属于 smp4.2 安全加密, master 和 slave 设备均会弹出显示 6 位 pincode 码以及 “YES” 和 “NO” 对话框, 用户需要检查两端设备显示的 pincode 是否一致, 并需要两端分别确认是否点击 “YES” 以确认 TK 校验是否通过)。

数据长度 n: 4。

回传指针 p: 指向一个 u32 型变量 pinCode, 该值即为 slave 需要通知应用层的 6 位 pincode 码, 应用层需要显示该 6 位码值, 并提供 “YES” 和 “NO” 的确认机制。

关于数值比较应用的实例, 用户可以参考 SDK 提供的 demo “sdk/8258_feature_test/feature_security.c”。

9) GAP_EVT_ATT_EXCHANGE_MTU

事件触发条件: 无论是 master 端发送 Exchange MTU Request, slave 回复 Exchange MTU Response, 还是 slave 端发送 Exchange MTU Request, master 回复 Exchange MTU Response, 两种情况下均会触发。

数据长度 n: 6。

回传指针 p: 指向一片内存数据, 对应如下结构体:

```
typedef struct {  
    u16 connHandle;  
    u16 peer_MTU;  
    u16 effective_MTU;  
} gap_gatt_mtuSizeExchangeEvt_t;
```

connHandle 表示当前连接句柄。

peer_MTU 表示对端的 RX MTU 值。

effective_MTU = min(CleintRxMTU, ServerRxMTU), CleintRxMTU 表示客户端的 RX MTU size 值, ServerRxMTU 表示服务端的 RX MTU size 值。Master 和 slave 交互了彼此的 MTU size 后, 取两者最小值作为彼此交互的最大 MTU size 值。

10) GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM

事件触发条件：应用层每调用一次 `bls_att_pushIndicateData`（或者调用 `blc_gatt_pushHandleValueIndicate`），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，slave 收到该 confirm 时触发。

数据长度 `n`：0。

回传指针 `p`：NULL。

Shared under NDA

4 低功耗管理（PM）

低功耗管理（Low Power Management）也可以称为功耗管理（Power Management），本文档中会简称为 PM。

4.1 低功耗 driver

4.1.1 低功耗模式

8x5x MCU 正常执行程序时处于 working mode，此时工作电流在 3~7mA 之间。如果需要省功耗需要进入低功耗模式。

低功耗模式（low power mode）又称 sleep mode，包括 3 种：suspend mode、deepsleep mode 和 deepsleep retention mode。

sleep mode module		suspend	deepsleep retention	deepsleep
Sram		100% keep	first 16K(or 32K) keep, others lost	100% lost
register	digital register	99% keep	100% lost	100% lost
	analog register	100% keep	99% lost	99% lost

上表为 3 种 sleep mode 下 Sram、数字寄存器（digital register）、模拟寄存器（analog register）状态保存的统计说明。

1) suspend mode

sleep mode 1

此时程序停止运行，类似一个暂停功能。MCU 大部分硬件模块断电，PM 模块维持正常工作。此时 IC 电流在 60~70 uA 之间。当 suspend 被唤醒后，程序继续执行。

suspend mode 下所有的 Sram 和 analog register 都能保存状态，绝大部分 digital register 都保持状态。digital register 中存在少量会掉电的，包括：

- baseband 电路中少量的 digital register，user 需要关注的是 API rf_set_power_level_index 设置的寄存器，本文档前面已经介绍，这个 API 需要在每次 suspend 醒来后都重新调用一次。

- b) 控制 Dfifo 状态的 digital register。对应 drivers/8258/dfifo.h 中的相关 API，user 在使用这些 API 的时候，必须确保每次 suspend wake_up 后都要重新设置。

2) deepsleep mode

sleep mode 2

此时程序停止运行，MCU 绝大部分的硬件模块都断电，PM 硬件模块维持工作。在 deepsleep mode 下 IC 电流小于 1uA。如果内置 flash 的 standby 电流出现较大的 1uA 左右，可能导致测量到 deepsleep 为 1~2uA。deepsleep mode wake_up 时，MCU 将重新启动，类似于上电的效果，程序会重新开始进行初始化。

Deepsleep mode 下，除了 analog register 上有少数几个 register 能保存状态，其他所有 Sram、digital register、analog register 全部掉电丢失。

3) deepsleep retention mode

sleep mode 3

上面的 deepsleep mode，电流很低，但是无法存储 Sram 信息；suspend mode Sram 和 register 可以保持不丢，但是电流偏高。

为了实现一些需要 sleep 时电流很低又要能够确保 sleep 唤醒后能立刻恢复状态的应用场景（比如 BLE 长睡眠维持连接），8x5x 增加了一种 sleep mode 3: deepsleep with Sram retention mode，简称 deepsleep retention（或 deep retention）。根据 Sram retention area 的大小不同，又分为 deepsleep retention 16K Sram 和 deepsleep retention 32K Sram。

Deepsleep retention mode 也是一种 deepsleep，MCU 绝大部分的硬件模块都断电，PM 硬件模块维持工作。功耗是在 deepsleep mode 基础上增加 retention Sram 消耗的电，电流在 2~3uA 之间。deepsleep mode wake_up 时，MCU 将重新启动，程序会重新开始进行初始化。

Telink 上一代 826x 系列 IC，使用 suspend（电流在 10uA 以内）来实现 BLE 长睡眠维持连接。而 8x5x 由于 suspend 电流偏大，需要使用 deepsleep retention mode 来代替实现。

deepsleep retention mode 和 deepsleep mode 在 register 状态保存方面表现一致，几乎全部掉电。deepsleep retention mode 跟 deepsleep mode 相比，Sram 的前 16K（或前 32K）可以保持不掉电，剩余的 Sram 全部掉电。

deepsleep mode 和 deepsleep retention mode 中在 analog register 部分都有极少数可以保持不掉电，这些不掉电的 analog register 包括：

1) 控制 GPIO 模拟上下拉电阻的 analog register

通过 API `gpio_setup_up_down_resistor` 设置的 GPIO 模拟上下拉电阻，或者在 `app_config.h` 中使用类似如下方式配置的 GPIO 模拟上下拉电阻，可以保持状态。

```
#define PULL_WAKEUP_SRC_PD5 PM_PIN_PULLDOWN_100K
```

参考文档第 2 章 GPIO 模块的介绍。使用 `gpio output` 属于 digital register 控制的状态。在 Telink 上一代 826x 系列 IC 上，suspend 期间可以用 `gpio output` 来控制一些外围设备，但到了 8x5x 上如果 suspend 被切换为 deepsleep retention mode 后，`gpio output` 状态失效，无法在 sleep 期间准确的控制外围设备。此时可以使用 GPIO 模拟上下拉电阻的状态来代替实现：上拉 10K 代替 `gpio output high`，下拉 100K 代替 `gpio output low`。

2) PM 模块特殊的不掉电 analog register

`Drivers/8258/pm.h` 文件中的 `DEEP_ANA_REG`，如下 code 所示：

```
#define DEEP_ANA_REG0 0x3a
#define DEEP_ANA_REG1 0x3b
#define DEEP_ANA_REG2 0x3c //system used, user can not use
```

需要注意的是，客户不允许使用 `ana_3c`，该模拟寄存器留给底层 stack 使用，如果应用层代码有用到该寄存器，需要修改为 `ana_3a`、`ana_3b`。因为不掉电模拟寄存器数量比较少，建议客户使用其每一个 bit 指示不同的状态位信息，具体可以参考 SDK 的 vendor 目录下的“8258_ble_remote”。

如下几组不掉电模拟寄存器可能会因为错误的 GPIO 唤醒而丢掉信息，比如 `GPIO_PAD` 高电平唤醒 deepsleep，但是在调用 `cpu_sleep_wakeup` 函数前 `gpio` 已经为高电平，这就会导致错误的 GPIO 唤醒，那么这些模拟寄存器值将会丢失。

```
#define DEEP_ANA_REG6 0x35
#define DEEP_ANA_REG7 0x36
#define DEEP_ANA_REG8 0x37
#define DEEP_ANA_REG9 0x38
#define DEEP_ANA_REG10 0x39
```

User 可以在这几个 analog register 中保存一些重要的信息，deepsleep/deepsleep retention wake_up 后还可以读到之前存储的值。

4.1.2 低功耗唤醒源

8x5x MCU 的低功耗唤醒源示意图如下，suspend/deepsleep/deepsleep retention 都可以被 GPIO PAD 和 timer 唤醒。该 BLE SDK 中，只关注 2 种唤醒源，如下所示（注意 code 中 PM_TIM_RECOVER_START 和 PM_TIM_RECOVER_END 两个定义不是唤醒源）：

```
typedef enum {
    PM_WAKEUP_PAD = BIT(4),
    PM_WAKEUP_TIMER = BIT(6),
} SleepWakeupSrc_TypeDef;
```

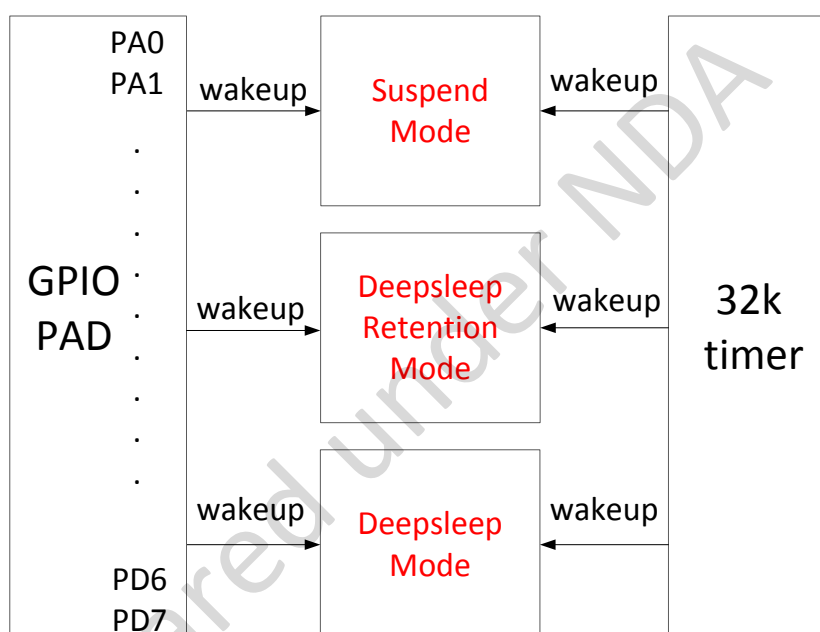


图 4-1 8x5x MCU 硬件唤醒源

如上图所示，MCU 的 suspend/deepsleep/deepsleep retention 在硬件上有 2 个唤醒源：TIMER、GPIO PAD。

- ✧ 唤醒源 PM_WAKEUP_TIMER 来自硬件 32k timer（32k RC timer or 32k Crystal timer）。32k 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_PAD 来自 GPIO 模块，除 MSPI 4 个管脚外所有的 GPIO（PAx/PBx/PCx/PDx）的高低电平都具有唤醒功能。

配置 GPIO PAD 唤醒 sleep mode 的 API:

```
typedef enum{
    Level_Low = 0,
    Level_High,
}GPIO_LevelTypeDef;

void cpu_set_gpio_wakeup (GPIO_PinTypeDef pin,
                           GPIO_LevelTypeDef pol, int en);
```

pin 为 GPIO 定义。

pol 为唤醒极性定义: Level_High 表示高电平唤醒, Level_Low 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, Level_High, 1); //GPIO_PC2 PAD 唤醒打开, 高电平唤醒
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 0); //GPIO_PC2 PAD 唤醒关闭
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 1); //GPIO_PB5 PAD 唤醒打开, 低电平唤醒
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 0); //GPIO_PB5 PAD 唤醒关闭
```

4.1.3 低功耗模式的进入和唤醒

设置 MCU 进入睡眠和唤醒的 API 为:

```
int cpu_sleep_wakeup (SleepMode_TypeDef sleep_mode,
                      SleepWakeupSrc_TypeDef wakeup_src,
                      unsigned int wakeup_tick);
```

第一个参数 sleep_mode: 设置 sleep mode, 有以下 4 个选择, 分别表示 suspend mode、deepsleep mode、deepsleep retention 16K Sram、deepsleep retention 32K Sram。

```
typedef enum {
    SUSPEND_MODE = 0,
    DEEPSLEEP_MODE = 0x80,
    DEEPSLEEP_MODE_RET_SRAM_LOW16K = 0x43,
    DEEPSLEEP_MODE_RET_SRAM_LOW32K = 0x07,
}SleepMode_TypeDef;
```

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_TIMER` 中的一个或者多个。如果 `wakeup_src` 为 0, 那么进入低功耗 `sleep mode` 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, 按照本文档前面介绍的 System Timer tick 来设置, 当 System Timer tick 的值达到这个设定的 `wakeup_tick` 后, `sleep mode` 被唤醒。`wakeup_tick` 的值需要根据当前的 System Timer tick 的值, 加上由需要睡眠的时间换算成的绝对时间, 才可以有效地控制睡眠时间。如果没有考虑当前的 System Timer tick, 直接对 `wakeup_tick` 进行设置, 唤醒的时间点就无法控制。

由于 `wakeup_tick` 是绝对时间, 必须在 32bit 的 System Timer tick 能表示的范围之内, 所以这个 API 能表示的最大睡眠时间是有限的。目前的设计是最大睡眠时间为 32bit 能表示的最大 System Timer tick 对应时间的 7/8。System Timer tick 最大能表示大概 268S, 那么最长 `sleep` 时间时间为 $268 \times 7/8 = 234$ S, 即下面 `delta_Tick` 不能超过 234 S。

```
cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER,
                 clock_time() + delta_tick);
```

返回值为当前 `sleep mode` 的唤醒源的集合, 该返回值各 bit 对应表示的唤醒源为:

```
enum {
    WAKEUP_STATUS_TIMER = BIT(1),
    WAKEUP_STATUS_PAD   = BIT(3),

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

- 1) `WAKEUP_STATUS_TIMER` 这个 bit 为 1, 说明当前 `sleep mode` 是被 Timer 唤醒。
- 2) `WAKEUP_STATUS_PAD` 这个 bit 为 1, 说明当前 `sleep mode` 是被 GPIO PAD 唤醒。
- 3) `WAKEUP_STATUS_TIMER` 和 `WAKEUP_STATUS_PAD` 同时为 1 时, 表示 Timer 和 GPIO PAD 两个唤醒源同时生效了。
- 4) `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 PAD 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_PAD` 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`。

一般采用如下的形式来控制睡眠时间：

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_TIMER,  
                  clock_time() + delta_Tick);
```

delta_Tick 是一个相对的时间（比如 $100 * \text{CLOCK_16M_SYS_TIMER_CLK_1MS}$ ），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 (SUSPEND_MODE , PM_WAKEUP_PAD, 0);`

程序执行该函数时进入 suspend mode，只能被 GPIO PAD 唤醒。

- 2) `cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_TIMER, clock_time() + 10*
CLOCK_16M_SYS_TIMER_CLK_1MS;`

程序执行该函数时进入 suspend mode，只能被 Timer 唤醒，唤醒时间为当前时间加上 10 ms，所以 suspend 时间为 10 ms。

- 3) `cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_PAD | PM_WAKEUP_TIMER,
clock_time() + 50* CLOCK_16M_SYS_TIMER_CLK_1MS);`

程序执行该函数时进入 suspend 模式，可被 GPIO PAD 和 Timer 唤醒，Timer 唤醒的时间设置为 50ms。如果在 50ms 结束之前触发了 GPIO 的唤醒动作，MCU 会被 GPIO PAD 唤醒；如果 50ms 内无 GPIO 动作，MCU 会被 Timer 唤醒。

- 4) `cpu_sleep_wakeup (DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);`

程序执行该函数时进入 deepsleep mode，可被 GPIO PAD 唤醒。

- 5) `cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K , PM_WAKEUP_TIMER,
clock_time() + 8* CLOCK_16M_SYS_TIMER_CLK_1S);`

程序执行该函数时进入 deepsleep retention 16K Sram mode，可被 Timer 唤醒，唤醒时间为执行该函数的 8 s 后。

- 6) `cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K , PM_WAKEUP_PAD |
PM_WAKEUP_TIMER, clock_time() + 10* CLOCK_16M_SYS_TIMER_CLK_1S);`

程序执行该函数时进入 deepsleep retention 32K Sram mode，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 结束之前触发了 GPIO 动作，MCU 会被 GPIO PAD 唤醒；如果 10 S 内无 GPIO 动作，MCU 会被 Timer 唤醒。

4.1.4 低功耗唤醒后运行流程

当 user 调用 API `cpu_sleep_wakeup` 后，MCU 进入 sleep mode；当唤醒源触发 MCU 唤醒后，对于不同的 sleep mode，MCU 的软件运行流程不一致。

下面详细介绍 suspend、deepsleep、deepsleep retention 3 种 sleep mode 被唤醒后的 MCU 运行流程。请参考下图。

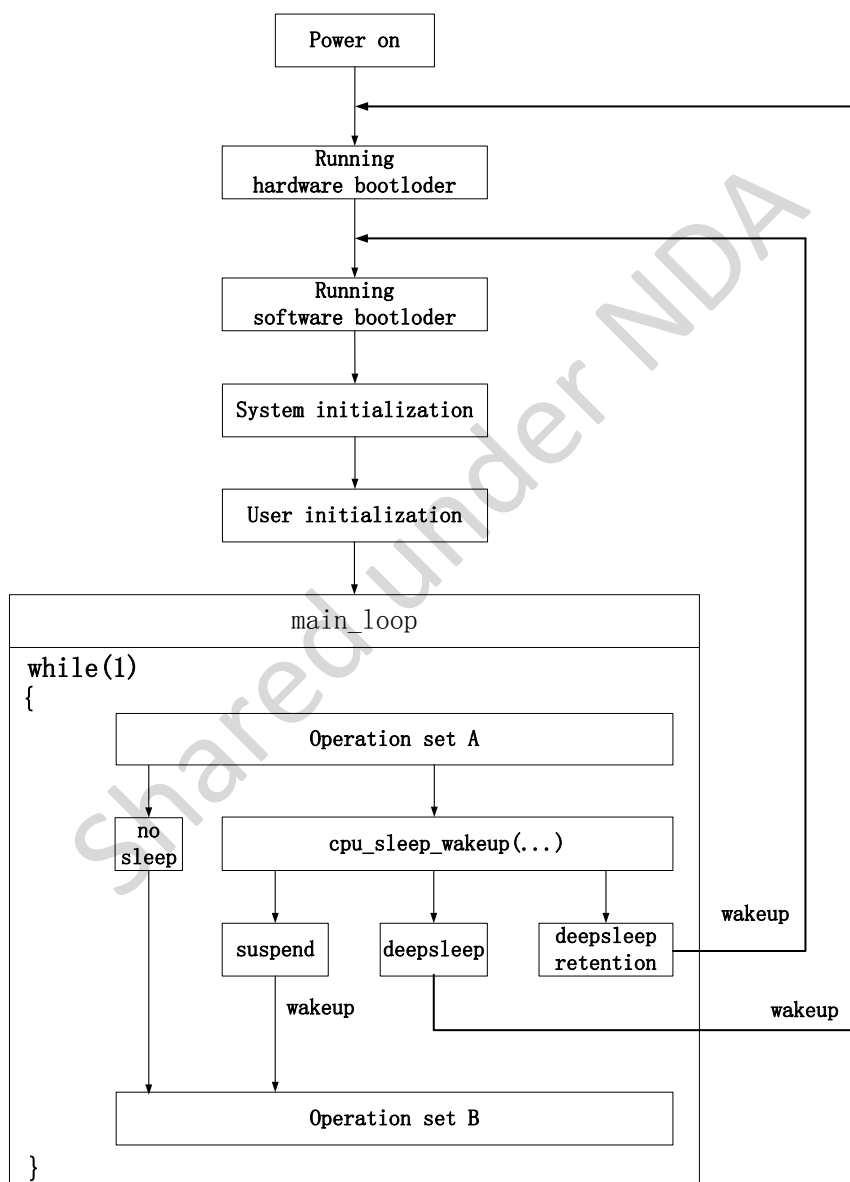


图 4-2 sleep mode wakeup work flow

MCU 上电（Power on）之后，各流程的介绍：

1) 运行硬件 bootloader（Run hardware bootloader）

MCU 硬件上执行一些固定的动作，这些动作固化在硬件上，软件无法修改。

举几个例子说明一下这些动作，比如：读 flash 的 boot 启动标记，判断当前应该运行的 firmware 是存储在 flash 地址 0 上的，还是在 flash 地址 0x20000 上的（跟 OTA 相关）；读 flash 相应位置的值，判断当前需要从 flash 上拷贝多少数据到 Sram，作为常驻内存的数据（参考第 2 章对 Sram 分配的介绍）。

运行硬件 bootloader 部分由于涉及到 flash 上数据拷贝到 sram，一般执行时间较长，比如拷贝 10K 数据大概耗时 5ms 左右。

2) 运行软件 bootloader（Run software bootloader）

hardware bootloader 运行结束之后，MCU 开始运行 software bootloader。

Software bootloader 就是前面介绍过的 vector 端（对应 cstartup_8258_16K_RET.S 里面的汇编程序）。

Software bootloader 是为了给后面 C 语言程序的运行设置好内存环境，可以理解为整个内存的初始化。

3) 系统初始化（System initialization）

System initialization 对应 main 函数中 cpu_wakeup_init 到 user_init 之前各硬件模块初始化（包括 cpu_wakeup_init、rf_drv_init、gpio_init、clock_init），设置各硬件模块的数字/模拟寄存器状态。

4) 用户初始化（User initialization）

User initialization 对应 SDK 中函数 user_init 或 user_init_normal/user_init_deepRetn。

5) main_loop

User initialization 完成后，进入 while(1)控制的 main_loop。main_loop 中进入 sleep mode 之前的一系列操作称为"Operation Set A"，sleep 唤醒之后一系列操作称为"Operation Set B"。

对照图示，sleep mode 流程分析：

1) no sleep

如果没有 sleep mode，MCU 的运行流程为在 while(1)中循环，反复执行 "Operation Set A" ->"Operation Set B"。

2) suspend

如果调用 `cpu_sleep_wakeup` 函数进入 suspend mode，当 suspend 被唤醒后，相当于 `cpu_sleep_wakeup` 函数的正常退出，MCU 运行到"Operation Set B"。

suspend 是最干净的 sleep mode，在 suspend 期间所有的 Sram 数据能保持不变，所有的数字/模拟寄存器状态也保持不变（只有几个特殊的例外）；suspend 唤醒后，程序接着原来的位置运行，几乎不需要考虑任何 sram 和寄存器状态的恢复。suspend 的缺点是功耗偏高。

3) deepsleep

如果调用 `cpu_sleep_wakeup` 函数进入 deepsleep mode，当 deepsleep 被唤醒后，MCU 会重新回到 Run hardware bootloader。

可以看出，deepsleep wake_up 跟 Power on 的流程是几乎一致的，所有的软硬件初始化都得重新做。

MCU 进入 deepsleep 后，所有的 Sram 和数字/模拟寄存器（只有几个模拟寄存器例外）都会掉电，所以功耗很低，MCU 电流小于 1uA。

4) deepsleep retention

如果调用 `cpu_sleep_wakeup` 函数进入 deepsleep retention mode，当 deepsleep retention 被唤醒后，MCU 会重新回到 Run software bootloader。

deepsleep retention 是介于 suspend 和 deepsleep 之间的一种 sleep mode。

suspend 因为要保存所有的 sram 和寄存器状态而导致电流偏高；deepsleep retention 不需要保存寄存器状态，Sram 只保留前 16K（或 32K）不掉电，所以功耗比 suspend 低很多，只有 2uA 左右。

deepsleep wake_up 后需要把所有的流程重新运行一遍，而 deepsleep retention 可以跳过"Run hardware bootloader"这一步，这是因为 Sram 的前 16K（32K）上数据是不丢的，不需要再从 flash 上重新拷贝一次。但由于 Sram 上 retention area 有限，"run software bootloader"无法跳过，必须得执行；由于 deepsleep retention 无法保存寄存器状态，所以 system initialization 必须执行，寄存器的初始化需要重新设置。deepsleep retention wake_up 后的 user initialization 可以做一些优化改进，和 MCU power on/deepsleep wake_up 后的 user initialization 做区分处理，参考文本档后面的介绍。

4.1.5 API pm_is_MCU_deepRetentionWakeup

由图"sleep mode wakeup work flow"可以看到，MCU power on、deepsleep wake_up、deepsleep retention wake_up 这 3 种情况都需要经过 Run software bootloader、System initialization、User initialization。

在运行 system initialization、user initialization 2 个步骤时，user 需要知道当前 MCU 是否被 deepsleep retention wake_up 的，以便做一些区分于 power on、deepsleep wake_up 的设置。PM driver 提供判断是否 deepsleep retention wake_up 的 API 为：

```
int pm_is_MCU_deepRetentionWakeup(void);
```

return 值为1，表示deepsleep retention wake_up; return 值为0，表示power on或deepsleep wake_up。

4.2 BLE 低功耗管理

4.2.1 BLE PM 初始化

如果使用了低功耗模式，需要将 BLE PM 模块初始化，调用下面 API 即可。

```
void blc_ll_initPowerManagement_module(void);
```

若不需要低功耗模式，不调用此 API，则 PM 相关的代码和变量都不会被编译到程序中，可以节省 firmware size 和 sram size。

4.2.2 BLE PM for Link Layer

该 BLE SDK 低功耗管理是对 BLE Link Layer 功耗的管理，请参考本文档前面对 Link Layer 的介绍。

SDK 中暂时只对 Advertising state 和 Connection state Slave role 做了低功耗管理，并且提供了一套 API 供 user 调用和配置。

对于 Scanning state、Initiating state 和 Connection state Master role，SDK 暂时不提供低功耗管理。

对于 Idle state，SDK 也不提供任何低功耗管理。由于此状态不涉及 BLE RF 任何动作（即 blt_sdk_main_loop 函数完全无效），user 可以自行调用 PM driver 去做一些低功耗管理。下面 code 为一个简单的 demo：当 Link Layer 处于 Idle state 时，每个 main_loop suspend 10ms。

```

void main_loop (void)
{
    ////////////////////////////////////////////////// BLE entry ///////////////////////////////////
    blt_sdk_main_loop();

    ////////////////////////////////////////////////// UI entry ///////////////////////////////////
    // add user task

    ////////////////////////////////////////////////// PM configuration ///////////////////////////////////
    if(blc_ll_getCurrentState() == BLS_LINK_STATE_IDLE ){ //Idle state
        cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER,
                        clock_time() + 10*CLOCK_16M_SYS_TIMER_CLK_1MS);
    }
    else{
        blt_pm_proc(); //BLE Adv & Conn state
    }
}

```

当 Link Layer 处于 Advertising state 或 Conn state Slave role 时，下图所示为 sleep mode 的时序。注意，图中 Conn state Slave role 为 connection latency = 0 的情况。

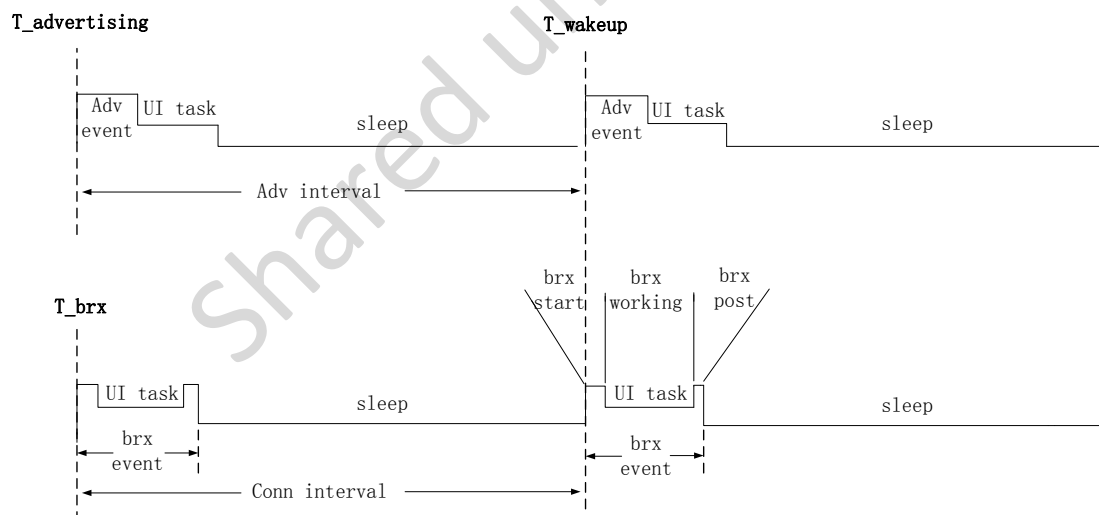


图 4-3 sleep timing for Advertising state & Conn state Slave role

- 1) 处于 Advertising state 时，每个 Adv Interval 里，Adv Event 时间是必须的，除去 UI task 所占用的时间，剩余时间 MCU 可以进入 sleep mode (suspend/deepsleep retention)。

图中，第一个 Adv interval 上 Adv event 开始的时间我们定义为 $T_{\text{advertising}}$ ；sleep 需要唤醒的时间我们定义为 T_{wakeup} ， T_{wakeup} 也是下一个 Adv interval 上 Adv event 的开始。 $T_{\text{advertising}}$ 和 T_{wakeup} 在本文档后面的介绍中需要使用到。

- 2) 处于 Conn state Slave role 时，每个 Conn interval 内，brx Event (brx start+brx working+brx post) 时间是必须的，除去 UI task 占用的时间，剩余时间 MCU 可以进入 sleep mode (suspend/deepsleep retention)。

图中，第一个 Connection interval 上 Brx event 开始的时间我们定义为 T_{brx} ；sleep 需要唤醒的时间我们定义为 T_{wakeup} ， T_{wakeup} 也是下一个 Connection interval 上 Brx event 的开始。 T_{brx} 和 T_{wakeup} 在本文档后面的介绍中需要使用到。

BLE 低功耗管理的实质是对上面两个状态的 sleep 时间进行管理，user 可以决定如何使用这些时间：不进入 sleep、进入 suspend mode 或进入 deepsleep retention mode。

8x5x 的 sleep mode 分 3 种：suspend、deepsleep、deepsleep retention。

对于 sleep mode 中的 suspend 和 deepsleep retention，user 不需要调用 API `cpu_sleep_wakeup` 来实现。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。

对于 deepsleep，BLE 低功耗管理不包括对它的处理，所以 user 需要手动在应用层调用 API `cpu_sleep_wakeup` 来进入 deepsleep。手动 deepsleep mode 的使用，可以参考 SDK project “8258_ble_remote” `blt_pm_proc` 函数中对 deepsleep 的处理。

下面开始对 Advertising state 和 Connection state Slave role 的低功耗管理做详细的介绍。

4.2.3 相关变量

BLE PM 软件处理流程部分会出现很多变量，用户有必要了解这些变量。

请在文件 ll_pm.h 找到结构体 “st_ll_pm_t” 的定义，下面只列出该结构体部分变量（API 介绍时需要用到的变量）。

```
typedef struct {  
    u8      suspend_mask;  
    u8      wakeup_src;  
    u16     sys_latency;  
    u16     user_latency;  
    u32     deepRet_advThresTick;  
    u32     deepRet_connThresTick;  
    u32     deepRet_earlyWakeupTick;  
}st_ll_pm_t;
```

在文件 ll_pm.c 中定义了如下结构体变量。

注意：该文件被封装在 library 中，这里给出定义只是为了方便后面的介绍，用户不允许对这个结构体变量进行任何操作。

```
st_ll_pm_t bltPm;
```

下面的介绍中会经常出现类似 “bltPm.suspend_mask” 的变量。

4.2.4 API bls_pm_setSuspendMask

用于配置 Link Layer Advertising state 和 Conn state Slave role 低功耗管理的 API:

```
void      bls_pm_setSuspendMask (u8 mask);  
u8        bls_pm_getSuspendMask (void);
```

使用 bls_pm_setSuspendMask 设置 bltPm.suspend_mask（默认值为 SUSPEND_DISABLE）。

这两个 API 的源码为:

```
void      bls_pm_setSuspendMask (u8 mask)  
{  
    bltPm.suspend_mask = mask;  
}  
  
u8        bls_pm_getSuspendMask (void)  
{  
    return bltPm.suspend_mask;
```

```
}

```

`bltPm.suspend_mask` 的设置，可以选择下面几个值中的一个，或者选择多个值的“或操作”。

```
#define SUSPEND_DISABLE 0
#define SUSPEND_ADV BIT(0)
#define SUSPEND_CONN BIT(1)
#define DEEPSLEEP_RETENTION_ADV BIT(2)
#define DEEPSLEEP_RETENTION_CONN BIT(3)
```

`SUSPEND_DISABLE` 表示 sleep disable，不允许 MCU 进入 suspend 和 deepsleep retention。

`SUSPEND_ADV` 和 `DEEPSLEEP_RETENTION_ADV` 分别用于控制 Advertising state 时 MCU 进入 suspend 和 deepsleep retention。

`SUSPEND_CONN` 和 `DEEPSLEEP_RETENTION_CONN` 分别用于控制 Conn state Slave role 时 MCU 进入 suspend 和 deepsleep retention。

SDK 低功耗 sleep mode 的设计上，deepsleep retention 是 suspend 的替代模式，目的是降低 sleep mode 的功耗。

以 Conn state slave role 为例，SDK 首先得看到 `bltPm.suspend_mask` 中 `SUSPEND_CONN` 是否生效，才可以进入 suspend。在可以进入 suspend 的基础上，根据实际情况再结合 `bltPm.suspend_mask` 中 `DEEPSLEEP_RETENTION_CONN` 是否生效，才能决定此时 suspend mode 是否被切换为 deepsleep retention mode。

所以如果 user 希望 MCU 进入 suspend，打开 `SUSPEND_ADV/SUSPEND_CONN` 即可；如果希望 MCU 进入 deepsleep retention mode，必须同时打开 `SUSPEND_CONN` 和 `DEEPSLEEP_RETENTION_CONN`。

该 API 最常用的 3 种情况如下：

- 1) `bls_pm_setSuspendMask(SUSPEND_DISABLE);`
MCU 不允许进入 sleep mode。
- 2) `bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN);`
MCU 在 Advertising state 和 Conn state Slave role 只允许进入 suspend，但是不允许进入 deepsleep retention。
- 3) `bls_pm_setSuspendMask(SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV | SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN);`
MCU 在 Advertising state 和 Conn state Slave role 允许进入 suspend 和 deepsleep retention，具体进入哪种 sleep mode 由当前 sleep 的时间长度决定，后面会详细介绍。

除了上面 3 种常用的情况，也可以出现一些特殊的用法，如：

- 1) `bls_pm_setSuspendMask(SUSPEND_ADV)`
只有 Advertising state 可以进入 suspend，Conn state Slave role 不允许进入 sleep mode。
- 2) `bls_pm_setSuspendMask(SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN)`，
只有 Conn state Slave role 可以进入 suspend 或 deepsleep retention，
Advertising state 不允许进入 sleep mode。

4.2.5 API `bls_pm_setWakeupSource`

user 通过上面的 `bls_pm_setSuspendMask` 设置 MCU 进入 sleep mode (suspend 或 deepsleep retention)，通过下面的 API 可设置 sleep mode 的唤醒源。

```
void bls_pm_setWakeupSource(u8 source);
```

source 可以选择唤醒源 `PM_WAKEUP_PAD`。

该 API 设置底层变量 `bltPm.wakeup_src`，SDK 中源码为：

```
void bls_pm_setWakeupSource(u8 src)
{
    bltPm.wakeup_src = src;
}
```

MCU 在 Advertising state 或 Conn state Slave role 进入 sleep mode，实际的唤醒源为：

```
bltPm.wakeup_src | 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 一定要在特定的时间点唤醒去处理接下来的 Adv Event 或 Brx Event。

每次调用 `bls_pm_setWakeupSource` 设置唤醒源后，一旦 MCU 进入 sleep mode 被唤醒后，`bltPm.wakeup_src` 会被清 0。

4.2.6 API `blc_pm_setDeepsleepRetentionType`

前面介绍了 deepsleep retention 根据 retention sram size 的差别有分为 16K sram retention 和 32K sram retention。当 sleep mode 中 deepsleep retention mode 生效时，SDK 会根据 user 的设置进入相应的 deepsleep retention mode。

下面 API 供 user 选择 deepsleep retention mode。

```
void blc_pm_setDeepsleepRetentionType (SleepMode_TypeDef sleep_type);
```

可选的模式只有以下两种：

```
typedef enum {  
    DEEPSLEEP_MODE_RET_SRAM_LOW16K    = 0x43,  
    DEEPSLEEP_MODE_RET_SRAM_LOW32K    = 0x07,  
} SleepMode_TypeDef;
```

SDK 中默认的 deepsleep retention mode 为 `DEEPSLEEP_MODE_RET_SRAM_LOW16K`，user 如果需要 retention 32K sram，初始化的时候调用如下 code 即可。

注意：该 API 的调用必须在 `blc_ll_initPowerManagement_module` 之后才能生效。

```
blc_pm_setDeepsleepRetentionType (DEEPSLEEP_MODE_RET_SRAM_LOW32K);
```

参考本文档第 2 章可知，Sram 内存分配默认是按照 deepsleep retention 16K Sram 设计的；根据第 1 章描述，我们知道选用不同的 IC 以及 deep retention size 值，需要选择不同的 software bootloader 启动文件和 boot.link，具体的映射关系及修改设置方法参考“software bootloader 介绍”小节。

如果当前 IC 为 8258，使用 deepsleep retention 32K Sram，则需要以下两个步骤的修改：

- 1) 选择 software bootloader 文件为 `cstartup_8258_RET_32K.S`;
- 2) 根据修改 boot.link 文件：将 SDK/boot/boot_32k_retn_8253_8258.link 文件内容替换到 SDK 根目录下的 boot.link 文件中。

其他 IC 的设置同上类似，用户根据实际情况修改即可。

4.2.7 PM 软件处理流程

低功耗管理的软件处理流程，下面将使用代码与伪代码相结合的方式来说明，目的是为了让 user 了解处理流程的所有逻辑细节。

4.2.7.1 blt_sdk_main_loop

SDK 中，blt_sdk_main_loop 在一个 while (1) 的结构中被反复调用。

```
while(1)
{
    //////////////////////////////////// BLE entry ////////////////////////////////////
    blt_sdk_main_loop();

    //////////////////////////////////// UI entry ////////////////////////////////////
    //UI task

    //////////////////////////////////// user PM config ////////////////////////////////////
    //blt_pm_proc();
}
```

blt_sdk_main_loop 函数在 while(1) 中不断被执行，BLE 低功耗管理的 code 在 blt_sdk_main_loop 函数中，所以低功耗管理的 code 也是一直在被执行。

下面是 blt_sdk_main_loop 函数中低功耗管理逻辑的实现。

```
int blt_sdk_main_loop(void)
{
    .....
    if(bltPm.suspend_mask == SUSPEND_DISABLE) // SUSPEND_DISABLE, can not
    {                                           // enter sleep mode
        return 0;
    }

    if( (Link Layer State == Advertising state) || (Link Layer State == Conn state Slave role) )
    {
        if(Link Layer is in Adv Event or Brx Event) //RF is working, can not enter
        {                                           //sleep mode
            return 0;
        }
        else
        {
            blt_brx_sleep(); //process sleep & wakeup
        }
    }
    return 0;
}
```

- 1) 当 bltPm.suspend_mask 为 SUSPEND_DISABLE 时，直接退出，不会执行 blt_brx_sleep 函数。所以 user 使用 bls_pm_setSuspendMask(SUSPEND_DISABLE)时，低功耗管理的逻辑就会完全失效，MCU 不会进入低功耗，while(1)的 loop 一直在执行。
- 2) 如果 Advertising State 的 Adv Event 或 Conn state Slave role 的 Brx Event 正在执行，blt_brx_sleep 函数也不会被执行，这是因为此时 RF 的任务正在运行，SDK 需要保证 Adv Event/Brx Event 结束之后才能进 sleep mode。

当以上两个条件都不满足时，才去执行 blt_brx_sleep 函数。

4.2.7.2 blt_brx_sleep

blt_brx_sleep 函数的逻辑实现如下所示。

注意：这里以默认的 deepsleep retention 16K Sram 来说明。

```
void blt_brx_sleep(void)
{
    if( (Link Layer state == Adv state)&& (bltPm.suspend_mask &SUSPEND_ADV) )
    {
        //当前广播状态，允许进 suspend
        T_wakeup = T_advertising + advInterval;

        "BLT_EV_FLAG_SUSPEND_ENTER" event callback execution

        T_sleep = T_wakeup - clock_time();
        if( bltPm.suspend_mask & DEEPSLEEP_RETENTION_ADV &&
            T_sleep > bltPm.deepRet_advThresTick )
        {
            //suspend is automatically switched to deepsleep retention
            cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K,
                             PM_WAKEUP_TIMER | bltPm.wakeup_src,
                             T_wakeup); //suspend
            //MCU被唤醒后PC值reset to 0, 将重新执行software bootloader
            //(cstartup_8258_16K.S)、system initialization等
        }
        else
        {
            cpu_sleep_wakeup ( SUSPEND_MODE,
                             PM_WAKEUP_TIMER | bltPm.wakeup_src,
                             T_wakeup);
        }
    }
}
```

“BLT_EV_FLAG_SUSPEND_EXIT” event callback execution

```

if(suspend 是被 GPIO PAD 唤醒)
{
    “BLT_EV_FLAG_GPIO_EARLY_WAKEUP” event callback execution
}
}
else if((Link Layer state == Conn state Slave role)&& (SuspendMask&SUSPEND_CONN) )
{
    //当前 Conn state, 进 suspend
    if(conn_latency != 0)
    {
        latency_use = bls_calculateLatency();
        T_wakeup = T_brx + (latency_use +1) * conn_interval;
    }
    else
    {
        T_wakeup = T_brx + conn_interval;
    }
}

```

“BLT_EV_FLAG_SUSPEND_ENTER” event callback execution

```

T_sleep = T_wakeup – clock_time();
if( bltPm.suspend_mask & DEEPSLEEP_RETENTION_CONN &&
    T_sleep > bltPm.deepRet_connThresTick )
{
    //suspend is automatically switched to deepsleep retention
    cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K,
        PM_WAKEUP_TIMER | bltPm.wakeup_src,
        T_wakeup); //suspend
    //MCU被唤醒后PC值reset to 0, 将重新执行software bootloader
    //(cstartup_8258_16K.S)、system initialization等
}
else
{
    cpu_sleep_wakeup ( SUSPEND_MODE,
        PM_WAKEUP_TIMER | bltPm.wakeup_src,
        T_wakeup);
}

```

“ BLT_EV_FLAG_SUSPEND_EXIT” event callback execution

```

if(suspend 是被 GPIO PAD 唤醒)
{
    “BLT_EV_FLAG_GPIO_EARLY_WAKEUP” event callback execution
}

```

调整 BLE 时序相关的处理

```

    }
}

bltPm.wakeup_src = 0;
bltPm.user_latency = 0xFFFF;
}

```

上面 `blt_brx_sleep` 函数的流程看起来比较复杂, 我们的分析从最简单的情况开始: 首先 `conn_latency` 方面只考虑 `conn_latency=0` 的情况。其次暂时不考虑 `deepsleep retention` 生效的情况。此时只有 `suspend`, 跟 Telink 上一代 826x 系列 IC 的低功耗管理是一致的。当应用层设置 `suspend mask` 时使用的方法是 `bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN)` 时, 就对应这种情况。

结合文档前面介绍的 `controller event`, 这里看到几个 `suspend` 相关 `event` 回调函数的执行的时机: `BLT_EV_FLAG_SUSPEND_ENTER`、`BLT_EV_FLAG_SUSPEND_EXIT`、`BLT_EV_FLAG_GPIO_EARLY_WAKEUP`。

Link Layer Advertising state 时 `bltPm.suspend_mask` 中 `SUSPEND_ADV` 生效, 或者 Link Layer Conn state slave role 时 `bltPm.suspend_mask` 中 `SUSPEND_CONN` 生效, 可以进入 `suspend`。

在 `suspend` 模式下, 最终调用了 `driver` 中的 API `cpu_sleep_wakeup`:

```

cpu_sleep_wakeup ( SUSPEND_MODE,
                  PM_WAKEUP_TIMER | bltPm.wakeup_src,
                  T_wakeup );

```

唤醒源为 `PM_WAKEUP_TIMER | bltPm.wakeup_src`, `Timer` 无条件生效, 是为了保证 MCU 一定要在下一个 `Adv Event`、`Brx Event` 到来前唤醒。唤醒时间 `T_wakeup` 请参考本文档前面的图“sleep timing for Advertising state & Conn state Slave role”。

`blt_brx_sleep` 函数退出时将 `bltPm.wakeup_src` 和 `bltPm.user_latency` 的值复位, 所以需要注意 API `bls_pm_setWakeupSource` 设置唤醒源的生命周期, 每次设置的值只对最近一次要进入的 `sleep mode` 有效。同理 API `bls_pm_setManualLatency` 也一样。

4.2.8 deepsleep retention 的详细分析

引入 `deepsleep retention`, 对上面的软件处理流程继续分析。

当应用层按如下设置时, `deepsleep retention mode` 被打开。

```

bls_pm_setSuspendMask( SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV

```

```
| SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN);
```

4.2.8.1 API blc_pm_setDeepsleepRetentionThreshold

Advertising state/Conn state slave role 中，满足以下条件，suspend 才会被自动切换为 deep retention:

```
if( bltPm.suspend_mask & DEEPSLEEP_RETENTION_ADV &&
    T_sleep > bltPm.deepRet_advThresTick )
```

```
if( bltPm.suspend_mask & DEEPSLEEP_RETENTION_CONN &&
    T_sleep > bltPm.deepRet_connThresTick )
```

第一个条件 bltPm.suspend_mask 中 DEEPSLEEP_RETENTION_ADV 需要生效，前面已经介绍过。

第二个条件 $T_{\text{sleep}} > \text{bltPm.deepRet_advThresTick}$ 或
 $T_{\text{sleep}} > \text{bltPm.deepRet_connThresTick}$ 中，

$T_{\text{sleep}} = T_{\text{wakeup}} - \text{clock_time}()$ 表示唤醒时间减去实时时间，即 sleep 的持续时间。这个条件的意义是：当 sleep 的持续时间超过特定的时间阈值时，MCU 的 sleep mode 才会被切换为 deep retention。

我们先介绍两个时间阈值设置的 API，如下所示为源码，设置时间单位为 mS。

```
void blc_pm_setDeepsleepRetentionThreshold( u32 adv_thres_ms,
                                             u32 conn_thres_ms)
{
    bltPm.deepRet_advThresTick = adv_thres_ms *
                                CLOCK_16M_SYS_TIMER_CLK_1MS;
    bltPm.deepRet_connThresTick = conn_thres_ms *
                                CLOCK_16M_SYS_TIMER_CLK_1MS;
}
```

API `blc_pm_setDeepsleepRetentionThreshold` 用于设置 `suspend` 切换到 `deepsleep retention` 触发条件中的时间阈值，这个设计是为了追求更低的功耗。

参考前文“sleep wake_up 后运行流程”部分的说明可知，`suspend mode wake_up` 后，可以立刻回到 `suspend` 前的环境继续运行。上面软件流程中在 `T_wakeup` 醒来之后，可以立刻开始执行 `Adv Event/Brx Event` 任务。

而 `deepsleep retention wake_up` 后需要回到“Run software bootloader”开始的地方，和 `suspend wake_up` 相比，需要多运行 3 个步骤（Run software bootloader + System initialization + User initialization），才能再次回到 `main_loop` 中执行 `Adv Event/Brx Event` 任务。

以 Conn state slave role 为例，下图表示 sleep mode 分别为 `suspend` 和 `deepsleep retention` 时的 timing（时序）& power（功耗）对比。

两个相邻的 `Brx event` 之间的时间差值 T_{cycle} 即当前时间周期。将 `Brx Event` 的功耗平均化，等效电流为 I_{brx} ，持续时间为 t_{brx} （这里取名 t_{brx} 是为了和前面已有概念 T_{brx} 做区分）。`Suspend` 的底电流为 I_{suspend} ，`deep retention` 的底电流为 I_{deepRet} 。

“Run software bootloader + System initialization + User initialization”的过程平均电流等效为 I_{init} ，持续的总时间为 T_{init} 。实际的应用中， T_{init} 的值需要 user 去把控和测量，后面会介绍如何实现。

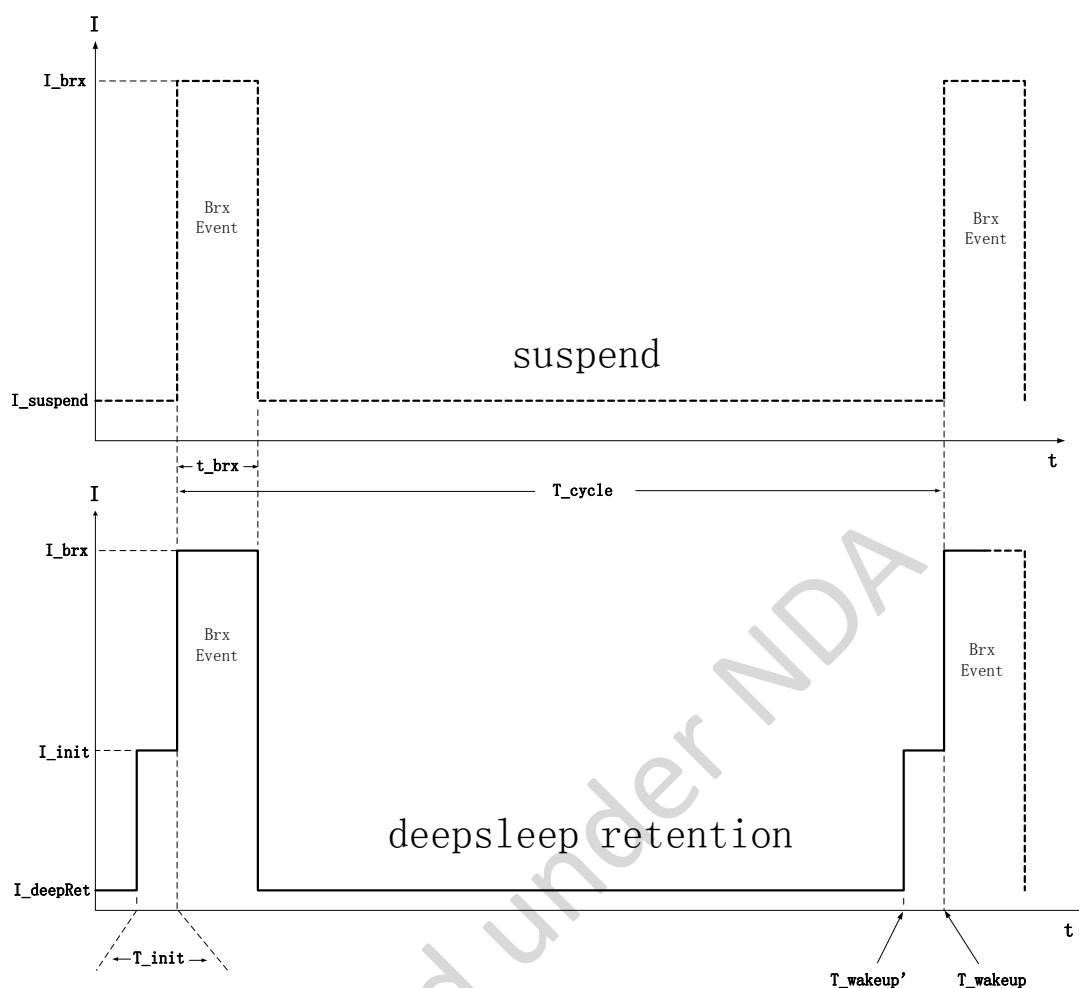


图 4-4 suspend & deepsleep retention timing & power

以下是图中几个名词说明。

- ✧ T_{cycle} : 两个相邻的 Brx event 之间的时间差值
- ✧ I_{brx} : 将 Brx Event 的功耗平均化, 等效电流为 I_{brx}
- ✧ t_{brx} : I_{brx} 持续时间
- ✧ $I_{suspend}$: suspend 底电流
- ✧ $I_{deepRet}$: deep retention 的底电流
- ✧ I_{init} : Software bootloader + System initialization + User initialization 过程的等效平均电流
- ✧ T_{init} : I_{init} 持续的总时间

那么，使用了 suspend mode 的 Brx 平均功耗为

$$I_{avgSuspend} = I_{brx} * t_{brx} + I_{suspend} * (T_{cycle} - t_{brx})$$

由于 T_{cycle} 远大于 t_{brx} ， $(T_{cycle} - t_{brx})$ 约等于 T_{cycle} 。

$$I_{avgSuspend} = I_{brx} * t_{brx} + I_{suspend} * T_{cycle}$$

使用了 deepsleep retention mode 的 Brx 平均功耗为

$$\begin{aligned} I_{avgDeepRet} &= I_{brx} * t_{brx} + I_{init} * T_{init} + I_{deepRet} * (T_{cycle} - t_{brx}) \\ &= I_{brx} * t_{brx} + I_{init} * T_{init} + I_{deepRet} * T_{cycle} \end{aligned}$$

比较 $I_{avgSuspend}$ 和 $I_{avgDeepRet}$ ，去掉相同的“ $I_{brx} * t_{brx}$ ”，最终比较的部分为：

$$\begin{aligned} I_{avgSuspend} - I_{avgDeepRet} &= I_{suspend} * T_{cycle} - I_{init} * T_{init} - I_{deepRet} * T_{cycle} \\ &= T_{cycle} (I_{suspend} - I_{deepRet}) - (T_{init} * I_{init}) / T_{cycle} \end{aligned}$$

对于功耗调试正确的应用程序（硬件电路和软件的功耗调试都正确），最终 $(I_{suspend} - I_{deepRet})$ 是一个固定的值，比如 suspend 为 30uA，deepsleep retention 为 2uA 时， $(I_{suspend} - I_{deepRet}) = 28uA$ ； $(T_{init} * I_{init})$ 最终也是固定的值，比如 I_{init} 为 3mA， T_{init} 为 400 uS， $(T_{init} * I_{init})$ 为 1200 uA*uS

$$\begin{aligned} I_{avgSuspend} - I_{avgDeepRet} &= T_{cycle} (28 - 1200 / T_{cycle}) \end{aligned}$$

可以看到，当 T_{cycle} 的值较小时（例子中 $T_{cycle} < 43$ mS），使用 suspend mode 最终的平均功耗更低；当 T_{cycle} 较大时（ $T_{cycle} > 43$ mS），使用 deepsleep retention mode 最终的平均功耗会更低。

注意，PM 软件处理流程部分可以看到，当 $T_{sleep} > 43ms$ 的时候才会将 suspend 自动切换为 deepsleep retention。一般我们认为 MCU working 时间（Brx Event + UI task）比较短，当 T_{cycle} 较大时，可以认为 T_{sleep} 约等于 T_{cycle} 。

那么初始化的时候按照如下设置的话，MCU 对于 T_{sleep} 大于 43ms 的 suspend 自动切换为 deepsleep retention，对于 T_{sleep} 小于 43ms 的 suspend 还是保持不变。

```
bhc_pm_setDeepsleepRetentionThreshold(43, 43);
```

以一个 10ms connection interval * (99 + 1) = 1S 的长连接为例进行说明：

在 Conn state slave role 时，由于应用层的任务、手动 latency 的设置等，会导致 MCU suspend 时可能出现 10ms、20ms、50ms、100ms、1S 等时间值。根据 43ms 的阈值设置，MCU 会自动将 50ms、100ms、1S 等 suspend 切换为 deepsleep retention，而 10ms、20ms 等 suspend 还是维持 suspend，这样的处理可以保证一个最优的功耗。

由于 deepsleep retention 的功耗比 suspend 低，并且“Run software bootloader + System initialization + User initialization”3 个步骤的存在会多出一部分的功耗，根据以上分析，一定是 T_{cycle} 大于某个临界值之后，使用 deepsleep retention 才会更省功耗。上面例子中的数值只是简单的 demo，user 在实现功耗优化时需要根据一定的方法先测出上面公式中对应的数值，最终才可以确定临界值。

只要程序设计上参考 SDK 中的 demo，且在 user initialization 这部分没有错误的增加很大的时间消耗，上面的 T_{cycle} 临界值都不会太大。一般超过 100ms 以上的 T_{cycle} ，使用 deepsleep retention mode 的功耗都会更低。

4.2.8.2 bhc_pm_setDeepsleepRetentionEarlyWakeupTiming

参考上面的图“suspend & deepsleep retention timing & power”，可以看到，suspend wake_up 的时间点 T_{wakeup} 正好是下一个 Brx Event 开始的时间点，对应 BLE master 端开始发包的时间。

deepsleep retention wake_up 的时间点如果也设置为 T_{wakeup} 的话，就会有问题：此时 MCU 醒来，还需要经过 T_{init} 的时间（Run software bootloader + System initialization + User initialization 3 个步骤消耗的时间）才能开始 Brx Event，已经错过了 BLE master 端发包的时间点。

为了解决这个问题，MCU wake_up 的时间点需要提前到 T_{wakeup}' 。

$$T_{wakeup}' = T_{wakeup} - T_{init}$$

以 Conn state slave role 为例（Advertising state 的处理完全相同），上面 PM 软件处理流程对 blt_brx_sleep 函数中 deepsleep retention wake_up 的时间点进行优化改善，如下：

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K,
                  PM_WAKEUP_TIMER | bltPm.wakeup_src,
                  T_wakeup - bltPm.deepRet_earlyWakeupTick);
```

T_wakeup 是 BLE stack 自动计算的时间点，user 只需要知道 T_init 的值，就可以通过下面 API 来设置 deepsleep retention wake_up 的提前量。

```
void blc_pm_setDeepsleepRetentionEarlyWakeupTiming(u32 earlyWakeup_us)
{
    bltPm.deepRet_earlyWakeupTick = earlyWakeup_us *
                                    CLOCK_16M_SYS_TIMER_CLK_1US;
}
```

User 将测量到的 T_init 的值直接设置到上面 API 即可，或者设置的值比 T_init 略大一点，但不能小于这个值。

4.2.8.3 T_init 的优化和测量

这部分的介绍大量用到 ram_code、retention_data、deepsleep retention area 等 Sram 相关的概念，请 user 先参考第 2 章中 Sram 空间分配的详细介绍。

1) T_init timing

由图"suspend & deepsleep retention timing & power”，结合上面已有的分析可知，对于 T_cycle 较大的情况，sleep mode 使用 deepsleep retention 功耗更低，但这种模式下 T_init 的时间是必须的，无法避开。为了尽量降低长时间睡眠的功耗，需要将 T_init 的时间尽量优化到最小。T_init 的值基本上是保持稳定的，不需要去做优化。

T_init 是 Run software bootloader + System initialization + User initialization 3 个步骤消耗的时间总和，将这 3 个步骤拆开分析，先定义各步骤的时间。

- ✧ T_cstatup 为 Run software bootloader 的时间（Run software bootloader 就是执行汇编文件 cstartup_XXX.S）。
- ✧ T_sysInit 为执行 system initialization 的时间。
- ✧ T_userInit 为执行 user initialization 的时间。

那么，

$T_{init} = T_{cstatup} + T_{sysInit} + T_{userInit}$ 。

下图为 T_init 的时序图。

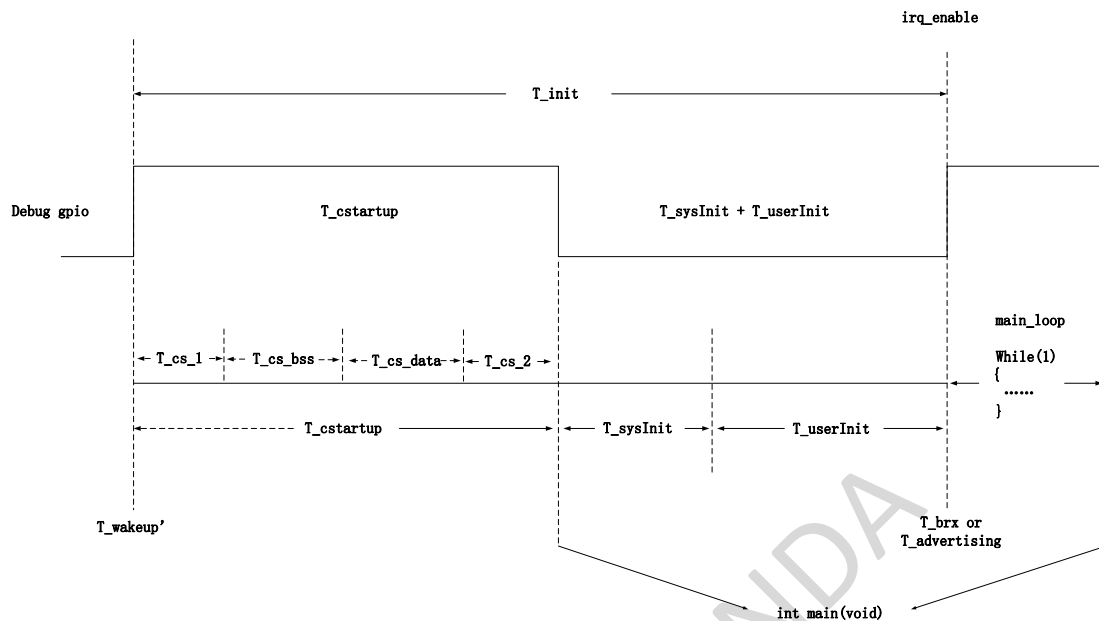


图 4-5 T_init timing

结合前面已有的图和概念继续分析。

T_wakeup 是下一个 Adv Event/Brx Event 开始的时间点, T_wakeup' 是 MCU 提前 wake_up 的时间。

MCU wake_up 后, 执行 cstartup_xxx.S, 然后跳转到 main 函数, 执行 System initialization + User initialization, 之后进入 main_loop。一旦进入 main_loop 便可以处理 Adv Event/Brx Event, 所以 T_userInit 结束的时间对应 Adv Event/Brx Event 开始的时间 T_brx/T_advertising。图中 irq_enable 这个函数是 T_userInit 和 main_loop 分界线, 和 SDK 上 code 是对应的。

SDK 上, T_sysInit 时间包括 cpu_wakeup_init、rf_drv_init、gpio_init、clock_init 执行的时间, 这些时间 SDK 已经做了优化, 运行时间达到最低。所以 T_sysInit 是一个固定的值, user 不用关注这个时间。SDK 优化这部分时间的方法是将这些初始化函数全部放到 ram_code 中运行, 所以和 Telink 上一代 826x 系列 IC 相比, 该 BLE SDK 消耗了更多的 ram_code, 需要更大的 Sram。

SDK 只对 T_cstartup 和 T_userInit 进行详细的介绍。

2) T_userInit

由前面的介绍可知，user initialization 在 power on、deepsleep wake_up、deepsleep retention wake_up 的时候都需要被执行。

对于没有使用 deepsleep retention mode 的应用来说，user initialization 不需要区分 deepsleep retention wake_up 和 power on/deepsleep wake_up。Telink 上一代 826x 系列 BLE SDK 中，所有的 user initialization 都是用下面函数即可完成。该 BLE SDK 中"8258_master_kma_dongle" project 也一样。

```
void user_init(void);
```

而对于 8x5x 使用了 deepsleep retention mode 的应用，为了尽量降低功耗，T_userInit 需要最优化。所以需要区分，deepsleep retention wake_up 时，user initialization 需要尽量快。

user_init 函数中所有的 initialization 可以被分为两类：一是硬件寄存器的初始化，二是 Sram 上逻辑变量的初始化。

根据 deepsleep retention mode 可以保持 Sram 前 16K（32K）不掉电的特性，我们可以将逻辑变量定义成 retention_data，这样的话 deepsleep retention wake_up 时就可以省掉逻辑变量初始化的时间。由于寄存器的状态无法被保持，deepsleep retention wake_up 时寄存器的初始化必须重新执行。

最终的实现方法是 deepsleep retention wake_up 时，执行优化过的 user_init_deepRetn 函数，power on 和 deepsleep wake_up 时执行 user_init_normal。如下 code 所示：

```
int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();

if( deepRetWakeUp ){
    user_init_deepRetn ();
}
else{
    user_init_normal ();
}
```

user 可以对比一下这两个函数的实现，下面是 SDK demo “8258_ble_remote” user_init_deepRetn 函数的实现。

```
_attribute_ram_code_ void user_init_deepRetn(void)
{
    #if (PM_DEEPSLEEP_RETENTION_ENABLE)
        blc_app_loadCustomizedParameters();
        blc_ll_initBasicMCU(); //mandatory
        rf_set_power_level_index (MY_RF_POWER_INDEX);
        blc_ll_recoverDeepRetention();
        app_ui_init_deepRetn();
    #endif
}
```

前 3 句(code blc_app_loadCustomizedParameters 到 rf_set_power_level_index) 是 BLE 初始化中必不可少的相关硬件寄存器的初始化;

blc_ll_recoverDeepRetention 是对 Link Layer 相关软硬件状态的恢复，由 stack 底层处理。

以上几个初始化都属于固定写法，user 不要去修改。

最后 app_ui_init_deepRetn 是 user 对应用层使用到的硬件寄存器的重新初始化。SDK demo “8258_ble_remote”中 GPIO 的唤醒设置、Led 灯状态的设置都属于硬件初始化。SDK demo “8258_module”中 UART 硬件寄存器的状态都需要重新初始化。

在 SDK demo 基础上，user initialization 如果增加了其他功能，这些新增功能的 initialization 节省时间的原则是：对每一条 initialization 的 code 进行分析，判断出是纯 Sram 变量还是硬件寄存器的操作。

- ✧ 如果是纯 Sram 变量的操作，将相应的 Sram 变量添加关键字 “_attribute_data_retention_”定义到 “retention_data”段，就可以保证 deepsleep retention wake_up 后不需要重新初始化，只需要该操作放到 user_init_normal 函数中就行了。
- ✧ 如果是硬件寄存器的操作，那么必须放到 user_init_deepRetn 函数中，确保硬件状态的正确性。

deepsleep retention wake_up 后的 T_userInit 就是 user_init_deepRetn 函数的执行时间，SDK 中也尽量将这部分的函数放到 ram_code 中以节省运行时间。

在 deepsleep retention area 空间足够的前提下，user 也需要将增加的硬件初始化相关函数放到 ram_code 中。

3) T_userInit 在 Conn state slave role 的优化

待补充。

4) T_cstartup

T_cstartup 是执行 cstartup_xxx.S (比如 cstartup_8258_RET_16K.S) 所消耗的时间, 请 user 参考 SDK 中 cstartup_8258_RET_16K.S 文件。

T_cstartup 按照时间顺序可以被拆成 4 个时间组成:

$$T_{cstartup} = T_{cs_1} + T_{cs_bss} + T_{cs_data} + T_{cs_2}$$

T_cs_1 和 T_cs_2 两个时间是固定的, user 无法修改它们, 不需要去关注。

T_cs_data 是 Sram 中“data”段的初始化时间。“data”段是已初始化的全局变量, 它们的初始值存储在 flash 的“data initial value”区域上。“data”段初始化的时间就是 MCU 从 flash “data initial value”区域上的初值拷贝到 Sram “data”段的过程。对应的汇编 code 如下:

```
tloadr    r1, DATA_I
tloadr    r2, DATA_I+4
tloadr    r3, DATA_I+8
COPY_DATA:
tcmp      r2, r3
tjge      COPY_DATA_END
tloadr    r0, [r1, #0]
tstorer   r0, [r2, #0]
tadd      r1, #4
tadd      r2, #4
tj        COPY_DATA
COPY_DATA_END:
```

因为 flash 数据拷贝速度相对比较慢 (给个参考: 16 byte 数据大概需要 7uS 时间), 如果“data”段的数据较多, 就会造成 T_cs_data 时间偏大, 最终导致 T_init 偏大。

SDK 中“data”段数据越少越好。user 可以参考文档前面介绍的方法去查看 list 文件中“data”段的大小。

如果“data”段较大, 优化方法为: 在 deepsleep retention area 空间足够的前提下, 将原先属于“data”段的变量加关键字“_attribute_data_retention_”定义到“retention_data”段上。

T_cs_bss 是 Sram 中“bss”段的初始化时间。“bss”段的初值为 0, 不需要从 flash 上去拷贝 data, 只需要将“bss”段对应的 Sram 全部清 0 即可。下面为“bss”段 Sram 清 0 操作对应的汇编 code:

```
tmov    r0, #0
tloadr  r1, DAT0 + 16
tloadr  r2, DAT0 + 20
```

ZERO:

```
tcmp    r1, r2
tjge    ZERO_END
tstorer  r0, [r1, #0]
tadd    r1, #4
tj      ZERO
```

ZERO_END:

T_cs_bss 是“bss”段数据清 0 操作的时间, 每个 word (4 byte)清 0 的速度非常快, 当“bss”较小时, T_cs_bss 很小。但如果“bss”段很大 (比如程序中定义了一个很大的全局数组 int AAA[2000] = {0}), T_cs_bss 的时间也会变大很多, 所以 user 还是需要注意, 可以在 list 文件中查看“bss”段的大小。

如果“bss”段偏大, 需要优化 T_cs_bss。优化方法和“data”段一样, 在 deepsleep retention area 空间足够的前提下, 将原先属于“bss”段的变量加关键字 “_attribute_data_retention_”定义到“retention_data”段上。

5) T_init 测量

根据以上介绍, 对 T_cstartup 和 T_userInit 的时间做优化, 将 T_init 优化到最小时间, 然后需要测量出 T_init, 填到 API blc_pm_setDeepSleepRetentionEarlyWakeupTiming。

T_init 的起点即 T_cstartup 的起点。T_cstartup 的起点是 cstartup_8258_RET_16K.S 文件中“__reset”这个点, 如下 code 所示。

__reset:

```
#if 0
@ add debug, PB4 output 1
tloadr    r1, DEBUG_GPIO    @0x80058a PB oen
tmov      r0, #139          @0b 11101111
tstorerb  r0, [r1, #0]

tmov      r0, #16           @0b 00010000
```

```
tstorerb    r0, [r1, #1]    @0x800583    PB output
#endif
```

结合图“T_init timing”中 Debug gpio 的示意,在“__reset”这里放了 Debug GPIO PB4 输出高的操作, user 只要将“#if 0”改成“#if 1”便可打开 PB4 输出高的操作。

T_cstartup 结束时间是下面“tjl main”的时间点。

```
tjl main

END:    tj    END
```

那么 main 函数开始的时间与 T_cstartup 结束的时间几乎是相等的。在 main 函数一开始的地方让 PB4 输出低, 如下所示。注意这个 DBG_CHN0_LOW 依赖于 app_config.h 中“DEBUG_GPIO_ENABLE”的打开。

```
_attribute_ram_code_ int main (void)    //must run in ramcode
{
    DBG_CHN0_LOW;    //debug
    cpu_wakeup_init();
    .....
}
```

PB4 的一高一低可以测量出 T_cstartup 的持续时间。

在 user_init_deepRetn 函数里面 T_userInit 结束地方添加 PB4 输出高的操作, 这样就可以实现图中 Debug gpio 的效果。User 可以通过示波器、逻辑分析仪等设备测量出 T_init、T_cstartup 的时间。User 在理解 GPIO 操作的基础上, 可根据自己的需要, 对 Debug gpio 的 code 进行修改, 以得到更多时间参数测量的结果, 如 T_sysInit、T_userInit 等。

4.2.9 Connection Latency

4.2.9.1 Connection latency 生效时的 Sleep 时序

前面关于 Conn state slave role 的 sleep mode 的介绍(参考图“sleep timing for Advertising state & Conn state Slave role”所示), 都是基于 connection latency (简称 conn_latency) 没有生效时的前提。

PM 软件处理流程上, $T_{wakeup} = T_{brx} + conn_interval$, 对应的 code 如下。

```
if(conn_latency != 0)
{
    latency_use = bls_calculateLatency();
    T_wakeup = T_brx + (latency_use + 1) * conn_interval;
}
else
```

```
{
    T_wakeup = T_brx + conn_interval;
}
```

当 BLE slave 经过 connection parameters update（连接参数更新）流程，conn_latency 生效后，sleep wake_up 的时间为

$$T_{\text{wakeup}} = T_{\text{brx}} + (\text{latency_use} + 1) * \text{conn_interval};$$

下图所示为一个 conn_latency 生效时的 sleep 时序，此时 latency_use=2。

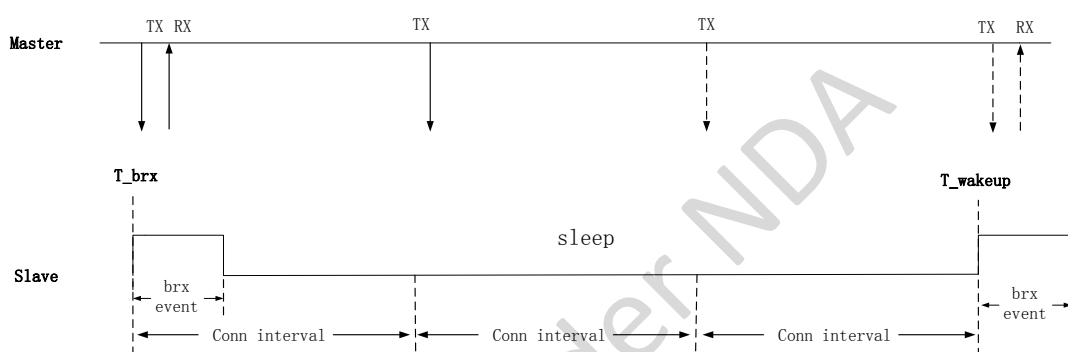


图 4-6 sleep timing for conn_latency valid

conn_latency 没有生效时，sleep 的时间最长不超过 1 个 connection interval（一般都比较小）。由于 conn_latency 的生效，sleep 的时间可能会出现一个比较大的值，如 1S、2S 等，系统功耗可以变得非常低。长 sleep 期间使用功耗更小的 deepsleep retention mode 才变得有意义。

4.2.9.2 latency_use 的计算

当 conn_latency 生效时，T_wakeup 的值是由 latency_use 决定的。说明 latency_use 并不是直接等于 conn_latency。

```
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)
{
    bltPm.user_latency = latency;
}
```

```
}

```

`bltPm.user_latency` 这个变量的初值为 `0xFFFF`。注意 PM 软件处理流程中 `blt_brx_sleep` 函数最后会强制将它再次复位为 `0xFFFF`，说明 API `bls_pm_setManualLatency` 设置的 `user_latency` 只对最近一次 sleep 管用，每次不同的 sleep 都需要重新设置。

`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 `connection 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

$$\text{latency_use} = \min(\text{system latency}, \text{user_latency})$$

即 `latency_use` 取 `system latency` 和 `user_latency` 中的较小值。

以上逻辑可以看出：如果 user 调用 API `bls_pm_setManualLatency` 设置的 `user_latency` 比 `system latency` 小，`user_latency` 将会作为最终的 `latency_use`，否则 `system latency` 将作为最终的 `latency_use`。

4.2.10 API `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 Timer tick)，即 `T_wakeup`。

```
u32 bls_pm_getSystemWakeupTick(void);
```

从 PM 软件处理流程 `blt_brx_sleep` 函数中可以看到，`T_wakeup` 的计算比较晚，已经很接近 `cpu_sleep_wakeup` 函数了，应用层只能在 `BLT_EV_FLAG_SUSPEND_ENTER` 事件回调函数里才能得到准确的 `T_wakeup`。

下面以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);

void ble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( blc_ll_getCurrentState() == BLS_LINK_STATE_CONN &&
        ((u32) (bls_pm_getSystemWakeupTick() - clock_time())) >
            80 * CLOCK_SYS_CLOCK_1MS) {
        bls_pm_setWakeupSource(PM_WAKEUP_PAD);
    }
}
```

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 sleep mode，在 conn_latency 没有生效的前提下，只要 Adv interval 和 conn_interval 的值不是特别大（一般设置在 100ms 以内），sleep 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 conn_latency 生效后（比如 conn_interval 为 10ms, conn_latency 为 99），可能某次 Conn state 时的 sleep 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 PAD 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，按键就不会丢失。

4.3 GPIO 唤醒的注意事项

4.3.1 唤醒电平有效时无法进入 sleep mode

由于 8x5x 的 GPIO 唤醒是靠高低电平唤醒，而不是上升沿下降沿唤醒，所以当配置了 GPIO PAD 唤醒时，比如设置了某个 GPIO PAD 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。

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。

user 在使用 Telink 的 GPIO PAD 唤醒时，要注意避免这个问题。

如果应用层没有很好的规避这个问题，在调用 `cpu_wakeup_sleep` 函数时发生了 GPIO PAD 唤醒源已经生效的情况，为了防止程序进入不可预知的逻辑，PM driver 做了一些改善：

1) suspend 和 deepsleep retention mode

如果是 suspend 和 deepsleep retention mode，都会很快退出函数 `cpu_wakeup_sleep`，该函数给出的返回值可能出现两种情况：

- ✧ PM 模块上检测到了 GPIO PAD 生效的状态，返回 `WAKEUP_STATUS_PAD`;
- ✧ PM 模块上没有检测到 GPIO PAD 生效的状态，返回 `STATUS_GPIO_ERR_NO_ENTER_PM`

2) deepsleep mode

如果是 deepsleep mode，PM driver 会在底层自动将 MCU reset（此时的 reset 跟 watchdog reset 效果一致），程序回到“Run hardware bootloader”开始重新运行。

针对以上问题，在 SDK demo “8258 ble remote”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 PAD 唤醒。

```
void ble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( blc_ll_getCurrentState() == BLS_LINK_STATE_CONN &&
        ((u32)(bls_pm_getSystemWakeupTick() - clock_time())) >
            80 * CLOCK_SYS_CLOCK_1MS) {
        bls_pm_setWakeupSource(PM_WAKEUP_PAD);
    }
}
```

当按键没有释放时，通过手动设置 latency 为 0 或者一个很小的值，使得 sleep 时间较短，确保 sleep 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 PAD 高电平唤醒。如下代码所示。

```
int user_task_flg = ota_is_working || scan_pin_need || key_not_released || DEVICE_LED_BUSY;

if(user_task_flg){

    bls_pm_setSuspendMask (SUSPEND_ADV | SUSPEND_CONN);

    #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 optimize
            bls_pm_setManualLatency(3);
        }
        else{
            bls_pm_setManualLatency(0); //latency off: 0
        }
    #else
        bls_pm_setManualLatency(0);
    #endif
}
```

MCU 进入 deepsleep 的两种情况：

- ✧ 一是连续 60S 没有任何事件会进入 deepsleep，这里的事件包括按键被按下，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，SDK 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（参考按键扫描章节的卡键处理）。

4.4 BLE 系统低功耗管理参考

在了解了该 BLE SDK 低功耗管理的实现原理基础上，user 可以很灵活地配置自己的低功耗管理，请参考 SDK demo “8258 ble remote” 低功耗管理的参考 code。下面做一些解释。

在 main_loop 的 PM configuration 部分添加 blt_pm_proc 函数，注意这个函数要放在 main_loop 的最后面，以保证运行时间上最接近 blt_sdk_main_loop。因此 blt_pm_proc 函数里面低功耗管理的配置，需要根据 UI entry 部分各种任务的处理情况来做相应设置。

blt_pm_proc 函数低功耗管理配置几个要点总结如下：

- 1) 某些任务需要关闭 sleep mode 时，如语音(ui_mic_enable)、红外等任务运行时，设置 bltm.suspend_mask 为 SUSPEND_DISABLE。
- 2) Advertising state 下连续广播时间达到 60S，设置 MCU 进入 deepsleep，唤醒源为 GPIO PAD（需要在 user initialization 部分提前设置按键 GPIO PAD）。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 Timer 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 连接时还一直在广播。user 需要根据自己的需求，对功耗进行评估后，决定如何处理 advertising state 的时间问题。

- 3) Conn state slave role 时，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 MCU 进入 deepsleep，唤醒源为 GPIO PAD，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（deepsleep 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。user 需要根据自己的需求和功耗状况决定如何实现。

Conn state slave role 时要进入 deepsleep，先调用 `bls_ll_terminateConnection` 向 master 发送一个 TERMINATE 命令，等到这个命令被 ack 后（此时会触发 `BLT_EV_FLAG_TERMINATE` 事件回调函数）再进入 deepsleep。这样做是为了确保 master 收到 slave 主动断连的请求后立刻断开。如果 slave 没有发送断连请求就进入 deepsleep，master 仍然处于连接状态并一直尝试去和 slave 同步，直到 connection timeout 触发。这个 connection timeout 时间可能很大（比如 20S），如果在 20S connection timeout 之前 slave 被唤醒并发广播尝试和 master 建立连接，由于 master 还处于上一次的连接状态中，会导致无法立刻和 slave 建立连接。应用上的体验就是回连速度很慢。

- 4) 当有一些任务不能被长时间的 sleep 破坏时，可以手动设置 `user_latency` 为 0。如 `key_not_released`、`DEVICE_LED_BUSY` 时调用 API `bls_pm_setManualLatency` 将 `user_latency` 设为 0，那么 `latency_use` 就是 0，`conn_interval` 为 10ms 时 sleep 时间不超过 10ms。
- 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，功耗稍微有点高，且按键扫描和 LED 任务的处理并不需要每个 `conn_interval` 都做一次，此时可以再做一些功耗优化。

`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，调用 `bls_pm_setManualLatency` (3) 后，sleep 的时间不会超过 4 个 `conn_interval`。`conn_interval` 为 10 ms 时，每 40 ms 醒来一次处理 LED 和按键扫描。

user 在使用这个优化时，需要自行根据 `conn_interval` 的值和任务响应时间来评估。

4.5 应用层定时唤醒

在 Advertising state 和 Conn state Slave role，不考虑 GPIO PAD 唤醒前提下，一旦进入 sleep mode，只能在 BLE SDK 计算好的时间点 T_{wakeup} 唤醒，user 无法在某一个特定的时间点将 sleep 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 API 和它的回调函数。

应用层定时唤醒 API:

```
void bls_pm_setAppWakeupLowPower(u32 wakeup_tick, u8 enable);
```

wakeup_tick 为定时唤醒的 System Timer tick 值;

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);
```

以 Conn state Slave role 为例:

当 user 使用 bls_pm_setAppWakeupLowPower 设置了应用层定时唤醒的 app_wakeup_tick, SDK 在进入 sleep 前, 会检查 app_wakeup_tick 是否在 T_{wakeup} 之前。

- ✧ 如果 app_wakeup_tick 在 T_{wakeup} 之前，如下图所示，就会在 app_wakeup_tick 触发 sleep 提前唤醒；
- ✧ 如果 app_wakeup_tick 在 T_{wakeup} 之后，MCU 还是会在 T_{wakeup} 唤醒。

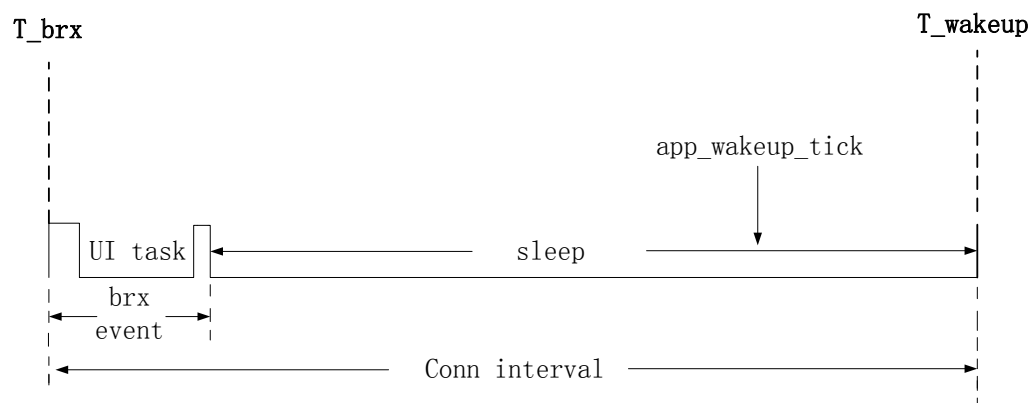


图 4-7 Early wake_up at app_wakup_tick

5 低电检测

电池电量检测（battery power detect/check），在 Telink BLE SDK 和相关文档中也可能出现其他的名字，包括：电池电量检测（battery power detect/check）、低电池检测（low battery detect/check）、低电量检测（low power detect/check）、电池检查（battery detect/check）等。比如 SDK 中相关文件和函数出现 battery_check、battery_detect、battery_power_check 等命名。

本文档统一以“低电检测（low battery detect）”这个名称进行说明。

5.1 低电检测的重要性

使用电池供电的产品，由于电池电量会逐渐下降，当电压低到一定的值后会引起很多问题：

- 1) 8x5x 工作电压的范围为 1.8V~3.6V。当电压低于 1.8V 时，8x5x 已经无法保证稳定的工作。
- 2) 当电池电压较低时，由于电源的不稳定，Flash 的“write”和“erase”操作可能有出错的风险，造成 program firmware 和用户数据被异常修改，最终导致产品失效。根据以往的量产经验，我们将这个可能出风险的低压阈值设定为 2.0V。

根据上面的描述，使用电池供电的产品，必须设定一个安全电压值（secure voltage），只有当电压高于这个安全电压的时候才允许 MCU 继续工作；一旦电压低于安全电压，MCU 停止运行，需要立刻被 shutdown（SDK 上使用进入 deepsleep mode 来实现）。

MCU 被 shutdown 之前，可以通过 UI 的一些行为（如 SDK demo “8258_ble_remote” 使用 LED 灯的快速闪烁）来告知产品使用者，这种 UI 行为称为低压报警。产品使用者看到低压报警的行为后，了解到当前电池已经处于低电状态，可以对电池进行充电或更换。

安全电压也称为报警电压，这个电压值的选取，目前 SDK 默认使用 2.0V。如果 user 在硬件电路中出现了不合理的设计，导致电源网络稳定性的恶化，安全电压值还需要继续提高，比如 2.1V、2.2V 等。

对于 Telink BLE SDK 开发实现的产品，只要使用了电池供电，低电检测都必须是该产品整个生命周期实时运行的任务，以保证产品的稳定性。

5.2 低电检测的实现

低电检测需要使用 ADC 对电源电压进行测量。user 请参考文档《8258 Datasheet》和 ADC driver 相关说明文档，先对 8x5x 的 ADC 模块进行必要的了解。

低电检测的实现，结合 SDK demo “8258_ble_remote”给出的实现来说明，参考文件 battery_check.h 和 battery_check.c。

必须确保 app_config.h 文件中宏“BATT_CHECK_ENABLE”是被打开的，这个宏默认是打开的，user 不要去修改它。

```
#define BATT_CHECK_ENABLE 1 //must enable
```

5.2.1 低电检测的注意事项

低电检测是一个基本的 ADC 采样任务，在实现 ADC 采样电源电压时，有一些需要注意的问题，说明如下。

5.2.1.1 必须使用 GPIO 输入通道

Telink 上一代 8267/8269 IC 上，支持在“VCC/VBAT”输入通道上对电源电压进行 ADC 采样。8x5x 的 ADC 输入通道上也保留了这个设计，对应下面变量 ADC_InputPchTypeDef 中最后一个“VBAT”。

但由于某些特殊的原因，8x5x “VBAT” channel 不能使用，所以 Telink 规定：不允许使用“VBAT”输入通道，必须使用 GPIO 输入通道。

可用的 GPIO 输入通道为 PB0~PB7、PC4、PC5 对应的 input channel。

```
/*ADC analog positive input channel selection enum*/
typedef enum {
    .....
    B0P,
    B1P,
    B2P,
    B3P,
    B4P,
    B5P,
    B6P,
    B7P,
    C4P,
    C5P,
    .....
}
```

```

    VBAT,
}ADC_InputPchTypeDef;

```

使用 GPIO input channel 对电源电压进行 ADC 采样，有两种实现方式。

1) 电源连接到 GPIO input channel

在硬件电路设计上，将电源直接和 GPIO input channel 连接。ADC 初始化时，将 GPIO 设为高阻态（ie、oe、output 全部设 0），此时 GPIO 上的电压等于电源电压，直接进行 ADC 采样即可。

2) 电源不接触 GPIO input channel

硬件电路不需要电源和 GPIO input channel 连接。需要借助 GPIO 的输出高电平来测量。8x5x 内部电路结构设计可以保证 GPIO 输出高电平的电压值和电源电压值永远相等。

那么 GPIO 输出的高电平可以作为电源电压，通过该 GPIO input channel 进行 ADC 采样。

目前“8258_ble_remote”选择的 GPIO input channel 是 PB7，采样了第 2 种“电源不接触 GPIO input channel”方式。

选择 PB7 为 GPIO input channel，PB7 作为普通 GPIO 功能，初始化时所有状态（ie、oe、output）使用默认状态即可，不做特殊修改。

```

#define GPIO_VBAT_DETECT      GPIO_PB7
#define PB7_FUNC              AS_GPIO
#define PB7_INPUT_ENABLE      0
#define ADC_INPUT_PCHN        B7P

```

需要进行 ADC 采样时，PB7 输出高电平：

```

gpio_set_output_en(GPIO_VBAT_DETECT, 1);
gpio_write(GPIO_VBAT_DETECT, 1);

```

ADC 采样结束后，可以将 PB7 的输出态关掉。由于“8258_ble_remote”硬件电路上 PB7 管脚是悬空的（没有和其他电路连接），输出高电平并不会造成任何漏电，所以 SDK 上并没有将 PB7 的输出态关掉。

5.2.1.2 只能使用差分模式

虽然 8x5x ADC input mode 同时支持单端模式 (Single Ended Mode) 和差分模式 (Differential Mode)，但由于某些特定的原因，Telink 规定：只能使用差分模式，单端模式不允许使用。

差分模式的 input channel 分为 positive input channel (正端输入通道) 和 negative input channel (负端输入通道)，被测量的电压值为 positive input channel 电压减去 negative input channel 电压得到的电压差。

如果 ADC 采样的 input channel 只有 1 个，使用差分模式时，将当前 input channel 设置为 positive input channel，将 GND 设为 negative input channel。这样二者的电压差和 positive input channel 电压相等。

SDK 中低压检测使用了差分模式，code 如下。“#if 1”与“#else”分支是完全相同的功能设置，“#if 1”只是为了让 code 运行更快以节省时间。可以通过看“#else”来理解，adc_set_ain_channel_differential_mode API 中选择了 PB7 作为 positive input channel，GND 作为 negative input channel。

```
#if 1 //optimize, for saving time
    //set misc channel use differential_mode,
    //set misc channel resolution 14 bit, misc channel differential mode
    analog_write (anareg_adc_res_m, RES14 | FLD_ADC_EN_DIFF_CHN_M);
    adc_set_ain_chn_misc(ADC_INPUT_PCHN, GND);
#else
    ///set misc channel use differential_mode,
    adc_set_ain_channel_differential_mode(ADC_MISC_CHN,
        ADC_INPUT_PCHN, GND);
    //set misc channel resolution 14 bit
    adc_set_resolution(ADC_MISC_CHN, RES14);
#endif
```

5.2.1.3 必须使用 Dfifo 模式获得 ADC 采样值

Telink 上一代 826x 系列 IC 都是使用读寄存器的方式来获取 ADC 采样结果。而对于 8x5x，Telink 规定：只能使用 Dfifo 模式来实现 ADC 采样值的读取。可参考 dirver 中如下函数的实现。

```
unsigned int adc_sample_and_get_result(void);
```

5.2.1.4 不同的 ADC 任务需要切换

参考《8258 Datasheet》可知,ADC 状态机包括 Left、Right、Misc 等几个 channel。由于一些特殊的原因,这些 state channel 无法同时工作,Telink 规定:ADC 状态机中的 channel 必须独立运行,不能同时工作。

低压检测作为一种最基本的 ADC 采样,使用的是 Misc channel。User 如果需要除低压检测外其他的 ADC 的任务,也需要使用 Misc channel。Amic Audio 使用的是 Left channel。低压检测无法与 Amic Audio 以及其他 ADC 任务同时运行,必须采用切换的方式来实现。

5.2.2 低电检测单独使用

user 将“8258_ble_remote” app_config.h 文件中的宏“BLE_AUDIO_ENABLE”定义为 0 (关闭 Audio 所有功能),就可以获得 ADC 只被低压检测使用的 demo。或者直接参考“8258_module”的低压检测 demo。

5.2.2.1 低电检测初始化

参考 adc_vbat_detect_init 函数的实现。

ADC 初始化的顺序必须满足下面的流程:先 power off (掉电) sar adc,然后配置其他参数,最后 power on (上电) sar adc。所有 ADC 采样的初始化都必须遵循这个流程。

```
void adc_vbat_detect_init(void)
{
    /*****power off sar adc*****/
    adc_power_on_sar_adc(0);

    //add ADC configuration

    /*****power on sar adc*****/
    //note: this setting must be set after all other settings
    adc_power_on_sar_adc(1);
}
```

Sar adc power on 与 power off 之前的配置,user 尽量不要去修改,使用这些默认的设置就行。User 如果选择了不同的 GPIO input channel,直接修改宏“ADC_INPUT_PCHN”的定义即可。User 的硬件电路如果采用了“电源连接到 GPIO input channel”的设计,需要将“GPIO_VBAT_DETECT”输出高电平的操作去掉。

adc_vbat_detect_init 初始化函数在 app_battery_power_check 中调用的 code 为:

```
if(!adc_hw_initialized){  
    adc_hw_initialized = 1;  
    adc_vbat_detect_init();  
}
```

这里使用了一个变量 `adc_hw_initialized`, 只有该变量为 0 时调用一次初始化, 并将其置 1; 该变量为 1 时不再初始化。`adc_hw_initialized` 在下面 API 中也会被操作。

```
void battery_set_detect_enable (int en)  
{  
    lowBattDet_enable = en;  
    if(!en){  
        adc_hw_initialized = 0;    //need initialized again  
    }  
}
```

使用了 `adc_hw_initialized` 的设计可以实现的功能有:

1) 与其他 ADC 任务 (“ADC other task”) 的切换

先不考虑 sleep mode (suspend/deepsleep retention) 的影响, 只分析低电检测与其他 ADC 任务的切换。

因为需要考虑低电检测与其他 ADC 任务的切换使用, 可能出现 `adc_vbat_detect_init` 被多次执行, 所以不能写到 user initialization 中, 必须在 `main_loop` 里实现。

第一次执行 `app_battery_power_check` 函数时, `adc_vbat_detect_init` 被执行, 且后面不会被反复执行。

一旦 “ADC other task” 需要执行时, 将抢走 ADC 的使用权, 确保 “ADC other task” 初始化时必须调用 `battery_set_detect_enable(0)`, 此时会将 `adc_hw_initialized` 清 0。

等 “ADC other task” 完成后, 交出 ADC 的使用权。`app_battery_power_check` 再次执行, 由于 `adc_hw_initialized` 值为 0, 必须再次执行 `adc_vbat_detect_init`, 这样就保证了低电检测每次切回来时都会重新初始化。

2) 对 suspend 和 deepsleep retention 的自适应处理

将 sleep mode 考虑进来。

adc_hw_initialized 这个变量使用必须定义成一个“data”段或“bss”段上的变量，不能定义到 retention_data 上。定义在“data”段或“bss”上可以保证每次 deepsleep retention wake_up 后在执行 software bootloader(即 cstartup_xxx.S) 时这个变量会被重新初始化为 0；而 sleep wake_up 后这个变量可以保持不变。

adc_vbat_detect_init 函数里面配置的 register 的共同特征是：在 suspend mode 下不掉电，可以保存状态；在 deepsleep retention mode 下会掉电。

如果 MCU 进入 suspend mode，醒来后再次执行 app_battery_power_check 时，adc_hw_initialized 的值和 suspend 之前一致，不需要重新执行 adc_vbat_detect_init 函数。

如果 MCU 进入 deepsleep retention mode，醒来后 adc_hw_initialized 为 0，必须重新执行 adc_vbat_detect_init，ADC 相关的 register 状态需要被重新配置。

adc_vbat_detect_init 函数中设定 register 的状态可以在 suspend 期间保持不掉电。

参考文档“低功耗管理”部分对 suspend mode 的说明可知，Dfifo 相关的寄存器在 suspend mode 会掉电，所以以下两句 code 没有放到 adc_vbat_detect_init 函数中，而是在 app_battery_power_check 函数中，确保每次低电检测前都重新设置。

```
adc_config_misc_channel_buf((u16 *)adc_dat_buf, ADC_SAMPLE_NUM<<2);  
dfifo_enable_dfifo2();
```

SDK 中对 adc_vbat_detect_init 函数添加了关键字“_attribute_ram_code_”以设置为 ram_code，最终目的是为了优化长睡眠连接态的功耗。比如对典型的 $10\text{ms} * (99+1) = 1\text{S}$ 的长睡眠连接，每 1S 醒来一次，中间的长睡眠使用的是 deepsleep retention mode，那么每次醒来后 adc_vbat_detect_init 一定会重新执行一次，加入到 ram_code 后执行速度会变得更快。

这个“_attribute_ram_code_”不是必须的。在产品应用中，user 可以根据 deepsleep retention area 的使用情况，结合功耗测试的结果，来决定是否将此函数放入到 ram_code 中。

5.2.2.2 低电检测处理

在 `main_loop` 中，调用 `app_battery_power_check` 函数实现低电检测的处理，相关 code 如下：

```
_attribute_data_retention_ u8 lowBattDet_enable = 1;

void battery_set_detect_enable (int en)
{
    lowBattDet_enable = en;
    if(!en){
        adc_hw_initialized = 0; //need initialized again
    }
}

int battery_get_detect_enable (void)
{
    return lowBattDet_enable;
}

if(battery_get_detect_enable() &&
    clock_time_exceed(lowBattDet_tick, 500000) ){
    lowBattDet_tick = clock_time();
    app_battery_power_check(VBAT_ALRAM_THRES_MV);
}
```

`lowBattDet_enable` 默认值为1，低电检测是默认允许的，MCU上电后立刻可以开始低电检测。该变量需要设置成 `retention_data`，确保 `deepsleep retention` 不能修改它的状态。

只有在其他ADC任务需要抢占ADC使用权时，才能改变 `lowBattDet_enable` 的值：当其他ADC任务开始时，调用 `battery_set_detect_enable(0)`，此时 `main_loop` 中不会再调用 `app_battery_power_check` 函数；在其他ADC任务结束后，调用 `battery_set_detect_enable(1)`，交出ADC使用权，此时 `main_loop` 中又可以调用 `app_battery_power_check` 函数。

通过变量 `lowBattDet_tick` 来控制低电检测的频率，Demo中为每500mS执行一次低电检测。User可以根据自己的需求来修改这个时间值。

app_battery_power_check函数的具体实现看起来比较繁琐，涉及到低电检测的初始化、Dfifo的准备、数据的获取、数据的处理、低电报警的处理等等。

由于ADC的使用比较复杂，加上硬件电路上有一些特殊的限制，user很难理解所有的细节。这部分的处理流程上每个细节（本文档不会把每个细节都介绍清楚）的处理都是有讲究的，所以user不要尝试去修改，尽量使用原始的demo code。只有少量一些可以修改的地方，本文档会很明确的指出来；凡是文档里没有明确指出可以修改的地方，user都不能有任何改动。

ADC采样数据的获取使用了Dfifo mode，Dfifo默认采样8笔数据，去掉最大最小值后计算平均值。adc_vbat_detect_init函数里可以看到每个adc采样的周期为10.4uS，所以获取数据过程大概83us。

可以看到Demo中宏“ADC_SAMPLE_NUM”可以被修改为4，缩短ADC采样时间到41uS。推荐使用8笔数据的方法，计算结果会更加准确。

```
#define ADC_SAMPLE_NUM      8

#if (ADC_SAMPLE_NUM == 4)    //use middle 2 data (index: 1,2)
    u32 adc_average = (adc_sample[1] + adc_sample[2])/2;
#elif (ADC_SAMPLE_NUM == 8) //use middle 4 data (index: 2,3,4,5)
    u32 adc_average = (adc_sample[2] + adc_sample[3] + adc_sample[4] +
                      adc_sample[5])/4;
#endif
```

app_battery_power_check 函数被放到 ram_code 上，参考上面对“adc_vbat_detect_init” ram_code 的说明，也是为了节省运行时间，优化功耗。

这个“_attribute_ram_code_”不是必须的。在产品应用中，user 可以根据 deepsleep retention area 的使用情况，结合功耗测试的结果，来决定是否将此函数放入到 ram_code 中。

```
_attribute_ram_code_ int app_battery_power_check(u16 a1ram_vol_mv);
```

5.2.2.3 低压报警

app_battery_power_check 的参数 a1ram_vol_mv 指定低电检测的报警电压，单位为 mV。根据前文介绍，SDK 中默认设置为 2000 mV。在 main_loop 的低压检测中，当电源电压低于 2000mV 时，进入低压范围。

低压报警的处理 demo code 如下所示。低压后必须 shutdown MCU，不能再进行其他工作。

“8258_ble_remote”使用进入 deepsleep 的方式来实现 shutdown MCU，并且设置了按键可以唤醒遥控器。

低压报警的处理，除了必须 shutdown 外，user 可以修改其他的报警行为。

下面 code 中，使用 LED 灯做了 3 次快闪，告知产品使用者需要充电或更换电池。

```
if(batt_vol_mv < alram_vol_mv){
    #if (1 && BLT_APP_LED_ENABLE) //led indicate
        gpio_set_output_en(GPIO_LED, 1); //output enable
        for(int k=0;k<3;k++){
            gpio_write(GPIO_LED, LED_ON_LEVAL);
            sleep_us(200000);
            gpio_write(GPIO_LED, !LED_ON_LEVAL);
            sleep_us(200000);
        }
    #endif

    analog_write(DEEP_ANA_REG2, LOW_BATT_FLG); //mark
    cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);
}
```

“8258_ble_remote”被 shutdown 后，进入可被唤醒的 deepsleep mode。此时如果发生按键唤醒，SDK 会在 user initialization 的时候先快速做一次低电检测，而不是等到 main_loop 中检测。这样处理的原因是为了避免应用上的错误，举例说明如下：

如果低电报警时 LED 闪烁已经提示了产品使用者，然后进入 deepsleep 又被唤醒，从 main_loop 的处理来看，需要至少 500mS 的时间才会去做低电检测。在 500mS 之前，slave 的广播包已经发很久了，很可能会跟 master 已经连接上了。这样的话，就出现已经低电报警的设备又继续工作的 bug 了。

因为这个原因，SDK 必须在 user initialization 的时候就提前做低电检测，必须在这一步就阻止发生上面的情况。所以在 user initialization 的时候，添加低电检测：

```
if(analog_read(DEEP_ANA_REG2) == LOW_BATT_FLG){
    app_battery_power_check(VBAT_ALRAM_THRES_MV + 200); //2.2 V
}
```

根据 DEEP_ANA_REG2 模拟寄存器的值可以判断是否低电报警 shutdown 被唤醒的情况，此时进行快速低电检测，并且将之前的 2000mV 报警电压提高到 2200mV（称为恢复电压）。提高 200mV 的原因是：

低压检测会有一些误差，无法保证测量结果的准确性和一致性。比如误差在 20mV，可能第一次检测到的电压是 1990mV 进入 shutdown 模式，然后唤醒后在 user initialization 的时候再次检测到的电压值是 2005mV。如果还是以 2000mV 为报警电压的话，还是无法阻止上面描述的 bug。

所以需要在 shutdown 模式唤醒后的快速低电检测时,将报警电压稍微调高一些,调高的幅度比低电检测的最大误差稍大。

只有当某次低电检测发现电压低于 2000mV 进入 shutdown 模式后,才会出现恢复电压 2200mV,所以 user 不用担心这个 2200mV 会对实际电压 2V~2.2V 的产品误报低压。产品使用者看到低压报警指示后,进行充电或更换电池后,满足恢复电压的要求,产品恢复正常使用。

5.2.2.4 低电检测 debug 模式

“8258_ble_remote” Demo code 中留出了两个 debug 相关的宏,提供给 user 进行 debug。

```
#define DBG_ADC_ON_RF_PKT          0
#define DBG_ADC_SAMPLE_DAT        0
```

只有在 debug 的时候才可能打开上面两个“宏”。

“DBG_ADC_ON_RF_PKT”打开后,会在广播包和连接状态时按键值的数据包上显示 ADC 采样结果的信息。注意:此时广播包和按键数据被修改,所以只能用于 debug。

“DBG_ADC_SAMPLE_DAT”打开后,可以将 ADC 采样的中间结果存储在 Sram 上。

5.2.3 低电检测和 Amic Audio

参考低电检测单独使用模式中详细的介绍,对于需要实现 Amic Audio 的产品,只要做好低电检测和 Amic Audio 的切换即可。

按照低电检测单独使用的方式,程序开始运行后,默认低电检测先开启。当 Amic Audio 被触发时,做以下两件事:

1) 关闭低电检测

调用 `battery_set_detect_enable(0)`,告知低电检测模块 ADC 资源已被抢占。

2) Amic Audio ADC 初始化

由于使用 ADC 的方式和低电检测不一样,需要对 ADC 重新进行初始化。具体方法参考本文档“Audio”章节的介绍。

Amic Audio 结束时,调用 `battery_set_detect_enable(1)`,告知低电检测模块 ADC 资源已经被释放。此时低电检测需要重新初始化 ADC 模块,然后开始进行低电检测。

如果是低电检测和其他非 Amic Audio 的 ADC 任务同时存在，其他 ADC 任务的_{处理可模仿 Amic Audio 的处理流程。}

如果是低电检测、Amic Audio、其他 ADC 任务共 3 种任务同时存在，user 可根据“ADC 电路需要切换使用”的原则，参考低电检测和 Amic Audio 切换实现的方法，去自行实现。

Shared under NDA

6 Audio

6.1 Audio 初始化

Audio 的来源可以是 Amic 或 Dmic。

- ✧ Dmic 是直接使用外围 audio 处理的芯片，将数字信号读到 8x5x 上；
- ✧ Amic 需要使用 8x5x 内部的多个模拟电路模块（包括 PGA、ADC、filter 等），对原始的 Audio 信号进行采样后处理，最终转化为数字信号传输到 MCU。

6.1.1 Amic 和低电检测

参考本文档对“低电检测”的介绍可知，Amic Audio 和低电检测使用 ADC 模块时，必须对 ADC 进行切换使用。

同理，如果应用上同时出现 Amic Audio 和其他 ADC 任务，这 2 个任务也需要对 ADC 进行切换使用。如果同时出现 Amic Audio、低电检测、其他 ADC 任务，这 3 个任务也需要对 ADC 进行切换使用。

Telink 上一代 826x 系列 IC 上 Amic 可以在 user initialization 时设置，而 8x5x Amic 需要在 Audio 任务开启时设置，这样才能实现低电检测和 Amic 对 ADC 模块的切换使用。

6.1.2 Amic 初始化设置

参考 SDK demo “8258_ble_remote”语音处理相关 code。

```
void ui_enable_mic (int en)
{
    ui_mic_enable = en;

    gpio_set_output_en (GPIO_AMIC_BIAS, en); //AMIC Bias output
    gpio_write (GPIO_AMIC_BIAS, en);

    if(en){ //audio on
        audio_config_mic_buf ( buffer_mic, TL_MIC_BUFFER_SIZE);
        audio_amic_init(AUDIO_16K);
    }
    else{ //audio off
        adc_power_on_sar_adc(0); //power off sar adc
    }
}
```

```
#if (BATT_CHECK_ENABLE)
    battery_set_detect_enable(!en);
#endif
}
```

上面ui_enable_mic函数中，en=1对应Audio任务的开启，en=0对应Audio任务的结束。

Audio开始时，GPIO_AMIC_BIAS需要输出高电平来驱动Amic；Audio结束后，GPIO_AMIC_BIAS需要关闭，防止这个管脚在sleep mode漏电。

Amic初始化设置为

```
audio_config_mic_buf ( buffer_mic, TL_MIC_BUFFER_SIZE);
audio_amic_init(AUDIO_16K);
```

Audio在工作过程中，使用指定的Dfifo将数据源源不断拷贝到Sram上。audio_config_mic_buf用于配置该Dfifo在Sram上的起始地址和长度。

Dfifo 的配置在 ui_enable_mic 函数中处理，相当于每次 Audio 开始都要重新做一遍，原因是 Dfifo 控制 register 在 suspend 时会掉电丢失。

Audio任务结束的时候，必须关闭SAR ADC，防止在suspend时漏电：

```
adc_power_on_sar_adc(0);
```

由于 Amic 和低电检测需要切换使用 ADC 模块，在 ui_enable_mic 函数里添加 battery_set_detect_enable(!en)，用于关闭和开启低电检测，请参考本文档低电检测部分的介绍。

语音任务的执行放在 main_loop 的 UI entry 部分。

```
#if (BLE_AUDIO_ENABLE)
    if(ui_mic_enable){ //audio
        task_audio();
    }
#endif
```

6.1.3 Dmic 初始化设置

待补充。

6.2 Audio 数据处理

6.2.1 Audio 数据量和 RF 传送方法

Amic 采样出来的原声数据是 pcm 格式的，采用 pcm to adpcm 算法将其压缩为 adpcm 格式，压缩率为 25%，用以减低 BLE RF 数据量，master 端收到的 adpcm 格式数据解压缩还原为 pcm 格式。

Amic 采样率为 16K*16bit，每秒钟 16K 个 sample，每 ms 16 个 sample，即每 ms 16*16bit = 32byte。

每 15.5ms，产生 15.5*16=248 个 sample 共 496 bytes 的原声数据。对这 496 bytes 进行 pcm 到 adpcm 转换：1/4 压缩为 124 bytes，同时加上 4 个 bytes 的头信息，得到 128 个 bytes 的数据。

128 bytes 的数据，在 L2cap 层上发送给 master，会分成 5 个 packet 上进行，因为每个包最大长度是 27，第一个包必须带 7 个 bytes 的 l2cap 的说明信息：

l2caplen: 2 bytes, chanid: 2 bytes, opcode: 1 byte, AttHandle: 2 bytes

下图所示为空中抓到的 RF 数据，可以看到第一个包中有 7 个额外的信息，后面紧跟 20 bytes 的 audio 数据，后面的包 27 bytes 全是 audio 数据。第一个包只放 20 bytes 的 audio 数据，后面 4 个包由于是分包，不需要再带 l2cap 说明信息，每个包可以放 27 个 bytes：20 + 27*4 = 128 bytes。

Data Type	Data Header	CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length 1 1 1 0 0	0x8FEFDC	-38	OK
Data Type	Data Header	L2CAP Header		
L2CAP-S	LLID NESN SN MD PDU-Length 2 0 1 1 27	L2CAP-Length ChanId 0x0083 0x0004	Opcode AttHandle AttValue 0x1B 0x002B 3F 03 07 7C A9 BE 13 65 21 43 51 B1 43 22 14 10 C3 40 22 25	
Data Type	Data Header	CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length 1 0 0 0 0	0x8FE4A9	-38	OK
Data Type	Data Header	Generic L2CAP Payload	CRC	RSSI (dBm) FCS
L2CAP-C	LLID NESN SN MD PDU-Length 1 1 0 1 27	80 94 38 33 73 08 11 2A 32 61 94 11 99 53 41 92 99 A9 E9 81 8B 1C 9A 09 AA D1 8B	0x132A61	-38 OK
Data Type	Data Header	CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length 1 1 1 0 0	0x8FEFDC	-38	OK
Data Type	Data Header	Generic L2CAP Payload	CRC	RSSI (dBm) FCS
L2CAP-C	LLID NESN SN MD PDU-Length 1 0 1 1 27	AC BB C9 B9 C9 8A 8D CB 4B 9C 09 AB 99 29 0F AB 0B 1A 0F 04 15 21 53 30 C8 17 90	0x36A693	-38 OK
Data Type	Data Header	CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length 1 0 0 0 0	0x8FE4A9	-38	OK
Data Type	Data Header	Generic L2CAP Payload	CRC	RSSI (dBm) FCS
L2CAP-C	LLID NESN SN MD PDU-Length 1 1 0 1 27	19 09 89 89 89 AB 08 8A 50 E9 19 8A B8 D0 08 AA F9 88 C1 A0 9A B1 1B 9A 9E CA C9	0x441600	-38 OK
Data Type	Data Header	CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length 1 1 1 0 0	0x8FEFDC	-38	OK
Data Type	Data Header	Generic L2CAP Payload	CRC	RSSI (dBm) FCS
L2CAP-C	LLID NESN SN MD PDU-Length 1 0 1 0 27	E0 81 0B 09 1A DB B3 99 A9 D2 99 0F B9 91 C9 B0 B1 CB B2 E1 1A AA 13 0F 3A 47 32	0xF05ACB	-38 OK
Data Type	Data Header	CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length 1 0 0 0 0	0x8FE4A9	-38	OK
Data Type	Data Header	CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length 1 1 0 0 0	0x8FE27A	-38	OK

图 6-1 audio 数据抓包

结合前面 BLE 模块 ATT & GATT 部分对 Exchange MTU size 部分的说明可知，这里 audio 数据属于 128 byte 的长包在 slave 端进行了分包处理，如果希望 peer device(对端设备)收到这些包后能够重新拼装成功，就一定要通过 Exchange MTU size 确定对方 peer device 的最大 ClientRxMTU，只有当 ClientRxMTU 大于等于 128 时，slave 端的这个 128byte 长包才能被 peer device 正确处理。

所以当 audio 任务开启，需要发送 128 byte 长包时，会调用 blc_att_requestMtuSizeExchange 进行 Exchange MTU size。

```
void voice_press_proc(void)
{
    key_voice_press = 0;
    ui_enable_mic (1);

    if(ui_mtu_size_exchange_req &&
        blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){

        ui_mtu_size_exchange_req = 0;
        blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, 0x009e);
    }
}
```

推荐的做法是：通过 blc_att_registerMtuSizeExchangeCb 注册 MTU size Exchange 的 callback，在 callback 里去判断 peer device 的 ClientRxMTU 是否大于等于 128。由于一般 master 设备的 ClientRxMTU 都比 128 大，SDK 并没有通过 callback 判断实际 ClientRxMTU。

audio service 在 Attribute Table 中的描述为：

```
// 0033 - 0036 MIC
{0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characterUUID), (u8*)(&PROP_READ_NOTIFY), 0}, //prop
{0,ATT_PERMISSIONS_READ,16,sizeof(my_MicData),(u8*)(&my_MicUUID), (u8*)(&my_MicData), 0}, //value
{0,ATT_PERMISSIONS_RDWR,2,sizeof(micDataCCC),(u8*)(&clientCharacterCfgUUID), (u8*)(&micDataCCC), 0}, //value
{0,ATT_PERMISSIONS_RDWR,2,sizeof(my_MicName),(u8*)(&userdesc UUID), (u8*)(&my_MicName), 0},
```

图 6-2 MIC service in Attribute Table

图上第 2 个 Attribute 是负责 audio 数据传送的 Attribute。在这个 Attribute 上使用 Handle Value Notification 将数据发送给 master。master 收到 Handle Value Notification 后，可以将连续 5 个分包对应的 Attribute Value 数据进行拼包成为 128 个 bytes，对其进行解压缩还原为 pcm 格式的 audio 数据。

6.2.2 Audio 数据压缩

根据以上说明，在 app_config.h 中定义相关的宏：

```
#define ADPCM_PACKET_LEN 128
#define TL_MIC_ADPCM_UNIT_SIZE 248
#define TL_MIC_BUFFER_SIZE 992
```

每一笔 adpcm 压缩数据量为 248 个 sample，496 个 bytes。由于 Amic 一直在进行采样并把处理过的 pcm 格式数据放到事先设置好的 buffer 上（buffer_mic）。将这个 buffer 设置为能够存储 2 笔压缩数据，也就是 496 个 sample，以实现数据的缓冲和保存。使用 16K 采样，496 个 sample 为 992 个 bytes，TL_MIC_BUFFER_SIZE 为 992。

定义 buffer_mic:

```
s16 buffer_mic[TL_MIC_BUFFER_SIZE>>1]; //496 sample,992 bytes
config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE);
```

硬件控制数据填充到 buffer_mic 的机制说明如下:

Amic 采样的数据按照 16K 的速度匀速放入从 buffer_mic 地址开始的内存，向后移动，并且最大长度为 992，一旦到最大长度，重新回到 buffer_mic 地址开始放数据。这个过程不对内存上的数据进行任何是否已经被读走的判断，直接覆盖老的数据。向 RAM 放数据的过程中，维护一个写指针用于记录当前最新的 audio 数据已经到 RAM 的哪个地址了。

软件上定义一个 buffer_mic_enc，用来存放压缩后的 128 个 bytes 的数据，将 buffer_mic_enc 的 number 设为 4，最多可以缓存 4 笔压缩后的数据。

```
int buffer_mic_enc[BUFFER_PACKET_SIZE];
```

BUFFER_PACKET_SIZE 为 128，由于 int 占 4 个 bytes，等同于 128*4 个 signed char。

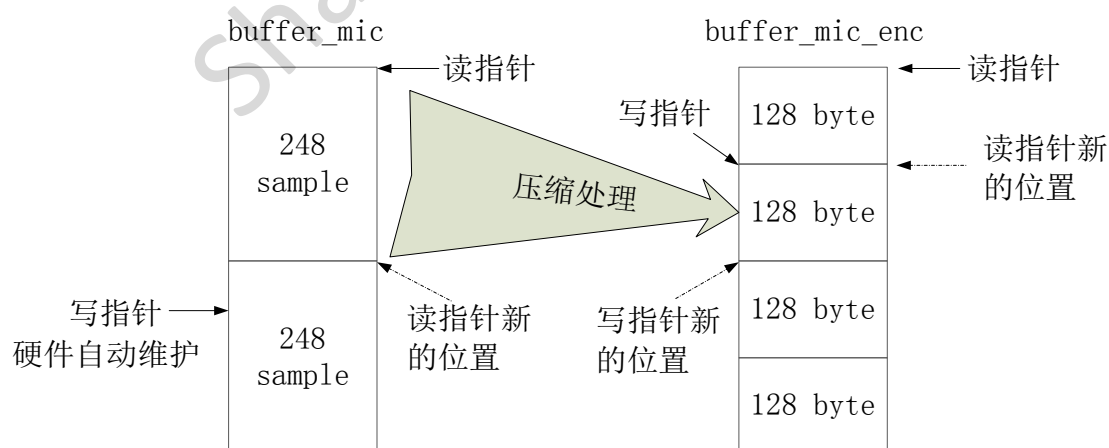


图 6-3 数据压缩处理

上图所示为数据压缩处理的方法。

`buffer_mic` 自动维护一个硬件写指针，同时在软件上维护一个读指针。

当软件上检测到写指针与读指针中间的差值已经满足 248 个 sample 时，就开始调用压缩处理函数，从读指针开始取出 248 个 sample 的数据压缩为 128 个 bytes，同时将读指针移到图上新位置，表示最新的未读的数据从新的位置开始。如此循环往复，不断检测是否有足够的 248 个 sample 的数据，只要达到这个数据量，就开始做压缩处理。

由于 248 个 sample 的产生时间为 15.5ms，需要保证程序至少 15.5 ms 才查询一次。由前面的介绍可知，程序在每个 `main_loop` 只执行一次 `task_audio`，那么 `main_loop` 的时间必须小于 15.5 ms 才能保证音频数据不丢。在连接状态，`main_loop` 的时间等于 `connection interval`，所以有音频任务的应用，`connection interval` 一定要小于 15.5 ms。实际应用中推荐 10 mS。

`buffer_mic_enc` 在软件上维护写指针和读指针，当 248 sample 数据压缩为 128 bytes 后，将这个 128 个 bytes 拷贝到写指针开始的地方，拷贝完之后检查一下这个 buffer 是否溢出。若溢出，将最老的一笔数据放弃（将读指针向后移动 128 bytes 即可）。

将压缩后的数据拷贝到 BLE RF 数据发送缓冲区的方法为：

检查 `buffer_mic_enc` 是否为非空（写指针和读指针相等时空，不等为非空）。若非空，从读指针开始的地址拿出 128 bytes 拷贝到 BLE RF 数据发送缓冲区，然后将读指针移到图上所示新的位置。

Audio 数据压缩处理对应的函数为 `proc_mic_encoder`，请参考 SDK 中的实现。

6.3 压缩与解压缩算法

压缩算法调用的函数为：

```
void mic_to_adpcm_split (signed short *ps, int len, signed short *pds, int start);
```

`ps` 指向压缩前数据内存的首地址，对应图 6-3 中 `buffer_mic` 的读指针的位置。

`len` 取 `TL_MIC_ADPCM_UNIT_SIZE`（248），表示 248 个 sample。

`pds` 指向压缩后数据内存的首地址，对应图 6-3 中 `buffer_mic_enc` 写指针的位置。

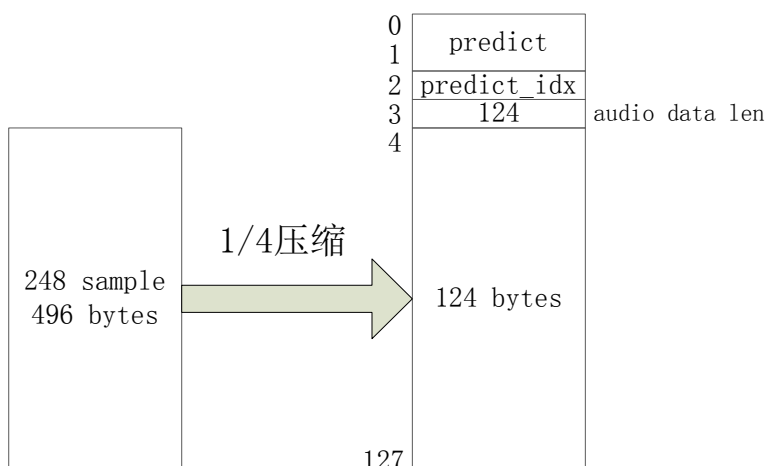


图 6-4 压缩算法对应数据

如上图所示：压缩后的数据内存的前两个 bytes 存 predict；第三个 byte 存 predict_idx；第 4 个 byte 为当前 adpcm 格式的 audio 数据的有效数据量，也就是 124；后面的 124 个 bytes 由 496 bytes 的原声数据 1/4 压缩而来，压缩的具体算法不介绍，只要能根据这个方法对应解压缩即可。

解压缩算法对应函数为：

```
void adpcm_to_pcm (signed short *ps, signed short *pd, int len);
```

ps 指向需要解压缩的数据内存开始的地址，也就是指向 128 bytes 的 adpcm 格式数据，这个地址需要 user 定义 buffer，从 BLE RF 收到的 128 bytes 数据拷贝到该 buffer；

pd 指向解压缩后还原的 496 bytes pcm 格式音频数据内存开始的地址，这个需要 user 定义 buffer，播放声音时直接从该 buffer 拿数据；

len 与压缩端长度一样，为 248。

解压缩的时候，对应图 6-4 所示，从前两个 bytes 读到的数据为 predict，第三个 byte 为 predict_idx，第 4 个为 audio 数据有效长度 124，后面的 124 bytes 对应转换为 496bytes pcm 格式 audio 数据。

7 OTA

为了实现 8x5x BLE slave 的 OTA 功能，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（需要在 APP 里实现 OTA），也可以使用 Telink 的 BLE master kma dongle，下面以 Telink 的 BLE master kma dongle 作为 ota master 来详细介绍 OTA。

8x5x 支持 Flash 多地址启动：除了 flash 地址 0x00000，还支持从 flash 地址 0x20000、0x40000 读取 firmware 运行。本文档 0x20000 为例来介绍 OTA。

7.1 Flash 架构设计和 OTA 流程

7.1.1 FLASH 存储架构

使用启动地址 0x20000 时，SDK 编译出来的 firmware size 应不大于 128K，即 flash 的 0~0x20000 之间的区域存储 firmware，但是由于一些特殊的原因，如果使用启动地址为 0 和 0x20000 交替 OTA 升级，其 firmware size 不得超过 124K；如果超过 124K 必须使用启动地址 0 和 0x40000 交替升级，此时最大 firmware size 不得超过 252K。

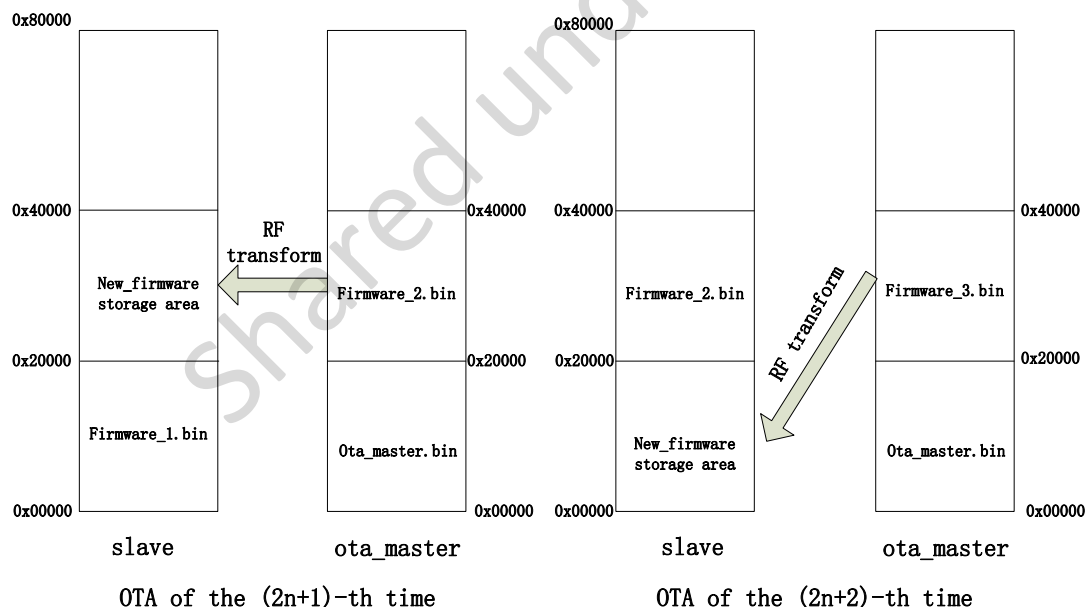


图 7-1 Flash 存储结构

- 1) ota master 将新的 firmware2 烧写到 0x20000~0x40000 的区域。
- 2) 第 1 次 OTA:
 - A. slave 上电时从 flash 的 0~0x20000 区域读程序启动，运行 firmware1；
 - B. firmware1 运行，初始化的时候将 0x20000~0x40000 区域清空，该区域将作为新的 firmware 的存储区。

- C. 启动 OTA, master 通过 RF 将 firmware2 空运到 slave 的 0x20000~0x40000 区域。slave reboot (重新启动, 类似一次断电并重新上电)
- 3) 将新的 firmware3 烧写到 ota master 的 0x20000~0x40000 的区域。
- 4) 第 2 次 OTA:
 - A. slave 上电时从 flash 的 0x20000~0x40000 区域读程序启动, 运行 firmware2;
 - B. firmware2 运行, 初始化的时候将 0~0x20000 区域清空, 该区域将作为新的 firmware 的存储区。
 - C. 启动 OTA, master 通过 RF 将 firmware3 空运到 slave 的 0~0x20000 区域。slave reboot。
- 5) 后面的 OTA 过程重复上面 (1) ~ (4) 过程, 可理解为 (2) 代表第 $2n+1$ 次 OTA, (3) 代表第 $2n+2$ 次 OTA。

7.1.2 OTA 更新流程

以上面的 FLASH 存储结构为基础, 详细说明 OTA 程序更新的过程。

首先介绍一下多地址启动机制 (只介绍前两个启动地址 0x00000 和 0x20000): MCU 上电后, 默认从 0 地址启动, 首先去读 flash 0x8 的内容, 若该值为 0x4b, 则从 0 地址开始搬移代码到 RAM, 并且之后所有的取指都是从 0 地址开始, 即取指地址 = 0+PC 指针的值; 若 0x8 的值不为 0x4b, MCU 直接去读 0x20008 的值, 若该值为 0x4b, 则 MCU 从 0x20000 开始搬代码到 RAM, 并且之后所有的取指都是从 0x20000 地址开始, 即取指地址 = 0x20000+PC 指针的值。

所以只要修改 0x8 和 0x20008 标志位的值, 即可指定 MCU 执行 FLASH 哪部分的代码。

SDK 上某一次 ($2n+1$ 或 $2n+2$) 上电及 OTA 过程为:

- 1) MCU 上电, 通过读 0x8 和 0x20008 的值和 0x4b 作比较, 确定启动地址, 然后从对应的地址启动并执行代码。此功能由 MCU 硬件自动完成。
- 2) 程序初始化过程中, 读 MCU 硬件寄存器判断 MCU 刚才是从哪个地址启动:
若从 0 启动, 将 ota_program_offset 设为 0x20000, 并将 0x20000 区域非 0xff 的内容全部擦除为 0xff, 表示下一次 OTA 获得的新 firmware 会存入 0x20000 开始的区域;
若从 0x20000 启动, 将 ota_program_offset 设为 0x0, 并将 0x0 区域非 0xff 的内容全部擦除为 0xff, 表示下一次 OTA 获得的新 firmware 会存入 0x0 开始的区域。

- 3) Slave 程序正常运行，OTA master 上电运行，并与 slave 建立 BLE 连接。
- 4) 在 OTA master 端 UI 触发进入 OTA 模式（可以是按键、PC 工具写内存等）。OTA master 进入 OTA 模式后，首先需要获取 slave OTA service 数据 Attribute 的 Attribute Handle 的值（可以 slave 事先和 master 约定好，也可以通过 read_by_type 获取这个 handle 值）。
- 5) OTA master 获取了 slave OTA service 数据 Attribute 的 Attribute Handle 值后，获取当前 slave FLASH 程序的 firmware 版本号。
注：获取版本号并对比这一步 user 自己决定是否需要做和该怎么做。
- 6) master 确定要做 OTA 更新后，先发一个 OTA_start 命令通知 slave 进入 OTA 模式。
- 7) Slave 收到 OTA start 命令后，进入 OTA 模式，等待 master 发 OTA 数据。
- 8) Master 从 0x20000 开始的区域读预先存储好的 firmware，不间断的向 slave 发送 OTA 数据，直至整个 firmware 都发过去。
- 9) Slave 接收 OTA 数据，向 ota_program_offset 开始的区域存储。
- 10) master 端发完所有的 OTA 数据后，检查这些数据 slave 是否都正确收到（调用底层 BLE 的相关函数判断 link layer 的数据是否都被正确 ack）。
- 11) master 确定所有的 OTA 数据都被 slave 正确收到后，发送一个 OTA_END 命令。
- 12) Slave 收到 OTA_END 命令，将新 firmware 区域偏移地址 8（即 ota_program_offset+8）写为 0x4b，将之前老的 firmware 存储区域偏移地址 8 的地方写为 0x00，表示下一次程序启动后将从新的区域搬代码执行。
- 13) 将 slave reboot，新的 firmware 生效。
- 14) 在整个 OTA 更新过程中，slave 会不断检查是否有错包和丢包，同时也会不断检查是否超时（OTA 开始的时候启动一个计时），一旦有错包、丢包或超

时，slave 会认为更新失败，程序 reboot，使用之前老的 firmware。

以上流程 slave 端相关操作在 SDK 上已经实现，user 不需要添加任何东西，master 端需要额外的程序设计，后面会详细介绍。

7.1.3 修改 Firmware size 和 boot address

API `bls_ota_set_fwSize_and_fwBootAddr` 同时支持两个功能：修改最大 firmware size 和启动地址。这个启动地址指的是 OTA 设计中除了 0 地址外另一个存储 New_firmware 的地址（只能是 0x20000 或 0x40000）。

SDK 中默认的最大 firmware size 为 124K，对应的启动地址为 0x20000（但是由于一些特殊的原因，启动地址为 0x20000，firmware size 不得大于 124K）。这两个值和前文的描述一致。

```
//firmware_size_k must be 4k aligned  
void bls_ota_set_fwSize_and_fwBootAddr(int firmware_size_k,  
                                         int boot_addr);
```

firmware_size_k 的设置一定要 4K byte 对齐，比如 size 为 97K 时需要设为 100K。

这个 API 只能在 main 函数中 `cpu_wakeup_init` 之前调用，否则无效。原因是 `cpu_sleep_wakeup` 函数中需要根据 firmware_size_k 和 boot_addr 的值做一些设置。

如果最大 firmware_size 发生变化，超过了 124K，此时需要将启动地址挪到 0x40000（最大不得超过 252K）。比如最大 firmware size 可能到 200K，设置如下：

```
bls_ota_set_fwSize_and_fwBootAddr(200, 0x40000);
```

除了修改启动地址，这个 API 还可以对 flash 空间的使用进行优化。

默认的 Firmware size 为 128K，那么 0x00000 ~ 0x40000 只能用来存放 Firmware。如果 Firmware 不是特别大（比如不超过 60K），那么实际 0x00000 ~ 0x20000 和 0x20000 ~ 0x40000 中有两段比较大的空间会浪费掉。当 user 希望将这些浪费掉的空间作为自己的数据存储空间时，可以按照如下设置：

```
bls_ota_set_fwSize_and_fwBootAddr(60, 0x20000);
```

按照上面的设定之后，Flash 的 0x00000 ~ 0x0F000 和 0x20000 ~ 0x2F000 两段 60K 的空间作为 Firmware 存放地址，而 0x0F000 ~ 0x20000 和 0x2F000 ~ 0x40000 两段 68K 的空间都可以作为 user 的数据存储空间了。

7.2 OTA 模式 RF 数据处理

7.2.1 Slave 端 Attribute Table 中 OTA 的处理

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;

{4,ATT_PERMISSIONS_READ, 2,16,(u8*)(&my_primaryServiceUUID),
    (u8*)(&my_OtaServiceUUID), 0},
{0,ATT_PERMISSIONS_READ, 2, 1,(u8*)(&my_characterUUID),
    (u8*)(&PROP_READ_WRITE_NORSP), 0},          //prop
{0,ATT_PERMISSIONS_RDWR,16,sizeof(my_OtaData),(u8*)(&my_OtaUUID),
    (&my_OtaData), &otaWrite, &otaRead},        //value
{0,ATT_PERMISSIONS_READ, 2,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 3 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，可以直接在 master 端定义 Attribute Handle 的值。

7.2.2 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

图 7-2 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。

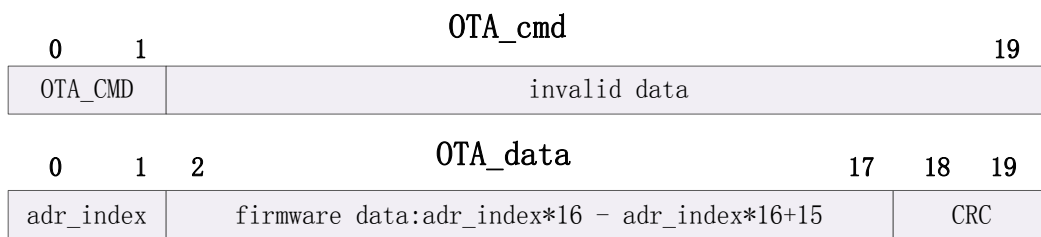


图 7-3 OTA 命令和数据的格式

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 之内时，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数据。

7.2.3 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 不检查每一个 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。

Telink 所有 BLE SDK 的 OTA data 的 UUID 都是 16bytes，且永远都是下面这个值：

```
#define TELINK_SPP_DATA_OTA
    {0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,
    0x03,0x02,0x01,0x00}    //!< TELINK_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。

Data Header						L2CAP Header		ATT_Read_By_Type_Req																		
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	StartingHandle	EndingHandle	AttType															
L2CAP-S	2	0	0	0	25	0x0015	0x0004	0x08	0x0001	0xFFFF	12 2B 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00															
Data Header						CRC	RSSI (dBm)	FCS																		
Empty PDU	1	1	1	0	0	0x8FEFDC	0	OK																		
Data Header						L2CAP Header		ATT_Read_By_Type_Rsp				CRC	RSSI (dBm)	FCS												
Data Type	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	Length	AttData	CRC	RSSI (dBm)	FCS													
L2CAP-S	2	0	1	0	9	0x0005	0x0004	0x09	0x03	31 00 00	0x79893F	0	OK													

图 7-4 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 自己选择是否要做。该 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 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）。

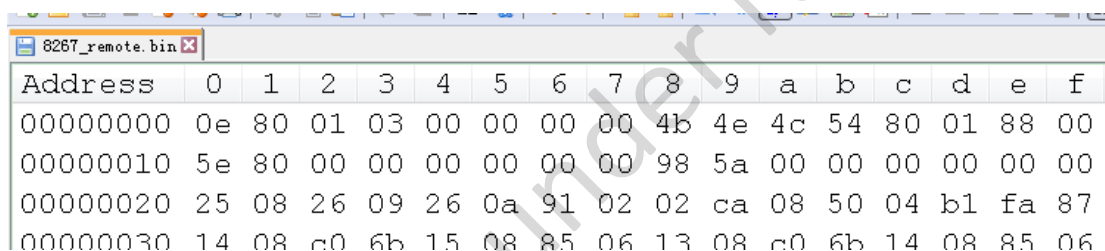
- 4) 启动 OTA 开始的一个计时，后面要不断检测该计时是否超过 15 秒（这只是个参考时间，实际根据 user 测试的正常 OTA 需要多少时间后再做评估）。

如果超过 15 秒认为 OTA 超时失败，因为 slave 端收到 OTA 数据后会校验 CRC，一旦 CRC 错误或者出现其他错误（如烧写 flash 错误），就会认为 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。

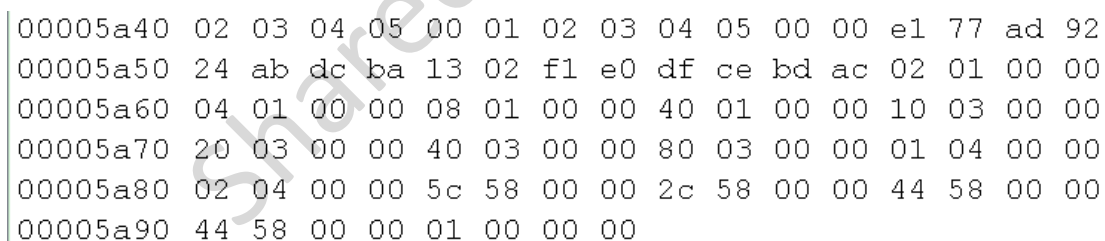
这个 size 是由我们的编译器实现的，假设 firmware 的 size 为 20k = 0x5000，那么 firmware 的 0x18~0x1b 的值为 0x00005000，所以在 0x20018~0x2001b 可以读到 firmware 的大小。

如下图所示的 8258_remote.bin，0x18~0x1b 内容为 0x00005a98，所以大小为 0x5a98 = 23192 bytes，从 0x0000 到 0x5a97。



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

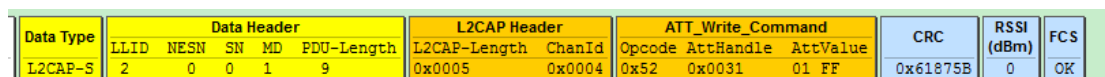
图 7-5 firmware 示例--开头部分



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 7-6 firmware 示例--结尾部分

- 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	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue		(dBm)	
L2CAP-S	2	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF	0x61875B	0	OK

图 7-7 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 8258_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
L2CAP-S	LLID	NESH	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x61875B	0	OK
	2	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF			
Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESH	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x8FE27A	0	OK
	1	1	0	0	0								
Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESH	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x8FE90F	0	OK
	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			
Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESH	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x8FE27A	0	OK
	1	0	1	0	0								
Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESH	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x8FE27A	0	OK
	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			
Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
Empty PDU	LLID	NESH	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x8FE27A	0	OK
	1	1	0	0	0								
Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
L2CAP-S	LLID	NESH	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue	0x8FE27A	0	OK
	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			

Data Type		Data Header					L2CAP Header		ATT_Write_Command											
L2CAP-S		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue									
		2	0	0	1	27	0x0017	0x0004	0x52	0x0031	A9 05 44 58 00 00 01 00 00 00 FF FF FF FF FF FF FF FF 44 47									
Data Type		Data Header					CRC		RSSI (dBm)	FCS										
Empty PDU		LLID	NESN	SN	MD	PDU-Length	0x8FE27A		0	OK										
		1	1	0	0	0														
Data Type		Data Header					L2CAP Header		ATT_Write_Command				CRC	RSSI (dBm)	FCS					
L2CAP-S		LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue									
		2	1	1	0	13	0x0009	0x0004	0x52	0x0031	02 FF A9 05 56 FA			0x013FFA	0	OK				

图 7-8 master OTA 数据

- 8) firmware 数据发送完毕后，检查 BLE link layer 的数据是否已经完全发送出去（因为只有当 link layer 的数据被 slave ack 了，才会认为该数据发送成功）。若完全发送出去，master 发送一个 ota_end 命令，通知 slave 所有数据已发送完毕。

OTA end 的 packet 有效字节设为 6 个，前两个为 0xff02，中间的两个 bytes 为新的 firmware 最大的 adr_index 值（这个是为了让 slave 端再次确认没有丢掉最后一条或几条 OTA 数据），最后两个 bytes 为中间最大的 adr_index 值的取反，相当于一个简单的校验。OTA end 不需要 CRC 校验。

以上图所示的 8258_remote.bin 为例，最大的 adr_index 为 0x05a9，其取反值为 0xfa56，最终的 OTA end 包如上图所示。

- 9) 检查 master 端 link layer TX fifo 是否为空。
若为空，说明之前所有的数据和命令都已成功发送出去，即 master 端的 OTA 任务已经全部完成。

CRC_16 计算函数见本文档后面的附录。

7.2.4 slave 端 RF receive 处理方法

按照前面所述，Slave 端在 OTA Attribute 中直接调用 otaWrite 和 otaRead 即可，master 端发送过来的 write command 命令，BLE 协议栈会自动解析并最终调用到 otaWrite 函数进行处理。

在 otaWrite 函数里对 packet 20 byte 的数据进行解析，首先判断是 OTA CMD 还是 OTA data，对 OTA cmd 进行相应的响应，对 OTA 数据进行 CRC 校验并烧写到 flash 对应位置。

slave 端 OTA 相关的操作为:

- 1) 收到 OTA version 命令(前两 bytes 为 0xff00: OTA_FIRMWARE_VERSION 命令):

master 要求获得 slave firmware 版本号, 该 BLE SDK 收到这个命令时, 不做处理, 只是根据 user 是否注册了收到 version 的回调函数, 判断是否触发回调函数。

在 ble_ll_ota.h 中看到注册该回调函数的接口为:

```
typedef void (*ota_versionCb_t) (void);  
void bls_ota_registerVersionReqCb(ota_versionCb_t cb);
```

- 2) 收到 OTA start 命令 (前两 bytes 为 0xff01):

此时 slave 进入 OTA 模式。

若用户使用 bls_ota_registerStartCmdCb 函数注册了 OTA start 时的回调函数, 则执行此函数, 这个函数的目的是让用户在进入 OTA 模式后, 修改一些参数状态等, 比如将 PM 关掉 (使得 OTA 数据传输更加稳定)。

另外 slave 启动并维护一个 slave_adr_index, 初值为-1, 记录最近一次正确 OTA data 的 adr_index, 用于判断整个 OTA 过程中是否有丢包。一旦丢包, 认为 OTA 失败, 退出 OTA, MCU 重启, master 端由于收不到 slave 的 ack 包, 也会由于 OTA 任务超时使得软件发现 OTA 失败。

注册 OTA start 的回调函数:

```
typedef void (*ota_startCb_t) (void);  
void bls_ota_registerStartCmdCb(ota_startCb_t cb);
```

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 的校验。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 再次启动后运行 New_firmware；

若 OTA 失败，会将错误的新程序擦掉后重新启动，还是运行 Old_firmware。

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` 后，将 MCU 用 `while(1)` 停住，来了解当前是何种原因导致的 OTA 失败。

User 可以参考 SDK demo “8258_ble_remote”中 debug OTA 的 code，其中使用了 LED 闪烁效果来判断 OTA 是否成功。

Shared under NDA

8 按键扫描

Telink 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。

文档以 SDK demo “8258_ble_remote”中的按键扫描为例来说明。

8.1 键盘矩阵

如下图所示，这是一个 5*6 的 Key matrix（键盘矩阵），最多支持 30 个按键。Row0 ~ Row4 是 5 个 drive pin（驱动管脚），用来输出驱动电平；CoL0 ~ CoL5 是 6 个 scan pin（扫描管脚），用来扫描当前列上是否有按键被按下。

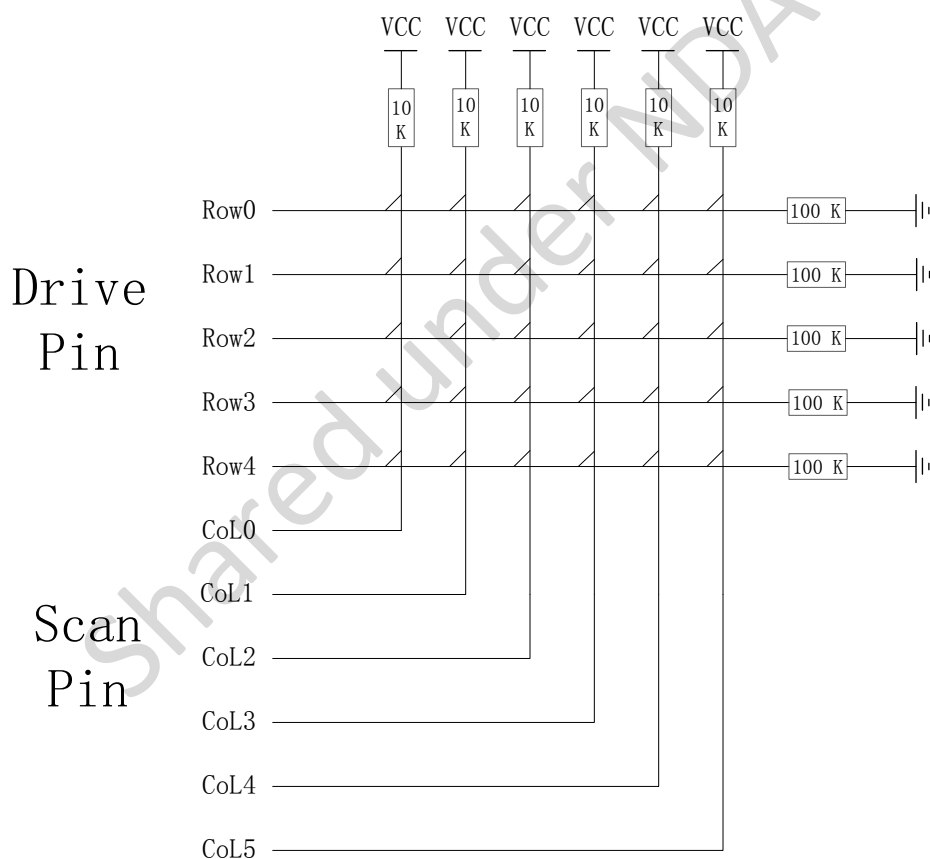


图 8-1 行列式键盘结构

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 ~ Row4 为 PD5、PD2、PD4、PD6、PD7。CoL0 ~ CoL5 为 PC5、PA0、PB2、PA4、PA3、PD3。

定义 drive pin 数组和 scan pin 数组：

```
#define KB_DRIVE_PINS {GPIO_PD5, GPIO_PD2, GPIO_PD4, GPIO_PD6,
                      GPIO_PD7}
#define KB_SCAN_PINS {GPIO_PC5, GPIO_PA0, GPIO_PB2, GPIO_PA4,
                      GPIO_PA3, GPIO_PD3}
```

keyscan 使用的上下拉电阻都使用 GPIO 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_PD5 MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PD2 MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PD4 MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PD6 MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PD7 MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PC5 MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PA0 MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PB2 MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PA4 MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PA3 MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PD3 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 PC5_INPUT_ENABLE 1
#define PA0_INPUT_ENABLE 1
#define PB2_INPUT_ENABLE 1
#define PA4_INPUT_ENABLE 1
#define PA3_INPUT_ENABLE 1
#define PD3_INPUT_ENABLE 1
```

当 MCU 进入 sleep mode 时，需要设置 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 PD5_INPUT_ENABLE    1
#define PD2_INPUT_ENABLE    1
#define PD4_INPUT_ENABLE    1
#define PD6_INPUT_ENABLE    1
#define PD7_INPUT_ENABLE    1
```

8.2 Keyscan、keymap

8.2.1 Keyscan

按照上面的配置完成后，在 main_loop 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 main_loop 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 main_loop 中调用的，根据 BLE 时序可知，main_loop 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，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所有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所有按键都是松开的。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……

8.2.2 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）。

所有按键对应的 keycode 在 app_config.h 中定义：

```
#define KB_MAP_NORMAL { \
    VK_B,   CR_POWER,  VK_NONE,   VK_C,           CR_HOME,   \
    VOICE,  VK_NONE,   VK_NONE,   CR_VOL_UP,    CR_VOL_DN, \
    VK_2,   VK_RIGHT,  CR_VOL_DN, VK_3,           VK_1,      \
    VK_5,   VK_ENTER,  CR_VOL_UP, VK_6,           VK_4,      \
    VK_8,   VK_DOWN,   VK_UP,     VK_9,           VK_7,      \
    VK_0,   CR_BACK,   VK_LEFT,   CR_VOL_MUTE,  CR_MENU,   \
}
```

这个 keymap 的格式和 5*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 VK_B。

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。

- 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; //clear keycode[0]
int det_key = kb_scan_key (0, 1);

if (det_key)
{
    key_not_released = 1;

    u8 key0 = kb_event.keycode[0];
    if (kb_event.cnt == 2) //two key press, do not process
    {
    }
    else if (kb_event.cnt == 1)
    {
        key_buf[2] = key0;
    }
}
```

```

        //send key press
        bls_att_pushNotifyData (HID_NORMAL_KB_REPORT_INPUT_DP_H,
                                key_buf, 8);
    }
    else //key release
    {
        key_not_released = 0;
        key_buf[2] = 0;
        //send key release
        bls_att_pushNotifyData (HID_NORMAL_KB_REPORT_INPUT_DP_H,
                                key_buf, 8);
    }
}

```

8.3 Keyscan flow

调用 `kb_scan_key` 时，一个最基本的 `keyscan` 的流程如下：

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` 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 debounce` 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 ~ ROW4 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[5]（可看出最多支持 40 列）来存储，pressed_matrix[0] 的 bit0~bit5 标记 Row0 上 Col0 ~ Col5 是否有按键，.....，pressed_matrix[4] 的 bit0~bit5 标记 Row4 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，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。

8.4 Deepsleep 唤醒快速扫键 (wake_up 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 main_loop 中先发广播包, 等到连上后才能将按键的值发送给 ble master。

该 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 main_loop 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;
        }
    }
    #endif
}
```

```

        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` 变量拷贝到缓存变量 `kb_event_cache`。

`main_loop` 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
}

```

`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，实现了快速扫描键值的缓存处理。

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。

8.5 Repeat Key 处理

以上介绍的最基本的 **keyscan** 只有在按键状态变化时产生一个变化事件，通过 **kb_event** 来读取当前 **key** 值，就无法实现 **repeat key** 功能：一个按键一直按着时，需要定时发送一个按键值。

加入 **repeat key** 处理，在 **app_config.h** 中配置相关的宏如下。

KB_REPEAT_KEY_ENABLE 用来打开或关闭 **repeat key** 功能，默认这个功能是关闭的。

```
//repeat key
#define KB_REPEAT_KEY_ENABLE 0
#define KB_REPEAT_KEY_INTERVAL_MS 200
#define KB_REPEAT_KEY_NUM 1
#define KB_MAP_REPEAT {VK_1, }
```

1) KB_REPEAT_KEY_ENABLE

用来打开或关闭 **repeat key** 功能。

若要实现 **repeat key**，首先要将 **KB_REPEAT_KEY_ENABLE** 设为 1。

2) KB_REPEAT_KEY_INTERVAL_MS

定义 **repeat key** 的 **repeat** 时间。

若设为 200ms，表示当一个键被一直按着时，每过 200 ms，**kb_key_scan** 会返回一个变化，并且在 **kb_event** 里面给出当前这个按键状态。

3) KB_REPEAT_KEY_NUM 和 KB_MAP_REPEAT

定义当前需要 **repeat** 的键值。

KB_REPEAT_KEY_NUM 定义数量；**KB_MAP_REPEAT** 定义一个 **map**，给出所有需要 **repeat** 的 **keycode**，注意这个 **map** 中 **keycode** 一定要是 **KB_MAP_NORMAL** 中的值。

应用举例：

如下所示的一个 6*6 的矩阵按键，四个宏定义实现的功能是：8 个按键 UP、DOWN、LEFT、RIGHT、V+、V-、CHN+、CHN-支持 repeat，每 100ms repeat 一次；其他的按键都不支持 repeat key。

```
#define KB_MAP_NORMAL { \
    {VK_POWER, VK_LOW_BATT, VK_TV_PLUS, VK_TV_MINUS, VK_IN_OUTPUT, VK_VOL_UP, }, \
    {VK_VOICE_SEARCH, VK_PROGRAM, VK_RETURN, VK_HOME, VK_MENU, VK_EXIT, }, \
    {VK_UP, VK_CH_UP, VK_W_MUTE, VK_LEFT, VK_CONFIRM, VK_RIGHT, }, \
    {VK_VOL_DN, VK_DOWN, VK_CH_DN, VK_FAST_BACKWARD, VK_PLAY_PAUSE, VK_1, }, \
    {VK_2, VK_3, VK_4, VK_5, VK_6, VK_7, }, \
    {VK_9, VKPAD_ASTERIX, VK_0, VK_NUMBER, VK_W_SRCH, VK_8, }, \
}

#define KB_REPEAT_KEY_ENABLE 1
#define KB_REPEAT_KEY_INTERVAL_MS 100
#define KB_REPEAT_KEY_NUM 8
#define KB_MAP_REPEAT { VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, \
    VK_VOL_UP, VK_VOL_DN, VK_CH_UP, VK_CH_DN, }
```

repeat key 代码的实现这里不介绍，user 自行理解，只要在工程上搜索以上四个宏就可以找到所有代码了。

8.6 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，比如家里的茶杯/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[5]，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

    .....
}
```

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，释放按键可以唤醒整个 MCU。

Shared under NDA

9 LED 管理

9.1 LED 任务相关调用函数

该 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);
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 main_loop 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。

9.2 LED 任务的配置和管理

9.2.1 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重复多少次。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);
```

9.2.2 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 $10\text{ms} * 50 = 500\text{ms}$ ），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 8258 ble remote 工程里代码 LED 相关的处理方法。

10 Blt 软件定时器（Software Timer）

为了方便 user 一些简单的定时器任务，Telink 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 main_loop 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 main_loop 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
```

10.1 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);
}
```

10.2 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 main_loop 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 有如下两种情况: 0 表示在 main_loop 中查询进入, 1 表示发生了 timer 提前唤醒时进入该函数。

```
#define MAIN_LOOP_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, 所以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) ){
    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(0x40000, 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_16M_SYS_TIMER_CLK_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，则以该返回值作为新的定时周期（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_16M_SYS_TIMER_CLK_1MS){
    bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t, 1);
}
else{
    bls_pm_setAppWakeupLowPower(0, 0); //disable
}
```

10.3 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) 返回值小于 0，则该任务执行后被自动删除。可以使用这个特性来控制定时器执行的次数。
- 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_16M_SYS_TIMER_CLK_1US;
        blt_timer.timer[blt_timer.currentNum].t = now + blt_timer.timer[blt_timer.currentNum].interval;
        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。

10.4 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);
```

10.5 Demo

blt soft timer 的 Demo code 请参考 8258 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;
}
```

```
}
```

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。

11 IR

11.1 PWM Driver

请参考《8258 Datasheet》第 8 章 PWM 的介绍。

PWM 相关的硬件配置很简单，基本都是直接操作寄存器来实现。操作寄存器的 API 都定义在 `pwm.h` 中（不需要 `c` 文件），使用 `static inline function` 来实现，这样可以提高运行效率，也节省 `code size`。

11.1.1 PWM id 和管脚

8x5x 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,
    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。比如 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。

这 12 路 PWM 对应的 IC 管脚如下所示：

PWMx	Pin	PWMx_n	Pin
PWM0	PA2/PC1/PC2/PD5	PWM0_N	PA0/PB3/PC4/PD5
PWM1	PA3/PC3	PWM1_N	PC1/PD3
PWM2	PA4/PC4	PWM2_N	PD4
PWM3	PB0/PD2	PWM3_N	PC5
PWM4	PB1/PB4	PWM4_N	PC0/PC6
PWM5	PB2/PB5	PWM5_N	PC7

使用 `void gpio_set_func(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func)` 来设置对应管脚的 PWM 功能。

`pin` 用实际 PWM 波形对应的 GPIO;

`func` 必须选择 `GPIO_FuncTypeDef` 定义里的 `AS_PWM0 ~ AS_PWM5_N`，根据上面表中 `gpio` 实际的 `pwm` 功能对应选择。如下所示。

```
typedef enum{
    .....
    AS_PWM0      = 20,
    AS_PWM1      = 21,
    AS_PWM2      = 22,
    AS_PWM3      = 23,
    AS_PWM4      = 24,
    AS_PWM5      = 25,
    AS_PWM0_N    = 26,
    AS_PWM1_N    = 27,
    AS_PWM2_N    = 28,
    AS_PWM3_N    = 29,
    AS_PWM4_N    = 30,
    AS_PWM5_N    = 31,
}GPIO_FuncTypeDef;
```

比如要使用 PA2 作为 PWM0 来用:

```
gpio_set_func(GPIO_PA2, AS_PWM0);
```

11.1.2 PWM 时钟

使用 API `void pwm_set_clk(int system_clock_hz, int pwm_clk)` 来设置 PWM 的 clock (时钟)。

`system_clock_hz` 填当前的系统时钟 `CLOCK_SYS_CLOCK_HZ` (这个宏是在 `app_config.h` 中定义的)；

`pwm_clk` 为要设置的 clock，`system_clock_hz` 一定要能被 `pwm_clk` 整除，才能分频得到这个正确的 PWM clock。

为了让 PWM 波形精准，PWM clock 需要尽量大，最大值不能超过系统时钟，推荐设置 `pwm_clk` 为 `CLOCK_SYS_CLOCK_HZ`，即：

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_HZ);
```

比如当前系统时钟 `CLOCK_SYS_CLOCK_HZ` 为 16000000，上面设置的 PWM clock 时钟和系统时钟相等，为 16M。

如果想要 PWM 时钟为 8M，可按如下设置，不管系统时钟如何变化（CLOCK_SYS_CLOCK_HZ 为 16000000、24000000 或 32000000），PWM clock 都是 8M。

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, 8000000);
```

11.1.3 PWM 周期（cycle）和占空比（duty）

PWM 波形的基本单位是 PWM 信号帧（Signal Frame）。

配置一个 PWM Signal Frame 需要设置 cycle 和 cmp 两个参数。

`void pwm_set_cycle(pwm_id id, unsigned short cycle_tick)` 用于设置 PWM cycle，单位为 PWM clock 的个数。

`void pwm_set_cmp(pwm_id id, unsigned short cmp_tick)` 用于设置 PWM cmp，单位为 PWM clock 的个数。

下面 API 是将上面两个 API 合二为一，可以提高设置的效率。

```
void pwm_set_cycle_and_duty(pwm_id id, unsigned short cycle_tick,
                           unsigned short cmp_tick)
```

那么对于一个 PWM signal frame，计算 PWM 占空比(PWM duty):

$$\text{PWM duty} = \text{PWM cmp} / \text{PWM cycle}$$

下图相当于 `pwm_set_cycle_and_duty(PWM0_ID, 5, 2)` 得到的结果。一个 Signal Frame 的 cycle 为 5 个 PWM clock，高电平时间为 2 个 PWM clock，PWM duty 为 40%。

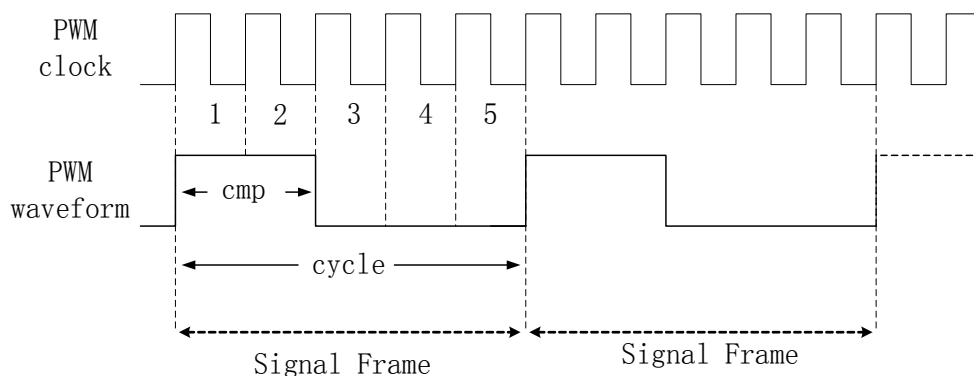


图 11-1 PWM cycle & duty

对于 PWM0 ~ PWM5，硬件上会自动将高电平放前面，低电平放后面。如果想要低电平在前的话，有以下几种方法：

- 1) 使用对应的 PWM0_N ~ PWM5_N，和 PWM0 ~ PWM5 相反。
- 2) 使用 API `static inline void pwm_revert (pwm_id id)` 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 50% 的 PWM0 一个 frame 方法为：

```
pwm_set_cycle(PWM0_ID, 16000)
pwm_set_cmp (PWM0_ID, 8000)
或
pwm_set_cycle_and_duty(PWM0_ID, 16000, 8000);
```

11.1.4 PWM 波形取反

API `void pwm_revert (pwm_id id)` 用于将 PWM0 ~ PWM5 波形取反。

API `void pwm_n_revert (pwm_id id)` 用于将 PWM0_N ~ PWM5_N 波形取反。

11.1.5 PWM 开启和停止

下面两个 API 用于开启和停止某一路 PWM：

```
void pwm_start (pwm_id id);
void pwm_stop (pwm_id id);
```

11.1.6 PWM 模式

PWM 共 5 种模式：Normal mode(也称 Continuous mode)、Counting mode、IR mode、IR FIFO mode 和 IR DMA FIFO mode。如下定义：

```
typedef enum{
    PWM_NORMAL_MODE    = 0x00,
    PWM_COUNT_MODE     = 0x01,
    PWM_IR_MODE         = 0x03,
    PWM_IR_FIFO_MODE    = 0x07,
    PWM_IR_DMA_FIFO_MODE = 0x0F,
}pwm_mode;
```

设定 PWM 模式的 API 为:

```
void pwm_set_mode(pwm_id id, pwm_mode mode)
```

PWM0 具有 Normal mode、Counting mode、IR mode、IR FIFO mode 和 IR DMA FIFO mode 所有的 5 种模式；而 PWM1 ~ PWM5 都只有 normal mode。

也就是说，只有 PWM0 具有除 normal mode 外的 4 种特殊的模式。

PWM 模式的详细说明请参考《8258 Datasheet》的 8.5 节。

11.1.7 PWM 脉冲数 (pulse number)

API `void pwm_set_pulse_num(pwm_id id, unsigned short pulse_num)` 用于设置指定的 PWM 波形中 Signal Frame 的个数。这里“pulse”和 Signal Frames 是同一个概念。

Normal mode (Continuous mode) 不受 Signal Frame 个数的限制，所以此 API 对 Normal mode 无意义。只有其他 4 种特殊的 mode 才有可能用到这个 API。

11.1.8 PWM 中断

先介绍一下 Telink MCU 中断的一些基本概念。

中断“status”是由硬件的特定动作（也就是中断动作）产生的状态标记位，它不依赖于软件中任何设定，不管中断“mask”是否打开，只要中断动作发生，“status”必然置位（值为 1）。一般通过对“status”写 1 的操作可以清除这个“status”（值回到 0）。

定义中断响应的概念：中断响应，表示硬件中断动作产生后，软件上程序指针 PC 跳转到 irq_handler 去进行相关处理。

如果 user 希望中断被响应，就需要确保当前中断对应的所有“mask”全部被打开。“mask”可能有多个，多个“mask”之间是逻辑“与”的关系。只有当所有“mask”都打开了，中断“status”才会触发最终的中断响应，MCU 跳转到 irq_handler 去执行；只要有一个“mask”没打开，中断“status”的产生都无法触发中断响应。

PWM driver 中 user 可能需要用到的中断如下所示（code 在文件 register_8258.h 中）。除此之外的中断是用不上的，user 不用关注。

```
#define reg_pwm_irq_mask      REG_ADDR8(0x7b0)
#define reg_pwm_irq_sta      REG_ADDR8(0x7b1)
enum{
    FLD_IRQ_PWM0_PNUM =          BIT(0),
    FLD_IRQ_PWM0_IR_DMA_FIFO_DONE = 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),
};
```

上面列出来的中断有 8 个，对应 core_7b0/7b1 的 BIT<0:7>。core_7b0 是这个 8 个中断的“mask”，core_7b1 是 8 个中断的“status”。

将 8 个中断“status”分为 3 类。参考下图来说明，假设 PWM0 工作在 IR mode，PWM Signal Frame 的占空比为 50%，每个 IR task 对应的脉冲个数 pulse number（或 Signal Frame number）为 3。

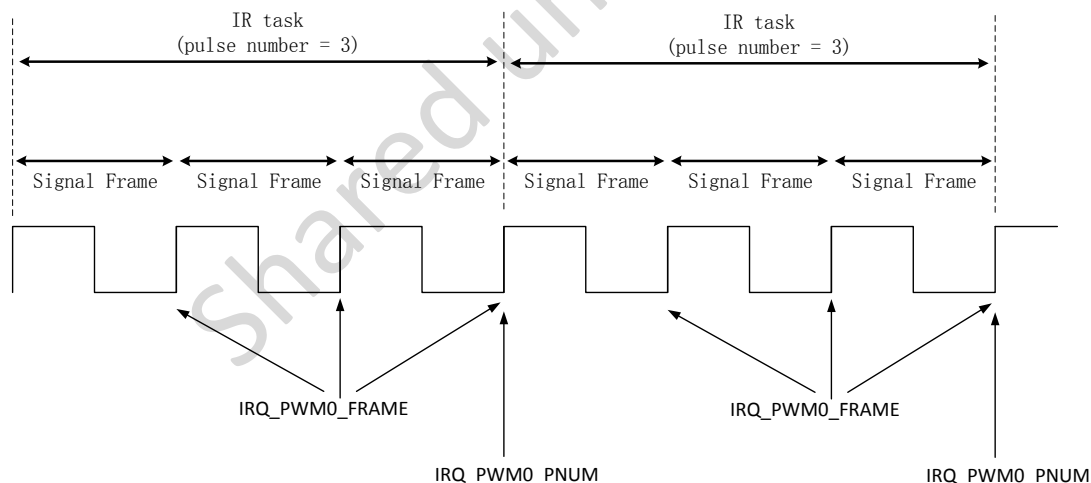


图 11-2 PWM interrupt

1) 第一类：IRQ_PWMn_FRAME(n=0,1,2,3,4,5)

中断源后面的 6 个，是同一种中断，分别在 PWM0~PWM5 上产生。

如图所示，IRQ_PWMn_FRAME 是每个 PWM Signal Frame 结束后都会产生的中断。PWM 5 种模式里，Signal Frame 都是 PWM 波形的基本单位。所以不管哪种 PWM 模式，IRQ_PWMn_FRAME 都会出现。

2) 第二类: IRQ_PWM0_PNUM

IRQ_PWM0_PNUM 是一组 Signal Frame (个数由 API `pwm_set_pulse_num` 决定) 结束时产生的中断。图中每 3 个 Signal Frame 后产生一个 IRQ_PWM0_PNUM。

PWM 的 Counting mode、IR mode 会用到 API `pwm_set_pulse_num`。所以, 只有 PWM0 的 Counting mode、IR mode 才会产生 IRQ_PWM0_PNUM。

3) 第三类: IRQ_PWM0_IR_DMA_FIFO_DONE

PWM0 工作在 IR DMA FIFO mode 时, 当 DMA 上所有配置好的 PWM 波形全部发送完毕后, 触发 IRQ_PWM0_IR_DMA_FIFO_DONE。

上面说到所有相关中断“mask”同时被打开时, 才会触发中断响应, 对于 PWM 中断, 以 FLD_IRQ_PWM0_PNUM 为例, 共有 3 层“mask”需要打开:

1) FLD_IRQ_PWM0_PNUM 的“mask”

即 `core_7b0` 对应的“mask”, 打开方法为:

```
reg_pwm_irq_mask |= FLD_IRQ_PWM0_PNUM;
```

一般在打开 mask 之前先清掉之前的 status, 防止中断响应被误触发:

```
reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;
```

2) MCU 系统中断上的 PWM “mask”

即 `core_640` 的 BIT<14>。

```
#define reg_irq_mask          REG_ADDR32(0x640)
enum{
    .....
    FLD_IRQ_SW_PWM_EN =      BIT(14),  //irq_software | irq_pwm
    .....
};
```

打开方法为:

```
reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
```

3) MCU 总中断使能位, 即 `irq_enable()`。

11.1.9 PWM phase

```
void pwm_set_phase(pwm_id id, unsigned short phase)
```

用于设置 PWM 开始前的延时时间。

phase 为延时时间，单位为 PWM clock 的个数。一般不需要延时，设为 0。

11.1.10 API for IR DMA FIFO mode

下面介绍 IR DMA FIFO mode 专用的几个 API。User 可结合 SDK 上 PWM demo code 来理解。

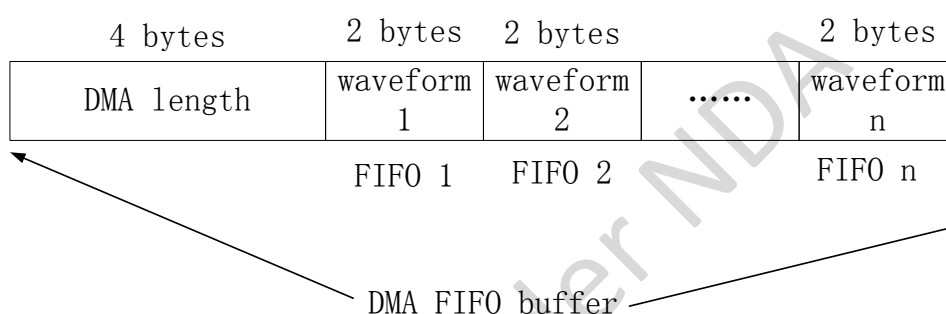


图 11-3 DMA FIFO buffer for IR DMA FIFO mode

如上图所示，DMA FIFO buffer 是在 Sram 上定义的一块数据，前 4byte DMA length 是 FIFO 占用 byte 的数量，图上 $DMA\ length = n * 2$ 。

FIFO 的数量为 n，每个 FIFO 2 个 bytes 可以用来表示一个 PWM 波形 (waveform)。8x5x 支持最大的 n 为 256。

上面的 DMA data buffer 生效后，PWM 硬件模块会按照时间顺序将 PWM waveform 1、waveform 2、waveform n 连续发送出去。

发送结束后 PWM 自动结束，并触发中断 IRQ_PWM0_IR_DMA_FIFO_DONE。

11.1.10.1 DMA FIFO 的配置

每个 DMA FIFO 上，使用 2bytes (16 bits)来配置一个 PWM waveform。调用下面 API 返回 2 bytes 的 DMA FIFO 数据。

```
unsigned short pwm_config_dma_fifo_waveform(int carrier_en,
      Pwm0Pulse_SelectDef pulse, unsigned short pulse_num);
```

该 API 有三个参数：“carrier_en”、“pulse”和“pulse_num”，配置出来的 PWM waveform 是“pulse_num”个 PWM Signal Frame 的集合。

参考《8258 Datasheet》PWM waveform 的配置格式，BIT(15)决定当前 PWM waveform 的基本单位 Signal Frame 的格式，对应 API 中的“carrier_en”：

- ✧ “carrier_en”为 1 时，Signal Frame 中高脉冲是生效的；
- ✧ “carrier_en”为 0 时，Signal Frame 是全 0 数据，高脉冲无效。

“pulse_num”为当前 PWM waveform 中 Signal Frame 的数量。

“pulse”可选择如下两个定义。

```
typedef enum{
    PWM0_PULSE_NORMAL = 0,
    PWM0_PULSE_SHADOW = BIT(14),
}Pwm0Pulse_SelectDef;
```

“pulse”为 PWM0_PULSE_NORMAL 时，Signal Frame 来自 API pwm_set_cycle_and_duty 的配置；“pulse”为 PWM0_PULSE_SHADOW 时，Signal Frame 来自 PWM shadow mode 的配置。

PWM shadow mode 是为了增加一组 Signal Frame 的配置，从而为 IR DMA FIFO mode 的 PWM waveform 配置增加更多的灵活性。配置 API 如下，方法和 API pwm_set_cycle_and_duty 完全一致。

```
void pwm_set_pwm0_shadow_cycle_and_duty(unsigned short cycle_tick,
                                         unsigned short cmp_tick);
```

11.1.10.2 设置 DMA FIFO buffer

DMA FIFO buffer 的配置完成后，调用下面 API，将此 buffer 的首地址设置到 DMA 模块。

```
void pwm_set_dma_address(void * pdat);
```

11.1.10.3 IR DMA FIFO mode 的开启与停止

DMA FIFO buffer 准备好之后，调用下面 API 开启 PWM waveform 的发送：

```
void pwm_start_dma_ir_sending(void);
```

DMA FIFO buffer 上所有 PWM waveform 发送结束后，PWM 模块会自动停止。如果需要在此之前手动停止 PWM 模块，调用下面 API：

```
void pwm_stop_dma_ir_sending(void);
```

11.2 IR Demo

请参考 SDK demo “8258_ble_remote” 上 IR 的 code，将 app_config.h 中的宏 “REMOTE_IR_ENABLE”。

11.2.1 PWM 模式的选择

IR 的发送需要在特定的时间切换 PWM 的输出，对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 main_loop 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。

所以 IR 不能使用 PWM Normal mode。

Telink 上一代 826x 系列 BLE SDK 上，使用 PWM IR mode 来实现 IR，user 可以参考“826x BLE SDK handbook”的介绍。

PWM IR mode 的缺陷是 826x IR mode 最多支持 2 个 IR fifo 的预存数据，如果出现 PWM Signal Frame 时间非常短（比 irq_handler 里面 BLE 的中断处理时间更短）时，可能出现 PWM waveform 被延时的情况。所以 IR mode 也存在出错的风险。

IR DMA FIFO mode 是 8x5x 上新增的模式（826x 系列 IC 不支持这种模式）。IR DMA FIFO mode 由于存储 FIFO 可以定义在 Sram 上，使得 FIFO 的数量显著提高，可以有效解决上面描述的 PWM IR mode 的缺陷。

IR DMA FIFO mode 可以提前将多组 PWM waveform 存储在 Sram 上，一旦启动 DMA 就不需要软件的参与，这样即可以节省软件频繁处理的时间，又可以防止中断系统上多个中断同时响应导致的 PWM waveform 被延迟。

由于只有 PWM0 具有 IR DMA FIFO mode，IR 的实现只能通过 PWM0 来实现，硬件电路设计时 IR 的控制 GPIO 必须选择 PWM0（或者 PWM0_n）对应的管脚。

11.2.2 Demo IR 协议

SDK 上 demo IR 协议如下图所示。

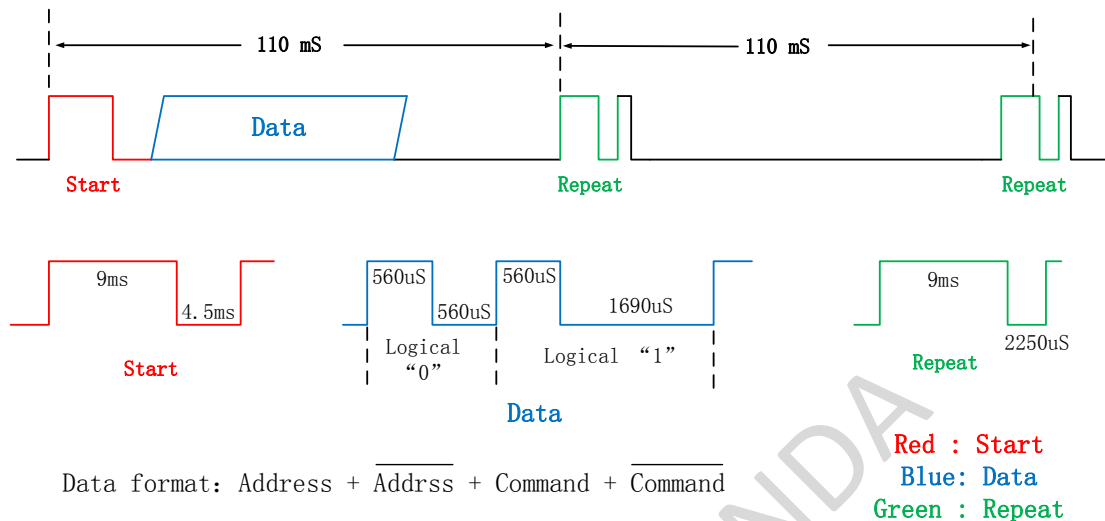


图 11-4 demo IR 协议

11.2.3 IR 时序设计

首先需要设计 IR 时序。根据 demo IR 的协议，结合 IR DMA FIFO mode 的特点，我们得到下面图示的 IR 时序。

IR DMA FIFO mode 一个完整的任务定义为 FifoTask。先按照 SDK demo 中对 IR repeat 信号的处理采用“add repeat one by one（逐个添加 repeat）”的方式介绍，即下面的宏定义为 1。

```
#define ADD_REPEAT_ONE_BY_ONE 1
```

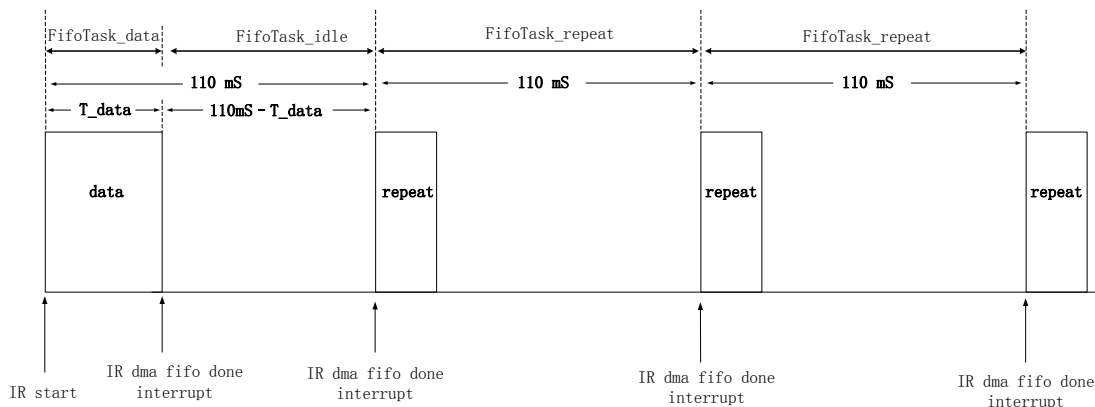


图 11-5 IR timing 1

当一个按键按下触发 IR 发送开始后，将 IR 拆分成图上的 FifoTask。

- 1) IR start 后，运行 FifoTask_data，发送有效数据。FifoTask_data 持续时间为 T_{data} ，由于数据的不确定性， T_{data} 也不确定。FifoTask_data 结束后，触发中断 IRQ_PWM0_IR_DMA_FIFO_DONE。
- 2) 在 IRQ_PWM0_IR_DMA_FIFO_DONE 中断函数里，开启 FifoTask_idle，这个阶段发送无载波信号，时间为 $110ms - T_{data}$ 。FifoTask_idle 存在的意义是为了控制第 1 个 FifoTask_repeat 时间点正好在 IR start 后的 110ms。FifoTask_idle 结束后，触发中断 IRQ_PWM0_IR_DMA_FIFO_DONE。
- 3) 在 IRQ_PWM0_IR_DMA_FIFO_DONE 中断函数里，开启第 1 个 FifoTask_repeat。每个 FifoTask_repeat 的持续时间都是 110ms，只要在其对应的 IRQ_PWM0_IR_DMA_FIFO_DONE 中断函数继续添加下一个 FifoTask_repeat，就可以控制 IR repeat 信号的持续发送。
- 4) IR stop 的时间点是不确定的，取决于按键 release 的时间。应用层检测到按键 release 后，在确保 FifoTask_data 正确完成的前提下，手动停止 IR DMA FIFO mode 即可结束 IR 的发送。

对上面时序设计进行一些优化。优化的步骤包括：

- 1) 由于 FifoTask_repeat 是固定的时序，而 IR DMA FIFO mode 中 Dma fifo 数量比较大，可以将多个 110ms 的 FifoTask_repeat 合为 1 个 $FifoTask_repeat * n$ ，这样可以减少软件中处理 IRQ_PWM0_IR_DMA_FIFO_DONE 的次数。对应 Demo 中宏“ADD_REPEAT_ONE_BY_ONE”为 0 的处理，此时 Demo 使用了 5 个 IR repeat signal 合成一个 $FifoTask_repeat * 5$ 。User 可以根据 Dma Fifo 的使用情况，继续做优化。
- 2) 在步骤 1 优化基础上，将 FifoTask_idle 和第一个 $FifoTask_repeat * n$ 合一起，组成 $FifoTask_idle_repeat * n$ 。

优化后的 IR 时序如下图所示：

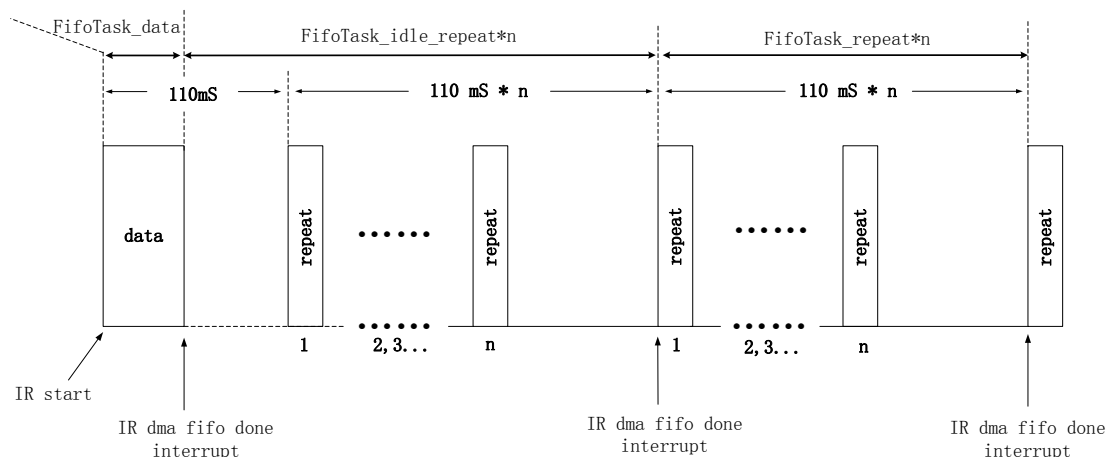


图 11-6 IR timing 2

根据上面 IR 时序设计，软件流程上对应如下 code:

IR start 时调用 `ir_nec_send` 函数，开启 `FifoTask_data`，后面全部用中断去控制。`FifoTask_data` 结束的中断里，开启 `FifoTask_idle`。`FifoTask_idle` 结束的中断里，开启 `FifoTask_repeat`。在手动停止 IR DMA FIFO mode 之前，`FifoTask_repeat` 连续执行。

```
void ir_nec_send(u8 addr1, u8 addr2, u8 cmd)
{
    //Add FifoTask_data to Dma
    ir_send_ctrl.is_sending = IR_SENDING_DATA;
    ir_send_ctrl.sending_start_time = clock_time();
    pwm_start_dma_ir_sending();
}

void rc_ir_irq_prc(void)
{
    if(reg_pwm_irq_sta & FLD_IRQ_PWM0_IR_DMA_FIFO_DONE)
    {
        reg_pwm_irq_sta = FLD_IRQ_PWM0_IR_DMA_FIFO_DONE;

        if(ir_send_ctrl.repeat_enable){
            if(ir_send_ctrl.is_sending == IR_SENDING_DATA){
                ir_send_ctrl.is_sending = IR_SENDING_REPEAT;
                //Add FifoTask_idle_repeat*n to Dma
                pwm_start_dma_ir_sending();
            }
            else if(ir_send_ctrl.is_sending == IR_SENDING_REPEAT){
                //Add FifoTask_repeat*n to Dma
                pwm_start_dma_ir_sending();
            }
        }
        else{
            ir_send_release();
        }
    }
}
```

11.2.4 IR 初始化

11.2.4.1 rc_ir_init

IR 初始化函数如下，user 请参考 SDK 上 code。

```
void rc_ir_init(void)
```

IR的初始化没有放到user initialization里面进行，而是在IR按键触发时，按如下调用。

```
if(!ir_hw_initialed){
    ir_hw_initialed = 1;
    rc_ir_init();
}
```

因为deepsleep retention mode的存在，MCU可能出现反复从deepsleep retention wake_up的情况，而IR的使用没那么频繁，放在IR按键触发时才初始化可以有效的减少rc_ir_init执行的次数。

IR的初始化分两种类型：

- ✧ 一是对硬件register的设定（如pwm_set_mode），每次deepsleep retention wake_up后必须重新设置。
- ✧ 二是Sram逻辑变量的初始化（如waveform_logic_0_1st）。这些变量如果在“data/bss”段上，必须每次都重新设置；如果在“retention_data”段上，只要设置一次就可以。

SDK demo上将这些变量放在“data/bss”段上了，所以需要每次都重新设置。User在做功耗优化时，可以根据Sram retention area的使用情况，决定是否将它们放到“retention_data”段。

变量ir_hw_initialed必须放在“data/bss”段，不能放在“retention_data”段，以确保每次deepsleep retention wake_up后都要重新执行rc_ir_init函数。

11.2.4.2 IR 硬件配置

Demo code如下。

```
pwm_n_revert(PWM0_ID);
gpio_set_func(GPIO_PB3, AS_PWM0_N);
pwm_set_mode(PWM0_ID, PWM_IR_DMA_FIFO_MODE);
pwm_set_phase(PWM0_ID, 0); //no phase at pwm beginning
pwm_set_cycle_and_duty(PWM0_ID, PWM_CARRIER_CYCLE_TICK,
                        PWM_CARRIER_HIGH_TICK );
```

```
pwm_set_dma_address(&T_dmaData_buf);
reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
reg_pwm_irq_sta = FLD_IRQ_PWM0_IR_DMA_FIFO_DONE;
```

只有 PWM0 支持 ID DMA FIFO mode, 所以选用 PB3 对应的 PWM0_N 来实现。

Demo IR 载波频率为 38K, 周期为 26.3uS, 占空比为 1/3。使用 API `pwm_set_cycle_and_duty` 配置周期和占空比。在 Demo IR 中, 没有出现多种不同载波频率的情况, 这个 38K 的载波足以满足所有 FifoTask 的配置。所以不需要使用 PWM shadow 模式。

DMA FIFO buffer 为 `T_dmaData_buf`。

打开系统中断 mask “FLD_IRQ_SW_PWM_EN”。

清除中断状态“FLD_IRQ_PWM0_IR_DMA_FIFO_DONE”。

11.2.4.3 IR 变量初始化

对应 SDK demo 中变量 `waveform_start_bit_1st`、`waveform_start_bit_2nd` 等。

结合 IR 时序设计的介绍, 需要配置出 `FifoTask_data`、`FifoTask_repeat`。

Start 信号是 9ms 的载波信号+4.5ms 无载波信号, 对应的两个 DMA FIFO 数据的配置调用 `pwm_config_dma_fifo_waveform` 实现如下:

```
//start bit, 9000 us carrier, 4500 us low
waveform_start_bit_1st = pwm_config_dma_fifo_waveform(1,
PWM0_PULSE_NORMAL, 9000 * CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
waveform_start_bit_2nd = pwm_config_dma_fifo_waveform(0,
PWM0_PULSE_NORMAL, 4500 * CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
u16 waveform_stop_bit_2nd;
```

按照同样的方法, 可以得到 stop 信号、repeat 信号、data 逻辑“1”信号、data 逻辑“0”信号的配置。

11.2.5 FifoTask 的配置

11.2.5.1 FifoTask_data

根据 demo IR 的协议, 如果要发送一个 cmd (比如 7), 先发 start 信号 (9ms 载波信号+4.5ms 无载波信号), 然后是 address+ ~address+ cmd + ~cmd。SDK Demo code 中我们取 address 为 0x88。

当发送~cmd 的最后一个 bit 时, logical “0”或 logical “1”的后面都是一段无载波信号。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 的无载波信号时间是 560us 还是

1690us，导致无法识别这是一个 logical “0”还是 logical “1”。

为了解决这个问题，我们在 Data 信号尾巴上添加一个 stop 信号，stop 信号的构成是：560uS 载波信号+500uS 无载波信号。

根据上面的描述，FifoTask_data 主要分为 3 部分：

1. start 信号：9ms 载波信号+4.5ms 无载波信号
2. data 信号：address+ ~address+ cmd + ~cmd
3. stop 信号：自定义的 560uS 载波信号+500uS 无载波信号

根据以上 3 段信号，配置 Dma Fifo buffer 启动 IR 的发送，code 如下：

```

//// set waveform input in sequence /////
T_dmaData_buf.data_num = 0;

//waveform for start bit
T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_start_bit_1st;
T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_start_bit_2nd;

//add data
u32 data = (~cmd)<<24 | cmd<<16 | addr2<<8 | addr1;
for(int i=0;i<32;i++){
    if(data & BIT(i)){
        //waveform for logic_1
        T_dmaData_buf.data[T_dmaData_buf.data_num++] =
            waveform_logic_1_1st;
        T_dmaData_buf.data[T_dmaData_buf.data_num++] =
            waveform_logic_1_2nd;
    }
    else{
        //waveform for logic_0
        T_dmaData_buf.data[T_dmaData_buf.data_num++] =
            waveform_logic_0_1st;
        T_dmaData_buf.data[T_dmaData_buf.data_num++] =
            waveform_logic_0_2nd;
    }
}

//waveform for stop bit
T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_stop_bit_1st;
T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_stop_bit_2nd;

T_dmaData_buf.dma_len = T_dmaData_buf.data_num * 2;

```

.....

```
pwm_start_dma_ir_sending();
```

11.2.5.2 FifoTask_idle

参考 IR 时序设计，FifoTask_idle 的持续时间 $110\text{mS} - T_{\text{data}}$ 。FifoTask_data 开始时记录时间：

```
ir_send_ctrl.sending_start_time = clock_time();
```

那么在 FifoTask_data 结束触发的中断里，计算 FifoTask_idle 的时序时间为：

$110\text{mS} - (\text{clock_time}() - \text{ir_send_ctrl.sending_start_time})$

对应code如下：

```
u32 tick_2_repeat_sysClockTimer16M = 110 * CLOCK_16M_SYS_TIMER_CLK_1MS -
    (clock_time() - ir_send_ctrl.sending_start_time);
u32 tick_2_repeat_sysTimer =
    (tick_2_repeat_sysClockTimer16M * CLOCK_SYS_CLOCK_1US >> 4);
```

这里要注意两个时间单位的切换问题。参考本文档“时钟模块”的介绍可知，软件计时使用的 System Timer 频率是固定的 16M。而 PWM clock 的来源是 system clock，需要考虑当 system clock 频率为非 16M（24M、32M）时候的情况。

FifoTask_idle 不发送 PWM waveform，也可以认为是一直在发送无载波信号。将 API pwm_config_dma_fifo_waveform 中第一个参数 carrier_en 配置为 0 即可实现。

```
waveform_wait_to_repeat = pwm_config_dma_fifo_waveform(0,
    PWM0_PULSE_NORMAL, tick_2_repeat_sysTimer / PWM_CARRIER_CYCLE_TICK);
```

11.2.5.3 FifoTask_repeat

根据 demo IR 的协议，repeat 信号的组成是 9mS 的载波信号+2.25mS 无载波信号。

类似于 FifoTask_data 的处理，需要在 repeat 信号尾巴上添加一小段载波信号作为结束判断标志，时间设为 560uS。

由 IR 时序设计可知，repeat 信号要求持续时间为 110mS，那么 560uS 载波信号之后的无载波信号持续时间为：

$$110\text{ms} - 9\text{ms} - 2.25\text{ms} - 560\mu\text{s} = 99190\mu\text{s}$$

一个完整的 repeat 信号的配置如下 code 所示:

```
//repeat signal first part, 9000 us carrier, 2250 us low
    waveform_repeat_1st = pwm_config_dma_fifo_waveform(1,
PWM0_PULSE_NORMAL, 9000 * CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
    waveform_repeat_2nd = pwm_config_dma_fifo_waveform(0,
PWM0_PULSE_NORMAL, 2250 * CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

//repeat signal second part, 560 us carrier, 99190 us low
    waveform_repeat_3rd = pwm_config_dma_fifo_waveform(1,
PWM0_PULSE_NORMAL, 560 * CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
    waveform_repeat_4th = pwm_config_dma_fifo_waveform(0,
PWM0_PULSE_NORMAL, 99190 *
CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_repeat_1st;
T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_repeat_2nd;
T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_repeat_3rd;
T_dmaData_buf.data[T_dmaData_buf.data_num++] = waveform_repeat_4th;
```

11.2.5.4 FifoTask_repeat*n & FifoTask_idle_repeat*n

以 FifoTask_idle、FifoTask_repeat 为基础, 在 Dma Fifo buffer 上简单的叠加即可实现 FifoTask_repeat*n 和 FifoTask_idle_repeat*n。

11.2.6 应用层判断 IR busy

user 在应用层通过变量“ir_send_ctrl.is_sending”来判断当前 IR 是否在发送数据或 repeat 信号。

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)
{
    bls_pm_setSuspendMask(SUSPEND_DISABLE);
}
```

12 其他模块

12.1 24M 晶体外部电容

参考下图中的 24M 晶体匹配电容的位置 C19/C20。

SDK 默认使用 8x5x 内部电容（即 ana_8a<5:0>对应的 cap）作为 24M 晶振的匹配电容，此时 C19/C20 不用焊接电容。使用该方案的优势是：在 Telink 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。

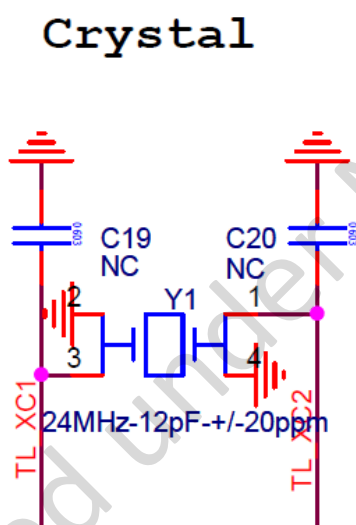


图 12-1 24M 晶体电路

如果需要使用外部焊接电容作为 24M 晶振的匹配电容(C19/C20 焊接电容)，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 会自动处理所有的设置，包括关掉内部匹配电容、不再读取频偏校正值等。

12.2 32k 时钟源选择

SDK 默认使用 MCU 内部 32kRC 振荡电路，简称 32k RC。32k RC 的误差比较大，所以对于 suspend 或者 deep retention 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 3s（当前 SDK 对外部 32k 晶体也做了同样限定），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。

如果用户需要实现更低的连接功耗，包括低功耗睡眠情况下时钟计时更加准确，可以选择使用外部 32k 晶体，简称 32k Pad，目前 SDK 支持该模式。

用户只需要在 main 函数开始的地方（一定要在 cpu_wakeup_init 函数之前）调用下面两个 API 中的一个：

```
void blc_pm_select_internal_32k_crystal(void);
```

```
void blc_pm_select_external_32k_crystal(void);
```

他们分别是选择 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，将其替换成 blc_pm_select_external_32k_crystal 即可。

12.3 PA

如果需要使用 RF PA 的话，请参考 drivers/8258/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
```

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。

12.4 PhyTest

PhyTest 即 PHY test，是指对 BLE controller RF 性能的测试。

详情请参照《Core_v5.0》(Vol 2/Part E/7.8.28~7.8.30)和《Core_v5.0》(Vol 6/Part F “Direct Test Mode”)。

12.4.1 PhyTest API

PhyTest 的源码被封装在 library 文件中，提供相关 API 供 user 使用，请参考 stack/ble/phy/ble_test.h 文件。

```
void    blc_phy_initPhyTest_module(void);

ble_sts_t blc_phy_setPhyTestEnable (u8 en);
bool     blc_phy_isPhyTestEnable(void);

//user for PhyTest 2 wire uart mode
int      phy_test_2_wire_rx_from_uart (void);
int      phy_test_2_wire_tx_to_uart (void);
```

初始化的时候，调用 blc_phy_initPhyTest_module 将 PhyTest 模块设置好。

应用层触发 PhyTest 后，调用 blc_phy_setPhyTestEnable(1)开启 PhyTest 模式。

SDK demo “8258_feature_test”中初始化时就直接触发 phytest 开始；

SDK demo “8258 ble remote”中设置了一个组合按键去触发，只有用户按下这组键才会进入 PhyTest 模式。

PhyTest 是一个特殊的模式，和正常的 BLE 功能是互斥的，一旦进入 PhyTest

模式，广播和连接都不可用了。所以运行正常 BLE 功能时不能触发 PhyTest。

PhyTest 结束后，要么直接重新上电，要么调用 `blc_phy_setPhyTestEnable(0)`，此时 MCU 会自动 reboot。

使用 `blc_phy_isPhyTestEnable` 判断当前 PhyTest 是否被触发了，可以看到代码中使用该 API 来实现低功耗管理，PhyTest 模式不能进入低功耗。

当 PhyTest 使用 uart 两线模式（`PHYTEST_MODE_THROUGH_2_WIRE_UART`）时，初始化的时候按如下设置：

```
blc_register_hci_handler ( phy_test_2_wire_rx_from_uart,
                          phy_test_2_wire_tx_to_uart);
```

`phy_test_2_wire_rx_from_uart` 实现上位机下发的 cmd 的解析和执行，`phy_test_2_wire_tx_to_uart` 实现将相应的结果和数据反馈给上位机。

12.4.2 PhyTest demo

12.4.2.1 Demo1: 8258_feature_test

SDK demo “8258_feature_test”的 `app_config.h` 中，测试模式修改为“`TEST_BLE_PHY`”，如下所示：

```
#define FEATURE_TEST_MODE TEST_BLE_PHY
```

根据物理接口和测试命令格式的不同，PhyTest 可分为三种测试模式，如下所示，“`PHYTEST_MODE_DISABLE`”表示 PhyTest 禁用。

```
#ifndef PHYTEST_MODE_DISABLE
#define PHYTEST_MODE_DISABLE 0
#endif

#ifndef PHYTEST_MODE_THROUGH_2_WIRE_UART
#define PHYTEST_MODE_THROUGH_2_WIRE_UART 1
#endif

#ifndef PHYTEST_MODE_OVER_HCI_WITH_USB
#define PHYTEST_MODE_OVER_HCI_WITH_USB 2
#endif

#ifndef PHYTEST_MODE_OVER_HCI_WITH_UART
#define PHYTEST_MODE_OVER_HCI_WITH_UART 3
```

```
#endif
```

选择 PhyTest 的测试模式：

```
#if (FEATURE_TEST_MODE == TEST_BLE_PHY)
    #define BLE_PHYTEST_MODE    PHYTEST_MODE_THROUGH_2_WIRE_UART
#endif
```

如下定义为 uart 两线模式：

```
#define BLE_PHYTEST_MODE    PHYTEST_MODE_THROUGH_2_WIRE_UART
```

如下定义为 HCI 模式 UART 接口（硬件接口还是 uart）phytest：

```
#define BLE_PHYTEST_MODE    PHYTEST_MODE_OVER_HCI_WITH_UART
```

HCI 模式 USB 接口，暂时不支持。

按照以上定义，编译 8258_feature_test 生成的 bin 文件直接测试可以通过。
user 可研究一下 code 的实现，掌握相关接口的使用。

12.4.2.2 Demo2: 8258_ble_remote

SDK demo “8258_ble_remote”的 app_config.h 中，“BLE_PHYTEST_MODE”默认为“PHYTEST_MODE_DISABLE”，此时 PhyTest 相关代码被屏蔽，功能不生效。

```
#define BLE_PHYTEST_MODE    PHYTEST_MODE_DISABLE
```

如下定义为将开启遥控器应用中的 PhyTest 功能，选择测试模式为 uart 两线模式 PhyTest：

```
#define BLE_PHYTEST_MODE    PHYTEST_MODE_THROUGH_2_WIRE_UART
```

如下定义为将开启遥控器应用中的 PhyTest 功能，选择测试模式为 HCI 模式（硬件接口还是 uart）：

```
#define BLE_PHYTEST_MODE    PHYTEST_MODE_OVER_HCI_WITH_UART
```

按照以上定义，编译 8258_ble_remote 生成的 bin 文件直接测试可以通过。
user 可研究一下 code 的实现，掌握相关接口的使用。

12.4.2.3 PhyTest 参数调整

如果 PhyTest 无法通过，可供调节的参数有 rf packet preamble 的长度和收包带宽的调节等。

对 rf packet preamble 的调整，写 core_402 寄存器即可。8258_feature_test demo 里可以看到初始化的时候直接对 rf packet preamble 进行了调节：

```
blc_phy_initPhyTest_module();
blc_phy_setPhyTestEnable( BLC_PHYTEST_ENABLE );
blc_phy_preamble_length_set(11);
```

8258_ble_remote 里在手动触发 PhyTest 后才调整 rf packet preamble。

```
void app_phytest_init(void)
{
    blc_phy_initPhyTest_module();
    blc_phy_preamble_length_set(11);
    .....
}
```

如果遇到特殊的需要，在以上 demo 的相应位置，直接设置寄存器即可。如对收包带宽的调整，可对模拟 ana_ac 直接赋值。

12.5 EMI

12.5.1 EMI Test

EMI Test 在测试时，需要调用 rfdrv 相关的接口，比如 rf_drv_init()、这些操作接口都封装到 library 中，在 rf_drv.h 中可以看到 API 声明。

EMI Test 有四种测试模式：carrier only 模式（单载波模式）、continue 模式（载波上有数据的发送模式，连续发送）、RX 模式、三种 TX burst 模式（发送的数据包 payload 类型不同）。如下定义所示：

```
Struct test_list_sate_list[] = {
    {0x01,emicarrieronly},//单载波模式
    {0x02,emi_con_prbs9}, //tx continue mode
    {0x03,emirx}, //rx mode
    {0x04,emitxprbs9}, //tx burst
    {0x05,emitx55}, //tx burst
    {0x06,emitx0f}, //tx burst
};
```

12.5.1.1 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 (RF_ModeTypeDef rf_mode);
```

其中参数 `rf_mode` 用来选择 rf 模式，但是在 8258 ble SDK 中暂只支持 `RF_MODE_BLE_1M`。

- 2) 设置完 rf 初始化后，需调用 `app_emi_init()`函数，这个函数会初始化上位机接口命令。

```
write_reg32(0x408,0x29417671 );//rf access code
write_reg8(0x840005,tx_cnt);//tx_cnt 初始化为0
write_reg8(0x840006,run);//run 命令 1: 开始测试项, 0: 结束测试项
write_reg8(0x840007,cmd_now);//cmd:测试项设置
write_reg8(0x840008,power_level);//power_level: 发送power初始化
write_reg8(0x840009,chn);//chn: RF channel 初始化
write_reg8(0x84000a,mode);//mode: RF 模式初始化,
//在BLE SDK中暂只支持BLE 1M模式
write_reg8(0x840004,0); // 4bytes RSSI统计平均值初始化为0
write_reg32(0x84000c,0); // 4bytes rx packet 统计接收个数初始化为0
```

- 3) 在 `main_loop` 中调用 `app_rf_emi_test_start()`，用来轮询测试项。

12.5.1.2 Power level（功率）和 Channel（频点）

在测试时，可以通过配置 `rf power level` 和 `rf channel` 来设置发包的功率以及发包的 `channel`。

RF Power: 可以根据 `RF_PowerTypeDef rf_power_Level_list[60]`设置不同的 `power` 值。

RF Channel: 设置的频率值等于 $(2400+chn)$ MHz。

$$(0 \leq chn \leq 100)$$

其中，设置 `power level` 时，需要注意一下，其发射功率以实际值为准，因为不同的板子或者不同的天线匹配值输出的 `power` 会略有差异。用户可以通过调用下面的 2 个函数来实现 `power` 设置：

```
1. static void rf_set_power_level_index_singleton (RF_PowerTypeDef
level); //单载波下和连续发包模式下调节 Power level
```

```
2. void rf_set_power_level_index (RF_PowerTypeDef level); //tx burst  
模式下调节 power level 设置
```

其中参数 level 按照枚举类型 RF_PowerTypeDef 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 rf_set_channel (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。

12.5.1.3 Emi Carrier Only (单载波)

Carrier 模式为 EMI Test 单载波发送模式, 用户直接调用 emicarrieronly () 函数即可, 无需其它设置。

```
void emicarrieronly(RF_ModeTypeDef rf_mode, RF_PowerTypeDef  
pwr, signed char rf_chn)
```

其中, 参数 rf_mode 为 RF_MODE_BLE_1M, 参数 pwr 和参数 rf_chn 按照前面介绍的设置方法来设置即可。

12.5.1.4 emi_con_prbs9

continue 模式为 EMI Test 载波上有连续调制的数据发送模式, 其中载波上的数据由 rf_continue_mode_loop () 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 emi_con_prbs9 () 函数即可进入 continue 模式, 无需其它设置。

在设置 continue 模式时, emi_con_prbs9 () 函数会调用 rf_emi_tx_continue_setup () 函数来完成 continue 模式的设置, 如 rf_mode、power level、chn 等。也会调用 rf_continue_mode_loop () 函数来更新载波上的数据。

```
void emi_con_prbs9(RF_ModeTypeDef rf_mode, RF_PowerTypeDef  
pwr, signed char rf_chn)
```

其中, 参数 rf_mode、power level 和参数 rf_chn 参照前面的介绍来设置即可。

12.5.1.5 Emi TX Burst

Tx Burst 模式为可以发送三种类型的数据包：PRBS9 packet payload、00001111b packet payload、10101010b packet payload。用户可以通过 cmd 来选择不同 TX 模式。

用户直接调用 emitxprbs9()、emitx55()、emitx0f() 其中一个函数即可进入 TX Burst 模式，无需其它设置。

```
void emitxprbs9(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr, signed char rf_chn);
```

```
void emitx55(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr, signed char rf_chn);
```

```
void emitx0f(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr, signed char rf_chn);
```

其中，参数 rf_mode、power level 和参数 rf_chn 参照之前的介绍来设置即可。

emitxprbs9()、emitx55()、emitx0f() 函数都会调用 rf_emi_tx_burst_setup 函数来完成 tx burst 初始化设置，做完 TX 初始化之后，结合 rf_emi_tx_burst_loop() 函数来触发发包，以及更新 payload 内容。

```
void rf_emi_tx_burst_setup(RF_ModeTypeDef rf_mode, unsigned char power_level, signed char rf_chn, unsigned char pkt_type)
```

其中，参数 rf_mode、power level 和参数 rf_chn 参照之前的介绍设置即可。参数 pkt_type 0 为发包 payload 为 PRBS9，1 为 00001111b，2 为 10101010b。

12.5.1.6 EMI RX

通过调用 emirx() 进入 rx mode，在 main_loop() 中调用 rf_emi_rx_loop() 来轮询 RX 是否收到数据，并对收到的 RX 数据进行数量和 RSSI 统计。

```
void emirx(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr, signed char rf_chn);
```

```
void rf_emi_rx_loop(void);
```

其中，参数 rf_mdoe、pwr 和参数 rf_chn 参照之前的介绍来设置即可。

12.5.1.7 上位机配置参数设置

Run:

0	Default	1	Start test
---	---------	---	------------

Cmd:

1	CarrierOnly	2	ContinuePRBS9	3	RX
4	TXBurst(PRBS9)	5	TXBurst(0x55)	6	TXBurst(0x0f)

Power 、 **channel** 前面已经介绍。

Mode:

0	Reserve	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 10.4dbm 的发射功率发射单载波。

12.5.2 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：

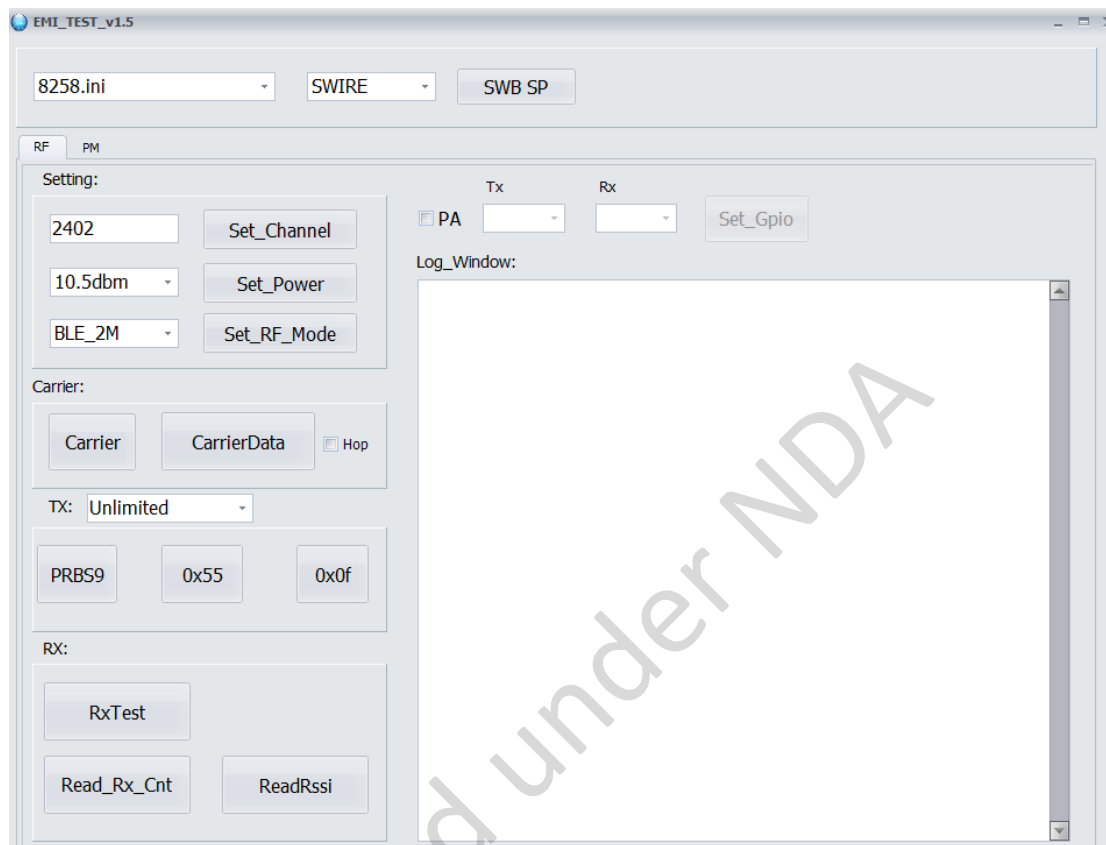


图 12-2 EMI test tool

第一步：选择芯片型号

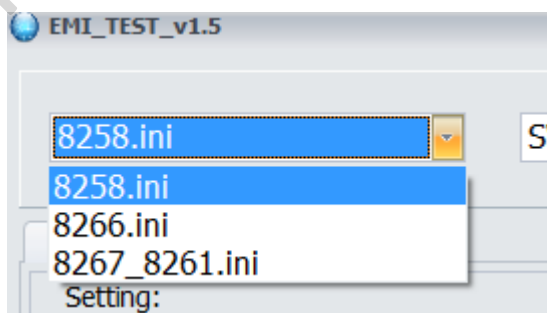


图 12-3 选择芯片型号

第二步：用户可以选择与硬件连接的方式。当选择 **Swire** 时，如果系统时钟在 16MHz 或以下，需要通过 **WTCDB** 工具 **SWB SP** 一下，保证能正常通信。

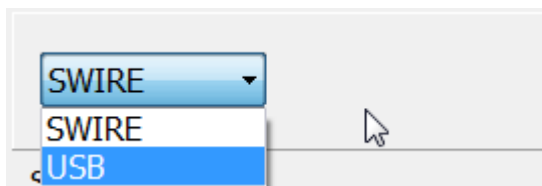


图 12-4 选择数据总线

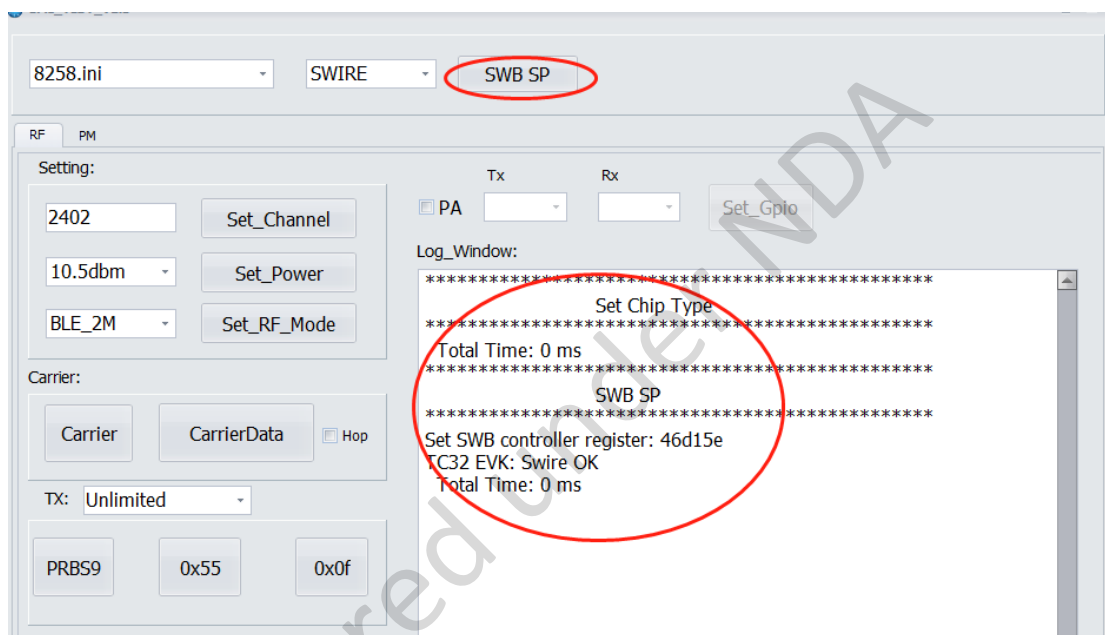


图 12-5 Swire 同步操作

第三步：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。

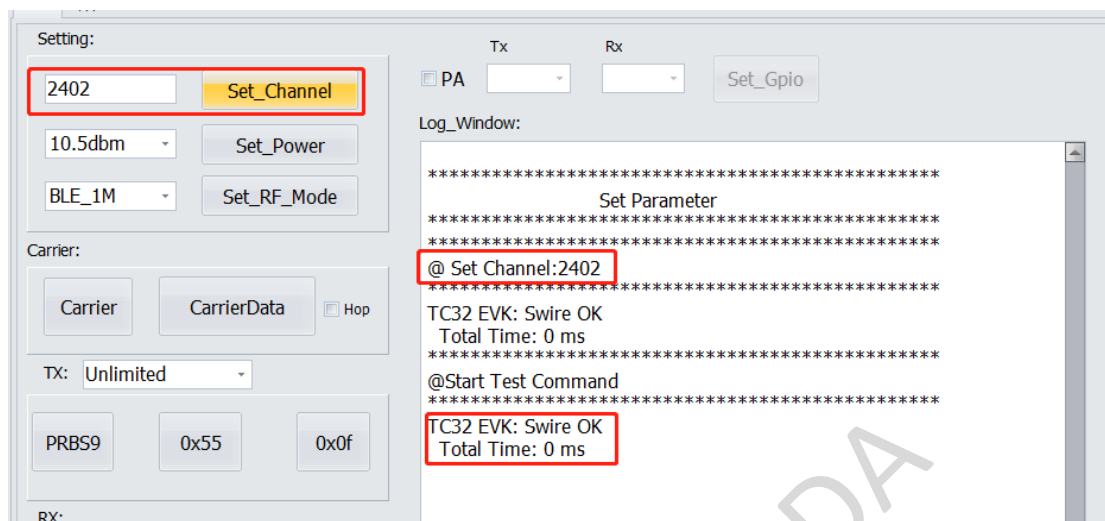


图 12-6 set channel

第四步：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。

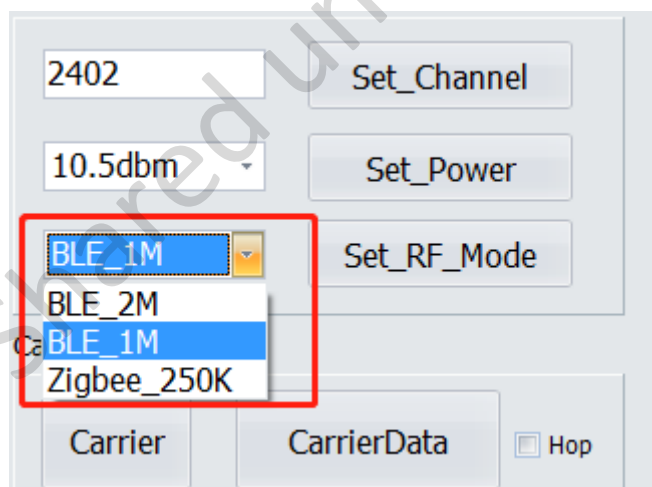


图 12-7 选择 RF 模式

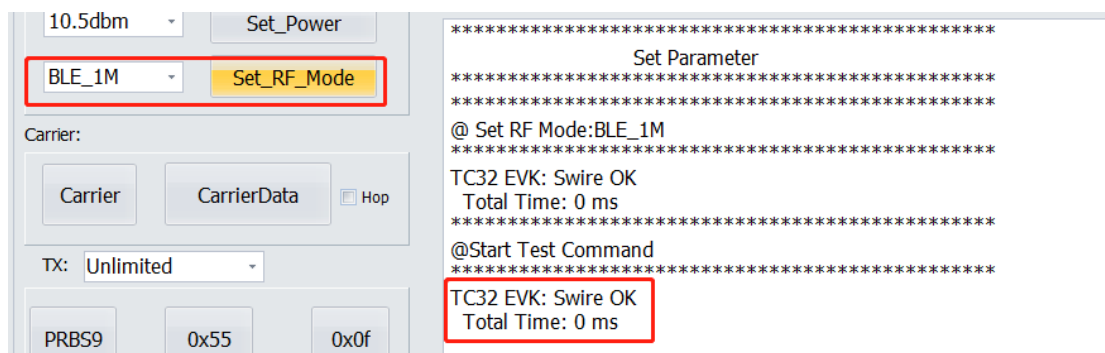


图 12-8 set RF 模式显示界面

第五步：点击 Carrier、CarrierData、RXTest、PRBS9、0x55、0x0f 即可进入不同的模式。

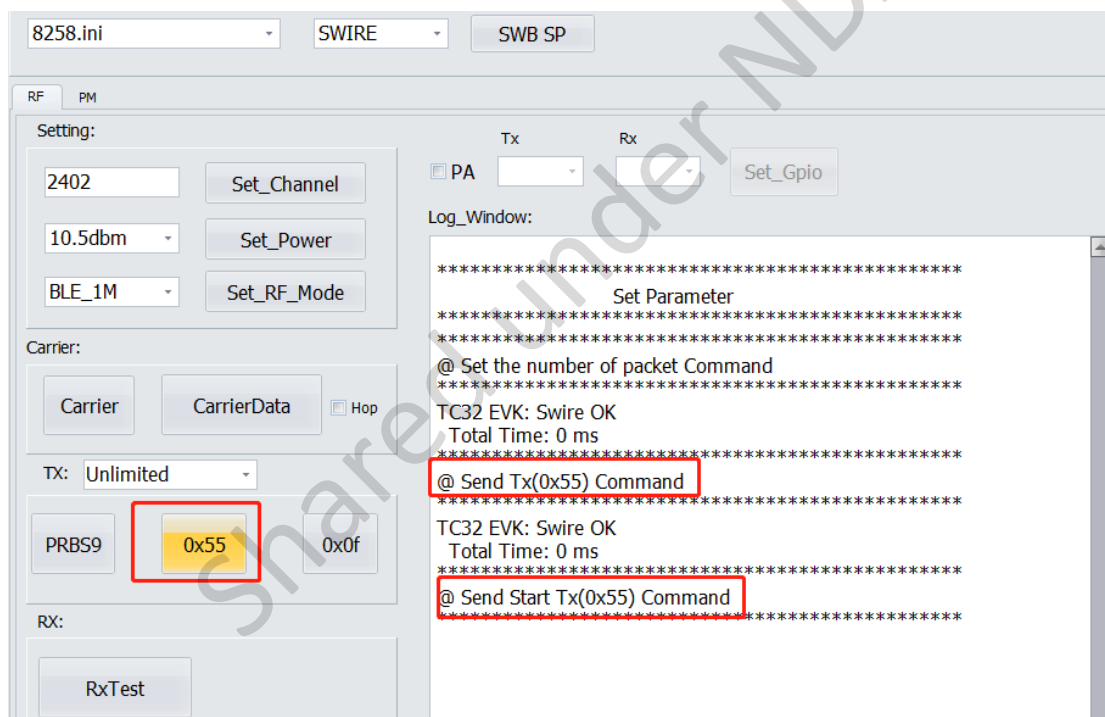


图 12-9 选择测试模式

第六步：TX 模式下，可以选择发 1000 个包或无限发包。

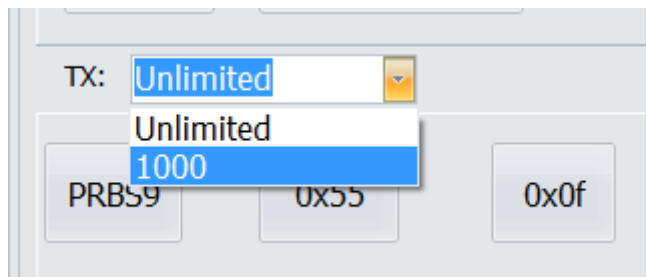


图 12-10 设置 TX packet number

第七步：RX 模式下，可以通过点击 Read_Rx_Cnt 读取收包个数，点击 ReadRssi 获取当前 RSSI，如下图所示。

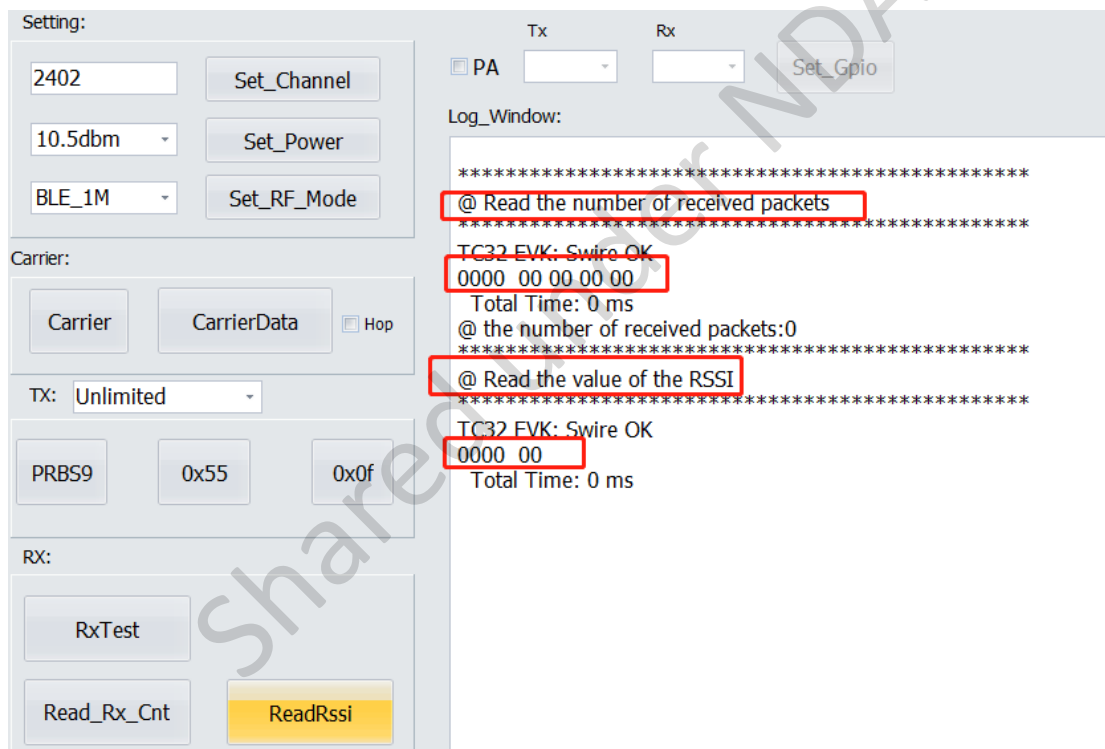


图 12-11 RX packet number 和 RSSI

13 附录

13.1 附录 1: crc16 算法

```
unsigned short crc16 (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;
}
```

14 FreeRTOS SDK 说明及注意事项

本章节关于 freeRTOS 部分文档及说明, 仅仅适用于特定的关于 freeRTOS 版 SDK。

14.1 freeRTOS 主要参数

系统时钟

系统时钟是由 app_config.h 里面的 #define CLOCK_SYS_CLOCK_HZ 16000000 指定, freeRTOSconfig.h 里面的 #define configCPU_CLOCK_HZ (16000000UL) 无效

Tick Rate

```
#define configTICK_RATE_HZ 1000
```

Tick Rate 用户可以自行设置, 一般是 100 – 1000。不能大于 1000

14.2 系统初始化

Main.c 初始化代码尽量保持不变。例如以下两句请保留

```
#if (BATT_CHECK_ENABLE)
adc_hw_initialized = 0;
#endif
blt_dma_tx_rptr = 0;
```

以下这句是从 deep wakeup 起来后恢复任务, 请保留

```
if(deepRetWakeUp){
    DEBUG_GPIO(GPIO_CHN4, 0);
    DEBUG_GPIO(GPIO_CHN0, 1);
    task_restore(1);
    // never reach here
```

14.3 创建任务

创建协议栈任务如下

```
xTaskCreate( proto_task, "tProto", 128, (void*)0, TASK_PROT_PRIORITY, &handle_proto_task );
```

tProto 任务是必须运行的, 不能删除。协议栈任务优先级一般为最高优先级

UI 任务在每次 wakeup 唤醒都会调用一次, 所以可以将一些跟唤醒周期相关的操作放在 ui_task 里面

```
xTaskCreate( ui_task, "tUI", configMINIMAL_STACK_SIZE, (void*)0, TASK_UI_PRIORITY, &handle_ui );
```

在任务调度 vTaskStartScheduler 前, 不要使能 IRQ