



TELINK SEMICONDUCTOR

Application Note :

Telink Kite BLE SDK

Developer Handbook

AN-19011501-E4

Ver1.3.0

2019/9/26

Brief:

This document is the guide for Telink Kite BLE SDK
3.4.0 for 8x5x family.



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:

telinkcnsales@telink-semi.com

telinkcnssupport@telink-semi.com

Change Log

Version	Main Changes	Date	Authors
1.0.0	Initial Release	2019/2	WSH, FQH, PT, Cynthia
1.1.0	<p>Updated the sections including:</p> <p>1.4.2 BLE Master demo, 1.4.3 Feature demo and driver demo, 2.1.3.2 Flash space operation, 2.3.4 Volatile GPIO digital states over deepsleep retention mode, Telink BLE Slave, 3.2.4.5 Timing sequence in Conn state Slave role, Link Layer TX fifo & RX fifo, 3.2.7 Controller Event, 3.2.8 Data Length Extension, 3.2.9.7 bls_ll_setAdvParam, 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 Low power mode.</p> <p>Added the sections including:</p> <p>3.3.1 BLE Host, 3.3.4 SMP, 3.3.5 GAP, 12.2 Select 32kHz clock source.</p>	2019/5	TYF, WSH, FQH, Cynthia
1.2.0	<p>As per SDK 3.3.1, updated the sections including:</p> <p>1.1 Software architecture 2.1.2 SRAM space partition 4.2.6 API blc_pm_setDeepsleepRetentionType</p> <p>Added the sections including:</p> <p>1.2 Applicable ICs 1.3 Software bootloader</p>	2019/5	TYF, WSH, FQH, Cynthia

Version	Main Changes	Date	Authors
1.3.0	<p>Updated the sections including :</p> <p>1.3 software bootloader 2.1.2.1 Sram and Firmware space 3.2.3.1 Link Layer state machine initialization 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 storage architecture 7.1.3 Modify FW size and booting address</p> <p>As per SDK 3.4.0, add sections including :</p> <p>3.2.10 Coded PHY/2M PHY 3.2.11 Channel Selection Algorithm#2 3.2.12 Extended Advertising 14 FreeRTOS SDK Support Appendix</p>	2019/8	FQH, TYF, WSH, Ryan, JF

Table of contents

Table of contents.....	4
1 SDK Overview.....	17
1.1 Software architecture	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 Applicable ICs	20
1.3 Software bootloader	20
1.4 Demo.....	22
1.4.1 BLE Slave demo	23
1.4.2 BLE Master demo	23
1.4.3 Feature demo and driver demo.....	24
2 MCU Basic Modules.....	25
2.1 MCU address space.....	25
2.1.1 MCU address space allocation.....	25
2.1.2 SRAM space partition.....	26
2.1.2.1 Sram and Firmware space	26
2.1.2.2 List file analysis demo.....	34
2.1.3 MCU address space access	38
2.1.3.1 Peripheral space access.....	38
2.1.3.2 Flash space operation.....	39
2.1.4 SDK flash space partition	42
2.2 Clock module.....	45
2.2.1 System clock & System Timer	45
2.2.2 System Timer usage	47
2.3 GPIO module	48

2.3.1	GPIO definition.....	48
2.3.2	GPIO state control.....	49
2.3.3	GPIO initialization	51
2.3.4	Volatile GPIO digital states over deepsleep retention mode	53
2.3.5	Configure SWS to avoid MCU error	54
3	BLE Module.....	55
3.1	BLE SDK software architecture	55
3.1.1	Standard BLE SDK architecture	55
3.1.2	Telink BLE SDK architecture	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 introduction.....	60
3.2.2	Link Layer state machine	60
3.2.3	Link Layer state machine combined application	62
3.2.3.1	Link Layer state machine initialization	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.....	67
3.2.4	Link Layer timing sequence.....	69
3.2.4.1	Timing sequence in Idle state.....	69
3.2.4.2	Timing sequence in Advertising state.....	69
3.2.4.3	Timing sequence in Scanning state	70
3.2.4.4	Timing sequence in Initiating state	71
3.2.4.5	Timing sequence in Conn state Slave role	71
3.2.4.6	Timing sequence in Conn state Master role	73
3.2.4.7	Conn state Slave role timing protection	74
3.2.5	Link Layer state machine extension.....	75

3.2.5.1 Scanning in Advertising state	76
3.2.5.2 Scanning in ConnSlaveRole.....	76
3.2.5.3 Advertising in ConnSlaveRole	77
3.2.5.4 Advertising and Scanning in ConnSlaveRole.....	78
3.2.6 Link Layer TX fifo & RX fifo.....	79
3.2.7 Controller Event	83
3.2.7.1 Controller HCI Event	83
3.2.7.2 Telink defined event	90
3.2.8 Data Length Extension	101
3.2.9 Controller API.....	103
3.2.9.1 Controller API brief.....	103
3.2.9.2 API return type ble_sts_t.....	104
3.2.9.3 MAC address initialization.....	104
3.2.9.4 Link Layer state machine initialization	105
3.2.9.5 bls_ll_setAdvData.....	105
3.2.9.6 bls_ll_setScanRspData.....	106
3.2.9.7 bls_ll_setAdvParam.....	107
3.2.9.8 bls_ll_setAdvEnable	111
3.2.9.9 bls_ll_setAdvDuration	112
3.2.9.10 blc_ll_setAdvCustomedChannel.....	113
3.2.9.11 rf_set_power_level_index.....	113
3.2.9.12 blc_ll_setScanParameter	114
3.2.9.13 blc_ll_setScanEnable	116
3.2.9.14 blc_ll_createConnection.....	117
3.2.9.15 blc_ll_setCreateConnectionTimeout.....	119
3.2.9.16 blm_ll_updateConnection.....	119
3.2.9.17 bls_ll_terminateConnection	119
3.2.9.18 blm_ll_disconnect	121
3.2.9.19 Get Connection Parameters	121
3.2.9.20 blc_ll_getCurrentState.....	122
3.2.9.21 blc_ll_getLatestAvgRSSI.....	122

3.2.9.22	Whitelist & Resolvinglist.....	122
3.2.10	Coded PHY/2M PHY	124
3.2.10.1	Coded PHY/2M PHY.....	124
3.2.10.2	Coded PHY/2M PHY Demo	124
3.2.10.3	Coded PHY/2M PHY API.....	125
3.2.11	Channel Selection Algorithm #2	126
3.2.12	Extended Advertising	126
3.2.12.1	Extended Advertising Introduction	126
3.2.12.2	Extended Advertising Demo setup.....	127
3.2.12.3	Extended Advertising Related API	128
3.3	BLE Host	131
3.3.1	BLE Host	131
3.3.2	L2CAP	131
3.3.2.1	Register L2CAP data processing function	132
3.3.2.2	Update connection parameters	133
3.3.3	ATT & GATT	138
3.3.3.1	GATT basic unit “Attribute”	138
3.3.3.2	Attribute and ATT Table.....	140
3.3.3.3	Attribute PDU & GATT API.....	149
3.3.3.4	GATT Service Security.....	162
3.3.3.5	8258 master GATT	165
3.3.4	SMP	167
3.3.4.1	SMP security level	167
3.3.4.2	SMP parameter configuration	168
3.3.4.3	SMP Security Request	175
3.3.4.4	SMP bonding info	178
3.3.4.5	Master SMP	182
3.3.5	GAP.....	190
3.3.5.1	GAP initialization	190
3.3.5.2	GAP event.....	190

4	Low Power Management (PM).....	197
4.1	Low power driver	197
4.1.1	Low power mode	197
4.1.2	Low power wakeup source	200
4.1.3	Sleep and wakeup from low power mode.....	202
4.1.4	Low power wakeup procedure	204
4.1.5	API pm_is_MCU_deepRetentionWakeup.....	208
4.2	BLE low power management	208
4.2.1	BLE PM Initialization	208
4.2.2	BLE PM for Link Layer.....	209
4.2.3	BLE PM variables.....	211
4.2.4	API bls_pm_setSuspendMask	212
4.2.5	API bls_pm_setWakeupSource	214
4.2.6	API blc_pm_setDeepsleepRetentionType	214
4.2.7	PM software processing flow	215
4.2.7.1	blt_sdk_main_loop.....	215
4.2.7.2	blt_brx_sleep.....	216
4.2.8	Analysis of deepsleep retention	219
4.2.8.1	API blc_pm_setDeepsleepRetentionThreshold.....	219
4.2.8.2	blc_pm_setDeepsleepRetentionEarlyWakeupTiming	223
4.2.8.3	Optimization and measurement of T_init	224
4.2.9	Connection Latency	229
4.2.9.1	Sleep timing with non-zero connection latency.....	229
4.2.9.2	latency_use calculation	230
4.2.10	API bls_pm_getSystemWakeupTick.....	232
4.3	Potential issues in GPIO wakeup	233
4.3.1	Failure to enter sleep at effective wakeup level	233
4.4	BLE System Low Power Management	234
4.5	Timer wakeup by Application Layer.....	236

5	Low Battery Detect	238
5.1	Impact of Low Battery Detect	238
5.2	Implementation of Low Battery Detect	239
5.2.1	Cautions	239
5.2.1.1	MUST use GPIO input channel	239
5.2.1.2	MUST use ADC Differential Mode	241
5.2.1.3	Must use Dfifo for ADC sampling value.....	241
5.2.1.4	ADC Channel Switch	242
5.2.2	Dedicated Low Battery Detect Demo	242
5.2.2.1	Initialization of Low Battery Detect.....	242
5.2.2.2	Low Battery Detect Processing.....	245
5.2.2.3	Low battery voltage alarm.....	247
5.2.2.4	Debug mode for Low Battery Detect.....	248
5.2.3	Low battery detect and Amic Audio	249
6	Audio	250
6.1	Audio initialization	250
6.1.1	AMIC and Low Battery Detect	250
6.1.2	AMIC initialization setting	250
6.1.3	DMIC initialization setting	251
6.2	Audio data processing	252
6.2.1	Audio data volume and RF transfer	252
6.2.2	Audio data compression	254
6.3	Compression and decompression algorithm	256
7	OTA	258
7.1	Flash architecture and OTA procedure	258
7.1.1	FLASH storage architecture	258
7.1.2	OTA update procedure	259
7.1.3	Modify FW size and booting address.....	261
7.2	RF data proceesing for OTA mode	262

7.2.1	OTA processing in Attribute Table on Slave side	262
7.2.2	OTA data packet format.....	263
7.2.3	RF transfer processing on Master side	264
7.2.4	RF receive processing on Slave side.....	267
8	Key Scan	271
8.1	Key matrix	271
8.2	Keyscan and keymap.....	273
8.2.1	Keyscan	273
8.2.2	Keymap &kb_event.....	274
8.3	Keyscan flow	277
8.4	Deepsleep wake_up fast keyscan	279
8.5	Repeat Key processing.....	280
8.6	Stuck Key processing.....	282
9	LED Management	284
9.1	LED task related functions	284
9.2	LED task configuration and management.....	284
9.2.1	LED event definition.....	284
9.2.2	LED event priority	285
10	Blt Software Timer.....	287
10.1	Timer initialization	287
10.2	Timer inquiry processing.....	288
10.3	Add timer task.....	290
10.4	Delete timer task.....	291
10.5	Demo.....	291
11	IR	294
11.1	PWM Driver.....	294
11.1.1	PWM id and pin.....	294
11.1.2	PWM clock	295

11.1.3	PWM cycle and duty	296
11.1.4	PWM revert.....	297
11.1.5	PWM start and stop	297
11.1.6	PWM mode	298
11.1.7	PWM pulse number	298
11.1.8	PWM interrupt.....	298
11.1.9	PWM phase.....	301
11.1.10	API for IR DMA FIFO mode.....	301
11.1.10.1	Configuration for DMA FIFO	302
11.1.10.2	Set DMA FIFO buffer.....	302
11.1.10.3	Start and Stop for IR DMA FIFO mode.....	303
11.2	IR Demo.....	303
11.2.1	PWM mode selection.....	303
11.2.2	Demo IR protocol	304
11.2.3	IR timing design.....	304
11.2.4	IR initialization.....	307
11.2.4.1	rc_ir_init	307
11.2.4.2	IR hardware configuration.....	308
11.2.4.3	IR variable initialization	308
11.2.5	FifoTask configuration	309
11.2.5.1	FifoTask_data.....	309
11.2.5.2	FifoTask_idle	310
11.2.5.3	FifoTask_repeat	311
11.2.5.4	FifoTask_repeat*n & FifoTask_idle_repeat*n	312
11.2.6	Check IR busy status in APP layer	312
12	Other Modules.....	313
12.1	External capacitor for 24MHz crystal.....	313
12.2	Select 32kHz clock source	314
12.3	PA	314

12.4	PHY test.....	315
12.4.1	PhyTest API	315
12.4.2	PhyTest demo	316
12.4.2.1	Demo1: 8258_feature_test	316
12.4.2.2	Demo2: 8258_ble_remote	317
12.4.2.3	Adjust PhyTest parameters.....	318
12.5	EMI	318
12.5.1	EMI Test	318
12.5.1.1	EMI initialization setting.....	319
12.5.1.2	Power level and Channel.....	319
12.5.1.3	EMI Carrier Only	320
12.5.1.4	TX Continue mode.....	320
12.5.1.5	EMI TX Burst	321
12.5.1.6	EMI RX	322
12.5.1.7	Set Host configuration parameters	322
12.5.2	EMI Test Tool	323
13	Appendix.....	328
14	FreeRTOS SDK support appendix.....	329
14.1	FreeRTOS configuration	329
14.1.1	System CLock	329
14.1.2	Tick Rate	329
14.2	System Initialization	329
14.3	Task Creation.....	330

Table of figures

Figure 1-1	SDK file structure	17
Figure 1-2	Folder with 8251/8253/8258 bootloader and boot.link	20
Figure 1-3	software bootloader setting	21
Figure 1- 4	Demo code supplied in BLE SDK	23
Figure 2-1	MCU address space allocation	25
Figure 2-2	IC SRAM space allocation in 16k/32k retention mode	27
Figure 2-3	SRAM & FW space allocation	28
Figure 2-4	Section distribution in list file	34
Figure 2-5	Section address in list file.....	35
Figure 2-6	512kB flash space partition.....	43
Figure 2-7	system clock & System Timer.....	45
Figure3-1	BLE SDK standard architecture	55
Figure3-2	HCI data transfer between Host and Controller	56
Figure3-3	8258 HCI architecture	57
Figure3-4	Telink BLE Slave architecture	58
Figure3-5	Telink BLE Master architecture	59
Figure3-6	State diagram of Link Layer state machine in BLE Spec.....	61
Figure3-7	Telink Link Layer state machine	61
Figure3-8	Idle + Advertising	63
Figure3-9	Idle + Scanning.....	64
Figure3-10	BLE Slave LL state	65
Figure3-11	BLE Master LL state.....	67
Figure 3-12	Timing sequence chart in Advertising State	69
Figure3-13	Timing sequence chart in Scanning state	70
Figure3-14	Timing sequence chart in Initiating state	71
Figure3-15	Timing sequence chart in Conn state Slave role.....	71
Figure3-16	Timing sequence chart in ConnMasterRole	73
Figure3-17	Timing sequence chart with Scanning in Advertising state	76
Figure3-18	Timing sequence chart with Scanning in ConnSlaveRole	77
Figure3-19	Timing sequence chart with Advertising in ConnSlaveRole	78

Figure3-20 Timing sequence chart with Advertising and Scanning in ConnSlaveRole.....	78
Figure3-21 RX overflow case 1	80
Figure3-22 RX overflow case 2	81
Figure3-23 BLE SDK event architecture.....	83
Figure3-24 HCI event	84
Figure3-25 Disconnection Complete Event.....	85
Figure3-26 Read Remote Version Information Complete Event.....	86
Figure3-27 LE Connection Complete Event	87
Figure3-28 LE Advertising Report Event	87
Figure3-29 LE Connection Update Complete Event	88
Figure3-30 Connect request PDU	94
Figure3-31 LL_CONNECTION_UPDATE_REQ format in BLE stack	100
Figure3-32 Adv packet format in BLE stack.....	105
Figure3-33 Advertising Event in BLE stack	107
Figure3-34 Four adv events in BLE stack.....	108
Figure3-35 BLE L2CAP structure and ATT packet assembly model.....	132
Figure3-36 Connection Para update Req format in BLE stack	134
Figure3-37 BLE sniffer packet sample: conn para update request & response	134
Figure3-38 conn para update rsp format in BLE stack	136
Figure3-39 BLE sniffer packet sample: ll conn update req.....	137
Figure3-40 GATT service containing Attribute group.....	139
Figure3-41 Attribute Table in this BLE SDK	140
Figure3-42 BLE sniffer packet sample when Master reads hidInformation ...	144
Figure3-43 Write Request in BLE stack	145
Figure3-44 Write Command in BLE stack.....	145
Figure3-45 Execute Write Request in BLE stack.....	146
Figure3-46 Service/Attribute Layout.....	148
Figure3-47 Read by Group Type Request/Read by Group Type Response	150
Figure3-48 Find by Type Value Request/Find by Type Value Response.....	151
Figure3-49 Read by Type Request/Read by Type Response	152

Figure3-50	Find information request/Find information response.....	153
Figure3-51	Read Request/Read Response.....	153
Figure3-52	Read Blob Request/Read Blob Response	154
Figure3-53	Exchange MTU Request/Exchange MTU Response	154
Figure3-54	Write Request/Write Respons	156
Figure3-55	Example for Write Long Characteristic Values	158
Figure3-56	Handle Value Notification in BLE Spec.....	158
Figure3-57	Handle Value Indication in BLE spec.....	160
Figure3-58	Handle Value Confirmation in BLE Spec	162
Figure3-59	Mapping diagram for service request and response.....	163
Figure3-60	ATT Permission definition	164
Figure3-61	Local device pairing status.....	167
Figure3-62	Packet example for Pairing Disable	169
Figure3-63	Usage rule for MITM/OOB flag in legacy pairing mode	171
Figure3-64	Mapping relationship for KEY generation method and IO capability ..	172
Figure3-65	Packet example for Pairing Peer Trigger	177
Figure3-66	Packet example for Pairing Conn Trigger	178
Figure3-67	Master initiates Pairing_Req	192
Figure6-1	Audio data sample	252
Figure6-2	MIC service in Attribute Table	253
Figure6-3	Data compression processing	255
Figure6-4	Data corresponding to compression algorithm.....	256
Figure7-1	Flash storage structure	258
Figure7-2	Write Command format in BLE stack.....	263
Figure7-3	Format of OTA command and data.....	263
Figure7-4	Master obtains OTA Attribute Handle via “Read By Type Request”	264
Figure7-5	firmware sample: starting part.....	265
Figure7-6	firmware sample: ending part	265
Figure7-7	“OTA start” sent from Master.....	266
Figure7-8	Master OTA data	267

Figure8-1 Row/Column key matrix	271
Figure11-1 PWM cycle & duty cycle	296
Figure11-2 PWM interrupt.....	299
Figure11-3 DMA FIFO buffer for IR DMA FIFO mode.....	301
Figure11-4 Demo IR protocol.....	304
Figure11-5 IR timing 1.....	304
Figure11-6 IR timing 2	306
Figure12-1 24MHz crystal circuit.....	313
Figure12-2 EMI test tool	323
Figure12-3 Select chip type.....	323
Figure12-4 Select data bus.....	324
Figure12-5 Swire synchronization operation	324
Figure12-6 Set channel	325
Figure12-7 Select RF mode	325
Figure12-8 Interface after RF mode setting.....	326
Figure12-9 Select test mode	326
Figure12-10 Set TX packet number.....	327
Figure12-11 Read RX packet number and RSSI.....	327

1 SDK Overview

This BLE SDK supplies demo code for BLE Slave/Master development, based on which user can develop his own application program.

1.1 Software architecture

Software architecture for this BLE SDK includes APP layer and BLE protocol stack.

Figure 1-1 shows the file structure after the SDK project is imported in Telink IDE, which mainly contains 7 top-layer folders below: “application”, “boot”, “common”, “drivers”, “proj_lib”, “stack” and “vendor”.

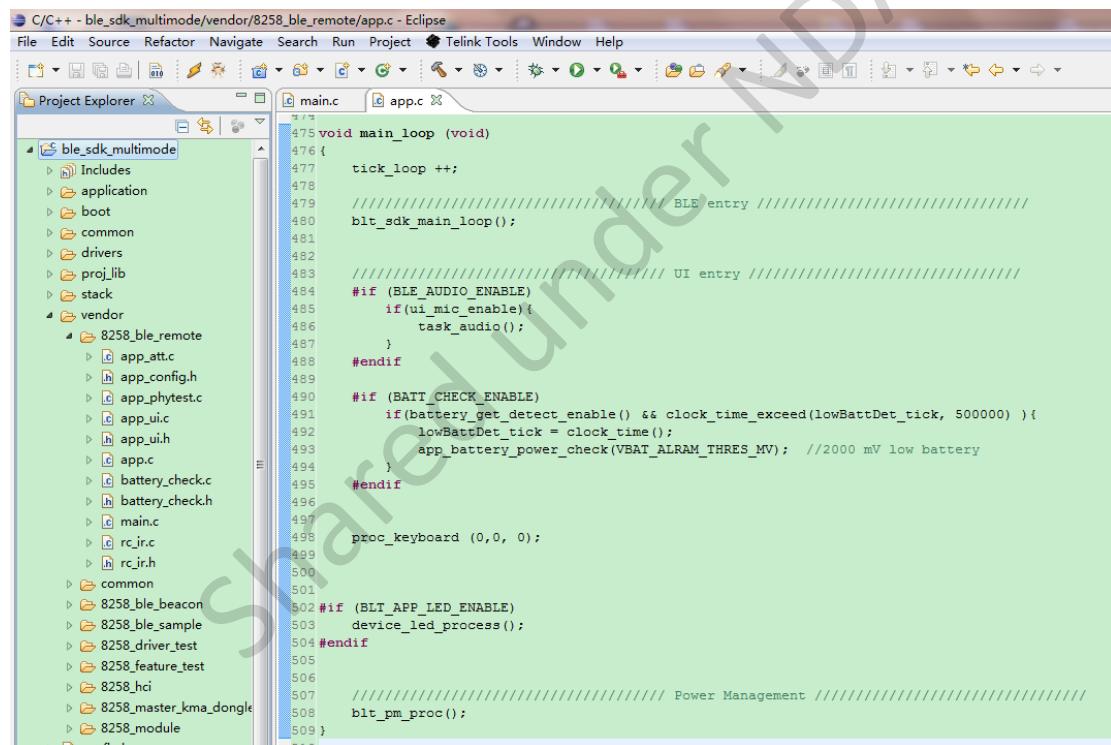


Figure 1-1 SDK file structure

- ❖ **application:** This folder contains general application program, e.g. print, keyboard, and etc.
- ❖ **boot:** This folder contains software bootloader for chip, i.e., assembly code after MCU power on or deepsleep wakeup, so as to establish environment for C program running.
- ❖ **common:** This folder contains generic handling functions across platforms, e.g. SRAM handling function, string handling function, and etc.

- ✧ drivers: This folder contains hardware configuration and peripheral drivers closely related to MCU, e.g. clock, flash, i2c, usb, gpio, uart.
- ✧ proj_lib: This folder contains library files necessary for SDK running, e.g. BLE stack, RF driver, PM driver. Since this folder is supplied in the form of library files (e.g. liblt_8258.a), the source files are not open to users.
- ✧ stack: This folder contains header files for BLE stack. Source files supplied in the form of library files are not open to users.
- ✧ vendor: This folder contains user APP-layer code.

1.1.1 main.c

The “main.c” file includes main function entry, system initialization functions and endless loop “while(1)”. It’s not recommended to make any modification to this file.

```
int main (void) {

    blc_pm_select_internal_32k_crystal(); //select 32k rc as 32k counter timer
    cpu_wakeup_init(); //MCU HW init

    int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();
                           //judge if deep retention is used

    rf_drv_init(RF_MODE_BLE_1M); //RF init

    gpio_init(!deepRetWakeUp); //gpio init, paras configurable for user in app_config.h

    if( deepRetWakeUp ){
        user_init_deepRetn(); //fast init after deep retention wakeup
    }
    else{
        user_init_normal(); //ble init, whole system init by user
    }

    irq_enable(); //enable global INT

    while (1){
        #if (MODULE_WATCHDOG_ENABLE)
            wd_clear(); //clear watch dog
        #endif

        main_loop(); //include BLE BRx, PM and UI task
    }
}
```

1.1.2 app_config.h

The user configuration file “app_config.h” serves to configure parameters of the whole system, including parameters related to BLE, GPIO, PM (low-power management), and etc. Parameter details of each module will be illustrated in following sections.

1.1.3 application file

- ✧ “app.c”: User file for system initialization, data processing and low power management.
- ✧ “app_att.c” of BLE Slave project: configuration files for services and profiles. Based on Telink Attribute structure, as well as Attributes such as GATT, standard HID, proprietary OTA and MIC, user can add his own services and profiles as needed.
- ✧ UI task files: IR, battery detect, and other user tasks.

1.1.4 BLE stack entry

There are two entry functions in BLE stack code of Telink BLE SDK.

- 1) BLE related interrupt handling entry in “irq_handler” function of “main.c” file

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

- 2) BLE logic and data processing function entry in application file mainloop

“blt_sdk_main_loop”.

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

1.2 Applicable ICs

Telink Kite BLE SDK is applicable to the 8251, 8253 and 8258 of the 8x5x series which share the common IP core and almost the same hardware modules except SRAM size.

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

Thus, for the three ICs above, the SDK file architecture is also consistent except SDK/boot/boot script (i.e. software bootloader file) and boot.link file.

1.3 Software bootloader

The three ICs have different software bootloader files which are saved under the directory “SDK/boot/”. Each IC supports three software bootloader files, corresponding to the case when enabling 16k deep retention, enabling 32k deep retention, or disabling deep retention. For the description of deep retention, please refer to **section 4 Low Power Management (PM)**.

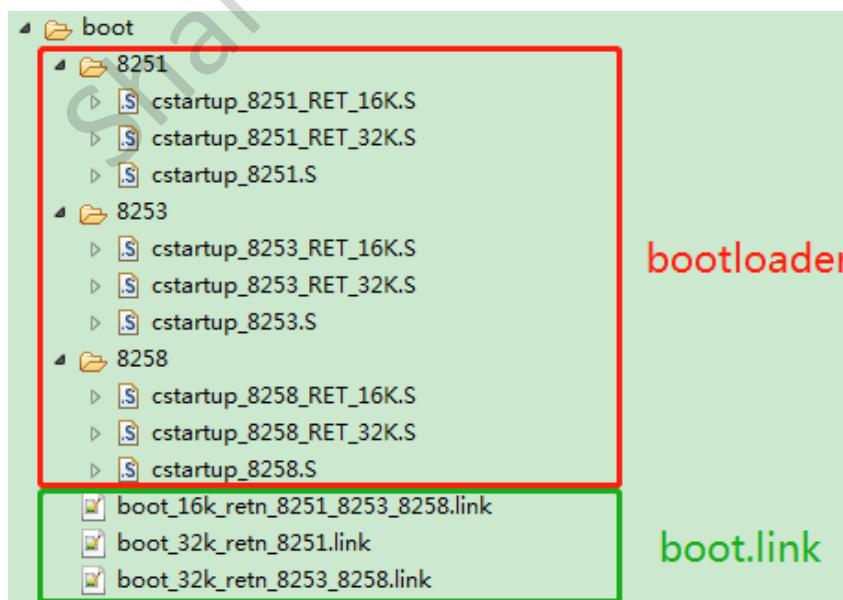


Figure 1-2 Folder with 8251/8253/8258 bootloader and boot.link

Take the “cstartup_8258_RET_16K.S” as an example: The first sentence “#ifdef MCU_STARTUP_8258_RET_16K” indicates that only when user has defined the “MCU_STARTUP_8258_RET_16K”, will this bootloader take effect.

Based on IC to be used and whether to adopt (16k or 32k) deep retention function, user can select software bootloader accordingly.

By default, Kite BLE SDK uses Sram size 64K and deepsleep retention 16K sram, the corresponding software bootloader and link file will be cstartup_8258_RET_16K.S and boot_16k_retn_8251_8253_8258.link. User could modify the retention size according to the application (Please refer to SRAM space partition).

The demo “8258_ble_remote” is taken as example to illustrate how to change the software bootloader for the 8258 to support deepsleep retention 32K sram.

1. user needs to define “-DMCU_STARTUP_8258_RET_32K” in the corresponding project setting, as shown in the figure below.

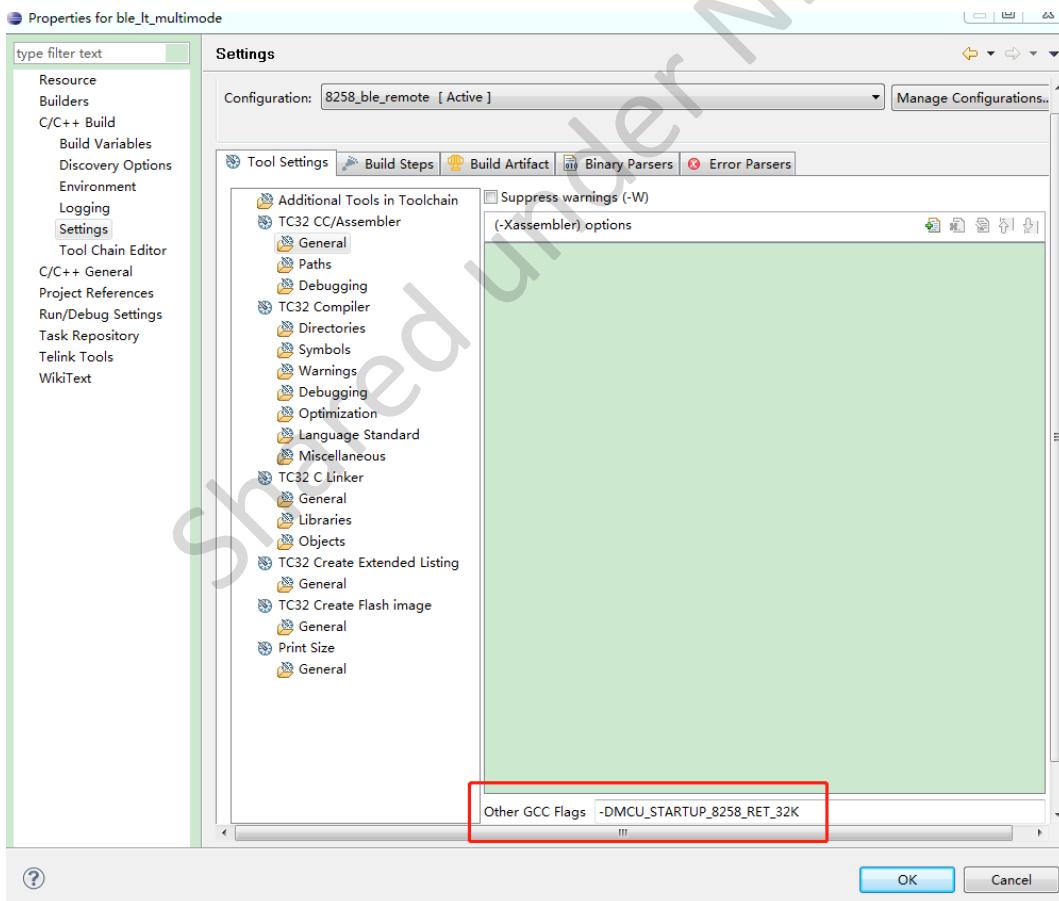


Figure 1-3 software bootloader setting

2. the software bootloader file is cstartup_8258_RET_16K.S, user also needs to use the SDK/boot/boot_32k_retn_8253_8258.link to replace the contents of the

boot.link file under the root directory of the SDK.

3. In the API user_init(), after calling blc_ll_initPowerManagement_module(), user needs to call API blc_pm_setDeepsleepRetentionType(DEEPSLEEP_MODE_RET_SRAM_LOW32K) in order to configure the Retention space.

***Note:** Since SRAM size for the 8251, 8253 and 8258 is different, after user selects the right software bootloader file, it's also needed to modify the boot.link file under the root directory of the SDK, i.e. replace its contents by the corresponding link file in the table below.

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

1.4 Demo

Telink BLE SDK supplies multiple BLE demos for user.

Each demo code corresponds to specific hardware, based on which user can run demo, observe effect and modify demo code for his own application development.

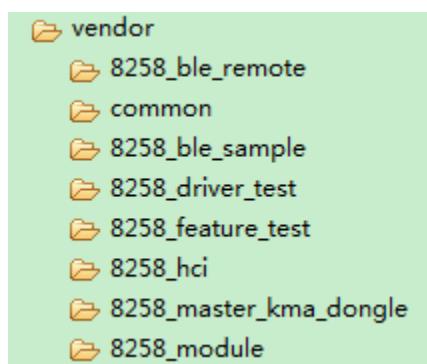


Figure 1- 4 Demo code supplied in BLE SDK

1.4.1 BLE Slave demo

The table below lists BLE Slave demos and their differences.

Demo	Stack	Application	MCU function
8258 hci	BLE controller	No	Controller, communicate with MCU Host via HCI interface
8258 module	BLE controller + host	Application in Host MCU	BLE SPP module
8258 ble remote	BLE controller + host	Remote control application	Host MCU
8258 ble sample	BLE controller + host	Simple Slave demo with only advertising and connection function	Host MCU

- ✧ “8258 hci” is a BLE Slave controller. It supplies USB/UART-based HCI to communicate with MCU Host and forms a complete BLE Slave system.
- ✧ Both “8258 ble remote” and “8258 module” are complete BLE slave stack provided by Telink.
“8258 module” only acts as BLE SPP module to communicate with Host MCU via UART interface. Generally application code is written in peer Host MCU.
- ✧ “8258 ble sample” is simplified demo based on 8258_ble_remote, which can pair and connect with standard iOS/Android device.

1.4.2 BLE Master demo

“8258 master kma dongle” is a demo of BLE Master single connection. It can connect and communicate with 8258 ble sample/8258 ble remote/8258 module.

Libraries corresponding to “8258 ble remote”/“8258 ble sample” supply standard BLE stack (Master and Slave share the same library), including BLE controller + BLE host. User only needs to add his own application code in APP layer by using APIs of controller and Host, with no need to process BLE Host.

The new SDK merges the library of Slave with that of Master. During code compiling of “8258 master kma dongle”, only standard BLE controller function part in the library is invoked, and standard Host function of Master is not contained in the library. The demo code of “8258 master kma dongle” gives BLE Host implementation in APP layer for reference, including ATT, simple SDP (service discovery protocol), the most common SMP (security management protocol), and etc.

For BLE Master, the most complex function is service discovery of Slave server and recognition of all services, which generally can be implemented in Android/linux system. Limited by Flash size and SRAM size, Telink 8258 IC cannot supply complete service discovery. However, SDK supplies all ATT interfaces needed for service discovery. Based on service discovery process of 8258 ble remote by 8258 master kma dongle, user can implement traversal of specific services.

1.4.3 Feature demo and driver demo

“8258_feature_test” gives demo code for some common features related to BLE. User can implement his own functions based on these demos. All features will be introduced in BLE section.

In the project “8258_feature_test”, by selectively defining the macro “FEATURE_TEST_MODE” in the “app_config.h”, user can switch to demos of different feature test modes.

“8258 driver test” gives sample code for basic drivers, based on which user can implement his own driver functions. The Driver section will introduce various drivers in detail.

In the project “8258 driver test”, by selectively defining the macro “DRIVER_TEST_MODE” in the “app_config.h”, user can switch to demos of different driver test modes.

2 MCU Basic Modules

2.1 MCU address space

2.1.1 MCU address space allocation

A typical 64kB SRAM option is shown as an example to illustrate MCU address space allocation of 8x5x family.

Telink 8x5x MCU supports maximum addressing space of 16M bytes, including:

- ✧ 8M-byte program space from 0 to 0x7FFFFF
- ✧ 8M-byte peripheral space (e.g. SRAM, register space) from 0x800000 to 0xFFFFFFF. 0x800000~0x80FFFF is register space; 0x840000~0x84FFFF is 64kB SRAM space.

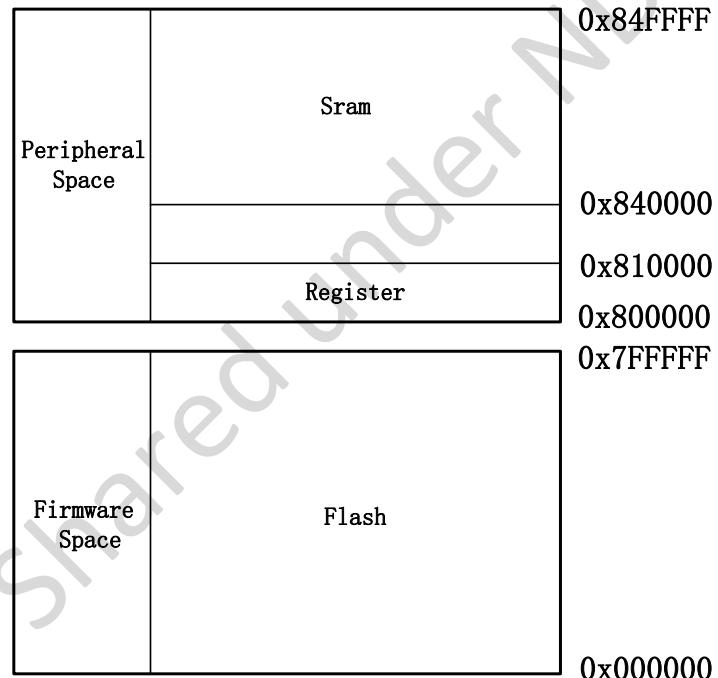


Figure 2-1 MCU address space allocation

During physical addressing of 8x5x MCU, address line BIT (23) serves to differentiate program space / peripheral space:

- ✧ Address line BIT (23) is 0: access program space
- ✧ Address line BIT (23) is 1: access peripheral space

When addressing space is peripheral space (BIT(23) is 1), address line BIT (18) serves to differentiate Register / SRAM.

- ✧ Address line BIT (18) is 0: access Register
- ✧ Address line BIT (18) is 1: access SRAM.

2.1.2 SRAM space partition

8x5x SRAM space partition is closely tied to deepsleep retention. Getting familiar with 8x5x deepsleep retention operation will help you understand SRAM partition.

For suspend and deepsleep only (i.e. without deepsleep retention), 8x5x SRAM space partition is the same as 826x family. Please refer to the “826x BLE SDK handbook” for more details.

2.1.2.1 Sram and Firmware space

For 32kB SRAM, address space range is 0x840000 ~ 0x848000.

For 48kB SRAM, address space range is 0x840000 ~ 0x84C000.

For 64kB SRAM, address space range is 0x840000 ~ 0x850000.

The figure below shows SRAM space allocation in 16k retention/32k retention mode for the 8258, 8253 and 8251.

***Note:** For the 8251 in deepsleep retention 32K sram mode, all allocated sections of its SRAM space are dynamically adjustable. Please refer to corresponding software bootloader and link file.

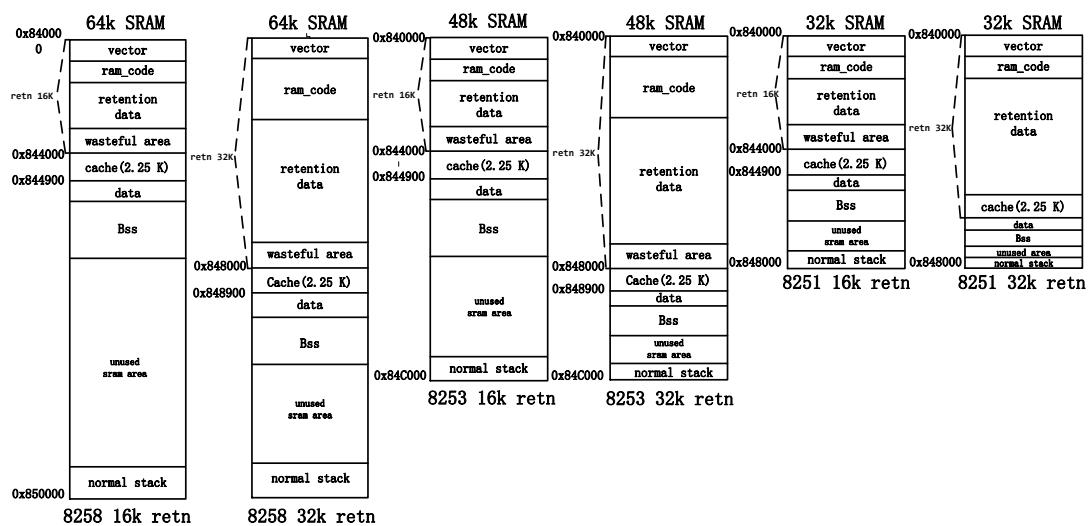


Figure 2-2 IC SRAM space allocation in 16k/32k retention mode

The 8258 with 64kB SRAM and default deepsleep retention 16K sram mode in the SDK is taken as an example to illustrate each part of the SRAM space.

By reference to this example, user can get SRAM partition details for 48k/32k SRAM size and deepsleep retention 32K sram mode.

The figure below shows SRAM and Firmware space allocation for 64kB SRAM.

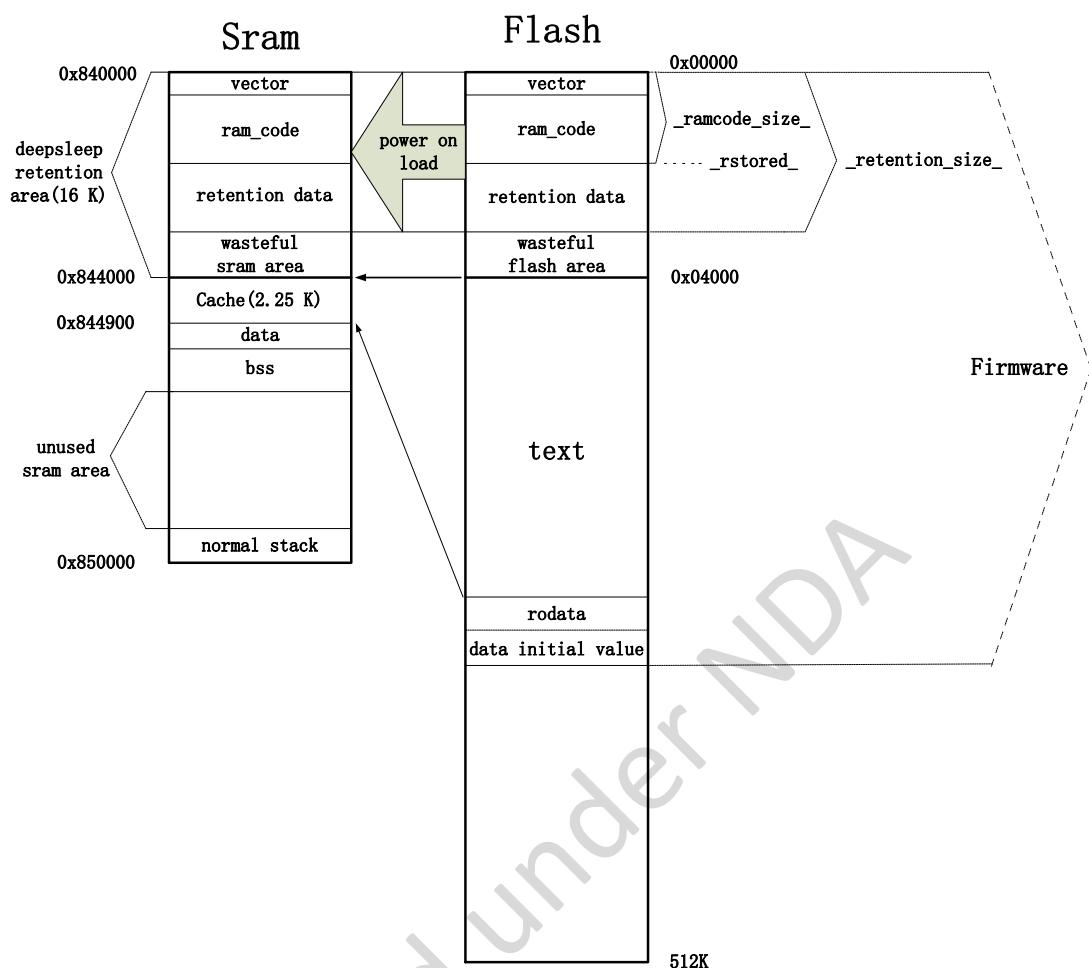


Figure 2-3 SRAM & FW space allocation

In the SDK, files related to SRAM space allocation include “boot.link” and “cstartup_8258_RET_16K.S”. As shown in **section 1.3 Software bootloader**, the contents of the boot.link herein are consistent with the file “boot_16k_retn_8251_8253_8258.link”. In the mode of deepsleep retention 32K Sram, corresponding bootloader and link file are cstartup_8258_RET_32K.S and boot_32k_retn_8253_8258.link, respectively.

Firmware in Flash includes vector, ramcode, retention_data, text, Rodata, and Data initial value. SRAM includes vector, ramcode, retention_data, Cache, data, bss, stack, and unused sram area. Note that vector/ramcode/retention_data in SRAM is a copy of vector/ramcode/ retention_data in Flash.

1) vectors, ram_code

The “vectors” section is software startup code (software bootloader) and it corresponds to the assembly file “cstartup_8258_RET_16K.S”.

The “ramcode” section is memory resident code in Flash Firmware, and it corresponds to all functions with the keyword “`_attribute_ram_code_`” (e.g. `flash_erase_sector` function).

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

In the following two cases, functions should be memory resident:

- ✧ Some functions (e.g. Flash operation functions) involve timing multiplex with four Flash MSPI pins: If these functions are placed in Flash, it will cause timing conflict and system crash.
- ✧ Whenever functions resident in RAM are invoked, there's no need to re-read them from Flash, and thus time will be saved. Therefore, the functions with limited execution time should be memory resident to increase execution efficiency. In the SDK, some functions related to BLE timing sequence need frequent execution, in order to decrease execution time and save power consumption, these functions are set as memory resident.

User can set a function as memory resident by adding the keyword “`_attribute_ram_code_`” (please refer to `flash_erase_sector`). After compiling, user can find this function in the ramcode section of list file.

The vector and ramcode in firmware should be loaded to SRAM at power on. After compiling, the total size of the two parts is “`_ramcode_size_`”, which is a variable recognizable by compiler. Its calculation is implemented in “boot.link”. As shown below, the compiling result “`_ramcode_size_`” equals the sum of the size of vector and ramcode.

```
. = 0x0;  
.vectors :  
{  
    * (.vectors)  
    * (.vectors.*)  
}  
.ram_code :  

```

ramcode)

2) retention_data

After the 8258 enters deepsleep retention mode, the beginning 16kB or 32kB data of SRAM are non-volatile (i.e. data will be retained and won't be lost).

Normally after compiling, global variables in the firmware are assigned to the "data" or "bss" section. Since neither of the sections locate in the retention area, they will be lost in deepsleep retention mode.

However, if placed in the "retention data" section by using the keyword `_attribute_data_retention_`, the specific variables will be retained through deepsleep retention mode.

Examples:

```
_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};
```

By reference to "data/bss" section in later section, initial value of global variables in the "data" section of SRAM should be pre-stored in flash; initial value of variables in the "bss" section is 0, and it can be directly set to 0 in SRAM when running bootloader.

Initial values of global variables in the "retention_data", regardless 0 or not, will be stored in the retention_data area of flash. After power on or normal deepsleep wakeup, they will be copied to the retention_data area of SRAM.

The "ram_code" section is followed by the "retention_data" section, i.e., the "vector + ramcode + retention_data" reside in sequence at top sections of flash, total size of which is `_retention_size_`. After power up or deepsleep wake_up, the 3 sections are copied to top of SRAM as a whole. Afterwards, as long as MCU does not enter deepsleep, the 3 sections will be resident in SRAM, with no more flash transfer needed. The 3 sections are non-volatile during suspend and deepsleep retention mode.

Configuration for the retention_data section in boot.link is shown as below:

```
. = (0x840000 + (_rstored_));  
.retention_data :  
    AT ( _rstored_ )  
{
```

```
    . = (((. + 3) / 4)*4);
    PROVIDE(_retention_data_start_ = . );
    *(.retention_data)
    *(.retention_data.*)
    PROVIDE(_retention_data_end_ = . );
}

}
```

Compiler assigns variable with the keyword “retention_data” to flash firmware with starting address of “_rstored”, corresponding to SRAM address 0x840000 + (_rstored). The “_rstored” is the end of the “ram_code” section.

If user’s configuration is using deepsleep retention 16K Sram mode, but the chose the “_retention_size_” is over the configuration 16K size, the compiler will throw out the error as below picture.

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

User can use the following methods to fix the issue

1. reduce the definte “attribute_data_retention” attribute number
2. switch to the right deepsleep retention size (e.g 32K Sram in this case, please refer to section 1.3)

When the “_retention_size_” is no more than 16kB, e.g. 12kB, there is a 4kB “wasteful flash area” in flash. The corresponding firmware binary file is all invalid Os from 12K to 16K. After copied to SRAM, this 4kB section is “wasteful SRAM area”.

To save flash/SRAM space, user can switch more variables or functions to ram_code/retention_data by adding corresponding keyword. It takes less time to run functions in ram_code and intialize variables in rentetion_data, which both contribute to power reduction (see **Section 4 Low Power Management (PM)**).

3) Cache

Cache is high-speed instruction buffer of MCU, and it must be configured as a section in SRAM. Cache size is fixed as 2.25kB (0x900), including 256-byte tag and 2048-byte Instructions cache.

Memory resident code can be directly read and executed from memory; however, only a small portion of firmware is memory resident code, and the majority are still in Flash.

According to program locality principle, a portion of Flash code can be stored in the Cache. Thus, if the code to be executed is in the Cache, instructions can be directly read and executed from the Cache; otherwise it's needed to transfer code from Flash to replace the old code in the Cache, then read and execute instructions from the Cache.

As shown in Figure 2-3, the “text” section in firmware is Flash code not placed in SRAM. According to program locality principle, it's needed to load this part to the Cache so that it can be executed.

Cache size is fixed as 2.25kB, and its starting address is configurable in SRAM. It is currently placed behind the SRAM 16K retention area, i.e., from 0x844000 to 0x844900.

4) data / bss

The “data” section in SRAM serves to store initialized global variables of program (i.e. global variables with non-zero initial value). The initial value of the global variables in the “data” section is “data init value” in firmware, as shown in Figure 2-3.

The “bss” section in SRAM serves to store global variables of program not initialized (i.e. global variables with initial value of zero).

Cache is followed by the “data” section, while the “data” section is followed by the “bss” section. The starting address of the “data + bss” is Cache ending address, i.e. 0x844900.

Following shows the code in “boot.link” which directly defines the starting address of the “data” section.

```
. = 0x844900;  
.data :
```

5) stack / unused area

For 64kB SRAM, “stack” in SRAM starts from 0x850000 (0x84C000 instead for 48kB SRAM, and 0x848000 for 32kB SRAM), which is the highest address. Its SP pointer will descend during push operation, and ascend during pop operation.

By default, size of stack used by SDK library does not exceed 256 bytes. However, since the size of used stack depends on stack depth (i.e. the address of the deepest location), final size of used stack is relevant to user upper-layer program design. Any case which causes deep stack, e.g. complex recursive function invoking is used, or large local array variable is used in a function, will increase the final size of used stack.

When large area of SRAM is used, user needs to know the size of stack used by program. This cannot be obtained by analyzing list file; instead, user should run actual product application, ensure all of the code which may use deep stack have been executed, then reset MCU and read SRAM space to determine the size of used stack.

“unused area” in SRAM is the space from deepest stack address to bss ending address. This area should exist to ensure non-overlap of stack and bss; otherwise it indicates SRAM size is not enough.

“bss” ending address can be obtained via list file, thus the maximum size for stack is determined. User needs to analyze whether this space is enough for stack usage. Please refer to **Section 2.1.2.2 List file analysis demo** for analysis method.

6) text

The “text” section is a part of Flash firmware. Functions with “_attribute_ram_code_” in firmware will be compiled as “ram_code”, while other functions without this keyword will be compiled as “text”.

The “text” section occupies the maximum space in firmware, which largely exceeds SRAM size generally. Therefore, it’s needed to use Cache buffer function, i.e. load code into Cache and then execute it.

7) rodata/data init value

The remaining data other than “vector”, “ram_code” and “text” in firmware are “rodata” and “data initial value”.

The “rodata” section is read-only data in firmware, i.e. variables with keyword “const”. E.g. ATT table in Slave:

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

User can view the “my_Attributes” in the “rodata” section by checking corresponding list file.

As introduced above, the “data” section is initialized global variables in firmware, e.g.

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

The compiler will store the initial value “0x1234” in the “data initial value”. When the bootloader is executed, this initial value will be copied to memory address corresponding to the “testValue”.

2.1.2.2 List file analysis demo

A simple BLE Slave demo “8258 ble sample” is taken as an example to illustrate SRAM and Flash address space allocation (please refer to Figure 2-3).

Bin file and list file of this demo are available under the directory “SDK” -> “Demo” -> “list file analyze”.

All screenshots herein are available from the files including “boot.link”, “cstartup_8258_RET_16K.S”, “8258 ble sample.bin” and “8258 ble sample.list”.

The figure below shows section distribution in the list file. (Please note the alignment of Align bytes.)

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

Figure 2-4 Section distribution in list file

Following lists the sections in the list file.

- 1) vectors: start from Flash 0, size is 0x170, calculated end address is 0x170
- 2) ram_code: start from Flash 0x170, size is 0x23f0, calculated end address is 0x2560
- 3) retention_data: start from Flash 0x2560, size is 0xce8, calculated end address is 0x3248
- 4) text: start from Flash 0x4000, size is 0x614c, calculated end address is 0xa14c
- 5) rodata: start from Flash 0xa14c, size is 0x8ec, calculated end address is 0xaa38
- 6) data: start from SRAM 0x844900, size is 0x2c, calculated end address is 0x84492c
- 7) bss: start from SRAM 0x844930, size is 0x259, calculated end address is 0x844b89

The remaining SRAM space size is $0x850000 - 0x844b89 = 0xb477 = 46199$ bytes. Except for 256 bytes required by stack, there are 45943 bytes left for use.

```
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_
```

Figure 2-5 Section address in list file

The figure above shows start/end address of various sections by searching “section” in the list file. Following analysis is based upon Figure 2-4 and Figure 2-5:

1) vector

For the “vector” section, start address in flash firmware is 0, end address is 0x170 (last data address is 0x16e~0x16f), and size is 0x170. After power on and loading to SRAM, the corresponding address in SRAM is 0x840000 ~ 0x840170.

2) ram_code

For the “ram_code” section, start address is 0x170, and end address is 0x2560 (last data address is 0x255c~0x255f). After power on and loading to SRAM, the corresponding address in SRAM is 0x840170 ~ 0x842560.

Shared under NDA

3) retention_data

“retention_data” in flash starts at “_rstored” 0x2560, which is also the end of “ram_code”. In SRAM, it starts at 0x842560, and ends at 0x843248 (last data address is 0x843244~0x843247).

The total size of “vector + ram_code + retention_data”, “_retention_size_”, is 0x3248. Therefore, the first 16kB of flash firmware contains only 0x3248-byte valid data, while the 3.43kB or so space from 0x3248 to 0x400 are invalid “waste flash area” (corresponding section in “8258_ble_sample.bin” are all 0s). In SRAM, the 3.43kB space from 0x843248 to 0x84400 is “waste SRAM area”.

4) Cache

SRAM address space for Cache is: 0x844000~0x844900.

The list file does not show Cache related information.

5) text

For the “text” section in flash firmware, start address is 0x4000, end address is 0xa14c (last data address is 0xa148~0xa14b), and size is 0xa14c – 0x4000 = 0x614c, which is the same as explained before.

6) rodata

The “rodata” section starts at 0xa14c (the end of “text”), and ends at 0xaa38 (last data address is 0xaa34~0xaa37).

7) data

For the “data” section in SRAM, start address is 0x844900 (the end of Cache), size is 0x2c (as shown in the list file), and end address is 0x84492c (last data address is 0x844928~0x84492b).

8) bss

For the “bss” section in SRAM, start address is 0x844930 (the end of “data”, 16-byte aligned), size is 0x259 (as shown in the list file), and end address is 0x844b89 (last data address is 0x844b84~0x844b88).

The remaining SRAM space size is 0x850000 – 0x844b89 = 0xb477 = 46199 byte. Except for the 256 bytes required by stack, there are 45943 bytes left for use.

2.1.3 MCU address space access

MCU address space 0x000000 ~ 0xffff can be accessed in firmware as follows:

2.1.3.1 Peripheral space access

The peripheral space (register & SRAM) is directly accessed (read/write) via pointer.

```
u8 x = *(volatile u8*)0x800066; // read register 0x66  
*(volatile u8*)0x800066 = 0x26; //write register 0x66  
u32 y = *(volatile u32*)0x840000; //read SRAM 0x40000~0x40003  
*(volatile u32*)0x840000 = 0x12345678; //write SRAM 0x40000~0x40003
```

In firmware, functions including “write_reg8”, “write_reg16”, “write_reg32”, “read_reg8”, “read_reg16” and “read_reg32”, which implement pointer operation, are used to write or read the peripheral space correspondingly. Please refer to “drivers/8258/bsp.h” for details.

Note: For operations such as write_reg8(0x40000) / read_reg16(0x40000), to ensure the access space is Register/SRAM rather than Flash, the base address “0x800000” is automatically added (address line BIT(23) is 1), as shown below.

```
#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))
```

Please pay attention to memory alignment: If a pointer pointing to 2 bytes/4 bytes is used to access the peripheral space, make sure the address is 2-byte/4-byte aligned to avoid data read/write error. Following shows two incorrect formats:

```
u16 x = *(volatile u16*)0x840001; //0x840001 is not 2-byte aligned  
*(volatile u32*)0x840005 = 0x12345678; //0x840005 is not 4-byte aligned
```

The correct formats should be:

```
u16 x = *(volatile u16*)0x840000; //0x840000 is 2-byte aligned  
*(volatile u32*)0x840004 = 0x12345678; //0x840004 is 4-byte aligned
```

2.1.3.2 Flash space operation

Read/Write/Erase operation of the Flash space is implemented by using the function “flash_read_page”/“flash_write_page”/“flash_erase_sector”.

1) Flash Read/Write access operation

The functions including “flash_read_page” and “flash_write_page” serve to read or write the Flash space correspondingly.

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

Flash read operation example via “flash_read_page”:

```
void flash_read_page(u32 addr, u32 len, u8 *buf);  
u8 data[6] = {0 };  
  
flash_read_page(0x11000, 6, data); //read 6 bytes starting from flash  
0x11000 into the array “data”
```

Flash write operation example via “flash_write_page”:

```
flash_write_page(u32 addr, u32 len, u8 *buf);  
u8 data[6] = {0x11,0x22,0x33,0x44,0x55,0x66 };  
  
flash_write_page(0x12000, 6, data); //write 6-byte data “0x665544332211”  
into flash starting from 0x12000
```

Since the “flash_write_page” function accesses flash area starting from the “addr” within a page, the maximum allowed “len” should be the page size, i.e. 256 bytes. It’s not allowed to operate flash area across two or more pages in this function.

- ❖ If the “addr” is the starting address of one page, the “len” cannot exceed 256.

```
flash_write_page(0x12000, 256 , data) //correct, write 256 bytes into the page starting  
from 0x12000
```

```
flash_write_page(0x12000, 257 , data) //wrong, 257 bytes exceed page size “256”, and  
the final byte belongs to the next page
```

- ❖ If the “addr” is not the starting address of one page, the “len” cannot exceed (the end address of the page - “addr” + 1). For example, if the “addr” is 0x120f0, the “len” cannot exceed (0x120ff - 0x120f0 + 1)=16.

`flash_write_page(0x120f0, 20, data)` // wrong, 20 bytes exceed the maximum allowed length “16”, the first 16 bytes belong to the page starting from 0x12000, but the last 4 bytes belong to the page starting from 0x12100.

For the “`flash_read_page`” function, one operation can read data more than 256 bytes, i.e. it’s allowed to read flash area across pages in this function.

2) flash erase operation

The function “`flash_erase_sector`” serves to erase flash.

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

One sector contains 4096 bytes, e.g. 0x13000 ~0x13fff.

The “addr” must be the starting address of one sector, and each erase operation erases a complete sector.

In the case of 16M system clock, it takes 30~100ms or even longer time to erase a sector.

3) Influence on system interrupt caused by flash access/erase operation

System interrupt must be disabled via “`irq_disable()`” when the flash access or erase function is executed, and then restored via “`irq_restore()`” after the operation is finished. This will ensure integrity and continuity of flash MSPI timing operation, and avoid hardware resource reentry due to MSPI bus lines invoking by flash operation in interrupt.

Since timing sequence of BLE SDK RF packet transmission and reception is always controlled by interrupt, when system interrupt is disabled during flash operation, it may ruin the timing sequence, thus MCU fails to respond in time.

The influence on BLE interrupt by execution time of the flash access function is almost negligible; however, the “len” in the function will determine the time to access the flash area, so it’s highly recommended not to set the “len” as large value in BLE connection state during main_loop.

It takes tens of milliseconds to hundreds of milliseconds to execute the “`flash_erase_sector`” function. Therefore, during main_loop of main program, once MCU enters BLE connection state, it’s not allowed to invoke the “`flash_erase_sector`” to avoid disconnection. If it’s inevitable to erase flash during BLE connection, BLE timing sequence protection as introduced in **3.2.4.7 Conn state Slave role timing protection** should be adopted.

4) Read flash via pointer

Firmware of BLE SDK is stored in Flash. When the firmware is running, only top portion of the code in Flash (memory resident) is stored and executed in RAM, and the majority will be transferred to the high-speed “Cache” of RAM from Flash when needed. MCU will automatically control internal MSPI hardware module to read Flash.

Flash can also be read via pointer: When data are accessed by MCU system bus, if the data address is not in the memory resident ramcode, system bus will automatically switch to MSPI, and read data from flash by using MSCN, MCLK, MSDI and MSDO lines to operate SPI timing sequence.

Following shows three examples:

```
u16 x = *(volatile u16*)0x10000; //read two bytes from flash 0x10000
u8 data[16];
memcpy(data, 0x20000, 16);      //read 16 bytes from flash 0x20000 and copy
to data
if(!memcmp(data, 0x30000, 16)){ // read 16 bytes from flash 0x30000 and
compare with data
//.....
}
```

In user_init, when calibration values are read from flash and set to corresponding registers, the reading is implemented via pointer. Please refer to the function below in the SDK:

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

Flash can be read by using the function “flash_read_page” or pointer, but it can be written via the “flash_write_page” function only. Pointer access is not supported for Flash writing operation.

*Note: When flash is read by using pointer, since data read by system bus will be buffered in cache, MCU may directly use the buffered data as the result of the new reading operation.

Example:

```
u8 result;
result = *(volatile u16*)0x40000; //read flash via pointer
```

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

The original data in flash 0x40000 is 0xff; the result of the first reading operation is 0xff; then 0x5A is written into flash 0x40000 by the following writing operation; in theory, the result of the second reading operation should be the new value “0x5A”, but the actual result is still the old data buffered in the cache, i.e. “0xff”.

Therefore, in the case of multiple reading of the same address, if its value will be modified, use the API “flash_read_page” rather than pointer, to ensure the result of reading operation is the new value written into this address rather than the old value in the cache.

The following format is correct:

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

2.1.4 SDK flash space partition

Flash uses a sector (4K bytes) as unit to store and erase information (Note: Erase function is “flash_erase_sector”). In theory, information of the same type should be stored in a sector, and different information types should be stored in different sectors to avoid unexpected erasing. It’s recommended for user to follow this rule to store customized information in Flash.

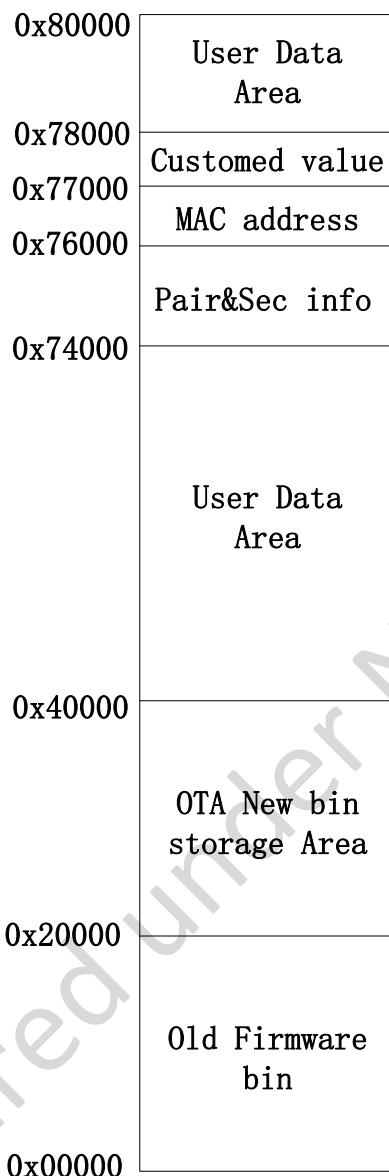


Figure 2-6 512kB flash space partition

The figure above shows the default address allocation for the 512kB flash of 8x5x. Corresponding interfaces are supplied for user to modify flash address allocation.

1. The sector from 0x76000 to 0x76fff serves to store MAC address. Actually the 6-byte MAC address is stored in flash area from 0x76000 (for lower byte of MAC address) to 0x76005 (for higher byte of MAC address). For example, if “0x11 0x22 0x33 0x44 0x55 0x66” are stored in FLASH 0x76000~0x76005, the MAC address is “0x665544332211”.

Corresponding to the SDK, MAC address of actual product will be downloaded into its flash starting from 0x76000 by Telink jig system. If it's needed to modify

this starting address to store MAC address, user should ensure the consistency.

The “user_init” function in the SDK will read MAC address from flash area starting from the macro “CFG_ADR_MAC”. This macro is modifiable in the “stack/ble/blt_config.h”.

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

2. The sector from 0x77000 to 0x77fff serves store customized calibration information for Telink MCU. Only this sector does not follow the rule that puts different information types into different sectors; the 4096 bytes in this sector are divided into 64 units with 64 bytes each, and each unit stores one type of calibration information. Since calibration information are burned to corresponding addresses by jig, they can be stored in the same sector; when firmware is running, these calibration information are read only and they're not allowed to be written or erased.
 - 1) The first 64-byte unit serves to store frequency offset calibration information. Actually this calibration value is 1 byte stored in 0x77000.
 - 2) The second 64-byte unit was used by 826x family to store calibration value of TP value. 8x5x does not need TP calibration anymore, but still reserves the design of this unit.
 - 3) The third 64-byte unit serves to store capacitance calibration value of external 32kHz crystal.

Following units are reserved for other potential calibration values.

3. The two sectors 0x74000 ~ 0x75FFF are occupied by BLE stack system, and the 8kB area is used to store pairing and security information. User can modify the starting address of this 8kB area to store pairing and security information by invoking the function below:

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

4. The 256kB area 0x00000 ~ 0x3FFFF is used as program space by default:
 - ❖ The first 128kB area 0x00000 ~ 0x1FFFF is used as storage space for old firmware.
 - ❖ The second 128kB area 0x20000 ~ 0x3FFFF is used as storage space for OTA new firmware.
 - ❖ If firmware doesn't need to occupy the whole 128kB space 0x00000 ~

0x3FFFF, user can use corresponding API to modify the allocation as needed, thus the remaining space can be used as data storage space. Please refer to **section 7.1.3 Modify FW size and booting address** for details.

5. The remaining flash space are all used as user data area (storage space for user data).

2.2 Clock module

2.2.1 System clock & System Timer

MCU program runs on system clock.

System Timer is a read only timer which mainly supplies time reference for BLE timing control. User can also use this resource.

In previous 826x family, System Timer is derived from system clock.

For 8x5x family, System Timer is separate from system clock. As shown in the figure below, the 16M System Timer is derived from external 24M Crystal Oscillator via 3/2 Divider.

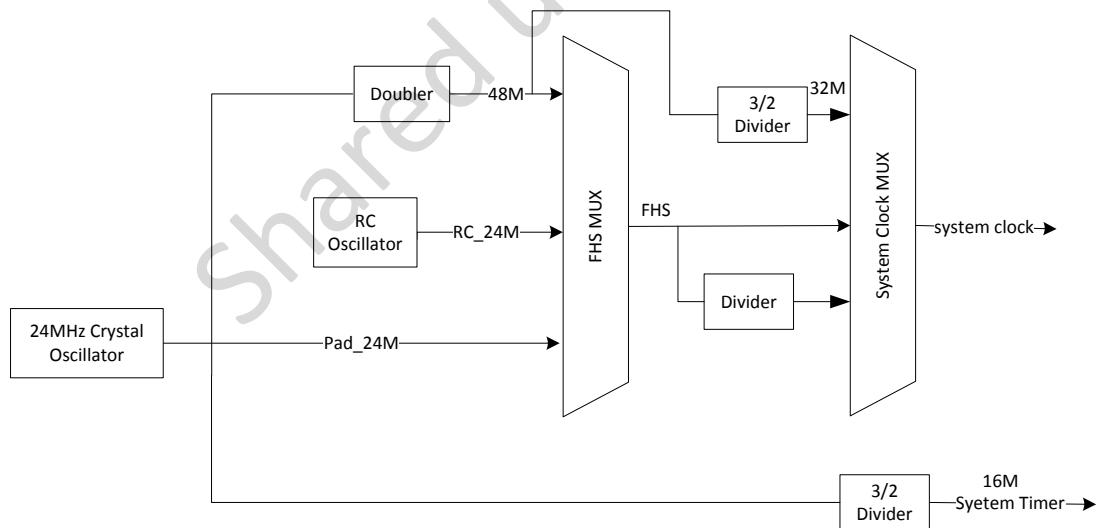


Figure 2-7 system clock & System Timer

System clock can be either derived from external 24M crystal, doubled through “Doubler” to 48M, then divided into 16M, 24M, 32M, or go straight out as 48M. This type is called crystal clock (e.g. 16M crystal system clock, 24M crystal system clock); or derived from internal 24M RC Oscillitor to get RC clock (e.g. 24M RC clock, 32M RC clock, 48M RC clock). Note that current BLE SDK does not support the RC clock.

Crystal clock is recommended in the BLE SDK.

The API below serves to initialize system clock by selecting the required clock for the enum variable “SYS_CLK_TYPEDEF”.

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

For 8x5x, since System Timer is different from system clock, user needs to know whether clock source for each module on MCU is system clock or System Timer. For example, suppose system clock is 24M crystal, system clock is 24M, while System Timer is 16M.

Following shows definition for system clock and 1S, 1mS, 1uS in the “app_config.h”.

```
#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),
};
```

All hardware modules driven by system clock can only select clock source above (e.g. CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_1S). In other words, if user finds clock sources above in a certain module, this module is driven by system clock.

In the example below, period and duty cycle of PWM is driven by CLOCK_SYS_CLOCK_1US, so PWM is driven by system clock.

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

System Timer is fixed as 16M. For System Timer, following code in the SDK represents S, mS, and 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,
};
```

All of the APIs below in the SDK contain System Timer related operations, so these APIs should use “CLOCK_16M_SYS_TIMER_CLK_xxx” to represent time.

```
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)
```

Since System Timer is BLE timing reference, all BLE timing related parameters and variables should use “*CLOCK_16M_SYS_TIMER_CLK_xxx*” to represent time.

2.2.2 System Timer usage

After `cpu_wakeup_init` initialization in the main function, System Timer starts working. User can read System Timer counter value (System Timer tick).

The 32-bit System Timer tick value will be increased by 1 for each clock cycle (i.e. 1/16us). It takes 268 seconds or so (i.e. (1/16) us * (2^32)) for the System Timer tick to loop from the initial value 0x0 to the maximum value 0xffffffff.

The system tick won't stop counting during firmware running process.

The function “`clock_time()`” serves to read System Timer tick value.

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

In this BLE SDK, the whole BLE timing sequence is based on System Timer tick. It's highly recommended for user to follow the common usage in firmware, i.e. use System Timer tick to implement simple software timer and timeout judgment.

The software timer based on query mechanism generally applies to applications without high real-time and small error requirement. The usage of the software timer is shown as below:

- 1) Start timing: Set an u32 variable, read and record current System Timer tick.

```
u32 start_tick = clock_time(); // clock_time() returns System Timer tick value
```

- 2) At somewhere of the firmware, continuously query and compare (current System Timer tick - start_tick) with pre-determined timer value. If the difference exceeds the timer value, the timer is triggered to execute corresponding operation, and clear timer or start a new timing cycle as needed.

Suppose timer value is 100ms, the following sentence can be used to query the timer:

```
if (u32) (clock_time() - start_tick) > 100 * CLOCK_16M_SYS_TIMER_CLK_1MS
```

The difference value is switched to u32 type considering the case when system clock tick counts from 0xffffffff to 0.

In the SDK, an unified invoking function is provided irrespective of system clock frequency:

```
if( clock_time_exceed(start_tick,100 * 1000)) // The unit of the second parameter is us
```

*Note: For 16M clock, this function only applies to timing within 268s, if exceeds, it's needed to add timer correspondingly.

Application example: If condition A is triggered (only once), after 2 seconds, B() operation is executed.

```
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 module

For details about GPIO module, please refer to source code in “drivers/8258/gpio_8258.h”, “gpio_default_8258.h” and “gpio_8258.c”.

To understand register operation, please refer to 8x5x “pio_lookuptable”.

2.3.1 GPIO definition

Telink 8258 supports 36 GPIOs divided into 5 groups, including:

GPIO_PA0 - GPIO_PA7, GPIO_PB0 - GPIO_PB7, GPIO_PC0 - GPIO_PC7
GPIO_PD0 - GPIO_PD7, GPIO_PE0 - GPIO_PE3

Note: Not all of the 36 GPIOs in IC core have corresponding external pins in actual IC packages. Please refer to the corresponding pin layout.

Please follow the format above to use GPIOs, and refer to the “drivers/8258/gpio_8258.h” for details.

There are 7 special GPIOs:

- 1) MSPI pins: The four GPIOs are dedicated for Flash memory access and correspond to Master SPI bus lines. They are used as SPI function by default, and it's forbidden to use them as GPIO function or operate them in firmware. For 8x5x, MSPI pins are PE0~ PE3.
- 2) SWS (Single Wire Slave): It's used as SWS function by default for debugging and firmware burning. Generally it is not used in firmware. For 8x5x, SWS pin is PA7.
- 3) DM and DP: They are used as GPIO function by default. If USB function is needed, the two pins should be set as USB DM and DP function; otherwise the two pins can be used as GPIO function. For 8x5x, DM and DP pins are PA5 and PA6.

2.3.2 GPIO state control

In this section only the basic GPIO states are listed.

- 1) func: Configure pin as special function or general GPIO. To use input/output function, the pin should be configured as general GPIO.

```
void gpio_set_func(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func);
```

“pin” always means GPIO definition (e.g. GPIO_PA0).

“func” can be configured as “AS_GPIO” or other special function.

- 2) ie: Input enable.

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

“value”: 1-enable, 0-disable.

- 3) datai: Data input. When input is enabled for a GPIO pin, the datai value indicates its current input level.

```
unsigned int gpio_read(GPIO_PinTypeDef pin);
```

Note: If GPIO input is low level, 0 is returned; if GPIO input is high level, non-zero value (**may not be 1**) is returned.

In firmware, it's recommended to invert the read values rather than use the format such as "if(gpio_read(GPIO_PA0) == 1)". Inverted values will be either 1 or 0.

```
if( !gpio_read(GPIO_PA0) ) // high/low level judgment
```

- 4) oe(output enable): output enable

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

"value": 1-enable, 0-disable.

- 5) dataO (data output): When output is enabled, this value should be 1 to output high voltage or 0 to output low voltage.

```
void gpio_write(GPIO_PinTypeDef pin, unsigned int value)
```

- 6) Internal analog pull-up/pull-down resistor: Configurable as 1M pull-up, 10K pull-up, 100K pull-down or float.

```
void gpio_setup_up_down_resistor( GPIO_PinTypeDef gpio,
                                  GPIO_PullTypeDef up_down);
```

"up_down" is configurable as shown below:

```
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;
```

In deepsleep or deepsleep retention, GPIO input and output state are all invalid except for pullup or pulldown resistors.

GPIO configuration examples:

- 1) Configure GPIO_PA4 as high level output

```
gpio_set_func(GPIO_PA4, AS_GPIO); // PA4 is GPIO by default, can be skipped
gpio_set_input_en(GPIO_PA4, 0);
gpio_set_output_en(GPIO_PA4, 1);
```

- ```
gpio_write(GPIO_PA4,1)

2) Configure GPIO_PC6 as input, and check whether it's low-level input. Enable 10K
pull up resistor to avoid influence of float level.

 gpio_set_func(GPIO_PC6, AS_GPIO); // PC6 is GPIO by default, can be skipped
 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)){ //judge if low voltage

 }

3) Configure PA5 and PA6 as USB DM and DP

 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 initialization

The “gpio\_init” function is invoked in the main.c file to initialize states of all GPIOs except 4 MSPI GPIOs. Each of the 32 GPIOs will be initialized to its default states by the “gpio\_init” function, unless related GPIO parameters are pre-configured in the app\_config.h.

Default states for the 32 GPIOs:

- 1) func  
Except SWS, other GPIOs are generic GPIO function.
- 2) ie  
For SWS, default ie is 1; for other GPIOs, default ie is 0.
- 3) oe  
For all GPIOs, default oe is 0.
- 4) dataO  
For all GPIOs, default dataO is 0.
- 5) Internal pull up/down resistor  
For all GPIOs, default internal pull up/down resistor is float.

Please refer to drivers/8258/ gpio\_8258.h, drivers/8258/ gpio\_default\_8258.h for more information.

If one or multiple GPIOs are configured in the app\_config.h, the gpio\_init will use the value specified in the app\_config.h instead of default value. The reason is that GPIO default states are all defined by macro.

Macro example:

```
#ifndef PA0_INPUT_ENABLE
#define PA0_INPUT_ENABLE 1
#endif
```

Once these macros are pre-defined in the app\_config, these macros will not use default values.

Following takes PA0 as an example to illustrate GPIO state configuration in the app\_config.h.

- 1) configure func: #define PA0\_FUNC AS\_GPIO
- 2) configure ie: #define PA0\_INPUT\_ENABLE 1
- 3) configure oe: #define PA0\_OUTPUT\_ENABLE 0
- 4) configure dataO: #define PA0\_DATA\_OUT 0
- 5) configure internal pull up/down resistor:  

```
#define PULL_WAKEUP_SRC_PA0 PM_PIN_UP_DOWN_FLOAT
```

Summary for GPIO initialization:

- 1) User can pre-define GPIO initial states in the app\_config.h, and initialize corresponding GPIOs to the configured values by the gpio\_init;
- 2) User can also set the GPIO states by the GPIO state control function (e.g. gpio\_set\_input\_en) in the user\_init;
- 3) The two methods above can be combined to configure the GPIO state.

Note that if some state of one GPIO is configured to different values by the app\_config.h and user\_init, the configuration in the user\_init will take effect finally according to firmware timing sequence.

Following shows the implementation for the gpio\_init function. The value of anaRes\_init\_en determines whether internal pull up/down resistor is configured.

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

As explained later in Low Power Management, registers for GPIO pull up/down control are non-volatile during deepsleep retention, so GPIO pull up/down states can be retained across deepsleep retention.

To ensure GPIO pull up/down states are not changed after deepsleep retention wakeup, the gpio\_init needs to check whether it's wake\_up from deepsleep retention and set the anaRes\_init\_en accordingly.

```
int deepRetWakeUp = pm_is MCU_deepRetentionWakeup();
gpio_init(!deepRetWakeUp);
```

#### 2.3.4 Volatile GPIO digital states over deepsleep retention mode

Except analog pull up/down resistors are controlled by analog registers, all other states (func, ie, oe, dataO, etc) are controlled by digital registers.

As to be explained later, all digital registers will lose their values over deepsleep retention.

In Telink 826x family, during suspend gpio output can be used to control some peripheral devices.

However, in 8x5x, if suspend is switched to deepsleep retention mode, gpio output becomes invalid, and thus cannot be used to control external devices. Instead, GPIO analog pull up/down resistors can be used to serve the same purpose, i.e., pull up 10K corresponds to gpio output high, pull down 100K corresponds to gpio output low.

Note: During deepsleep retention, DO NOT use pull up 1M (pull up voltage may be slightly lower than VCC). In addition, DO NOT use pull up 10K on PC0~PC7, since there will be a transient ripple/spike at deepsleep retention wake\_up; for other GPIOs, pull up 10K is all right.

### 2.3.5 Configure SWS to avoid MCU error

Telink MCU uses the SWS (Single Wire Slave) pin for debugging and firmware burning. In final application code, the state of SWS is shown as below:

- 1) Set as SWS function rather than general GPIO.
- 2) ie =1: set as “input enable” so as to receive commands from EVK to operate MCU.
- 3) Both “oe” and “dataO” are set as 0.

The settings above may bring a risk: since SWS is in float state, large jitter of system power (e.g. transient current may approach 100mA when IR command is sent) may lead to incorrect command reception and firmware malfunction.

By enabling internal 1M pull-up resistor for SWS to replace its float state, this problem can be solved.

In 8x5x, SWS is multiplexed with GPIO\_PA7, and 1M pull up on PA7 should be enabled in the drivers/8258/gpio\_default\_8258.h.

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

### 3 BLE Module

#### 3.1 BLE SDK software architecture

##### 3.1.1 Standard BLE SDK architecture

Figure3-1 shows standard BLE SDK software architecture compliant with BLE spec.

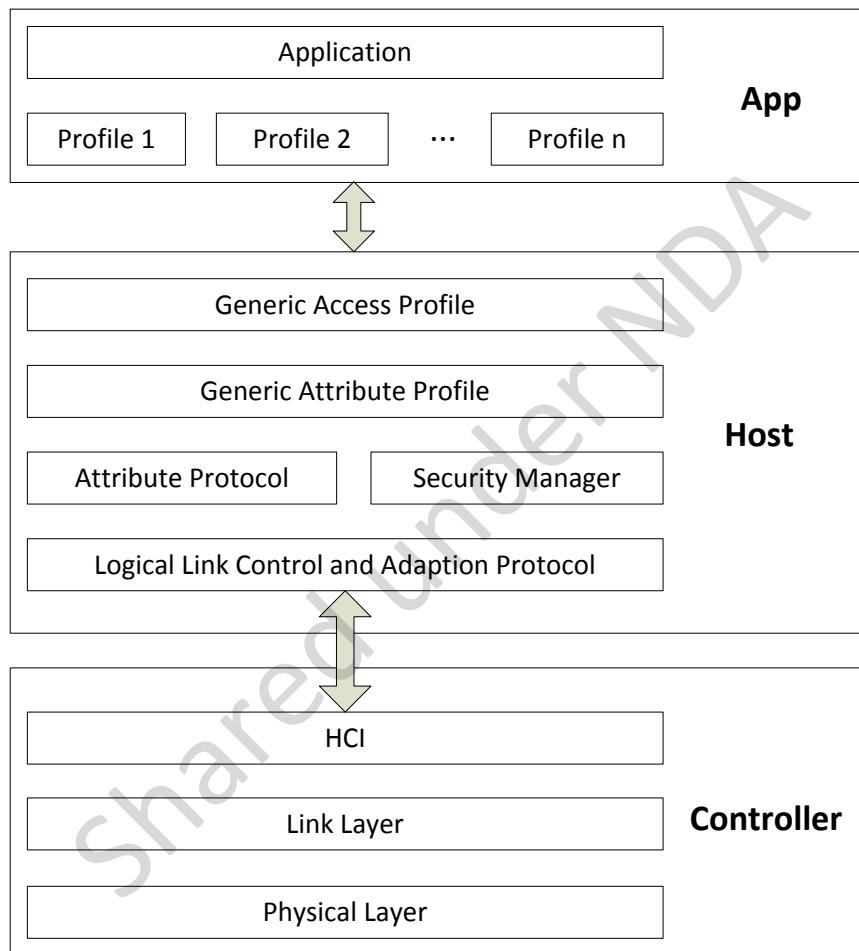


Figure3-1 BLE SDK standard architecture

As shown above, BLE protocol stack includes Host and Controller.

- ❖ As BLE bottom-layer protocol, the “Controller” contains Physical Layer (PHY) and Link Layer (LL). Host Controller Interface (HCI) is the sole communication interface for all data transfer between Controller and Host.
- ❖ As BLE upper-layer protocol, the “Host” contains protocols including Logic Link Control and Adaption Protocol (L2CAP), Attribute Protocol (ATT), Security Manager Protocol (SMP), as well as Profiles including Generic Access Profile (GAP) and Generic Attribute Profile (GATT).

- ✧ The “Application” (APP) layer contains user application codes and Profiles corresponding to various Services. User controls and accesses Host via “GAP”, while Host transfers data with Controller via “HCI”.

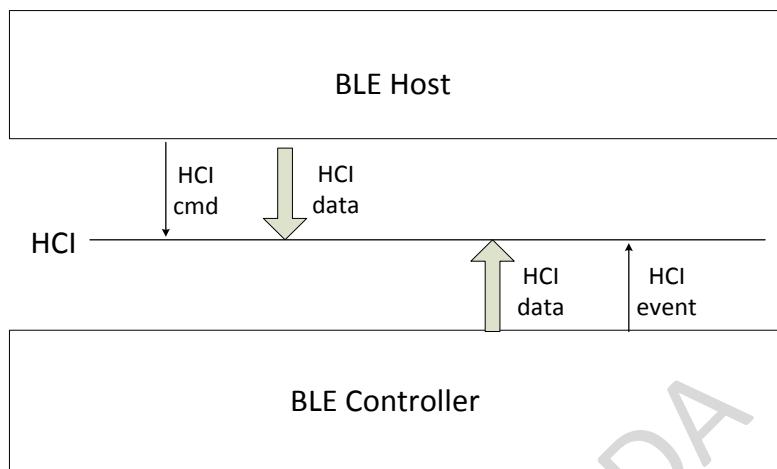


Figure3-2 HCI data transfer between Host and Controller

- 1) BLE Host will use HCI cmd to operate and set Controller. Controller API corresponding to each HCI cmd will be introduced in this chapter.
- 2) Controller will report various HCI events to Host via HCI.
- 3) Host will send target data to Controller via HCI, while Controller will directly load data to Physical Layer for transfer.
- 4) When Controller receives RF data in Physical Layer, it will first check whether the data belong to Link Layer or Host, and then process correspondingly: If the data belong to LL, the data will be processed directly; if the data belong to Host, the data will be sent to Host via HCI.

### 3.1.2 Telink BLE SDK architecture

#### 3.1.2.1 Telink BLE controller

Telink BLE SDK supports standard BLE Controller, including HCI, PHY (Physical Layer) and LL (Link Layer).

Telink BLE SDK contains five standard states of Link Layer (standby, advertising, scanning, initiating, and connection), and supports Slave role and Master role in connection state. Currently both Slave role and Master role only support single connection, i.e. LL can only sustain single connection, concurrent existence of multiple Slave/Master or Slave and Master is not supported.

In the SDK, 8258 HCI is a Controller of BLE Slave; to form a standard BLE Slave system, another BLE Host MCU is needed.

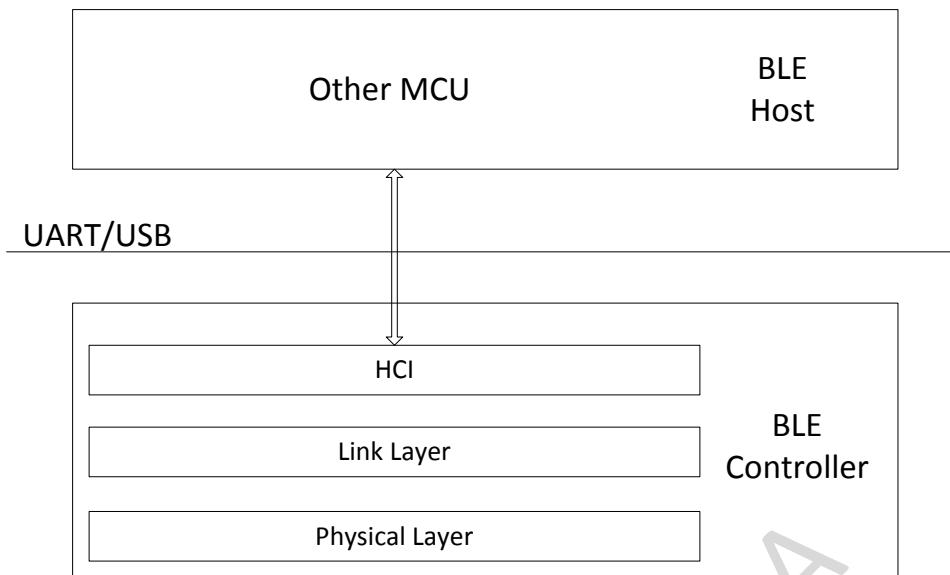


Figure3-3 8258 HCI architecture

Link Layer connection state supports Slave and Master of single connection, thus 8258 HCI can also be used as BLE Master Controller actually. However, when a BLE Host is running in a complex system (Linux/Android), Master Controller of single connection can only connect with a single device, which is almost meaningless. Therefore, the SDK does not include Master role initialization in 8258 HCI.

### 3.1.2.2 Telink BLE Slave

Telink BLE SDK in BLE Host fully supports stack of Slave; for Master with complex SDP (Service Discovery), it's not fully supported yet.

When user only needs to use standard BLE Slave, and Telink BLE SDK runs Host (Slave part) + standard Controller, the actual stack architecture will be simplified based on the standard architecture, so as to minimize system resource consumption of the whole SDK (including SRAM, running time, power consumption, and etc.). Following shows Telink BLE Slave architecture. In the SDK, 8258 ble sample, 8258 remote and 8258 module are all based on this architecture.

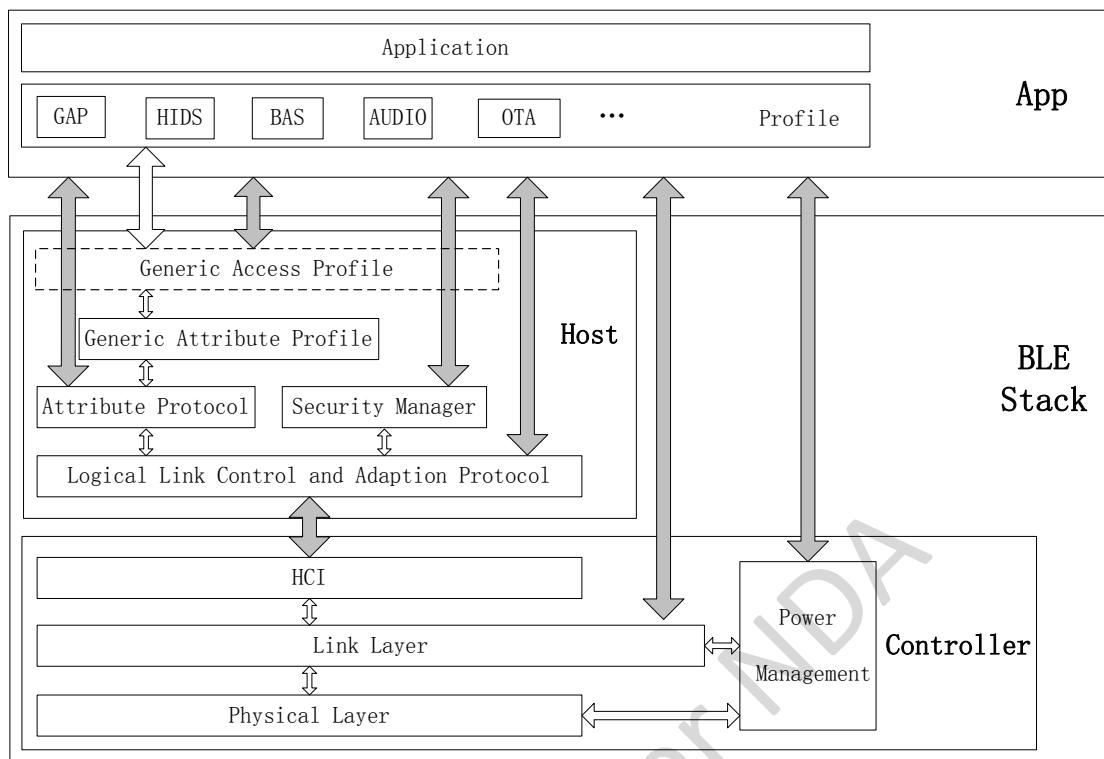


Figure3-4 Telink BLE Slave Architecture

In Figure3-4, solid arrows indicate data transfer controllable via user APIs, while hollow arrows indicate data transfer within the protocol stack not involved in user.

Controller can still communicate with Host (L2CAP layer) via HCI; however, the HCI is no longer the sole interface, and the APP layer can directly exchange data with Link Layer of the Controller. Power Management (PM) Module is embedded in the Link Layer, and the APP layer can invoke related PM interfaces to set power management.

Considering efficiency, data transfer between the APP layer and the Host is not controlled via GAP; the ATT, SMP and L2CAP can directly communicate with the APP layer via corresponding interface. However, the event of the Host should be communicated with the APP layer via the GAP layer.

Generic Attribute Profile (GATT) is implemented in the Host layer based on Attribute Protocol. Various Profiles and Services can be defined in the APP layer based on GATT. Basic Profiles including HIDS, BAS, AUDIO and OTA are provided in demo code of this BLE SDK.

Following sections explain each layer of the 8x5x BLE stack according to the structure above, as well as user APIs for each layer.

Physical Layer is totally controlled by Link Layer, since it does not involve the APP layer, it will not be covered in this document.

Though HCI still implements part of data transfer between Host and Controller, it is basically implemented by the protocol stack of Host and Controller with little involvement of the APP layer. User only needs to register HCI data callback handling function in the L2CAP layer.

### 3.1.2.3 Telink BLE master

Implementation of Telink BLE Master is different from that of Slave: Standard Controller is supplied in the SDK and assembled in library, while the APP layer implements Host and user application.

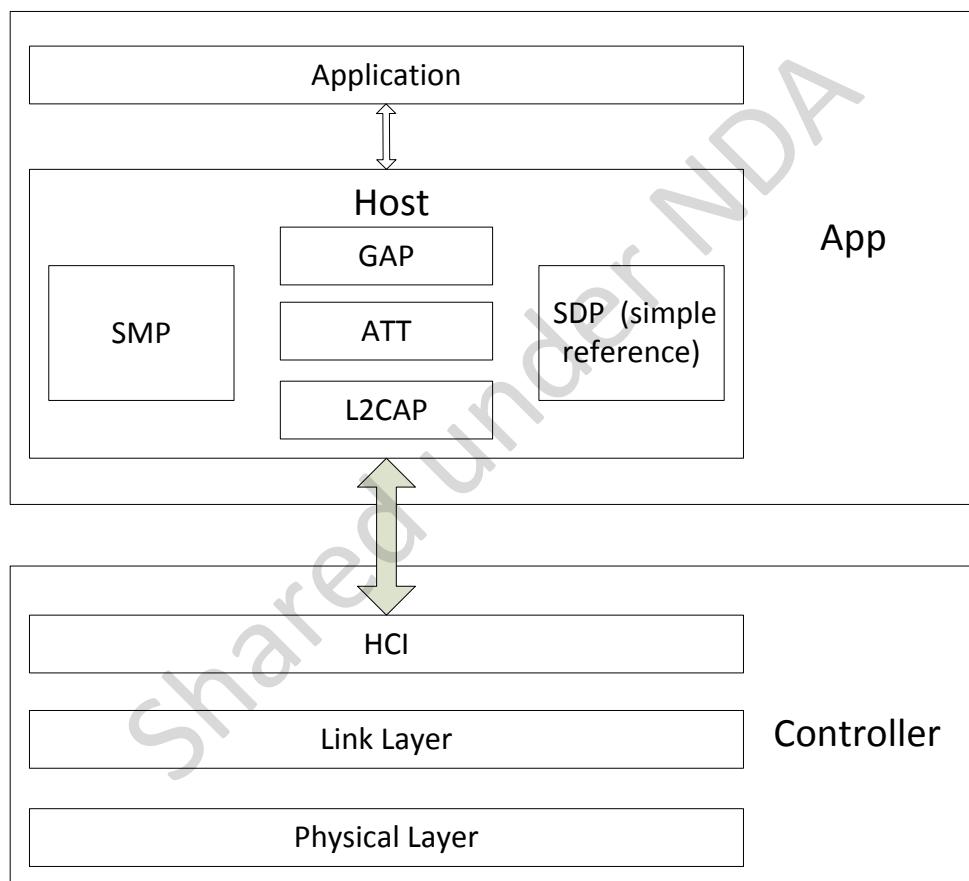


Figure3-5 Telink BLE Master Architecture

In the SDK, demo code of the “8258 master kma dongle” project is implemented based on this architecture. Almost all coding of Host layer is implemented in APP, and the SDK also supplies various standard interfaces for user to use these functions.

Standard l2cap and att processing are implemented in APP layer, while the SMP only supplies the basic “just work” method. In the “8258 master kma dongle”, SMP is disabled by default, so user needs to enable the corresponding macro to enable SMP. Since SMP implementation is complex, the code is assembled in the library, and the APP layer only needs to invoke related interface. User can search for the

corresponding code via the keyword “BLE\_HOST\_SMP\_ENABLE”.

```
#define BLE_HOST_SMP_ENABLE 0
//1 for standard security management,
// 0 for telink referenced paring&bonding(no security)
```

Telink BLE Master does not supply standard implementation for the most complex SDP part, but only gives a simple reference: service discovery of 8258 remote. In the “8258 master kma dongle”, this simple reference SDP is enabled by default.

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

In the SDK, standard interfaces are supplied for all ATT operations related to service discovery. User can refer to service discovery of 8258 remote to implement his own service discovery, or disable the “BLE\_HOST\_SIMPLE\_SDPA\_ENABLE”, and use the service ATT handle agreed by Slave to implement data access.

Since suspend processing is not included for scanning and connection master role of Link Layer, Telink BLE Master does not support Power Management.

## 3.2 BLE controller

### 3.2.1 BLE controller introduction

BLE Controller contains Physical Layer, Link Layer, HCI and Power Management.

Telink BLE SDK fully assembles Physical Layer in the library (corresponding to c file of rf\_drv.h in driver file), and user does not need to learn about it. Power Management will be introduced in detail in **section 4 Low Power Management (PM)**.

This section will focus on Link Layer, and also introduce HCI related interfaces to operate Link Layer and obtain data of Link Layer.

### 3.2.2 Link Layer state machine

Figure3-6 shows Link Layer state machine in BLE spec. Please refer to “Core\_v5.0” (Vol 6/Part B/1.1 “LINK LAYER STATES”) for more information.

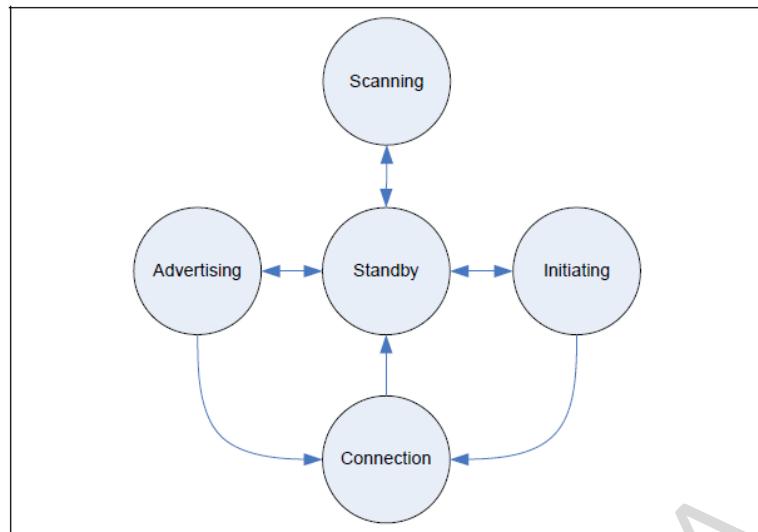


Figure3-6 State diagram of Link Layer state machine in BLE Spec

Telink BLE SDK Link Layer state machine is shown as below.

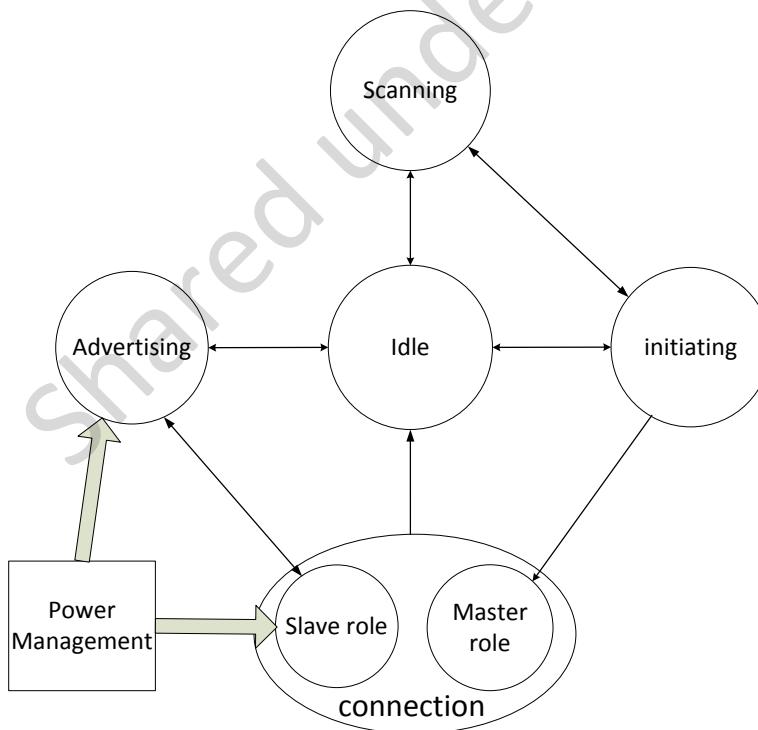


Figure3-7 Telink Link Layer state machine

Telink BLE SDK Link Layer state machine is consistent with BLE spec, and it contains five basic states: Idle (Standby), Scanning, Advertising, Initiating, and Connection. Connection state contains Slave Role and Master Role.

As introduced above, currently both Slave Role and Master Role design are based on single connection. Slave Role is single connection by default; while Master Role is marked as “Master role single connection”, so as to differentiate from “Master Role multi connection” which will be supported in the future.

In this document, Slave Role will be marked as “Conn state Slave role” or “ConnSlaveRole/Connection Slave Role”, or “ConnSlaveRole” in brief; while Master Role will be marked as “Conn state Master role” or “ConnMasterRole/Connection Master Role”, or “ConnMasterRole” in brief.

“Power Management” in Figure3-7 is not a state of LL, but a functional module which indicates the SDK only implements low power processing for Advertising and Connection Slave Role. If Idle state needs low power, user can invoke related APIs in the APP layer. For the other states, the SDK does not contain low power management, and user cannot implement low power in the APP layer.

Based on the five states above, corresponding state machine names are defined in the “stack/ble/ll/ll.h”. “ConnSlaveRole” and “ConnMasterRole” correspond to state name “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)
```

Switch of Link Layer state machine is automatically implemented in BLE stack bottom layer. Therefore, user cannot modify state in APP layer, but can obtain current state by invoking the API below. The return value will be one of the five states.

```
u8 b1c_ll_getCurrentState(void);
```

### 3.2.3 Link Layer state machine combined application

#### 3.2.3.1 Link Layer state machine initialization

Telink BLE SDK Link Layer fully supports all states; however, it's flexible in design. Each state can be assembled as a module; be default there's only the basic Idle module, and user needs to add modules and establish state machine combination for his application.

For example, for BLE Slave application, user needs to add Advertising module and ConnSlaveRole, while the remaining Scanning/Initiating modules are not

included so as to save code size and ramcode. The code of unused states won't be compiled.

The API below is used for MCU initialization. This API is necessary for all BLE applications.

```
void b1c_ll_initBasicMCU (void);
```

The API below serves to add the basic Idle module. This API is also necessary for all BLE applications.

```
void b1c_ll_initStandby_module (u8 *public_adr);
```

Following are initialization APIs of modules corresponding to the other states (Scanning, Advertising, Initiating, Slave Role, Master Role Single Connection).

```
void b1c_ll_initAdvertising_module (u8 *public_adr);
void b1c_ll_initScanning_module (u8 *public_adr);
void b1c_ll_initInitiating_module (void);
void b1c_ll_initConnection_module (void);
void b1c_ll_initSlaveRole_module (void);
void b1c_ll_initMasterRoleSingleConn_module (void);
```

The actual parameter "public\_adr" is the pointer of BLE public mac address.

```
void b1c_ll_initConnection_module (void);
```

Used to initialize the common module share between master and slave mode.

User can flexibly establish Link Layer state machine combination by using the APIs above. Following shows some common combination methods as well as corresponding application scenes.

### 3.2.3.2 Idle + Advtersing

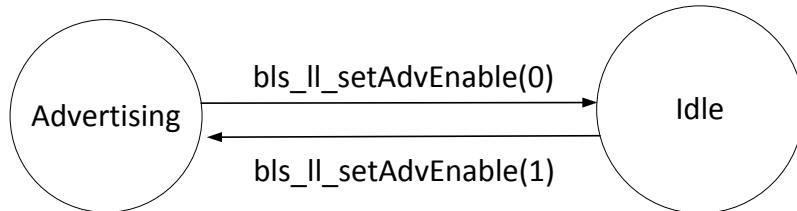


Figure3-8 Idle + Advertising

As shown above, only Idle and Advertising module are initialized, and it applies to applications which use basic advertising function to advertise product information in single direction, e.g. beacon.

Following is module initialization code of Link Layer state machine.

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

State switch of Idle and Advertising is implemented via the “bls\_ll\_setAdvEnable”.

### 3.2.3.3 Idle + Scanning

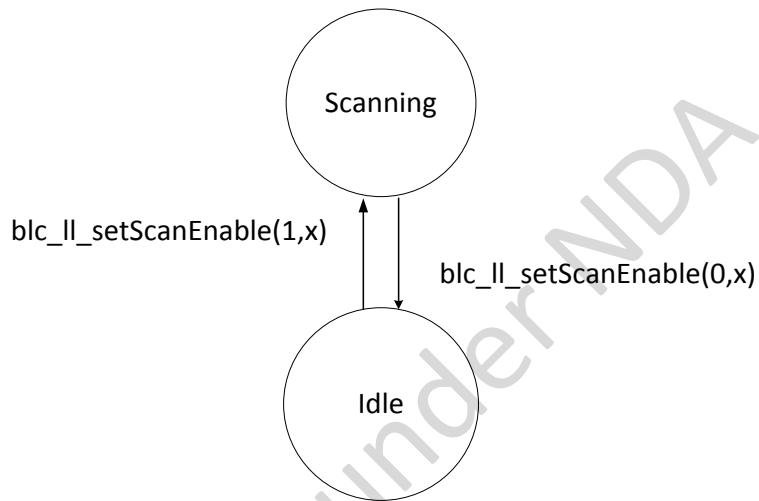


Figure3-9 Idle + Scanning

As shown above, only Idle and Scanning module are initialized, and it applies to applications which use basic scanning function to implement scanning and discovery of product advertising information, e.g. beacon.

Following is module initialization code of Link Layer state machine.

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

State switch of Idle and Scanning is implemented via the “blc\_ll\_setScanEnable”.

### 3.2.3.4 Idle + Advtersing + ConnSlaveRole

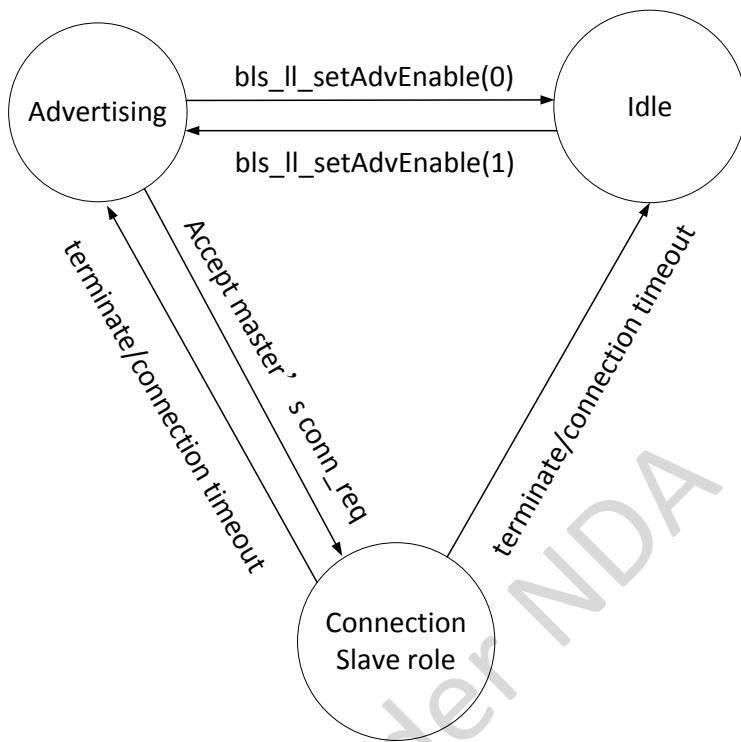


Figure3-10 BLE Slave LL state

The figure above shows a Link Layer state machine combination for a basic BLE Slave application. In the SDK, 8258 hci/8258 ble sample/8258 remote/8258 module are all based on this combination.

Following is module initialization code of Link Layer state machine.

```

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();
```

State switch in this combination is shown as below:

- 1) After power on, 8x5x MCU enters Idle state. In Idle state, when adv is enabled, Link Layer switches to Advertising state; when adv is disabled, it will return to Idle state.

The API “bls\_ll\_setAdvEnable” serves to enable/disable Adv.

After power on, Link Layer is in Idle state by default. Typically it's needed to enable Adv in the “user\_init” so as to enter Advertising state.

- 2) When Link Layer is in Idle state, Physical Layer won't take any RF operation including packet transmission and reception.
- 3) When Link Layer is in Advertising state, advertising packets are transmitted in adv channels. Master will send connection request if it receives adv packet. After Link Layer receives this connection request, it will respond, establish connection and enter ConnSlaveRole.
- 4) When Link Layer is in ConnSlaveRole, it will return to Idle State or Advertising state in any of the following cases:
  - a) Master sends "terminate" command to Slave and requests disconnection. Slave will exit ConnSlaveRole after it receives this command.
  - b) By sending "terminate" command to Master, Slave actively terminates the connection and exits ConnSlaveRole.
  - c) If Slave fails to receive any packet due to Slave RF Rx abnormality or Master Tx abnormality until BLE connection supervision timeout is triggered, Slave will exit ConnSlaveRole.

When Link Layer exits ConnSlaveRole state, it will switch to Adv or Idle state according to whether Adv is enabled or disabled which depends on the value configured during last invoking of the "bls\_ll\_setAdvEnable" in APP layer.

- ❖ If Adv is enabled, Link Layer returns to Advertising state.
- ❖ If Adv is disabled, Link Layer returns to Idle state.

### 3.2.3.5 Idle + Scanning + Initiating + ConnMasterRole

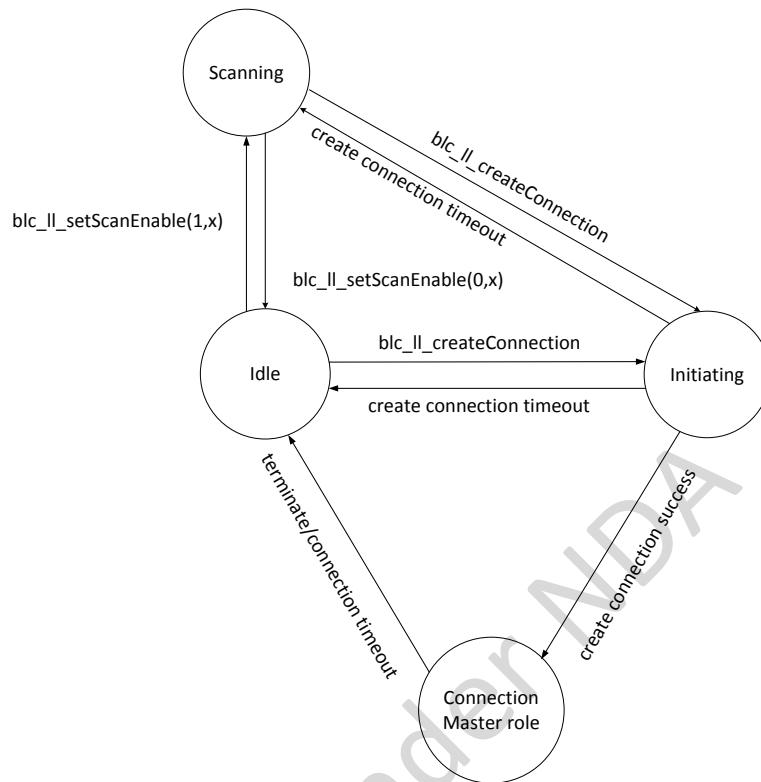


Figure3-11 BLE Master LL state

The figure above shows a Link Layer state machine combination for a basic BLE Master application. In the SDK, 8258 master kma dongle is based on this combination.

Following is module initialization code of Link Layer state machine.

```

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();

```

State switch in this combination is shown as below:

- 1) After power on, 8x5x MCU enters Idle state. In Idle state, when scan is enabled, Link Layer switches to Scanning State; in Scanning State, when scan is disabled, it will return to Idle state.

The API “blc\_ll\_setScanEnable” serves to enable/disable scan.

After power on, Link Layer is in Idle state by default. Typically it's needed to enable Scan in the “user\_init” so as to enter Scanning state.

When Link Layer is in Scanning state, the scanned adv packet will be reported to BLE Host via the event “HCI\_SUB\_EVT\_LE\_ADVERTISING\_REPORT”.

- 2) In Idle and Scanning state, Link Layer can be triggered to enter Initiating state via the API “blc\_ll\_createConnection”.

The “blc\_ll\_createConnection” specifies MAC address of one or multiple BLE devices to be connected.

After Link Layer enters Initiating state, it will continuously scan specific BLE device; after it receives a correct and connectable adv packet, it will send connection request and enter ConnMasterRole. If specific BLE device is not scanned in Initiating state, and fails to initiate connection until “create connection timeout” is triggered, it will return to Idle state or Scanning state.

Note that Link Layer can enter Initiating state from Idle state or Scanning state (for example, in the “8258 master kma dongle”, LL directly enters Initiating state from Scanning state). After create connection timeout, it will return to previous Idle state or Scanning state.

- 3) When Link Layer is in ConnMasterRole, it will return to Idle State in any of the following cases:
  - a) Slave sends “terminate” command to Master and requests disconnection. Master will exit ConnMasterRole after it receives this command.
  - b) By sending “terminate” command to Slave, Master actively terminates the connection and exits ConnMasterRole.
  - c) If Master fails to receive any packet due to Master RF Rx abnormality or Slave Tx abnormality until BLE connection supervision timeout is triggered, Master will exit ConnMasterRole.

When Link Layer exits ConnMasterRole state, it will switch to Idle state. If it's needed to continue scanning, the API “blc\_ll\_setScanEnable” should be used to set Link Layer to re-enter Scanning state.

### 3.2.4 Link Layer timing sequence

In this section, Link Layer timing sequence in various states will be illustrated combining with irq\_handler and main\_loop of this BLE SDK.

```
_attribute_ram_code_ void irq_handler(void)
{

 irq_blt_sdk_handler ();

}

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

}
```

The “blt\_sdk\_main\_loop” function at BLE entry serves to process data and events related to BLE protocol stack. UI entry is for user application code.

#### 3.2.4.1 Timing sequence in Idle state

When Link Layer is in Idle state, no task is processed in Link Layer and Physical Layer; the “blt\_sdk\_main\_loop” function doesn’t act and won’t generate any interrupt, i.e. the whole timing sequence of main\_loop is occupied by UI entry.

#### 3.2.4.2 Timing sequence in Advertising state

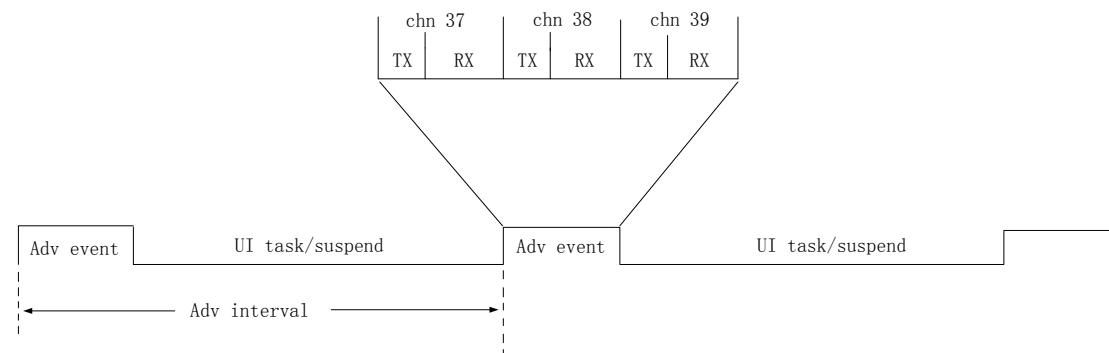


Figure 3-12 Timing sequence chart in Advertising State

As shown in Figure 3-12, an Adv event is triggered by Link Layer during each adv interval. A typical Adv event with three active adv channels will send an advertising packet in channel 37, 38 and 39, respectively. After an adv packet is sent, Slave enters Rx state, and waits for response from Master:

- ✧ If Slave receives a scan request from Master, it will send a scan response to Master.
- ✧ If Slave receives a connect request from Master, it will establish BLE connection with Master and enter Connection state Slave Role.

Code of UI entry in main\_loop is executed during UI task/suspend part in Figure 3-12. This duration can be used for UI task only, or MCU can enter sleep (suspend or deepsleep retention) for the redundant time to reduce power consumption.

In Advertising state, the “blt\_sdk\_main\_loop” function does not need to process many tasks, and only some callback events related to Adv will be triggered, including BLT\_EV\_FLAG\_ADV\_DURATION\_TIMEOUT, BLT\_EV\_FLAG\_SCAN\_RSP, BLT\_EV\_FLAG\_CONNECT, etc.

### 3.2.4.3 Timing sequence in Scanning state

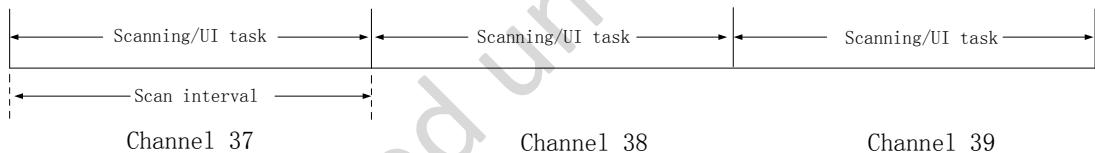


Figure3-13 Timing sequence chart in Scanning state

Scan interval is configured by the API “blc\_ll\_setScanParameter”. During a whole Scan interval, packet reception is implemented in one channel, and Scan window is not designed in the SDK. Therefore, the SDK won’t process the setting of Scan window in the “blc\_ll\_setScanParameter”.

After the end of each Scan interval, it will switch to the next listening channel, and enters next Scan interval. Channel switch action is triggered by interrupt, and it’s executed in irq which takes very short time.

In Scanning interval, PHY Layer of Scan state is always in RX state, and it depends on MCU hardware to implement packet reception. Therefore, all timing in software are for UI task.

After correct BLE packet is received in Scan interval, the data are first buffered in software RX fifo (corresponding to “my\_fifo\_t blt\_rxfifo” in code), and the “blt\_sdk\_main\_loop” function will check whether there are data in software RX fifo. If correct adv data are discovered, the data will be reported to BLE Host via the event “HCI\_SUB\_EVT\_LE\_ADVERTISING\_REPORT”.

### 3.2.4.4 Timing sequence in Initiating state

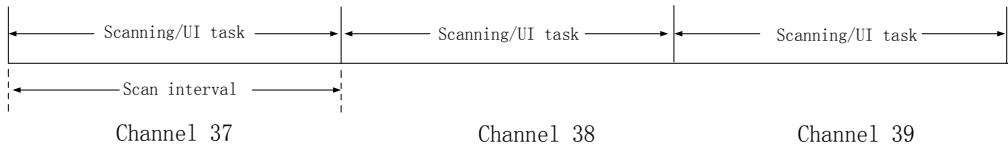


Figure3-14 Timing sequence chart in Initiating state

Timing sequence of Initiating state is similar to that of Scanning state, except that Scan interval is configured by the API “blc\_ll\_createConnection”. During a whole Scan interval, packet reception is implemented in one channel, and Scan window is not designed in the SDK. Therefore, the SDK won’t process the setting of Scan window in the “blc\_ll\_createConnection”.

After the end of each Scan interval, it will switch to the next listening channel, and start a new Scan interval. Channel switch action is triggered by interrupt, and it’s executed in irq which takes very short time.

In Scanning state, BLE Controller will report the received adv packet to BLE Host; however, in Initiating state, adv won’t be reported to BLE Host, and it only scans for the device specified by the “blc\_ll\_createConnection”. If the specific device is scanned, it will send connection\_request and establish connection, then Link Layer enters ConnMasterRole.

### 3.2.4.5 Timing sequence chart in Conn state Slave role

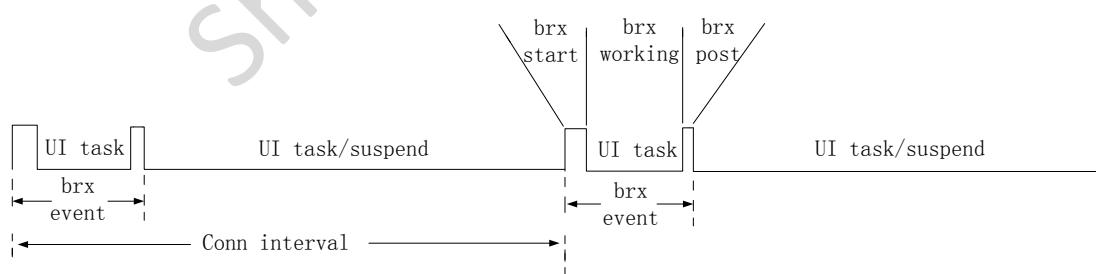


Figure3-15 Timing sequence chart in Conn state Slave role

As shown in Figure3-15, each conn interval starts with a brx event, i.e. transfer process of BLE RF packets by Link Layer: PHY enters Rx state, and an ack packet will be sent to respond to each received data packet from Master.

In this BLE SDK, each brx process consists of three phases.

1) brx start phase

When Master is about to send packet, an interrupt is triggered by system tick irq to enter brx start phase. During this interrupt, MCU sets BLE state machine of PHY to enter brx state, hardware in bottom layer prepares for packet transfer, and then MCU exits from the interrupt irq.

2) brx working phase

After brx start phase ends and MCU exits from irq, hardware in bottom layer enters Rx state first and waits for packet from Master. During the brx working phase, all packet reception and transmission are implemented automatically without involvement of software.

3) brx post phase

After packet transfer is finished, the brx working phase is completed. System tick irq triggers an interrupt to switch to the brx post phase. During this phase, protocol stack will process BLE data and timing sequence according to packet transfer in the brx working phase.

During the three phases, brx start and brx post are implemented in interrupt, while brx working phase does not need the involvement of software, and UI task can be executed normally (Note that during brx working phase, UI task can be executed in the time slots except RX, TX, and System Timer interrupt handler). During the brx working phase, MCU can't enter sleep (suspend or deepsleep retention) since hardware needs to transfer packets.

Within each conn interval, the duration except for brx event can be used for UI task only, or MCU can enter sleep (suspend or deepsleep retention) for the redundant time to reduce power consumption.

In the ConnSlaveRole, the “blt\_sdk\_main\_loop” needs to process the data received during the brx process. During the brx working phase, the data packet received from Master will be copied out during RX interrupt irq handler; these data won't be processed immediately, but buffered in software RX fifo (corresponding to my\_fifo\_t blt\_rx fifo in code). The “blt\_sdk\_main\_loop” function will check whether there are data in software RX fifo, and process the detected data packet correspondingly:

- 1) Decrypt data packet
- 2) Analyze data packet

If the analyzed data belongs to the control command sent by Master to Link Layer, this command will be executed immediately; if it's the data sent by Master to Host layer, the data will be transferred to L2CAP layer via HCI interface.

### 3.2.4.6 Timing sequence in Conn state Master role

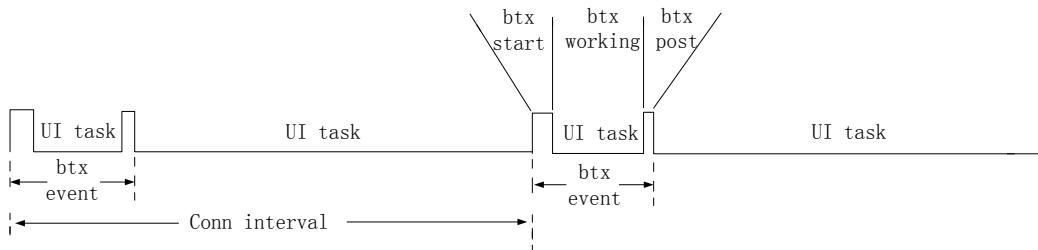


Figure3-16 Timing sequence chart in ConnMasterRole

As shown in Figure3-16, each conn interval starts with a btx event, i.e. transfer process of BLE RF packets by Link Layer: PHY enters Tx state, and waits for an ack packet from Slave for each transmitted data packet. Data transmission to slave will continue if there are more data.

In this BLE SDK, each btx process consists of three phases.

#### 1) btx start phase

When Master is about to send packet, an interrupt is triggered by system tick irq to enter btx start phase. During this interrupt, MCU sets BLE state machine of PHY to enter btx state, hardware in bottom layer prepares for packet transfer, and then MCU exits from the interrupt irq.

#### 2) btx working phase

After btx start phase ends and MCU exits from irq, hardware in bottom layer enters Tx state first. During the btx working phase, all packet reception and transmission are implemented automatically without involvement of software.

#### 3) btx post phase

After packet transfer is finished, the btx working phase is completed. System tick irq triggeres an interrupt to switch to the btx post phase. During this phase, protocol stack will process BLE data and timing sequence according to packet transfer in the btx working phase.

During the three phases, btx start and btx post are implemented in interrupt, while btx working phase does not need the involvement of software, and UI task can be executed normally.

In the ConnMasterRole, the “blt\_sdk\_main\_loop” needs to process the data received during the btx process. During the btx working phase, the data packet received from Master will be copied out during RX interrupt irq handler; these data won’t be processed immediately, but buffered in software RX fifo.

The “blt\_sdk\_main\_loop” function will check whether there are data in software RX fifo, and process the detected data packet correspondingly:

- 1) Decrypt data packet
- 2) Analyze data packet

If the analyzed data belongs to the control command sent by Slave to Link Layer, this command will be executed immediately; if it's the data sent by Master to Host layer, the data will be transferred to L2CAP layer via HCI interface.

### 3.2.4.7 Conn state Slave role timing protection

In ConnSlaveRole state, each interval contains a Brx Event to transfer BLE RF packets. In this SDK, since Brx Event is triggered by interrupt, it's needed to enable MCU system interrupt all the time. If user needs to process some time-consuming tasks and must disable system interrupt in Conn state (e.g. erase flash), Brx Event will be stopped, BLE timing sequence will be disturbed, thus connection is terminated.

A timing sequence protection mechanism is supplied in this SDK. User should strictly follow this mechanism, so that BLE timing sequence won't be disturbed when Brx Event is stopped. Corresponding APIs are shown as below:

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

The API “bls\_ll\_requestConnBrxEventDisable” serves to send a request to disable Brx Event.

- 1) If the return value is 0, it indicates the request to disable Brx Event is rejected. During Brx working phase in Conn state, the return value must be 0; this request won't be accepted until a whole Brx Event is finished, i.e. it can be accepted only during the remaining UI task/suspend duration.
- 2) If the return value is not zero, it indicates this request can be accepted, and the returned non-zero value indicates the time (unit: ms) allowed to stop Brx Event.
  - A. If Link Layer is in Advertising state or Idle state without Brx Event, the return value is “0xffff”. In this case, user can disable system interrupt as long as needed.
  - B. If Link Layer is in Conn state, and Slave receives “update map” or “update connection parameter” request from Master but does not start updating yet, the return value should be the time to start updating minus current time, i.e. it's only allowed to stop Brx Event before the time to start updating, otherwise all following packets won't be received and it will result

in disconnection.

- C. If Link Layer is in Conn state, and no update request is received from Master, the return value should be half of the current connection supervision timeout value. For example, suppose current timeout is 1s, the return value should be 500ms.

After the API “bls\_ll\_requestConnBtxEventDisable” is invoked and the request is accepted, if the time (ms) corresponding to the return value is enough to process user task, the task will be executed. Before the task starts, the API “bls\_ll\_disableConnBtxEvent” should be invoked to disable Btx Event. After the task is completed, the API “bls\_ll\_restoreConnBtxEvent” should be invoked to enable Btx Event and restore BLE timing sequence.

The reference code is shown as below. Timing parameter values in the code depend on actual task.

```
7 if(bls_ll_requestConnBtxEventDisable() > 300)
8 {
9
10 bls_ll_disableConnBtxEvent();
11
12 #if 0 //test 1
13 irq_disable();
14 DBG_CHN3_HIGH;
15 sleep_us(287*1000);
16 DBG_CHN3_LOW;
17 irq_enable();
18 #else //test 2
19 DBG_CHN3_HIGH;
20 flash_erase_sector(0x40000);
21 DBG_CHN3_LOW;
22 #endif
23
24 bls_ll_restoreConnBtxEvent();
25
26 }
```

### 3.2.5 Link Layer state machine extension

The sections about BLE Link Layer state machine and timing sequence introduced some basic states, which can meet requirements of basic BLE Slave/Master applications.

However, considering the requirement of some special applications (e.g. advertising is needed in Conn state Slave role), some special extended functions are added to Link Layer state machine in Telink BLE SDK.

### 3.2.5.1 Scanning in Advertising state

When Link Layer is in Advertising state, Scanning feature can be added.

The API below serves to add Scanning feature:

```
ble_sts_t b1c_ll_addScanningInAdvState(void);
```

The API below serves to remove Scanning feature:

```
ble_sts_t b1c_ll_removeScanningFromAdvState(void);
```

For the two APIs above, the return value of ble\_sts\_t type should be BLE\_SUCCESS.

By combining timing sequence chart of Advertising state and Scanning state, when Scanning feature is added to Advertising state, the extended timing sequence is shown as below.

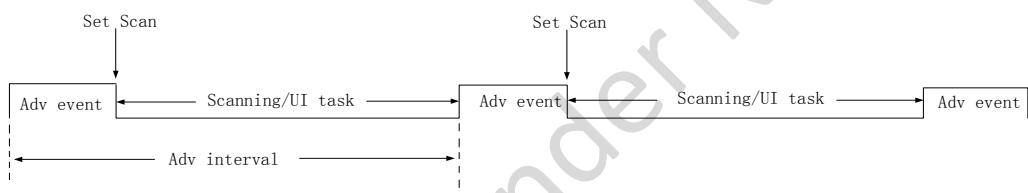


Figure3-17 Timing sequence chart with Scanning in Advertising state

Currently Link Layer is still in Advertising state (BLS\_LINK\_STATE\_ADV). During each Adv interval, the remaining time except for Adv event is used for Scanning.

During each “Set Scan”, the difference of current time and previous “Set Scan” will be checked whether it exceeds a Scan interval (setting from the “b1c\_ll\_setScanParameter”). If the difference exceeds a Scan interval, Scan channel (channel 37/38/39) will be switched.

For usage of Scanning in Advertising state, please refer to the “TEST\_SCANNING\_IN\_ADV\_AND\_CONN\_SLAVE\_ROLE” in 8258\_feature\_test.

### 3.2.5.2 Scanning in ConnSlaveRole

When Link Layer is in ConnSlaveRole state, Scanning feature can be added.

The API below serves to add Scanning feature:

```
ble_sts_t b1c_ll_addScanningInConnSlaveRole(void);
```

The API below serves to remove Scanning feature:

```
ble_sts_t b1c_ll_removeScanningFromConnSlaveRole(void);
```

For the two APIs above, the return value of ble\_sts\_t type should be BLE\_SUCCESS.

By combining timing sequence chart of Scanning state and ConnSlaveRole, when Scanning feature is added to ConnSlaveRole, the extended timing sequence is shown as below.

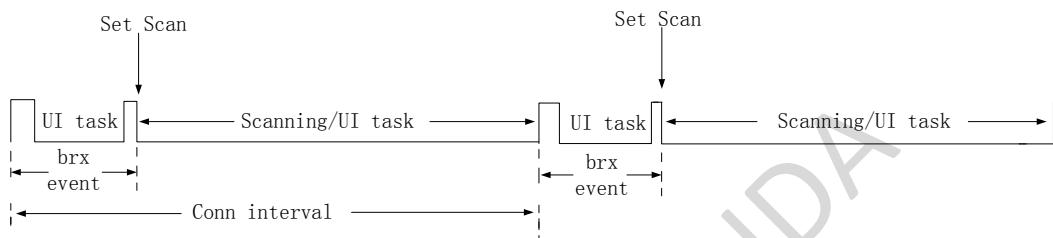


Figure3-18 Timing sequence chart with Scanning in ConnSlaveRole

Currently Link Layer is still in ConnSlaveRole (BLS\_LINK\_STATE\_CONN). During each Conn interval, the remaining time except for brx event is used for Scanning.

During each “Set Scan”, the difference of current time and previous “Set Scan” will be checked whether it exceeds a Scan interval (setting from the “b1c\_ll\_setScanParameter”). If the difference exceeds a Scan interval, Scan channel (channel 37/38/39) will be switched.

For usage of Scanning in ConnSlaveRole, please refer to the “TEST\_SCANNING\_IN\_ADV\_AND\_CONN\_SLAVE\_ROLE” in 8258\_feature\_test.

### 3.2.5.3 Advertising in ConnSlaveRole

When Link Layer is in ConnSlaveRole, Advertising feature can be added.

The API below serves to add Advertising feature:

```
ble_sts_t b1c_ll_addAdvertisingInConnSlaveRole(void);
```

The API below serves to remove Advertising feature:

```
ble_sts_t b1c_ll_removeAdvertisingFromConnSlaveRole(void);
```

For the two APIs above, the return value of ble\_sts\_t type should be BLE\_SUCCESS.

By combining timing sequence chart of Advertising state and ConnSlaveRole, when Advertising feature is added to ConnSlaveRole, the extended timing sequence is shown as below.

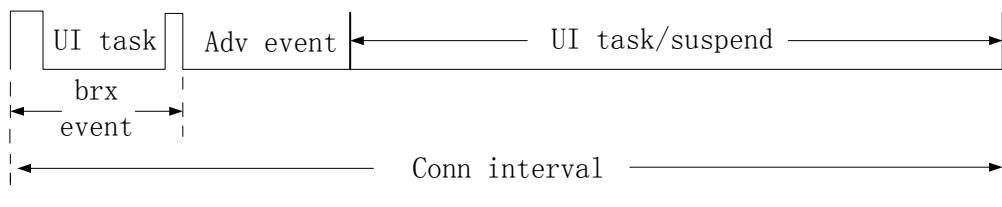


Figure3-19 Timing sequence chart with Advertising in ConnSlaveRole

Currently Link Layer is still in ConnSlaveRole (BLS\_LINK\_STATE\_CONN). During each Conn interval, after a brx event is finished, an adv event is executed immediately, and the remaining time is used for UI task or sleep (suspend/deepsleep retention) to save power.

For usage of Advertising in ConnSlaveRole, please refer to the "TEST\_ADVERTISING\_IN\_CONN\_SLAVE\_ROLE" in 8258\_feature\_test.

### 3.2.5.4 Advertising and Scanning in ConnSlaveRole

By combining usage of Scanning in ConnSlaveRole and Advertising in ConnSlaveRole, Scanning and Advertising can be added to ConnSlaveRole. Timing sequence is shown as below.

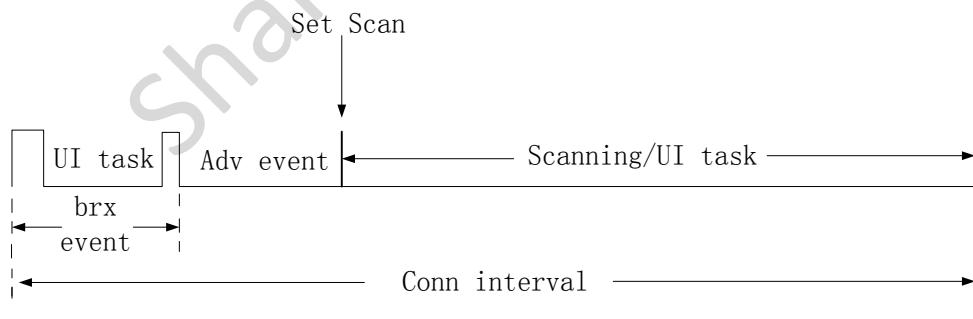


Figure3-20 Timing sequence chart with Advertising and Scanning in ConnSlaveRole

Currently Link Layer is still in ConnSlaveRole (BLS\_LINK\_STATE\_CONN). During each Conn interval, after a brx event is finished, an adv event is executed immediately, and the remaining time is used for Scanning.

During each “Set Scan”, the difference of current time and previous “Set Scan” will be checked whether it exceeds a Scan interval (setting from the “blc\_ll\_setScanParameter”). If the difference exceeds a Scan interval, Scan channel (channel 37/38/39) will be switched.

For usage of Advertising and Scanning in ConnSlaveRole, please refer to the “TEST\_ADVERTISING\_SCANNING\_IN\_CONN\_SLAVE\_ROLE” in 8258\_feature\_test.

### 3.2.6 Link Layer TX fifo & RX fifo

All RF data of the APP layer and BLE Host should be transmitted via Link Layer of Controller. A BLE TX fifo is designed in the Link Layer, which can be used to buffer the received data and send data after brx/btx starts.

All data received from peer device during Link Layer brx/btx will be buffered in a BLE RX fifo, and then transmitted to BLE Host or APP layer for processing.

BLE TX fifo and BLE RX fifo are defined in the APP layer, and they adopt the same processing method for Slave role and Master role.

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

By default, RX fifo size is 64, and TX fifo size is 40. It's not allowed to modify the two size values unless it's needed to use “data length extension”.

Both TX fifo number and RX fifo number must be configured as a power of 2, i.e. 2, 4, 8, 16, and etc. User can modify as needed.

Default RX fifo number is 8, which is a reasonable value to ensure up to 8 data packets can be buffered in Link Layer bottom layer. If set as a large value, it will occupy large SRAM area. If set as a small value, it may bring the risk of data corruption due to wraparound or overflow.

During brx event, Link Layer is likely to be in More Data (MD) mode during an interval and continuously receive multiple packets. If RX fifo number is set as 4, there may be five or six packets in an interval (e.g. OTA, play Master audio data); however, due to long decryption time, response to these data by upper layer cannot be processed in real time, then some data may be overflowed.

Following shows an example of RX overflow with the four assumptions below.

- 1) RX fifo number is 8.
- 2) Before brx\_event(n) starts, read pointer and write pointer of RX fifo are 0 and 2, respectively.
- 3) During brx\_event(n) and brx\_event(n+1), due to task blockade in main\_loop, data of RX fifo are not fetched in time.
- 4) The two brx\_event phases include multiple packets.

As per the description in section **3.2.4.5 Timing sequence in Conn state Slave role**, BLE packets received during brx\_working phase will only be copied into RX fifo (RX fifo write pointer ++), and the operations to fetch and process data from RX fifo are implemented in main\_loop (RX fifo read pointer ++). As shown below, the sixth packet will cover the area of rptr (read pointer) 0. Note that during brx working phase, UI task occupies time slots other than interrupt handler time for RX, TX, System Timer and etc.

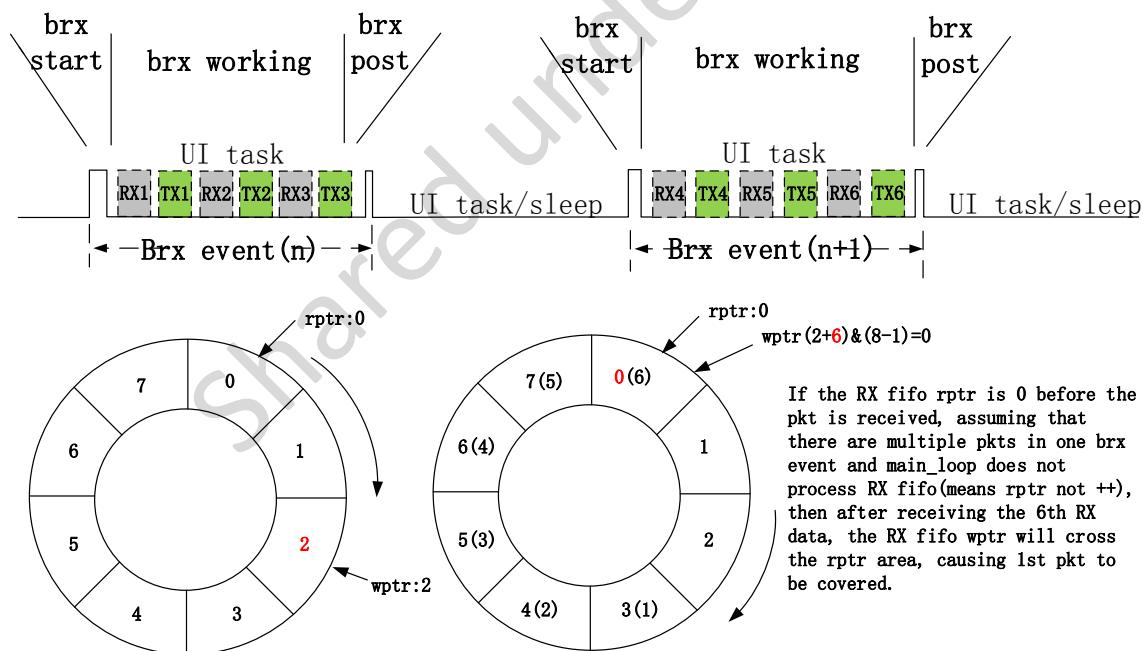


Figure3-21 RX overflow case 1

Relative to the extreme case above with long task blockade duration due to one connection interval, the case below is more likely to occur:

During one brx\_event, since Master writes multiple packets (e.g. 7/8 packets) into Slave, Slave fails to process the received data in time. As shown below, the rptr (read pointer) is increased by two, but the wptr (write pointer) is also increased by eight, which thus causes data overflow.

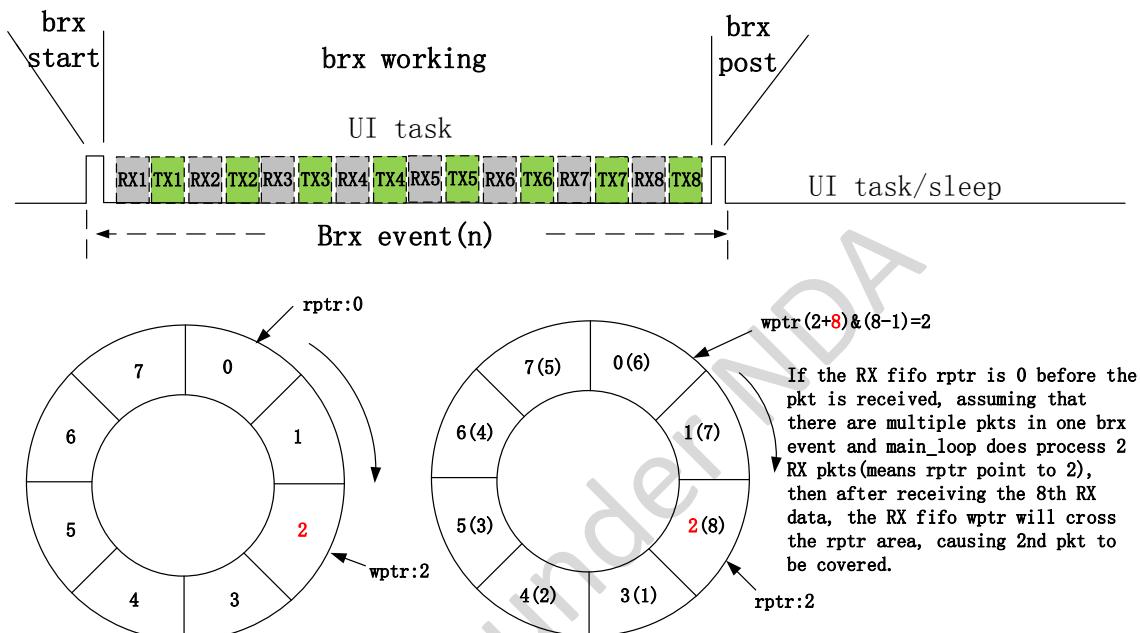


Figure3-22 RX overflow case 2

Data loss due to overflow will bring MIC failure and disconnection for the encryption system.

In the old SDK, during brx event Rx IRQ, since data are filled into Rx fifo without overflow check, if main\_loop fails to process data in time, it will bring the risk of data overflow. Therefore, Master should not send too many data in a connection interval, and the time to process UI tasks should be as short as possible, so as to avoid blockade.

Considering this, Rx overflow check is added in the new SDK: During the brx/btx event Rx IRQ, check whether the difference of current RX fifo wptr and rptr exceeds Rx fifo number. If the check result shows current Rx fifo is fully occupied, the RF won't send ACK to the peer, data re-transmission is ensured by BLE protocol, and the SDK also supplies the callback function of Rx overflow to inform user (see **section 3.2.7.2 Telink defined event**).

Similarly, if there are more than 8 valid packets in an interval, the default number 8 is not enough.

Default TX fifo number is 16, which is enough to process common audio remote control function with large data volume. User can modify this number to 8 to save fifo space.

If set as a large value (e.g. 32), it will occupy large SRAM area.

In TX fifo, stack in SDK bottom layer needs two fifos, while APP layer can use the remaining fifos. If TX fifo number is 16, APP layer can use 14 fifos; if TX fifo number is 8, APP layer can use 6 fifos.

To send data in APP layer (e.g. invoke the “bls\_att\_pushNotifyData”), user should check current number of TX fifos available for Link Layer.

The API below serves to check current number of occupied TX fifos (note that it's not the number of remaining available fifos).

```
u8 blc_ll_getTxFifoNumber (void);
```

For example, suppose TX fifo number is the default value 16, and 14 fifos are available for user. Therefore, as long as the return value is less than 14, there are still fifos available: if the return value is 13, there is 1 fifo available; if the return value is 0, there are 14 fifos available.

When using TX fifo, if user wants to check the number of remaining fifos before determining whether to directly push data, to avoid any border problem, one fifo should be reserved.

During audio processing of 8258 remote, since a sum of audio data is disassembled into five packets, five TX fifos are needed, and occupied fifos should not be more than 9. In case of exception situation, e.g. when BLE stack needs to respond to master command by inserting a data packet, max number of occupied fifos is set to 8. Implementation is shown as below (The number of occupied TX fifos should not exceed 8 to push audio data into TX fifo).

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

}
```

Besides the auto-processing mechanism to avoid data overflow, the bottom layer of the SDK also provides the API below to limit the number of “more data” to be received during an interval. The API applies to the case when user wants to limit data number even if RX fifos are enough for use.

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

The parameter “num” specifies the maximum number of “more data” which should not exceed RX fifo number.

Note: The API (“num” > 0) should be invoked in the APP layer, so that the function to limit More Data during a connection event can be enabled.

### 3.2.7 Controller Event

Considering user may need to record and process some key actions of BLE stack bottom layer in the APP layer, Telink BLE SDK provides three types of event: Standard HCI event defined by BLE Controller; Telink defined event; event-notification type GAP event (Host event) defined by BLE Host for stack flow interaction (see section **3.3.5.2 GAP event**).

As shown in the BLE SDK event architecture below: HCI event and Telink defined event are Controller event, while GAP event is BLE Host event.

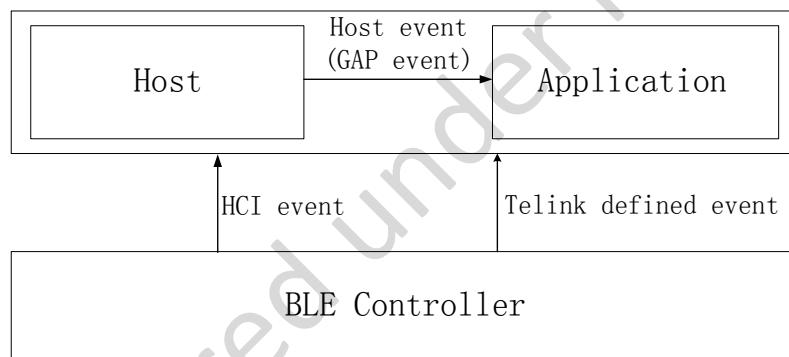


Figure3-23 BLE SDK event architecture

#### 3.2.7.1 Controller HCI Event

HCI event is designed according to BLE Spec; Telink defined event only applies to BLE Slave (8258 remote/8258 module etc).

- ✧ BLE Master only supports HCI event.
- ✧ BLE Slave supports both HCI event and Telink defined event.

For BLE Slave, basically the two sets of event are independent of each other, except for the connect and disconnect event of Link Layer.

User can select one set or use both as needed. In Telink BLE SDK, 8258 remote/8258 module use Telink defined event, while 8258 hci/8258 master kma dongle use Controller HCI event.

As shown in the “Host + Controller” architecture below, Controller HCI event indicates all events of Controller are reported to Host via HCI.

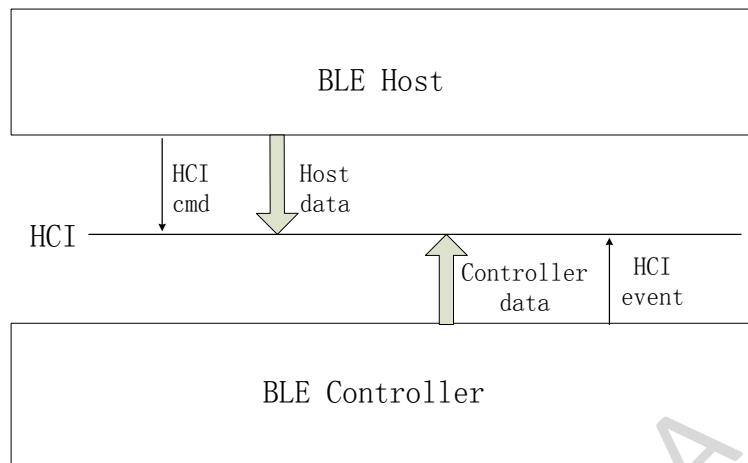


Figure3-24 HCI event

For definition of Controller HCI event, please refer to “Core\_v5.0” (Vol 2/Part E/7.7 “Events”). “LE Meta Event” in 7.7.65 indicates HCI LE (low energy) Event, while the others are common HCI events.

As defined in Spec, Telink BLE SDK also divides Controller HCI event into two types: HCI Event and HCI LE event. Since Telink BLE SDK focuses on BLE, it supports most HCI LE events and only a few basic HCI events.

For the definition of macros and interfaces related to Controller HCI event, please refer to head files under “stack/ble/hci”.

To receive Controller HCI event in Host or APP layer, user should register callback function of Controller HCI event, and then enable mask of corresponding event.

Following are callback function prototype and register interface of Controller HCI event:

```

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

```

In the callback function prototype, “**u32 h**” is a mark which will be frequently used in bottom-layer stack, and user only needs to know the following:

```

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

```

“**HCI\_FLAG\_EVENT\_TLK\_MODULE**” will be introduced in “Telink defined event”, while “**HCI\_FLAG\_EVENT\_BT\_STD**” indicates current event is Controller HCI event.

In the callback function prototype, “para” and “n” indicate data and data length of event. The data is consistent with the definition in BLE spec.

User can refer to usage in 8258 master kma dongle as well as implementation of the “controller\_event\_callback” function.

```
blc_hci_registerControllerEventHandler(controller_event_callback);
```

## 1. HCI event

Telink BLE SDK supports a few HCI events. Following lists some events for user.

|                                               |      |
|-----------------------------------------------|------|
| #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

Please refer to “Core\_v5.0” (Vol 2/Part E/7.7.5 “Disconnection Complete Event”).

Total data length of this event is 7, and 1-byte “param len” is 4, as shown below.

Please refer to BLE spec for data definition.

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

Figure3-25 Disconnection Complete Event

### 2) HCI\_EVT\_ENCRYPTION\_CHANGE & HCI\_EVT\_ENCRYPTION\_KEY\_REFRESH

Please refer to “Core\_v5.0” (Vol 2/Part E/7.7.8 & 7.7.39).

The two events (available in 8258 master kma dongle) are related to Controller encryption, and the processing is assembled in library.

### 3) HCI\_EVT\_READ\_REMOTE\_VER\_INFO\_COMPLETE

Please refer to “Core\_v5.0” (Vol 2/Part E/7.7.12).

When Host uses the “HCI\_CMD\_READ\_REMOTE\_VER\_INFO” command to exchange version information between Controller and BLE peer device, and version of peer device is received, this event will be reported to Host.

Total data length of this event is 11, and 1-byte “param len” is 8, as shown below. Please refer to BLE spec for data definition.

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

Figure3-26 Read Remote Version Information Complete Event

### 4) HCI\_EVT\_LE\_META

It indicates current event is HCI LE event, and event type can be judged according to sub event code.

Except for HCI\_EVT\_LE\_META, other HCI events should use the API below to enable corresponding event mask.

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

Definition of event mask:

|                                                               |               |
|---------------------------------------------------------------|---------------|
| #define HCI_EVT_MASK_DISCONNECTION_COMPLETE                   | 0x0000000010  |
| #define HCI_EVT_MASK_ENCRYPTION_CHANGE                        | 0x0000000080  |
| #define HCI_EVT_MASK_READ_REMOTE_VERSION_INFORMATION_COMPLETE | 0x00000000800 |

If HCI event mask is set via this API, by default only the mask corresponding to the “HCI\_CMD\_DISCONNECTION\_COMPLETE” is enabled in the SDK, i.e. the SDK only ensures report of “Controller disconnect event” by default.

## 2. HCI LE event

When event code in HCI event is “HCI\_EVT\_LE\_META” to indicate HCI LE event, common subevent code are shown as below:

|                                                   |      |
|---------------------------------------------------|------|
| #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

Please refer to “Core\_v5.0” (Vol 2/Part E/7.7.65.1 “LE Connection Complete Event”).

When connection is established between Controller Link Layer and peer device, this event will be reported.

Total data length of this event is 22, and 1-byte “param len” is 19, as shown below. Please refer to BLE spec for data definition.

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

Figure3-27 LE Connection Complete Event

### 2) HCI\_SUB\_EVT\_LE\_ADVERTISING\_REPORT

Please refer to “Core\_v5.0” (Vol 2/Part E/7.7.65.2 “LE Advertising Report Event”).

When Link Layer of Controller scans right adv packet, it will be reported to Host via the “HCI\_SUB\_EVT\_LE\_ADVERTISING\_REPORT”.

Data length of this event is not fixed and it depends on payload of adv packet, as shown below. Please refer to BLE spec for data definition.

|               |            |           |               |            |            |                    |
|---------------|------------|-----------|---------------|------------|------------|--------------------|
| 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]         |

Figure3-28 LE Advertising Report Event

Note: In Telink BLE SDK, each “LE Advertising Report Event” only reports an adv packet, i.e. “i” in Figure3-28 is 1.

### 3) HCI\_SUB\_EVT\_LE\_CONNECTION\_UPDATE\_COMPLETE

Please refer to “Core\_v5.0” (Vol 2/Part E/7.7.65.3 “LE Connection Update Complete Event”).

When “connection update” in Controller takes effect, the “HCI\_SUB\_EVT\_LE\_CONNECTION\_UPDATE\_COMPLETE” will be reported to Host.

Total data length of this event is 13, and 1-byte “param len” is 10, as shown below. Please refer to BLE spec for data definition.

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

Figure3-29 LE Connection Update Complete Event

### 4) HCI\_SUB\_EVT\_LE\_CONNECTION\_ESTABLISH

The “HCI\_SUB\_EVT\_LE\_CONNECTION\_ESTABLISH” is a supplement to the “HCI\_SUB\_EVT\_LE\_CONNECTION\_COMPLETE”, so all the parameters except for subevent is the same. In the SDK, 8258 master kma dongle uses this event.

This Telink private defined event is the sole event which is not standard in BLE spec. This event is only used in 8258 master kma dongle.

Following illustrates the reason for Telink to define this event.

When BLE Controller in Initiating state scans adv packet from specific device to be connected, it will send connection request packet to peer device; no matter whether this connection request is received, it will be considered as “Connection complete”, “LE Connection Complete Event” will be reported to Host, and Link Layer immediately enters Master role.

Since this packet does not support ack/retry mechanism, Slave may miss the connection request, thus it cannot enter Slave role, and won’t enter brx mode to transfer packets.

In this case, Master Controller will process according to the mechanism below: After it enters Master role, it will check whether there’s any packet received from Slave during the beginning 6~10 conn intervals (CRC check is negligible).

- ❖ If no packet is received, it’s considered that Slave does not receive connection request; suppose “LE Connection Complete Event” has already been reported, it must report a “Disconnection Complete Event” quickly, and indicate disconnect reason is “0x3E (HCI\_ERR\_CONN\_FAILED\_TO\_ESTABLISH)”.

- ✧ If there's packet received from Slave, it can confirm that Connection is established, thus Master can continue rest of the flow.

According to the description above, the processing method of BLE Host should be:

After it receives “LE Connection Complete Event” of Controller, it cannot confirm that connection has already been established, but instead, starts a timer based on conn interval (timing value should be configured as 10 intervals or above to cover the longest time).

After the timer is started, it will check whether there is “Disconnection Complete Event” with disconnect reason of 0x3E; if there is no such event, it will be considered as “Connection Established”.

Considering this processing of BLE Host is very complex and error prone, the SDK defines the “HCI\_SUB\_EVT\_LE\_CONNECTION\_ESTABLISH” in the bottom layer. When Host receives this event, it indicates that Controller has confirmed connection is OK on Slave side and can continue rest of the flow.

“HCI LE event” needs the API below to enable mask.

```
ble_sts_t b1c_hci_le_setEventMask_cmd(u32 evtMask);
 //eventMask: LE
```

Following lists some evtMask definitions. User can view the other events in the “hci\_const.h”.

|                                                    |            |
|----------------------------------------------------|------------|
| #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

If HCI LE event mask is not set via this API, mask of all HCI LE events in the SDK are disabled by default.

### 3.2.7.2 Telink defined event

Besides standard Controller HCI event, the SDK also provides Telink defined event.

Up to 20 Telink defined events are supported, which are defined by using macros in the “stack/ble/ll/ll.h”.

Current SDK version supports the following callback events. The “BLT\_EV\_FLAG\_CONNECT/BLT\_EV\_FLAG\_TERMINATE” has the same function as the “HCI\_SUB\_EVT\_LE\_CONNECTION\_COMPLETE” /“HCI\_EVT\_DISCONNECTION\_COMPLETE” in HCI event, but data definition of these events are different.

|         |                                  |    |
|---------|----------------------------------|----|
| #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_ABANDOM      | 6  |
| #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 |

Telink defined event is only triggered in BLE slave applications. There are two ways to implement callback of Telink defined event in BLE slave application.

- 1) The first method, which is called “independent registration”, is to independently register callback function for each event.

Prototype of callback function is shown as below:

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

“e”: event number.

“p”: It’s the pointer to the data transmitted from the bottom layer when callback function is executed, and it varies with the callback function.

“n”: length of valid data pointed by pointer.

API to register callback function:

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

Whether each event will respond depends on whether corresponding callback function is registered in APP layer.

- 2) The second method, which is called “shared event entry”, is that all event callback functions share the same entry.

Whether each event will respond depends on whether its event mask is enabled.

This method uses the same API as HCI event to register event callback:

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

void blc_hci_registerControllerEventHandler(
 hci_event_handler_t handler);
```

Although registered callback function of HCI event is shared, they are different in implementation. In HCI event callback function:

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

While in Telink defined event “shared event entry”:

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

Where “e” is event number of Telink defined event.

Telink defined event “shared event entry” is similar to mask of HCI event; the API below serves to set the mask to determine whether each event will be responded.

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

Relationship between evtMask and event number is:

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

The two methods for Telink defined event are exclusive to each other. The first method is recommended and is adopted by most demo code of the SDK (e.g. “8258\_remote”); only “8258\_module” uses the “shared event entry” method.

Coding for these two methods are explained using connect and terminate event callback.

1) The first method: “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) The second method: “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));
```

Following will introduce details about all events, event trigger condition and parameters of corresponding callback function for Controller.

### 1) **BLT\_EV\_FLAG\_ADV**

This event is not used in current SDK.

### 2) **BLT\_EV\_FLAG\_ADV\_DURATION\_TIMEOUT**

- 1) Event trigger condition: If the API “bls\_ll\_setAdvDuration” is invoked to set advertising duration, a timer will be started in BLE stack bottom layer. When the timer reaches the specified duration, advertising is stopped, and this event is triggered.

In the callback function of this event, user can modify adv event type, re-enable advertising, re-configure advertising duration, and etc.

- 2) Pointer “p”: null pointer.
- 3) Data length “n”: 0.

Note: This event won't be triggered in “advertising in ConnSlaveRole” which is an extended state of Link Layer.

### 3) **BLT\_EV\_FLAG\_SCAN\_RSP**

- 1) Event trigger condition: When Slave is in advertising state, this event will be triggered if Slave responds with scan response to the scan request from Master.
- 2) Pointer “p”: null pointer.
- 3) Data length “n”: 0.

#### 4) BLT\_EV\_FLAG\_CONNECT

- 1) Event trigger condition: When Link Layer is in advertising state, this event will be triggered if it responds to connect request from Master and enters Conn state Slave role.
- 2) Data length "n": 34.
- 3) Pointer "p": p points to one 34-byte RAM area, corresponding to the "connect request PDU" below.

| Payload             |                    |                       |
|---------------------|--------------------|-----------------------|
| InitA<br>(6 octets) | AdvA<br>(6 octets) | LLData<br>(22 octets) |

Figure 2.10: CONNECT\_REQ PDU payload

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

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

Figure 2.11: LLData field structure in CONNECT\_REQ PDU's payload

Figure3-30 Connect request PDU

Please refer to the "rf\_packet\_connect\_t" defined in the "ble\_common.h". In the structure below, the connect request PDU is from scanA[6] (corresponding to InitA in Figure3-30) to hop.

```

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

- 1) Event trigger condition: This event will be triggered when Link Layer state machine exits from Conn state Slave role in any of the three specific cases.
- 2) Pointer “p”: p points to an u8-type variable “terminate\_reason”. This variable indicates the reason for disconnection of Link Layer.
- 3) Data length “n”: 1.

Three cases to exit Conn state Slave role and corresponding reasons are listed as below:

- A. If Slave fails to receive packet from Master for a duration due to RF communication problem (e.g. bad RF or Master is powered off), and “connection supervision timeout” expires, this event will be triggered to terminate connection and return to None Conn state. The terminate reason is HCI\_ERR\_CONN\_TIMEOUT (0x08).
- B. If Master sends “terminate” command to actively terminate connection, after Slave responds to the command with an ack, this event will be triggered to terminate connection and return to None Conn state.

The terminate reason is the Error Code in the “LL\_TERMINATE\_IND” control packet received in Slave Link Layer. The Error Code is determined by Master. Common Error Codes include HCI\_ERR\_REMOTE\_USER\_TERM\_CONN (0x13), HCI\_ERR\_CONN\_TERM\_MIC\_FAILURE (0x3D), and etc.

- C. If Slave invokes the API “bls\_ll\_terminateConnection(u8 reason)” to actively terminate connection, this event will be triggered. The terminate reason is the actual parameter “reason” of this API.

## 6) BLT\_EV\_FLAG\_LL\_REJECT\_IND

- 1) Event trigger condition: When Master sends a “LL\_ENC\_REQ” (encryption request) in the Link Layer and it’s declared to use the pre-allocated LTK, if Slave fails to find corresponding LTK and responds with a “LL\_REJECT\_IND” (or “LL\_REJECT\_EXT\_IND”), this event will be triggered.
- 2) Pointer “p”: p points to the response command (LL\_REJECT\_IND or LL\_REJECT\_EXT\_IND).
- 3) Data length “n”: 1.

For more information, please refer to “Core\_v5.0” Vol 6/Part B/2.4.2.

## 7) BLT\_EV\_FLAG\_RX\_DATA\_ABANDOM

- 1) Event trigger condition: This event will be triggered when BLE RX fifo overflows (see section **3.2.6 Link Layer TX fifo & RX fifo**), or the number of More Data received in an interval exceeds the preset threshold (Note: User needs to invoke the API “blc\_ll\_init\_max\_md\_nums” with non-zero parameter, so that SDK bottom layer will check the number of More Data.)
- 2) Pointer “p”: null pointer.
- 3) Data length “n”: 0.

## 8) BLT\_EV\_FLAG\_PHY\_UPDATE

- 1) Event trigger condition: This event will be triggered after the update succeeds or fails when the slave or master proactively initiates LL\_PHY\_REQ; Or when the slave or master passively receives LL\_PHY\_REQ and meanwhile PHY is updated successfully, this event will be triggered.
- 2) Data length “n”: 1.
- 3) Pointer “p”: p points to an u8-type variable indicating the current connection of 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

- 1) Event trigger condition: This event will be triggered when Slave and Master exchange max data length of Link Layer, i.e. one side sends “ll\_length\_req”, while the peer responds with “ll\_length\_rsp”.  
If Slave actively sends “ll\_length\_req”, this event won’t be triggered until “ll\_length\_rsp” is received.  
If Master initiates “ll\_length\_req”, this event will be triggered immediately after Slave responds with “ll\_length\_rsp”.
- 2) Data length “n”: 12.
- 3) Pointer “p”: p points to data of a memory area, corresponding to the beginning six u16-type variables in the structure below.

```
typedef struct {
```

```
u16 connEffectiveMaxRxOctets;
u16 connEffectiveMaxTxOctets;
u16 connMaxRxOctets;
u16 connMaxTxOctets;
u16 connRemoteMaxRxOctets;
u16 connRemoteMaxTxOctets;

.....
}ll_data_extension_t;
```

“connEffectiveMaxRxOctets” and “connEffectiveMaxTxOctets” are max RX and TX data length finally allowed in current connection;

“connMaxRxOctets” and “connMaxTxOctets” are max RX and TX data length of the device;

“connRemoteMaxRxOctets” and “connRemoteMaxTxOctets” are max RX and TX data length of peer device.

```
connEffectiveMaxRxOctets = min(supportedMaxRxOctets,connRemoteMaxTxOctets);
connEffectiveMaxTxOctets = min(supportedMaxTxOctets, connRemoteMaxRxOctets);
```

## 10) BLT\_EV\_FLAG\_GPIO\_EARLY\_WAKEUP

### 1) Event trigger condition:

Slave will calculate wakeup time before it enters sleep (suspend or deepsleep retention), so that it can wake up when the wakeup time is due (It's realized via timer in sleep).

Since user tasks won't be processed until wakeup from sleep, long sleep time may bring problem for real-time demanding applications.

Take keyboard scanning as an example: If user presses keys fast, to avoid key press loss and process debouncing, it's recommended to set the scan interval as 10~20ms; longer sleep time (e.g. 400ms or 1s, which may be reached when latency is enabled) will lead to key press loss. So it's needed to judge current sleep time before MCU enters sleep; if it's too long, the wakeup method of user key press should be enabled, so that MCU can wake up from sleep (suspend or deepsleep retention) in advance (i.e. before timer timeout) if any key press is detected. This will be introduced in details in following PM module section.

The event “BLT\_EV\_FLAG\_GPIO\_EARLY\_WAKEUP” will be triggered if MCU is woke up from sleep (suspend or deepsleep) by GPIO in advance before wakeup timer expires.

### 2) Data length “n”: 1.

- 3) Pointer “p”: p points to an u8-type variable “wakeup\_status”. This variable indicates valid wakeup source status for current sleep.

Following types of wakeup status are defined in the “drivers/8258/pm.h”:

```
enum {
 WAKEUP_STATUS_TIMER = BIT(1),
 WAKEUP_STATUS_PAD = BIT(3),

 STATUS_GPIO_ERR_NO_ENTER_PM = BIT(7),
 STATUS_ENTER_SUSPEND = BIT(30),
};
```

For parameter definition above, please refer to “Power Management” section.

### 11) BLT\_EV\_FLAG\_CHN\_MAP\_REQ

- 1) Event trigger condition: When Slave is in Conn state, if Master needs to update current connection channel list, it will send a “LL\_CHANNEL\_MAP\_REQ” command to Slave; this event will be triggered after Slave receives this request from Master and has not processed the request yet.
- 2) Data length “n”: 5.
- 3) Pointer “p”: p points to the starting address of the following channel list array.

unsigned char type bltc.conn\_chn\_map[5]

Note: When the callback function is executed, p points to the old channel map before update.

Five bytes are used in the “conn\_chn\_map” to indicate current channel list by mapping. Each bit indicates a channel:

conn\_chn\_map[0] bit0-bit7 indicate channel0~channel7, respectively.

conn\_chn\_map[1] bit0-bit7 indicate channel8~channel15, respectively.

conn\_chn\_map[2] bit0-bit7 indicate channel16~channel23, respectively.

conn\_chn\_map[3] bit0-bit7 indicate channel24~channel31, respectively.

conn\_chn\_map[4] bit0-bit4 indicate channel32~channel36, respectively.

### 12) BLT\_EV\_FLAG\_CHN\_MAP\_UPDATE

- 1) Event trigger condition: When Slave is in connection state, this event will be triggered if Slave has updated channel map after it receives the “LL\_CHANNEL\_MAP\_REQ” command from Master.
- 2) Pointer “p”: p points to the starting address of the new channel map conn\_chn\_map[5] after update.
- 3) Data length “n”: 5.

### 13) BLT\_EV\_FLAG\_CONN\_PARA\_REQ

- 1) Event trigger condition: When Slave is in connection state (Conn state Slave role), if Master needs to update current connection parameters, it will send a “LL\_CONNECTION\_UPDATE\_REQ” command to Slave; this event will be triggered after Slave receives this request from Master and has not processed the request yet.
- 2) Data length “n”: 11.
- 3) Pointer “p”: p points to the 11-byte PDU of the LL\_CONNECTION\_UPDATE\_REQ.

| CtrData              |                         |                        |                       |                       |                       |
|----------------------|-------------------------|------------------------|-----------------------|-----------------------|-----------------------|
| WinSize<br>(1 octet) | WinOffset<br>(2 octets) | Interval<br>(2 octets) | Latency<br>(2 octets) | Timeout<br>(2 octets) | Instant<br>(2 octets) |
|                      |                         |                        |                       |                       |                       |

Figure 2.15: CtrData field of the LL\_CONNECTION\_UPDATE\_REQ PDU

Figure3-31 LL\_CONNECTION\_UPDATE\_REQ format in BLE stack

#### 14) BLT\_EV\_FLAG\_CONN\_PARA\_UPDATE

- 1) Event trigger condition: When Slave is in connection state, this event will be triggered if Slave has updated connection parameters after it receives the “LL\_CONNECTION\_UPDATE\_REQ” from Master.
- 2) Data length “n”: 6.
- 3) Pointer “p”: p points to the new connection parameters after update, as shown below.
  - p[0] | p[1]<<8: new connection interval in unit of 1.25ms.
  - p[2] | p[3]<<8: new connection latency.
  - p[4] | p[5]<<8: new connection timeout in unit of 10ms.

#### 15) BLT\_EV\_FLAG\_SUSPEND\_ENETR

- 1) Event trigger condition: When Slave executes the function “blt\_sdk\_main\_loop”, this event will be triggered before Slave enters suspend.
- 2) Pointer “p”: Null pointer.
- 3) Data length “n”: 0.

#### 16) BLT\_EV\_FLAG\_SUSPEND\_EXIT

- 1) Event trigger condition: When Slave executes the function “blt\_sdk\_main\_loop”, this event will be triggered after Slave is woke up from suspend.
- 2) Pointer “p”: Null pointer.
- 3) Data length “n”: 0.

Note: This callback is executed after SDK bottom layer executes “cpu\_sleep\_wakeup” and Slave is woke up, and this event will be triggered no matter whether the actual wakeup source is gpio or timer. If the event “BLT\_EV\_FLAG\_GPIO\_EARLY\_WAKEUP” occurs at the same time, for the sequence to execute the two events, please refer to pseudo code in “Power Management – PM Working Mechanism”.

### 3.2.8 Data Length Extension

BLE Spec core\_4.2 and above supports Data Length Extension (DLE). Please refer to “Core\_v5.0” (Vol 6/Part B/2.4.2.21 “LL\_LENGTH\_REQ and LL\_LENGTH\_RSP”).

Link Layer in this BLE SDK supports data length extension to max rf\_len of 251 bytes per BLE Spec. Following steps explains how to use data length extension.

#### 1) Configure suitable TX & RX fifo size

To receive and transmit long packet, bigger Tx & Rx fifo size is required and thus occupies large SRAM space. So be cautious when setting fifo size to avoid waste of SRAM space.

Tx fifo size should be increased to transmit long packet. Tx fifo size should be larger than Tx rf\_len by 12, and it should be integer multiples of 4, for example:

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(blt\_txfifo, 204, 8);

Rx fifo size should be increased to receive long packet. Rx fifo size should be larger than Rx rf\_len by 24, and it should be integer multiples of 16, for example:

RX rf\_len = 56 bytes: MYFIFO\_INIT(blt\_rxfifo, 80, 8);

RX rf\_len = 141 bytes: MYFIFO\_INIT(blt\_rxfifo, 176, 8);

RX rf\_len = 191 bytes: MYFIFO\_INIT(blt\_rxfifo, 224, 8);

E.g. The setting below applies if both Tx and Rx need to support up to 200bytes:

MYFIFO\_INIT(blt\_txfifo, 212, 8);

MYFIFO\_INIT(blt\_rxfifo, 224, 8);

#### 2) data length exchange

Before transfer of long packets, please make sure the “data length exchange” flow has already been completed in BLE connection.

Data length exchange is an interactive process in Link Layer by LL\_LENGTH\_REQ and LL\_LENGTH\_RSP. Either master or slave can initiate the process by sending LL\_LENGTH\_REQ, while the peer responds with LL\_LENGTH\_RSP.

Through this interaction, master and slave obtain the max Tx and Rx packet size from each other, and adopt the minimum of the two as the max Tx and Rx packet size in current connection.

No matter which side initiates LL\_LENGTH\_REQ, at the end of data length exchange process, the SDK will generate

“BLT\_EV\_FLAG\_DATA\_LENGTH\_EXCHANGE” event callback assuming this callback has been registered. User can refer to “Telink defined event” section to understand the parameters of this event callback function.

The final max Tx and Rx packet size can be obtained from the “BLT\_EV\_FLAG\_DATA\_LENGTH\_EXCHANGE” event callback function.

When 8x5x acts as slave device in actual applications, master may or may not initiate LL\_LENGTH\_REQ. If master does not initiate it, slave should initiate LL\_LENGTH\_REQ by the following API in the SDK:

```
ble_sts_t b1c_ll_exchangeDataLength (u8 opcode, u16
maxTxOct);
```

In this API, “opcode” is “LL\_LENGTH\_REQ”, and “maxTxOct” is the max Tx packet size supported by current device.

For example, if max Tx packet size is 200bytes, the setting below applies:

```
b1c_ll_exchangeDataLength(LL_LENGTH_REQ, 200);
```

Since slave does not know whether master would initiate LL\_LENGTH\_REQ, a recommended approach is:

Register a BLT\_EV\_FLAG\_DATA\_LENGTH\_EXCHANGE event callback; after connection is established, start a software timer (e.g. 2s).

When the timer expires, if the callback is not triggered yet, it means master has not initiated LL\_LENGTH\_REQ. Slave can then initiate LL\_LENGTH\_REQ via the API “b1c\_ll\_exchangeDataLength”.

### 3) MTU size exchange

In addition to data length exchange, MTU size exchange flow should also be executed to ensure large MTU size takes effect, so that the peer can process long packet in BLE L2CAP layer. MTU size should be equal or larger than max packet size of Tx & Rx.

Please refer to “ATT & GATT” section or the demo of the 8258\_feature\_test for the implementation of MTU size exchange.

### 4) Transmission/Reception of long packet

Please refer to “ATT & GATT” section for illustration of Handle Value Notification, Handle Value Indication, Write request and Write Command.

Transmission and reception of long packet can start after correct completion of the three steps above.

The APIs corresponding to “Handle Value Notification” and “Handle Value Indication” can be invoked in ATT layer to transmit long packet.

As shown below, fill in the address and length of data to be sent to the parameters “\*p” and “len”, respectively:

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

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

To receive long packet, it's only needed to use callback function “w” corresponding to “Write Request” and “Write Command” as explained in “ATT & GATT” section.

### 3.2.9 Controller API

#### 3.2.9.1 Controller API brief

In standard BLE stack architecture (see Figure3-1), APP layer cannot directly communicate with Link Layer of Controller, i.e. data of APP layer must be first transferred to Host, and then Host can transfer control command to Link Layer via HCI. All control commands from Host to LL via HCI follow the definition in BLE spec “Core\_v5.0”, please refer to “Core\_v5.0” (Vol 2/Part E/ Host Controller Interface Functional Specification) for more information.

Telink BLE SDK based on standard BLE architecture can serve as a Controller and work together with Host system. Therefore, all APIs to operate Link Layer strictly follow the data format of Host commands in the spec.

Although the architecture in Figure3-4 is used in Telink BLE SDK, during which APP layer can directly operate Link Layer, it still use the standard APIs of HCI part.

In BLE spec, all HCI commands to operate Controller have corresponding “HCI command complete event” or “HCI command status event” in response to Host layer. However, in Telink BLE SDK, it is handled case by case:

- 1) For applications such as 8258\_hci, Telink IC only serves as BLE controller, and needs to work together with BLE Host MCU from other vendors. Each HCI command will generate corresponding “HCI command complete event” or “HCI command status event”.
- 2) For applications such as 8258 master kma dongle, both BLE Host and Controller run on Telink IC. When Host invokes interface to send HCI command to Controller, Controller can receive all data correctly without loss. Therefore, when Controller processes HCI command, it won't reply

with “HCl command complete event” or “HCl command status event”.

Controller API declaration is available in head files under “stack/ble/ll” and “stack/ble/hci”. Corresponding to Link Layer state machine functions, the “ll” directory contains ll.h, ll\_adv.h, ll\_scan.h, ll\_init.h, ll\_slave.h and ll\_master.h, e.g. APIs related to advertising function should be in ll\_adv.h.

### 3.2.9.2 API return type ble\_sts\_t

An enum type “ble\_sts\_t” defined in the “stack/ble/ble\_common.h” is used as return value type for most APIs in the SDK.

When API invoking with right parameter setting is accepted by the protocol stack, it will return “0” to indicate BLE\_SUCCESS; if any non-zero value is returned, it indicates a unique error type. All possible return values and corresponding error reason will be listed in the subsections below for each API.

The “ble\_sts\_t” applies to both APIs in Link Layer and some APIs in Host layer.

### 3.2.9.3 MAC address initialization

In this document, “BLE MAC address” includes both “public address” and “random static address”.

In this BLE SDK, the API below serves to obtain public address and random static address:

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

“flash\_addr” is the flash address to store MAC address. As explained earlier, this address in 8x5x 512kB flash is 0x76000.

If random static address is not needed, set “mac\_random\_static” as “NULL”.

The Link Layer initialization API can be invoked to load the obtained MAC address into BLE protocol stack:

```
blc_ll_initStandby_module (tbl_mac); //mandatory
```

In order to use Advertising state or Scanning state in Link Layer state machine, it's also needed to load MAC address, as shown below:

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

### 3.2.9.4 Link Layer state machine initialization

The APIs below serve to configure initializaion of each module when BLE state machine is established. Please refer to introduction of Link Layer state machine.

```

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

Please refer to “Core\_v5.0” (Vol 2/Part E/ 7.8.7 “LE Set Advertising Data Command”).

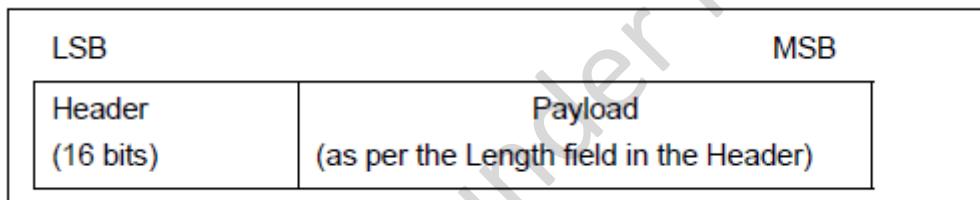


Figure3-32 Adv packet format in BLE stack

As shown above, an Adv packet in BLE stack contains 2-byte header, and Payload (PDU). The maximum length of Payload is 31 bytes.

The API below serves to set PDU data of adv packet:

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

Note: The “data” pointer points to the starting address of the PDU, while the “len” indicates data length.

The table below lists possible results for the return type “ble\_sts\_t”.

| <b>ble_sts_t</b>               | <b>Value</b> | <b>ERR Reason</b>                  |
|--------------------------------|--------------|------------------------------------|
| BLE_SUCCESS                    | 0            |                                    |
| HCI_ERR_INVALID_HCI_CMD_PARAMS | 0x12         | Len exceeds the maximum length 31. |

This API can be invoked during initialization to set adv data, or invoked in main\_loop to modify adv data when firmware is running.

In the “8258 ble remote” project of this BLE SDK, Adv PDU definition is shown as below. Please refer to “Data Type Specification” in BLE Spec “CSS v6” (Core Specification Supplement v6.0) for introduction to various fields.

```
u8 tbl_advData[] = {
 0x05, 0x09, 'k', 'h', 'i', 'd',
 0x02, 0x01, 0x05,
 0x03, 0x19, 0x80, 0x01,
 0x05, 0x02, 0x12, 0x18, 0x0F, 0x18,
};

};
```

As shown in the adv data above, the adv device name is set as "khid".

### 3.2.9.6 bls\_ll\_setScanRspData

Please refer to “Core\_v5.0” (Vol 2/Part E/ 7.8.8 “LE Set Scan response Data Command”).

The API below serves to set PDU data of scan response packet.

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

Note: The “data” pointer points to the starting address of the PDU, while the “len” indicates data length.

The table below lists possible results for the return type “ble\_sts\_t”.

| ble_sts_t                      | Value | ERR Reason                         |
|--------------------------------|-------|------------------------------------|
| BLE_SUCCESS                    | 0     |                                    |
| HCI_ERR_INVALID_HCI_CMD_PARAMS | 0x12  | Len exceeds the maximum length 31. |

This API can be invoked during initialization to set Scan response data, or invoked in main\_loop to modify Scan response data when firmware is running.

In the “8258 ble remote” project of this BLE SDK, definition of Scan response data is shown as below. Please refer to “Data Type Specification” in BLE Spec “CSS v6” (Core Specification Supplement v6.0) for introduction to various fields.

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

};
```

As shown in the Scan response data above, the scan device name is set as "KRemote".

Since device name configured in Adv data and scan response data differ, the device name scanned by a mobile phone or IOS system may be different:

- 1) If some device only listens for Adv packets, the scanned device name is "khid".
- 2) If some device sends scan request after Adv packet is received, and reads the scan response, the scanned device name may be "KRemote".

User can configure device name in the two packets (Adv packet & scan response packet) as the same name, so that the scanned device name is consistent.

Actually when Master reads device's Attribute Table after connection is established, the obtained "gap device name" of device will be shown according to the configuration in Attribute Table. Please refer to Attribute Table section for details.

### 3.2.9.7 bls\_ll\_setAdvParam

Please refer to "Core\_v5.0" (Vol 2/Part E/ 7.8.5 "LE Set Advertising Parameters Command").

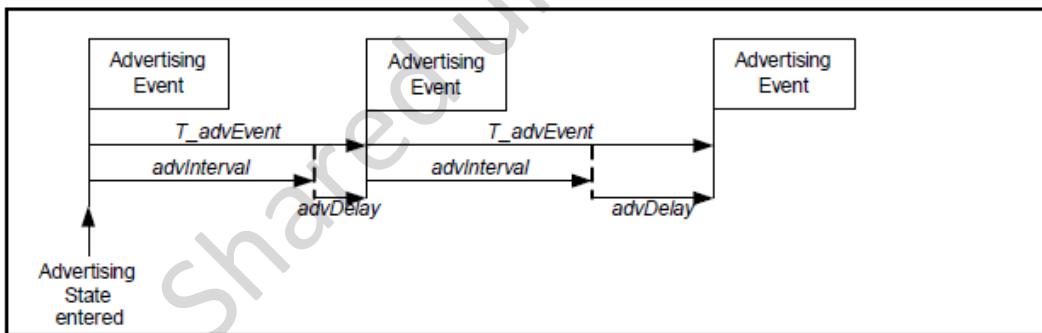


Figure3-33 Advertising Event in BLE stack

The figure above shows Advertising Event (Adv Event in brief) in BLE stack. It indicates during each  $T_{advEvent}$ , Slave implements one advertising process, and sends one packet in three advertising channels (channel 37, 38 and 39) respectively.

The API below serves to set parameters related to Adv Event.

```

ble_sts_t bls_ll_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 & intervalMax

The two parameters serve to set the range of advertising interval in integer multiples of 0.625ms. The valid range is from 20ms to 10.24s, and intervalMin should not exceed intervalMax.

As required by BLE spec, it's not recommended to set adv interval as fixed value; in Telink BLE SDK, the eventual adv interval is random variable within the range of intervalMin ~ intervalMax. If intervalMin and intervalMax are set as same value, adv interval will be fixed as the intervalMin.

Adv packet type has limits to the setting of intervalMin and intervalMax. Please refer to "Core 5.0" (Vol 6/Part B/ 4.4.2.2 "Advertising Events") for details.

## 2) advType

AS per BLE spec, the following four basic advertising event types are supported.

| 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

Figure3-34 Four adv events in BLE stack

In the "Allowable response PDUs for advertising event" column, "YES" and "NO" indicate whether corresponding adv event type can respond to "Scan request" and "Connect Request" from other device. For example, "Connectable Undirected Event" can respond to both "Scan request" and "Connect Request", while "Non-connectable Undirected Event" will respond to neither "Scan request" nor "Connect Request".

For "Connectable Directed Event", "YES" marked with an asterisk indicates the matched "Connect Request" received won't be filtered by whitelist and this event will surely respond to it. Other "YES" not marked with asterisk indicate corresponding request can be responded depending on the setting of whitelist filter.

The “Connectable Directed Event” supports two sub-types including “Low Duty Cycle Directed Advertising” and “High Duty Cycle Directed Advertising”. Therefore, five types of adv events are supported in all, as defined in the “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;
```

By default, the most common adv event type is  
“[ADV\\_TYPE\\_CONNECTABLE\\_UNDIRECTED](#)”.

### 3) ownAddrType

There are four optional values for “ownAddrType” to specify adv address type.

```
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;
```

First two parameters are explained herein.

The “OWN\_ADDRESS\_PUBLIC” indicates that public MAC address is used during advertising. Actual address is the setting from the API “[blc\\_ll\\_initAdvertising\\_module\(u8 \\*public\\_addr\)](#)” during MAC address initialization.

The “OWN\_ADDRESS\_RANDOM” indicates random static MAC address is used during advertising, and the address comes from the setting of the API below:

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

#### 4) peerAddrType & \*peerAddr

When advType is set as directed adv type (ADV\_TYPE\_CONNECTABLE\_DIRECTED\_HIGH\_DUTY or ADV\_TYPE\_CONNECTABLE\_DIRECTED\_LOW\_DUTY), the “peerAddrType” and “\*peerAddr” serve to specify the type and address of peer device MAC Address.

When advType is set as type other than directed adv, the two parameters are invalid, and they can be set as “0” and “NULL”.

#### 5) adv\_channelMap

The “adv\_channelMap” serves to set advertising channel. It can be selectable from channel 37, 38, 39 or combination.

```
#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

The “advFilterPolicy” serves to set filtering policy for scan request/connect request from other device when adv packet is transmitted. Address to be filtered needs to be pre-loaded in whitelist.

Filtering type options are shown as below. The “ADV\_FP\_NONE” can be selected if whitelist filter is not needed.

```
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;
```

The table below lists possible values and reasons for the return value “ble\_sts\_t”.

| ble_sts_t                      | Value | ERR Reason         |
|--------------------------------|-------|--------------------|
| BLE_SUCCESS                    | 0     |                    |
| HCI_ERR_INVALID_HCI_CMD_PARAMS | 0x12  | The intervalMin or |

|  |  |                                                              |
|--|--|--------------------------------------------------------------|
|  |  | intervalMax value does not meet the requirement of BLE spec. |
|--|--|--------------------------------------------------------------|

According to Host command design in HCI part of BLE spec, eight parameters are configured simultaneously by the “bls\_ll\_setAdvParam” API. This setting also takes some coupling parameters into consideration. For example, the “advType” has limits to the setting of intervalMin and intervalMax, and range check depends on the advType; if advType and advInterval are set in two APIs, the range check is uncontrollable.

Since user often needs to modify some parameters, three independent APIs are supplied, so that user can directly invoke one API to modify corresponding parameter(s), rather than invoking the “bls\_ll\_setAdvParam” to set eight parameters simultaneously.

```
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);
```

Please refer to the “bls\_ll\_setAdvParam” API for the parameters of the three APIs above.

Return value `ble_sts_t`:

- 1) “bls\_ll\_setAdvChannelMap” and “bls\_ll\_setAdvFilterPolicy” will always return “BLE\_SUCCESS”.
- 2) “bls\_ll\_setAdvInterval” will return “BLE\_SUCCESS” or “HCI\_ERR\_INVALID\_HCI\_CMD\_PARAMS”.

### 3.2.9.8 bls\_ll\_setAdvEnable

Please refer to “Core\_v5.0” (Vol 2/Part E/ 7.8.9 “LE Set Advertising Enable Command”).

```
ble_sts_t bls_ll_setAdvEnable(int en);
```

en”: 1 - Enable Advertising; 0 - Disable Advertising.

- 1) In Idle state, by enabling Advertising, Link Layer will enter Advertising state.
- 2) In Advertising state, by disabling Advertising, Link Layer will enter Idle state.
- 3) In other states, Link Layer state won’t be influenced by enabling or disabling Advertising.

- 4) `ble_sts_t` will always return “BLE\_SUCCESS”.

### 3.2.9.9 `bls_ll_setAdvDuration`

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

After the “`bls_ll_setAdvParam`” is invoked to set all adv parameters successfully, and the “`bls_ll_setAdvEnable (1)`” is invoked to start advertising, the API “`bls_ll_setAdvDuration`” can be invoked to set duration of adv event, so that advertising will be automatically disabled after this duration.

“`duration_en`”: 1-enable timing function; 0-disable timing function.

“`duration_us`”: The “`duration_us`” is valid only when the “`duration_en`” is set as 1, and it indicates the advertising duration in unit of us. When this duration expires, “AdvEnable” becomes unvalid, and advertising is stopped. None Conn state will switch to Idle State. The Link Layer event “`BLT_EV_FLAG_ADV_DURATION_TIMEOUT`” will be triggered.

As specified in BLE spec, for the adv type “`ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY`”, the duration time is fixed as 1.28s, i.e. advertising will be stopped after the 1.28s duration. Therefore, for this adv type, the setting of “`bls_ll_setAdvDuration`” won’t take effect.

The return value “`ble_sts_t`” is shown as below.

| <code>ble_sts_t</code>         | Value | ERR Reason                                                                                      |
|--------------------------------|-------|-------------------------------------------------------------------------------------------------|
| BLE_SUCCESS                    | 0     |                                                                                                 |
| HCI_ERR_INVALID_HCI_CMD_PARAMS | 0x12  | Duration Time can't be configured for “ <code>ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY</code> ”. |

When Adv Duratrion Time expires, advertising is stopped, if user needs to re-configure adv parameters (such as AdvType, AdvInterval, AdvChannelMap), first the parameters should be set in the callback function of the event “`BLT_EV_FLAG_ADV_DURATION_TIMEOUT`”, then the “`bls_ll_setAdvEnable (1)`” should be invoked to start new advertising.

To trigger the “`BLT_EV_FLAG_ADV_DURATION_TIMEOUT`”, a special case should be noted:

Suppose the “`duration_us`” is set as “2000000” (i.e. 2s).

- If Slave stays in advertising state, when adv time reaches the preset 2s timeout,

- the “BLT\_EV\_FLAG\_ADV\_DURATION\_TIMEOUT” will be triggered to execute corresponding callback function.
- If Slave is connected with Master when adv time is less than the 2s timeout (suppose adv time is 0.5s), the timeout timing is not cleared but cached in bottom layer. When Slave stays in connection state for 1.5s (i.e. the preset 2s timeout moment is reached), since Slave won’t check adv event timeout in connection state, the callback of “BLT\_EV\_FLAG\_ADV\_DURATION\_TIMEOUT” won’t be triggered. When Slave stays in connection state for certain duration (e.g. 10s), then terminates connection and returns to adv state, before it sends out the first adv packet, the Stack will regard current time exceeds the preset 2s timeout and trigger the callback of “BLT\_EV\_FLAG\_ADV\_DURATION\_TIMEOUT”. In this case, the callback triggering time largely exceeds the preset timeout moment. User should be aware of this situation.

### 3.2.9.10 blc\_ll\_setAdvCustomedChannel

The API below serves to customize special advertising channel/scanning channel, and it only applies some special applications such as BLE mesh. It’s not recommended to use this API for other conventional application cases.

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

chn0/chn1/chn2: customized channel. Default standard channel is 37/38/39.

For example, to set three advertising channels as 2420MHz, 2430MHz and 2450MHz, the API below should be invoked:

```
blc_ll_setAdvCustomedChannel (8, 12, 22);
```

### 3.2.9.11 rf\_set\_power\_level\_index

This BLE SDK supplies the API to set output power for BLE RF packet, as shown below.

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

The “level” is selectable from the corresponding enum variable `RF_PowerTypeDef` in the “drivers/8258/rf\_drv.h”.

The Tx power configured by this API will take effect for both adv packet and conn packet, and it can be set freely in firmware. The actual Tx power will be determined by the latest setting.

Please note that the “`rf_set_power_level_index`” configures registers related to MCU RF. Once MCU enters sleep (suspend/deepsleep retention), these registers’ values will be lost, so they should be reconfigured after each wakeup. For

example, SDK demo employs the event callback “BLT\_EV\_FLAG\_SUSPEND\_EXIT” to guarantee RF power is recovered after wakeup from sleep.

```
_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

Please refer to “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);
```

#### Parameters:

##### 1) scan\_type

This parameter can be set as “passive scan” or “active scan”. The difference is: For active scan, when adv packet is received, scan\_req will be sent to obtain more information of scan\_rsp, and scan rsp packet will also be transmitted to BLE Host via adv report event. For passive scan, scan req won’t be sent.

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

##### 2) scan\_inetrvl(scan window)

“scan\_interval” serves to set channel switch time in Scanning state (unit: 0.625ms).

“scan\_window” is not processed in current Telink BLE SDK. Actual scan window is set as scan\_interval.

### 3) ownAddrType

There are four optional values for “ownAddrType” to specify address type of scan req packet.

```
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;
```

The “OWN\_ADDRESS\_PUBLIC” indicates public MAC address is used during scan. Actual address is the setting from the API “blc\_ll\_initScanning\_module(u8 \*public\_addr)” during MAC address initialization.

The “OWN\_ADDRESS\_RANDOM” indicates random static MAC address is used during scan, and the address comes from the setting of the API below:

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

### 4) filter\_policy

Currently supported scan filter policies include:

```
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” indicates Link Layer won’t filter scanned adv packet, but directly report it to BLE Host.

“SCAN\_FP\_ALLOW\_ADV\_WL” indicates scanned adv packet must be in whitelist so that it can be reported to BLE Host.

The return value “ble\_sts\_t” is always “BLE\_SUCCESS”. Since API won’t check rationality of parameters, user should pay attention to this point when setting parameters.

### 3.2.9.13 blc\_ll\_setScanEnable

Please refer to “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);
```

There are two optional values for the parameter type of the “scan\_enable”.

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

“scan\_enable”: 1 - Enable Scanning; 0 - Disable Scanning.

- 1) In Idle state, by enabling Scanning, Link Layer will enter Scanning state.
- 2) In Scanning state, by disabling Scanning, Link Layer will enter Idle state.

There are two optional values for the parameter type of the “filter\_duplicate”.

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

“filter\_duplicate”: If it’s set as 1, it indicates enabling filter for repeated packet, i.e. for each different adv packet, Controller only reports one “adv report event” to Host. If it’s set as 0, it indicates disabling filter for repeated packet, i.e. all scanned adv packets will be reported to Host.

The return value “ble\_sts\_t” is shown as below.

| ble_sts_t                                   | Value                         | ERR Reason                                                      |
|---------------------------------------------|-------------------------------|-----------------------------------------------------------------|
| BLE_SUCCESS                                 | 0                             |                                                                 |
| LL_ERR_CURRENT_STATE_NOT_SUPPORTED_THIS_CMD | See the definition in the SDK | Link Layer is in BLS_LINK_STATE_ADV /BLS_LINK_STATE_CONN state. |

When “scan\_type” is set as “active scan”, and Scanning is enabled, for each device, scan\_rsp will be read only once and reported to Host. Since after each “enable scanning”, Controller will record and store scan\_resp of different devices in a scan\_rsp list, thus scan\_req won’t be read from the device repeatedly.

In order to report scan\_rsp of a device for multiple times, user can use “blc\_ll\_setScanEnable” to repeatedly set “Enable Scanning”, since scan\_rsp list will be cleared for each “Enable/Disable Scanning”.

### 3.2.9.14 blc\_ll\_createConnection

Please refer to “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\_inetrvl/scan window

“scan\_interval” serves to set Scan channel switch time in Initiating state (unit: 0.625ms).

“scan\_window” is not processed in current Telink BLE SDK. Actual scan window is set as scan\_interval.

#### 2) initiator\_filter\_policy

This parameter serves to specify device filter policy for current connection, and it has two options as shown below:

```
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” indicates device address of connection is specified by adr\_type/mac;

“INITIATE\_FP\_ADV\_WL” device connection depends on whitelist rather than adr\_type/mac.

#### 3) adr\_type/ mac

When “initiator\_filter\_policy” is set as “INITIATE\_FP\_ADV\_SPECIFY”, the device with address type of adr\_type (BLE\_ADDR\_PUBLIC or BLE\_ADDR\_RANDOM) and

address of mac[5...0] will be connected.

#### 4) own\_addr\_type

This parameter serves to specify MAC address type used by Master role to establish connection. “ownAddrType” has four optional values, as shown below.

```
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;
```

The “OWN\_ADDRESS\_PUBLIC” indicates public MAC address is used during connection. Actual address is the setting from the API “blc\_ll\_initStandby\_module (u8 \*public\_addr)” during MAC address initialization.

The “OWN\_ADDRESS\_RANDOM” indicates random static MAC address is used during connection, and the address comes from the setting of the API below:

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

#### 5) conn\_min/ conn\_max/ conn\_latency/ timeout

The four parameters specify connection parameters of Master role after connection is established. Since these parameters will be sent to Slave via “connection request”, Slave will use the same connection parameters.

“conn\_min” and “conn\_max” specify the range of conn interval. In Telink BLE SDK, Master role Single Connection directly uses the value of “conn\_min”. Unit is 0.625ms.

“conn\_latency” specifies connection latency, and generally it's set as 0.

“timeout” specifies connection supervision timeout in unit of 10ms.

#### 6) ce\_min/ ce\_max

“ce\_min”/“ce\_max” are not processed in current SDK.

The return value is shown as below.

| ble_sts_t                          | Value | ERR Reason                                                                            |
|------------------------------------|-------|---------------------------------------------------------------------------------------|
| BLE_SUCCESS                        | 0     |                                                                                       |
| HCI_ERR_CONN_REJ_LIMITED_RESOURCES | 0x0D  | Link Layer is already in Initiating state, and won't receive new “create connection”. |
| HCI_ERR_CONTROLLER_BUSY            | 0x3A  | Link Layer is in Advertising state or                                                 |

|  |  |                   |
|--|--|-------------------|
|  |  | Connection state. |
|--|--|-------------------|

### 3.2.9.15 blc\_ll\_setCreateConnectionTimeout

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

The return value is “BLE\_SUCCESS”, and the unit of “timeout\_ms” is ms.

As introduced in Link Layer state machine, when “blc\_ll\_createConnection” triggers Idle state/Scanning state to enter Initiating state, if if connection fails to be established until “Initiate timeout” is triggered, it will exit Initiating state.

Whenever “blc\_ll\_createConnection” is invoked, by default current “Initiate timeout” is set as “connection supervision timeout \*2” in SDK. User can modify this “Initiate timeout” as needed by invoking “blc\_ll\_setCreateConnectionTimeout” following “blc\_ll\_createConnection”.

### 3.2.9.16 bIm\_ll\_updateConnection

Please refer to “Core\_v5.0” (Vol 2/Part E/ 7.8.18 “LE Connection Update Command”).

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

1) connection handle

This parameter serves to specify connection to updata connection parameters.

2) conn\_min/ conn\_max/ conn\_latency/ timeout

The four parameters serve to specify new connection parameters. Currently “Master role single connection” directly use “conn\_min” as new interval.

3) ce\_min/ce\_max

The two parameters are not processed currently.

The return value “ble\_sts\_t” is always “BLE\_SUCCESS”. Since API won’t check rationality of parameters, user should pay attention to this point when setting parameters.

### 3.2.9.17 bls\_ll\_terminateConnection

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

This API is used for BLE Slave device, and it only applies to Connection state Slave role.

In order to actively terminate connection, this API can be invoked by APP Layer to send a “Terminate” to Master in Link Layer. “reason” indicates reason for disconnection. Please refer to “Core\_v5.0” (Vol 2/Part D/2 “Error Code Descriptions”).

If connection is not terminated due to system operation abnormality, generally APP layer specifies the “reason” as:

```
HCI_ERR_REMOTE_USER_TERM_CONN = 0x13
bls_ll_terminateConnection(HCI_ERR_REMOTE_USER_TERM_CONN);
```

In bottom-layer stack of Telink BLE SDK, this API is invoked only in one case to actively terminate connection: When data packets from peer device are decrypted, if an authentication data MIC error is detected, the “bls\_ll\_terminateConnection(HCI\_ERR\_CONN\_TERM\_MIC\_FAILURE)” will be invoked to inform the peer device of the decryption error, and connection is terminated.

After Slave invokes this API to actively initiate disconnection, the event “BLT\_EV\_FLAG\_TERMINATE” will be triggered. The terminate reason in the callback function of this event will be consistent with the reason manually configured in this API.

In Connection state Slave role, generally connection will be terminated successfully by invoking this API; however, in some special cases, the API may fail to terminate connection, and the error reason will be indicated by the return value “`ble_sts_t`”. It’s recommended to check whether the return value is “BLE\_SUCCESS” when this API is invoked by APP layer.

| <code>ble_sts_t</code>     | Value | ERR Reason                                                                                         |
|----------------------------|-------|----------------------------------------------------------------------------------------------------|
| BLE_SUCCESS                | 0     |                                                                                                    |
| HCI_ERR_CONN_NOT_ESTABLISH | 0x3E  | Link Layer is not in Connection state Slave role.                                                  |
| HCI_ERR_CONTROLLER_BUSY    | 0x3A  | Controller busy (mass data are being transferred), this command cannot be accepted for the moment. |

### 3.2.9.18 blm\_ll\_disconnect

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

This API is used for BLE Master device and it only applies to Connection Master role.

This API is similar to the function of the API “API bls\_ll\_terminateConnection” of Slave role, except that a conn handle parameter is added. Since in Telink BLE SDK, Slave role design can only sustain single connection, while Master role supports multi connection, it’s necessary to specify connection handle of disconnect.

The return value is shown as below:

| <b>ble_sts_t</b>        | <b>Value</b> | <b>ERR Reason</b>                                                                                  |
|-------------------------|--------------|----------------------------------------------------------------------------------------------------|
| BLE_SUCCESS             | 0            |                                                                                                    |
| HCI_ERR_UNKNOWN_CONN_ID | 0x02         | Handle error, cannot find corresponding connection.                                                |
| HCI_ERR_CONTROLLER_BUSY | 0x3A         | Controller busy (mass data are being transferred), this command cannot be accepted for the moment. |

### 3.2.9.19 Get Connection Parameters

The following APIs serves to obtain current connection paramters including Connection Interval, Connection Latency and Connection Timeout (only apply to Slave role).

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

- 1) If return value is 0, it indicates current Link Layer state is None Conn state without connection parameters available.
- 2) The returned non-zero value indicates the corresponding parameter value.
  - ◊ Actual conn interval divided by 1.25ms will be returned by the API “bls\_ll\_getConnectionInterval”. Suppose current conn interval is 10ms, the return value should be 10ms/1.25ms=8.
  - ◊ Acutal Latency value will be returned by the API “bls\_ll\_getConnectionLatency”.

- ❖ Actual conn timeout divided by 10ms will be returned by the API “bls\_ll\_getConnectionTimeout”. Suppose current conn timeout is 1000ms, the return value would be 1000ms/10ms=100.

### 3.2.9.20 blc\_ll\_getCurrentState

The API below serves to obtain current Link Layer state.

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

User can invoke the “bls\_ll\_getCurrentState()” in APP layer to judge current state, e.g.

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

### 3.2.9.21 blc\_ll\_getLatestAvgRSSI

The API serves to obtain latest average RSSI of connected peer device after Link Layer enters Slave role or Master role.

```
u8 blc_ll_getLatestAvgRSSI(void)
```

The return value is u8-type rssi\_raw, and the real RSSI should be: rssi\_real = rssi\_raw - 110. Suppose the return value is 50, rssi = -60 db.

### 3.2.9.22 Whitelist & Resolvinglist

As introduced above, “filter\_policy” of Advertising/Scanning/Initiating state involves Whitelist, and actual operation may depend on devices in Whitelist. Actually Whitelist contains two parts: Whitelist and Resolvinglist.

User can check whether address type of peer device is RPA (Resolvable Private Address) via “peer\_addr\_type” and “peer\_addr”. The API below can be invoked directly.

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

Only non-RPA address can be stored in whitelist. In current SDK, whitelist can store up to four devices.

```
#define MAX_WHITE_LIST_SIZE
```

4

The API below serves to reset whitelist:

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

The return value is “BLE\_SUCCESS”.

The API below serves to add a device into whitelist:

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

The return value is shown as below.

| ble_sts_t                | Value | ERR Reason                             |
|--------------------------|-------|----------------------------------------|
| BLE_SUCCESS              | 0     | Add success                            |
| HCI_ERR_MEM_CAP_EXCEEDED | 0x07  | Whitelist is already full, add failure |

The API below serves to delete a device from whitelist:

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

The return value is “BLE\_SUCCESS”.

RPA (Resolvable Private Address) device needs to use Resolvinglist. To save RAM space, “Resolvinglist” can store up to two devices in current SDK.

```
#define MAX_WHITE_IRK_LIST_SIZE 2
```

The API below serves to reset Resolvinglist.

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

The return value is “BLE\_SUCCESS”.

The API below serves to enable/disable device address resolving for Resolvinglist.

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

The API below serves to add device using RPA address into Resolvinglist.

```
ble_sts_t ll_resolvingList_add(u8 peerIdAddrType, u8 *peerIdAddr, u8 *peer_irk, u8 *local_irk);
```

peerIdAddrType/ peerIdAddr and peer-irk indicate identity address and irk

declared by peer device. These information will be stored into flash during pairing encryption process, and corresponding interfaces to obtain the info are available in SMP part. “local\_irk” is not processed in current SDK, and it can be set as “NULL”.

The API below serves to delete a RPA device from Resolvinglist.

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

For usage of address filter based on Whitelist/Resolvinglist, please refer to “TEST\_WHITELIST” in feature test demo of the SDK.

### 3.2.10 Coded PHY/2M PHY

#### 3.2.10.1 Coded PHY/2M PHY

Coded PHY and 2M PHY are new features to <Core\_5.0>, this expands the BLE application scenario, Coded PHY includes S2 (500kbps) and S8 (125kbps) in order to support long range application. 2M PHY (2Mbps) improved the BLE bandwidth. Coded PHY and 2M PHY could be used under both the advertising channel and data channel when in connected state. Connected state application will be introduced in the following section, advertising channel application will be introduced in “Extended Advertising).

#### 3.2.10.2 Coded PHY/2M PHY Demo

In the Kite BLE SDK, in order to save the sram space, Code PHY and 2M PHY is disabled by default. If user wants to enable this feature, you can enable it manually.

You can refer to the Kite BLE SDK demo:

- ◆ Slave mode reference demo “8258\_feature\_test”

In the vendor/8258\_feature\_test/app\_config.h, macro definition:

```
#define FEATURE_TEST_MODE TEST_2M_CODED_PHY_CONNECTION
```

- ◆ Master mode reference demo “8258\_master\_kma\_dongle”

User could also use other vendor’s device, as long as it also supports Code PHY/2M PHY, it’ll work with Telink’s Slave mode device.

In the Master mode, the Coded PHY and 2M PHY are both disabled by default, user will need to enable it as follow:

In vendor/8258\_master\_kma\_dongle/app.c, add below API to **void user\_init(**void**)**

```
blc_ll_init2MPHYCodedPhy_feature();
```

### 3.2.10.3 Coded PHY/2M PHY API

1. API blc\_ll\_init2MPhyCodedPhy\_feature()

```
void blc_ll_init2MPhyCodedPhy_feature(void)
```

used to enable Code PHY and 2M PHY.

2. A new event - BLT\_EV\_FLAG\_PHY\_UPDATE is introduced to Telink Defined Event in order to support Coded and 2M PHY, the detail implementation could refer to section “Controller Event”

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);
```

This is a BLE Spec standard interface, please refer to <Core\_5.0>, Vol 2/Part7/7.8.49, “LE Set PHY Command”

connHandle:

slave mode: it should set to BLS\_CONN\_HANDLE;  
master mode: it should set to BLM\_CONN\_HANDLE.

For other parameters, please refer to Spec’s definition along with SDK’s enumeration definition.

4. API blc\_ll\_setDefaultConnCodingIndication()

```
ble_sts_t blc_ll_setDefaultConnCodingIndication(le_ci_prefer_t
prefer_CI);
```

This is not a standard interface, it is used when Peer Device uses API blc\_ll\_setPhy () to first send PHY\_Req request, the receipt could use this API to configure local device’s encoded mode (S2/S8)

### 3.2.11 Channel Selection Algorithm #2

Channel Selection Algorithm #2 is a new feature added to <Core\_5.0>, with a better interference avoidance capability. You can refer to <Core\_5.0> (Vol 6/Part B/4.5.8.3 “Channel Selection Algorithm #2”) for further information. Reference demo could be found in the following:

- ◆ Slave mode reference demo “Demo “8258\_feature\_test”

In vendor/8258\_feature\_test/app\_config.h, enable the following macro

```
#define FEATURE_TEST_MODE TEST_CSA2
```

1. If using <Core\_4.2> API, user could choose to use or not use the hopping algothrim#2. In Kite SDK, it is also not using it by default, if user want to use the hopping algothrim #2, user could use below API to choose the algorithm #.

```
void blc_ll_initChannelSelectionAlgorithm_2_feature(void)
```

2. If using <Core\_5.0> extended advertising and initiate connect through Extend ADV, user will have to use above API to choose Algorithm #2 according to the spec <Core\_5.0>. Because if the connection is initiated through Extended Adv, it'll choose Algorithm#2 by default, and on the othe hand, if only uses advertising, in order to save sram space, Algorithm #2 is not recommended.

- ◆ Mastermode referenc demo “8258\_master\_kma\_dongle”

In master mode, Algorighm#2 is also disabled by default, user will need to enable it from user\_init() with the same API.

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

### 3.2.12 Extended Advertising

#### 3.2.12.1 Extended Advertising Introduction

Extended Advertising is a new feature to <Core\_5.0>

Due to the new feature to Advertising in <Core\_5.0>, SDK has new APIs in order to support the legacy Advertising function in <Core\_4.2> and the new Advertising function in <Core\_5.0>. These APIs will be covered in later secions, named as <Core\_5.0> API. (following secion will use this name as reference) , and <Core\_4.2> APIs refered in section <Controller API>, like bls\_ll\_setAdvData(), bls\_ll\_setScanRspData(), bls\_ll\_setAdvParam(), will only support for <Core\_4.2>’s Advertising function, but not <Core\_5.0> Advertising new function.

Extended Advertising primary feature as following:

1. Increase the Advertising PDUs – In <Core\_4.2>, the Advertising PDU length is ranging from 6 to 37 bytes, and in <Core\_5.0>, the extended Advertising PDU is ranging from 0 to 255 bytes (single PDU). If the Advertising Data length > Adv PDU, it'll be fragmented into N Advertising PDU and send it out.
2. It could chose different PHYs (1Mbps, 2Mbps, 125kbps, 500kbps) based on different application.

### 3.2.12.2 Extended Advertising Demo setup

Extended Advertising Demo “8258\_feature\_test”:

Demo1: use to illustrate all the basic advertising functions in <Core\_5.0>

1. In vendor/8258\_feature\_test/app\_config.h, declares the following macro

```
#define FEATURE_TEST_MODE TEST_EXTENDED_ADVERTISING
```

2. Based on the type of Advertising, select the corresponding macro.

The demo could also test all the supported Advertising type in <Core\_5.0>, below are all the type that Kite SDK currently supported.

```
/* 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: Based on Demo1, enable the Coded PHY/2M PHY option

1. In vendor/8258\_feature\_test/app\_config.h, declares the following macro

```
#define FEATURE_TEST_MODE TEST_2M_CODED_PHY_EXT_ADV
```

2. Based on the type of Advertising and PHY mode, select the corresponding macro

Note: if you're seeing the following error, it could because the type of data you define, might need more than 16K sram, since Kite SDK is choosing 16K sram by default.

```
./VCLDK1/COMMAND/B10_S21C_G1M1.0 CC 010 2H ./VCLDK1/S250_Feature_BLE/Feature_Low_Gui.v
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 lt multimode.elf] Error 1
```

You can fix it by either options (also refer to section “Sram and Firmware space”)

1. Reduce the data that defined with “\_attribute\_data\_retention\_” to save the sram space
2. Change to deepsleep retention 32K Sram, you can refer to section “software bootloader” for further information.

### 3.2.12.3 Extended Advertising Related API

Extended Advertising is using module design. Due to the variable length of adv data length/scan response data where the maximum length will be up to more than 1000 bytes, instead of statically defining the maximum value in BLE stack that might waste the SRAM space, we leave the definition of SRAM space to developer, so that it would have the flexibility for user to review their needs to best use of the SRAM space.

Current SDK only support one Advertising set, but with the design that has flexibility to support multiple adv set for future as well. So you could see the APIs' parameters are all designed in the way to support multiple adv sets for future.

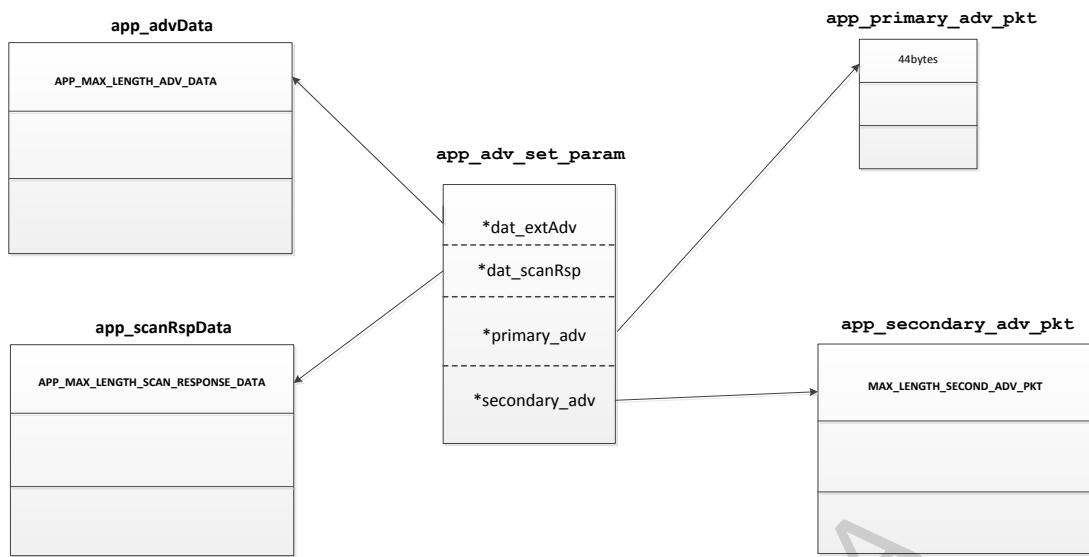
With that design, following are the APIs.

1. Initialization stage, you would need to call the following APIs to allocate the 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);
```

According to above API, the memory allocation is depicted as below:



- ◆ APP\_MAX\_LENGTH\_ADV\_DATA: Advertising Set length, developer could adjust the macro to define the size based on the needs in order to save the DeepRetention space.
- ◆ APP\_MAX\_LENGTH\_SCAN\_RESPONSE\_DATA: Scan response data length, developer could adjust the macro to define the size based on the needs in order to save the DeepRetention space.
- ◆ app\_primary\_adv\_pkt: Primary Advertising PDU data length, the size is allocated as 44 bytes, application can't change it.
- ◆ app\_Secondary\_adv\_pkt: Secondary Advertising PDU data length, the size is allocated as 264 bytes, application can't change it.

In the demo of “8258\_feature\_test”,  
 (vendor/8258\_feature\_test/feature\_2m\_coded\_phy\_adv.c), developer can use the following macro to allocate the sram based on your requirement in order to best use the sram.

```

#define APP_ADV_SETS_NUMBER 1 // Number of
Supported Advertising Sets (Current SDK only support one Adv Set)
#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. blc\_ll\_setExtAdvParam()

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

This is a BLE Spec standard interface, used to configure Advertising parameter, please refer to <Core\_5.0> (Vol 2/Part E/7.8.53 “LE Set Extended Advertising Parameters Command”) for further information.

Note: Adjusting the Tx Power is currently not supported, so the parameter Advertising\_Tx\_Power will not take effect, to adjust the TX power, please use the other APIs `void rf_set_power_level_index (RF_PowerTypeDef level)`

### 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);
```

This is a BLE Spec standard interface, used to configure the Scan Response Data, please refer to <Core\_5.0> (Vol 2/Part E/7.8.53 “LE Set Extended Scan Response Command”).

### 4. API blc\_ll\_setExtAdvEnable\_n()

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

This is a BLE Spec standard interface, used to enable/disable Extended Advertising, please refer to <Core\_5.0> (Vol 2/Part E/7.8.56 “LE Set Extended Advertising Enable Command”).

Since Kite SDK currently only support one Advertising Set, so this API is for future support, and not functioning at this moment, And a separate API is added to support one Advertising 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);
```

This is a BLE Spec standard interface, used to configure the Random address, please refer to <Core\_5.0> (Vol 2/Part E/7.8.4 “LE Set Random Address Command”).

### 6. API blc\_ll\_setDefaultExtAdvCodingIndication()

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

This is a none BLE Spec standard interface, if developer use BLE standard interface blc\_ll\_setExtAdvParam() to configure the Advertising parameters, and also if configure it to Coded PHY (for either S2 or S8) in the same time, but didn't specify which Coded PHY mode is, S2 will be chosen by default. This API is added for developer to specify the Coded PHY mode.

#### 7. API blc\_ll\_setAuxAdvChnIdxByCustomers()

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

This is a none BLE Spec standard interface, used to configure Auxiliary Advertising channel value, could be used for debug purpose, if developer didn't configure this value, Auxiliary Advertising channel will be randomly chosen in between 0-31.

#### 8. API blc\_ll\_setMaxAdvDelay\_for\_AdvEvent()

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

This is a none BLE Spec standard interface, used to configure the AdvDelay timing based on the Adv Interval, the input range is from 0, 1, 2, 4, 8 in the unit of ms.

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

If max\_delay\_ms = 0 , T\_advEvent is right on the advInterval timing  
If max\_delay\_ms = 8, T\_advEvent is based on the advInterval with a random offset in between 0-8ms.

#### 9. Following APIs are added for future Multiple Advertising Sets support, and is currently not supported in this SDK.

- ble\_sts\_t blc\_ll\_removeAdvSet(u8 advHandle);
- ble\_sts\_t blc\_ll\_clearAdvSets(void);

### 3.3 BLE Host

#### 3.3.1 BLE Host

BLE Host consists of L2CAP, ATT, SMP, GATT and GAP layer, and user-layer applications are implemented on the basis of the Host layer.

#### 3.3.2 L2CAP

The L2CAP, Logical Link Control and Adaptation Protocol, connects to the upper

APP layer and the lower Controller layer. By acting as an adaptor between the Host and the Controller, the L2CAP makes data processing details of the Controller become negligible to the upper-layer application operations.

The L2CAP layer of BLE is a simplified version of classical Bluetooth. In basic mode, it does not implement segmentation and re-assembly, has no involvement of flow control and re-transmission, and only uses fixed channels for communication.

The figure below shows simple L2CAP structure: Data of the APP layer are sent in packets to the BLE Controller. The BLE Controller assembles the received data into different CID data and report them to the Host layer.

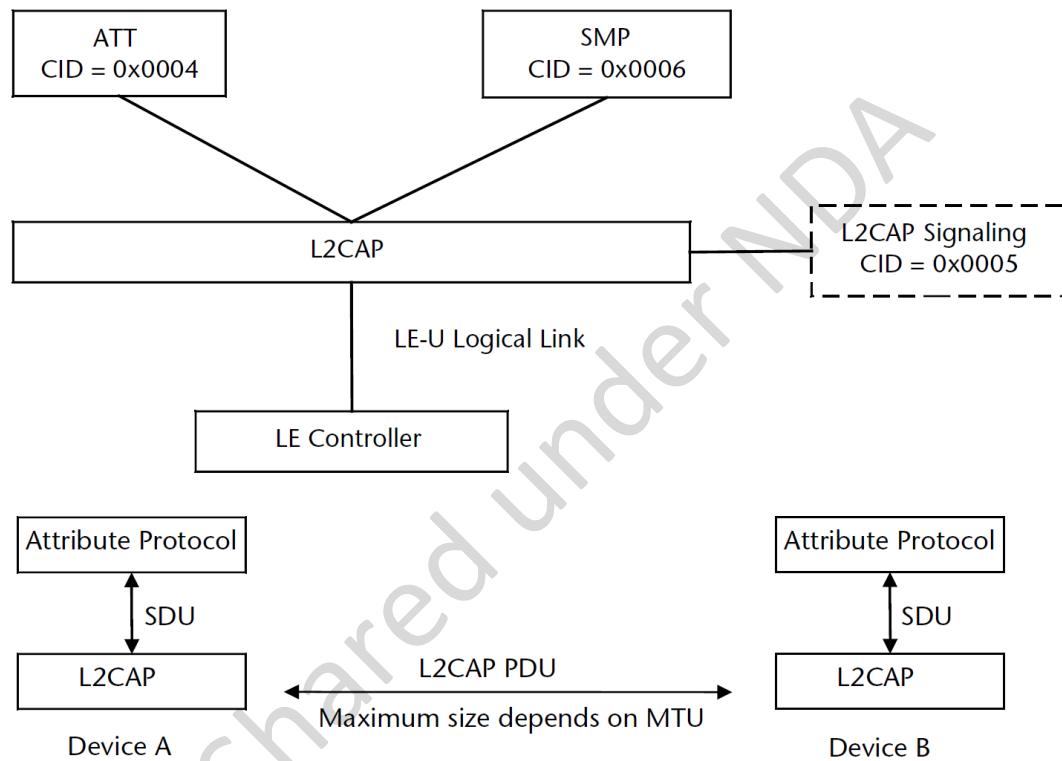


Figure 3-35 BLE L2CAP structure and ATT packet assembly model

As specified in BLE Spec, L2CAP is mainly used for data transfer between Controller and Host. Most work are finished in stack bottom layer with little involvement of user. User only needs to invoke the following APIs to set correspondingly.

### 3.3.2.1 Register L2CAP data processing function

In BLE SDK architecture, Controller transfers data with Host via HCI. Data from HCI to Host will be processed in L2CAP layer first. The API below serves to register this processing function.

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

In BLE Slave applications such as 8258 remote/8258 module, the function to process data of Controller in L2CAP layer of SDK is shown as below:

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

This function is already implemented in stack, which it will analyze the received data and transfer the data to ATT, SIG or SMP.

Initialization:

```
blc_l2cap_register_handler (blc_l2cap_packet_receive);
```

In 8258 master kma dongle, APP layer contains BLE Host function, and its processing function is supplied in source code for user reference:

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

Initialization:

```
blc_l2cap_register_handler (app_l2cap_handler);
```

In 8258 hci, only Slave controller is implemented. The function “blc\_hci\_sendACLDData2Host” serves to transmit data of controller to BLE Host device via hardware interface such as UART/USB.

```
int blc_hci_sendACLDData2Host (u16 handle, u8 *p)
```

Initialization: blc\_l2cap\_register\_handler (blc\_hci\_sendACLDData2Host);

### 3.3.2.2 Update connection parameters

#### 1) Slave requests for connection parameter update

In BLE stack, Slave can actively apply for a new set of connection parameters by sending a “CONNECTION PARAMETER UPDATE REQUEST” command to Master in L2CAP layer. The figure below shows the command format. Please refer to “Core\_v5.0” (Vol 3/Part A/ 4.20 “CONNECTION PARAMETER UPDATE REQUEST”).

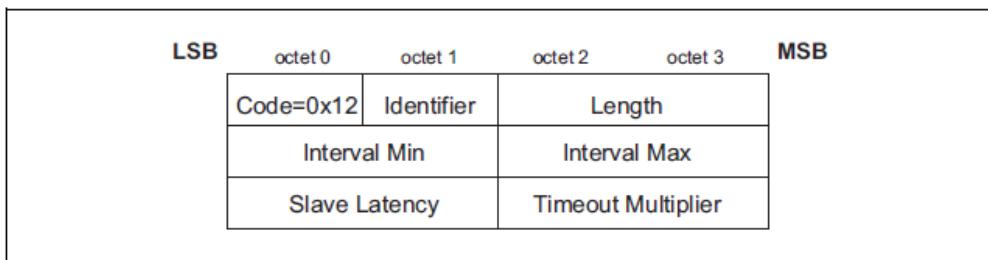


Figure 4.22: Connection Parameters Update Request Packet

Figure3-36 Connection Para update Req format in BLE stack

This BLE SDK supplies an API in L2CAP layer for Slave to send a “CONNECTION PARAMETER UPDATE REQUEST” command to Master and actively apply for a new set of connection parameters.

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

\*Note: The four parameters of this API correspond to the parameters in the “data” field of the “CONNECTION PARAMETER UPDATE REQUEST”. The “min\_interval” and “max\_interval” are the actual interval time divided by 1.25ms (e.g. for 7.5ms connection interval, the value should be 6); the “timeout” is actual supervision timeout divided by 10ms (e.g. for 1s timeout, the value should be 100).

**Application example:** Slave requests for new connection parameters when connection is established.

```
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
);
}
```

|     |           |                            |  |  |  |                     |  |  |                     |  |  |                                                        |  |  |  |        |
|-----|-----------|----------------------------|--|--|--|---------------------|--|--|---------------------|--|--|--------------------------------------------------------|--|--|--|--------|
| tus | Data Type | Data Header                |  |  |  | L2CAP Header        |  |  | SIG Pkt Header      |  |  | SIG_Connection_Param_Update_Req                        |  |  |  | CRC    |
|     | L2CAP-S   | LLID NESN SN MD PDU-Length |  |  |  | L2CAP-Length ChanId |  |  | Code Id Data-Length |  |  | IntervalMin IntervalMax SlaveLatency TimeoutMultiplier |  |  |  | 0x28D8 |
|     | L2CAP-S   |                            |  |  |  |                     |  |  |                     |  |  |                                                        |  |  |  |        |
|     | L2CAP-S   |                            |  |  |  |                     |  |  |                     |  |  |                                                        |  |  |  |        |
|     | Data Type | Data Header                |  |  |  | CRC                 |  |  | RSSI                |  |  | FCS                                                    |  |  |  |        |

Figure3-37 BLE sniffer packet sample: conn para update request & response

The API “`void bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(int time_ms)`” serves to make the Slave wait for `time_ms` miliseconds after connection is

established, and then invoke the API “bls\_l2cap\_requestConnParamUpdate” to update connection parameters.

After conection is established, if user only invokes the “bls\_l2cap\_requestConnParamUpdate”, the Slave will wait for 1s to execute this request command.

For Slave applications, the SDK provides register callback function interface of obtaining Conn\_UpdateRsp result, so as to inform user whether connection parameter update request from Slave is rejected or accepted by Master. As shown in the figure above, Master accepts Connection\_Param\_Update\_Req from Slave.

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

Please refer to the use case of Slave initialization:

```
blc_l2cap_register_handler (blc_l2cap_packet_receive);
```

Following shows the reference for the function “blc\_l2cap\_packet\_receive”.

```
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 responds to connection parameter update request

After Master receives the “CONNECTION PARAMETER UPDATE REQUEST” command from Slave, it will respond with a “CONNECTION PARAMETER UPDATE RESPONSE” command. Please refer to “Core\_v5.0” (Vol 3/Part A/ 4.20 “CONNECTION PARAMETER UPDATE RESPONSE”).

The figure below shows the command format: if “result” is “0x0000”, it indicates the request command is accepted; if “result” is “0x0001”, it indicates the request command is rejected. Whether actual Android/iOS device will accept or reject the connection parameter update request is determined by corresponding BLE Master.

User can refer to Master compatibility test.

As shown in Figure3-37, Master accepts the request.

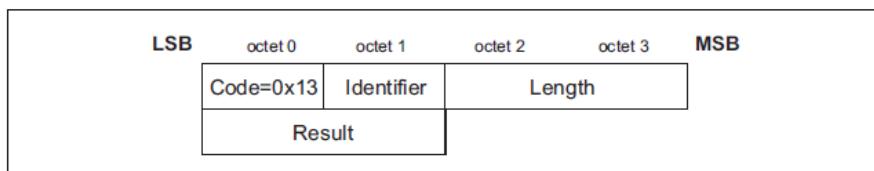


Figure 4.23: Connection Parameters Update Response Packet

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                       |

Figure3-38 conn para update rsp format in BLE stack

Following shows demo code to process connection parameter update request of Slave in Telink 8258 master kma dongle.

```

else if(ptrL2cap->chanId == L2CAP_CID_SIG_CHANNEL) //signal
{
 if(ptrL2cap->opcode == L2CAP_CMD_CONN_UPD PARA_REQ) //slave send conn param update req on l2cap
 {
 rf_packet_l2cap_connParaUpReq_t * req = (rf_packet_l2cap_connParaUpReq_t *)ptrL2cap;

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

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

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

```

After “L2CAP\_CMD\_CONN\_UPD PARA\_REQ” is received in “L2CAP\_CID\_SIG\_CHANNEL”, it will read interval\_min (used as eventual interval), supervision timeout and long suspend time (interval \* (latency +1)), and check the rationality of these data. If interval < 200ms, long suspend time<20s and supervision timeout >= 2\* long suspend time, this request will be accepted; otherwise this request will be rejected. User can modify as needed based on this simple demo design.

No matter whether parameter request of Slave is accepted, the API below can be invoked to respond to this request.

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

“connHandle” indicates current connection ID.

“result” has two options to indicate “accept” and “reject”, respectively.

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

If 8258 master kma dongle accepts request of Slave, it must send a update cmd to Controller via the API “blm\_ll\_updateConnection” within certain duration. In demo code, “host\_update\_conn\_param\_req” is used as mark, and a 50ms delay is set in main\_loop to initiate this 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_p {
 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 updates connection parameters in Link Layer

After Master responds with “conn para update rsp” to accept the “conn para update req” from Slave, Master will send a “LL\_CONNECTION\_UPDATE\_REQ” command in Link Layer.

|    |                        |                                                                 |                                          |                                                                                                                                    |
|----|------------------------|-----------------------------------------------------------------|------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| IS | Data Type<br>Control   | Data Header<br>LLID 1<br>NESN 1<br>SN 0<br>MD 12                | LL_Opcode<br>Connection_Update_Req(0x00) | LL_Connect_Update_Req<br>WinSize 0x02<br>WinOffset 0x001F<br>Interval 0x0006<br>Latency 0x0063<br>Timeout 0x0190<br>Instant 0x006C |
| IS | Data Type<br>Empty PDU | Data Header<br>LLID 1<br>NESN 0<br>SN 1<br>MD 0<br>PDU-Length 0 | CRC 0x8FE90F<br>RSSI (dBm) 0<br>FCS OK   |                                                                                                                                    |

Figure3-39 BLE sniffer packet sample: ll conn update req

Slave will mark the final parameter as the instant value of Master after it receives the update request. When the instant value of Slave reaches this value, connection parameters are updated, and the callback of the event “BLT\_EV\_FLAG\_CONN\_PARA\_UPDATE” is triggered.

The “instant” indicates connection event count value maintained by Master and Slave, and it ranges from 0x0000 to 0xffff. During a connection, Master and Slave should always have consistent “instant” value. When Master sends “conn\_req” and establishes connection with Slave, Master switches state from scanning to connection, and clears the “instant” of Master to “0”. When Slave receives the “conn\_req”, it switches state from advertising to connection, and clears the instant of Slave to “0”. Each connection packet of Master and Slave is a connection event. For the first connection event after the “conn\_req”, the instant value is “1”; for the second connection event, the instant value is 2, and so on.

When Master sends a “LL\_CONNECTION\_UPDATE\_REQ”, the final parameter “instant” indicates during the connection event marked with “instant”, Master will use the values corresponding to the former connection parameters of the “LL\_CONNECTION\_UPDATE\_REQ” packet. After the “LL\_CONNECTION\_UPDATE\_REQ” is received, the new connection parameters will be used during the connection event when the instant of Slave equals the declared instant of Master, thus Slave and Master can finish switch of connection parameters synchronously.

### 3.3.3 ATT & GATT

#### 3.3.3.1 GATT basic unit “Attribute”

GATT defines two roles: Server and Client. In this BLE SDK, Slave is Server, and corresponding Android/iOS device is Client. Server needs to supply multiple Services for Client to access.

Each Service of GATT consists of multiple Attributes, and each Attribute contains certain information.

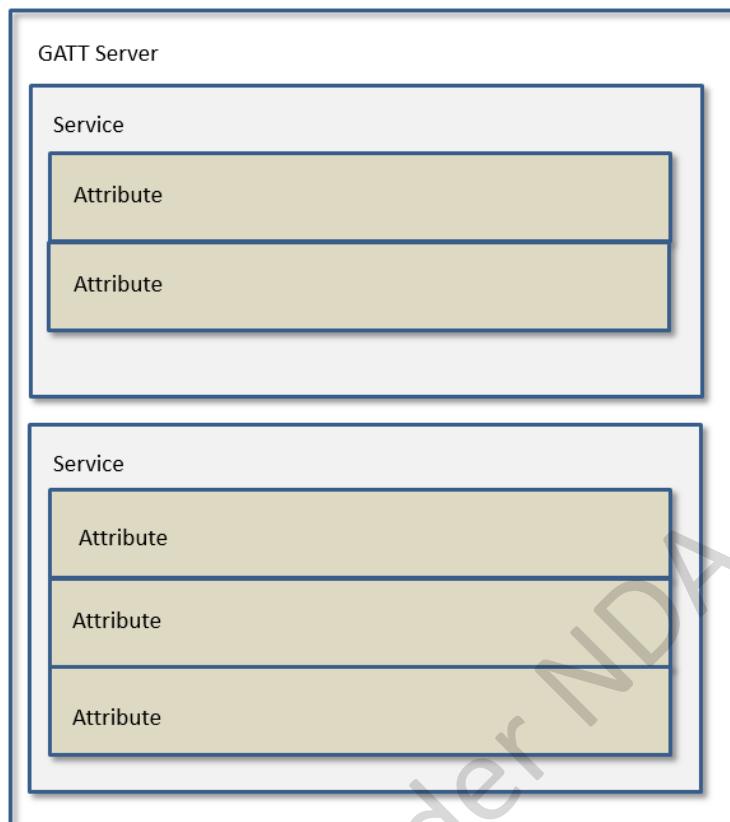


Figure3-40 GATT service containing Attribute group

The basic contents of Attribute are shown as below:

### 1) Attribute Type: UUID

The UUID is used to identify Attribute type, and its total length is 16 bytes. In BLE standard protocol, the UUID length is defined as two bytes, since Master devices follow the same method to transform 2-byte UUID into 16 bytes.

When standard 2-byte UUID is directly used, Master should know device types indicated by various UUIDs. 8x5x BLE stack defines some standard UUIDs in “stack/service/hids.h” and “stack/ble /uuid.h”.

Telink proprietary profiles (OTA, MIC, SPEAKER, and etc.) are not supported in standard Bluetooth. The 16-byte proprietary device UUIDs are defined in “stack/ble/uuid.h”.

### 2) Attribute Handle

Slave supports multiple Attributes which compose an Attribute Table. In Attribute Table, each Attribute is identified by an Attribute Handle value. After connection is established, Master will analyze and obtain the Attribute Table of Slave via “Service Discovery” process, then it can identify Attribute data via the Attribute Handle during data transfer.

### 3) Attribute Value

Attribute Value corresponding to each Attribute is used as request, response, notification and indication data. In 8x5x BLE stack, Attribute Value is indicated by one pointer and the length of the area pointed by the pointer.

#### 3.3.3.2 Attribute and ATT Table

To implement GATT Service on Slave, an Attribute Table is defined in this BLE SDK and it consists of multiple basic Attributes. Attribute definition is shown as below.

```
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;
```

Attribute Table code in this BLE SDK is available in “app\_att.c”, as shown below:

```
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_appearanceCharVal), (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_serviceChangeCharVal), 0},
{0,ATT_PERMISSIONS_READ,2,sizeof(serviceChangeVal),(u8*)(&serviceChangeUUID), (u8*)(&serviceChangeVal), 0},
{0,ATT_PERMISSIONS_RDWR,2,sizeof(serviceChangeCCC),(u8*)(&clientCharacterCfgUUID), (u8*)(&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_PnUUID), (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},
```

Figure3-41 Attribute Table in this BLE SDK

\*Note: The key word “const” is added before Attribute Table definition:

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

By adding the “const”, the compiler will store the array data to flash rather than RAM, while all contents of the Attribute Table defined in flash are read only and not modifiable.

## 1. attNum

The “attNum” supports two functions.

- 1) The “attNum” can be used to indicate the number of valid Attributes in current Attribute Table, i.e. the maximum Attribute Handle value. This number is only used in the invalid Attribute item 0 of Attribute Table array.

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

“attNum = 57” indicates there are 57 valid Attributes in current Attribute Table.

In BLE, Attribute Handle value starts from 0x0001 with increment step of 1, while the array index starts from 0. When this virtual Attribute is added to the Attribute Table, each Attribute index equals its Attribute Handle value. After the Attribute Table is defined, Attribute Handle value of an Attribute can be obtained by counting its index in current Attribute Table array.

The final index is the number of valid Attributes (i.e. attNum) in current Attribute Table. In current SDK, the attNum is set as 57; if user adds or deletes any Attribute, the attNum needs to be modified correspondingly.

- 2) The “attNum” can also be used to specify Attributes constituting current Service.

The UUID of the first Attribute for each Service must be “GATT\_UUID\_PRIMARY\_SERVICE(0x2800)”; the first Attribute of a Service sets “attNum” and it indicates following “attNum” Attributes constitute current Service.

As shown in Figure3-41, for the gap service, the Attribute with UUID of “GATT\_UUID\_PRIMARY\_SERVICE” sets the “attNum” as 7, it indicates the seven Attributes from Attribute Handle 1 to Attribute Handle 7 constitute the gap service.

Similarly, for the HID service, the “attNum” of the first Attribute is set as 27, and it indicates the following 27 Attributes constitute the HID service.

Except for Attribute item 0 and the first Attribute of each Service, attNum values of all Attributes must be set as 0.

## 2. perm

The “perm” is the simplified form of “permission” and it serves to specify access permission of current Attribute by Client.

The “perm” of each Attribute should be configured as one or combination of following values.

```
#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
```

Note: Current SDK version does not support PERMISSION READ and PERMISSION WRITE yet.

## 3. uuid and uuidLen

As introduced above, UUID supports two types: BLE standard 2-byte UUID, and Telink proprietary 16-byte UUID. The “uuid” and “uuidLen” can be used to describe the two UUID types simultaneously.

The “uuid” is an u8-type pointer, and “uuidLen” specifies current UUID length, i.e. the uuidLen bytes starting from the pointer are current UUID. Since Attribute Table and all UUIDs are stored in flash, the “uuid” is a pointer pointing to flash.

### 1) BLE standard 2-byte UUID:

E.g. For the Attribute “devNameCharacter” with Attribute Handle of 2, related code is shown as below:

```
#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” indicates “character” in BLE and the “uuid” points to the address of “my\_devNameCharVal” in flash. The “uuidLen” is 2. When Master reads this Attribute, the UUID would be “0x2803”.

## 2) Telink proprietary 16-byte UUID:

E.g. For the Attribute MIC of audio, related code is shown as below:

```
#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},
```

The “uuid” points to the address of “my\_MicData” in flash, and the “uuidLen” is 16. When Master reads this Attribute, the UUID would be “0x000102030405060708090a0b0c0d2b18”.

## 4. pAttrValue and attrLen

Each Attribute corresponds to an Attribute Value. The “pAttrValue” is an u8-type pointer which points to starting address of Attribute Value in RAM/Flash, while the “attrLen” specifies the data length. When Master reads the Attribute Value of certain Attribute from Slave, the “attrLen” bytes of data starting from the area (RAM/Flash) pointed by the “pAttrValue” will be read by this BLE SDK to Master.

Since UUID is read only, the “uuid” is a pointer pointing to flash; while Attribute Value may involve write operation into RAM, so the “pAttrValue” may points to RAM or flash.

E.g. For the Attribute hid Information with Attribute Handle of 35, related code is as shown below:

```
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},
```

In actual application, the key word “const” can be used to store the read-only 4-byte hid information “0x01 0x00 0x01 0x11” into flash. The “pAttrValue” points to the starting address of hidInformation in flash, while the “attrLen” is the actual

length of hidInformation. When Master reads this Attribute, “0x01000111” will be returned to Master correspondingly.

Figure3-42 shows a packet example captured by BLE sniffer when Master reads this Attribute. Master uses the “ATT\_Read\_Req” command to set the target AttHandle as 0x23 (35), corresponding to the hid information in Attribute Table of SDK.

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

Figure3-42 BLE sniffer packet sample when Master reads hidInformation

E.g. For the Attribute “battery value” with Attribute Handle of 40, related code is as shown below:

```
u8 my_batVal[1] = {99};
{0,1,2,1,(u8*)(&my_batCharUUID), (u8*)(my_batVal), 0},
```

In actual application, the “my\_batVal” indicates current battery level and it will be updated according to ADC sampling result; then Slave will actively notify or Master will actively read to transfer the “my\_batVal” to Master. The starting address of the “my\_batVal” stored in RAM will be pointed by the “pAttrValue”.

## 5. Callback function w

The callback function w is write function with prototype as below:

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

User must follow the format above to define callback write function. The callback function w is optional, i.e. for an Attribute, user can select whether to set the callback write function as needed (null pointer 0 indicates not setting callback write function).

The trigger condition for callback function w is: When Slave receives any Attribute PDU with Attribute Opcode as shown below, Slave will check whether the callback function w is set.

- 1) opcode = 0x12, Write Request.
- 2) opcode = 0x52, Write Command.
- 3) opcode = 0x18, Execute Write Request

After Slave receives a write command above, if the callback function w is not set, Slave will automatically write the area pointed by the “pAttrValue” with the value sent from Master, and the data length equals the “l2capLen” in Master packet format minus 3; if the callback function w is set, Slave will execute user-defined callback function w after it receives the write command, rather than writing data into the area pointed by the “pAttrValue”. Note: Only one of the two write operations is allowed to take effect.

By setting the callback function w, user can process Write Request, Write Command, and Execute Write Request in ATT layer of Master. If the callback function w is not set, user needs to evaluate whether the area pointed by the “pAttrValue” can process the command (e.g. If the “pAttrValue” points to flash, write operation is not allowed; or if the “attrLen” is not long enough for Master write operation, some data will be modified unexpectedly.)

#### 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  |

Figure3-43 Write Request in BLE stack

#### 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 |

Figure3-44 Write Command in BLE stack

### 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 |

Figure3-45 Execute Write Request in BLE stack

The void-type pointer “p” of the callback function w points to the value of Master write command. Actually “p” points to a memory area, the value of which is shown as the following structure.

```
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” points to “dma\_len”, valid length of data is l2cap minus 3, and the first valid data is 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;
}
```

The structure “rf\_packet\_att\_data\_t” above is available in the “stack/ble/ble\_common.h”.

## 6. Callback function r

The callback function r is read function with prototype as below:

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

User must follow the format above to define callback read function. The callback function r is also optional, i.e. for an Attribute, user can select whether to set the callback read function as needed (null pointer 0 indicates not setting callback read function).

The trigger condition for callback function r is: When Slave receives any Attribute PDU with Attribute Opcode as shown below, Slave will check whether the callback function r is set.

- 1) opcode = 0x0A, Read Request.
- 2) opcode = 0x0C, Read Blob Request.

After Slave receives a read command above,

- 1) If the callback read function is set, Slave will execute this function, and determine whether to respond with “Read Response/Read Blob Response” according to the return value of this function.
  - A. If the return value is 1, Slave won’t respond with “Read Response/Read Blob Response” to Master.
  - B. If the return value is not 1, Slave will automatically read “attrLen” bytes of data from the area pointed by the “pAttrValue”, and the data will be responded to Master via “Read Response/Read Blob Response”.
- 2) If the callback read function is not set, Slave will automatically read “attrLen” bytes of data from the area pointed by the “pAttrValue”, and the data will be responded to Master via “Read Response/Read Blob Response”.

Therefore, after a Read Request/Read Blob Request is received from Master, if it’s needed to modify the content of Read Response/Read Blob Response, user can register corresponding callback function r, modify contents in RAM pointed by the “pAttrValue” in this callback function, and the return value must be 0.

## 7. Attribute Table layout

Figure3-46 shows Service/Attribute layout based on Attribute Table. The “attnum” of the first Attribute indicates the number of valid Attributes in current ATT Table; the remaining Attributes are assigned to different Services, the first Attribute of each Service is the “declaration”, and the following “attnum” Attributes constitute current Service. Actually the first item of each Service is a Primary Service.

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

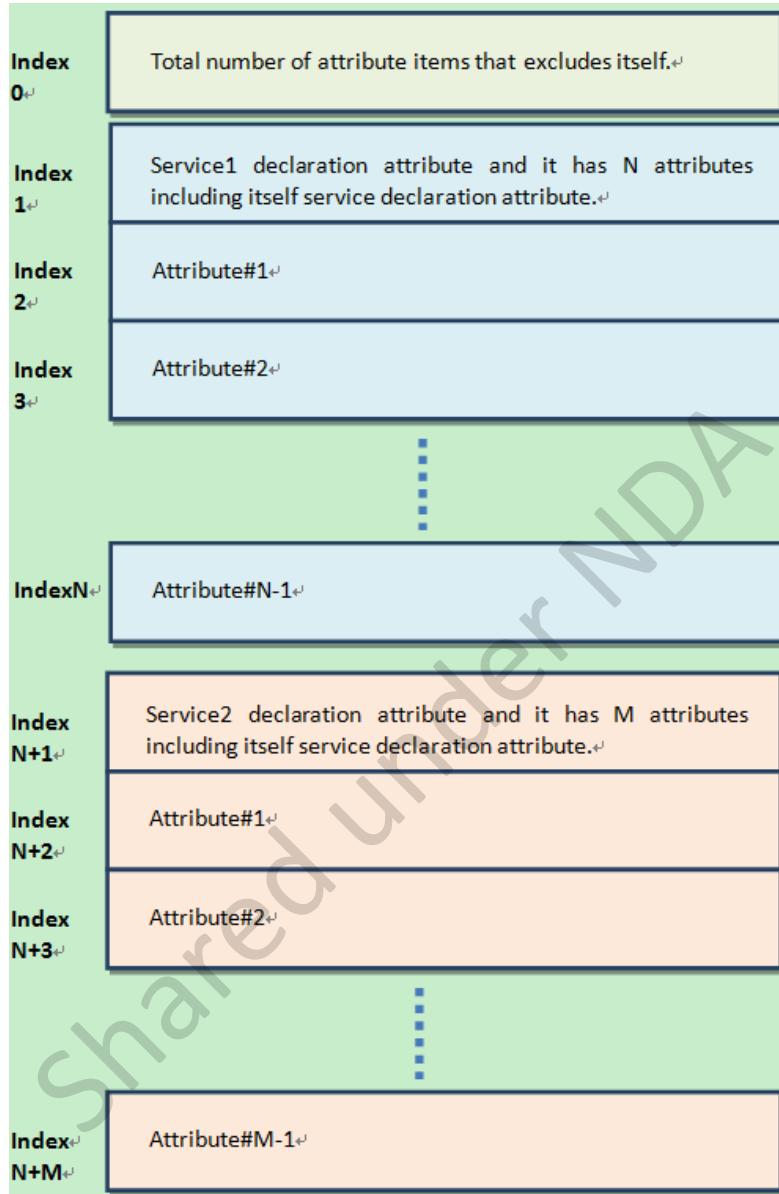


Figure3-46 Service/Attribute Layout

## 8. ATT table Initialization

GATT & ATT initialization only needs to transfer the pointer of Attribute Table in APP layer to protocol stack, and the API below is supplied:

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

“p” is the pointer of Attribute Table.

### 3.3.3.3 Attribute PDU & GATT API

As required by BLE spec, the following Attribute PDU types are supported in current SDK.

- ✧ Requests: Data request sent from Client to Server.
- ✧ Responses: Data response sent by Server after it receives request from Client.
- ✧ Commands: Command sent from Client to Server.
- ✧ Notifications: Data sent from Server to Client.
- ✧ Indications: Data sent from Server to Client.
- ✧ Confirmations: Confirmation sent from Client after it receives data from Server.

The subsections below will introduce all ATT PDUs in ATT layer. Please refer to structure of Attribute and Attribute Table to help understanding.

#### 1. Read by Group Type Request, Read by Group Type Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.4.9 and 3.4.4.10) for details about the “Read by Group Type Request” and “Read by Group Type Response” commands.

The “Read by Group Type Request” command sent by Master specifies starting and ending attHandle, as well as attGroupType. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute Group that matches the specified attGroupType. Then Slave will respond to Master with Attribute Group information via the “Read by Group Type Response” command.

|           |             |      |    |    |              |            |                            |                |              |                                                                 |                      |            |     |  |
|-----------|-------------|------|----|----|--------------|------------|----------------------------|----------------|--------------|-----------------------------------------------------------------|----------------------|------------|-----|--|
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Read_By_Group_Type_Req |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | StartingHandle | EndingHandle | AttGroupType                                                    | 0x89E67B             | -38        | OK  |  |
|           | 2           | 0    | 1  | 0  | 0x0007       |            | 0x0004                     | 0x0001         | 0xFFFF       | 00 28                                                           |                      |            |     |  |
| Data Type | Data Header |      |    |    | CRC          | RSSI (dBm) | FCS                        |                |              |                                                                 |                      |            |     |  |
| Empty PDU | LLID        | NESN | SN | MD | L2CAP-Length |            | 0xAED005                   | -38            | OK           |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Read_By_Group_Type_Rsp |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | Length         | AttData      | 0x11 0x06 01 00 07 00 00 18 08 00 0A 00 0A 18 0B 00 25 00 12 18 | 0x58FC67             | -38        | OK  |  |
|           | 2           | 0    | 0  | 0  | 0x0014       |            | 0x0004                     |                |              |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Read_By_Group_Type_Req |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | StartingHandle | EndingHandle | AttGroupType                                                    | 0x5A6275             | -38        | OK  |  |
|           | 2           | 1    | 0  | 0  | 0x0007       |            | 0x0004                     | 0x10           | 0x0026       | 0xFFFF                                                          | 00 28                |            |     |  |
| Data Type | Data Header |      |    |    | CRC          | RSSI (dBm) | FCS                        |                |              |                                                                 |                      |            |     |  |
| Empty PDU | LLID        | NESN | SN | MD | L2CAP-Length |            | 0xAE0BA0                   | -38            | OK           |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | CRC          | RSSI (dBm) | FCS                        |                |              |                                                                 |                      |            |     |  |
| Empty PDU | LLID        | NESN | SN | MD | L2CAP-Length |            | 0xAE0D73                   | -38            | OK           |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Read_By_Group_Type_Rsp |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | Length         | AttData      | 0x11 0x06 26 00 28 00 0F 18                                     | 0x158866             | -38        | OK  |  |
|           | 2           | 0    | 0  | 0  | 0x0008       |            | 0x0004                     |                |              |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Read_By_Group_Type_Req |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | StartingHandle | EndingHandle | AttGroupType                                                    | 0x055C4D             | -38        | OK  |  |
|           | 2           | 1    | 0  | 0  | 0x0007       |            | 0x0004                     | 0x10           | 0x0029       | 0xFFFF                                                          | 00 28                |            |     |  |
| Data Type | Data Header |      |    |    | CRC          | RSSI (dBm) | FCS                        |                |              |                                                                 |                      |            |     |  |
| Empty PDU | LLID        | NESN | SN | MD | L2CAP-Length |            | 0xAE0BA0                   | -38            | OK           |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | CRC          | RSSI (dBm) | FCS                        |                |              |                                                                 |                      |            |     |  |
| Empty PDU | LLID        | NESN | SN | MD | L2CAP-Length |            | 0xAE0D73                   | -38            | OK           |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Read_By_Group_Type_Rsp |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | Length         | AttData      | 0x11 0x06 26 00 28 00 0F 18                                     | 0x89D99              | -38        | OK  |  |
|           | 2           | 0    | 0  | 0  | 0x0016       |            | 0x0004                     |                |              |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Read_By_Group_Type_Req |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | StartingHandle | EndingHandle | AttGroupType                                                    | 0x3C57D1             | -38        | OK  |  |
|           | 2           | 1    | 0  | 0  | 0x0007       |            | 0x0004                     | 0x10           | 0x0033       | 0xFFFF                                                          | 00 28                |            |     |  |
| Data Type | Data Header |      |    |    | CRC          | RSSI (dBm) | FCS                        |                |              |                                                                 |                      |            |     |  |
| Empty PDU | LLID        | NESN | SN | MD | L2CAP-Length |            | 0xAE0BA0                   | -38            | OK           |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | CRC          | RSSI (dBm) | FCS                        |                |              |                                                                 |                      |            |     |  |
| Empty PDU | LLID        | NESN | SN | MD | L2CAP-Length |            | 0xAE0D73                   | -38            | OK           |                                                                 |                      |            |     |  |
| Data Type | Data Header |      |    |    | L2CAP Header |            | ATT_Error_Response         |                |              |                                                                 | CRC                  | RSSI (dBm) | FCS |  |
| L2CAP-S   | LLID        | NESN | SN | MD | L2CAP-Length |            | Opcode                     | ReqOpCode      | AttHandle    | ErrorCode                                                       | 0x600F0A1            | -38        | OK  |  |
|           | 2           | 0    | 0  | 0  | 0x0005       |            | 0x0004                     | 0x01           | 0x10         | 0xNNNN                                                          | ATT NOT FOUND(0x0A1) |            |     |  |

Figure3-47 Read by Group Type Request/Read by Group Type Response

As shown above, Master requests from Slave for Attribute Group information of the “primaryServiceUUID” with UUID of 0x2800.

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

The following groups in Slave Attribute Table meet the requirement according to current demo code.

- 1) Attribute Group with attHandle from 0x0001 to 0x0007, Attribute Value is SERVICE\_UUID\_GENERIC\_ACCESS (0x1800).
- 2) Attribute Group with attHandle from 0x0008 to 0x000a, Attribute Value is SERVICE\_UUID\_DEVICE\_INFORMATION (0x180A).
- 3) Attribute Group with attHandle from 0x000B to 0x0025, Attribute Value is SERVICE\_UUID\_HUMAN\_INTERFACE\_DEVICE (0x1812).
- 4) Attribute Group with attHandle from 0x0026 to 0x0028, Attribute Value is SERVICE\_UUID\_BATTERY (0x180F).
- 5) Attribute Group with attHandle from 0x0029 to 0x0032, Attribute Value is TELINK\_AUDIO\_UUID\_SERVICE(0x11,0x19,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00).

Slave responds to Master with the attHandle and attValue information of the five Groups above via the “Read by Group Type Response” command. The final ATT\_Error\_Response indicates end of response. When Master receives this packet, it will stop sending “Read by Group Type Request”.

## 2. Find by Type Value Request, Find by Type Value Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.3.3 and 3.4.3.4) for details about the “Find by Type Value Request” and “Find by Type Value Response” commands.

The “Find by Type Value Request” command sent by Master specifies starting and ending attHandle, as well as AttributeType and Attribute Value. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute that matches the specified AttributeType and Attribute Value. Then Slave will respond to Master with the Attribute via the “Find by Type Value Response” command.

|           |             |   |   |   |              |                            |                            |      |             |          |        |            |          |     |    |
|-----------|-------------|---|---|---|--------------|----------------------------|----------------------------|------|-------------|----------|--------|------------|----------|-----|----|
| Data Type | Data Header |   |   |   | L2CAP Header |                            | ATT_Find_By_Type_Value_Req |      |             |          | CRC    | RSSI (dBm) | FCS      |     |    |
| L2CAP-S   | 2           | 1 | 1 | 0 | 13           | 0x0009                     | 0x0004                     | 0x06 | 0x0001      | 0xFFFF   | 0x2800 | 0A 18      | 0x4CEA12 | -54 | OK |
| Data Type | Data Header |   |   |   | CRC          | RSSI (dBm)                 | FCS                        |      |             |          |        |            |          |     |    |
| Empty PDU | 1           | 0 | 0 | 0 | 0            | 0xC4C0E8                   | -54                        |      |             |          |        |            |          |     |    |
| Data Type | Data Header |   |   |   | L2CAP Header | ATT_Find_By_Type_Value_Rsp |                            | CRC  | RSSI (dBm)  | FCS      |        |            |          |     |    |
| L2CAP-S   | 2           | 1 | 0 | 0 | 9            | 0x0005                     | 0x0004                     | 0x07 | 0C 00 0E 00 | 0xF92ED9 | -54    | OK         |          |     |    |

Figure3-48 Find by Type Value Request/Find by Type Value Response

## 3. Read by Type Request, Read by Type Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.4.1 and 3.4.4.2) for details about the “Read by Type Request” and “Read by Type Response” commands.

The “Read by Type Request” command sent by Master specifies starting and ending attHandle, as well as AttributeType. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute that matches the specified AttributeType. Then Slave will respond to Master with the Attribute via the “Read by Type Response”.

| 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 |         |
| Empty PDU | 2           | 0    | 0  | 1  | 11           | 0x0007       | 0x0004               | 0x08   | 0x0001         | 0xFFFF                  | 00 2A   | 0x      |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | CRC          | RSSI (dBm)           | FCS    |                |                         |         |         |
| Empty PDU | 1           | 1    | 0  | 0  | 0            | 0x898717     | 0                    | OK     |                |                         |         |         |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | CRC          | RSSI (dBm)           | FCS    |                |                         |         |         |
| Empty PDU | 1           | 1    | 1  | 0  | 0            | 0x898AB1     | 0                    | OK     |                |                         |         |         |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | CRC          | RSSI (dBm)           | FCS    |                |                         |         |         |
| Empty PDU | 1           | 0    | 1  | 0  | 0            | 0x898C62     | 0                    | OK     |                |                         |         |         |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | CRC          | RSSI (dBm)           | FCS    |                |                         |         |         |
| Empty PDU | 1           | 0    | 0  | 0  | 0            | 0x8981C4     | 0                    | OK     |                |                         |         |         |
| 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                 |         | 0xDB602 |
| Empty PDU | 2           | 1    | 0  | 0  | 14           | 0x000A       | 0x0004               | 0x09   | 0x08           | 03 00 74 53 65 6C 66 69 |         |         |

Figure3-49 Read by Type Request/Read by Type Response

As shown above, Master reads the Attribute with attType of 0x2A00, i.e. the Attribute with Attribute Handle of 00 03 in Slave.

```
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},
```

In the “Read by Type response”, attData length is 8, the first two bytes are current attHandle “0003”, followed by 6-byte Attribute Value.

#### 4. Find information Request, Find information Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.3.1 and 3.4.3.2) for details about the “Find information request” and “Find information response” commands.

The “Find information request” command sent by Master specifies starting and ending attHandle. After the request is received, Slave will respond to Master with Attribute UUIDs according to the specified starting and ending attHandle via the “Find information response”.

As shown below, Master requests for information of three Attributes with attHandle of 0x0016~0x0018, and Slave responds with corresponding UUIDs.

|                        |                                |                                                   |                                                                                      |              |                |   |
|------------------------|--------------------------------|---------------------------------------------------|--------------------------------------------------------------------------------------|--------------|----------------|---|
| Data Type<br>L2CAP-S   | Data Header<br>LLID 0 1 0 9    | L2CAP Header<br>L2CAP-Length 0x0005 ChanId 0x0004 | ATT_Find_Req<br>Opcode 0x04 StartingHandle 0x0016 EndingHandle 0x0018                | CRC 0x362A2F | RSSI (dBm) -38 | F |
| Data Type<br>Empty PDU | Data Header<br>LLID 0 0 0 0    | CRC 0xAE00D5                                      | RSSI (dBm) -38                                                                       | FCS OK       |                |   |
| Data Type<br>Empty PDU | Data Header<br>LLID 1 0 0 0    | CRC 0xAE0606                                      | RSSI (dBm) -38                                                                       | FCS OK       |                |   |
| Data Type<br>L2CAP-S   | Data Header<br>LLID 2 1 1 0 18 | L2CAP Header<br>L2CAP-Length 0x000E ChanId 0x0004 | ATT_Find_Rsp<br>Opcode Format 0x05 InfoData 0x01 16 00 02 29 17 00 08 29 18 00 03 28 | CRC 0x000000 | RSSI (dBm) -38 | F |

Figure3-50 Find information request/Find information response

## 5. Read Request, Read Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.4.3 and 3.4.4.4) for details about the “Read Request” and “Read Response” commands.

The “Read Request” command sent by Master specifies certain attHandle. After the request is received, Slave will respond to Master with the Attribute Value of the specified Attribute via the “Read Response” command (If the callback function r is set, this function will be executed), as shown below.

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

Figure3-51 Read Request/Read Response

## 6. Read Blob Request, Read Blob Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.4.5 and 3.4.4.6) for details about the “Read Blob Request” and “Read Blob Response” commands.

If some Slave Attribute corresponds to Attribute Value with length exceeding MTU\_SIZE (It's set as 23 in current SDK), Master needs to read the Attribute Value via the “Read Blob Request” command, so that the Attribute Value can be sent in packets. This command specifies the attHandle and ValueOffset. After the request is received, Slave will find corresponding Attribute, and respond to Master with the Attribute Value via the “Read Blob Response” command according to the specified ValueOffset. (If the callback function r is set, this function will be executed.)

As shown below, when Master needs the HID report map of Slave (report map length largely exceeds 23), first Master sends “Read Request”, then Slave responds to Master with part of the report map data via “Read response”; Master sends “Read Blob Request”, and then Slave responds to Master with data via “Read Blob Response”.

|           |                                          |  |  |  |                                      |            |                                             |  |                                                                   |                                                                   |          |
|-----------|------------------------------------------|--|--|--|--------------------------------------|------------|---------------------------------------------|--|-------------------------------------------------------------------|-------------------------------------------------------------------|----------|
| Data Type | Data Header                              |  |  |  | L2CAP Header                         |            | ATT_Read_Req                                |  | CRC                                                               | RSSI (dBm)                                                        | FCS      |
| L2CAP-S   | LLID NESN SN MD PDU-Length<br>2 0 1 0 7  |  |  |  | L2CAP-Length ChanId<br>0x0003 0x0004 |            | Opcode AttHandle<br>0x0A                    |  | 0x0020                                                            | 0xF4DC27                                                          | -38 OK   |
| Data Type | Data Header                              |  |  |  | CRC                                  | RSSI (dBm) | FCS                                         |  |                                                                   |                                                                   |          |
| Empty PDU | LLID NESN SN MD PDU-Length<br>1 0 0 0 0  |  |  |  | 0xAE00D5                             | -38        | OK                                          |  |                                                                   |                                                                   |          |
| Data Type | Data Header                              |  |  |  | CRC                                  | RSSI (dBm) | FCS                                         |  |                                                                   |                                                                   |          |
| Empty PDU | LLID NESN SN MD PDU-Length<br>1 1 0 0 0  |  |  |  | 0xAE0606                             | -38        | OK                                          |  |                                                                   |                                                                   |          |
| Data Type | Data Header                              |  |  |  | L2CAP Header                         |            | ATT_Read_Rsp                                |  |                                                                   |                                                                   | CRC      |
| L2CAP-S   | LLID NESN SN MD PDU-Length<br>2 1 1 0 27 |  |  |  | L2CAP-Length ChanId<br>0x0017 0x0004 |            | Opcode AttValue                             |  | 0x0B                                                              | 05 01 09 02 A1 01 85 01 09 01 A1 00 05 09 19 01 29 03 15 00 25 01 | 0xEE69DD |
| Data Type | Data Header                              |  |  |  | L2CAP Header                         |            | ATT_Read_Blob_Req                           |  | CRC                                                               | RSSI (dBm)                                                        | FCS      |
| L2CAP-S   | LLID NESN SN MD PDU-Length<br>2 0 1 0 9  |  |  |  | L2CAP-Length ChanId<br>0x0005 0x0004 |            | Opcode AttHandle ValueOffset<br>0x0C 0x0020 |  | 0x0016                                                            | 0x8F3E95                                                          | -38 OK   |
| Data Type | Data Header                              |  |  |  | CRC                                  | RSSI (dBm) | FCS                                         |  |                                                                   |                                                                   |          |
| Empty PDU | LLID NESN SN MD PDU-Length<br>1 0 0 0 0  |  |  |  | 0xAE00D5                             | -38        | OK                                          |  |                                                                   |                                                                   |          |
| Data Type | Data Header                              |  |  |  | CRC                                  | RSSI (dBm) | FCS                                         |  |                                                                   |                                                                   |          |
| Empty PDU | LLID NESN SN MD PDU-Length<br>1 1 0 0 0  |  |  |  | 0xAE0606                             | -38        | OK                                          |  |                                                                   |                                                                   |          |
| Data Type | Data Header                              |  |  |  | L2CAP Header                         |            | ATT_Read_Blob_Rsp                           |  |                                                                   |                                                                   | CRC      |
| L2CAP-S   | LLID NESN SN MD PDU-Length<br>2 1 1 0 27 |  |  |  | L2CAP-Length ChanId<br>0x0017 0x0004 |            | Opcode PartAttValue<br>0x0D                 |  | 75 01 95 03 81 02 75 05 95 01 81 01 05 01 09 30 09 31 09 38 15 81 | 0x2DE6F2                                                          | -38 OK   |
| Data Type | Data Header                              |  |  |  | L2CAP Header                         |            | ATT_Read_Blob_Req                           |  | CRC                                                               | RSSI (dBm)                                                        | FCS      |
| L2CAP-S   | LLID NESN SN MD PDU-Length<br>2 0 1 0 9  |  |  |  | L2CAP-Length ChanId<br>0x0005 0x0004 |            | Opcode AttHandle ValueOffset<br>0x0C 0x0020 |  | 0x002C                                                            | 0x557D8E                                                          | -38 OK   |

Figure3-52 Read Blob Request/Read Blob Response

## 7. Exchange MTU Request, Exchange MTU Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.2.1 and 3.4.2.2) for details about the “Exchange MTU Request” and “Exchange MTU Response” commands.

As shown below, Master and Slave obtain MTU size of each other via the “Exchange MTU Request” and “Exchange MTU Response” commands.

|           |                                         |  |  |  |                                      |  |                            |  |        |            |        |
|-----------|-----------------------------------------|--|--|--|--------------------------------------|--|----------------------------|--|--------|------------|--------|
| Data Type | Data Header                             |  |  |  | L2CAP Header                         |  | ATT_Exchange_MTU_Req       |  | CRC    | RSSI (dBm) | FCS    |
| L2CAP-S   | LLID NESN SN MD PDU-Length<br>2 0 1 0 7 |  |  |  | L2CAP-Length ChanId<br>0x0003 0x0004 |  | Opcode ClientRxMTU<br>0x02 |  | 0x009E | 0xC70102   | -38 OK |
| Data Type | Data Header                             |  |  |  | L2CAP Header                         |  | ATT_Exchange_MTU_Rsp       |  | CRC    | RSSI (dBm) | FCS    |
| L2CAP-S   | LLID NESN SN MD PDU-Length<br>2 0 0 0 7 |  |  |  | L2CAP-Length ChanId<br>0x0003 0x0004 |  | Opcode ServerRxMTU<br>0x03 |  | 0x0017 | 0x1D88E1   | -38 OK |

Figure3-53 Exchange MTU Request/Exchange MTU Response

During data access process of Telink BLE Slave GATT layer, if there's data exceeding a RF packet length, which involves packet assembly and disassembly in GATT layer, Slave and Master need to exchange RX MTU size of each other in advance. Transfer of long packet data in GATT layer can be implemented via MTU size exchange.

- 1) User can register callback of GAP event (see **section 3.3.5.2 GAP event**) and enable the eventMask “GAP\_EVT\_MASK\_ATT\_EXCHANGE\_MTU” to obtain EffectiveRxMTU.  

$$\text{EffectiveRxMTU} = \min(\text{ClientRxMTU}, \text{ServerRxMTU})$$
  
- 2) Processing of long Rx packet data in 8258 Slave GATT layer  
 8258 Slave ServerRxMTU is set as 23 by default. Actually maximum ServerRxMTU can reach 250, i.e. 250-byte packet data on Master can be correctly re-assembled on Slave. When it's needed to use packet re-assembly of Master in an application, the API below should be invoked to modify RX size of Slave first.

```
ble_sts_t b1c_att_setRxMtuSize(u16 mtu_size);
```

The return value is shown as below:

| ble_sts_t                  | Value                         | ERR Reason                          |
|----------------------------|-------------------------------|-------------------------------------|
| BLE_SUCCESS                | 0                             |                                     |
| GATT_ERR_INVALID_PARAMETER | See the definition in the SDK | mtu_size exceeds the max value 250. |

When Master GATT layer needs to send long packet data to Slave, Master will actively initiate “ATT\_Exchange\_MTU\_req”, and Slave will respond with “ATT\_Exchange\_MTU\_rsp”. “ServerRxMTU” is the configured value of the API “b1c\_att\_setRxMtuSize”. If user has registered GAP event and enabled the eventMask “GAP\_EVT\_MASK\_ATT\_EXCHANGE\_MTU”, “EffectiveRxMTU” and “ClientRxMTU” of Master can be obtained in the callback function of GAP event.

- 3) Processing of long Tx packet data in 8258 Slave GATT layer

When 8258 Slave needs to send long packet data in GATT layer, it should obtain Client RxMTU of Master first, and the eventual data length should not exceed ClientRxMTU.

First Slave should invoke the API “b1c\_att\_setRxMtuSize” to set its ServerRxMTU. Suppose it's set as 158.

```
b1c_att_setRxMtuSize(158);
```

Then the API below should be invoked to actively initiate an “ATT\_Exchange\_MTU\_req”.

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

“connHandle” is ID of Slave connection, i.e. “BLS\_CONN\_HANDLE”, while “mtu\_size” is ServerRxMTU.

```
blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, 158);
```

After the “ATT\_Exchange\_MTU\_req” is received, Master will respond with “ATT\_Exchange\_MTU\_rsp”. After receiving the response, the SDK will calculate EffectiveRxMTU. If user has registered GAP event and enabled the eventMask “GAP\_EVT\_MASK\_ATT\_EXCHANGE\_MTU”, “EffectiveRxMTU” and “ClientRxMTU” will be reported to user.

## 8. Write Request, Write Response

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.5.1 and 3.4.5.2) for details about the “Write Request” and “Write Response” commands.

The “Write Request” command sent by Master specifies certain attHandle and attaches related data. After the request is received, Slave will find the specified Attribute, determine whether to process the data by using the callback function w or directly write the data into corresponding Attribute Value depending on whether the callback function w is set by user. Finally Slave will respond to Master via “Write Response”.

As shown in below, by sending “Write Request”, Master writes Attribute Value of 0x0001 to the Slave Attribute with the attHandle of 0x0016. Then Slave will execute the write operation and respond to Master via “Write Response”.

| Data Type | Data Header |      |    |    |            | L2CAP Header |            | ATT_Write_Req |           |          | CRC      | RSSI (dBm) | FCS |
|-----------|-------------|------|----|----|------------|--------------|------------|---------------|-----------|----------|----------|------------|-----|
| L2CAP-S   | LLID        | NESN | SN | MD | PDU-Length | L2CAP-Length | ChanId     | Opcode        | AttHandle | AttValue | 0xDC8476 | -38        | OK  |
| Empty PDU | 2           | 0    | 1  | 0  | 9          | 0x0005       | 0x0004     | 0x12          | 0x0016    | 01 00    |          |            |     |
| Data Type | Data Header |      |    |    |            | CRC          | RSSI (dBm) | FCS           |           |          |          |            |     |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length | 0xAE00D5     | -38        | OK            |           |          |          |            |     |
| Data Type | Data Header |      |    |    |            | CRC          | RSSI (dBm) | FCS           |           |          |          |            |     |
| Empty PDU | 1           | 0    | 0  | 0  | 0          | 0xAE0606     | -38        | OK            |           |          |          |            |     |
| Data Type | Data Header |      |    |    |            | L2CAP Header |            | ATT_Write_Rsp |           |          | CRC      | RSSI (dBm) | FCS |
| L2CAP-S   | LLID        | NESN | SN | MD | PDU-Length | L2CAP-Length | ChanId     | Opcode        |           |          | 0xFBDB12 | -38        | OK  |
|           | 2           | 1    | 1  | 0  | 5          | 0x0001       | 0x0004     | 0x13          |           |          |          |            |     |

Figure3-54 Write Request/Write Respons

## 9. Write Command

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.5.3) for details about the “Write Command”.

The “Write Command” sent by Master specifies certain attHandle and attaches related data. After the command is received, Slave will find the specified Attribute, determine whether to process the data by using the callback function w or directly write the data into corresponding Attribute Value depending on whether the callback function w is set by user. Slave won’t respond to Master with any information.

## 10. Queued Writes

“Queued Writes” refers to ATT protocol including “Prepare Write Request/Response” and “Execute Write Request/Response”. Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.6/Queued Writes).

“Prepare Write Request” and “Execute Write Request” can implement the two functions below.

- 1) Provide write function for long attribute value.
- 2) Allow to write multiple values in an atomic operation that is executed separately.

Similar to “Read\_Blob\_Req/Rsp”, “Prepare Write Request” contains AttHandle, ValueOffset and PartAttValue. That means Client can prepare multiple attribute values or various parts of a long attribute value in the queue. Thus, before executing the prepared queue indeed, Client can confirm that all parts of some attribute can be written into Server.

Note: Current SDK version only supports the write function of long attribute value with the maximum length not exceeding 244 bytes.

The figure below shows the case when Master writes a long character string “I am not sure what a new song” (byte number is far more than 23, and use the default MTU) into certain characteristic of Slave.

First Master sends a “Prepare Write Request” with offset of 0x0000, to write the data “I am not sure what” into Slave, and Slave responds to Master with a “Prepare Write Response”.

Then Master sends a “Prepare Write Request” with offset of 0x12, to write the data “ a new song” into Slave, and Slave responds to Master with a “Prepare Write Response”.

After the write operation of the long attribute value is finished, Master sends an “Execute Write Request” to Slave. “Flags=1” indicates write result takes effect immediately. Then Slave responds with an “Execute Write Response” to complete the

whole Prepare Write process.

As we can see, “Prepare Write Response” also contains AttHandle, ValueOffset and PartAttValue in the request, so as to ensure reliable data transfer. Client can compare field value of Response with that of Request, to ensure correct reception of the prepared data.

|           |             |      |    |    |              |   |                       |   |   |    |          |        |        |            |             |                                                       |
|-----------|-------------|------|----|----|--------------|---|-----------------------|---|---|----|----------|--------|--------|------------|-------------|-------------------------------------------------------|
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Prepate_Write_Rsp |   |   |    |          |        |        |            |             |                                                       |
| L2CAP-S   | LLID        | NESN | SN | MD | PDU-Length   | 2 | 0                     | 1 | 0 | 27 | 0x0017   | 0x0004 | Opcode | AttHandle  | ValueOffset | PartAttValue                                          |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | 0x17   | 0x0015     | 0x0000      | 49 20 61 6D 20 6E 6F 74 20 73 75 72 65 20 77 68 61 74 |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Prepate_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| L2CAP-S   | LLID        | NESN | SN | MD | PDU-Length   | 2 | 0                     | 0 | 0 | 20 | 0x0010   | 0x0004 | Opcode | AttHandle  | ValueOffset | PartAttValue                                          |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | 0x16   | 0x0015     | 0x0012      | 20 61 20 6E 65 77 20 73 6F 6E 67                      |
|           | Data Header |      |    |    | L2CAP Header |   | ATT_Prepate_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | 1 | 1                     | 0 | 0 | 0  | 0x071388 | 0x0004 | CRC    | RSSI (dBm) | FCS         |                                                       |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | -54    | OK         |             |                                                       |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Prepate_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | 1 | 1                     | 1 | 0 | 0  | 0x071E2E | 0x0004 | CRC    | RSSI (dBm) | FCS         |                                                       |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | -54    | OK         |             |                                                       |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Prepate_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| L2CAP-S   | LLID        | NESN | SN | MD | PDU-Length   | 2 | 0                     | 1 | 0 | 20 | 0x0010   | 0x0004 | Opcode | AttHandle  | ValueOffset | PartAttValue                                          |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | 0x17   | 0x0015     | 0x0012      | 20 61 20 6E 65 77 20 73 6F 6E 67                      |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Execute_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| L2CAP-S   | LLID        | NESN | SN | MD | PDU-Length   | 2 | 0                     | 0 | 0 | 6  | 0x0002   | 0x0004 | CRC    | RSSI (dBm) | FCS         |                                                       |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | 0x16   | 0x01       | 0x24D166    | -54 OK                                                |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Execute_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | 1 | 1                     | 0 | 0 | 0  | 0x071388 | 0x0004 | CRC    | RSSI (dBm) | FCS         |                                                       |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | -54    | OK         |             |                                                       |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Execute_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | 1 | 1                     | 1 | 0 | 0  | 0x071E2E | 0x0004 | CRC    | RSSI (dBm) | FCS         |                                                       |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | -54    | OK         |             |                                                       |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Execute_Write_Req |   |   |    |          |        |        |            |             |                                                       |
| Empty PDU | LLID        | NESN | SN | MD | PDU-Length   | 1 | 0                     | 0 | 0 | 0  | 0x07155B | 0x0004 | CRC    | RSSI (dBm) | FCS         |                                                       |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | -54    | OK         |             |                                                       |
| Data Type | Data Header |      |    |    | L2CAP Header |   | ATT_Execute_Write_Rsp |   |   |    |          |        |        |            |             |                                                       |
| L2CAP-S   | LLID        | NESN | SN | MD | PDU-Length   | 2 | 0                     | 1 | 0 | 5  | 0x0001   | 0x0004 | CRC    | RSSI (dBm) | FCS         |                                                       |
|           |             |      |    |    |              |   |                       |   |   |    |          |        | 0x19   | 0x430D57   | -54 OK      |                                                       |

Figure3-55 Example for Write Long Characteristic Values

## 11. Handle Value Notification

Please refer to “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

Figure3-56 Handle Value Notification in BLE Spec

The figure above shows the format of “Handle Value Notification” in BLE Spec.

This BLE SDK supplies an API for Handle Value Notification of an Attribute. By invoking this API, user can push the notify data into bottom-layer BLE software fifo. Stack will push the data of software fifo into hardware fifo during the latest packet transfer interval, and finally send the data out via RF.

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

“handle” is attHandle of Attribute, “p” is the head pointer of successive memory data to be sent, while “len” specifies byte number of data to be sent. Since this API supports auto packet disassembly based on EffectiveMaxTxOctets, long notify data to be sent can be disassembled into multiple BLE RF packets, and large “len” is supported. (EffectiveMaxTxOctets indicates the maximum RF TX octets to be sent in the Link Layer. Its default value is 27, and DLE may modify it. Another API as a replacement will be introduced later.)

When Link Layer is in Conn state, generally data can be successfully pushed into bottom-layer software fifo by invoking this API. However, some special cases may result in invoking failure, and the return value “ble\_sts\_t” will indicate the corresponding error reason.

When this API is invoked in APP layer, it's recommended to check whether the return value is “BLE\_SUCCESS”. If the return value is not “BLE\_SUCCESS”, a delay is needed to re-push the data. The return value is shown as below:

| ble_sts_t                                           | Value                         | ERR reason                                                                   |
|-----------------------------------------------------|-------------------------------|------------------------------------------------------------------------------|
| BLE_SUCCESS                                         | 0                             |                                                                              |
| LL_ERR_CONNECTION_NOT_ESTABLISHED                   | See the definition in the SDK | Link Layer is in None Conn state                                             |
| LL_ERR_ENCRYPTION_BUSY                              | See the definition in the SDK | Data cannot be sent during pairing or encryption phase.                      |
| LL_ERR_TX_FIFO_NOT_ENOUGH                           | See the definition in the SDK | Since task with mass data is being executed, software Tx fifo is not enough. |
| GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY | See the definition in the SDK | Data cannot be sent during service discovery phase.                          |

**Note:** The new SDK adds one replacement API which uses min(EffectiveMaxTxOctets, EffectiveRxMTU) as minimum unit for packet disassembly and is invokable by both Master and Slave. It's recommended to use this new API.

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

When invoking this API, it's recommended to check whether the return value is “BLE\_SUCCESS”.

Corresponding to the return value of the “bls\_att\_pushNotifyData”, the return value of the “blc\_gatt\_pushHandleValueNotify” has one difference:

- 1) In pairing phase, the return value of the new API is SMP\_ERR\_PAIRING\_BUSY.
- 2) In encryption phase, the return value of the new API is LL\_ERR\_ENCRYPTION\_BUSY.

- 3) When ( $\text{len} > \text{ATT\_MTU}-3$ ), meaning it is going to send more than the maximum ATT\_MT could support, the return value is GATT\_ERR\_DATA\_LENGTH\_EXCEED\_MTU\_SIZE.

## 12. Handle Value Indication

Please refer to “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

Figure3-57 Handle Value Indication in BLE spec

The figure above shows the format of “Handle Value Indication” in BLE Spec.

This BLE SDK supplies an API for Handle Value Indication of an Attribute. By invoking this API, user can push the indicate data into bottom-layer BLE software fifo. Stack will push the data of software fifo into hardware fifo during the latest packet transfer interval, and finally send the data out via RF.

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

“handle” is attHandle corresponding to Attribute, “p” is the head pointer of successive memory data to be sent, while “len” specifies byte number of data to be sent. Since this API supports auto packet disassembly based on EffectiveMaxTxOctets, long indicate data to be sent can be disassembled into multiple BLE RF packets, large “len” is supported. (EffectiveMaxTxOctets indicates the maximum RF TX octets to be sent in the Link Layer. Its default value is 27, and DLE may modify it. Another API as a replacement will be introduced later.)

As specified in BLE Spec, Slave won’t regard data indication as success until Master confirms the data, and the next indicate data won’t be sent until the previous data indication is successful.

When Link Layer is in Conn state, generally data will be successfully pushed into bottom-layer software FIFO by invoking this API; however, some special cases may result in invoking failure, and the return value “ble\_sts\_t” will indicate the corresponding error reason.

When this API is invoked in APP layer, it’s recommended to check whether the return value is “BLE\_SUCCESS”. If the return value is not “BLE\_SUCCESS”, a delay is

needed to re-push the data. The return value is shown as below:

| ble_sts_t                                           | Value                         | ERR reason                                                                 |
|-----------------------------------------------------|-------------------------------|----------------------------------------------------------------------------|
| BLE_SUCCESS                                         | 0                             |                                                                            |
| LL_ERR_CONNECTION_NOT_ESTABLISH                     | See the definition in the SDK | Link Layer is in None Conn state                                           |
| LL_ERR_ENCRYPTION_BUSY                              | See the definition in the SDK | Data cannot be sent during pairing or encryption phase.                    |
| LL_ERR_TX_FIFO_NOT_ENOUGH                           | See the definition in the SDK | Task with mass data is being executed, and software Tx fifo is not enough. |
| GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY | See the definition in the SDK | Data cannot be sent during service discovery phase.                        |
| GATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED   | See the definition in the SDK | The previous indicate data has not been confirmed by Master.               |

**Note:** The new SDK adds one replacement API which uses min(EffectiveMaxTxOctets, EffectiveRxMTU) as minimum unit for packet disassembly and is invokable by both Master and Slave. It's recommended to use this new API.

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

When invoking this API, it's recommended to check whether the return value is "BLE\_SUCCESS".

Corresponding to the return value of the "bls\_att\_pushIndicateData", the return value of the "blc\_gatt\_pushHandleValueIndicate" has one difference:

- 1) In pairing phase, the return value of the new API is SMP\_ERR\_PAIRING\_BUSY.
- 2) In encryption phase, the return value of the new API is LL\_ERR\_ENCRYPTION\_BUSY.
- 3) When (len > ATT\_MTU-3), meaning it is going to send more than the maximum ATT\_MT could support, the return value is GATT\_ERR\_DATA\_LENGTH\_EXCEED\_MTU\_SIZE.

### 13. Handle Value Confirmation

Please refer to “Core\_v5.0” (Vol 3/Part F/3.4.7.3).

Whenever the API “bls\_att\_pushIndicateData” (or “blc\_gatt\_pushHandleValueIndicate”) is invoked by APP layer to send an indicate data to Master, Master will respond with “Confirmation” to confirm the data, then Slave can continue to send the next indicate data.

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

Table 3.36: Format of Handle Value Confirmation

Figure3-58 Handle Value Confirmation in BLE Spec

As shown above, “Confirmation” is not specific to indicate data of certain handle, and the same “Confirmation” will be responded irrespective of handle.

To enable the APP layer to know whether the indicate data has already been confirmed by Master, user can register the callback of GAP event (see **section 3.3.5.2 GAP event**), and enable corresponding eventMask “GAP\_EVT\_GATT\_HANDLE\_VLAUE\_CONFIRM” to obtain Confirm event.

#### 3.3.3.4 GATT Service Security

Before reading “GATT Service Security”, user can refer to **section 3.3.4 SMP** to learn basic knowledge related to SMP including LE pairing method, security level, and etc.

The figure below shows the mapping relationship of service request for GATT Service Security level given by BLE spec. Please refer to “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<br>No STK                                  | Unauthenticated LTK or<br>Unauthenticated STK | Authenticated LTK or<br>Authenticated STK | Authenticated LTK with<br>Secure Connections |
| Unencrypted           | <b>None</b>                                            | Request succeeds                                  | Request succeeds                              | Request succeeds                          | Request succeeds                             |
|                       | <b>Encryption, No MITM Protection</b>                  | Error Resp.: Insufficient Authentication          | Error Resp.: Insufficient Encryption          | Error Resp.: Insufficient Encryption      | Error Resp.: Insufficient Encryption         |
|                       | <b>Encryption, MITM Protection</b>                     | Error Resp.: Insufficient Authentication          | Error Resp.: Insufficient Encryption          | Error Resp.: Insufficient Encryption      | Error Resp.: Insufficient Encryption         |
|                       | <b>Encryption, MITM Protection, Secure Connections</b> | Error Resp.: Insufficient Authentication          | Error Resp.: Insufficient Encryption          | Error Resp.: Insufficient Encryption      | Error Resp.: Insufficient Encryption         |
| Encrypted             | <b>None</b>                                            | N/A<br>(Not possible to be encrypted without LTK) | Request succeeds                              | Request succeeds                          | Request succeeds                             |
|                       | <b>Encryption, No MITM Protection</b>                  |                                                   | Request succeeds                              | Request succeeds                          | Request succeeds                             |
|                       | <b>Encryption, MITM Protection</b>                     |                                                   | Error Resp.: Insufficient Authentication      | Request succeeds                          | Request succeeds                             |
|                       | <b>Encryption, MITM Protection, Secure Connections</b> |                                                   | Error Resp.: Insufficient Authentication      | Error Resp.: Insufficient Authentication  | Request succeeds                             |

Table 10.2: Local device responds to a service request

Figure3-59 Mapping diagram for service request and response

As shown in the figure above:

- ❖ The first column marks whether currently connected Slave device is in encryption state;
- ❖ The second column (local Device's Access Requirement for service) is related to Permission Access setting for attributes in ATT table;
- ❖ The third column includes four sub-columns corresponding to four levels of LE security mode1 for current device pairing state:
  - 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 Authentication
#define ATT_PERMISSIONS_AUTHEN_WRITE (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Write requires Authentication
#define ATT_PERMISSIONS_AUTHEN_RDWR (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN) //!< Read & Write requires Authentication

#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)

```

Figure3-60 ATT Permission definition

The final implementation of GATT Service Security is related to parameter settings during SMP initialization, including the highest security level, permission access of attributes in ATT table.

It is also related to Master, for example, suppose Slave sets the highest security level supported by SMP as “Authenticated pairing with encryption”, but the highest level supported by Master is “Unauthenticated pairing with encryption”; if the permission for some write attribute in ATT table is “ATT\_PERMISSIONS\_AUTHEN\_WRITE”, when Master writes this attribute, an error will be responded to indicate “encryption level is not enough”.

User can set permission of attributes in ATT table to implement the application below:

Suppose the highest security level supported by Slave is “Unauthenticated pairing with encryption”, but it’s not hoped to trigger Master pairing by sending “Security Request” after connection, user can set the permission for CCC (Client Characteristic Configuration) attribute with nofly attribute as “ATT\_PERMISSIONS\_ENCRYPT\_WRITE”. Only when Master writes the CCC, will Slave respond that security level is not enough and trigger Master to start pairing encryption.

Note: Security level set by user only indicates the highest security level supported by device, and GATT Service Secuiry can be used to realize control as long as ATT Permission does not exceed the highest level that takes effect indeed. For LE security mode1 level 4, if use only sets the level “Authenticated LE Secure Connections”, the setting supports LE Secure Connections only.

For example of GATT security level, please refer to “8258\_feature\_test/feature\_gatt\_security.c”.

### 3.3.3.5 8258 master GATT

In 8258 master kma dongle, the following GATT APIs are supplied for simple service discovery or other data access functions.

```
void att_req_find_info(u8 *dat, u16 start_attHandle, u16 end_attHandle);
```

Actual length (byte) of dat: 11.

```
void att_req_find_by_type (u8 *dat, u16 start_attHandle, u16 end_attHandle, u8 *uuid, u8* attr_value, int len);
```

Actual length (byte) of dat: 13 + attr\_value length.

```
void att_req_read_by_type (u8 *dat, u16 start_attHandle, u16 end_attHandle, u8 *uuid, int uuid_len);
```

Actual length (byte) of dat: 11 + uuid length.

```
void att_req_read (u8 *dat, u16 attHandle);
```

Actual length (byte) of dat: 9.

```
void att_req_read_blob (u8 *dat, u16 attHandle, u16 offset);
```

Actual length (byte) of dat: 11.

```
void att_req_read_by_group_type (u8 *dat, u16 start_attHandle, u16 end_attHandle, u8 *uuid, int uuid_len);
```

Actual length (byte) of dat: 11 + uuid length.

```
void att_req_write (u8 *dat, u16 attHandle, u8 *buf, int len);
```

Actual length (byte) of dat: 9 + buf data length.

```
void att_req_write_cmd (u8 *dat, u16 attHandle, u8 *buf, int len);
```

Actual length (byte) of dat: 9 + buf data length.

For the APIs above, it's needed to pre-define memory space \*dat, then invoke corresponding API to assemble data, finally invoke "blm\_push\_fifo" to send "dat" to Controller for transmission. Note that it's needed to check whether the return value is TRUE. The API "att\_req\_find\_info" is taken as an example for user reference.

```
u8 cmd[12];

att_req_find_info(cmd, 0x0001, 0x0003);

if(blm_push_fifo (BLM_CONN_HANDLE, cmd)) {
 //cmd send OK

}
```

As shown above, after a cmd (e.g. “find info req”/“read req”) is received, Slave will respond with corresponding response information (e.g. “find info rsp”/“read rsp”) soon. It’s only needed to process in “int app\_l2cap\_handler (u16 conn\_handle, u8 \*raw\_pkt)” according to the frame below:

```
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) in BLE is mainly used to provide various encryption keys for LE device to ensure data security. Encrypted link can protect the original contents of data in the air from being intercepted, decoded or read by any attacker.

For details about the SMP, please refer to “Core\_v5.0” (Vol 3/Part H/ Security Manager Specification).

#### 3.3.4.1 SMP security level

BLE 4.2 Spec adds a new pairing method “LE Secure Connections” which further strengthens security. The pairing method in earlier version is called “LE legacy pairing”.

As shown in the **section 3.3.3.4 GATT Service Security**, local device supports pairing states below:

| Local Device Pairing Status |                                                     |                                                 |                                                    |
|-----------------------------|-----------------------------------------------------|-------------------------------------------------|----------------------------------------------------|
| No LTK<br>No STK            | Unauthenticated<br>LTK or<br>Unauthenticated<br>STK | Authenticated<br>LTK or<br>Authenticated<br>STK | Authenticated<br>LTK with<br>Secure<br>Connections |

Figure3-61 Local device pairing status

The four states correspond to the four levels of LE security mode1:

- 1) No authentication and no encryption (LE security **mode1 level1**)
- 2) Unauthenticated pairing with encryption (LE security **mode1 level2**)
- 3) Authenticated pairing with encryption (LE security **mode1 level3**)
- 4) Authenticated LE Secure Connections (LE security **mode1 level4**)

For more details, please refer to “Core\_v5.0” (Vol 3//Part C/10.2 LE SECURITY MODES).

Note: Security level set by local device only indicates the highest security level that local device may reach. However, to reach the preset level indeed, the two factors below are important:

- 1) The supported highest security level set by peer Master device  $\geq$  the supported highest security level set by local Slave device.

- 
- 2) Both local device and peer device complete the whole pairing process (if pairing exists) correctly as per the preset SMP parameters.

For example, even if the highest security level supported by Slave is set as “mode1 level3” (Authenticated pairing with encryption), when the highest security level supported by peer Master is set as “mode1 level1” (No authentication and no encryption), after connection Slave and Master won’t execute pairing, and indeed Slave uses security mode1 level 1.

User can use the API below to set the highest security level supported by SM:

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

Following shows the definition for the enum type le\_security\_mode\_level\_t:

```
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 parameter configuration

SMP parameter configuration In Telink BLE SDK is introduced according to the configuration of four SMP security levels.

For Slave, SMP function currently can support the highest security level “LE security mode1 level4”; for Master, SMP function currently can support the highest security level “LE security mode1 level2” in “LE legacy pairing” method, i.e. “legacy pairing Just Works” method.

#### 1) LE security mode1 level1

Level 1 indicates device does not support encryption pairing. If it’s needed to disable SMP function, user only needs to invoke the function below during initialization:

```
blc_smp_setSecurityLevel(No_Security);
```

It means the device won't implement pairing encryption for current connection. Even if the peer requests for pairing encryption, the device will reject it. It generally applies to the device that does not support encryption pairing process.

As shown in the figure below, Master sends a pairing request, and Slave responds with "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 |     |
| Empty PDU |             |             |      |    |              |              |                   |                |       |               |         |               |             |             |          |     |
|           | Data Type   | Data Header |      |    |              | CRC          | RSSI<br>(dBm)     | FCS            |       |               |         |               |             |             |          |     |
| Empty PDU |             | LLID        | NESN | SN | MD           | PDU-Length   | 0x000014          | -54            | OK    |               |         |               |             |             |          |     |
|           | Data Type   | Data Header |      |    |              | CRC          | RSSI<br>(dBm)     | FCS            |       |               |         |               |             |             |          |     |
| Empty PDU |             | LLID        | NESN | SN | MD           | PDU-Length   | 0x000015          | -62            | OK    |               |         |               |             |             |          |     |
|           | Data Type   | Data Header |      |    |              | L2CAP Header | SM_Pairing_Failed |                | CRC   | RSSI<br>(dBm) | FCS     |               |             |             |          |     |
| Empty PDU |             | LLID        | NESN | SN | MD           | PDU-Length   | 0x0002            | 0x0006         | 0x05  | 0x05          | 0x0000E | -54           | OK          |             |          |     |
|           | L2CAP-S     | LLID        | NESN | SN | MD           | PDU-Length   | 0x0002            | 0x0006         | 0x05  | 0x05          | 0x0000E | -54           | OK          |             |          |     |

Figure3-62 Packet example for Pairing Disable

## 2) LE security mode1 level2

Level 2 indicates device supports the highest security level "Unauthenticated\_Paring\_with\_Encryption", e.g. "Just Works" pairing mode in legacy pairing and secure connection pairing method.

- A. As introduced earlier, SMP supports legacy encryption and secure connection pairing. The SDK provides the API below to set whether the new encryption feature in BLE4.2 is supported.

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

Following shows the definition for the enum type paring\_methods\_t:

```
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. When using security level other than LE security mode1 level1, the API below must be invoked to initialize SMP parameter configuration, including flash initialization setting of bonded area.

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

If only this API is invoked during initialization, the SDK will use default parameters to configure SMP:

- ❖ The highest security level supported by default: Unauthenticated\_Paring\_with\_Encryption.
- ❖ Default bonding mode: Bondable\_Mode (store KEY that is distributed

after pairing encryption into flash).

- ✧ Default IO capability: IO\_CAPABILITY\_NO\_INPUT\_NO\_OUTPUT.

The default parameters above follow the configuration of legacy pairing “Just Works” mode. Therefore invoking this API only is equivalent to configure LE security mode1 level2.

LE security mode1 level2 has two types of setting:

- A. Device supports initialization setting of “Just Works” in legacy pairing.

`blc_smp_peripheral_init();`

- B. Device supports initialization setting of “Just Works” in secure connections.

`blc_smp_setParingMethods(LE_Secure_Connection);`

`blc_smp_peripheral_init();//SMP para setting must precede this API`

### 3) LE security mode1 level3

Level 3 indicates device supports the highest security level “Authenticated pairing with encryption”, e.g. “Passkey Entry” / “Out of Band” in legacy pairing mode.

As required by this level, device should support Authentication, i.e. legal identity of two pairing sides should be ensured.

The three Authentication methods below are supported in BLE:

- ✧ Method 1 with involvement of user, e.g. device has button or display capability, so that one side can display TK, while the other side can input the same TK (e.g. Passkey Entry).
- ✧ Method 2: The two pairing sides can exchange information using the method of non-BLE RF transfer to implement pairing (e.g. Out of Band which transfers TK via NFC generally).
- ✧ Method 3: Use the TK negotiated and agreed by two device sides (e.g. Just Works with TK 0 used by two sides). Since this method is Unauthenticated, the security level of “Just Works” corresponds to LE security mode1 level2.

Authentication can ensure the legality of two pairing sides, and this protection method is called MITM (Man-in-the-Middle) protection.

- A. Device with Authentication should set its MITM flag or OOB flag. The SDK provides the two APIs below to set MITM flag and OOB flag.

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

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

“MITM\_en”/“OOB\_en”: 1 - enable; 0 - disable.

- B. As introduced earlier, SM provides three Authentication methods selectable depending on IO capability of two sides. The SDK provides the API below to set IO capability for current device.

```
void blc_smp_setIoCapability (io_capability_t ioCapability);
```

Following shows the definition for the enum type `io_capability_t`:

```
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. The figure below shows the rule to use MITM flag and OOB flag in legacy pairing mode.

|           |              | 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<br>IO Capabilities | Use<br>IO Capabilities |
|           | MITM Not Set |            |             | Use<br>IO Capabilities | Use<br>Just Works      |

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

Figure3-63 Usage rule for MITM/OOB flag in legacy pairing mode

OOB and MITM flag of local device and peer device will be checked to determine whether to use OOB method or select certain KEY generation method as per IO capability.

As shown in the figure below, the SDK will select different KEY generation methods according to IO capability (Row/Column parameter type `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_Dsplt_Init_Input, JustWorks, PK_Resp_Dsplt_Init_Input },
 { JustWorks, JustWorks, PK_Resp_Dsplt_Init_Input, JustWorks, PK_Resp_Dsplt_Init_Input },
 { PK_Init_Dsplt_Resp_Input, PK_Init_Dsplt_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsplt_Resp_Input },
 { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
 { PK_Init_Dsplt_Resp_Input, PK_Init_Dsplt_Resp_Input, PK_Resp_Dsplt_Init_Input, JustWorks, PK_Init_Dsplt_Resp_Input }
};

#ifndef SECURE_CONNECTION_ENABLE
static const stk_generationMethod_t gen_method_sc[5 /*Responder*/][5 /*Initiator*/] = {
 { JustWorks, JustWorks, PK_Resp_Dsplt_Init_Input, JustWorks, PK_Resp_Dsplt_Init_Input },
 { JustWorks, Numric_Comparison, PK_Resp_Dsplt_Init_Input, JustWorks, Numric_Comparison },
 { PK_Init_Dsplt_Resp_Input, PK_Init_Dsplt_Resp_Input, PK_BOTH_INPUT, JustWorks, PK_Init_Dsplt_Resp_Input },
 { JustWorks, JustWorks, JustWorks, JustWorks, JustWorks },
 { PK_Init_Dsplt_Resp_Input, Numric_Comparison, PK_Resp_Dsplt_Init_Input, JustWorks, Numric_Comparison },
};
#endif

```

Figure3-64 Mapping relationship for KEY generation method and IO capability

For details about the mapping relationship, please refer to “core5.0” (Vol3/Part H/2.3.5.1 Selecting Key Generation Method).

LE security mode1 level3 supports the methods below to configure initial values:

A. Initialization setting of OOB for device with legacy pairing:

```

blc_smp_enableOobAuthentication(1);
blc_smp_peripheral_init(); //SMP para setting must precede this API

```

Considering TK value transfer by OOB, the SDK provides related GAP event in the APP layer (see **section 3.3.5.2 GAP event**).

The API below serves to set TK value of OOB.

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

The parameter “oobData” indicates the head pointer for the array of 16-digit TK value to be set.

B. Initialization setting of Passkey Entry (PK\_Resp\_Dsplt\_Init\_Input) for device with legacy pairing:

```

blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_ONLY);
blc_smp_peripheral_init(); //SMP para setting must precede this API

```

C. Initialization setting of Passkey Entry (PK\_Init\_Dsplt\_Resp\_Input or PK\_BOTH\_INPUT) for device with legacy pairing:

```
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_KEYBOARD_ONLY);
blc_smp_peripheral_init(); //SMP para setting must precede this API
```

Considering TK value input by user, the SDK provides related GAP event in the APP layer (see **section 3.3.5.2 GAP event**).

The API below serves to set TK value of Passkey Entry:

```
void blc_smp_setTK_by_PasskeyEntry (u32 pinCodeInput);
```

The parameter “pinCodeInput” indicates the pincode value to be set and its range is 0~999999. It applies to the case of Passkey Entry method in which Master displays TK and Slave needs to input TK.

KEY generation method finally adopted is related to SMP security level supported by two pairing sides. If Master only supports LE security mode1 level1, since Master does not support pairing encryption, Slave won’t enable SMP function.

#### 4) LE security mode1 level4

Level 4 indicates device supports the highest security level “Authenticated LE Secure Connections”, e.g. Numeric Comparison/Passkey Entry/Out of Band in secure connection pairing mode.

LE security mode1 level4 supports the methods below to configure initial values:

- Initialization setting of Numeric Comparison for device with secure connection pairing:

```
blc_smp_setParingMethods(LE_Secure_Connection);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_YESNO);
```

Considering display of numerical comparison result to user, the SDK provides related GAP event in the APP layer (see **section 3.3.5.2 GAP event**).

The API below serves to set numerical comparison result as “YES” (1) or “NO” (0).

```
void blc_smp_setNumericComparisonResult (bool YES_or_NO);
```

The parameter “YES\_or\_NO” serves to confirm whether six-digit values on two sides are consistent. If yes, input 1 to indicate “YES”; otherwise input 0 to indicate “NO”.

- B. Initialization setting of Passkey Entry for device with secure connection pairing:

User initialization code of this part is almost the same with that of the configuration mode B/C (Passkey Entry in legacy pairing) in LE security mode1 level3, except that pairing method herein should be set as “secure connection pairing” at the start of initialization.

```
blc_smp_setParingMethods(LE_Secure_Connection);
.....//Refer to configuration method B/C in LE security mode1 level3
```

- C. Initialization setting of Out of Band for device with secure connection pairing:

This part is not implemented in current SDK yet.

## 5) Several APIs related to SMP parameter configuration:

- A. The API below serves to set whether to enable bonding function:

```
void blc_smp_setBondingMode(bonding_mode_t mode);
```

Following shows the enum type bonding\_mode\_t:

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

For device with security level other than mode1 level1, bonding function must be enabled. Since the SDK has enabled bonding function by default, generally user does not need to invoke this API.

- B. The API below serves to set whether to enable Key Press function:

```
void blc_smp_enableKeypress (int keyPress_en);
```

It indicates whether it's supported to provide some necessary input status information for KeyboardOnly device during Passkey Entry. Since the current SDK does not support this function yet, the parameter must be set as 0.

- C. The API below serves to set whether to enable key pairs for ECDH (Elliptic Curve Diffie-Hellman) debug mode:

```
void b1c_smp_setEcdhDebugMode(ecdh_keys_mode_t mode);
```

Following shows the definition for the enum type ecdh\_keys\_mode\_t:

```
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;
```

This API only applies to the case with secure connection pairing. The ellipse encryption algorithm can prevent eavesdropping effectively, but at the same time, it's not very friendly to debugging and development, since user cannot capture BLE packet in the air by sniffer and analyze the data. Thus, as defined in BLE spec, ellipse encryption mode with private and public key pairs is provided for debugging. As long as this mode is enabled, BLE sniffer tool can use the known key to decrypt the link.

- D. Following is a unified API to set whether to enable bonding, whether to enable MITM flag, whether to support OOB, whether to support Keypress notification, as well as to set supported IO capability.

```
void b1c_smp_setSecurityParamters (bonding_mode_t mode, int
 MITM_en,
 int OOB_en, int keyPress_en, io_capability_t ioCapabllity);
```

Definition for each parameter herein is consistent with the same parameter in the corresponding independent API.

### 3.3.4.3 SMP Security Request

Only Slave can send SMP Security Request, so this part only applies to Slave device.

During phase 1 of pairing process, there's an optional Security Request packet which serves to enable Slave to actively trigger pairing process to start.

The SDK provides the API below to flexibly set whether Slave sends Security Request to Master immediately after connection/re-connection, or delay for pending\_ms milliseconds before sending Security Request, or does not send Security Request, so as to implement different pairing trigger combination.

```
b1c_smp_configSecurityRequestSending(secReq_cfg newConn_cfg,
```

```
secReq_cfg reConn_cfg, u16 pending_ms);
```

Following shows the definition for the enum type secReq\_cfg:

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

- ✧ SecReq\_NOT\_SEND: After connection is established, Slave won't send Security Request actively.
- ✧ SecReq\_IMM\_SEND: After connection is established, Slave will send Security Request immediately.
- ✧ SecReq\_PEND\_SEND: After connection is established, Slave will wait for pending\_ms milliseconds and then determine whether to send Security Request.
  - 1) For the first connection, Slave receives Pairing\_request from Master before pending\_ms milliseconds, and it won't send Security Request;
  - 2) For re-connection, if Master has already sent LL\_ENC\_REQ before pending\_ms milliseconds to encrypt reconnection link, Slave won't send Security Request.

The parameter “newConn\_cfg” serves to configure new device, while the parameter “reConn\_cfg” serves to configure device to be reconnected.

During reconnection, the SDK also supports the configuration whether to send purpose of pairing request: During reconnection for a bonded device, Master may not actively initiate LL\_ENC\_REQ to encrypt link, and Security Request sent by Slave will trigger Master to actively encrypt the link. Therefore, the SDK provides reConn\_cfg configuration, and user can configure it as needed.

Note: This API must be invoked before connection. It's recommended to invoke it during initialization.

The input parameters for the API “blc\_smp\_configSecurityRequestSending” supports the nine combinations below:

| <del>reConn_cfg<br/>newConn_cfg</del> | <i>SecReq_NOT_SEND</i>                                                                                                       | <i>SecReq_IMM_SEND</i>                                                                                                      | <i>SecReq_PEND_SEND</i>                                                                                                     |
|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <i>SecReq_NOT_SEND</i>                | Not send SecReq after the first connection or reconnection (the para pending_ms is invalid).                                 | Not send ecReq after the first connection, and immediately send SecReq after reconnection (the para pending_ms is invalid). | Not send ecReq after the first connection, and wait for pending_ms milliseconds to send SecReq after reconnection.          |
| <i>SecReq_IMM_SEND</i>                | Immediately send SecReq after the first connection, and not send SecReq after reconnection (the para pending_ms is invalid). | Immediately send SecReq after the first connection or reconnection (the para pending_ms is invalid).                        | Immediately send SecReq after the first connection, and wait for pending_ms milliseconds to send SecReq after reconnection. |
| <i>SecReq_PEND_SEND</i>               | Wait for pending_ms milliseconds to send SecReq after the first connection, and not send SecReq after reconnection.          | Wait for pending_ms milliseconds to send SecReq after the first connection, and immediately send SecReq after reconnection. | Wait for pending_ms milliseconds to send SecReq after the first connection or reconnection.                                 |

Following shows two examples:

- 1) newConn\_cfg: SecReq\_NOT\_SEND

reConn\_cfg: SecReq\_NOT\_SEND

pending\_ms: This parameter does not take effect.

When newConn\_cfg is set as SecReq\_NOT\_SEND, it means new Slave device won't actively initiate Security Request, and it will only respond to the pairing request from the peer device. If the peer device does not send pairing request, encryption pairing won't be executed.

As shown in the figure below, when Master sends a pairing request packet “SM\_Pairing\_Req”, Slave will respond to it, but won't actively trigger Master to initiate pairing request.

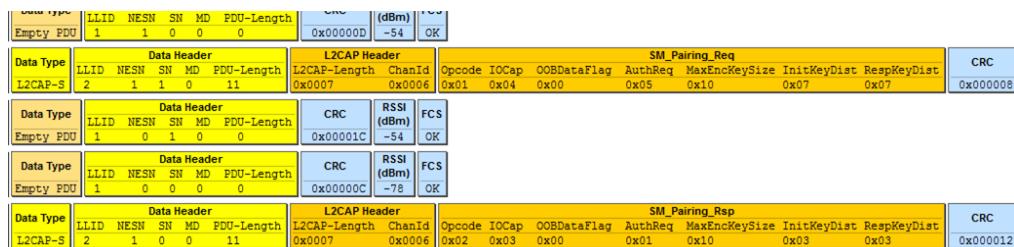


Figure3-65 Packet example for Pairing Peer Trigger

When reConn\_cfg is set as SecReq\_NOT\_SEND, it means device pairing has already been completed, and Slave won't send Security Request after reconnection.

## 2) newConn\_cfg: SecReq\_IMM\_SEND

reConn\_cfg: SecReq\_NOT\_SEND

pending\_ms: This parameter does not take effect.

When newConn\_cfg is set as SecReq\_IMM\_SEND, it means new Slave device will immediately send Security Request to Master after connection, to trigger Master to start pairing process.

As shown in the figure below, Slave actively sends a SM\_Security\_Req to trigger Master to send pairing request.

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

Figure3-66 Packet example for Pairing Conn Trigger

When reConn\_cfg is set as SecReq\_NOT\_SEND, it means Slave won't send Security Request after reconnection.

The SDK also provides an API to send Security Request packet only for special use case. The APP layer can invoke this API to send Security Request at any time.

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

Note: If user invokes the “blc\_smp\_configSecurityRequestSending” to control secure pairing request packet, the “blc\_smp\_sendSecurityRequest” should not be invoked.

### 3.3.4.4 SMP bonding info

SMP bonding information herein is discussed relative to Slave device. User can refer to the code of “direct advertising” setting during initialization in the demo “8258\_ble\_remote”.

Slave can store pairing information of up to four Master devices at the same time, so that all of the four devices can be reconnected successfully.

The API below serves to set the max number of bonding devices with the upper limit of 4 (SMP\_BONDING\_DEVICE\_MAX\_NUM) which is also the default value.

```
#define SMP_BONDING_DEVICE_MAX_NUM 4
ble_sts_t blc_smp_param_setBondingDeviceMaxNumber(int device_num);
```

If using blc\_smp\_param\_setBondingDeviceMaxNumber (4) to set the max number as 4, after four devices have been paired, executing pairing for the fifth device will automatically delete the pairing info of the earliest connected (first) device, so as to store the pairing info of the fifth device.

If using blc\_smp\_param\_setBondingDeviceMaxNumber (2) to set the max number as 2, after two devices have been paired, executing pairing for the third device will automatically delete the pairing info of the earliest connected (first) device, so as to store the pairing info of the third device.

The API below serves to obtain the number of currently bonded Master devices (successfully paired with Slave) stored in the flash.

```
u8 blc_smp_param_getCurrentBondingDeviceNumber(void);
```

If the return value is 3, it means the number of currently bonded Master devices is 3, and all of the three devices can be reconnected successfully.

## 1. Storage sequence for bonding info

Index is a concept related to BondingDeviceNumber. If current BondingDeviceNumber is 1, there's only one bonding device whose index is 0; if BondingDeviceNumber is 2, there're two bonding devices with index 0 and 1.

The SDK provides two methods to update device index, Index\_Update\_by\_Connect\_Order and Index\_Update\_by\_Pairing\_Order, i.e. update index as per the time sequence of lastest connection or pairing for devices.

The API below serves to select index update method.

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

Following shows the enum type index\_updateMethod\_t:

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

Note: The old SDK version only supports the second method, while the new SDK version supports the two methods selectable via the API and the first method is selected by default.

## 1) Index\_Update\_by\_Connect\_Order

If BondingDeviceNumber is 2, device index stored in Slave flash includes 0 and 1. Index sequence is updated by the order of the latest successful connection rather than the latest pairing. Suppose Slave is paired with MasterA and MasterB in sequence, since MasterB is the latest connected device, the index for MasterA is 0, while the index for MasterB is 1. Then reconnect Slave with MasterA. Now MasterA becomes the latest connected device, so the index for MasterB is 0, and the index for MasterA is 1.

If BondingDeviceNumber is 3, device index includes 0, 1 and 2. The index for the latest connected device is 2, and index for the earliest connected device is 0.

If BondingDeviceNumber is 4, device index includes 0, 1, 2 and 3. The index for the latest connected device is 3, and index for the earliest connected device is 0. Suppose Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, the index for the latest connected MasterD is 3. If Slave is reconnected with MasterB, the index for the latest connected MasterB is 3.

Since the upper limit for bonding devices is 4, please note the case when more than four Master devices are paired: When Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, pairing Slave with MasterE will make Slave delete the pairing info for MasterA; however, if Slave is reconnected with MasterA before pairing Slave with MasterE, since the sequence changes to B-C-D-A, the latest pairing operation between Slave and MasterE will delete the pairing info for MasterB.

## 2) Index\_Update\_by\_Pairing\_Order

If BondingDeviceNumber is 2, device index stored in Slave flash includes 0 and 1. Index sequence is updated by the order of the latest pairing. Suppose Slave is paired with MasterA and MasterB in sequence, since MasterB is the latest paired device, the index for MasterA is 0, while the index for MasterB is 1. Then reconnect Slave with MasterA. Now the index sequence for MasterA and MasterB is not changed.

If BondingDeviceNumber is 4, device index includes 0, 1, 2 and 3. The index for the latest paired device is 3, and the index for the earliest paired device is 0. Suppose Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, the index for the latest paired MasterD is 3. No matter how Slave is reconnected with MasterA/B/C/D, the index sequence won't be changed.

Note: When Slave is paired with MasterA, MasterB, MasterC and MasterD in sequence, pairing Slave with MasterE will make Slave delete the pairing info for MasterA; if Slave is reconnected with MasterA before pairing Slave with MasterE, since the sequence is still A-B-C-D, the latest pairing operation between Slave and MasterE will delete the pairing info for MasterA.

## 2. Format for bonding info and related APIs

Bonding info of Master device is stored in flash with the format below:

```
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;
```

Bonding info includes 64 bytes.

- ✧ peer\_addr\_type and peer\_addr indicate Master connection address in the Link Layer which is used during device direct advertising.
- ✧ peer\_id\_addrType/peer\_id\_addr and peer\_irk are identity address and irk declared in the key distribution phase.

Only when the peer\_addr\_type and peer\_addr are Resolvable Private Address (RPA), and address filtering is needed, should related info be added into resolving list for Slave to analyze it (refer to TEST\_WHITELIST in the 8258\_feature\_test).

Other parameters are negligible to user.

The API below serves to obtain device information from flash by using index.

```
u32 bls_smp_param_loadByIndex(u8 index,
 smp_param_save_t* smp_param_load);
```

If the return value is 0, it indicates failure to get info; non-zero return value indicates starting flash address to store the info.

For example, suppose there're three bonded devices, user can invoke the "bls\_smp\_param\_loadByIndex(2, ...)" to get related info of the latest device.

The API below serves to obtain bonding device info from flash by using Master address (connection address in the Link Layer).

```
u32 bls_smp_param_loadByAddr(u8 addr_type,
 u8* addr, smp_param_save_t* smp_param_load);
```

If the return value is 0, it indicates failure to get info; non-zero return value indicates starting flash address to store the info.

The API below is used for Slave device to erase all pairing info stored in local flash.

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

Note: Before invoking this API, please ensure the device is in non-connection state.

The API below is used for Slave device to configure address to store pairing info in flash.

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

User can set the parameter “addr” as needed, and please refer to the **section 2.1.4 SDK flash space partition** so as to determine a suitable flash area for bonding info storage.

### 3.3.4.5 Master SMP

The highest security level supported by SMP function of Master in legacy pairing method is LE security mode1 level2 (Legacy pairing Just Works method). User can refer to the demo “8258 master kma dongle”, and it's only needed to modify the macro below in the file “8258 master kma dongle/app\_config.h”.

- ```
#define BLE_HOST_SMP_ENABLE          0
```
- ❖ If the macro is configured as 1, standard SMP is used, and the highest security level supported by Master is set as LE security mode1 level2 to support legacy pairing Just Works method.
 - ❖ If the macro is configured as 0, non-standard self-defined pairing management function is enabled.

1. Enable Master SMP by setting the macro `BLE_HOST_SMP_ENABLE` as 1

When using this security level configuration, the API below must be invoked to initialize SMP parameter setting, including initialization setting for bonding flash area.

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

If only this API is invoked during initialization, the SDK will use the default parameters below to configure SMP:

- ✧ The highest security level supported by default:
Unauthenticated_Paring_with_Encryption
- ✧ Default bonding mode: Bondable_Mode (store KEY that is distributed after pairing encryption into flash)
- ✧ Default IO capability: IO_CAPABILITY_NO_INPUT_NO_OUTPUT

When pairing device supports LE security mode1 level2, the three APIs below should be configured:

```
void blm_smp_configParingSecurityInfoStorageAddr (int addr);  
void blm_smp_registerSmpFinishCb (smp_finish_callback_t cb);  
void blm_host_smp_setSecurityTrigger(u8 trigger);
```

1) `blm_smp_configParingSecurityInfoStorageAddr`

This API can be used for Master device to configure flash address to store bonding info. User can modify the parameter “addr” as needed.

2) `blm_smp_registerSmpFinishCb`

This callback function is triggered after key distribution is completed in the third phase of pairing. User can register it in the APP layer to obtain pairing completed event.

3) `blm_host_smp_setSecurityTrigger`

This API mainly serves to configure whether Master will actively initiate encryption and whether actively encrypt the link during reconnection.

Supported parameter options:

<code>#define SLAVE_TRIGGER_SMP_FIRST_PAIRING</code>	0
<code>#define MASTER_TRIGGER_SMP_FIRST_PAIRING</code>	BIT(0)
<code>#define SLAVE_TRIGGER_SMP_AUTO_CONNECT</code>	0

```
#define MASTER_TRIGGER_SMP_AUTO_CONNECT BIT(1)
```

- A. For the first pairing, determine whether Master will actively initiate pairing request or start pairing after receiving Security Request from Slave.
- B. For reconnection of a bonded device, determine whether Master will actively send LL_ENC_REQ to encrypt the link or start link encryption after receiving Security Request from Slave.

Generally the adopted configuration is: Master actively initiates pairing request for the first pairing, and actively sends LL_ENC_REQ for reconnection.

Following shows user initialization code reference. Please refer to the demo “8258 master kma dongle” for details.

```
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);
```

The APIs below related to bonding info of Master side are used by Master SMP stack bottom layer, and are negligible to user.

```
inttbl_bond_slave_search(u8 adr_type, u8 * addr);  
inttbl_bond_slave_delete_by_addr(u8 adr_type, u8 *addr);  
voidtbl_bond_slave_unpair_proc(u8 adr_type, u8 *addr);
```

2. Non-standard self-defined pairing management (set the macro “BLE_HOST_SMP_ENABLE” as 0)

In the demo “8258 master kma dongle”, if SMP is disabled, the SDK cannot automatically complete pairing and un-pairing operation, so pairing management should be added in the APP layer.

When using self-defined pairing management, initialization related APIs are shown as below:

```
blc_smp_setSecurityLevel(No_Security);//disable SMP function  
user_master_host_pairing_flash_init();//custom method
```

1) Design flash storage method

The default flash sector used for pairing is 0x78000 ~ 0x78FFF, and it's modifiable in the “app_config.h”.

```
#define FLASH_ADR_PAIRING 0x78000
```

Starting from flash address 0x78000, every eight bytes form an area (named 8 bytes area). Each area can store MAC address of one Slave, and includes 1-byte bonding mark, 1-byte address type and 6-byte MAC address.

```
typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;
```

All valid Slave MAC addresses are stored in 8 bytes areas successively:

- ❖ The first valid Slave MAC address is stored in 0x78000~0x78007, and the mark in 0x78000 is set as “0x5A” to indicate current address is valid.
- ❖ The second valid Slave MAC address is stored in the next 8 bytes area 0x78008~0x7800f and the mark in 0x78008 is set as “0x5A”.
- ❖ The third valid Slave MAC address is stored in the next 8 bytes area 0x78010~0x78017 and the mark in 0x78010 is set as “0x5A”.

To un-pair certain Slave device, it's needed to erase its MAC address in the Dongle side by setting the mark of the corresponding 8 bytes area as “0x00”. For example, to erase the MAC address of the first Slave device as shown above, user should set 0x78000 as “0x00”.

The reason to adopt this design is: During execution of program, the SDK cannot invoke the function “flash_erase_sector” to erase flash, since this operation takes 20~200ms to erase a 4kB sector of flash and thus will result in BLE timing error.

Mark of “0x5A” and “0x00” are used to indicate pairing storage and un-pairing erasing of all Slave MAC addresses.

Considering 8 bytes areas may occupy the whole 4kB sector of flash and thus result in error, a special processing is added during initialization: Read info of 8 bytes areas starting from address 0x78000, and store all valid MAC addresses into Slave MAC table of RAM. During this process, it will check whether there're too many 8 bytes areas. If yes, erase the whole sector and then write the contents of Slave MAC table in RAM back to 8 bytes areas starting from 0x78000.

2) Slave MAC table

```

/*
 * define pair slave max num,
 * if exceed this max num, two methods to process new slave pairing
 * method 1: overwrite the oldest one(telink use this method)
 * method 2: not allow paring unness unpair happend  */
#define USER_PAIR_SLAVE_MAX_NUM      1 //telink use max 1

typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;

typedef struct {
    u32 bond_flash_idx[USER_PAIR_SLAVE_MAX_NUM]; //mark paired slave mac address
    macAddr_t bond_device[USER_PAIR_SLAVE_MAX_NUM]; //macAddr_t already defined
    u8 curNum;
} user_slaveMac_t;

user_slaveMac_t user_tbl_slaveMac;

```

The structure above serves to use Slave MAC table in RAM to maintain all paired devices.

The macro “USER_PAIR_SLAVE_MAX_NUM” serves to set the max allowed number of maintainable paired devices, and the default value is 1 which indicates one paired device is maintainable. User can modify this value as needed.

Suppose the “USER_PAIR_SLAVE_MAX_NUM” is set as 3 to indicate up to three paired devices can be maintained. In the “user_tbl_slaveMac”, the “curNum” indicates the number of current valid Slave devices in flash, the array “bond_flash_idx” records offset relative to 0x78000 for starting address of each valid 8 bytes area in flash (When un-pairing certain device, based on corresponding offset, user can locate the mark of the 8 bytes area, and then write the mark as 0x00), while the array “bond_device” records MAC address.

3) Related APIs

Based on the design of flash storage and Slave MAC table above, user can invoke the APIs below.

A. user_master_host_pairing_flash_init

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

This API should be invoked to implement flash initialization when enabling user-defined pairing management.

B. user_tbl_slave_mac_add

```
int user_tbl_slave_mac_add(u8 adr_type, u8 *adr);
```

The API above should be invoked when a new device is paired, and it serves to add one Slave MAC address.

The return value should be either 1 (success) or 0 (failure).

The API will check whether current number of devices in flash and Slave MAC table has reached the maximum.

- ✧ If not, directly add the MAC address of the new device into Slave MAC table, and store it in an 8 bytes area of flash.
- ✧ If yes, the viable processing policy may be: “pairing is not allowed”, or “directly delete the earliest MAC address”. Telink demos adopts the latter. Since Telink supported max number of paired device is 1, this method will preempt current paired device, i.e. delete current device by using the “user_tbl_slave_mac_delete_by_index(0)” and then add MAC address of new device into Slave MAC table.

User can modify the implementation of this API as per his own policy.

C. user_tbl_slave_mac_search

```
int user_tbl_slave_mac_search(u8 adr_type, u8 * adr)
```

This API serves to check whether the device is already available in Slave MAC table according to device address reported by adv, i.e. whether the device sending adv packet currently has already been paired with Master. The device that has already been paired can be directly reconnected.

D. user_tbl_slave_mac_delete_by_addr

```
int user_tbl_slave_mac_delete_by_addr(u8 adr_type, u8 *adr)
```

This API serves to delete MAC addr of certain paired device from Slave MAC table by specified address.

E. user_tbl_slave_mac_delete_by_index

```
void user_tbl_slave_mac_delete_by_index(int index)
```

This API serves to delete MAC addr of certain paired device from Slave MAC table by specified index.

The parameter “index” indicates device pairing sequence. If the max pairing number is 1, the index for the paired device is always 0; if the max pairing

number is 2, the index for the first paired device is 0, and the index for the second paired device is 1.....

F. user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

This API serves to delete MAC addr of all the paired devices from Slave MAC table.

G. user_tbl_salve_mac_unpair_proc

```
void user_tbl_salve_mac_unpair_proc(void)
```

This API serves to process un-pairing. The demo code adopts the processing method using the default max pairing number (1) to delete all paired devices. User can modify the implementation of the API.

4) Connection and pairing

When Master receives adv packet reported by Controller, it will establish connection with Slave in the two cases below:

- Invoke the function “user_tbl_slave_mac_search” to check whether current Slave device has already been paired with Master and un-pairing has not been executed. If yes, Master can automatically establish connection with the device.

```
master_auto_connect = user_tbl_slave_mac_search(pa->adr_type,  
pa->mac);  
  
if(master_auto_connect) { create connection }
```

- If current adv device is not available in Slave MAC table, auto connection won't be initiated, and it's needed to check whether manual pairing condition is met. The SDK provides two manual pairing solutions by default.

Premise: Current adv device is close enough.

Solution 1: The pairing button on Master Dongle is pressed.

Solution 2: Current adv data is pairing adv packet data defined by Telink.

```
//user design manual paring methods  
user_manual_paring = dongle_pairing_enable && (rssi > -56); //button trigger pairing(rssi threshold, short distance)  
if(!user_manual_paring){ //special adv pair data can also trigger pairing  
    user_manual_paring = (memcmp(pa->data, telink_adv_trigger_paring, sizeof(telink_adv_trigger_paring)) == 0) && (rssi > -56  
}
```

```
if(user_manual_paring) { create connection }
```

After connection triggered by manual pairing is established successfully, the current device is added into Slave MAC table when reporting “HCI LE CONNECTION ESTABLISHED EVENT”.

```
// 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);  
}
```

5) Un-pairing

```
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_slave_mac_unpair_proc();  
                #endif  
            }  
        }  
        if(master_disconnect_flag && blc_ll_getCurrentState() != BLS_LINK_STATE_CONN){  
            master_disconnect_flag = 0;  
        }  
    }  
}
```

As shown in the code above, when un-pairing condition is triggered, Master first invokes the “blm_ll_disconnect” to terminate connection, and then invokes the “user_tbl_slave_mac_unpair_proc” to process un-pairing. The demo code will directly delete all paired devices. In the default case, the max pairing number is 1, so only one device will be deleted. If user sets the max number larger than 1, the “user_tbl_slave_mac_delete_by_adr” or “user_tbl_slave_mac_delete_by_index” should be invoked to delete specified device.

The demo code provides two conditions to trigger un-pairing:

- ✧ The un-pairing button on Master Dongle is pressed.
- ✧ The un-pairing key value “0xFF” is received in “HID keyboard report service”.

User can modify un-pairing trigger method as needed.

3.3.5 GAP

3.3.5.1 GAP initialization

GAP initialization for Master and Slave is different.

Slave uses the API below to initialize GAP.

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

Master uses the API below to initialize GAP.

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

As introduced earlier, data transfer between the APP layer and the Host is not controlled via GAP; the ATT, SMP and L2CAP can directly communicate with the APP layer via corresponding interface. In current SDK version, the GAP layer mainly serves to process events in the Host layer, and GAP initialization mainly registers processing function entry for events in the Host layer.

3.3.5.2 GAP event

GAP event is generated during the communication process of Host protocol layers such as ATT, GATT, SMP and GAP. As introduced earlier, current SDK supports two types of event: Controller event, and GAP (Host) event. Controller event also includes two sub types: HCI event, and Telink defined event.

GAP event processing is added in current BLE SDK, which enables the protocol stack to layer events more clearly and to process event communication in the user layer more conveniently. SMP related processing, such as Passkey input and notification of pairing result to user, is also included.

If user wants to receive GAP event in the APP layer, it's needed to register the corresponding callback function, and then enable the corresponding mask.

Following shows the prototype and register interface for callback function of GAP event.

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

The "u32 h" in the callback function prototype is the mark of GAP event which will be frequently used in the bottom layer protocol stack.

Following lists some events which user may use.

#define GAP_EVT_SMP_PARING_BEAGIN	0
#define GAP_EVT_SMP_PARING_SUCCESS	1
#define GAP_EVT_SMP_PARING_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

In the callback function prototype, “para” and “n” indicate data and data length of event. User can refer to the usage below in the “8258 featuretest/feature_security.c” and the implementation of the function “app_host_event_callback”.

```
blc_gap_registerHostEventHandler( app_host_event_callback );
```

The API below serves to enable the mask for GAP event.

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

Following lists the definition for some common eventMasks. For other event masks, user can refer to the “ble/gap/gap_event.h”.

```
#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)
```

If user does not set GAP event mask via this API, the APP layer won't receive notification when corresponding GAP event is generated.

Note: For the description about GAP event below, it's supposed that GAP event callback has been registered, and corresponding eventMask has been enabled.

1. GAP_EVT_SMP_PARING_BEAGIN

- 1) Event trigger condition: When entering connection state, Slave sends a SM_Security_Req command, and Master sends a SM_Pairing_Req to request for pairing. When Slave receives the pairing request, this event will be triggered to indicate that pairing starts.

Data Type	Data Header				L2CAP Header		SM_Security_Req								
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AuthReq	0x0B	01				
L2CAP-S	2	1	0	0	6	0x0002	0x0006								
Data Type	Data Header				L2CAP Header		SM_Pairing_Req								
L2CAP-S	2	1	1	0	11	0x0007	0x0006	Opcode	IOCap	OBBDATAFlag	AuthReq	MaxEncKeySize	InitKeyDist	RespKeyDist	0x03
								0x01	0x03	0x00	0x01	0x10	0x02	0x03	

Figure3-67 Master initiates Pairing_Req

- 2) Data length "n": 4.
- 3) Pointer "p": p points to data of a memory area, corresponding to the structure below.

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

◊ "connHandle": current connection handle.

◊ "secure_conn": If it's 1, secure encryption feature (LE Secure Connections) will be used; otherwise LE legacy pairing will be used.

◊ "tk_method": It indicates the method of TK value to be used in the subsequent pairing, e.g. JustWorks, PK_Init_Dsply_Resp_Input, PK_Resp_Dsply_Init_Input, Numric_Comparison.
```

2. GAP_EVT_SMP_PARING_SUCCESS

- 1) Event trigger condition: This event will be generated when the whole pairing process is completed correctly. This phase is called “Key Distribution, Phase 3” of LE pairing phase. If there’s key to be distributed, the pairing success event will be triggered after the two sides have completed key distribution; otherwise the pairing success event will be triggered directly.
- 2) Data length “n”: 4.
- 3) Pointer “p”: p points to data of a memory area, corresponding to the structure below.

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

◆ "connHandle": current connection handle.

◆ "bonding": If it's 1, bonding function is enabled; otherwise bonding function is disabled.

◆ "bonding_result": It indicates bonding result. If bonding function is disabled, the result value should be 0. If bonding function is enabled, it's also needed to check whether encryption Key is correctly stored in flash; if yes, the result value is 1; otherwise the result value is 0.
```

3. GAP_EVT_SMP_PARING_FAIL

- 1) Event trigger condition: If Slave or Master does not conform to standard pairing flow, or pairing process is terminated due to abnormality such as error report during communication, this event will be triggered.
- 2) Data length “n”: 2.
- 3) Pointer “p”: p points to data of a memory area, corresponding to the structure below.

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

◆ "connHandle": current connection handle.

◆ "reason": It indicates the reason for pairing failure. Following lists some common reason values, and for other values, please refer to the file "stack/ble/smp/smp_const.h".
```

For the definition of pairing failure values, please refer to “Core_v5.0” (Vol 3/Part H/3.5.5 “Pairing Failed”).

<code>#define PARING_FAIL_REASON_CONFIRM_FAILED</code>	0x04
<code>#define PARING_FAIL_REASON_PARING_NOT_SUPPORTED</code>	0x05
<code>#define PARING_FAIL_REASON_DHKEY_CHECK_FAIL</code>	0x0B
<code>#define PARING_FAIL_REASON_NUMUERIC_FAILED</code>	0x0C
<code>#define PARING_FAIL_REASON_PARING_TIEMOUT</code>	0x80
<code>#define PARING_FAIL_REASON_CONN_DISCONNECT</code>	0x81

4. GAP_EVT_SMP_CONN_ENCRYPTION_DONE

- 1) Event trigger condition: When Link Layer encryption is completed (the LL receives “start encryption response” from Master), this event will be triggered.
- 2) Data length “n”: 3.
- 3) Pointer “p”: p points to data of a memory area, corresponding to the structure below.

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

- ✧ “connHandle”: current connection handle.
- ✧ “re_connect”: If it’s 1, it indicates fast reconnection (The LTK distributed previously will be used to encrypt the link); if it’s 0, it indicates current encryption is the first encryption.

5. GAP_EVT_SMP_TK_DISPALY

- 1) Event trigger condition: After Slave receives a Pairing_Req from Master, as per the pairing parameter configuration of the peer device and the local device, the method of TK (pincode) value to be used for pairing will be known. If the method “PK_Resp_Dsplay_Init_Input” is enabled, which means Slave displays 6-digit pincode and Master inputs 6-digit pincode, this event will be triggered.
- 2) Data length “n”: 4.
- 3) Pointer “p”: p points to an u32-type variable “tk_set”. The value is 6-digit pincode that Slave needs to inform the APP layer, and the APP layer needs to display the pincode.

The user can also manually set a user-specified pincode code such as "123456" without using the 6-digit pincode code generated by underlying layer randomly.

```
case GAP_EVT_SMP_TK_DISPALY:  
{  
    char pc[7];  
    #if 1 //Set pincode manually  
    u32 pinCode = 123456;  
    memset(smp_param_own.paring_tk, 0, 16);  
    memcpy(smp_param_own.paring_tk, &pinCode, 4);  
  
    #else// Using the pincode generated by underlying layer  
    randomly.  
    u32 pinCode = *(u32*)para;  
    #endif  
}  
  
break;
```

User should get the 6-digit pincode from Slave and input the pincode on Master side (e.g. Mobile phone), to finish TK input and continue pairing process. If user has input wrong pincode, or has clicked “cancel”, the pairing process fails.

The demo “sdk/8258_feature_test/feature_security.c” provides an example for Passkey Entry application.

6. GAP_EVT_SMP_TK_DISPALY

- 1) Event trigger condition: When Slave device enables the Passkey Entry method, and “PK_Init_Dsply_Resp_Input” or “PK_BOTH_INPUT” is used for pairing, this event will be triggered to inform user that TK value should be input. After this event is received, user needs to input TK value within 30s via IO input capability, otherwise pairing will fail due to timeout.) For the API “blc_smp_setTK_by_PasskeyEntry” to input TK value, please refer to **section 3.3.4.2 SMP parameter configuration**.
- 2) Data length “n”: 0.
- 3) Pointer “p”: Null pointer.

7. GAP_EVT_SMP_TK_REQUEST_OOB

- 1) Event trigger condition: When Slave device enables the OOB method of legacy pairing, this event will be triggered to inform user that 16-digit TK value should be input by the OOB method. After this event is received, user needs to input 16-digit TK value within 30s via IO input capability, otherwise pairing will fail due to timeout. For the API “blc_smp_setTK_by_OOB” to input TK value, please refer

to section 3.3.4.2 SMP parameter configuration.

- 2) Data length “n”: 0.
- 3) Pointer “p”: Null pointer.

8. GAP_EVT_SMP_TK_NUMERIC_COMPARE

- 1) Event trigger condition: After Slave receives a Pairing_Req from Master, as per the pairing parameter configuration of the peer device and the local device, the method of TK (pincode) value to be used for pairing will be known. If the method “Numeric_Comparison” is enabled, this event will be triggered immediately.

For “Numeric_Comparison”, a method of SMP4.2 secure encryption, dialog window will pop up on both Master and Slave to show 6-digit pincode, “YES” and “NO”; user needs to check whether pincodes on the two sides are consistent, and decide whether to click “YES” to confirm TK check result is OK.

- 2) Data length “n”: 4.
- 3) Pointer “p”: p points to an u32-type variable “pinCode”. The value is 6-digit pincode that Slave needs to inform the APP layer. The APP layer needs to display the pincode, and supplies “YES or “NO” confirmation mechanism.

The demo “sdk/8258_feature_test/feature_security.c” provides an example for Numeric_Comparison application.

9. GAP_EVT_ATT_EXCHANGE_MTU

- 1) Event trigger condition: This event will be triggered in either of the two cases below.
 - a) Master sends “Exchange MTU Request”, and Slave responds with “Exchange MTU Response”.
 - b) Slave sends “Exchange MTU Request”, and Master responds with “Exchange MTU Response”.
- 2) Data length “n”: 6.
- 3) Pointer “p”: p points to data of a memory area, corresponding to the structure below.

```
typedef struct {
    u16 connHandle;
    u16 peer_MTU;
    u16 effective_MTU;
} gap_gatt_mtuSizeExchangeEvt_t;
```

- ✧ connHandle: current connection handle.
- ✧ peer_MTU: RX MTU value of the peer device.
- ✧ effective_MTU = min(CleintRxMTU, ServerRxMTU). “CleintRxMTU” and “ServerRxMTU” indicate RX MTU size value of Client and Server respectively. After Master and Slave exchanges MTU size of each other, the minimum of the two values is used as the maximum MTU size value for mutual communication between them.

10. GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM

- 1) Event trigger condition: Whenever the APP layer invokes the “bls_att_pushIndicateData” or “blc_gatt_pushHandleValueIndicate” to send indicate data to Master, Master will respond with a confirmation for the data. This event will be triggered when Slave receives the confirmation.
- 2) Data length “n”: 0.
- 3) Pointer “p”: Null pointer.

4 Low Power Management (PM)

Low Power Management is also called Power Management, or PM as referred by this document.

4.1 Low power driver

4.1.1 Low power mode

For the 8x5x family, when MCU works in normal mode, or working mode, current is about 3~7mA. To save power consumption, MCU should enter low power mode.

There are three low power modes, or sleep modes: Suspend mode, Deepsleep mode, and Deepsleep retention mode.

The table below illustrates statistically data retention and loss for SRAM, digester registers and analog registers during each sleep mode.

sleep mode module	suspend	deepsleep retention	deepsleep
----------------------	---------	---------------------	-----------

Sram		100% keep	Retain first 16K(or 32K), others lost	100% lost
register	digital register	99% keep	100% lost	100% lost
	analog register	100% keep	99% lost	99% lost

1) Suspend mode (sleep mode 1)

In this mode, program execution pauses, most hardware modules of MCU are powered off, and the PM module still works normally.

In this mode, IC current is about 60-70uA. Program execution continues after wakeup from suspend mode.

In suspend mode, data of the SRAM and all analog registers are maintained, and most of the digital registers are also non-volatile except a few such as:

- a) A few digital registers of baseband.

User should pay close attention to the registers configurd by the API “rf_set_power_level_index”. This API needs to be invoked after each wakeup from suspend mode.

- b) Digital registers to control Dfifo state.

Related APIs in the “divers/8258/dfifo.h” should be invoked after each wakeup from suspend mode.

2) Deepsleep mode (sleep mode 2)

In this mode, program execution pauses, vast majority of hardware modules are powered off, and the PM module still works.

In this mode, IC current is less than 1uA, but if flash standby current comes up at 1uA or so, total current may reach 1~2uA.

After wakeup from deepsleep mode, similar to power on reset, MCU will restart, and program will reboot and re-initialize.

In deepsleep mode, except a few retention analog registers, data of all registers (analog & digital) and SRAM are lost.

3) Deepsleep retention mode (sleep mode 3)

In deepsleep mode, current is very low, but all SRAM data are lost; while in suspend mode, though SRAM and most registers are non-volatile, current is increased.

Deepsleep with SRAM retention (deepsleep retention or deep retention) mode is designed in the 8x5x family, so as to achieve application scenes with low sleep current and quick wakeup to restore state, e.g. maintain BLE connection during long sleep.

Corresponding to 16K or 32K SRAM retention area (i.e. first 16k/32k SRAM data are non-volatile), deepsleep retention 16K Sram and deepsleep retention 32K Sram are introduced.

In this mode, program execution pauses, vast majority of hardware modules are powered off, and the PM module still works. Relative to deepsleep mode, with extra current required for SRAM retention, current in deep retention mode is 2~3uA.

After wakeup from deep retention mode, MCU will restart, and program will reboot and re-initialize.

For Telink 826x family, suspend mode with current no more than 10uA is adopted to maintain BLE connection during long sleep.

For the 8x5x family, deepsleep retention mode is adopted instead to avoid large current of suspend mode.

For both deepsleep retention mode and deepsleep mode, almost all of registers will lose their data; however, deepsleep retention mode supports 16k or 32k SRAM retention area.

The retention analog registers below apply to both deepsleep mode and deepsleep retention mode.

- 1) Analog registers to control GPIO pullup/down resistance

When configured via the API “gpio_setup_up_down_resistor” or the following method in the app_config.h, GPIO pullup/down resistance are non-volatile:

```
#define PULL_WAKEUP_SRC_PD5      PM_PIN_PULLDOWN_100K
```

As introduced in **section 2.3 GPIO module**, gpio output is controlled by digital registers.

For 826x family, during suspend mode, gpio output can be used to control some peripheral devices.

For 8x5x family, gpio output state data is lost when sleep mode is switch from suspend to deepsleep retention; to control peripherals, instead of using gpio output state, GPIO pull up/down states apply: use “pullup 10K” to replace “gpio output high”, and “pull down100K” to replace “gpio output low”.

2) Special retention analog registers of the PM module:

The code below shows the “DEEP_ANA_REG” in the “drivers/8258/pm.h”.

```
#define DEEP_ANA_REG0      0x3a  
#define DEEP_ANA_REG1      0x3b  
#define DEEP_ANA_REG2      0x3c //system used, user cannot use
```

Note: User is not allowed to use the ana_3c which is dedicated for bottom layer stack. If code in the APP layer has used this register, it's needed to use ana_3a or ana_3b to replace it. Due to the limited number of retention analog registers, it's recommended to use each bit to indicate different status info. Please refer to the “8258_ble_remote” under the vendor directory.

The retention analog registers below may lose information due to incorrect GPIO wakeup. For example, suppose user configures GPIO_PAD high level to wake up MCU from deepsleep, if the GPIO is already high level before invoking the function “cpu_sleep_wakeup”, it will lead to incorrect GPIO wakeup, and thus these analog registers will lose their contents.

```
#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 can use these retention analog regsites to store important info, so that the data are retained across deepsleep/deepsleep retention.

4.1.2 Low power wakeup source

8x5x MCU only supports GPIO PAD wakeup and timer wakeup from suspend/deepsleep/ deepsleep retention, as designed in this BLE SDK.

Note that neither the “PM_TIM_RECOVER_START” nor the “PM_TIM_RECOVER_END” indicate wakeup source.

```
typedef enum {  
    PM_WAKEUP_PAD = BIT(4),  
    PM_WAKEUP_TIMER = BIT(6),  
} SleepWakeupsSrc_TypeDef;
```

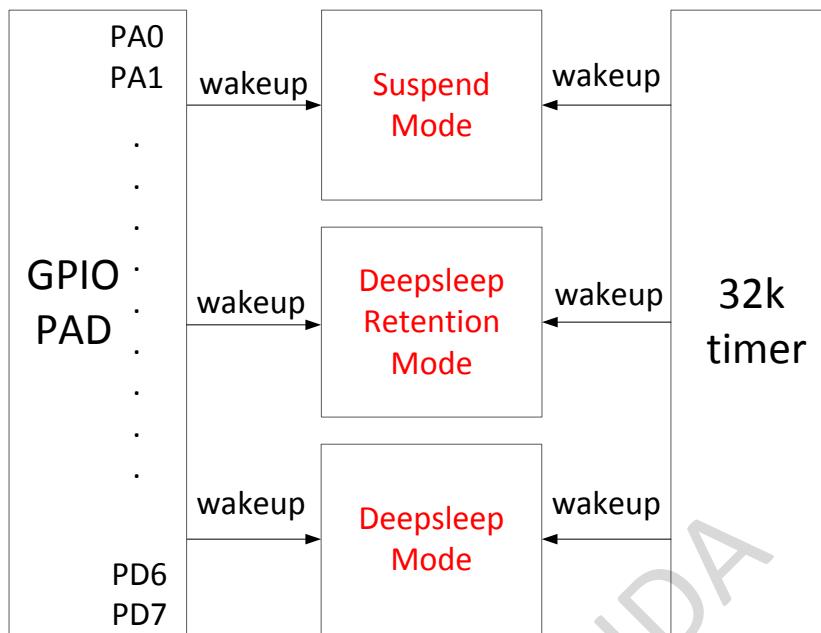


Figure 4-1 8x5x MCU HW wakeup source

As shown above, there are two hardware wakeup sources: TIMER and GPIO PAD.

- ✧ The “PM_WAKEUP_TIMER” comes from 32k HW timer (32k RC timer or 32k Crystal timer). Since 32k timer is correctly initialized in the SDK, no configuration is needed except setting wakeup source in the “cpu_sleep_wakeup ()”.
- ✧ The “PM_WAKEUP_PAD” comes from GPIO module. Except 4 MSPI pins, all GPIOs (PAx/PBx/PCx/PDx) support high or low level wakeup .

The API below serves to configure GPIO PAD as wakeup source for sleep mode.

```

typedef enum{
    Level_Low = 0,
    Level_High,
} GPIO_LevelTypeDef;

void cpu_set_gpio_wakeup (GPIO_PinTypeDef pin,
                           GPIO_LevelTypeDef pol, int en);
  
```

- ✧ “pin”: GPIO pin
- ✧ “pol”: wakeup polarity
Level_High: high level wakeup, Level_Low: low level wakeup
- ✧ “en”: 1-enable, 0-disable.

Examples:

```
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 1); //Enable GPIO_PC2 PAD high level wakeup
```

```
cpu_set_gpio_wakeup(GPIO_PC2, Level_High, 0); //Disable GPIO_PC2 PAD wakeup  
cpu_set_gpio_wakeup(GPIO_PB5, Level_Low, 1); //Enable GPIO_PB5 PAD low level wakeup  
cpu_set_gpio_wakeup(GPIO_PB5, Level_Low, 0); // Disable GPIO_PB5 PAD wakeup
```

4.1.3 Sleep and wakeup from low power mode

The API below serves to configure MCU sleep and wakeup.

```
int cpu_sleep_wakeup (SleepMode_TypeDef sleep_mode,  
                      SleepWakeupSrc_TypeDef wakeup_src,  
                      unsigned int wakeup_tick);
```

- ✧ “sleep_mode”: This para serves to set sleep mode as suspend mode, deepsleep mode, deepsleep retention 16K Sram or 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;
```

- ✧ wakeup_src: This para serves to set wakeup source for suspend/deep retention/deepsleep as one or combination of [PM_WAKEUP_PAD](#) and [PM_WAKEUP_TIMER](#).

If set as 0, MCU wakeup is disabled for sleep mode.

- ✧ “wakeup_tick”:

If [PM_WAKEUP_TIMER](#) is assigned as wakeup source, the “wakeup_tick” serves to set MCU wakeup time.

If [PM_WAKEUP_TIMER](#) is not assigned, this para is negligible.

The “wakeup_tick” is an absolute value, which equals current value of System Timer tick plus intended sleep duration. When System Timer tick reaches the time defined by the wakeup_tick, MCU wakes up from sleep mode.

Without taking current System Timer tick value as reference point, wakeup time is uncontrollable.

Since the wakeup_tick is an absolute time, it follows the max range limit of 32bit System Timer tick. In current SDK, 32bit max sleep time corresponds to 7/8 of max System Timer tick. Since max System Timer tick is 268s or so, max sleep time is $268 * 7/8 = 234s$, which means the “delta_Tick” below should not exceed 234s.

```
cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER,
```

```
clock_time() + delta_tick);
```

- ✧ Return value: The return value is an ensemble of current wakeup sources. Following shows wakeup source for each bit of the return value.

```
enum {
    WAKEUP_STATUS_TIMER = BIT(1),
    WAKEUP_STATUS_PAD   = BIT(3),

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

- 1) If `WAKEUP_STATUS_TIMER` bit = 1, wakeup source is Timer.
- 2) If `WAKEUP_STATUS_PAD` bit = 1, wakeup source is GPIO PAD.
- 3) If both `WAKEUP_STATUS_TIMER` and `WAKEUP_STATUS_PAD` equal 1, wakeup source is Timer and GPIO PAD.
- 4) `STATUS_GPIO_ERR_NO_ENTER_PM` is a special state indicating GPIO wakeup error. E.g. Suppose a GPIO is set as high level PAD wakeup (PM_WAKEUP_PAD). When MCU attempts to invoke the “cpu_sleep_wakeup” to enter suspend, if this GPIO is already at high level, MCU will fail to enter suspend and immediately exit the “cpu_sleep_wakeup” with return value `STATUS_GPIO_ERR_NO_ENTER_PM`.

Sleep time is typically set in this way:

```
cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_TIMER,
                  clock_time() + delta_Tick);
```

The “delta_Tick”, a relative time (e.g. $100 * \text{CLOCK_16M_SYS_TIMER_CLK_1MS}$), plus “clock_time()” becomes an absolute time.

Some examples on cpu_sleep_wakeup:

- 1) `cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_PAD, 0);`

When it's invoked, MCU enters suspend, and wakeup source is GPIO PAD.

- 2) `cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_TIMER, clock_time() + 10 * CLOCK_16M_SYS_TIMER_CLK_1MS);`

When it's invoked, MCU enters suspend, wakeup source is timer, and wakeup time is current time plus 10ms, so the suspend duration is 10ms.

- 3) `cpu_sleep_wakeup (SUSPEND_MODE, PM_WAKEUP_PAD | PM_WAKEUP_TIMER,
clock_time() + 50* CLOCK_16M_SYS_TIMER_CLK_1MS);`

When it's invoked, MCU enters suspend, wakeup source includes timer and GPIO PAD, and timer wakeup time is current time plus 50ms.

If GPIO wakeup is triggered before 50ms expires, MCU will be woke up by GPIO PAD in advance; otherwise, MCU will be woke up by timer.

- 4) `cpu_sleep_wakeup (DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);`

When it's invoked, MCU enters deepsleep, and wakeup source is GPIO PAD.

- 5) `cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K, PM_WAKEUP_TIMER,
clock_time() + 8* CLOCK_16M_SYS_TIMER_CLK_1S);`

When it's invoked, MCU enters deepsleep retention 16K Sram mode, wakeup source is timer, and wakeup time is current time plus 8s.

- 6) `cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW32K, PM_WAKEUP_PAD |
PM_WAKEUP_TIMER, clock_time() + 10* CLOCK_16M_SYS_TIMER_CLK_1S);`

When it's invoked, MCU enters deepsleep retention 32K Sram mode, wakeup source includes GPIO PAD and Timer, and timer wakeup time is current time plus 10s. If GPIO wakeup is triggered before 10s expires, MCU will be woke up by GPIO PAD in advance; otherwise, MCU will be woke up by timer.

4.1.4 Low power wakeup procedure

After the API “cpu_sleep_wakeup” is invoked, MCU enters sleep mode.

After wakeup by certain source, MCU operation varies with different sleep modes, as shown below.

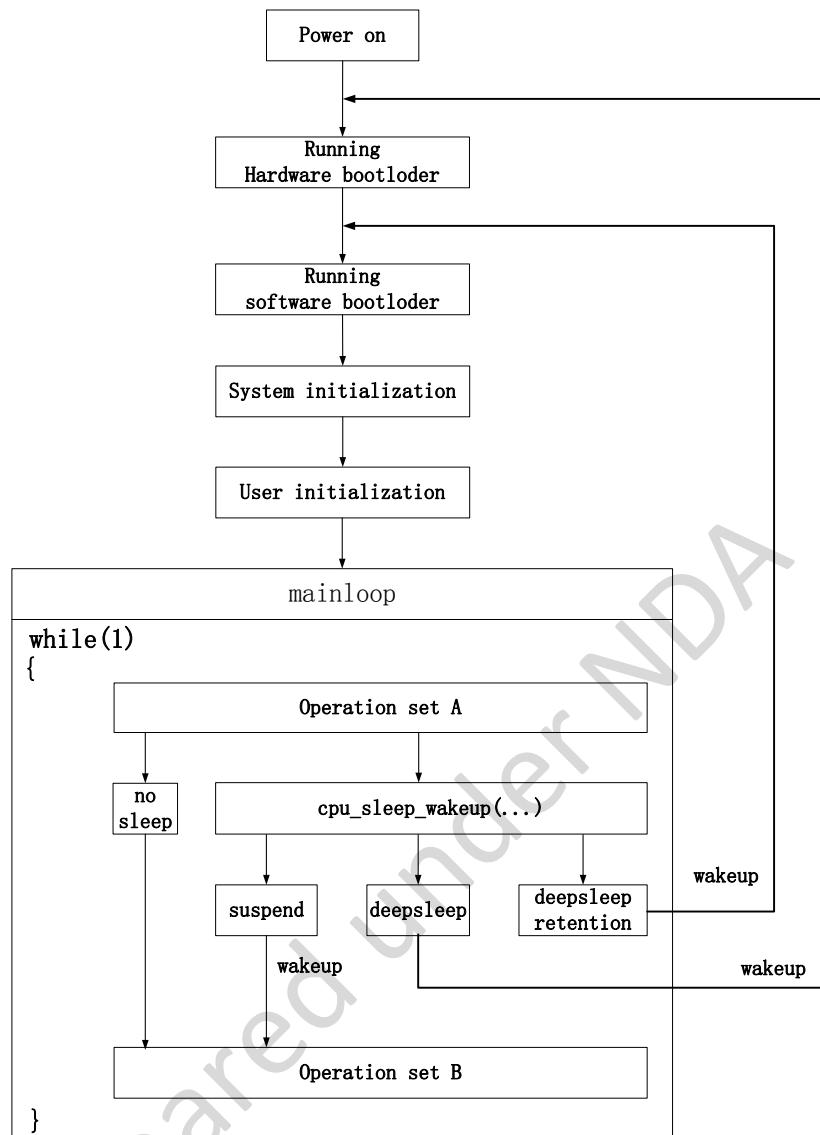


Figure 4-2 Sleep mode wakeup work flow

1) Running hardware bootloader

It is pure MCU hardware operation without involvement of software.

Couple of examples:

Read flash boot flag to determine whether current firmware is stored at flash address 0 or 0x20000 (see OTA);

Read corresponding flash address to determine the size of data to be copied from flash to SRAM as memory resident.

Due to data copying from flash to SRAM, it takes long time to execute “Running hardware bootloader” operation, for example, 10K data copying may take 5ms or so.

2) Running software bootloader

After hardware bootloader, MCU starts “Running software bootloader”. Software bootloader is vector side corresponding to assembly in the “cstartup_8258_16K_RET.S”.

Software bootloader serves to set up memory environment for C program execution, so it can be regarded as memory initialization.

3) System initialization

System initialization, hardware initialization in main function, starts from `cpu_wakeup_init` and ends before `user_init`. It includes `cpu_wakeup_init`, `rf_drv_init`, `gpio_init` and `clock_init`, and it serves to set digital and analog registers for each hardware module.

4) User initialization

User initialization corresponds to `user_init`, or `user_init_normal/ user_init_deepRetn` as explained in **4.2.8.3**.

5) main_loop

After User initialization, program enters `main_loop` inside `while(1)`.

The operation is called “Operation Set A” before `main_loop` enters sleep mode, and called “Operation Set B” after wakeup from sleep.

MCU flow for different sleep modes:

1) no sleep

Without sleep mode, MCU keeps looping inside while(1) between “Operation Set A” -> “Operation Set B”.

2) suspend

MCU enters suspend mode by invoking `cpu_sleep_wakeup`, wakes up from suspend after exiting from `cpu_sleep_wakeup`, and then executes “Operation Set B”.

Suspend can be regarded as the most “clean” sleep mode, in which data of SRAM, digital and analog registers (only a very few exceptions) are retained. After wakeup from suspend, program continues from the breakpoint, with almost no need to recover SRAM or registers.

However, in suspend current is relatively high.

3) deepsleep

MCU can also enter deepsleep by invoking `cpu_sleep_wakeup`.

After wakeup from deepsleep, MCU restarts from “Running hardware bootloader”.

Almost the same as power on reset, all hardware and software initialization are required after deepsleep wakeup.

Since SRAM and registers - except a few retention analog registers - will lose their data in deepsleep, MCU current is decreased to less than 1uA.

4) deepsleep retention

MCU can also enter deepsleep retention mode by invoking `cpu_sleep_wakeup`.

After wakeup from deepsleep retention, MCU restarts from “Running software bootloader”.

Deepsleep retention is an intermediate sleep state between suspend and deepsleep with following advantages:

- ❖ In suspend mode, both SRAM and most registers need to retain data, which thus ends up with higher current.

In deepsleep retention, it's only needed to maintain states of a few retention analog registers, as well as data of first 16K or 32K SRAM, so current is largely decreased to 2uA or so.

- ✧ After deepsleep wake_up, MCU needs to restart the whole flow.

Since first 16K or 32K SRAM are non-volatile in deepsleep retention, there's no need to re-load from flash to SRAM after wake_up, and thus "Running hardware bootloader" is skipped.

Due to limited SRAM retention size, "Running software bootloader" cannot be skipped.

Since deepsleep retention does not keep register state, system initialization must also be executed to re-initialize registers.

User initialization after deepsleep retention wake_up can actually be optimized to differentiate from MCU power on and deepsleep wake_up (see **4.2.8.3**).

4.1.5 API pm_is_MCU_deepRetentionWakeup

According to the "sleep mode wakeup work flow" above, MCU power on, deepsleep wake_up and deepsleep retention wake_up all need to go through "Running software bootloader", "system initialization", and "user initialization".

While running system initialization and user initialization, user needs to know whether MCU is woke up from deepsleep retention, so as to differentiate from power on and deepsleep wake_up. The following API in the PM driver serves to make this judgement.

```
int pm_is_MCU_deepRetentionWakeup(void);
```

- ✧ Return value: 1- deepsleep retention wake_up; 0- power on or deepsleep wake_up.

4.2 BLE low power management

4.2.1 BLE PM Initialization

For applications with low power mode, BLE PM module needs to be initialized by following API.

If low power is not required, DO NOT use this API, so as to skip compiling of related code and variables into program and thus save FW and SRAM space.,

```
void blc_ll_initPowerManagement_module(void);
```

4.2.2 BLE PM for Link Layer

In this BLE SDK, PM module manages power consumption in BLE Link Layer. It would be helpful referring to introduction to Link Layer in earlier chapter.

Current SDK only applies low power management to Advertising state and Connection state Slave role with a set of APIs for user. It's not applicable yet to Scanning state, Initiating state and Connection state Master role.

The SDK does not apply low power management to Idle state either. In Idle state, since there is no RF activity, i.e. the “blt_sdk_main_loop” function is not valid, user can use PM driver for certain low power management. E.g. In the demo code below, when Link Layer is in Idle state, every main_loop would suspend for 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
    }
}
```

The figure below shows timing of sleep mode when Link Layer is in Advertising state or Conn state Slave role with connection latency = 0.

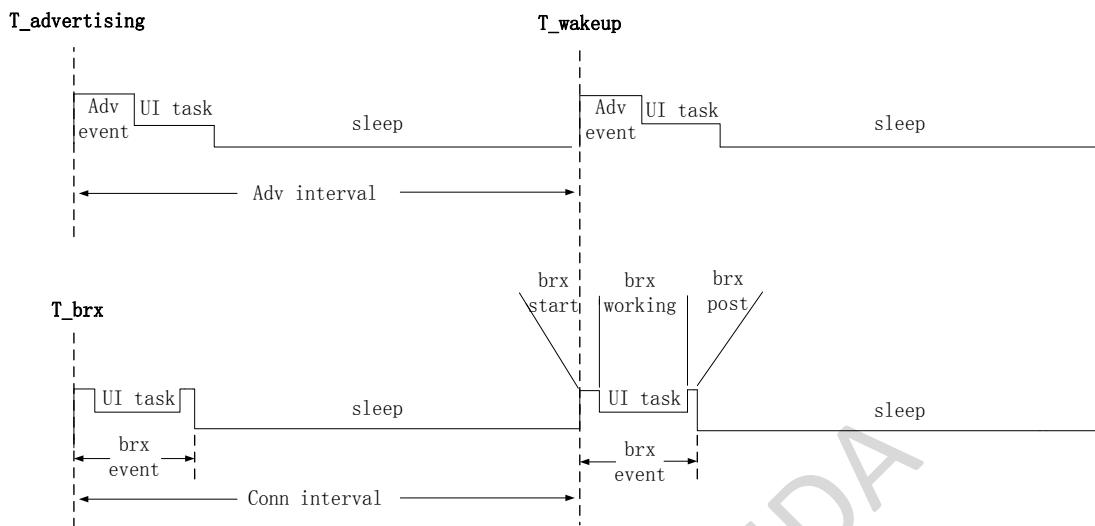


Figure 4-3 Sleep timing for Advertising state & Conn state Slave role

- 1) In Advertising state, during each Adv Internal, Adv Event is mandatory; MCU can enter sleep mode (suspend/deepsleep retention) during the rest time other than UI task.

The starting time of Adv event at first Adv interval is defined as $T_{advertising}$, and the time for MCU to wake up from sleep is defined as T_{wakeup} .

T_{wakeup} is also the start of Adv event at next Adv interval. Both these two parameters will be elaborated in later section.

- 2) During each Conn-interval at Conn state Slave role, the time for brx Event (brx start+brx working+brx post) is mandatory. MCU can enter sleep mode (suspend/deepsleep retention) during the rest time other than UI task.

The starting time of of Brx event at first Connection interval is defined as T_{brx} , and the time for MCU to wake up from sleep is T_{wakeup} .

T_{wakeup} is also the start of BRx event at next Connection interval. Both these two parameters will be elaborated in later section.

BLE PM is basically the sleep mode management in Advertising state or Conn state Slave role. User can select sleep mode and set related time parameters.

As explained earlier, the 8x5x family has 3 sleep modes: suspend, deepsleep, and deepsleep retention.

For suspend and deepsleep retention, since the blt_sdk_main_loop of the SDK includes low PM in BLE stack according to Link Layer state, to configure low power management, user only needs to invoke corresponding APIs instead of the “cpu_sleep_wakeup”.

Deepsleep is not included in BLE low PM, so user needs to manually invoke the API “cpu_sleep_wakeup” in APP layer to enter deepsleep. Please refer to the “blt_pm_proc” function in the project “8258_ble_remote” of the SDK.

Following sections illustrate details of low power management in Advertising state and Connection state Slave role.

4.2.3 BLE PM variables

The variables in this section are helpful to understand BLE PM software flow .

The definition of the struct “st_ll_pm_t” is available in the file “ll_pm.h”. Following lists some variables of the struct which will be used by PM APIs.

```
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;
```

Following struct is defined in the file “ll_pm.c” for understanding purpose. Please note that this file is assembled in library, and user is not allowed to make any operation on this struct variable.

```
st_ll_pm_t  bltPm;
```

There will be a lot variables like the “`bltPm. suspend_mask`” in later sections.

4.2.4 API bls_pm_setSuspendMask

The APIs below serve to configure low power management in Link Layer at “Advertising state” and “Conn state Slave role”.

```
void      bls_pm_setSuspendMask (u8 mask);  
u8       bls_pm_getSuspendMask (void);
```

The “bltPm.suspend_mask” is set by the “bls_pm_setSuspendMask” and its default value is SUSPEND_DISABLE.

Following shows source code of the 2 APIs.

```
void      bls_pm_setSuspendMask (u8 mask)  
{  
    bltPm.suspend_mask = mask;  
}  
  
u8  bls_pm_getSuspendMask (void)  
{  
    return bltPm.suspend_mask;  
}
```

The “bltPm.suspend_mask” can be set as any one or the “or-operation” of following values:

#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” means sleep is disabled which allows MCU to enter neither suspend nor deepsleep retention.

“SUSPEND_ADV” and “DEEPSLEEP_RETENTION_ADV” decide whether MCU at Advertising state can enter suspend and deepsleep retention.

“SUSPEND_CONN” and “DEEPSLEEP_RETENTION_CONN” decide whether MCU at Conn state Slave role can enter suspend and deepsleep retention.

In low power sleep mode design of the SDK, deepsleep retention is a substitute of suspend mode to reduce sleep power consumption.

Take Conn state slave role as an example:

The SDK will first check whether SUSPEND_CONN is enabled in the "bltPm.suspend_mask", and MCU can enter suspend only when SUSPEND_CONN is enabled. Further on, based on the value of the DEEPSLEEP_RETENTION_CONN, MCU can decide whether it will enter suspend mode or deepsleep retention mode.

Therefore, to enable MCU to enter suspend, user only needs to enable SUSPEND_ADV/SUSPEND_CONN. To enable MCU to enter deepsleep retention mode, both SUSPEND_CONN and DEEPSLEEP_RETENTION_CONN should be enabled.

Following shows 3 typical use cases:

- 1) bls_pm_setSuspendMask(SUSPEND_DISABLE);

MCU will not enter sleep mode (suspend/deepsleep retention).

- 2) bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN);

At Advertising state and Conn state Slave role, MCU can only enter suspend mode, and it's not allowed to enter deepsleep retention.

- 3) bls_pm_setSuspendMask(SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV
| SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN);

At Advertising state and Conn state Slave role, MCU can enter both suspend and deepsleep retention, but the sleep mode to enter depends on sleeping time which will be explained later.

There may be some special applications, for example:

- 1) bls_pm_setSuspendMask(SUSPEND_ADV)

Only at Advertising state can MCU enter suspend, and at Conn state Slave role it's not allowed to enter sleep mode.

- 2) bls_pm_setSuspendMask(SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN):
Only at Conn state Slave role, can MCU enter suspend or deepsleep retention, and at Advertising state it's not allowed to enter sleep mode.

4.2.5 API bls_pm_setWakeupSource

User can set the bls_pm_setSuspendMask to enable MCU to enter sleep mode (suspend or deepsleep retention), and use the following API to set wakeup source.

```
void      bls_pm_setWakeupSource (u8 source);  
"source": Wakeup source, can be set as PM\_WAKEUP\_PAD.
```

This API sets the bottom-layer variable “bltPm.wakeup_src”. Following shows source code in the SDK.

```
void      bls_pm_setWakeupSource (u8 src)  
{  
    bltPm.wakeup_src = src;  
}
```

When MCU enters sleep mode at Advertising state or Conn state Slave role, its actual wakeup source is:

```
bltPm.wakeup_src | PM\_WAKEUP\_TIMER
```

So [PM_WAKEUP_TIMER](#) is mandatory, not depending on user setup. This guarantees that MCU will wake up at specified time to handle Adv Event or Brx Event.

Everytime wakeup source is set by the “bls_pm_setWakeupSource”, after MCU wakes up from sleep mode, the [bltPm.wakeup_src](#) is set to 0.

4.2.6 API blc_pm_setDeepsleepRetentionType

Deepsleep retention further separates into 16K sram retention or 32K sram retention. When entering deepsleep retention mode, the following API can be set to decide which sub-mode to enter:

```
void blc_pm_setDeepsleepRetentionType(SleepMode_TypeDef  
sleep_type);
```

Only two options are available:

```
typedef enum {  
    DEEPSLEEP_MODE_RET_SRAM_LOW16K      = 0x43,  
    DEEPSLEEP_MODE_RET_SRAM_LOW32K      = 0x07,  
} SleepMode_TypeDef;
```

In the SDK, default deepsleep retention mode is set as [DEEPSLEEP_MODE_RET_SRAM_LOW16K](#), and to use 32K retention mode, user needs to invoke the API below during initialization. Please note that this API must be invoked after the “blc_ll_initPowerManagement_module” to take effect.

```
blc_pm_setDeepsleepRetentionType(DEEPSLEEP_MODE_RET_SRAM_LOW32K) ;
```

As introduced in **section 2 MCU Basic Modules**, SRAM space allocation uses the default design of “deepsleep retention 16K Sram”; as described in section 1, software bootloader and boot.link file should be selected according to IC type and deep retention size value (For the mapping relationship and method to modify setting, please refer to **section 1.3 Software bootloader**).

For the 8258, to use “deepsleep retention 32K Sram”, the two steps below should be followed.

- 1) Select the “software bootloader” file as “cstartup_8258_RET_32K.S”.
- 2) Modify the “boot.link” file: Use the contents of the “SDK/boot/boot_32k_retn_8253_8258.link” to replace the boot.link file under the root directory of the SDK.

For the 8251/8253, user can modify the setting as needed.

4.2.7 PM software processing flow

Both actual code and pseudo-code are used herein to explain the flow details.

4.2.7.1 blt_sdk_main_loop

As shown below, the “blt_sdk_main_loop” is repetitively executed in while (1) loop of the SDK. Accordingly BLE PM code is inside the “blt_sdk_main_loop” is also repetitively executed.

```
while(1)
{
    ///////////////// BLE entry /////////////
    blt_sdk_main_loop();

    //////////////// UI entry ///////////
    //UI task

    //////////////// user PM config ///////////
    //blt_pm_proc();
}
```

Following shows the implementation of BLE PM logic inside the “blt_sdk_main_loop”.

```
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) When the “bltPm.suspend_mask” is SUSPEND_DISABLE, the SW directly exits without executing the “blt_brx_sleep” function.
So when using the “bls_pm_setSuspendMask(SUSPEND_DISABLE)”, PM logic is completely ineffective; MCU will never enter sleep and the SW always execute while(1) loop.
- 2) When the SW is executing Adv Event at Advertising State or Brx Event at Conn state Slave role, the “blt_brx_sleep” won’t be executed either due to RF task ongoing. The SDK needs to guarantee completion of Adv Event/Brx Event before MCU enters sleep mode.
- 3) Only when both cases above are invalid, the blt_brx_sleep will be executed.

4.2.7.2 blt_brx_sleep

Following shows logic implementation of the “blt_brx_sleep” function in the case of default deepsleep retention 16K Sram.

```

void blt_brx_sleep (void)
{
    if( (Link Layer state == Adv state)&& (bltPm. suspend_mask &SUSPEND_ADV) )
    { //current state is adv state, suspend is allowed

```

```
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
        //PC reset to 0 after wakeup, restart on "software
        //bootloader" //(cstartup_8258_16K.S), system
        //initialization, etc
}
else
{
    cpu_sleep_wakeup ( SUSPEND_MODE,
                        PM_WAKEUP_TIMER | bltPm.wakeup_src,
                        T_wakeup);
}
```

```
"BLT_EV_FLAG_SUSPEND_EXIT" event callback execution
```

```
if(suspend is woke up by GPIO PAD)
{
    "BLT_EV_FLAG_GPIO_EARLY_WAKEUP" event callback execution
}
}
else if((Link Layer state == Conn state Slave role)&& (SuspendMask&SUSPEND_CONN) )
{
    //current state is Conn state, suspend enabled
    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
    //PC reset to 0 at MCU wakeup, rerun software
    bootloader //(cstartup_8258_16K.S), system
    initialization etc
}
else
{
    cpu_sleep_wakeup ( SUSPEND_MODE,
                       PM_WAKEUP_TIMER | bltPm.wakeup_src,
                       T_wakeup);
}

" BLT_EV_FLAG_SUSPEND_EXIT" event callback execution

if(suspend is woke up by GPIO PAD)
{
    "BLT_EV_FLAG_GPIO_EARLY_WAKEUP" event callback execution
    Adjust BLE timing
}
}

bltPm.wakeup_src = 0;
bltPm.user_latency = 0xFFFF;
}
```

To simplify the discussion, let's begin with an easy case: conn_latency =0, only suspend mode, no deepsleep retention.

The PM is the same as the 826x family when setting suspend mask in APP layer via the “bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN)”.

Referring to controller event introduced in **3.2.7**, please pay close attention to the timing of these suspend related events and callback functions:
BLT_EV_FLAG_SUSPEND_ENTER, BLT_EV_FLAG_SUSPEND_EXIT,

BLT_EV_FLAG_GPIO_EARLY_WAKEUP。

When Link Layer is in Advertising state with “bltPm. `suspend_mask`” set to SUSPEND_ADV, or at Conn state slave role with “bltPm. `suspend_mask`” set to SUSPEND_CONN, MCU can enter suspend mode.

In suspend mode, the API “cpu_sleep_wakeup” in the driver is finally invoked.

```
cpu_sleep_wakeup ( SUSPEND_MODE,  
                    PM_WAKEUP_TIMER | bltPm.wakeup_src,  
                    T_wakeup) ;
```

This API sets wakeup source as `PM_WAKEUP_TIMER` | `bltPm.wakeup_src`, so Timer wakeup is mandatory to guarantee MCU wakeup before next Adv Event or Brx Event. For wakeup time “`T_wakeup`”, please refer to earlier “sleep timing for Advertising state & Conn state Slave role” diagram.

When exiting the “`blt_brx_sleep`” function, both the “`bltPm.wakeup_src`” and the “`bltPm.user_latency`” are reset. So the API “`bls_pm_setWakeupSource`” and “`bls_pm_setManualLatency`” are only effective for current sleep mode.

4.2.8 Analysis of deepsleep retention

User can invoke the API below in APP layer to enable Deepsleep retention mode.

```
bls_pm_setSuspendMask( SUSPEND_ADV | DEEPSLEEP_RETENTION_ADV  
                      | SUSPEND_CONN | DEEPSLEEP_RETENTION_CONN);
```

4.2.8.1 API `blc_pm_setDeepsleepRetentionThreshold`

At Advertising state and Conn state slave role, suspend can switch to deep retention only when following conditions are met, respectively:

```
if( bltPm. suspend_mask & DEEPSLEEP_RETENTION_ADV &&  
    T_sleep > bltPm. deepRet_advThresTick )  
  
if( bltPm. suspend_mask & DEEPSLEEP_RETENTION_CONN &&  
    T_sleep > bltPm. deepRet_connThresTick )
```

Firstly, the “`bltPm. suspend_mask`” should be set to DEEPSLEEP_RETENTION_ADV or DEEPSLEEP_RETENTION_CONN, as explained before.

Secondly, for

```
T_sleep > bltPm. deepRet_advThresTick or
```

T_sleep > bltPm.deepRet_connThresTick ,

T_sleep, sleep duration time, equals Wakeup time “T_wakeup” minus current time “clock_time()”. It means that sleep duration should exceed certain threshold so that MCU can switch sleep mode from suspend to deepsleep retention.

Here is the API to set the two threshold in unit of ms for Advertising state and Conn state slave role.

```
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;
}
```

The threshold by the API “blc_pm_setDeepsleepRetentionThreshold” is designed to optimize power consumption.

Deepsleep retention does not necessarily mean lower power consumption than suspend mode all the time, since deepsleep retention needs longer wakeup time than suspend mode.

- ❖ Suspend mode can immediately start on Adv Event or Brx Event at T_wakeup;
- ❖ For deepsleep retention, however, it's needed to go through 3 extra steps of running software bootloader + system initialization + user initialization.
- ❖ By instinct, deepsleep retention wins on power consumption only when sleep duration is long enough, or exceeds certain threshold.

Using Conn state slave role as an example, following diagram illustrates the different timing and power between suspend mode and deepsleep retention mode.

T_cycle is the cycle time between two adjacent Brx events. Assuming equivalent average current at Brx Event is I_brx lasting for t_brx (Note t_brx is different from T_brx). Also assuming equivalent average current is I_suspend for suspend mode, and I_deepRet for deep retention.

Equivalent average current over “running software bootloader + system initialization + user initialization” is I_init lasting for T_init. In practice, user may need to measure T_init, the implementation of which is to be explained later.

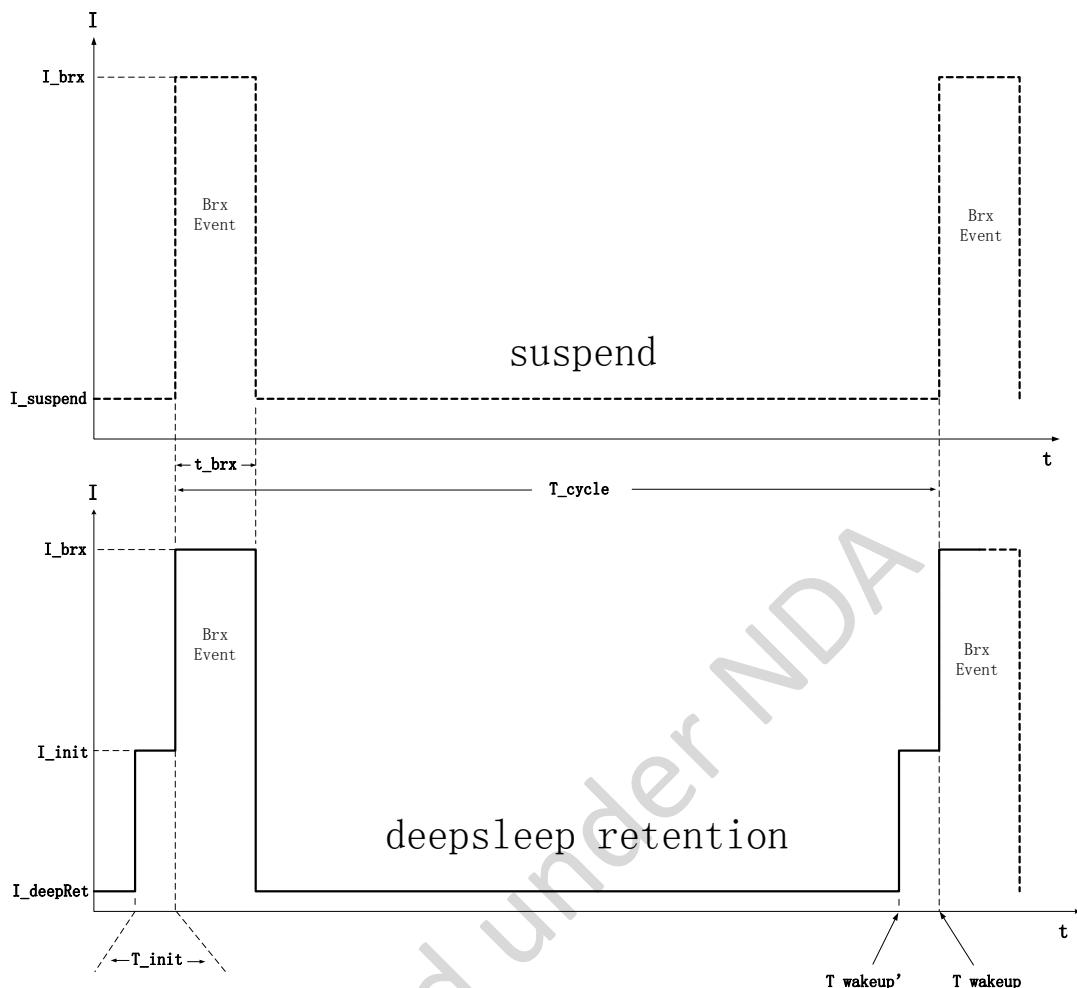


Figure 4-4 suspend & deepsleep retention timing & power

Average Brx current with suspend mode is:

$$I_{\text{avgSuspend}} = I_{\text{brx}} * t_{\text{brx}} + I_{\text{suspend}} * (T_{\text{cycle}} - t_{\text{brx}})$$

Simplified by $T_{\text{cycle}} \gg t_{\text{brx}}$, $(T_{\text{cycle}} - t_{\text{brx}})$ can be regarded as T_{cycle} .

$$I_{\text{avgSuspend}} = I_{\text{brx}} * t_{\text{brx}} + I_{\text{suspend}} * T_{\text{cycle}}$$

Average Brx current with deepsleep retention mode is:

$$\begin{aligned} I_{\text{avgDeepRet}} &= I_{\text{brx}} * t_{\text{brx}} + I_{\text{init}} * T_{\text{init}} + I_{\text{deepRet}} * (T_{\text{cycle}} - t_{\text{brx}}) \\ &= I_{\text{brx}} * t_{\text{brx}} + I_{\text{init}} * T_{\text{init}} + I_{\text{deepRet}} * T_{\text{cycle}} \end{aligned}$$

Calculate the delta between I_avgSuspend and I_avgDeepRet:

$$I_{avgSuspend} - I_{avgDeepRet}$$

$$\begin{aligned} &= I_{suspend} * T_{cycle} - I_{init} * T_{init} - I_{deepRet} * T_{cycle} \\ &= T_{cycle} (I_{suspend} - I_{deepRet}) - (T_{init} * I_{init}) / T_{cycle} \end{aligned}$$

For application program with correct power debugging on both HW circuit and SW, the “(I_suspend - I_deepRet)” and “(T_init*I_init)” can be regarded as fixed value.

Suppose I_suspend=30uA, I_deepRet=2uA, (I_suspend - I_deepRet) = 28uA; I_init=3mA, T_init=400 uS, (T_init*I_init)=1200 uA*uS:

$$\begin{aligned} I_{avgSuspend} - I_{avgDeepRet} \\ = T_{cycle} (28 - 1200/T_{cycle}) \end{aligned}$$

$$I_{avgSuspend} - I_{avgDeepRet}$$

>0 when Tcycle > (1200/28) = 43ms; DeepRet consumes less power

<0 when Tcycle < 43ms; Suspend mode consumes less power

Mathematically, when Tcycle < 43 ms, suspend mode is more power efficient; when Tcycle > 43 ms, deepsleep retention mode is a better choice.

By using the threshold setting API below, MCU will automatically switch suspend to deepsleep retention for T_sleep more than 43mS, and maintain suspend for T_sleep less than 43mS.

```
blc_pm_setDeepsleepRetentionThreshold(43, 43);
```

Take a long connection of 10ms connection interval * (99 + 1) = 1S as an example:

At Conn state slave role, even though user may choose different suspend duration such as 10ms, 20ms, 50ms, 100ms, or 1s due to UI task, latency etc, this threshold API will ensure optimum power consumption by auto switching between suspend mode and deepsleep retention mode.

It is reasonable to assume MCU working time (Brx Event + UI task) is short enough therefore when T_cycle is long, T_sleep approximately equals to T_cycle.

Fundamentally, user may need to measure and come up with more accurate current and timing values for a correct threshold value.

In practice, following demos in the SDK, as long as user initialization does not incorrectly run across extended time, for T_cycle larger than 100ms, deepsleep retention mode should end up with lower power in most application scenarios.

4.2.8.2 blc_pm_setDeepsleepRetentionEarlyWakeupTiming

According to the “suspend & deepsleep retention timing & power”, suspend wake_up time “T_wakeup” is exactly the starting point of next Brx Event, or the time point when BLE master starts sending packet.

For deepsleep retention, wake_up time needs to start earlier than T_wakeup to allow T_init: running software bootloader + system initialization + user initialization, or it will miss Brx Event, i.e., the time when BLE master starts sending packet. So MCU wake_up time should be pulled in to T_wakeup':

$$T_{wakeup}' = T_{wakeup} - T_{init}$$

When applying to:

```
cpu_sleep_wakeup (DEEPSLEEP_MODE_RET_SRAM_LOW16K,  
                    PM_WAKEUP_TIMER | bltPm.wakeup_src,  
                    T_wakeup - bltPm.deepRet_earlyWakeupTick);
```

T_wakeup is automatically calculated by the BLE stack, while the “bltPm.deepRet_earlyWakeupTick” can be assigned to the measured T_init (or slightly larger) by following API:

```
void blc_pm_setDeepsleepRetentionEarlyWakeupTiming (u32  
earlyWakeup_us)  
{  
    bltPm.deepRet_earlyWakeupTick = earlyWakeup_us *  
        CLOCK_16M_SYS_TIMER_CLK_1US;  
}
```

4.2.8.3 Optimization and measurement of T_init

For SRAM concept to be discussed in this section such as ram_code, retention_data, deepsleep retention area, please refer to **section 2.1.2 SRAM space partition**.

1) T_init timing

According to above analysis, when T_cycle is long enough, deepsleep retention has more power advantage than suspend, in return, T_init becomes mandatory. With I_init being mostly a constant value, minimizing T_init would reduce the power consuption.

T_init is the time sum of running software bootloader + system initialization + user initialization.

- ✧ T_cstartup is the time of running software bootloader, i.e. executing assembly file cstartup_xxx.S.
- ✧ T_sysInit is system initialization time.
- ✧ T_userInit is user initialization time.

$$T_{init} = T_{cstartup} + T_{sysInit} + T_{userInit}$$

Following is a complete timing diagram of T_init:

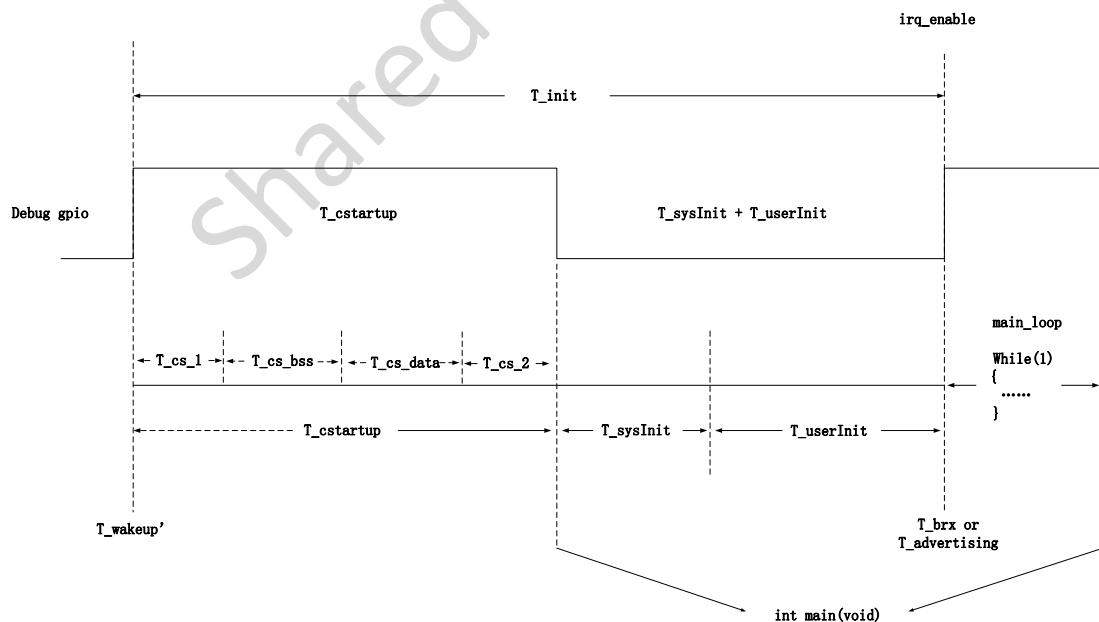


Figure 4-5 T_init timing

Based on earlier definition, T_wakeup is the starting point of next Adv Event/Brx Event, and T_wakeup' is MCU early wake_uptime.

After wake_up, MCU will execute cstsetup_xxx.S, jump to main() to start system initialization followed by user initialization, and then enter main_loop.

Once getting in main_loop, it can start processing of Adv Event/Brx Event. The end of T_userInit is the starting point of Adv Event/Brx Event, or T_brx/T_advertising as shown in above diagram. “irq_enable” in the diagram is the separation between T_userInit and main_loop, matching the code in the SDK.

In the SDK, T_sysInit includes execution time of cpu_wakeup_init, rf_drv_init, gpio_init and clock_init. These timing parameters have been optimized in the SDK by placing the associated code into the ram_code. This is the reason this BLE SDK occupies more SRAM space than the 826x family. User does not need to do any further optimization on T_sysInit.

T_cstsetup and T_userInit are elaborated herein.

2) T_userInit

User initialization is executed at power on, deepsleep wake_up, and deepsleep retention wake_up.

For applications without deepsleep retention mode, user initialization does not need to differentiate between deepsleep retention wake_up and power on/deepsleep wake_up. For this case, same as the 826x family, the “8258_master_kma_dongle” project of this BLE SDK also uses the API below to implement user initialization.

```
void user_init(void);
```

For applications with deepsleep retention mode, to reduce power, T_userInit needs to be as short as possible as explained earlier, so deepsleep retention wake_up would be different from power on / deepsleep wakeup.

Initialization tasks in the user_init falls into 2 categories: initialization of hardware registers, and initialization of logic variables in SRAM.

Since in deepsleep retention mode first 16K or 32K SRAM is non-volatile, logic variables can be defined as retention_data to save time for initialization. Since registers cannot retain data across deepsleep retention, re-initialization is required for registers.

In summary, for deepsleep retention wake_up, user_init_deepRetn applies; while for power on and deepsleep wake_up, user_init_normal function applies, as shown in following coding:

```
int deepRetWakeUp = pm_is MCU_deepRetentionWakeup();  
  
if( deepRetWakeUp ) {  
    user_init_deepRetn ();  
}  
else {  
    user_init_normal ();  
}
```

The “user_init_deepRetn” in the SDK demo “8258_ble_remote” is:

```
_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  
}
```

First 3 lines (from blc_app_loadCustomizedParameters() to rf_set_power_level_index) are mandatory BLE initialization of hardware registers.

The blc_ll_recoverDeepRetention() is to recover software and hardware state at Link Layer by low level stack.

User is not recommended to modify these 4 lines.

The app_ui_init_deepRetn() is re-initialization of hardware registers at APP layer, such as GPIO wakeup configuration and LED state setting in the demo “8258_ble_remote”, UART hardware register state in the demo “8258_module”.

On top of SDK demo, if additional items are added to user initialization, following judgement is recommended:

- (1) If it is SRAM variable, put it to the “retention_data” section by adding the keyword “_attribute_data_retention_”, so as to save re-initialization time after deepsleep retention wake_up. Then it can be run at user_init_normal function.
- (2) If it is hardware register, it should be placed inside user_init_deepRetn function.

With above implementation, after deepsleep retention wake_up, T_userInit is execution time of user_init_deepRetn. The SDK also tries to place these functions inside ram_code to save time. If deepsleep retention area allows, user should place added register initialization functions inside ram_code as well.

3) T_userInit Optimization for Conn state slave role

TBD

4) T_cstartup

T_cstartup is the execution time of cstartup_xxx.S, e.g. cstartup_8258_RET_16K.S in the SDK.

T_cstartup has 4 components, in time sequence:

$$T_{cstartup} = T_{cs_1} + T_{cs_bss} + T_{cs_data} + T_{cs_2}$$

T_{cs_1} and T_{cs_2} are fixed timing which user is not allowed to modify.

T_{cs_data} is initialization of "data" sector in SRAM. "data" is already initialized global variables with initial values stored in "data initial value" sector of flash. Therefore, T_{cs_data} is the time transferring "data" from flash "data initial value" sector to SRAM "data" sector. Corresponding assembly code is:

```
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:
```

Data transferring from flash is slow. As a reference, 16 bytes would take 7us. So more data are in "data" sector, the longer T_{cs_data} and T_{init} would be, or vice versa.

User can use method explained earlier to check size of "data" sector in list file.

If "data" sector is too big and there is enough space in deepsleep retention area, user can add the keyword "_attribute_data_retention_" to place some of the variables in "data" sector into "retention_data" sector, so as to reduce T_{cs_data} and T_{init} .

T_{cs_bss} is time to initialize SRAM "bss" sector. Intial values of "bss" sector are

all 0s. It's only need to reset SRAM “bss” sector to 0, and no flash transferring is needed. Corresponding assembly code is:

```
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:
```

Resetting each word (4 byte) to 0 can be very fast. So when “bss” is small, T_cs_bss is very small. But if “bss” sector is large, for example when a huge global data array is defined (int AAA[2000] = {0}), T_cs_bss can also be very long. So it is worth paying attention to “bss” size in list file.

To optimize T_cs_bss when “bss” sector is large, if retention area allows, some of them can also be defined as “_attribute_data_retention_” to place in “retention_data” sector.

5) T_init measurement

After T_cstartup and T_userInit are optimized to minimize T_init, it's also needed to measure T_init, and apply to API:

```
blc_pm_setDeepsleepRetentionEarlyWakeupTiming
```

T_init starts at the timing as T_cstartup, which is the “_reset” point in cstartup_8258_RET_16K.S file as shown below:

```
_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
```

As shown in above code, changing “#if 0” to “#if 1”, Debug GPIO PB4 will output high at “_reset”.

T_cstartup finishes at “tj main”:

```
tjl main
END:   tj END
```

Since main function starts almost at the end of T_cstartup, PB4 can be set to output low at beginning of main function as shown below. Please note that `DBG_CHNO_LOW` requires enabling “`DEBUG_GPIO_ENABLE`” in `app_config.h`.

```
_attribute_ram_code_ int main (void) //must run in ramcode
{
    DBG_CHNO_LOW; //debug
    cpu_wakeup_init();
    .....
}
```

By scoping signal of PB4, T_cstartup is obtained.

Adding PB4 output high at end of T_userInit inside `user_init_deepRetn` will generate same timing diagram as Debug gpio as shown above.

T_init and T_cstartup can be measured by oscilloscope or logic analyzer. Following understanding of GPIO operation, user can modify the Debug gpio code as needed, so as to get other timing parameters as well, e.g. T_sysInit, T_userInit etc.

4.2.9 Connection Latency

4.2.9.1 Sleep timing with non-zero connection latency

All sleep mode discussion so far on Conn state slave role assumes connection latency (or `conn_latency`) equals zero.

In software, `T_wakeup = T_brx + conn_interval`, as shown in below coding:

```
if(conn_latency != 0)
{
    latency_use = bls_calculateLatency();
    T_wakeup = T_brx + (latency_use +1) * conn_interval;
}
```

```

else
{
    T_wakeup = T_brx + conn_interval;
}
    
```

Once conn_latency is non-zero, after connection parameters update, sleep wake_up time becomes:

$$T_{wakeup} = T_{brx} + (\text{latency_use} + 1) * \text{conn_interval};$$

Following diagram illustrates sleep timing with non-zero conn_latency when latency_use= 2.

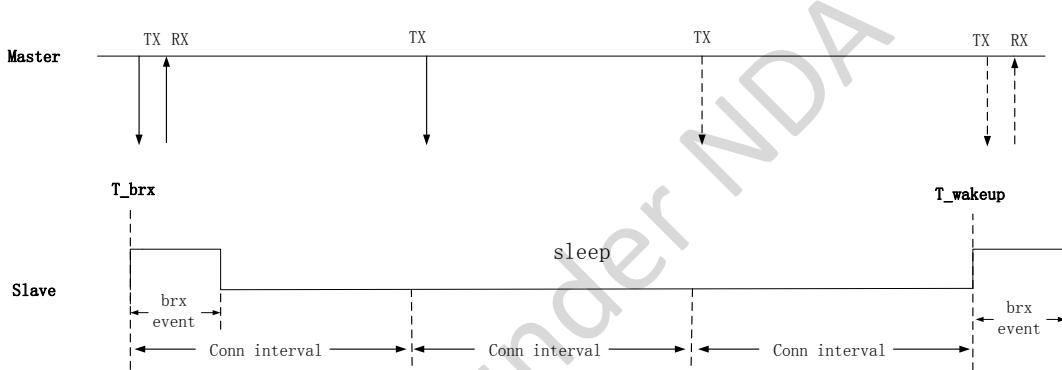


Figure 4-6 sleep timing for valid conn_latency

Before conn_latency becomes effective, sleep time is no more than 1 connection interval which is typically short.

After conn_latency becomes effective, sleep time may become as long as 1s, 2s, etc with reduced power consumption. It is also making sense to use min 43ms as a threshold point for deepsleep retention and suspend switch as explained before.

4.2.9.2 latency_use calculation

At effective conn_latency, T_{wakeup} is determined by latency_use, so it is not necessarily equal to conn_latency.

$$\text{latency_use} = \text{bls_calculateLatency}();$$

"latency_use" depends on 2 parameters: user_latency and system latency.

First one is user_latency, and its value can be set by following API:

```
void bls_pm_setManualLatency(u16 latency)
{
    bltPm.user_latency = latency;
}
```

Initial value of bltPm.user_latency is 0xFFFF, and at the end of blt_brx_sleep function it will be reset to 0xFFFF, which means the value set by the API bls_pm_setManualLatency is only valid for latest sleep, so it needs to be set on every sleep event.

Second one is system latency:

- 1) If connection latency=0, system latency=0
- 2) If connection latency > 0:
 - A. If system task is not done in current connection interval, MCU needs to wake up on next connection interval to continue the task such as transfer packet not completely sent out, or handle data from master not fully processed yet, and under this scenario, system latency=0.
 - B. If no task is left over, system latency = connection latency.

However, if slave receives master's update map request or update connection parameter request, and its updated timing is before (connection latency+1)*interval, then the actual system latency would force MCU to wake up before the updated timing point to ensure correct BLE timing sequence.

Combining user_latency and system_latency:

$$\text{latency_use} = \min(\text{system latency}, \text{user_latency})$$

Accordingly, if user_latency set by the API bls_pm_setManualLatency is less than system latency, user_latency would be the final latency_use; otherwise, system latency is the final latency_use.

4.2.10 API bls_pm_getSystemWakeupTick

Following API is used to obtain wakeup time out of suspend (System Timer tick), or T_wakeup:

```
u32 bls_pm_getSystemWakeupTick(void);
```

According to blt_brx_sleep explanation in **4.2.7.2**, T_wakeup is calculated fairly late, almost next to cpu_sleep_wakeup. Application layer can only get an accurate T_wakeup by BLT_EV_FLAG_SUSPEND_ENTER event callback function.

Following keyscan example explains usage of BLT_EV_FLAG_SUSPEND_ENTER event callback function and bls_pm_getSystemWakeupTick.

```
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);
    }
}
```

Above callback function is meant to prevent loss of key press.

A normal key press lasts for a few hundred ms, or at least 100~200ms for a fast press. When Advertising state and Conn state are configured by bls_pm_setSuspendMask to enter sleep mode:

(1) Without conn_latency in effect, as long as Adv interval or conn_interval is not very long, typically less than 100ms, sleep time will not exceed Adv Interval or conn_interval, in other words, sleep time is less than 100ms or a fast key press time, loss of key press can be prevented and there is no need to enable GPIO wakeup.

(2) With conn_latency ON, for example, with conn_interval = 10ms, connec_latency = 99, sleep time may last 1s, obviously key loss may occur.

If current state is Conn state and wakeup time of suspend to be entered is more than 80ms from current time as determined by BLT_EV_FLAG_SUSPEND_ENTER callback function, key loss can be prevented by using GPIO level trigger to wake up MCU for keyscan process in case timer wakeup is too late.

4.3 Potential issues in GPIO wakeup

4.3.1 Failure to enter sleep at effective wakeup level

In the 8x5x, GPIO wakeup is level triggered instead of edge triggered, so when GPIO PAD is configured as wakeup source, for example, suspend wakeup triggered by GPIO high level, MCU needs to make sure when MCU invokes `cpu_wakeup_suspend` to enter suspend, that the wakeup GPIO is not at high level. Otherwise, once entering `cpu_wakeup_suspend`, it would exit immediately and fail to enter suspend.

To ensure the software flow, DO avoid this problem when using GPIO PAD wakeup.

If the APP layer does not avoid this problem, and GPIO PAD wakeup source is already effective at invoking of `cpu_wakeup_sleep`, PM driver makes some improvement to avoid flow mess:

1) Suspend & deepsleep retention mode

For both suspend and deepsleep retention mode, the SW will fast exit `cpu_wakeup_sleep` with two potential return values:

- ✧ Return `WAKEUP_STATUS_PAD` if the PM module has detected effective GPIO PAD state.
- ✧ Return `STATUS_GPIO_ERR_NO_ENTER_PM` if the PM module has not detected effective GPIO PAD state.

2) deepsleep mode

For deepsleep mode, PM diver will reset MCU automatically in bottom layer (equivalent to watchdog reset). The SW restarts from “Run hardware bootloader”.

To prevent this problem, following is implemented in the SDK demo “8258 ble remote”.

In `BLT_EV_FLAG_SUSPEND_ENTER`, it is configured that only when suspend time is larger than a certain value, can GPIO PAD wakeup be enabled.

```
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);
    }
}
```

When key is pressed, manually set latency to 0 or a small value (as shown in below code), so as to ensure short sleep time, e.g. shorter than 80ms as set in above code. Therefore, the high level on drive pin due to a pressed key will never become a high-level GPIO PAD wakeup trigger.

```
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
}
}
```

There are 2 scenarios that will make MCU enter deepsleep.

- ✧ First one is if there's no event for 60s, MCU will enter deepsleep.
- ✧ The other scenario is if a key is stuck for more than 60s, MCU will enter deepsleep. Under the second scenario, the SDK will invert polarity from high level trigger to low level trigger to solve the problem.

4.4 BLE System Low Power Management

Based upon understanding of PM principle of this BLE SDK, user can configure PM under different application scenarios, referring to the demo “8258 ble remote” low power management code as explained below.

Function blt_pm_proc is added in PM configuration of main_loop. This function must be placed at the end of main_loop to ensure it is immediate to blt_sdk_main_loop in time, since blt_pm_proc needs to configure low power management according to different UI entry tasks.

Summary of highlights in blt_pm_proc function:

- 1) When UI task requires turning off sleep mode, such as audio (ui_mic_enable) and IR, set bltm.suspend_mask to SUSPEND_DISABLE.
- 2) After advertising for 60s in Advertising state, MCU enters deepsleep with wakeup source set to GPIO PAD in user initialization. 60s timeout is determined by software timer using advertise_begin_tick variable to capture advertising start time.

The design of 60s into deepsleep is to save power, prevent slave wasting power on advertising even when not connected with master. User can justify 60s setting based on different applications.

- 3) At Conn state slave role, under conditions of no key press, no audio or LED task for over 60s from last task, MCU enters deepsleep with GPIO PAD as wakeup source, and at the same time set DEEP_ANA_REG0 label in deepsleep register, so that once after wakeup slave will connect quickly with master.

The design of 60s into deepsleep is to save power. Actually if power consumption under connected state is tuned low enough as with deepsleep retention, it is not absolutely necessary to enter deepsleep.

To enter deepsleep at Conn state slave role, slave first issues a TERMINATE command to master by calling bls_ll_terminateConnection, after receiving ack which triggers BLT_EV_FLAG_TERMINATEcallback function, slave will enter deepslave.

If slave enters deepsleep without sending any request, since master is still at connected state and would constantly try to synchronize with slave till connection timeout. The connection timeout could be a very large value, e.g. 20s. If slave wakes up before 20s, slave would send advertising packet attempting to connect with master. But since master would assume it is already in connected state, it would not be able to connect to slave, and user experience is therefore very slow reconnection.

- 4) If certain task can not be disrupted by long sleep time, user_latency can be set to 0, so latency_use is 0.

Under applications such as key_not_released, or DEVICE_LED_BUSY, call API bls_pm_setManualLatency to set user_latency to 0. When conn_interval is 10ms, sleep time is no more than 10ms.

- 5) For scenario as in item 4, with latency set to 0, slave will wakeup at every conn interval, power might be unnecessarily too high since key scan and LED task does not repeat on every conn interval. Further power optimization can be done as following:

When LONG_PRESS_KEY_POWER_OPTIMIZE=1, once key press is stable (key_matrix_same_as_last_cnt > 5), manually set latency.

For example, with bls_pm_setManualLatency (3), sleep time will not exceed 4 *conn_interval. If conn_interval=10 ms, MCU will wake up every 40ms to process LED task and keyscale.

User can tweak this approach toward different conn intervals and task response time requirements.

4.5 Timer wakeup by Application Layer

At Advertising state or Conn state Slave role, without GPIO PAD wakeup, once MCU enters sleep mode, it only wakes up at T_wakeup pre-determined by BLE SDK. User can not wake up MCU at an earlier time which might be needed at certain scenario.

To provide more flexibility, application layer wakeup and associated callback function are added in the SDK:

Application layer wakeup API:

```
void bls_pm_setAppWakeupLowPower(u32 wakeup_tick, u8 enable);
```

“wakeup_tick” is wakeup time at System Timer tick value.

“enable”: 1-wakeup is enabled; 0-wakeup is disabled.

Registered call back function bls_pm_registerAppWakeupLowPowerCb is executed at application layer wakeup:

```
typedef void (*pm_appWakeupLowPower_callback_t) (int);  
  
void bls_pm_registerAppWakeupLowPowerCb(  
    pm_appWakeupLowPower_callback_t cb);
```

Taking Conn state Slave role as example, when wakeup time app_wakeup_tick is set by bls_pm_setAppWakeupLowPower, the SDK will check whether it is before T_wakeup.

If so, as shown in following diagram, MCU would wake up from sleep at an earlier app_wakeup_tick; if not, MCU would still wake up at T_wakeup.

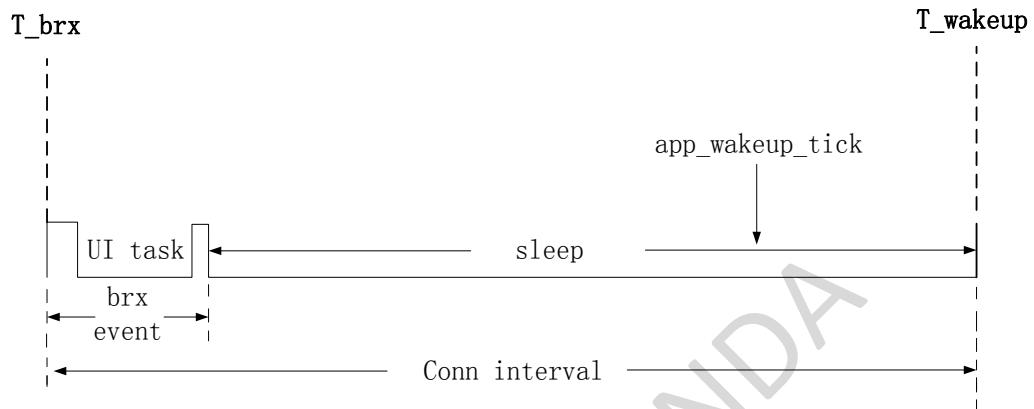


Figure 4-7 Early wake_up at app_wakup_tick

5 Low Battery Detect

Telink BLE SDK and related documents may refer to this subject with different terminologies such as battery power detect/check, battery power detect/check, low battery detect/check, low power detect/check, battery detect/check, etc. For example, in SDK code, there are file and function names such as battery_check, battery_detect, battery_power_check, etc.

In this document, “Low Battery Detect” is used to cover all the related subjects.

5.1 Impact of Low Battery Detect

Since battery voltage would gradually drop with time, for battery-powered products, problems would occur once battery voltage drops below a certain level.

- 1) For 8x5x family, operation voltage range is 1.8V~3.6V. Below 1.8V, normal operation is not guaranteed.
- 2) At low battery voltage, unstable voltage may result in error for flash “write” or “erase” operation, so that unexpected disruption of program FW or user data would lead to product function failure. Based on our MP experience, the threshold is set at 2.0V.

It is necessary to declare a “secure voltage”. Once battery drops below this voltage level, MCU should shutdown to stop working, or enter deepsleep as implemented in the SDK.

Before MCU shutdown, certain UI behavior, e.g. quick LED blinking in the “8258_ble_remote”), can be used as low battery alarm, so as to remind user that it’s time to recharge or replace the low battery.

“Secure voltage”, or “alarm voltage”, is set at 2.0V in current SDK. For certain extreme condition of power instability caused by HW design, secure voltage can be set at higher value, e.g. 2.1 or 2.2V.

For battery-powered products supported by Telink BLE SDK, Low Battery Detect must be implemented to ensure product stability across its lifetime.

5.2 Implementation of Low Battery Detect

For Low Battery Detect, ADC should be used to measure supply voltage. It's recommended to read 8258 Datasheet or ADC driver related document and get familiar with ADC module.

In the SDK, Low Battery Detect is implemented in the "battery_check.h" and "battery_check.c".

Please ensure the macro "BATT_CHECK_ENABLE" is enabled in the "app_config.h". DO NOT change the macro since it's enabled by default.

```
#define BATT_CHECK_ENABLE           1 //must enable
```

5.2.1 Cautions

Low Battery Detect is a basic ADC task to sample and measure supply voltage. Please pay attention to the following items.

5.2.1.1 MUST use GPIO input channel

For Telink 826x family, ADC module supports "VCC/VBAT" input channel to sample supply voltage.

For 8x5x family, though this design is reserved (corresponding to "VBAT" in the "ADC_InputPchTypeDef"), based on certain reasons, it's not allowed to use the "VBAT" channel. Therefore, user must adopt GPIO input channel of ADC instead, which includes PB0~PB7 and PC4~PC5.

```
/*ADC analog positive input channel selection enum*/
typedef enum {
    .....
    B0P,
    B1P,
    B2P,
    B3P,
    B4P,
    B5P,
    B6P,
    B7P,
    C4P,
    C5P,
    .....
    VBAT,
}ADC_InputPchTypeDef;
```

There are two ways to implement ADC sampling of supply voltage via GPIO input channel.

- 1) Directly connect power supply to GPIO input channel of ADC

At ADC initialization, by setting specific GPIO to high impedance (“ie”, “oe” and “output” all set to 0), voltage at the GPIO equals supply voltage, so ADC can directly sample supply voltage.

- 2) Use GPIO high-level output to sample supply voltage without connection between supply and GPIO input channel.

As per the design of 8x5x internal circuit, voltage of GPIO high-level output always equals supply voltage and thus can be sampled by ADC.

In the project “8258_ble_remote” of the SDK, the second method is employed with GPIO input channel selected as PB7.

To use PB7 as GPIO input channel, PB7 should act as general GPIO, and at initialization user should follow its default configurations (ie, oe, output) without the need of special modification.

```
#define GPIO_VBAT_DETECT      GPIO_PB7
#define PB7_FUNC                AS_GPIO
#define PB7_INPUT_ENABLE        0
#define ADC_INPUT_PCHN          B7P
```

PB7 should output high level at ADC sampling.

```
gpio_set_output_en(GPIO_VBAT_DETECT, 1);
gpio_write(GPIO_VBAT_DETECT, 1);
```

Generally, after ADC sampling, PB7 output can be disabled. However, for the “8258_ble_remote”, PB7 on the corresponding HW is floating (NC), high-level output won’t bring any current leakage, PB7 output is not disabled actually.

5.2.1.2 MUST use ADC Differential Mode

Though in theory 8x5x ADC input supports both Single Ended Mode and Differential Mode, in the SDK as well as actual applications, only Differential Mode is allowed, and Single Ended Mode is forbidden.

Differential mode supports positive and negative input channel, thus voltage to be measured equals the voltage difference of positive end and negative end.

If only one GPIO input channel is available for ADC, this GPIO should be set as positive input channel, while GND should be set as negative input channel. By this setting, voltage difference equals voltage of positive end.

Following shows the code for Low Battery Detect in the SDK.

“#if 1” and “#else” are two branches with the same function setting.

“#if 1” is an optimized branch only designed for shorter time on code running.

As better illustrated in “#else”, the API “adc_set_ain_channel_differential_mode” selects PB7 as positive input channel, and GND as 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 Must use Dfifo for ADC sampling value

For Telink 826x family, ADC result is readable via related register.

For 8x5x family, Dfifo mode is used instead to get ADC result. Please refer to the following function in driver.

```
unsigned int adc_sample_and_get_result(void);
```

5.2.1.4 ADC Channel Switch

As shown in 8258 Datasheet, ADC state machine supports left, Right and Misc channel which are time-division multiplexing.

For Low Battery Detect, ADC sampling uses Misc channel which is also needed for other ADC tasks. For AMIC Audio, Left channel is used. Therefore, Low Battery Detect, AMIC Audio and other ADC tasks should be time-division switched. And on every switch, ADC needs to be re-initialized.

5.2.2 Dedicated Low Battery Detect Demo

In the project “8258_ble_remote” of the “app_config.h” file, by setting the macro “BLE_AUDIO_ENABLE” to 0, audio function is disabled, thus ADC is dedicated for Low Battery Detect.

User can also refer to the project “8258_module” for Low Battery Detect demo.

5.2.2.1 Initialization of Low Battery Detect

Refer to the “adc_vbat_detect_init” function.

ADC initialization must always follow the flow below: power off SAR ADC; configure ADC parameters; power on SAR ADC.

```
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);
}
```

It's not recommended to make any change between SAR ADC “power on” line and “power off” line in above code. However, user can select other GPIO input channel via the macro “ADC_INPUT_PCHN”.

If on HW design, supply is directly connected to GPIO input channel, high level output needs to be removed from the “GPIO_VBAT_DETECT”.

Below is code of the “adc_vbat_detect_init” in the “app_battery_power_check”.

```
if (!adc_hw_initialized) {  
    adc_hw_initialized = 1;  
    adc_vbat_detect_init();  
  
}
```

Herein a variable “adc_hw_initialized” is used to invoke one initialization.

When the variable is set to 0, one initialization is invoked, and then the variable should be set to 1 to disable further initialization.

The variable “adc_hw_initialized” is also used in the API below.

```
void battery_set_detect_enable (int en)  
{  
    lowBattDet_enable = en;  
    if (!en) {  
        adc_hw_initialized = 0; //need initialized again  
    }  
}
```

By using the “adc_hw_initialized”, the following functions are realized.

- 1) ADC task switch between Low Battery Detect and other ADC task

To simplify discussion, sleep mode (suspend/deepsleep retention) is not taken into consideration herein but will be covered in next case.

Due to dynamic ADC task switch, the “adc_vbat_detect_init” may be executed multiple times, so it must be implemented in the main_loop rather than user intilialization.

On first invoking of the “app_battery_power_check”, the “adc_vbat_detect_init” is executed, and it won’t be executed repeatedly.

To switch to any other ADC task, for proper initialization in new task, the “battery_set_detect_enable(0)” must be invoked, and the “adc_hw_initialized” will be set to 0.

After completion of other ADC task, the “app_battery_power_check” will be invoked again. Since current adc_hw_initialized is 0, the “adc_vbat_detect_init” must be executed again, so as to guarantee re-initialization on every switch back.

2) Adaptive processing for suspend and deepsleep retention

In this case, sleep mode is taken into consideration.

The “adc_hw_initialized” must be specified to SRAM “data” or “bss” sector, rather than retention_data. This guarantees that after deepsleep retention wake_up, software bootloader (cstartup_xxx.S) will reset it to 0, and it will not be changed after sleep wake_up.

All registers involved in the “adc_vbat_detect_init” are non-volatile in suspend mode, but will lose data in deepsleep retention mode.

Therefore, after suspend wakeup to execute the “app_battery_power_check”, the “adc_hw_initialized” will remain its value before entering suspend, with no need to redo adc_vbat_detect_init.

However, after deepsleep retention wakeup, the “adc_hw_initialized” will become 0, so the “adc_vbat_detect_init” must be invoked again to re-configure related ADC registers.

During suspend mode, registers involved in the “adc_vbat_detect_init” are non-volatile, but Dfifo related registers will lose data. The following code part are placed in the “app_battery_power_check” rather than the “adc_vbat_detect_init”, so as to ensure it will be executed on each battery detect.

```
adc_config_misc_channel_buf((u16 *) adc_dat_buf,  
ADC_SAMPLE_NUM<<2);  
dfifo_enable_dfifo2();
```

In the SDK, by using the “_attribute_ram_code_”, the “adc_vbat_detect_init” is assigned as ram_code, so as to optimize power during long sleep connection. For a typical case of 10ms * (99+1) = 1s sleeping time, MCU wakes up from deepsleep retention mode with the interval of 1s, the “adc_vbat_detect_init” must be executed. When set in ram_code, execution speed will be optimized.

The setting of “_attribute_ram_code_” herein is not mandatory. In actual applications, user can add or remove this setting based on usage of deepsleep retention and power measurement.

5.2.2.2 Low Battery Detect Processing

In the main_loop, the function “app_battery_power_check” is invoked to process Low Battery Detect. Related code is shown as below:

```
_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);
}
```

Default “lowBattDet_enable” is 1, which means Low Battery Detect is enabled by default, so MCU will do battery detect after power up. This variable should be assigned as retention_data to ensure its value is retained during deepsleep retention.

The “lowBattDet_enable” won’t be changed unless ADC needs to switch to other ADC task. When other ADC task starts, the “battery_set_detect_enable(0)” is invoked, so that the main_loop won’t execute the “app_battery_power_check” again.

After completion of other ADC task, the “battery_set_detect_enable(1)” is executed to switch ADC back to Low Battery Detect, so that the main_loop can invoke the “app_battery_power_check” again.

The “lowBattDet_tick” can be used to set frequency of battery detect. In the demo, the period is set as 500ms by default, and it’s adjustable as needed.

Implementation of the “app_battery_power_check” involves details of battery detect initialization, Dfifo setup, data acquisition and processing, as well as low voltage alarming.

Complicated ADC usage and special HW limits may bring user difficulty in understanding. It is highly recommended to follow the demo code as much as possible. Except a few settings which is illustrated as modifiable in this document, DO NOT make any change.

Dfifo mode is used to acquire ADC result by sampling 8 times (default), removing the maximum and minimum and calculating the average of 6 values. As shown in the “adc_vbat_detect_init”, period for each sampling is 10.4us, so it takes 83us or so to get the result.

In demo code, the macro “`ADC_SAMPLE_NUM`” can be changed to 4, so as to reduce total ADC time to 41us. Sampling 8 times is recommended to get more accurate result.

```
#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
```

Same as the “`adc_vbat_detect_init`” ram_code as explained earlier, to speed up code execution and optimize power, the “`app_battery_power_check`” can also be assigned as ram_code.

The setting of “`_attribute_ram_code_`” herein is not mandatory. In actual applications, user can add or remove this setting based on usage of deepsleep retention and power measurement.

```
_attribute_ram_code_ int app_battery_power_check(u16
alram_vol_mv);
```

5.2.2.3 Low battery voltage alarm

The parameter “alram_vol_mv” of the “app_battery_power_check” specifies secure or alarm voltage in unit of mV. As explained earlier, default in the SDK is 2000 mV. In low battery detect of the main_loop, once supply voltage drops below 2000mV, MCU enters low voltage range.

Following shows demo code to process low battery alarm. MCU must shutdown once it enters low voltage range.

In the “8258_ble_remote”, RCU can shutdown by entering deepsleep, and wake up by key press.

Except mandatory MCU shutdown operation, user is allowed to modify other alarm behaviors.

In the code below, alarm indication is set as fast blinking for three times, reminding user to recharge or replace battery.

```
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_LEVEL);  
            sleep_us(200000);  
            gpio_write(GPIO_LED, !LED_ON_LEVEL);  
            sleep_us(200000);  
        }  
    #endif  
  
    analog_write(DEEP_ANA_REG2, LOW_BATT_FLG); //mark  
    cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);  
}
```

For the “8258_ble_remote”, after shutdown at low battery, MCU enters deepsleep which supports key press wakeup.

At the moment of wakeup by key press, SDK will perform one fast battery detect in user initialization rather than in the main_loop, so as to avoid errors as shown below.

Given that LED blinking has been issued for low battery, MCU entered deepsleep, but wakes up from deepsleep by key press. If processed in main_loop, battery detect needs to wait 500ms at least to be executed ; during this duration Slave could have sent many advertising packets, and may even connect with master, which will lead to a bug that device resumes working after low battery alarm.

For this reason, battery detect is performed at user initialization instead. So battery detect is added in user initialization:

```
if(analog_read(DEEP_ANA_REG2) == LOW_BATT_FLG) {  
    app_battery_power_check(VBAT_ALRAM_THRES_MV + 200); //2.2 V  
}
```

Value of the analog register “DEEP_ANA_REG2” can tell whether it’s wakeup from low battery shutdown. Battery detect after this wakeup will raise 2000mV alarm voltage to 2200mV recover voltage, based on the reason below:

Tolerance in battery detect makes it difficult to ensure consistent result on every measurement. Given 20mV error, for first time, 1990mV might trigger MCU to shutdown; however, after wakeup, at user initialization it could be measured at 2005mV, so 2000mV alarm voltage would lead to the bug above.

Considering this, at battery detect after wakeup from shutdown, alarm voltage is raised by a value slightly higher than maximum tolerance of low battery detect.

Since the example 2200mV recover voltage only occurs after voltage lower than 2000mV is detected to shutdown MCU, user does not need to worry that applications would give low voltage alarm when actual voltage is 2.0~2.2V. End user should recharge or replace battery at low battery alarm to ensure normal product performance.

5.2.2.4 Debug mode for Low Battery Detect

There are 2 macros in the demo code of “8258_ble_remote” for debug purpose:

```
#define DBG_ADC_ON_RF_PKT      0  
#define DBG_ADC_SAMPLE_DAT     0
```

Only enable them during debug.

Once the “DBG_ADC_ON_RF_PKT” is enabled, ADC sampling results will be included in advertising packet or data packet of key value at connection state. Since this would change advertising packet and key value packet, it only applies to debug.

After the “DBG_ADC_SAMPLE_DAT” is enabled, interim ADC sampling values can be stored in SRAM.

5.2.3 Low battery detect and Amic Audio

As explained earlier, it is important to properly switch ADC resource between AMIC audio and low battery detect task.

As program starts, battery detect is enabled by default. When Amic Audio task is triggered, perform following two actions:

- 1) Disable battery detect

Invoke the “battery_set_detect_enable(0)” to notify battery detect module that ADC has been switched to other ADC task.

- 2) Amic Audio ADC Initialization

Since AMIC audio uses ADC in a different way from battery detect, ADC re-initialization is required, referring to “Audio” section of this documentation.

At the end of Amic Audio, invoke the “battery_set_detect_enable(1)” to notify battery detect module that ADC resource has been released. ADC re-initialization is required to resume low battery detect.

Same approach applies when ADC tasks other than AMIC Audio co-exist with battery detect.

If battery detect, Amic Audio, and other ADC task co-exist, user can follow the principle that “ADC circuit should be time-division switched” and refer to the method explained herein to implement his own solution.

6 Audio

6.1 Audio initialization

Audio input source includes AMIC and DMIC.

- ✧ DMIC directly uses peripheral audio processing chip to read digital signal to 8x5x chip.
- ✧ For AMIC, multiple analog modules of 8x5x chip, including PGA, ADC and filters, are used to sample and process source audio signal, convert the analog signal to digital signal and then send it to MCU.

6.1.1 AMIC and Low Battery Detect

As explained in **section 5 “Low Battery Detect”**, AMIC Audio, Low Battery Detect, other ADC tasks should be time-division switched to use ADC.

For Telink 826x family, AMIC can be configured at user initialization; while for 8x5x family, AMIC should be configured at the start of Audio task, so as to implement ADC task switch.

6.1.2 AMIC initialization setting

For demo code to process audio, please refer to the project “8258_ble_remote” in the SDK.

```
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
}
```

{}

In the function “ui_enable_mic”, the parameter “en” serves to enable (1) or disable (0) Audio task.

When Audio starts, GPIO_AMIC_BIAS needs to output high level to drive AMIC. After Audio ends, it's needed to disable GPIO_AMIC_BIAS to avoid current leakage during sleep mode.

Following shows AMIC initialization setting:

```
audio_config_mic_buf ( buffer_mic, TL_MIC_BUFFER_SIZE);  
audio_amic_init(AUDIO_16K);
```

When Audio task is executed, specified Dfifo is used to copy all data to SRAM. The “audio_config_mic_buf” serves to configure starting address and size of the Dfifo buffer in SRAM.

When each Audio task starts, it's needed to redo Dfifo configuration in the “ui_enable_mic”, since Dfifo control registers will lose data during suspend.

When Audio task ends, SAR ADC must be disabled to avoid current leakage during suspend.

```
adc_power_on_sar_adc(0);
```

Due to ADC switch usage for AMIC and Low Battery Detect, the “battery_set_detect_enable(!en)” is included in the “ui_enable_mic” to disable or enable battery detect. Please refer to Low Battery Detect section.

Audio task should be executed at UI entry of main_loop.

```
#if (BLE_AUDIO_ENABLE)  
    if(ui_mic_enable){ //audio  
        task_audio();  
    }  
#endif
```

6.1.3 DMIC initialization setting

To be added.

6.2 Audio data processing

6.2.1 Audio data volume and RF transfer

The raw data sampled by AMIC adopt pcm format. The pcm-to-adpcm algorithm can be used to compress the raw data into adpcm format with compression ratio of 25%, thus BLE RF data volume will be decreased largely. Master will decompress the received adpcm-format data back to pcm format.

AMIC sampling rate is 16K*16bits, corresponding to 16K samples of raw data per second, i.e. 16 samples per millisecond (16*16bits=32bytes per ms).

For every 15.5ms, 496-byte (15.5*16=248 samples) raw data are generated. Via pcm-to-adpcm conversion with compression ratio of 1/4, the 496-byte data are compressed into 124 bytes.

The 128-byte data, including 4-byte header and 124-byte compression result, will be disassembled into five packets, and sent to Master in L2CAP layer; since the maximum length of each packet is 27 bytes, the first packet must contain 7-byte l2cap information, including: 2-byte l2caplen, 2-byte chanid, 1-byte opcode and 2-byte AttHandle.

Figure6-1 shows the RF data captured by sniffer. The first packet contains 7-byte extra information and 20-byte audio data, followed by four packets with 27-byte audio data each. As a result, total audio data length is $20 + 27*4 = 128$ bytes.

Data Type	Data Header				CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length				0x8FEFDC	-38	OK
Data Type	Data Header				L2CAP Header		
L2CAP-S	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length ChanId	Opcode AttHandle AttValue
	2	0	1	1	27	0x00083	0x0004 0x18 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				0x8FE4A9	-38	OK
Data Type	Data Header				Generic L2CAP Payload		
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	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	CRC RSSI (dBm) FCS
Empty PDU	Data Header				0x8FEFDC	-38	OK
Data Type	Data Header				Generic L2CAP Payload		
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	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	CRC RSSI (dBm) FCS
Empty PDU	Data Header				0x8FE4A9	-38	OK
Data Type	Data Header				Generic L2CAP Payload		
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	19 09 89 89 A8 08 8A 50 E9 19 8A B8 D0 08 AA F9 88 C1 A0 9A B1 1B 9A 9E CA C9	CRC RSSI (dBm) FCS
Empty PDU	Data Header				0x8FEFDC	-38	OK
Data Type	Data Header				Generic L2CAP Payload		
L2CAP-C	LLID	NESN	SN	MD	PDU-Length	E0 B1 0B 09 1A DB B3 99 A9 D2 99 OF B9 91 C9 B0 B1 CB B2 E1 1A AA 13 OF 3A 47 32	CRC RSSI (dBm) FCS
Empty PDU	Data Header				0x8FE4A9	-38	OK
Data Type	Data Header				CRC	RSSI (dBm)	FCS
Empty PDU	LLID NESN SN MD PDU-Length				0x8FE27A	-38	OK

Figure6-1 Audio data sample

According to “Exchange MTU size” in ATT & GATT ([section 3.3.3 ATT & GATT](#)), since 128-byte long audio data packet are disassembled on Slave side, if peer device needs to re-assemble these received packets, “Exchange MTU size” should be used to determine maximum ClientRxMTU of peer device. Only when “ClientRxMTU” is 128 or above, can the 128-byte long packet of Slave be correctly processed by peer device.

Therefore, if audio task is started, to send 128-byte long packe, the “blc_att_requestMtuSizeExchange” will be invoked to 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);
    }
}
```

The method below is recommended: Register the callback function of MTU size Exchange via the “blc_att_registerMtuSizeExchangeCb”; check in the callback whether “ClientRxMTU” of peer device exceeds or equals 128. Generally ClientRxMTU of Master device is larger than 128, the SDK does not check actual ClientRxMTU via callback.

Following is the audio service in Attribute Table:

```
// 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},
```

Figure6-2 MIC service in Attribute Table

The second Attribute above is used to transfer audio data. This Attribute uses “Handle Value Notification” to send Data to Master. After Master receives Handle Value Notification, the Attribute Value data corresponding to the five successive packets will be assembled into 128 bytes, and then decompressed back to the pcm-format audio data.

6.2.2 Audio data compression

Related macros are defined in the “app_config.h”, as shown below:

#define	ADPCM_PACKET_LEN	128
#define	TL_MIC_ADPCM_UNIT_SIZE	248
#define	TL_MIC_BUFFER_SIZE	992

Each compression needs to process 248-sample, i.e. 496-byte data. Since AMIC continuously samples audio data and transfers the processed pcm-format data into buffer_mic, considering data buffering and preservation, this buffer should be pre-configured so that it can store 496 samples for two compressions.

If 16K sampling rate is used, then 496 samples correspond to 992 bytes, i.e. “TL_MIC_BUFFER_SIZE” should be configured as 992.

“buffer_mic” is defined as below:

```
s16 buffer_mic[TL_MIC_BUFFER_SIZE>>1]; // 496 samples, 992 bytes  
config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE);
```

Following shows the mechanism of data filling into buffer_mic via HW control.

Data sampled by AMIC are transferred into memory starting from buffer_mic address with 16K speed; once the maximum length 992 is reached, data transfer returns to the buffer_mic address, the old data will be replaced directly without checking whether it's read.

It's needed to maintain a write pointer when transferring data into RAM; the pointer is used to indicate the address in RAM for current newest audio data.

The “buffer_mic_enc” is defined to store the 128-byte compression result data; the buffer number is configured as 4 to indicate result of up to four compressions can be buffered.

```
int buffer_mic_enc[BUFFER_PACKET_SIZE];
```

Since “BUFFER_PACKET_SIZE” is 128, and “int” occupies four bytes, it's equivalent to $128 * 4$ signed char.

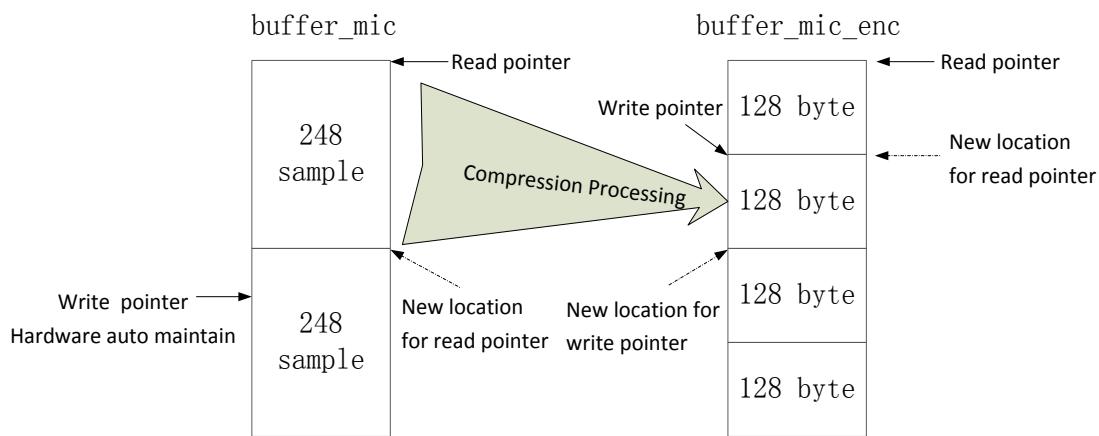


Figure6-3 Data compression processing

The figure above shows data compression processing method:

The buffer_mic automatically maintains a write pointer by hardware, and maintains a read pointer by software.

Whenever SW detects there're 248 samples between the two pointers, the compression handler is invoked to read 248-sample data starting from the read pointer and compress them into 128 bytes; the read pointer moves to a new location to indicate following data are new and not read.

The buffer_mic is continuously checked whether there're enough 248-sample data; if so, the data are compressed and transferred into the buffer_mic_enc.

Since 248-sample data are generated for every 15.5ms, the firmware must check the buffer_mic with maximum frequency of 1/15.5ms. The FW only executes the task_audio once during each main_loop, so the main_loop duration must be less than 15.5ms to avoid audio data loss. In Conn state, the main_loop duration equals connection interval; so for applications with audio task, connection interval must be less than 15.5ms. It's recommended to configure connection interval as 10ms.

The buffer_mic_enc maintains a write pointer and a read pointer by software: after the 248-sample data are compressed into 128 bytes, the compression result are copied into the buffer address starting from the write pointer, and the buffer_mic_enc is checked whether there's overflow; if so, the oldest 128-byte data are discarded and the read pointer switches to the next 128 bytes.

The compression result data are copied into BLE RF Tx buffer as below:

The buffer_mic_enc is checked if it's non-empty (when writer pointer equals read pointer, it indicates "empty", otherwise it indicates "non-empty"); if the buffer is non-empty, the 128-byte data starting from the read pointer are copied into the BLE RF Tx buffer, then the read pointer moves to the new location.

The function “proc_mic_encoder” is used to process Audio data compression.

6.3 Compression and decompression algorithm

The function below is used to invoke the compression algorithm:

```
void mic_to_adpcm_split (signed short *ps, int len, signed short *pds, int start);
```

Notes:

- ✧ “ps” points to the starting storage address for data before compression, which corresponds to the read pointer location of the buffer_mic as shown in Figure6-3;
- ✧ “len” is configured as “TL_MIC_ADPCM_UNIT_SIZE (248)”, which indicates 248 samples;
 - ✧ “pds” points to the starting storage address for compression result data, which corresponds to the write pointer location of the buffer_mic_enc as shown in Figure6-3.

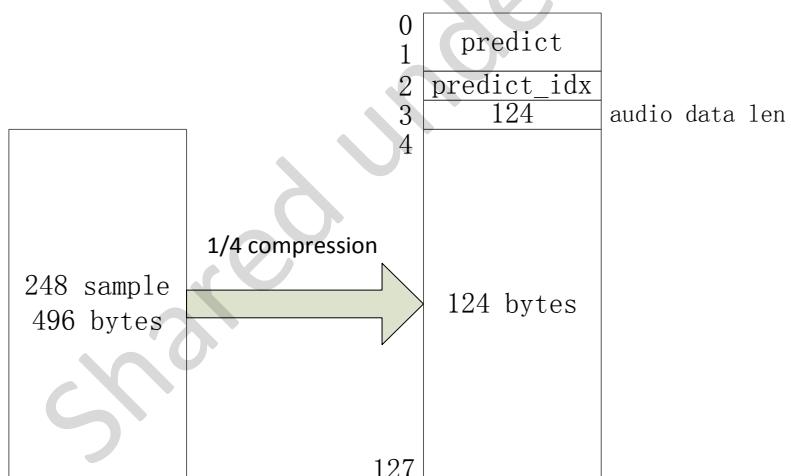


Figure6-4 Data corresponding to compression algorithm

After compression, the data space stores 2-byte predict, 1-byte predict_idx, 1-byte length of current valid adpcm-format audio data (i.e. 124), and 124-byte data compressed from the 496-byte raw data with compression ratio of 1/4.

The function below is used to invoke the decompression algorithm:

```
void adpcm_to_pcm (signed short *ps, signed short *pd, int len);
```

Notes:

- ✧ “ps” points to the starting storage address for data to be decompressed (i.e. 128-byte adpcm-format data). This address needs user to define a buffer to store 128-byte data copied from BLE RF.
- ✧ “pd” points to the starting storage address for 496-byte pcm-format audio data after decompression. This address needs user to define a buffer to store data to be transferred when playing audio.
- ✧ “len” is 248, same as the “len” during compression.

As shown in

Figure6-4, during decompression, the data read from the buffer are two-byte predict, 1-byte predict_idx, 1-byte valid audio data length “124”, and the 124-byte adpcm-format data which will be decompressed into 496-byte pcm-format audio data.

7 OTA

To implement OTA for 8x5x BLE Slave, a device is needed to act as BLE OTA Master, which can be the Bluetooth device (supporting OTA in APP) combined with Slave, or simply Telink BLE Master kma Dongle.

8x5x family supports flash multi-address booting from multiples of 128kB offsets (e.g. boot from 0x00000, 0x20000, 0x40000).

In the example of this section, to illustrate OTA details, Telink kma dongle acts as OTA Master, and booting address adopts 0x0 and 0x20000.

7.1 Flash architecture and OTA procedure

7.1.1 FLASH storage architecture

When booting address is 0x20000, size of firmware compiled by the SDK should not exceed 128kB, i.e. the flash area 0~0x20000 serves to store firmware.

If you're using boot address as 0x0 and 0x20000, the firmware size shouldn't be larger than 124K. if your firmware size is larger than 124K, then you would need to use 0x0 and 0x40000 as boot address, the firmware size shouldn't be larger than 252K.

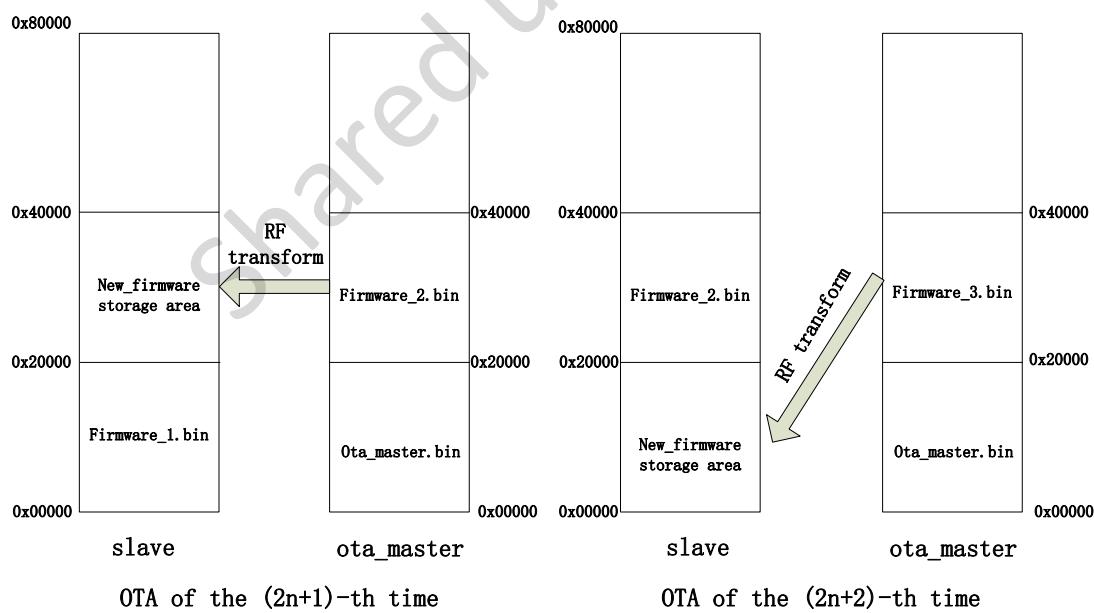


Figure 7-1 Flash storage structure

- 1) OTA Master burns new firmware2 into the Master flash area starting from 0x20000.
- 2) OTA for the first time:

- A. When power on, Slave starts booting and executing firmware1 from flash 0~0x20000.
 - B. When firmware1 is running, the area of Slave flash starting from 0x20000 (i.e. flash 0x20000~0x40000) is cleared during initialization and will be used as storage area for new firmware.
 - C. OTA process starts, Master transfers firmware2 into Slave flash area starting from 0x20000 via RF. Then Slave sets bootloader to boot from the new firmware offset and reboots (similar to power cycle).
- 3) For subsequent OTA updates, OTA Master first burns new firmware3 into the Master flash area starting from 0x20000.
 - 4) OTA for the second time:
 - A. When power on, Slave starts booting and executing firmware2 from flash 0x20000~0x40000.
 - B. When firmware2 is running, the area of Slave flash starting from 0x0 (i.e. flash 0~0x20000) is cleared during initialization and will be used as storage area for new firmware.
 - C. OTA process starts, Master transfers firmware3 into Slave flash area starting from 0x0 via RF. Then Slave sets bootloader to boot from the new firmware offset and reboots.
 - 5) Subsequent OTA process repeats steps 1)~4): 1)~2) represents OTA of the (2n+1)-th time, while 3)~4) represents OTA of the (2n+2)-th time.

7.1.2 OTA update procedure

Based on the flash storage structure introduced in **Section 7.1.1 FLASH storage architecture**, the OTA update procedure is illustrated as below:

8x5x multi-address booting mechanism: OTA only uses two addresses booting (boot from 0 or 0x20000). After MCU is powered on, Slave boots from flash address 0 by default. First flash address 0x8 is read, if its value is 0x4b, the code starting from 0 are transferred to RAM, and the following instruction fetch address equals 0 plus PC pointer value; if the value of flash 0x8 is not 0x4b, MCU directly reads flash address 0x20008, if its value is 0x4b, the code starting from 0x20000 are transferred to RAM, and the following instruction fetch address equals 0x20000 plus PC pointer value.

Therefore, by modifying flag bit value of flash 0x8 and 0x20008, the part of flash code to be executed will be determined.

In the SDK, the OTA upgrade process of the (2n+1)-th or (2n+2)-th time is shown as below:

- 1) After MCU is powered on, read flash address 0x8 and 0x20008, and compare the

value with 0x4b to determine the booting address; then Slave boots from corresponding address (0 or 0x20000) and starts executing the code. This function is automatically completed by MCU hardware.

- 2) During firmware initialization, read MCU hardware register to judge the booting address.
 - a) If booting address is 0, the “ota_program_offset” is set as 0x20000, and the area of Slave flash starting from 0x20000 (i.e. 0x20000~0x40000) will be all erased to “0xff”, which indicates the new firmware will be transferred into this area by Master during the following OTA process.
 - b) If booting address is 0x20000, the “ota_program_offset” is set as 0x0, and the area of Slave flash starting from 0x0 (i.e. 0~0x20000) will be all erased to 0xff, which indicates the new firmware will be transferred into this area by Master during the following OTA process.
- 3) Slave MCU executes the firmware after booting; OTA Master is powered on and establishes BLE connection with Slave.
- 4) Trigger OTA Master to enter OTA mode by UI (e.g. button press, write memory by PC tool, etc.). After entering OTA mode, OTA Master needs to obtain Handle value of Slave OTA Service Data Attribute (The handle value can be pre-appointed by Slave and Master, or obtained via “read_by_type”).
- 5) After the Attribute Handle value is obtained, OTA Master may need to obtain version number of current Slave Flash firmware, and compare it with the version number of local stored new firmware. This step is determined by user.
- 6) To enable OTA upgrade, OTA Master will send an OTA_start command to inform Slave to enter OTA mode.
- 7) After the OTA_start command is received, Slave enters OTA mode and waits for OTA data to be sent from Master.
- 8) Master reads the firmware stored in the flash area starting from 0x20000, and continuously sends OTA data to Slave until the entire firmware is sent.
- 9) After the OTA data are received, Slave stores the data into the area starting from the “ota_program_offset”.
- 10) After the OTA data are sent, Master will check if all data are correctly received by Slave (invoke related BLE function in bottom layer to judge whether Link Layer data are all correctly acknowledged).
- 11) After Master confirms all OTA data are correctly received by Slave, it will send an OTA_END command.
- 12) After Slave receives the OTA_END command, offset address 8 based on the new firmware starting address (i.e. ota_program_offset+8) is written with “0x4b”, and

offset address 8 based on the old firmware starting address is written with “0x00”. This indicates Slave will execute the firmware from the new area after the next booting.

- 13) Slave reboots, and the new firmware will take effect.
- 14) During the whole OTA upgrade process, Slave will continuously check whether there's packet error, packet loss or timeout (A timer is started when OTA starts). Once packet error, packet loss or timeout is detected, Slave will determine the OTA process fails. Then Slave reboots, and executes the old firmware.

The OTA related operations on Slave side described above have been realized in the SDK and can be used by user directly. On Master side, extra firmware design is needed and it will be introduced later.

7.1.3 Modify FW size and booting address

The API “bls_ota_set_fwSize_and_fwBootAddr” serves to modify maximum firmware size and booting address in the OTA design. Herein booting address means the address except 0 to store New_firmware, so it should be either 0x20000 or 0x40000.

In the SDK, by default, maximum firmware size is 124kB, and booting address is 0x20000.

```
//firmware_size_k must be 4k aligned  
  
void bls_ota_set_fwSize_and_fwBootAddr(int firmware_size_k,  
                                         int boot_addr);
```

*Notes:

- ❖ “firmware_size_k” must be 4kB aligned, i.e. it must be integral multiples of 4kB. For example, for 97kB size, the “firmware_size_k” must be set as 100kB.
- ❖ Since the “cpu_sleep_wakeup()” will configure the “firmware_size_k” and “boot_addr”, this API must be invoked before the “cpu_wakeup_init” to take effect.

If maximum firmware_size exceeds 124kB, it's needed to change booting address to 0x40000 (no more than 252K). For example, maximum firmware_size may reach 200kB; corresponding setting should be:

```
bls_ota_set_fwSize_and_fwBootAddr(200, 0x40000);
```

By using this API, not only booting address can be modified, but also flash area

usage can be optimized.

By default, maximum firmware size is 128kB, and the flash space 0x00000 ~ 0x40000 can be used to store firmware only. If firmware does not need such a large area, e.g. FW size does not exceed 60kB, only part of the two 128kB space (0x00000 ~ 0x20000, 0x20000 ~ 0x40000) are used.

To use the redundant space as data storage area, the setting below can be followed.

```
bls_ota_set_fwSize_and_fwBootAddr(60, 0x20000);
```

By the configuration above, the two 60kB flash areas 0x00000 ~ 0x0F000 and 0x20000 ~ 0x2F000 can be used as firmware storage space, while the two 68kB flash areas 0x0F000 ~ 0x20000 and 0x2F000 ~ 0x40000 can be used as user data storage space.

7.2 RF data processing for OTA mode

7.2.1 OTA processing in Attribute Table on Slave side

OTA related contents needs to be added in the Attribute Table.

The “att_readwrite_callback_t r” and “att_readwrite_callback_t w” of the OTA data Attribute should be set as otaRead and otaWrite, respectively; the attribute should be set as Read and Write_without_Rsp (Master sends data via “Write Command” with no need of ack from Slave to enable faster speed).

```
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},
```

When Master sends OTA data to Slave, it actually writes data to the second Attribute as shown above, so Master needs to know the Attribute Handle of this Attribute in the Attribute Table. To use the Attribute Handle value pre-appointed by Master and Slave, user can directly define it on Master side.

7.2.2 OTA data packet format

Master sends command and data to Slave via “Write Command” in L2CAP layer.

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

Figure7-2 Write Command format in BLE stack

The Attribute Handle value is the handle_value of OTA data on Slave side.

The Attribute Value length is set as 20, and its format is shown as below.

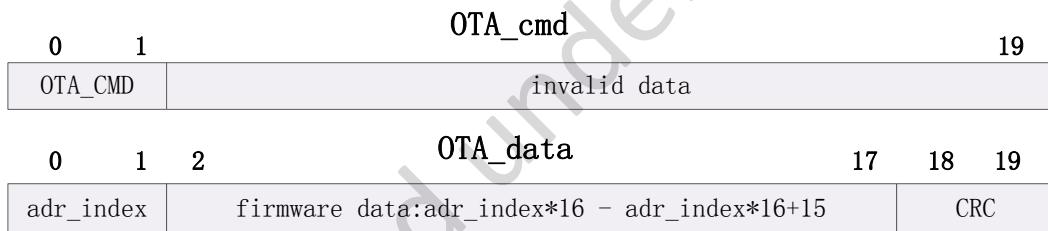


Figure7-3 Format of OTA command and data

When the first two bytes are 0xff00 ~0xff10, it indicates OTA command and the command type.

- 1) 0xff00: OTA_FW_VERSION, request to obtain current Slave firmware version number.

This command is reserved and optional. When using this command, related callback function is available on Slave for user to transfer FW version number.

- 2) 0xff01: OTA_Start command.

To start OTA upgrade process, Master needs to send this command to Slave.

- 3) 0xff02: OTA_end command.

When Master confirms all OTA data are correctly received by Slave, it will send this command, which can be followed by four valid bytes to re-confirm Slave has received all data from Master.

- 4) 0xff03 ~ 0xff0f: to be added.

When the first two bytes are 0~0x1000, it indicates it's an OTA data. Each OTA data packet transfers 16-byte firmware data, and the "adr_index" is the actual FW address divided by 16. "adr_index=0" indicates OTA data are values of FW addresses 0x0~0xf; "adr_index=1" indicates OTA data are values of FW addresses 0x10~0x1f. The last two bytes are the first CRC value calculated by CRC_16 operation to the former 18 bytes. After Slave receives the OTA data, it will also carry out CRC calculation, and the data will be regarded as valid only when the result matches the CRC (19th~20th byte) of the data.

7.2.3 RF transfer processing on Master side

Since BLE link-layer RF data will be automatically responded with ack to avoid packet loss, during OTA data transfer, Master won't check whether each OTA data is acknowledged. In other words, after sending an OTA data via write command, Master won't check if there's ack response from Slave by software, but will directly push the following data into HW TX buffer which yet does not have enough data to be sent.

OTA Master processes RF transfer by software as below:

- 1) Check if there's any behavior to trigger entering OTA mode. If so, Master enters OTA mode.
- 2) To send OTA commands and data to Slave, Master needs to know the Attribute Handle value of current OTA data Attribute on Slave side. User can decide to directly use the pre-appointed value or obtain the Handle value via "Read By Type Request".

UUID of OTA data in Telink BLE SDK is always 16-byte value as shown below:

```
#define TELINK_SPP_DATA_OTA
{0x12,0x2B,0x0d,0x0c,0x0b,0xa,0x09,0x08,0x07,0x06,0x05,0x04,
0x03,0x02,0x01,0x00} //!< TELINK_SPP data for ota
```

In "Read By Type Request" from Master, the "Type" is set as the 16-byte UUID. The Attribute Handle for the OTA UUID is available from "Read By Type Rsp" responded by Slave. In the figure below, the Attribute Handle value is shown as "0x0031".

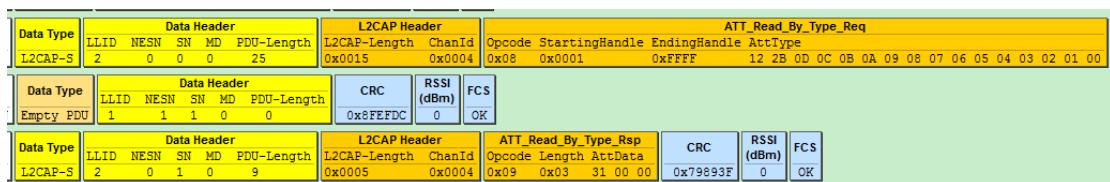


Figure 7-4 Master obtains OTA Attribute Handle via "Read By Type Request"

- 3) (optional) Obtain current Slave firmware version number. User can check if it's the newest version and decide whether to start OTA upgrade correspondingly.

In this BLE SDK, user needs to determine the method to obtain FW version number.

An OTA version command is reserved; however, the transfer of version number is not realized in current SDK. An “OTA version cmd” can be sent to Slave in the form of “write cmd”; Slave only supplies a callback function after it receives the request, and user needs to decide in the callback function how to transfer Slave FW version number to Master (e.g. manually send a NOTIFY/INDICATE data).

- 4) Start a timer when OTA starts, and continuously check if the count value exceeds the timeout duration (e.g. 15s, only for reference). If so, it's regarded as OTA failure due to timeout.
- 5) Since Slave will check CRC after the OTA data are received, once there's CRC error or any other error (e.g. flash burning error), OTA fails, and firmware is directly rebooted; the link layer can't respond to Master with ack, and Master fails to send data until timeout.
- 6) Read four bytes of Master flash 0x20018~0x2001b to determine firmware size which is realized by compiler.

Suppose FW size is 20kB (0x5000), the value of firmware 0x18~0x1b is 0x00005000, so the FW size can be read from 0x20018~0x2001b.

As shown below, 0x18~0x1b of “8258_remote.bin” is “0x00005a98”, so the firmware size is 0x5a98, i.e. 23192 bytes from 0 to 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

Figure7-5 firmware sample: starting part

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								

Figure7-6 firmware sample: ending part

- 7) Master sends an OTA start command “0xff01” to Slave, so as to inform it to enter OTA mode and wait for OTA data from Master, as shown below.

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	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF	0x61875B	0	OK

Figure7-7 “OTA start” sent from Master

- 8) Read 16 bytes of firmware each time starting from Master flash 0x20000, assemble them into OTA data packet, set corresponding adr_index, calculate CRC value, and push the packet into TX FIFO, until all data of the firmware are sent to Slave.

OTA data format is used in data transfer (see Figure7-3): 20-byte valid data contains 2-byte adr_index, 16-byte firmware data and 2-byte CRC value to the former 18 bytes.

Note: If firmware data for the final transfer are less than 16 bytes, the remaining bytes should be complemented with “0xff” and need to be considered for CRC calculation.

The 8258_remote.bin as shown in Figure7-5 and Figure7-6 is taken as an example to illustrate how to assemble OTA data.

Data for first transfer: “adr_index” is “0x00 00”, 16-byte data are values of addresses 0x0000 ~ 0x000f. Suppose CRC calculation result for the former 18 bytes is “0xXYZW”, the 20-byte data should be:

0x00 0x00 0x0e 0x80 ... (12 bytes not listed)... 0x88 0x00 0xZW 0xXY

Data for second transfer:

0x01 0x00 0x5e 0x80 ... (12 bytes not listed)... 0x00 0x00 0xJK 0xHI

Data for third transfer:

0x02 0x00 0x25 0x08 ... (12 bytes not listed)... 0xfa 0x87 0xNO 0xLM

.....

Data for penultimate transfer:

0xa8 0x05 0x02 0x04 ... (12 bytes not listed)... 0x00 0x00 0xST 0xPQ

Data for final transfer:

0xa9 0x05 0x44 0x58 0x00 0x00 0x01 0x00 0x00 0x00

0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xWX 0xUV

Since the firmware data for final transfer are only 8 bytes, eight “0xff” are added to complement 16 bytes. CRC calculation result for the former 18 bytes (0xa9 ~ 0xff) is “0xUVWX”.

Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 0 1 9	L2CAP Header L2CAP-Length ChanId 0x0005 0x0004	ATT_Write_Command Opcode AttHandle AttrValue 0x52 0x0031 01 FF	CRC 0x61875B	RSSI (dBm) 0	FCS OK
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 0 0 0	CRC 0x8FE27A	RSSI (dBm) 0	FCS OK		
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 1 1 1 27	L2CAP Header L2CAP-Length ChanId 0x0017 0x0004	ATT_Write_Command Opcode AttHandle AttrValue 0x52 0x0031 00 00 0E 80 01 03 00 00 00 00 4B 4E 4C 54 80 01 88 00 BA A5			
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 0 1 0 0	CRC 0x8FE90F	RSSI (dBm) 0	FCS OK		
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 0 1 27	L2CAP Header L2CAP-Length ChanId 0x0017 0x0004	ATT_Write_Command Opcode AttHandle AttrValue 0x52 0x0031 01 00 SE 80 00 00 00 00 00 00 98 5A 00 00 00 00 00 EA EF			
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 0 0 0	CRC 0x8FE27A	RSSI (dBm) 0	FCS OK		
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 1 1 0 27	L2CAP Header L2CAP-Length ChanId 0x0017 0x0004	ATT_Write_Command Opcode AttHandle AttrValue 0x52 0x0031 02 00 25 08 26 09 26 0A 91 02 02 CA 08 50 04 B1 FA 87 A7 0D			
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 0 0 1 27	L2CAP Header L2CAP-Length ChanId 0x0017 0x0004	ATT_Write_Command Opcode AttHandle AttrValue 0x52 0x0031 A9 05 44 58 00 00 01 00 00 00 FF FF FF FF FF FF FF FF 44 47			
Data Type Empty PDU	Data Header LLID NESN SN MD PDU-Length 1 1 0 0 0	CRC 0x8FE27A	RSSI (dBm) 0	FCS OK		
Data Type L2CAP-S	Data Header LLID NESN SN MD PDU-Length 2 1 1 0 13	L2CAP Header L2CAP-Length ChanId 0x0009 0x0004	ATT_Write_Command Opcode AttHandle AttrValue 0x52 0x0031 02 FF A9 05 56 FA	CRC 0xE13FFA	RSSI (dBm) 0	FCS OK

Figure7-8 Master OTA data

- 9) After firmware data are sent, Master checks if BLE link-layer data are all sent out (It's considered the data is sent successfully only when link-layer data is acked by Slave). If all data are sent, Master will send an ota_end command to inform Slave.

"OTA end" packet is set as 6 valid bytes: first two bytes are "0xff02", followed by maximum adr_index value of new firmware (the two bytes are used to re-confirm if there're OTA data lost on Slave side), the final two bytes are inverted value of the maximum adr_index (equivalent to simple check). CRC check is not needed for "OTA end".

The maximal adr_index and inverted value of "8258_remote.bin" are "0x05a9" and "0xfa56", respectively. Figure7-8 shows the final OTA end packet.

- 10) Check if link-layer TX FIFO on Master side is empty: If it's empty, it indicates all data and commands in above steps are sent successfully, i.e. OTA task on Master succeeds.

Please refer to Appendix for CRC_16 calculation function.

7.2.4 RF receive processing on Slave side

As introduced above, Slave can directly invoke the otaWrite and otaRead in OTA Attribute. After Slave receives write command from Master, it will be parsed and processed automatically in BLE stack by invoking the otaWrite function. In the otaWrite function, the 20-byte packet data will be parsed: first judge whether it's OTA CMD or OTA data, then process correspondingly (respond to OTA cmd; check CRC to OTA data and burn data into specific addresses of flash).

The OTA related operations on Slave side are shown as below::

- 1) OTA_FIRMWARE_VERSION command is received (first two bytes are 0xff00):
Master requests to obtain Slave firmware version number.

In this BLE SDK, after Slave receives this command, it will only check whether related callback function is registered and determine whether to trigger the callback function correspondingly.

The interface in ble_ll_ota.h to register this callback function is shown as below:

```
typedef void (*ota_versionCb_t) (void);  
void bls_ota_registerVersionReqCb(ota_versionCb_t cb);
```

- 2) OTA start command is received (first two bytes are 0xff01): Slave enters OTA mode.

If the “bls_ota_registerStartCmdCb” function is used to register the callback function of OTA start, then the callback function is executed to modify some parameter states after entering OTA mode (e.g. disable PM to stabilize OTA data transfer).

Slave also starts and maintains a slave_adr_index to record the adr_index of the latest correct OTA data. The slave_adr_index is used to check whether there's packet loss in the whole OTA process, and its initial value is -1. Once packet loss is detected, OTA fails, Slave MCU exits OTA and reboots; since Master cannot receive any ack from Slave, it will discover OTA failure by software after timeout.

The following interface is used to register the callback function of OTA start:

```
typedef void (*ota_startCb_t) (void);  
void bls_ota_registerStartCmdCb(ota_startCb_t cb);
```

User needs to register this callback function to carry out operations when OTA starts, for example, configure LED blinking to indicate OTA process.

After Slave receives “OTA start”, it enters OTA and starts a timer (The timeout duration is set as 30s by default in current SDK). If OTA process is not finished within the duration, it's regarded as OTA failure due to timeout. User can evaluate firmware size (larger size takes more time) and BLE data bandwidth on Master (narrow bandwidth will influence OTA speed), and modify this timeout duration accordingly via the interface as shown below.

```
void bls_ota_setTimeout(u32 timeout_us); // unit: us
```

3) Valid OTA data are received (first two bytes are 0~0x1000):

Whenever Slave receives one 20-byte OTA data packet, it will first check if the adr_index equals slave_adr_index plus 1. If not equal, it indicates packet loss and OTA failure; if equal, the slave_adr_index value is updated.

Then carry out CRC_16 check to the former 18 bytes. If not matched, OTA fails; if matched, the 16-byte valid data are written into corresponding flash area (ota_program_offset+adr_index*16 ~ ota_program_offset+adr_index*16 + 15). During flash writing process, if there's any error, OTA also fails.

4) "OTA end" command is received (first two bytes are 0xff02):

Check whether adr_max in OTA end packet and the inverted check value are correct. If yes, the adr_max can be used to double check whether maximum index value of data received by Slave from Master equals the adr_max in this packet. If equal, OTA succeeds; if not equal, OTA fails due to packet loss.

After successful OTA, Slave will set the booting flag of the old firmware address in flash as 0, set the booting flag of the new firmware address in flash as 0x4b, then reboot MCU.

5) Slave supplies OTA state callback function:

After Slave starts OTA, MCU will finally reboot regardless of OTA result.

If OTA succeeds, Slave will set flag before rebooting so that MCU executes the New_firmware.

If OTA fails, the incorrect new firmware will be erased before rebooting, so that MCU still executes the Old_firmware.

Before rebooting, user can judge whether the OTA state callback function is registered and determine whether to trigger it correspondingly.

```
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);
```

The “enum” lists the 6 options for the parameter “result”: the first value indicates OTA success; the other five values indicate reasons for OTA failure.

The “result” is mainly used for debugging: When OTA fails, user can read the “result”, stop MCU by using “while(1)”, and find the reason for current OTA failure.

For demo code of OTA debugging, please refer to the project “8258_ble_remote” in the SDK with LED blinking to indicate OTA result.

8 Key Scan

Keypad architecture based on row/column scan is used to detect and process key state update (press/release). User can directly use the demo code, or realize the function by developing his own code.

In this section, the demo “8258_ble_remote” in the SDK is taken as an example to illustrate key scan.

8.1 Key matrix

Figure8-1 shows a 5*6 Key matrix which supports up to 30 buttons. Five drive pins (Row0~Row4) serve to output drive level, while six scan pins (CoL0~CoL5) serve to scan for key press in current column.

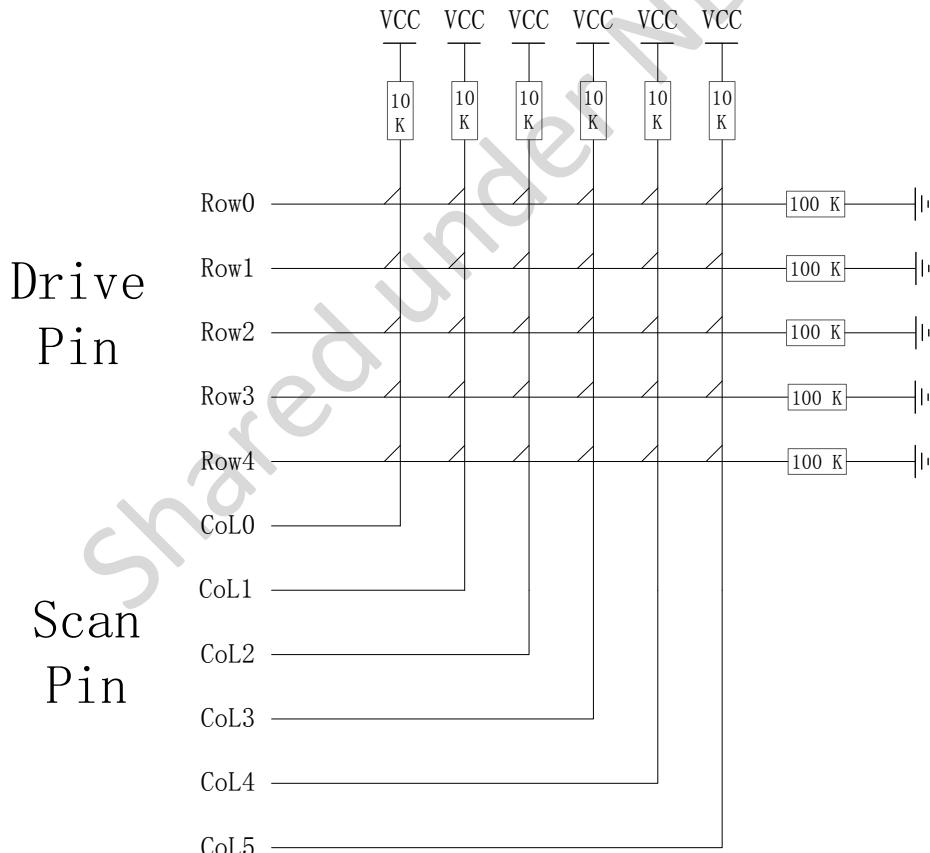


Figure8-1 Row/Column key matrix

Keypad related configurations in the app_config.h are shown as below:

On Telink demo board, Row0~Row4 pins are PD5, PD2, PD4, PD6 and PD7, while CoL0~CoL5 pins are PC5, PA0, PB2, PA4, PA3 and PD3.

Define drive pin array and scan pin array:

```
#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}
```

Keypress adopts analog pull-up/pull-down resistor of GPIO: drive pins use 100K pull-down resistor, and scan pins use 10K pull-up resistor.

When no button is pressed, scan pins act as input GPIOs and read high level due to 10K pull-up resistor.

When key scan starts, drive pins output low level; if low level is detected on a scan pin, it indicates there's button pressed in current column (Note: Drive pins are not in float state, if output is not enabled, scan pins still detect high level due to voltage division of 100K and 10K resistor.)

Define valid voltage level detected on scan pins when drive pins output low level in Row/Column scan:

```
#define KB_LINE_HIGH_VALID 0
```

Define pull-up resistor for scan pins and pull-down resistor for drive pins:

```
#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
```

Since “ie” of general GPIOs is set as 0 by default in gpio_init, to read level on scan pins, corresponding “ie” should be enabled.

```
#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
```

When MCU enters sleep mode, it's needed to configure PAD GPIO wakeup. Set drive pins as high level wakeup; when there's button pressed, drive pin reads high level, which is 10/11 VCC (i.e. $VCC * 100K/(100K+10K)$). To read level state of drive pins, corresponding “ie” should be enabled.

```
#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 Keypad and keymap

8.2.1 Keypad

After configuration as shown in **section 8.1 Key matrix**, the function below is invoked in main_loop to implement keypad.

```
u32 kb_scan_key (int numlock_status, int read_key)
```

- ❖ numlock_status: Generally set as 0 when invoked in main_loop. Set as “KB_NUMLOCK_STATUS_POWERON” only for fast keypad after wakeup from deepsleep (refer to **section 8.4 Deepsleep wake_up fast keypad**, corresponding to DEEPBACK_FAST_KEYSCAN_ENABLE).
- ❖ read_key: Buffer processing for key values, generally not used and set as 1 (if it's set as 0, key values will be pushed into buffer and not reported to upper layer).
- ❖ The return value is used to inform user whether matrix keyboard update is detected by current scan: if yes, return 1; otherwise return 0.

The “kb_scan_key” is invoked in main_loop. As introduced in **section 3.2.4 Link Layer timing sequence**, each main_loop is an adv_interval or conn_interval. In advertising state (suppose adv_interval is 30ms), key scan is processed once for each 30ms; in connection state (suppose conn_interval is 10ms), key scan is processed

once for each 10ms.

In theory, when button states in matrix are different during two adjacent key scans, it's considered as an update.

In actual code, a debounce filtering processing is enabled: It will be considered as a valid update, only when button states stay the same during two adjacent key scans, but different with the latest stored matrix keyboard state. "1" will be returned by the function to indicate valid update, matrix keyboard state will be indicated by the structure "kb_event", and current button state will be updated to the newest matrix keyboard state. Corresponding code in keyboard.c is shown as below:

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
```

The newest button state means press or release state set of all buttons in the matrix. When power on, initial matrix keyboard state shows all buttons are "released" by default, and debounce filtering processing is enabled. As long as valid update occurs to the button state, "1" will be returned, otherwise "0" will be returned.

For example: press a button, a valid update is returned; release a button, a valid update is returned; press another button with a button held, a valid update is returned; press the third button with two buttons held, a valid update is returned; release a button of the two pressed buttons, a valid update is returned.....

8.2.2 Keypad &kb_event

If a valid button state update is detected by invoking the "kb_scan_key", user can obtain current button state via a global structure variable "kb_event".

```
#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;
```

The "kb_event" consists of 8 bytes:

"cnt" serves to indicate valid count number of pressed buttons currently;

"ctrl_key" is not used generally except for standard USB HID keyboard (user is not allowed to set keycode in keymap as 0xe0~0xe7).

keycode[6] indicates keycode of up to six pressed buttons can be stored (if more than six buttons are pressed actually, only the former six can be reflected).

Keycode definition of all buttons in the “app_config.h” is shown as below:

```
#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, }
```

The keymap follows the format of 5*6 matrix structure. The keycode of pressed button can be configured accordingly, for example, the keycode of the button at the cross of Row0 and Col0 is “VK_B”.

In the “kb_scan_key” function, the “kb_event.cnt” will be cleared before each scan, while the array “kb_event.keycode[]” won’t be cleared automatically. Whenever “1” is returned to indicate valid update, the “kb_event.cnt” will be used to check current valid count number of pressed buttons.

1) If current kb_event.cnt = 0, previous valid matrix state “kb_event.cnt” must be uncertain non-zero value; the update must be button release, but the number of released button is uncertain. Data in kb_event.keycode[] (if available) is invalid.

2) If current kb_event.cnt = 1, the previous kb_event.cnt indicates button state update.

If previous kb_event.cnt is 0, it indicates the update is one button is pressed; if previous kb_event.cnt is 2, it indicates the update is one of the two pressed buttons is released; if previous kb_event.cnt is 3, it indicates the update is two of the three pressed buttons are released.....

kb_event.keycode[0] indicates the key value of currently pressed button. The subsequent keycodes are negligible.

3) If current kb_event.cnt = 2, the previous kb_event.cnt indicates button state update.

If previous kb_event.cnt is 0, it indicates the update is two buttons are pressed at the same time; if previous kb_event.cnt is 1, it indicates the update is another button is pressed with one button held; if previous kb_event.cnt is 3, it indicates the update is one of the three pressed buttons is released.....

kb_event.keycode[0] and kb_event.keycode[1] indicate key values of the two pressed buttons currently. The subsequent keycodes are negligible.

User can manually clear the “kb_event.keycode” before each key scan, so that it can be used to check whether valid update occurs, as shown in the example below.

In the sample code, when kb_event.keycode[0] is not zero, it's considered a button is pressed, but the code won't check further complex cases, such as whether two buttons are pressed at the same time or one of the two pressed buttons is released.

```
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

When “kb_scan_key” is invoked, a basic keyscale flow is shown as below:

- 1) Initial full scan through the whole matrix.

All drive pins output drive level (0). Meanwhile read all scan pins, check for valid level, and record the column on which valid level is read. (The scan_pin_need is used to mark valid column number.)

```
scan_pin_need = kb_key_pressed (gpio);
```

If row-by-row scan is directly adopted without initial full scan through the whole matrix, each time all rows should be scanned at least, even if no button is pressed. To save scan time, initial full scan through the whole matrix can be added, thus it will directly exit keyscale if no button press is detected on any column.

In the “kb_key_pressed” function, all rows output low level, and stabilized level of scan pins will be read after 20us delay. A release_cnt is set as 6; if a detection shows all pressed buttons in the matrix are released, it won’t consider no button is pressed and stop row-by-row scan immediately, but buffers for six frames. If six successive detections show buttons are all released, it will stop row-by-row scan. Thus key debounce processing is realized.

- 2) Scan row by row according to full scan result through the whole matrix.

If button press is detected by full scan, row-by-row scan is started: Drive pins (ROW0~ROW4) output valid drive level row by row; read level on columns, and find the pressed button. Following is related code:

```
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;
    }
}
```

The following methods are used to optimize code execution time for row-by-row scan.

- ❖ When a row outputs drive level, it’s not needed to read level of all columns

(CoL0~CoL5). Since the scan_pin_need marks valid column number, user can read the marked columns only.

- ❖ After a row outputs drive level, a 20us or so delay is needed to read stabilized level of scan pins, and a buffer processing is used to utilize the waiting duration.

The array variable “u32 pressed_matrix[5]” (up to 40 columns are supported) is used to store final matrix keyboard state: pressed_matrix[0] bit0~bit5 mark button state on CoL0~CoL5 crossed with Row0, ……, pressed_matrix[4] bit0~bit5 mark button state on CoL0~CoL5 crossed with Row4.

- 3) Debounce filtering for pressed_matrix[].

```
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);
```

During fast keyscan after wakeup from deepsleep, “numlock_status” equals “KB_NUMLOCK_STATUS_POWERON”; the “filt_en” is set as 0 to skip filtering and fast obtain key values.

In other cases, the “filt_en” is set as 1 to enable filtering. Only when pressed_matrix[] stays the same during two adjacent key scans, but different from the latest valid pressed_matrix[], will the “key_changed” set as 1 to indicate valid update in matrix keyboard.

- 4) Buffer processing for pressed_matrix[].

Push pressed_matrix[] into buffer. When the “read_key” in “kb_scan_key (int numlock_status, int read_key)” is set as 1, the data in the buffer will be read out immediately. When the “read_key” is set as 0, the buffer stores the data without notification to the upper layer; the buffered data won’t be read until the read_key is 1.

In current SDK, the “read_key” is fixed as 1, i.e. the buffer does not take effect actually.

- 5) According to pressed_matrix[], look up the KB_MAP_NORMAL table and return key values.

Corresponding functions are “kb_remap_key_code” and “kb_remap_key_row”.

8.4 Deepsleep wake_up fast keyscan

After Slave enters deepsleep during connection state, it can be woke up by key press. After wakeup, firmware is rebooted; in main_loop following user_init, Slave will first send adv packets, establishes connection, and then sends the key value to BLE Master.

Though this BLE SDK adopts some processing to speed up the deepback (resumption after wakeup from deepsleep), the duration may still reach several hundred milliseconds (e.g. 300ms). To avoid action loss of the wakeup pin, fast keyscan and data buffer are added.

- ✧ Fast keyscan is designed to avoid potential button action loss caused by re-initialization time after MCU reboots and debounce filter processing time during keyscan in main_loop.
- ✧ Data buffer is designed considering valid button data detected in adv state and pushed into BLE TX FIFO will be cleared after entering connection state.

The macro “DEEPBACK_FAST_KEYSCAN_ENABLE” in the “app_config.h” is used to control fast keyscan and data buffer.

```
#define DEEPBACK_FAST_KEYSCAN_ENABLE 1

void deep_wakeup_proc(void)
{
    #if (DEEPBACK_FAST_KEYSCAN_ENABLE)
    if(analog_read(DEEP_ANA_REG0) == CONN_DEEP_FLG) {
        if(kb_scan_key (KB_NUMLOCK_STATUS_POWERON,1) && kb_event.cnt) {
            deepback_key_state = DEEPBACK_KEY_CACHE;
            key_not_released = 1;
            memcpy(&kb_event_cache,&kb_event,sizeof(kb_event));
        }
    }
    #endif
}
```

During initialization key scan is processed before user_init. After it's detected by reading retention analog register that MCU enters deep wakeup from connection state, the “kb_scan_key” is invoked to directly obtain button state of the whole matrix without enabling the debounce filtering. If key scan process shows a button is pressed (button state update is returned, and kb_event.cnt in non-zero value), the “kb_event” variable will be copied to the cache variable “kb_event_cache”.

The “deepback_pre_proc” and “deepback_post_proc” processing are added in keyscan during main_loop.

```
void proc_keyboard (u8 e, u8 *p)
{
    kb_event.keycode[0] = 0;
    int det_key = kb_scan_key (0, 1);

#ifndef DEEPBACK_FAST_KEYSCAN_ENABLE
    if (deepback_key_state != DEEPBACK_KEY_IDLE) {
        deepback_pre_proc(&det_key);
    }
#endif

    if (det_key) {
        key_change_proc();
    }

#ifndef DEEPBACK_FAST_KEYSCAN_ENABLE
    if (deepback_key_state != DEEPBACK_KEY_IDLE) {
        deepback_post_proc();
    }
#endif
}
```

The “deepback_pre_proc” realizes buffer processing of fast keyscan value, as shown below: After connection is established between Slave and Master, if no button state update is detected in a kb_key_scan, the buffered kb_event_cache value will be used as the current newest button state update.

For button release processing, it's needed to check current matrix keyboard state: If there's button pressed, since actual button release generates a release action, it's not needed to add manual release; if current button is released, it's needed to mark that a manual release event should be added, otherwise button may fail to be released since buffered button press event stays valid.

The “deepback_pre_proc” specifies whether manual release is needed. The “deepback_post_proc” will determine whether to push a button release event into BLE TX FIFO accordingly.

8.5 Repeat Key processing

When a button is pressed and held, it's needed to enable repeat key function to repeatedly send the key value with a specific interval.

The “repeat key” function is masked by default. By configuring related macros in the “app_config.h”, this function can be controlled correspondingly.

```
//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

This macro serves to enable or mask the repeat key function. To use this function, first set “KB_REPEAT_KEY_ENABLE” as 1.

2) KB_REPEAT_KEY_INTERVAL_MS

This macro serves to set the repeat interval time. For example, if it's set as 200ms, it indicates when a button is held, kb_key_scan will return an update with the interval of 200ms. Current button state will be available in kb_event.

3) KB_REPEAT_KEY_NUM & KB_MAP_REPEAT

The two macros serve to define current repeat key values:

KB_REPEAT_KEY_NUM specifies the number of keycodes, while the KB_MAP_REPEAT defines a map to specify all repeat keycodes. Note that the keycodes in the KB_MAP_REPEAT must be the values in the KB_MAP_NORMAL.

Following example shows a 6*6 matrix keyboard: by configuring the four macros, eight buttons including UP, DOWN, LEFT, RIGHT, V+, V-, CHN+ and CHN- are set as repeat keys with repeat interval of 100ms, while other buttons are set as non-repeat keys.

```
#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, }
```

User can search for the four macros in the project to locate the code about repeat key.

8.6 Stuck Key processing

Stuck key processing is used to save power when one or multiple buttons of a remote control/keyboard is/are pressed and held for a long time unexpectedly, for example a RC is pressed by a cup or ashtray. If keyscan detects some button is pressed and held, without the stuck key processing, MCU won't enter deepsleep or other low power state since it always considers the button is not released.

Following are two related macros in the app_config.h:

```
//stuck key
#define STUCK_KEY_PROCESS_ENABLE 0
#define STUCK_KEY_ENTERDEEP_TIME 60 //in s
```

By default the stuck key processing function is masked. User can set the "STUCK_KEY_PROCESS_ENABLE" as 1 to enable this function.

The "STUCK_KEY_ENTERDEEP_TIME" serves to set the stuck key time: if it's set as 60s, it indicates when button state stays fixed for more than 60s with some button held, it's considered as stuck key, and MCU will enter deepsleep.

User can search for the macro "STUCK_KEY_PROCESS_ENABLE" to locate related code in the keyboard.c, as shown below:

```
#if (STUCK_KEY_PROCESS_ENABLE)
    u8 stuckKeyPress[ARRAY_SIZE(drive_pins)];
#endif
```

An u8-type array stuckKeyPress[5] is defined to record row(s) with stuck key in current key matrix. The array value is obtained in the function "key_debounce_filter".

Upper-layer processing is shown as below:

```
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

    .....
}
```

For each button state update, when button press is detected (i.e. kb_event.cnt is non-zero value), the “stuckKey_keyPressTime” is used to record the time for the latest button press state.

Processing in the “blt_pm_proc” is shown as below:

```
#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
```

Check whether the latest pressed button is held for more than 60s: if yes, it's considered as stuck key, all row numbers with stuck key will be obtained via the bottom-layer “stuckKeyPress[]”; then modify corresponding PAD wakeup polarity as low level from high level, so that MCU can enter deepsleep and wake up by button release normally (when button is pressed, corresponding drive pin reads high level of 10/11 VCC; after release, the drive pin turns to low level).

9 LED Management

9.1 LED task related functions

Source code about LED management is available in vendor/common/blt_led.c of this BLE SDK for user reference. User can directly include the “vendor/common/blt_led.h” into his C file.

User needs to invoke the following three functions:

```
void device_led_init(u32 gpio, u8 polarity);
int device_led_setup(led_cfg_t led_cfg);
static inline void device_led_process(void);
```

During initialization, the “device_led_init(u32 gpio,u8 polarity)” is used to set current GPIO and polarity corresponding to LED. If “polarity” is set as 1, it indicates LED will be turned on when GPIO outputs high level; if “polarity” is set as 0, it indicates LED will be turned on when GPIO outputs low level.

The “device_led_process” function is added at UI Entry of main_loop. It’s used to check whether LED task is not finished (DEVICE_LED_BUSY). If yes, MCU will carry out corresponding LED task operation.

9.2 LED task configuration and management

9.2.1 LED event definition

The following structure serves to define a 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;
```

- ❖ The unsigned short int type “onTime_ms” and “offTime_ms” specify light on and off time (unit: ms) for current LED event, respectively. The two variables can reach the maximum value 65535.

- ✧ The unsigned char type “repeatCount” specifies blinking times (i.e. repeat times for light on and off action specified by the “onTime_ms” and “offTime_ms”). The variable can reach the maximum value 255.
- ✧ The “priority” specifies the priority level for current LED event.

To define a LED event when the LED always stays on/off, set the “repeatCount” as 255(0xff), set “onTime_ms”/“offTime_ms” as 0 or non-zero correspondingly.

LED event examples:

- 1) Blink for 3s with 1Hz frequency: keep on for 500ms, stay off for 500ms, and repeat for 3 times.

```
led_cfg_t led_event1 = {500,      500,      3,        0x00,  };
```

- 2) Blink for 50s with 4Hz frequency: keep on for 125ms, stay off for 125ms, and repeat for 200 times.

```
led_cfg_t   led_event2  =  {125,     125,     200,      0x00,  };
```

- 3) Always on: onTime_ms is non-zero, offTime_ms is zero, and repeatCount is 0xff.

```
led_cfg_t   led_event3  = {100,     0,       0xff,     0x00,  };
```

- 4) Always off: onTime_ms is zero, offTime_ms is non-zero, and repeatCount is 0xff.

```
led_cfg_t   led_event4  =  {0,       100,     0xff,     0x00,  };
```

- 5) Keep on for 3s, and then turn off: onTime_ms is 1000, offTime_ms is 0, and repeatCount is 0x3.

```
led_cfg_t   led_event5  =  {1000,    0,       3,        0x00,  };
```

The “device_led_setup” can be invoked to deliver a led_event to LED task management.

```
device_led_setup(led_event1);
```

9.2.2 LED event priority

User can define multiple LED events in the SDK, however, only a LED event is allowed to be executed at the same time. No task list is set for the simple LED management: When LED is idle, LED will accept any LED event delivered by invoking the “device_led_setup”. When LED is busy with a LED event (old LED event), if another event (new LED event) comes, MCU will compare priority level of the two LED events; if the new LED event has higher priority level, the old LED event will be discarded and MCU starts to execute the new LED event; if the new LED event has the same or lower priority level, MCU continues executing the old LED event, while

the new LED event will be completely discarded, rather than buffered.

By defining LED events with different priority levels, user can realize corresponding LED indicating effect.

Since inquiry scheme is used for LED management, MCU should not enter long suspend (e.g. 10ms * 50 = 500ms) with latency enabled and LED task ongoing (DEVICE_LED_BUSY); otherwise LED event with small onTime_ms value (e.g. 250ms) won't be responded in time, thus LED blinking effect will be influenced.

```
#define DEVICE_LED_BUSY (device_led.repeatCount)
```

The corresponding processing is needed to add in blt_pm_proc, as shown below:

```
user_task_flg = scan_pin_need || key_not_released || DEVICE_LED_BUSY;  
if(user_task_flg){  
    bls_pm_setManualLatency(0); // manually disable latency  
}
```

User can refer to demo code in current 8258 ble remote project for LED related processing.

10 Blt Software Timer

Telink BLE SDK supplies source code of blt software timer demo for user reference on timer task. User can directly use this timer or modify as needed.

Source code are available in “vendor/common/blt_soft_timer.c” and “blt_soft_timer.h”. To use this timer, the macro below should be set as 1.

```
#define BLT_SOFTWARE_TIMER_ENABLE 0 //enable or disable
```

Since blt software timer is inquiry timer based on system tick, it cannot reach the accuracy of hardware timer, and it should be continuously inquired during main_loop. The blt soft timer applies to the use scenarios with timing value more than 5ms and without high requirement for time error.

Its key feature is: This timer will be inquired during main_loop, and it ensures MCU can wake up in time from suspend and execute timer task. This design is implemented based on “Timer wakeup by Application layer” (**section 4.5 Timer wakeup by Application Layer**).

Current design can run up to four timers, and maximum timer number is modifiable via the macro below:

```
#define MAX_TIMER_NUM 4 //timer max number
```

10.1 Timer initialization

The API below is used for blt software timer initialization:

```
void blt_soft_timer_init(void);
```

Timer initialization only registers “blt_soft_timer_process” as callback function of APP layer wakeup in advance.

```
void blt_soft_timer_init(void)
{
    bls_pm_registerAppWakeupLowPowerCb(blt_soft_timer_process);
}
```

10.2 Timer inquiry processing

The function “blt_soft_timer_process” serves to implement inquiry processing of blt software timer.

```
void blt_soft_timer_process(int type);
```

On one hand, main_loop should always invoke this function in the location as shown in the figure below. On the other hand, this function must be registered as callback function of APP layer wakeup in advance. Whenever MCU is woke up from suspend in advance by timer, this function will be quickly executed to process timer task.

```
3 void main_loop (void)
4 {
5     static u32 tick_loop;
6
7     tick_loop++;
8
9     blt_soft_timer_process(MAINLOOP_ENTRY);
10
11    blt_sdk_main_loop();
12 }
```

The parameter “type” of the “blt_soft_timer_process” indicates two cases to enter this function: If “type” is 0, it indicates entering this function via inquiry in main_loop; if “type” is 1, it indicates entering this function when MCU is woke up in advance by timer.

#define MAIN_LOOP_ENTRY	0
#define CALLBACK_ENTRY	1

The implementation of the “blt_soft_timer_process” is rather complex, and its basic principle is shown as below:

- 1) First check whether there is still user-defined timer in current timer table. If not, directly exit the function and disable timer wakeup of APP layer; if there's timer task, continue the flow.

```
if(!blt_timer.currentNum) {
    bls_pm_setAppWakeupLowPower(0, 0); //disable
    return;
}
```

- 2) Check whether the nearest timer task is reached: if the task is not reached, exit the function; otherwise continue the flow. Since the design will ensure all timers are time-ordered, herein it's only needed to check the nearest timer.

```

if( !blt_is_timer_expired(blt_timer.timer[0].t, now) ) {
    return;
}

```

- 3) Inquire all current timer tasks, and execute corresponding task as long as timer value is reached.

```

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;
            }
        }
    }
}

```

The code above shows processing of timer task function: If the return value of this function is less than 0, this timer task will be deleted and won't be responded; if the return value is 0, the previous timing value will be retained; if the return value is more than 0, this return value will be used as the new timing cycle (unit: us).

- 4) In step 3), if tasks in timer task table change, the previous time sequence may be disturbed, and re-ordering is needed.

```

if(change_flg) {
    blt_soft_timer_sort();
}

```

- 5) If the nearest timer task will be responded within 3s (it can be changed to a value larger than 3s as needed) from now, the response time will be set as wakeup time by APP layer in advance; otherwise APP layer wakeup in advance will be disabled.

```
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 Add timer task

The API below serves to add timer task.

```
typedef int (*blt_timer_callback_t) (void);  
  
int     blt_soft_timer_add(blt_timer_callback_t func, u32  
interval_us);
```

“func”: timer task function.

“interval_us”: timing value (unit: us).

The int-type return value corresponds to three processing methods:

- 1) If the return value is less than 0, this task will be automatically deleted after execution. This feature can be used to control the number of timer execution times.
- 2) If the return value is 0, the old interval_us will be used as timing cycle.
- 3) If the return value is more than 0, this return value will be used as new timing cycle (unit: 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;
}
```

As shown in the implementation code, if timer number exceeds the maximum value, the adding operation will fail. Whenever a new timer task is added, re-ordering must be implemented to ensure timer tasks are time-ordered, while the index corresponding to the nearest timer task should be 0.

10.4 Delete timer task

As introduced above, timer task will be automatically deleted when the return value is less than 0.

Except for this case, the API below can be invoked to specify the timer task to be deleted.

```
int     blt_soft_timer_delete(blt_timer_callback_t func);
```

10.5 Demo

For Demo code of blt soft timer, please refer to “TEST_USER_BLT_SOFT_TIMER” in 8258 feature.

```
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;
}
```

Initialization:

```
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);
```

Four timer tasks are defined with different features:

- 1) gpio_test0: Toggle once for every 23ms.
- 2) gpio_test1: Switch between two timers of 7ms/17ms.
- 3) gpio_test2: Delete itself after 5s, which can be implemented by invoking “blt_soft_timer_delete(&gpio_test2)” or “return -1”.

-
- 4) gpio_test3: Toggle once for every 27ms.

Shared under NDA

11 IR

11.1 PWM Driver

Please refer to PWM section in Telink 8258 Datasheet to help understanding PWM driver.

By operating registers, hardware configurations for PWM are very simple. To improve execution efficiency and save code size, related APIs, implemented via “static inline function”, are defined in the “pwm.h” (no need of c file).

11.1.1 PWM id and pin

8x5x supports up to 12-channel PWM: PWM0 ~ PWM5 and PWM0_N ~ PWM5_N. Six-channel PWM is defined in driver:

```
typedef enum {
    PWM0_ID = 0,
    PWM1_ID,
    PWM2_ID,
    PWM3_ID,
    PWM4_ID,
    PWM5_ID,
} pwm_id;
```

Only six channels PWM0~PWM5 are configured in software, while the other six channels PWM0_N~PWM5_N are inverted output of PWM0~PWM5 waveform.

For example: PWM0_N is inverted output of PWM0 waveform. When PWM0 is high level, PWM0_N is low level; When PWM0 is low level, PWM0_N is high level.

Therefore, as long as PWM0~PWM5 are configured, PWM0_N~PWM5_N are also configured.

For 8x5x family, IC pins corresponding to 12-channel PWM are shown as below:

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

The “void gpio_set_func(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func)” serves to set specific pin as PWM function.

- ✧ “pin”: GPIO pin corresponding to actual PWM channel
- ✧ “func”: Must set as corresponding PWM function, i.e. AS_PWM0 ~ AS_PWM5_N in the “GPIO_FuncTypeDef”.

```
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;
```

E.g. To use PA2 as PWM0:

```
gpio_set_func(GPIO_PA2, AS_PWM0);
```

11.1.2 PWM clock

The API below serves to set PWM clock.

```
void pwm_set_clk(int system_clock_hz, int pwm_clk)
```

- ✧ “system_clock_hz”: current system clock “CLOCK_SYS_CLOCK_HZ” (This macro is defined in the “app_config.h”).
- ✧ “pwm_clk”: clock to be configured.

Note that “system_clock_hz” must be integral multiples of “pwm_clk” so as to get the wanted PWM clock via frequency division.

To make PWM output waveforms as accurate as possible, it’s recommended to set “pwm_clk” as “CLOCK_SYS_CLOCK_HZ”, i.e.

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_HZ);
```

For example, suppose current system clock “CLOCK_SYS_CLOCK_HZ” is 16000000, by the setting above, both PWM clock and system clock equal 16MHz.

The setting below can be followed to set PWM clock as 8MHz irrespective of system clock (CLOCK_SYS_CLOCK_HZ=16000000, 24000000 or 32000000).

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, 8000000);
```

11.1.3 PWM cycle and duty

PWM waveform consist of Signal Frames. For a PWM Signal Frame, “cycle” and “cmp” need to be configured via related APIs.

- 1) `void pwm_set_cycle(pwm_id id, unsigned short cycle_tick)`

This API serves set PWM “cycle”. Unit: number of PWM clock cycles.

- 2) `void pwm_set_cmp(pwm_id id, unsigned short cmp_tick)`

This API serves set PWM “cmp”. Unit: number of PWM clock cycles.

- 3) `void pwm_set_cycle_and_duty(pwm_id id, unsigned short cycle_tick, unsigned short cmp_tick)`

To improve efficiency, this API below can be used instead.

After PWM “cycle” and “cmp” are configured, PWM duty cycle equals PWM “cmp”/PWM “cycle”.

The figure below shows PWM waveforms configured by `pwm_set_cycle_and_duty(PWM0_ID, 5, 2)`. For each Signal Frame of PWM0, “cycle” is 5 PWM clock cycles, “cmp” (high-level duration) is two PWM clock cycles, thus PWM cycle is $2/5=40\%$.

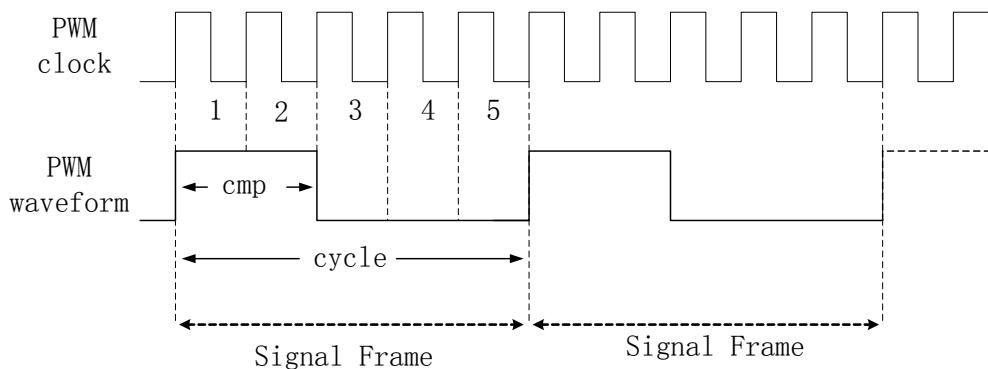


Figure11-1 PWM cycle & duty cycle

For PWM0~PWM5, by default hardware will set PWM output as high level (cmp) followed by low level during a frame cycle. To obtain PWM waveform with low level followed by high level, following two methods apply:

- 1) Use corresponding PWM0_N~PWM5_N (inverted output of PWM0 ~ PWM5).
- 2) Use the API “`static inline void pwm_revert(pwm_id id)`” to invert PWM0~PWM5 waveform.

Suppose current PWM clock is 16MHz, to set PWM cycle and duty cycle for PWM0 as 1ms and 50% respectively,

Use the two APIs:

```
pwm_set_cycle(PWM0_ID , 16000)
```

```
pwm_set_cmp (PWM0_ID , 8000)
```

or Directly use this API:

```
pwm_set_cycle_and_duty(PWM0_ID, 16000, 8000);
```

11.1.4 PWM revert

- 1) `void pwm_revert(pwm_id id)`

This API serves to invert PWM0~PWM5 waveform.

- 2) `void pwm_n_revert(pwm_id id)`

This API serves to invert PWM0_N~PWM5_N waveform.

11.1.5 PWM start and stop

- 1) `void pwm_start(pwm_id id) ;`

This API serves to enable (start) specified PWM channel.

- 2) `void pwm_stop(pwm_id id) ;`

This API serves to disable (stop) specified PWM channel.

11.1.6 PWM mode

For 8x5x family, PWM supports up to five modes:

PWM0~PWM5 support Normal mode (Continuous mode), while only PWM0 supports Counting mode, IR mode, IR FIFO mode and IR DMA FIFO mode.

```
typedef enum{
    PWM_NORMAL_MODE      = 0x00,
    PWM_COUNT_MODE       = 0x01,
    PWM_IR_MODE          = 0x03,
    PWM_IR_FIFO_MODE     = 0x07,
    PWM_IR_DMA_FIFO_MODE = 0xF,
} pwm_mode;
```

The API below serves to set PWM mode:

```
void pwm_set_mode(pwm_id id, pwm_mode mode)
```

For more details about PWM mode, please refer to section 8.5 in 8258 Datasheet.

11.1.7 PWM pulse number

The API below serves to set pulse number, i.e. number of Signal Frames, for output waveform of specified PWM channel.

```
void pwm_set_pulse_num(pwm_id id, unsigned short pulse_num)
```

This API is only used for Counting mode, IR mode, IR FIFO mode and IR DMA FIFO mode, but not applies to Normal mode with continuous pulses.

11.1.8 PWM interrupt

Basic knowledge about Telink MCU interrupt:

Interrupt “status” is state flag bit generated by hardware interrupt request of certain IRQ source, and it does not depend on software setting. No matter whether “mask” is enabled, interrupt request will always set corresponding IRQ “status” to 1. Generally, “status” can be cleared to 0 by writing it with “1”.

Interrupt response: When CPU receives an interrupt request (IRQ) from certain IRQ source, it will determine whether to respond to the IRQ. If yes, firmware pointer PC will jump to interrupt handling part “irq_handler”.

To enable interrupt response, please make sure all “mask” bits corresponding to current IRQ are enabled. One IRQ may correspond to multiple “mask” bits which are the relation of logic “And”. IRQ request won’t trigger interrupt response unless all of its related “mask” bits are enabled.

PWM driver in the “register_8258.h” only involves the following IRQ sources.

```
#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),
};

;
```

The eight IRQ sources listed in the enum correspond to core_7b0 BIT<0:7> (“mask”) / core_7b1 BIT<0:7> (“status”).

In the figure below, PWM0 works in IR mode, duty cycle of Signal Frame is 50%, pulse number (i.e. Signal Frame number) for each IR task is 3. This figure will help to illustrate the three types for PWM IRQ “status”.

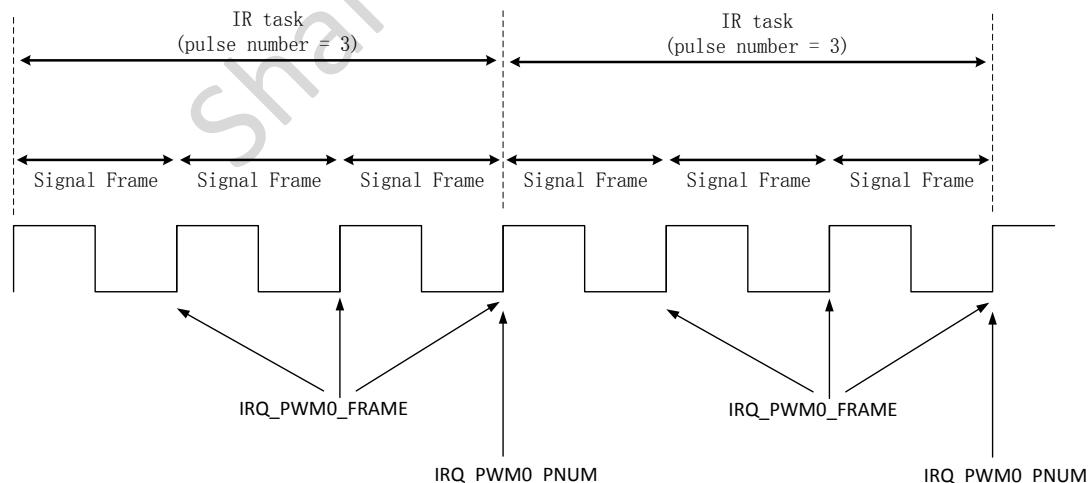


Figure11-2 PWM interrupt

- 1) IRQ_PWMn_FRAME(n=0,1,2,3,4,5) for PWM0~PWM5: After each signal frame, PWM#n (n=0~5) will generate a frame-done IRQ (Interrupt Request) signal “IRQ_PWMn_FRAME”.

As shown in the figure above, six frame-done IRQ signal are generated at the end of each PWM0 Signal Frame.

- 2) IRQ_PWM0_PNUM: In Counting mode and IR mode, PWM0 will generate a Pnum IRQ signal “IRQ_PWM0_PNUM” after completing a group of Signal Frames (pulse number is determined by the API `pwm_set_pulse_num`).

As shown in the figure above, PWM0 will generate a Pnum IRQ signal at the end of a pulse group containing three Signal Frames.

- 3) IRQ_PWM0_IR_DMA_FIFO_DONE

In IR DMA FIFO mode, PWM0 will generate an IR waveform send done IRQ signal “IRQ_PWM0_IR_DMA_FIFO_DONE”, after all PWM waveforms configured in DMA are sent.

As described above, IRQ request won't trigger interrupt response unless all of its related “mask” bits are enabled.

E.g. For the “FLD_IRQ_PWM0_PNUM”, it's needed to enable three “mask” bits.

- 1) Enable “mask” of FLD_IRQ_PWM0_PNUM, i.e. core_7b0:

```
reg_pwm_irq_mask |= FLD_IRQ_PWM0_PNUM;
```

Generally, to avoid false triggering of interrupt response, it's needed to clear previous “status” before enabling “mask”.

```
reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;
```

- 2) Enable PWM “mask” in MCU system interrupt, i.e. core_640 BIT<14>.

```
#define reg_irq_mask           REG_ADDR32(0x640)
enum{
    .....
    FLD_IRQ_SW_PWM_EN =      BIT(14), //irq_software | irq_pwm
    .....
};
```

Following is the method to enable this “mask”:

```
reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
```

- 3) Enable MCU global interrupt “mask”, i.e. `irq_enable()`.

11.1.9 PWM phase

```
void pwm_set_phase(pwm_id id, unsigned short phase)
```

This API serves to set delay time before PWM is started.

“phase”: delay time. Unit: number of PWM clock cycles.

Generally it can be set as 0 (no delay).

11.1.10 API for IR DMA FIFO mode

The following APIs are dedicated for IR DMA FIFO mode. Please refer to PWM demo code in the SDK.

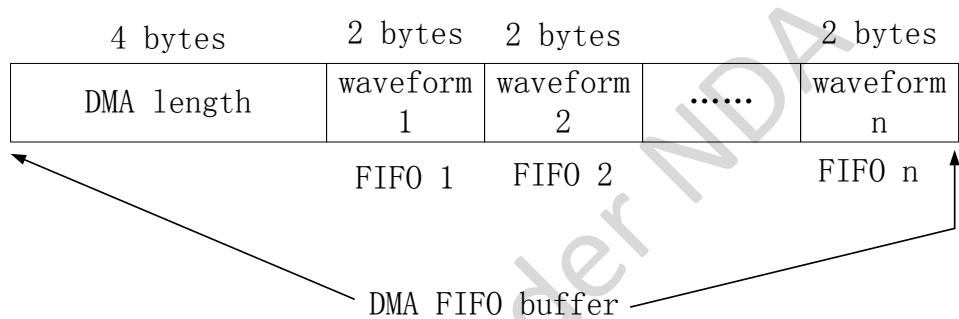


Figure11-3 DMA FIFO buffer for IR DMA FIFO mode

DMA FIFO buffer is a data block defined in SRAM, and the “DMA length” of the first 4 bytes indicates the number of bytes occupied by FIFO. As shown above, DMA length = $n \times 2$.

There are n FIFOs, and each FIFO has two bytes to indicate one PWM waveform. For the 8x5x family, “ n ” can be up to 256.

After DMA data buffer takes effect, PWM HW module will send out waveform 1 ~ waveform n successively.

After all waveforms are sent, PWM is stopped automatically and IRQ_PWM0_IR_DMA_FIFO_DONE is triggered.

11.1.10.1 Configuration for DMA FIFO

Each DMA FIFO uses 2 bytes (16 bits) to configure one PWM waveform. When the API below is invoked, 2-byte DMA FIFO data will be returned.

```
unsigned short pwm_config_dma_fifo_waveform(int carrier_en,  
    Pwm0Pulse_SelectDef pulse, unsigned short pulse_num);
```

By configuring the three parameters “carrier_en”, “pulse” and “pulse_num”, the PWM output waveform contains “pulse_num” PWM pulses (Signal Frames).

As shown in the configuration format in the 8258 Datasheet, BIT(15) specifies Signal Frame format of PWM waveform, and corresponds to the “carrier_en” of this API.

- ❖ When “carrier_en” is 1, PWM will output carrier pulse.
- ❖ When “carrier_en” is 0, PWM will output Signal Frames with low level only.

The “pulse_num” specifies the number of Signal Frames for current PWM waveform.

The “pulse” supports the definition below:

```
typedef enum{  
    PWM0_PULSE_NORMAL = 0,  
    PWM0_PULSE_SHADOW = BIT(14),  
} Pwm0Pulse_SelectDef;
```

- ❖ When “pulse” is PWM0_PULSE_NORMAL, Signal Frame uses the configuration of the API “pwm_set_cycle_and_duty”.
- ❖ When “pulse” is PWM0_PULSE_SHADOW, Signal Frame uses the configuration of PWM shadow mode.

PWM shadow mode enables more flexibility for PWM waveform configuration in IR DMA FIFO mode. Related API is shown as below, and its configuration is consistent with pwm_set_cycle_and_duty.

```
void pwm_set_pwm0_shadow_cycle_and_duty(unsigned short  
    cycle_tick,  
    unsigned short cmp_tick);
```

11.1.10.2 Set DMA FIFO buffer

After DMA FIFO buffer is configured, the API below should be invoked to set the starting address of the buffer to DMA module.

```
void pwm_set_dma_address(void * pdat);
```

11.1.10.3 Start and Stop for IR DMA FIFO mode

After DMA FIFO buffer is prepared, the API below should be invoked to start sending PWM waveforms.

```
void pwm_start_dma_ir_sending(void);
```

After all PWM waveforms in DMA FIFO buffer are sent, the PWM module will be stopped automatically. The API below can be invoked to manually stop the PWM module in advance.

```
void pwm_stop_dma_ir_sending(void);
```

11.2 IR Demo

Please refer to IR demo code of the project “8258_ble_remote” in the SDK.

The macro “REMOTE_IR_ENABLE” in the “app_config.h” should be enabled.

11.2.1 PWM mode selection

As required by IR transmission, PWM output needs to switch at specific time with small error tolerance of switch time accuracy to avoid incorrect IR.

As described in Link Layer timing sequence ([section 3.2.4](#)), Link Layers uses system interrupt to process brx event. (In the new SDK, adv event is processed in the main_loop and does not occupy system interrupt time.) When IR is about to switch PWM output soon, if brx event related interrupt comes first and occupies MCU time, the time to switch PWM output may be delayed, thus to result in IR error.

Therefore IR cannot use PWM Normal mode.

For Telink 826x BLE SDK, PWM IR mode is used to implement IR. Please refer to “826x BLE SDK handbook”.

Since 826x PWM IR mode only supports data pre-storage of two IR fifos, if PWM Signal Frame takes very short time, even shorter than BLE interrupt handling time in irq_handler, PWM waveform may be delayed, thus it brings a risk for IR mode.

The 8x5x family introduces an extra IR DMA FIFO mode which is not supported by 826x. In IR DMA FIFO mode, since FIFO can be defined in SRAM, more FIFOs are available, which can effectively solve the shortcoming of PWM IR mode above.

IR DMA FIFO mode supports pre-storage of multiple PWM waveforms into SRAM. Once DMA is started, no software involvement is needed. This can save

frequent SW processing time, and avoid PWM waveform delay caused by simultaneous response to multiple IRQs in interrupt system.

Only PWM0 with IR DMA FIFO mode can be used to implement IR. Therefore, in HW design, IR control GPIO must be PWM0 pin or PWM0_n pin.

11.2.2 Demo IR protocol

The figure below shows demo IR protocol in the SDK.

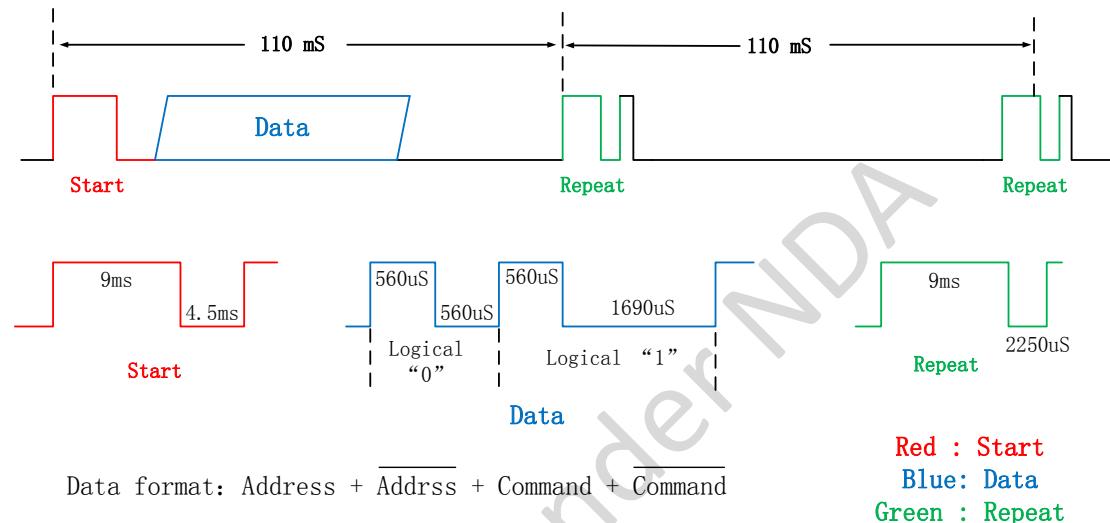


Figure11-4 Demo IR protocol

11.2.3 IR timing design

The figure below shows basic IR timing abased demo IR protocol and feature of IR DMA FIFO mode.

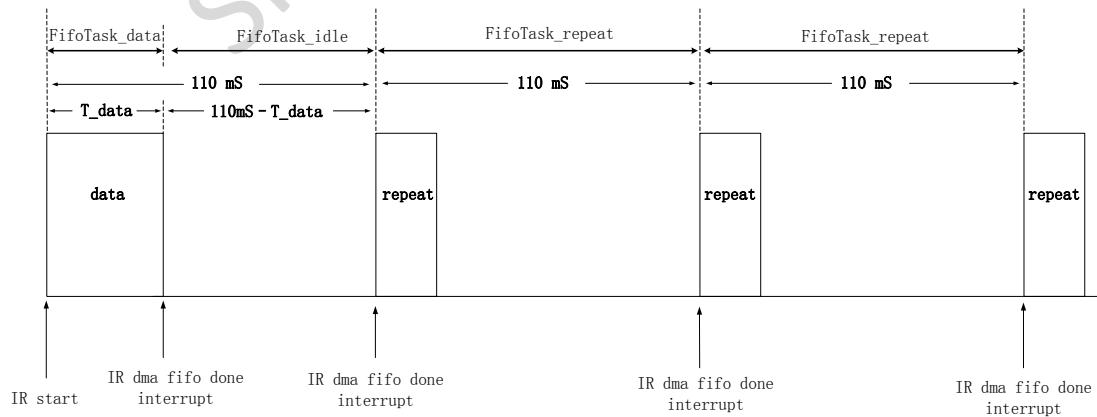


Figure11-5 IR timing 1

In IR DMA FIFO mode, a complete task is defined as FifoTask. Herein the processing of IR repeat signal adopts the method of “add repeat one by one”, i.e. the macro below is defined as 1.

```
#define ADD_REPEAT_ONE_BY_ONE 1
```

When a button is pressed to trigger IR transmission, IR is disassembled to FifoTasks as shown in the Figure11-5.

- 1) After IR is started, run FifoTask_data to send valid data. The duration of FifoTask_data, marked as T_data, is not certain due to the uncertainty of data. After FifoTask_data is finished, trigger IRQ_PWM0_IR_DMA_FIFO_DONE.
- 2) In interrupt function of IRQ_PWM0_IR_DMA_FIFO_DONE, start FifoTask_idle phase to send signal without carrier and it lasts for a duration of (110ms – T_data). This phase is designed to guarantee the time point the first FifoTask_repeat is 110ms later after IR is started. After FifoTask_idle is finished, trigger IRQ_PWM0_IR_DMA_FIFO_DONE.
- 3) In interrupt function of IRQ_PWM0_IR_DMA_FIFO_DONE, start the first FifoTask_repeat. Each FifoTask_repeat lasts for 110ms. By adding FifoTask_repeat in corresponding interrupt function, IR repeat signals can be sent continuously.
- 4) The time point to stop IR is not certain, and it depends on the time to release the button. After the APP layer detects key release, as long as FifoTask_data is correctly completed, IR transmission is finished by manually stopping IR DMA FIFO mode.

Following shows some optimization steps for the IR timing design above.

- 1) Since FifoTask_repeat timing is fixed, and there are many DMA fifos in IR DMA FIFO mode, multiple FifoTask_repeat of 110ms can be assembled into one FifoTask_repeat*n, so as to reduce the number of times to process IRQ_PWM0_IR_DMA_FIFO_DONE in SW.

Correponding to the processing of “ADD_REPEAT_ONE_BY_ONE” macro defined as 0, the Demo herein assembles five IR repeat signals into one FifoTask_repeat*5. User can further optimize it according to the usage of DMA fifos.

- 2) Based on step 1), combine FifoTask_ilde and the first “FifoTask_repeat*n” to form “FifoTask_idle_repeat*n”.

The figure below shows IR timing after optimization.

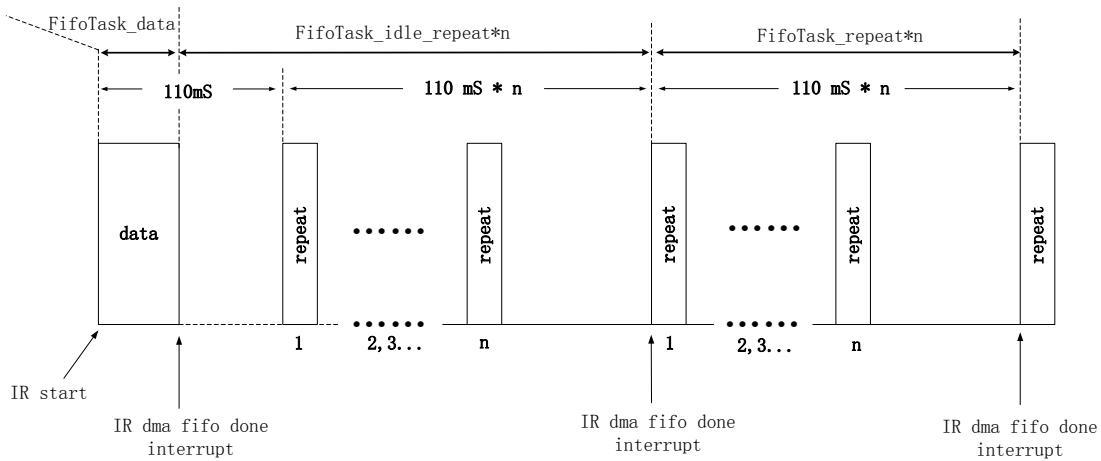


Figure 11-6 IR timing 2

As per the IR timing design above, corresponding code in SW flow is shown as below:

At IR start, invoke the function “ir_nec_send”, enable FifoTask_data, and use interrupt to control the following flow. In the interrupt when FifoTask_data is finished, enable FifoTask_idle. In the interrupt when FifoTask_idle is finished, enable FifoTask_repeat. Before manually stopping IR DMA FIFO mode, FifoTask_repeat is executed continually.

```

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_send();
}

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_send();
            }
        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 initialization

11.2.4.1 rc_ir_init

IR initialization function is shown as below. Please refer to demo code in the SDK.

```
void rc_ir_init(void)
```

IR initialization is not invoked in user initialization, but triggered by IR key press.

```
if(!ir_hw_initialed){
    ir_hw_initialed = 1;
    rc_ir_init();
}
```

Due to the existence of deepsleep retention mode, MCU may repeatedly wake up from deepsleep retention; since IR is not frequently used, by using IR key press to trigger IR initialization, the number of times to execute rc_ir_init can be reduced effectively.

IR initialization supports the two types below:

- ✧ Hardware register setting (e.g. pwm_set_mode): Must re-configure after each deepsleep retention wake_up.
- ✧ Initialization of SRAM logic variables (e.g. waveform_logic_0_1st): If defined in “data/bss” sector, the variables must be re-configured; if defined in “retention_data” sector, it’s only needed to set the variables once.

In the SDK demo, since the variables are placed in the “data/bss” sector, re-configuration is needed each time. During power optimization, user can determine whether to define them as “retention_data” according to the usage of SRAM retention area.

The variable “ir_hw_initaled” must be put in the “data/bss” sector rather than the “retention_data”, so as to ensure the function “rc_ir_init” is executed again after each deepsleep retention wake_up.

11.2.4.2 IR hardware configuration

Following shows the demo code.

```
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;
```

Since only PWM0 supports ID DMA FIFO mode, PB3 corresponding to PWM0_N is selected herein.

In the demo, IR carrier frequency is 38K, cycle is 26.3us, and duty cycle is 1/3. The API “pwm_set_cycle_and_duty” should be used to configure the cycle and duty cycle. Since all FifoTasks share the same carrier frequency, the carrier of 38K can meet the configuration requirement, and PWM shadow mode is not needed.

DMA FIFO buffer is *T_dmaData_buf*.

Enable system interrupt mask “*FLD_IRQ_SW_PWM_EN*”.

Clear system status “*FLD IRQ_PWM0_IR_DMA_FIFO_DONE*”.

11.2.4.3 IR variable initialization

Related variables in the SDK demo includes waveform_start_bit_1st, waveform_start_bit_2nd, and etc.

As introduced in IR timing design, FifoTask_data and FifoTask_repeat should be configured.

Start signal = 9ms carrier signal + 4.5ms low level signal (no carrier). The “*pwm_config_dma_fifo_waveform*” is invoked to configure the two corresponding DMA FIFO data.

```
//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;
```

The method also applies to configure stop signal, repeat signal, data logic “1” signal, and data logic “0” signal.

11.2.5 FifoTask configuration

11.2.5.1 FifoTask_data

As per demo IR protocol, to send a cmd (e.g. 7), first send start signal, i.e. 9ms carrier signal + 4.5ms low level signal (no carrier); then send “address+ ~address+ cmd + ~cmd”. In the demo code, address is 0x88.

When sending the final bit of “~cmd”, logical “0” or logical “1” always contains some non-carrier signals at the end. If “~cmd” is not followed by any data, there may be a problem on Rx side: Since there’s no boundary to differentiate carrier, the FW does not know whether the non-carrier signal duration of the final bit is 560us or 1690us, and fails to recognize whether it’s logical “0” or logical “1”.

To solve this problem, the Data signal should be followed by a “stop” signal which is defined as 560us carrier signal + 500us non-carrier signal.

Thus, the FifoTask_data mainly contains the three parts below:

- 1) start signal: 9ms carrier signal + 4.5ms low level signal (no carrier)
- 2) data signal: address+ ~address+ cmd + ~cmd
- 3) stop signal: 560us carrier signal + 500us non-carrier signal

The code below serves to configure DMA Fifo buffer and start IR transmission.

```
//// 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

As introduced in IR timing design, FifoTask_idle lasts for the duration “110mS – T_data”.

Record the time when FifoTask_data starts:

```
ir_send_ctrl.sending_start_time = clock_time();
```

Then calculate FifoTask_idle time in the interrupt triggered when FifoTask_data is finished:

```
110mS – (clock_time() - ir_send_ctrl.sending_start_time)
```

Demo code:

```

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);
    
```

Please pay attention to time unit switch. As introduced in Clock module, System Timer frequency used in software timer is fixed as 16MHz. Since PWM clock is derived from system clock, user needs to consider the case with system clock rather than 16MHz (e.g. 24MHz, 32MHz).

FifoTask_idle does not send PWM waveform, which can be considered to continually send non-carrier signal. It can be implemented by setting the first parameter “carrier_en” of the API “pwm_config_dma_fifo_waveform” as 0.

```

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

As per Demo IR protocol, repeat signal is 9ms carrier signal + 2.25ms non-carrier signal.

Similar to the processing of FifoTask_data, the end of repeat signal should be followed by 560us carrier signal as stop signal.

As introduced in IR timing design, repeat signal lasts for 110ms, so the duration of non-carrier signal after the 560us carrier signal should be:

$$110\text{mS} - 9\text{mS} - 2.25\text{mS} - 560\mu\text{s} = 99190\mu\text{s}$$

The code below shows the configuration for a complete repeat signal.

```

//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

By simple superposition in DMA Fifo buffer, “FifoTask_repeat*n” and “FifoTask_idle_repeat*n” can be implemented on the basis of FifoTask_idle and FifoTask_repeat.

11.2.6 Check IR busy status in APP layer

In the Application layer, user can use the variable “ir_send_ctrl.is_sending” to check whether IR is busy sending data or repeat signal.

As shown in the demo code below, when IR is busy, MCU cannot enter suspend.

```
if( ir_send_ctrl.is_sending)  
{  
    bls_pm_setSuspendMask(SUSPEND_DISABLE);  
}
```

12 Other Modules

12.1 External capacitor for 24MHz crystal

As shown in the Figure12-1, two locations of matching capacitor for 24MHz crystal are C19 and C20.

By default, the SDK uses internal capacitor of 8x5x (i.e. cap corresponding to ana_8a<5:0>) as matching capacitor of 24MHz crystal oscillator, while C19 and C20 are not connected with cap actually. By adopting this solution, matching capacitor is measurable and adjustable in Telink jig system to achieve optimal frequency point value of final application product.

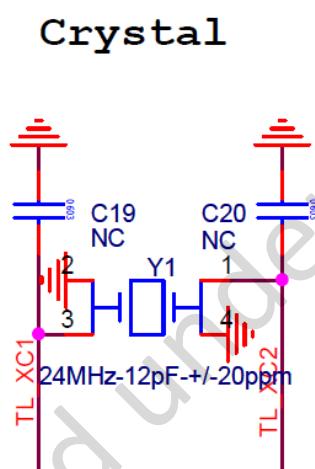


Figure12-1 24MHz crystal circuit

If it's needed to use external soldered capacitor C19 and C20 as matching capacitor of 24MHz crystal oscillator instead, the API below should be invoked at the beginning of main function and before "cpu_wakeup_init" function.

```
static inline void blc_app_setExternalCrystalCapEnable(u8 en)
{
    blt_miscParam.ext_cap_en = en;
}
```

As long as this API is invoked before the "cpu_wakeup_init", the SDK will automatically implement all operations (e.g. disable internal matching capacitor and stop reading frequency offset calibration value).

12.2 Select 32kHz clock source

The SDK uses the embedded 32kHz RC oscillator (named 32k RC) by default, the accuracy of which is worse for applications with long suspend or deep retention due to larger error of 32k RC. The default max duration of long connection currently supported by 32k RC cannot exceed 3s (The limit also applies to external 32kHz crystal in current SDK). Once connection duration exceeds 3s, ble_timing error will cause inaccurate time slot to receive packet, and thus will easily lead to RF retry operation, higher power consumption and even disconnection.

To implement lower connection current as well as more accurate clock timing in low power sleep, user can select to use external 32kHz crystal (named 32k Pad). Current SDK version supports this mode.

User only needs to invoke either of the two APIs below before the function “cpu_wakeup_init” at the beginning of main function, so as to select 32k RC (default) or 32k Pad.

```
void blc_pm_select_internal_32k_crystal(void);  
void blc_pm_select_external_32k_crystal(void);
```

12.3 PA

To use RF PA, please refer to “drivers/8258/rf_pa.c” and “rf_pa.h”.

First enable the macro below, which is disabled by default.

```
#ifndef PA_ENABLE  
#define PA_ENABLE 0  
#endif
```

Invoke PA initialization during system initialization.

```
void rf_pa_init(void);
```

In this initialization, “PA_TXEN_PIN” and “PA_RXEN_PIN” are set as GPIO output mode, and initial status is “output 0”. User needs to define GPIOs corresponding to TX and RX PA.

```
#ifndef PA_TXEN_PIN  
#define PA_TXEN_PIN GPIO_PB2  
#endif  
  
#ifndef PA_RXEN_PIN  
#define PA_RXEN_PIN GPIO_PB3  
#endif
```

```
#endif
```

And “void app_rf_pa_handler(int type)” is registered as callback function of PA. Acutally this function processes the three PA status below: disable PA, enable TX PA, and enable RX PA.

<code>#define PA_TYPE_OFF</code>	0
<code>#define PA_TYPE_TX_ON</code>	1
<code>#define PA_TYPE_RX_ON</code>	2

User only needs to invoke the “rf_pa_init” above; “app_rf_pa_handler” is registered as the bottom-layer callback, so that it will be automatically invoked to process correspondingly in various BLE states.

12.4 PHY test

PhyTest, i.e. PHY test, means RF performance test of BLE controller.

For more details, please refer to “Core_v5.0” (Vol 2/Part E/7.8.28~7.8.30 and Vol 6/Part F “Direct Test Mode”).

12.4.1 PhyTest API

Source code of PhyTest is assembled in library file, and related API is available for user. Please refer to “stack/ble/phy/ble_test.h”.

```
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);
```

During initialization, the “blc_phy_initPhyTest_module” is invoked to configure the PhyTest module.

After the APP layer triggers PhyTest, the “blc_phy_setPhyTestEnable(1)” is invoked to enable PhyTest mode.

In the demo “8258_feature_test” of the SDK, phytst is directly triggered during initialization to start.

In the demo “8258 ble remote”, a key combination is set to trigger PhyTest

mode.

PhyTest mode is a special mode, which does not allow normal BLE function. Once entering PhyTest mode, advertising and connection are disabled. Therefore, DO NOT trigger PhyTest when executing normal BLE function.

When PhyTest is finished, MCU will automatically reboot either after power cycle or after invoking the “blc_phy_setPhyTestEnable (0) ”.

The “blc_phy_isPhyTestEnable” is used to check whether PhyTest is triggered currently. As shown in the demo code, this API is used to implement low power management, and it's not allowed to enter low power in PhyTest mode.

When two-wire UART mode (PHYTEST_MODE_THROUGH_2_WIRE_UART) is used by PhyTest, the setting below can be followed during initialization:

```
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” is used to analyze and execute cmd sent from Host.

“phy_test_2_wire_tx_to_uart” is used to deliver corresponding result and data to Host.

12.4.2 PhyTest demo

12.4.2.1 Demo1: 8258_feature_test

In the “app_config.h” of the demo “8258_feature_test”, test mode is set as “TEST_BLE_PHY”, as shown below:

```
#define FEATURE_TEST_MODE           TEST_BLE_PHY
```

PhyTest supports three test modes corresponding to different physical interface and test command format. “PHYTEST_MODE_DISABLE” indicates PhyTest is disabled.

```
#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
```

Select test mode for PhyTest:

```
#if  (FEATURE_TEST_MODE == TEST_BLE_PHY)
    #define BLE_PHYTEST_MODE      PHYTEST_MODE_THROUGH_2_WIRE_UART
#endif
```

- 1) Define as 2-wire UART mode:

```
#define BLE_PHYTEST_MODE      PHYTEST_MODE_THROUGH_2_WIRE_UART
```

- 2) Define as HCI mode with UART interface

```
#define BLE_PHYTEST_MODE      PHYTEST_MODE_OVER_HCI_WITH_UART
```

- 3) HCI mode with USB interface: not added yet in current SDK version.

By compiling the “8258_feature_test”, the generated bin file can pass PhyTest. Please refer to the demo code and get the hang of related interfaces.

12.4.2.2 Demo2: 8258_ble_remote

In the “app_config.h” of the demo “8258_ble_remote”, “BLE_PHYTEST_MODE” is set as “PHYTEST_MODE_DISABLE” by default, so that PhyTest related code is masked.

```
#define BLE_PHYTEST_MODE      PHYTEST_MODE_DISABLE
```

- 1) As shown in the definition below, PhyTest is enabled in remote control application, and test mode is set as 2-wire UART mode.

```
#define BLE_PHYTEST_MODE      PHYTEST_MODE_THROUGH_2_WIRE_UART
```

- 2) As shown in the definition below, PhyTest is enabled in remote control application, and test mode is set as HCI mode with UART interface.

```
#define BLE_PHYTEST_MODE      PHYTEST_MODE_OVER_HCI_WITH_UART
```

By compiling the “8258_ble_remote”, the generated bin file can pass PhyTest. Please refer to the demo code and get the hang of related interfaces.

12.4.2.3 Adjust PhyTest parameters

If PhyTest fails, the parameters including length of RF packet preamble and bandwidth for packet reception are adjustable.

Length of RF packet preamble is adjustable by writing the register core_402. As shown in the demo “8258_feature_test”, length of RF packet preamble is adjusted during initialization.

```
blc_phy_initPhyTest_module();
blc_phy_setPhyTestEnable( BLC_PHYTEST_ENABLE );
blc_phy_preamble_length_set(11);
```

In the demo “8258_ble_remote”, length of RF packet preamble is adjusted after PhyTest is triggered manually.

```
void app_phystest_init(void)
{
    blc_phy_initPhyTest_module();
    blc_phy_preamble_length_set(11);
    ....
```

User is allowed to configure related registers in the demo code as needed, e.g. write the analog register “ana_ac” to adjust bandwidth for packet reception.

12.5 EMI

12.5.1 EMI Test

During EMI Test, it's needed to invoke interfaces related to rfdrv, e.g. rf_drv_init(). All of these interfaces are assembled in library. API declaration is viewable in the “rf_drv.h”.

EMI Test supports four test modes: Carrier only mode (send carrier only), TX continue mode (continuously send Carrier with data), RX mode, as well as three TX burst modes with different packet payload types.

```
Struct test_list_sate_list[] = {
    {0x01, emicarrieronly}, //carrier only mode
```

```
{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 initialization setting

- 1) Before EMI test, first it's needed to invoke the "rf_drv_init()" to initialize RF.

```
void rf_drv_init (RF_ModeTypeDef rf_mode);
```

"rf_mode" serves to select RF mode. However, in current SDK version, only RF_MODE_BLE_1M is supported.

- 2) After RF initialization is configured, it's needed to invoke the "app_emi_init()" function to initialize Host interface commands.

```
write_reg32(0x408,0x29417671 );//rf access code  
write_reg8(0x840005,tx_cnt);//tx_cnt initialized to 0  
write_reg8(0x840006,run);//run cmd 1: start test item, 0: end test item  
write_reg8(0x840007,cmd_now);//cmd: configure test item  
write_reg8(0x840008,power_level);//power_level: Tx power init  
write_reg8(0x840009,chn);//chn: RF channel init  
write_reg8(0x84000a,mode);//mode: RF mode init,  
//only support BLE 1M in current SDK  
write_reg8(0x840004,0); //4-byte RSSI statistic average initialized to 0  
write_reg32(0x84000c,0); //4-byte rx packet statistic Rx number initialized to 0
```

- 3) Invoke the "app_rf_emi_test_start()" in main_loop to poll test items.

12.5.1.2 Power level and Channel

During EMI test, user can configure "RF power level" and "RF channel", which will determine energy and channel for packet transmission.

RF Power is configurable according to "RF_PowerTypeDef rf_power_Level_list[60]".

Note that actual Tx power may have slight difference from the configured value,

corresponding to different boards or antenna matching parameters. User can invoke the two functions below to set power.

- 1) static void rf_set_power_level_index_singletone (RF_PowerTypeDef level); //set power level in Carrier only or Tx continue mode
- 2) void rf_set_power_level_index (RF_PowerTypeDef level); // set power level in tx burst modes

“level”: Power level as per the enum “RF_PowerTypeDef”.

RF Channel: Set frequency as (2400+chn) MHz. ($0 \leq chn \leq 100$)

For example, to set channel as 2405MHz, “chn” should be set as 5. The function below can be invoked.

```
void rf_set_channel (signed char chn, unsigned short set);
```

“chn”: RF channel.

“set”: set as 0.

12.5.1.3 EMI Carrier Only

For Carrier only mode, user only needs to directly invoke the “emicarrieronly ()” function below.

```
void emicarrieronly(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr, signed char rf_chn)
```

“rf_mode”: RF_MODE_BLE_1M

“pwr”: Power level. (See **section 12.5.1.2**).

“rf_chn”: RF channel. (See **section 12.5.1.2**).

12.5.1.4 TX Continue mode

In TX Continue mode, data in carrier are updated via the “rf_continue_mode_loop ()” function to ensure the data are random numbers.

User only needs to invoke the “emi_con_prbs9 ()” function to enter TX Continue mode.

```
void emi_con_prbs9(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr, signed char rf_chn)
```

“rf_mode”: RF_MODE_BLE_1M

“pwr”: Power level. (See **section 12.5.1.2**).

“rf_chn”: RF channel. (See [section 12.5.1.2](#)).

In TX Continue mode, the “emi_con_prbs9 ()” will invoke the “rf_emi_tx_continue_setup()” function to implement related setting (e.g. rf_mode, power level, chn), and invoke the “rf_continue_mode_loop()” function to update data in carrier.

12.5.1.5 EMI TX Burst

TX Burst mode supports three sub-modes with different packet payload types, including “PRBS9 packet payload”, “00001111b packet payload ” and “10101010b packet payload”. User can directly invoke “emitxprbs9()”, “emitx55()” or “emitx0f()” to enter corresponding sub mode.

```
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);
```

“rf_mode”: RF_MODE_BLE_1M

“pwr”: Power level. (See [section 12.5.1.2](#)).

“rf_chn”: RF channel. (See [section 12.5.1.2](#)).

The “emitxprbs9()”/“emitx55()”/“emitx0f()” will invoke the “rf_emi_tx_brust_setup” function to set TX Burst initialization, and invoke the “rf_emi_tx_brust_loop()” function to start packet transmission and update payload.

```
void rf_emi_tx_brust_setup(RF_ModeTypeDef rf_mode,unsigned char power_level,signed char rf_chn,unsigned char pkt_type)
```

“rf_mode”: RF_MODE_BLE_1M

“power_level”: Power level. (See [section 12.5.1.2](#)).

“rf_chn”: RF channel. (See [section 12.5.1.2](#)).

“pkt_type”: payload type. 0 - PRBS9, 1- 00001111b, 2 - 10101010b.

12.5.1.6 EMI RX

The “emirx()” serves to enter RX mode.

The “rf_emi_rx_loop()” in main_loop() serves to inquire whether RX has received data, and get statistic packet count number and RSSI value of received data.

```
void emirx(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr, signed char rf_chn) ;  
  
void rf_emi_rx_loop(void);  
  
“rf_mode”: RF_MODE_BLE_1M  
“pwr”: Power level. (See section 12.5.1.2).  
“rf_chn”: RF channel. (See section 12.5.1.2).
```

12.5.1.7 Set Host configuration parameters

Run:

0	Default	1	Start test
---	---------	---	------------

Cmd:

1	CarrierOnly	2	ContinuePRBS9	3	RX
4	TXBurst(PRBS9)	5	TXBurst(0x55)	6	TXBurst(0x0f)

For Power and channel, please refer to **section 12.5.1.2**.

Mode:

0	Reserve	1	Ble_1M
---	---------	---	--------

The default setting of these parameters are (mode=1; power=0; channel=2; cmd=1), i.e. send carrier only in ble_1M mode with 2402MHz frequency and 10.4dBm Tx power.

12.5.2 EMI Test Tool

“EMI Test Tool” can be used to implement EMI test. The tool interface is shown as below.

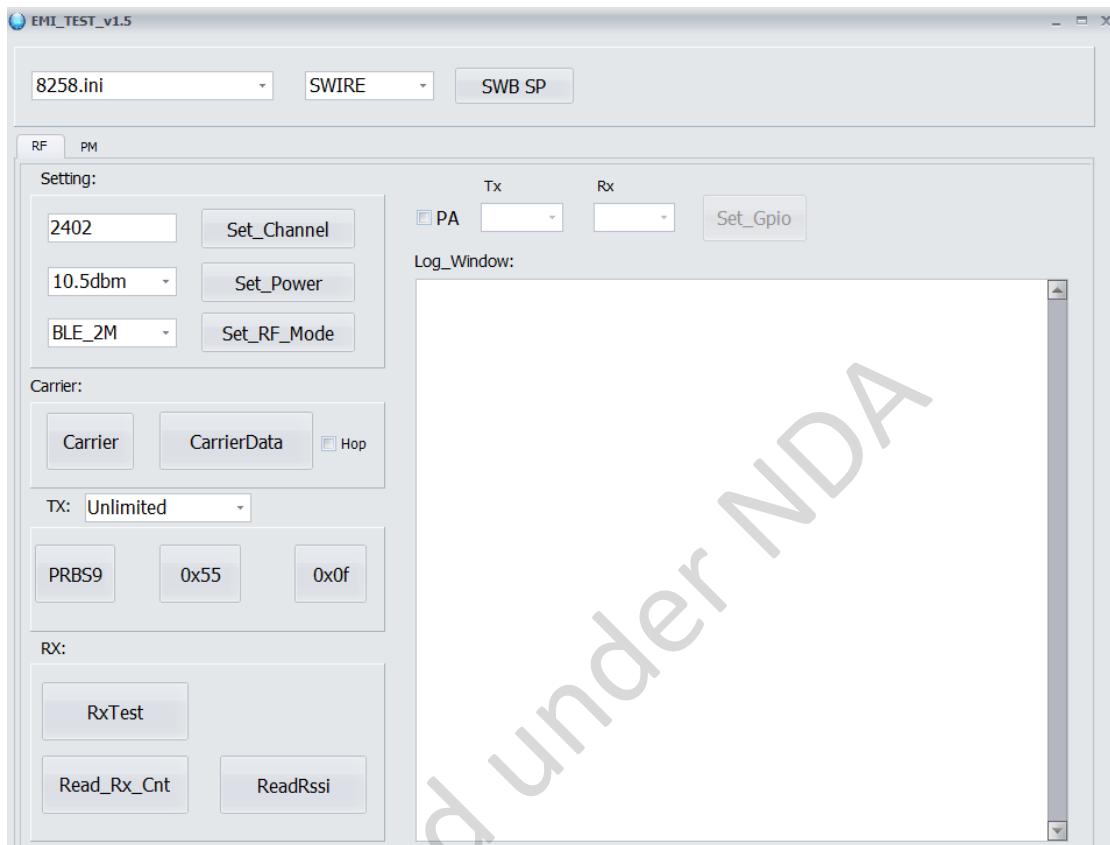


Figure12-2 EMI test tool

Step 1: Select chip part number.

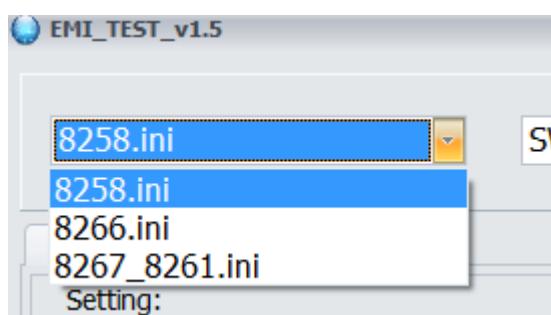


Figure12-3 Select chip type

Step 2: User can select hardware connection method as needed.

When “Swire” is selected, if system clock is 16MHz or below, it's needed to implement “SWB SPEED” (click “SWB SP”) on Wtcdb tool to ensure normal communication.

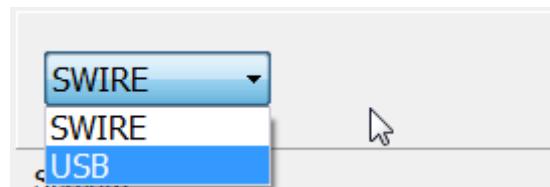


Figure12-4 Select data bus

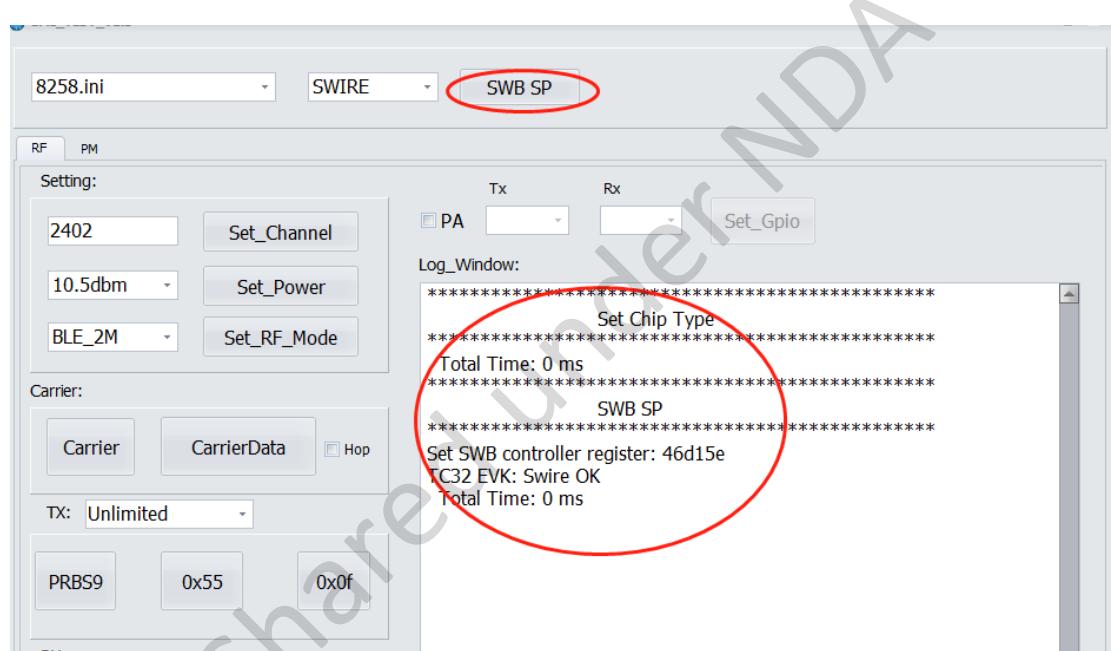


Figure12-5 Swire synchronization operation

Step 3: Set “chn”.

Input frequency (e.g. 2402) in the corresponding box and then click the “Set_Channel” button.

The log window will show “Swire OK” to indicate normal communication, as shown below.

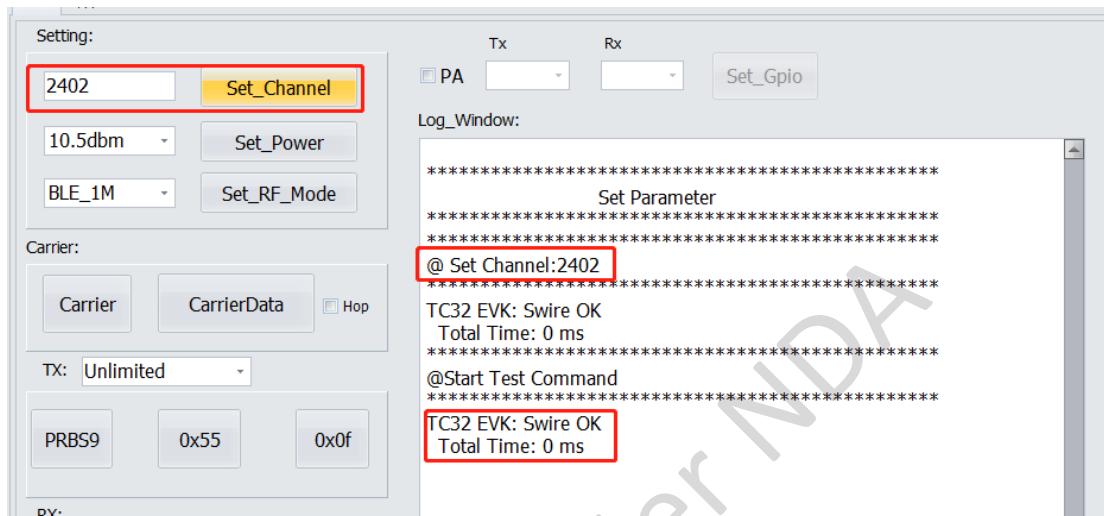


Figure12-6 Set channel

Step 4: Select power level and BLE mode via the corresponding drop-down box, and then click the “Set_Power”/“Set_RF_Mode” button.

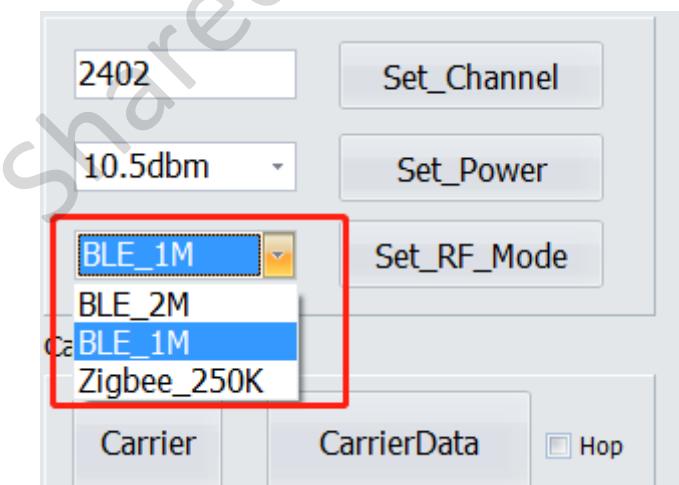


Figure12-7 Select RF mode

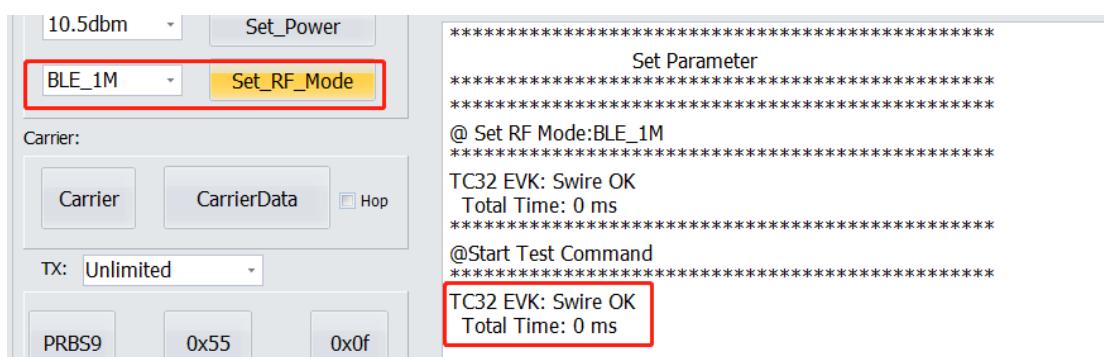


Figure12-8 Interface after RF mode setting

Step 5: Click “Carrier”/“CarrierData”/“RxTest”/“PRBS9”/“0x55”/“0x0f” to enter corresponding test mode.

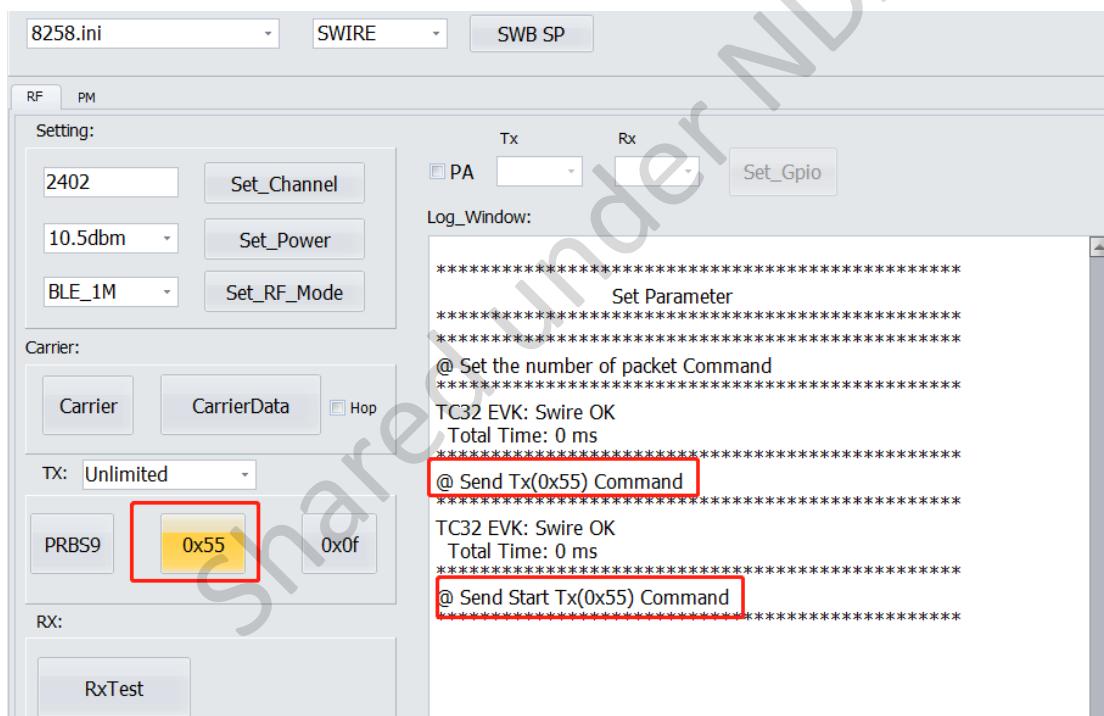


Figure12-9 Select test mode

Step 6: In TX mode, user can select to send 1000 packets or unlimited packets.

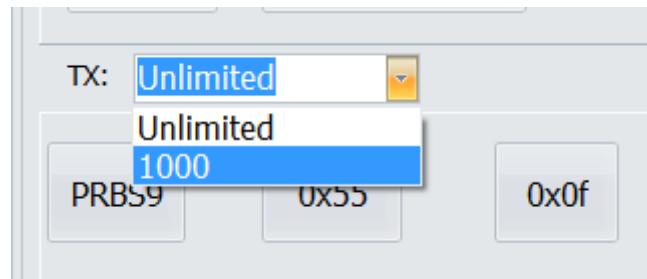


Figure12-10 Set TX packet number

Step 7: In RX mode, number of received packets can be read by clicking the “Read_Rx_Cnt” button, while current RSSI can be obtained by clicking the “ReadRssi” button, as shown below.

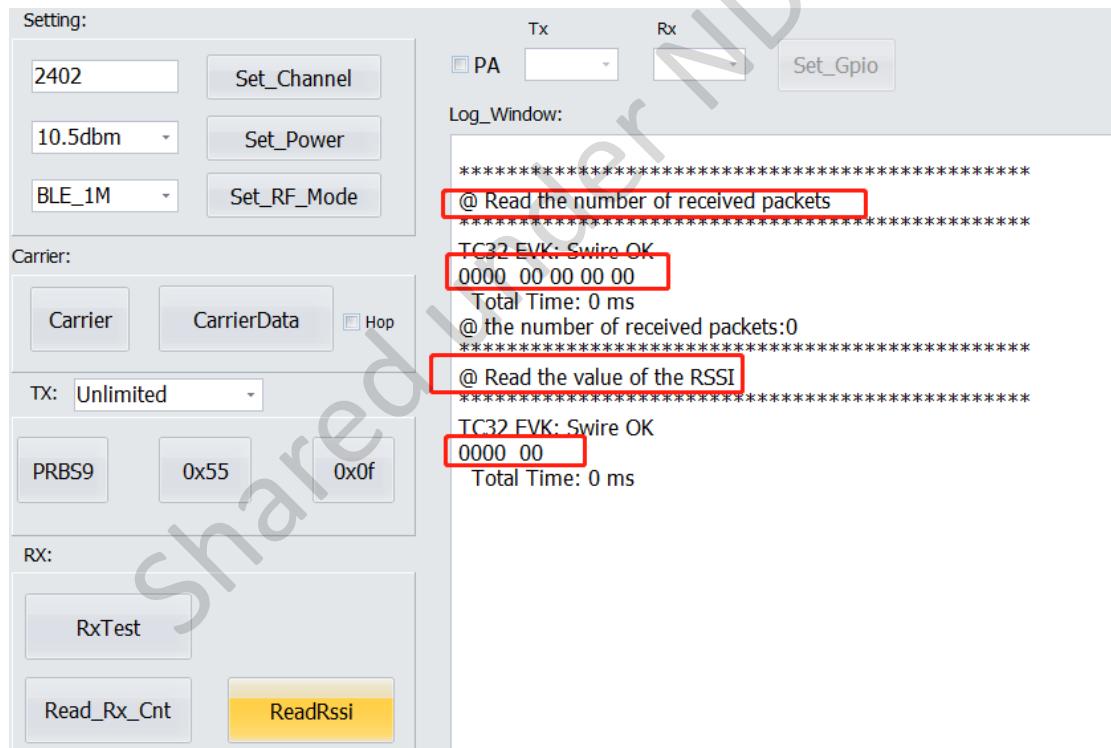


Figure12-11 Read RX packet number and RSSI

13 Appendix

Appendix 1: crc16 algorithm

```
unsigned shortcrc16 (unsigned char *pD, int len)

{
    static unsigned short poly[2]={0, 0xa001};

    unsigned short crc = 0xffff;
    unsigned char ds;

    int i,j;

    for(j=len; j>0; j--)
    {
        unsigned char ds = *pD++;
        for(i=0; i<8; i++)
        {
            crc = (crc >> 1) ^ poly[(crc ^ ds) & 1];
            ds = ds >> 1;
        }
    }

    return crc;
}
```

14 FreeRTOS SDK support appendix

This chapter will introduce the FreeRTOS usage and notice and will only be applicable to the FreeRTOS SDK, general FreeRTOS knowledge is not in the scope of this document, please refer to <http://www.freertos.org>.

14.1 FreeRTOS configuration

14.1.1 System Clock

System Clock is defined by CLOCK_SYS_CLOCK_HZ from vendor/8258_ble_remote/app_config.h. (freertos/config/FreeRTOSConfig.h configCPU_CLOCK_HZ is currently not taking effect)

```
#define CLOCK_SYS_CLOCK_HZ      16000000
```

14.1.2 Tick Rate

User can adjust the Tick Rate within the range of 100 to 1000, the maximum should be 1000.

The number higher than 1000 would affect the system resource to handle the task switching, and the result couldn't be guaranteed.

```
#define configTICK_RATE_HZ    1000
```

14.2 System Initialization

Main.c has the system initialization sample code, please keep it the same and especially the following two statement:

```
#if (BATT_CHECK_ENABLE)
    adc_hw_initialized = 0;
#endif
    blt_dma_tx_rptr = 0;
```

task_restore() is used to restore the task, parameter retention is used to indicate if this restore is called from DeepSleep retention mode.

```
void task_restore(int retention);
```

The following statement is the example to restore the task after DeepSleep.

```
if(deepRetWakeUp){
    DEBUG_GPIO(GPIO_CHN4, 0);
    DEBUG_GPIO(GPIO_CHN0, 1);
    task_restore(1); // never reach here
```

14.3 Task Creation

Following is the example to create the BLE stack and UI task:

```
xTaskCreate(&proto_task, "tProto", 128, (void*)0, TASK_PROT_PRIORITY, &handle_proto_task);  
xTaskCreate(&ui_task, "tUI", configMINIMAL_STACK_SIZE, (void*)0, TASK_UI_PRIORITY, &handle_ui);
```

tProto is the mandatory and main task to handle the BLE protocol, it should never be deleted. BLE stack's task priority should be set to the highest priority.

tUI task will be executed in every wakeup, so all other periodic operations could be arranged inside the ui_task().

IRQ shouldn't be enabled before calling vTaskStartScheduler() to start the task scheduler.