Telink BLE Multi-connection

SDK Developer Handbook

AN-20050601-E1

Ver.0.1.0

2020/09/21

Keyword

Multi-connection

Brief

This document is the development guide for Telink Multi-connection BLE SDK version 1.1.0, applicable to 8x5x series.

Published by

Telink Semiconductor

Bldg 3, 1500 Zuchongzhi Rd,

Zhangjiang Hi-Tech Park, Shanghai, China

Legal Disclaimer

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

telinkcnsupport@telink-semi.com

Revision History

Version	Change Description
V0.1.0	Initial release.

Table of Contents

Revision History....2
Table of Contents....3
List of Figures....6
List of Tables....10
1. SDK Introduction....11
1.1 Software Architecture....12
1.1.1 main.c....13
1.1.2 app_config.h....14
1.1.3 application file....14
1.1.4 BLE stack entry....15
1.2 Applicable IC....15
1.3 Software Bootloader Introduction....16
1.4 Library Introduction....17
1.5 Demo Introduction....18
1.5.1 M4S3 demo/M1S1 demo....19
1.5.2 Feature demo....20
2. Basic Modules....22
2.1 MCU Address Space....22
2.1.1 MCU Address Space Allocation....22
2.1.2 MCU Address Space Access....22
2.1.3 SDK FLASH Space Allocation....22
2.2 Clock Module....23
2.3 GPIO Module....24
3. BLE Module....25
3.1 BLE SDK Software Architecture....25
3.1.1 Standard BLE SDK Software Architecture....25
3.1.2 Telink BLE SDK Software Architecture....26
3.2 Link Layer....29
3.2.1 Connection Number & Connection Handle....29
3.2.2 Link Layer State Machine....31
3.2.3 Link Layer timing....35

3.2.4 Link Layer TX FIFO & RX FIFO 39
3.2.5 Controller event 47
3.2.6 MTU and DLE Concept and Usage 53
3.2.7 2M PHY 56
3.2.8 Channel Selection Algorithm #2 57
3.2.9 Link Layer API 57
3.3 L2CAP 71
3.3.1 Register L2CAP Data Processing Function 71
3.3.2 Update Connection Parameters 72
3.4 ATT & GATT 75
3.4.1 GATT basic unit Attribute 75
3.4.2 Attribute and ATT Table 77
3.4.3 GATT Service Security 84
3.4.4 Attribute PDU & GATT API 85
3.5 GAP 99
3.5.1 GAP Initialization 99
3.5.2 GAP Event 99
3.6 GATT Data processing 104
3.6.1 Master receiving ATT data processing 104
3.6.2 Slave receiving ATT data processing 104
3.7 SMP 105
3.7.1 SMP Security Level 105
3.7.2 SMP Parameter Configuration 107
3.7.3 SMP security request configuration 112
3.7.4 SMP binding information description 115
3.8 Custom Pair 117
3.9 Device Manage 121
4. Low Power Management 126
4.1 Low Power Driver 126
4.1.1 Low Power Mode 126
4.1.2 Low-power wake-up source 127
4.1.3 Low-power mode entry and wake-up 129
4.1.4 Process after low power consumption wake-up 130

4.2 Low Power Management 133
4.2.1 BLE PM Initialization 133
4.2.2 BLE PM for Link Layer 133
4.2.3 API blc_pm_setSuspendMask 134
4.2.4 API blc_pm_setWakeupSource 135
4.2.5 PM Software Processing Flow 136
4.2.6 API blc_pm_getSystemWakeupTick 138
4.3 Precautions for GIPO Wakeup 138
5. Low Battery Detect 140
5.1 Importance of low power detect 140
5.2 Implementation of low battery detect 140
5.2.1 Precautions for low battery detect 141
5.2.2 API Low Battery Detect API 143
6. Audio 148
6.1 Audio Initialization 148
6.2 Audio Data Processing 148
6.3 Decompression algorithm 149
7. OTA 151
8. Button Scan 152
9. LED Management 153
10. BLT Software timer 154
11. IR 155
12. Other Modules 156
12.1 24M crystal external capacitor 156
12.2 32K clock source selection 156
12.3 PA 157
12.4 PhyTest 158
12.4.1 PhyTest API 158
12.4.2 PhyTest demo 158
12.5 EMI 159
12.5.1 EMI Test 159
12.5.2 EMI Test Tool 162
13. Appendix 167

List of Figures

Figure 1-1 Multi-connection System Diagram....11
Figure 1-2 SDK Structure....12
Figure 1-3 Demo Project....13
Figure 1-4 8258_m4s3 Demo Project File Architecture....13
Figure 1-5 Bootloader File....16
Figure 1-6 cstartup Option....17
Figure 1-7 Project Library Option....18
Figure 1-8 SDK Library....18
Figure 1-9 M4S3 Demo Project....20
Figure 1-10 M1S1 Demo Project....20
Figure 1-11 Feature Demo....21
Figure 2-1 MAC and Calibration Information Default FLASH Storage Address....23
Figure 3-1 BLE SDK Standard Architecture....25
Figure 3-2 HCI Data Interaction of Host and Controller....26
Figure 3-3 BLE Multiple Connection Controller Architecture....27
Figure 3-4 Telink BLE Multiple Connection Whole Stack Architecture....28
Figure 3-5 M1S1 Advertising and Slave Switching....32
Figure 3-6 M1S1 Scanning Master Switching....33
Figure 3-7 M4S3 Advertising and Slave Switching....34
Figure 3-8 M4S3 Scanning and Master Switching....34
Figure 3-9 Status Indicators....36
Figure 3-10 Timing Sequence of M4S3 1A2B....36
Figure 3-11 Timing Sequence of M4S3 1B2A....37
Figure 3-12 Timing Sequence of M4S3 1B2B....37
Figure 3-13 Timing Sequence of M4S3 1C2C....38
Figure 3-14 Timing Sequence of M4S3 1F2H....39
Figure 3-15 Timing Sequence of M4S3 1E2F....39
Figure 3-16 Default Setting of TX FIFO....41
Figure 3-17 Buffer Status of Master Using DLE while Slave not....42
Figure 3-18 Buffer Status When Client using 3 Masters and 2 Slaves....43
Figure 3-19 TX Buffer of Single Connection....43

Figure 3-20 RX Buffer Setting 44
Figure 3-21 RX Buffer of Single Connection 45
Figure 3-22 RX Overflow Diagram 1 46
Figure 3-23 RX Overflow Diagram 2 46
Figure 3-24 Controller Event 48
Figure 3-25 Host + Controller Structure 48
Figure 3-26 Packet Format of DISCONNECTION_COMPLETE 49
Figure 3-27 Packet Format of READ_REMOTE_VER_INFO_COMPLETE 50
Figure 3-28 Packet Format of CONNECTION_COMPLETE 51
Figure 3-29 Packet Format of ADVERTISING_REPORT 51
Figure 3-30 Packet Format of CONNECTION_UPDATE_COMPLETE 51
Figure 3-31 MTU and DLE 53
Figure 3-32 ATT Packet Format 53
Figure 3-33 Link Layer Packet Format 54
Figure 3-34 Data Channel PDU 54
Figure 3-35 Protocol STACK BROADCAST PACKET FORMAT 58
Figure 3-36 Advertising Event in BLE Protocol Stack 60
Figure 3-37 Four Broadcast Events of BLE Protocol Stack 61
Figure 3-38 BLE L2CAP Architecture and ATT Packet Module 71
Figure 3-39 Connection Para update Req Format in BLE Protocol Stack 72
Figure 3-40 conn para update request and response Information when Receiving Packets 73
Figure 3-41 conn para update rsp Format in BLE Protocol Stack 74
Figure 3-42 ll conn update req information when Receiving Packet 75
Figure 3-43 Attributes Make GATT Service 76
Figure 3-44 BLE SDK Attribute Table 77
Figure 3-45 Master reads hidInformation's BLE packet capture 80
Figure 3-46 Write Request in BLE Protocol Stack 81
Figure 3-47 Write Command in BLE Protocol Stack 81
Figure 3-48 Execute Write Request in BLE Protocol Stack 81
Figure 3-49 Service/Attribute Layout 83
Figure 3-50 Local Device Responds to a Service Request 84
Figure 3-51 ATT Permission Definition 85
Figure 3-52 Read by Group Type Request/Read by Group Type Response Example 86

Figure 3-53 Find by Type Value Request/Find by Type Value Response 88
Figure 3-54 Read by Type Request/Read by Type Response 88
Figure 3-55 Find information request/Find information response 89
Figure 3-56 Read Request/Read Response 89
Figure 3-57 Read Blob Request/Read Blob Response 90
Figure 3-58 Exchange MTU Request/Exchange MTU Response 90
Figure 3-59 Write Request/Write Response 92
Figure 3-60 Write Long Characteristic Values 93
Figure 3-61 Handle Value Notification in BLE Spec 94
Figure 3-62 Handle Value Indication in BLE Spec 95
Figure 3-63 Handle Value Confirmation in BLE Spec 97
Figure 3-64 Trigger Event of SMP_PARING_BEAGIN 100
Figure 3-65 callback Example 105
Figure 3-66 Local Device Pairing Status 105
Figure 3-67 Pairing Disable in Packet Capturing 107
Figure 3-68 Rules for Using Out-of-ban and MITM Flag for LE Legacy Pairing 109
Figure 3-69 Different Key Generating Methods Based on Different IO Referencing 109
Figure 3-70 Paring Peer Trigger in Packet Capturing 114
Figure 3-71 Paring ConnTrigger in Packet Capturing 115
Figure 3-72 Bonding Information Format 116
Figure 3-73 Slave Mac Table 119
Figure 3-74 conn_dev_list array definition 122
Figure 3-75 connection completed event handle 122
Figure 3-76 service discovery 123
Figure 3-77 Connection Complete Event Processing char_handle in Functions 124
Figure 3-78 char_handle in app_service_discovery （） 124
Figure 3-79 char_handle in dev_char_info_store_peer_att_handle() 125
Figure 4-1 8x5x MCU Hardware Wake-up Source 128
Figure 4-2 Sleep Mode Wakeup Work Flow 131
Figure 4-3 Timing Sequence of M1S1 in ADV Status 133
Figure 4-4 M1S1 Suspend for Scan for Only Scan 133
Figure 4-5 suspend for connection 134
Figure 6-1 Data Decompression 149

Figure 6-2 Decompression Algorithm Data 150
Figure 12-1 24M Crystal Schematics 156
Figure 12-2 EMI test tool 162
Figure 12-3 Choose SoC 163
Figure 12-4 Choose Data Bus 163
Figure 12-5 Swire SP 163
Figure 12-6 Set Channel 164
Figure 12-7 Set RF Mode 164
Figure 12-8 Set RF Mode Interface 165
Figure 12-9 Set Test Mode 165
Figure 12-10 Set TX Packet Number 166
Figure 12-11 RX Packet Number and RSSI 166

List of Tables

Table 1-1 8x5x Resource 15
Table 1-2 8x5x Supporting Modes 15
Table 3-1 Maximum Supported Master/Slave Number of Libraries 29
Table 3-2 m1s1 Connection Handle 30
Table 3-3 m4s3 Connection Handle 31
Table 3-4 M1S1 Link Player Status 33
Table 3-5 M4S3 Link Layer Status 34
Table 3-6 Return Value of blc_llms_setAdvData 59
Table 3-7 Return Value of blc_llms_setScanRspData 59
Table 3-8 Return Value of advFilterPolicy 63
Table 3-9 Return Value of blc_llms_setAdvEnable 63
Table 3-10 Return Value of scanFilter_policy 65
Table 3-11 Return Value of blc_llms_setScanEnable 66
Table 3-12 Return Value of blc_llms_createConnection 68
Table 3-13 Return Value of blc_llms_disconnect 69
Table 3-14 Return Value when Adding Equipment to Whitelist 70
Table 3-15 Return Value of Exchange MTU Request 91
Table 3-16 Return Value of Handle Value Notification 94
Table 3-17 Return Value of Handle Value Indication 96
Table 3-18 Input Parameter Combination of blc_smp_configSecurityRequestSending 113
Table 4-1 Low Power Mode 126

1. SDK Introduction

This BLE SDK provides BLE master + slave development demos, users can develop their own applications based on these demos. For Demo's software and hardware usage environment, application introduction, operation instructions, demo video and corresponding code, bin file, etc., please download from the following gitlab link:

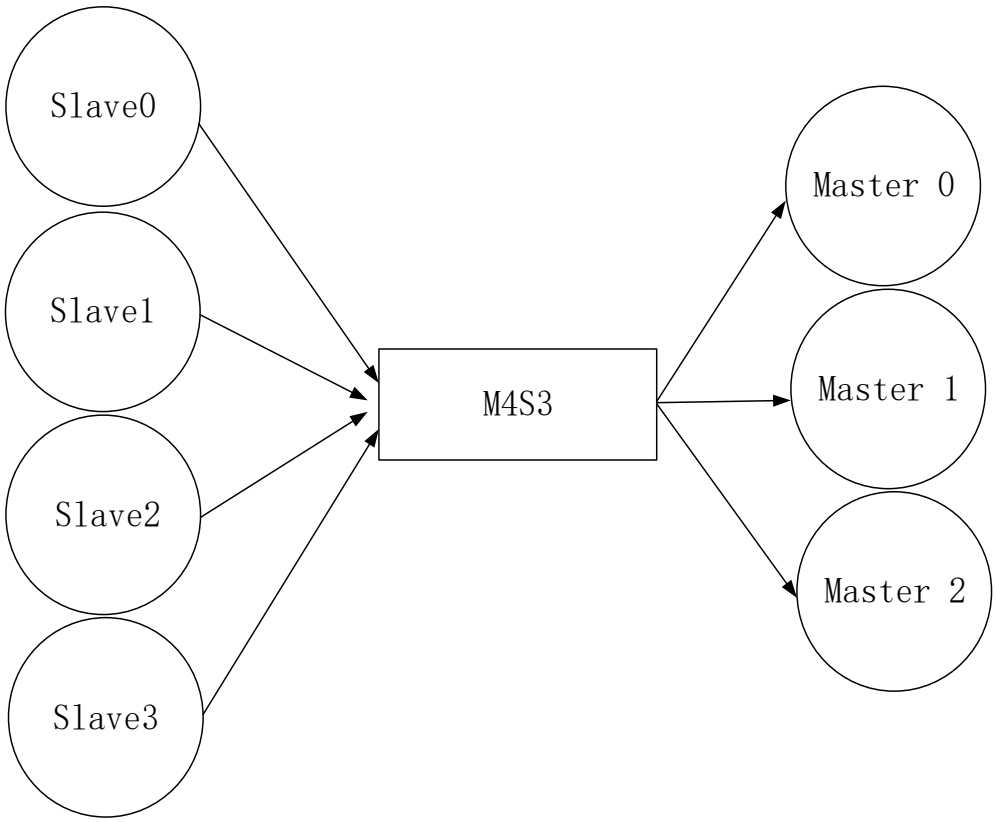
Gitlab:http://192.168.48.36/sdk_app/ble/telink_kite_ble_multi_connection_sdk/tree/master/doc

The general Kite BLE SDK provided by Telink before 2019 is Single Connection SDK (Single Master or Single Slave), the corresponding Single Connection SDK handbook is AN_19011501-E4_Telink Kite BLE SDK Developer Handbook.pdf

Telink provides Kite Multiple Connection BLE SDK since 2020. Multiple Connection here refers to Multiple Master and Multiple Slave, such as 4 Master 3 Slave (abbreviated as M4S3), 2 Master 2 Slave (abbreviated as M2S2), 1 Master 1 Slave (abbreviated as M1S1).

Note: These names are called by the role of the device itself, e.g., 4 Master 3 Slave, which means that the local device is connected to 4 slaves and 3 masters. As shown below:

Figure 1-1 Multi-connection System Diagram



The biggest difference between this SDK and the previous Single Master or Single Slave is that the Link Layer part of the BLE Stack is a brand new Link Layer design. At the same time, the Host layer also has some changes, mainly to achieve multiple connections.

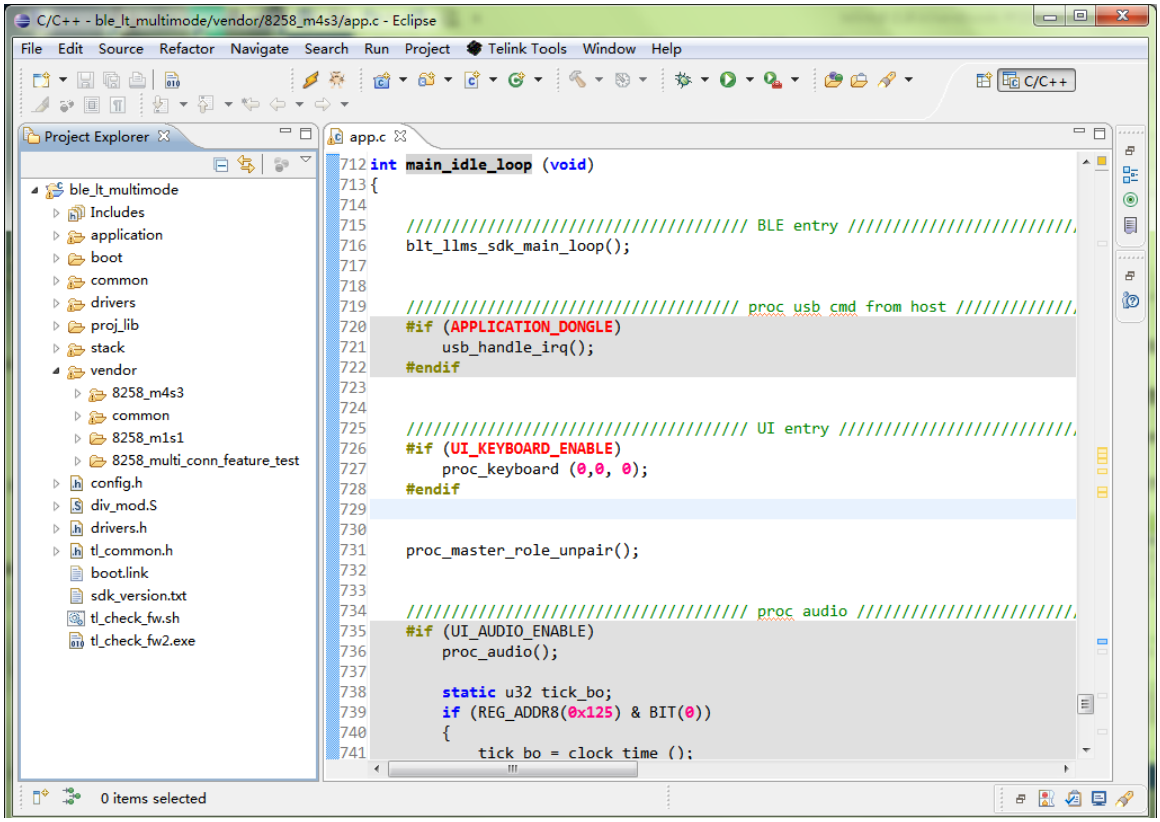
Besides the BLE Stack part, other software and hardware modules of the Multiple Connection SDK (such as Flash, clock, GPIO, IR, etc.) are the same as the previous Single Connection SDK. The introduction of each module in this document will specify whether the current module is exactly the same as in the Single Connection SDK handbook. If it is inconsistent, it will introduce in detail the differences.

1.1 Software Architecture

The multi-connection BLE SDK software architecture includes two parts: the application layer and the BLE protocol stack.

After importing the multi-connection SDK project in Telink IDE, the file structure is shown in the figure below. There are 7 top-level folders: application, boot, common, drivers, proj_lib, stack, vendor.

Figure 1-2 SDK Structure

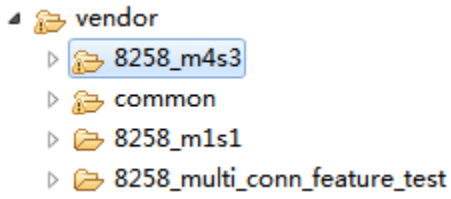


- application: This folder contains general application program, e.g. print, keyboard, and etc.
- boot: This folder contains software bootloader for chip, i.e., assembly code after MCU power on or deepsleep wakeup, so as to establish environment for C program running.
- common: This folder contains generic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

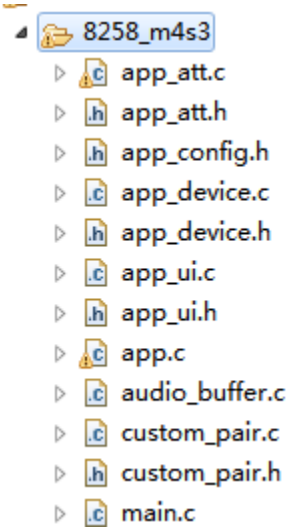
The multi-connection SDK provides three demos, i.e., a four-master three-slave demo (8258_m4s3), a master-slave demo (8258_m1s1) and BLE feature demo (8258_multi_conn_feature_test). These demo projects are in the vendor folder, as shown in figure below.

Figure 1-3 Demo Project



The demo (8258_m4s3) with four masters and three slaves is used as an example to explain the demo file architecture (see Figure 1-3). The 8258_m4s3 demo consists of the following files. This will be described in detail below.

Figure 1-4 8258_m4s3 Demo Project File Architecture



1.1.1 main.c

The "main.c" file includes main function entry and system initialization functions. It's not recommended to make any modification to this file.

```
int main (void)
{
	cpu_wakeup_init();//MCU HW init

#if (CLOCK_SYS_CLOCK_HZ == 32000000)
	clock_init(SYS_CLK_32M_Crystal);
#elif (CLOCK_SYS_CLOCK_HZ == 48000000)
```

```
	clock_init(SYS_CLK_48M_Crystal);
#endif

	gpio_init(1);//gpio init

	rf_drv_init(RF_MODE_BLE_1M); //RF init

#if (APPLICATION_DONGLE)
	usb_init ();
#endif

	if(1){ //read flash size
		blc_readFlashSize_autoConfigCustomFlashSector();
	}
	blc_app_loadCustomizedParameters();   //load customized freq_offset cap value

	user_init();

	irq_enable();//open global interrupt

	while (1)
 {
#if (MODULE_WATCHDOG_ENABLE)
		wd_clear(); //clear watch dog
#endif
		main_loop ();//include BLE, 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: configuration files for GATT services and profiles, the GATT service table already provides standard GATT services, standard GAP services, standard HID services, and some private services. Users can refer to these to add their own services and profiles.
- app_ui.c: Button function.
- app_device.c: information of the peer-slave devices connected to it in the master role (for example: connect handle, attribute handle, BLE device address, address type, etc.) This information is what users need for developing applications.
- custom_pair.c: Telink defined pair solution.

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_ master_slave_handler().`

```
_attribute_ram_code_ void irq_handler(void)
{
        ……
irq_ble_master_slave_handler ();
        ……
}
```

2. BLE logic and data processing entry in main_idle_loop() of app.c file.

   `blt_sdk_main_loop().`

```
int main_idle_loop (void)
{
    ////////////////////////// BLE entry /////////////////////
    blt_llms_sdk_main_loop();

////////////////////////// UI entry /////////////////////
……
}
```

The blt_llms_sdk_main_loop function in the BLE entry section is for processing data and events related to the BLE protocol stack. UI entry for users to write their own application code.

1.2 Applicable IC

Telink Multi-connection BLE SDK is applicable for the following 825X SoC series.

8251/8253/8258 have the same core, the peripherals are basically the same with different the SRAM size, as shown below. Please be noted that for different MCUs, different boot files are needed (boot files will be explained in Section 1.3). 8253/8258 can run all cases, 8251 can run only m1s1 or m2s2 case.

Table 1-1 8x5x Resource

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

Table 1-2 8x5x Supporting Modes

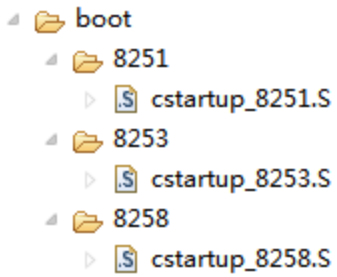
MCU \ project	m1s1	m2s2	m4s3

8251	√	√	×
8253	√	√	√
8258	√	√	√

1.3 Software Bootloader Introduction

Although the above three MCUs are basically the same in hardware configuration (except for SRAM size), the bootloader is different, and different types of MCUs should choose the corresponding bootloader file. In the SDK, the bootloader files of different chips are stored in the boot folder. Telink's bootloader file is composed of two parts, link file and cstartup.S assembly file, as shown in Figure 1-4.

Figure 1-5 Bootloader File

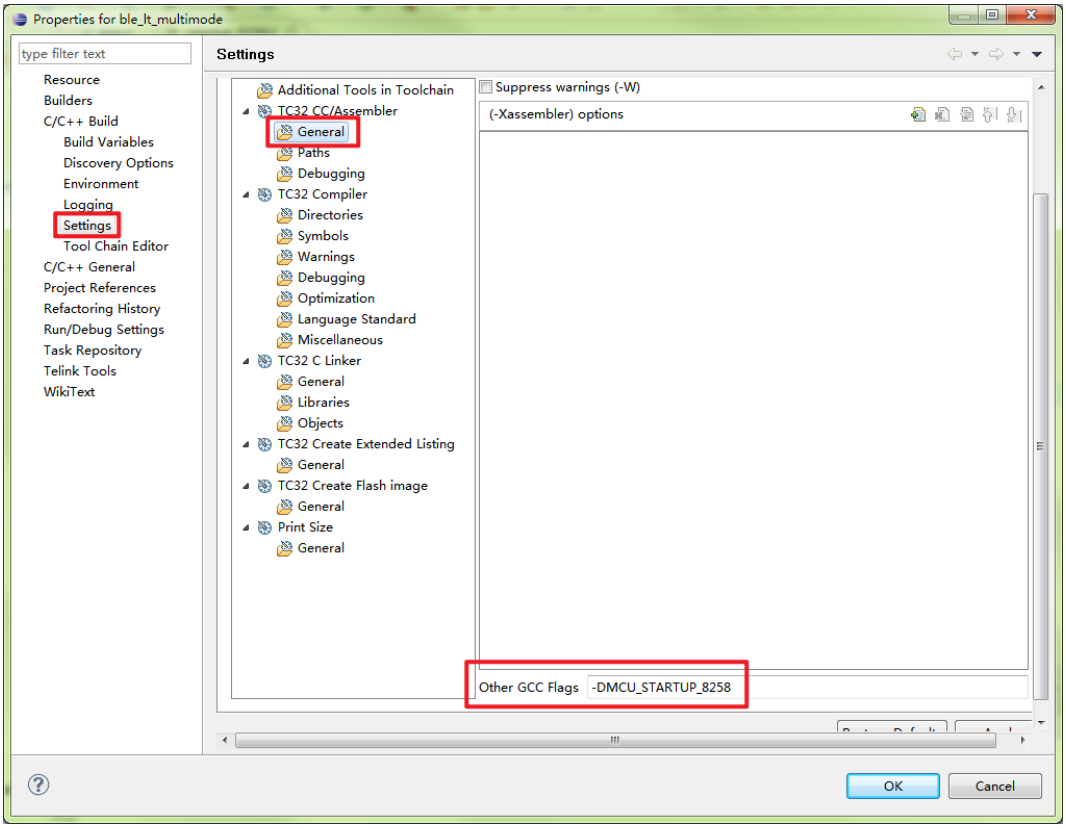


8251/8253/8258 use the same link file, and the cstartup.S file is different depending on the chip. Please be noted that the bootloader used by telink's multi-connection SDK is different from the bootloader used by the single connection SDK, because the retention mode is not used in the multi-connection SDK, so the bootloader is relatively simple.

The default cstartup.S file is cstartup_8258.S in the SDK. Users need to select the corresponding cstartup.S file according to the chip they use. The setting method is as follows (take 8258 chip as an example):

Find the "cstartup_8258.S" file in the boot / 8258 folder, find the macro MCU_STARTUP_8258 in the file, and then configure it in the way shown in Figure 1-5 in the project's property setting window.

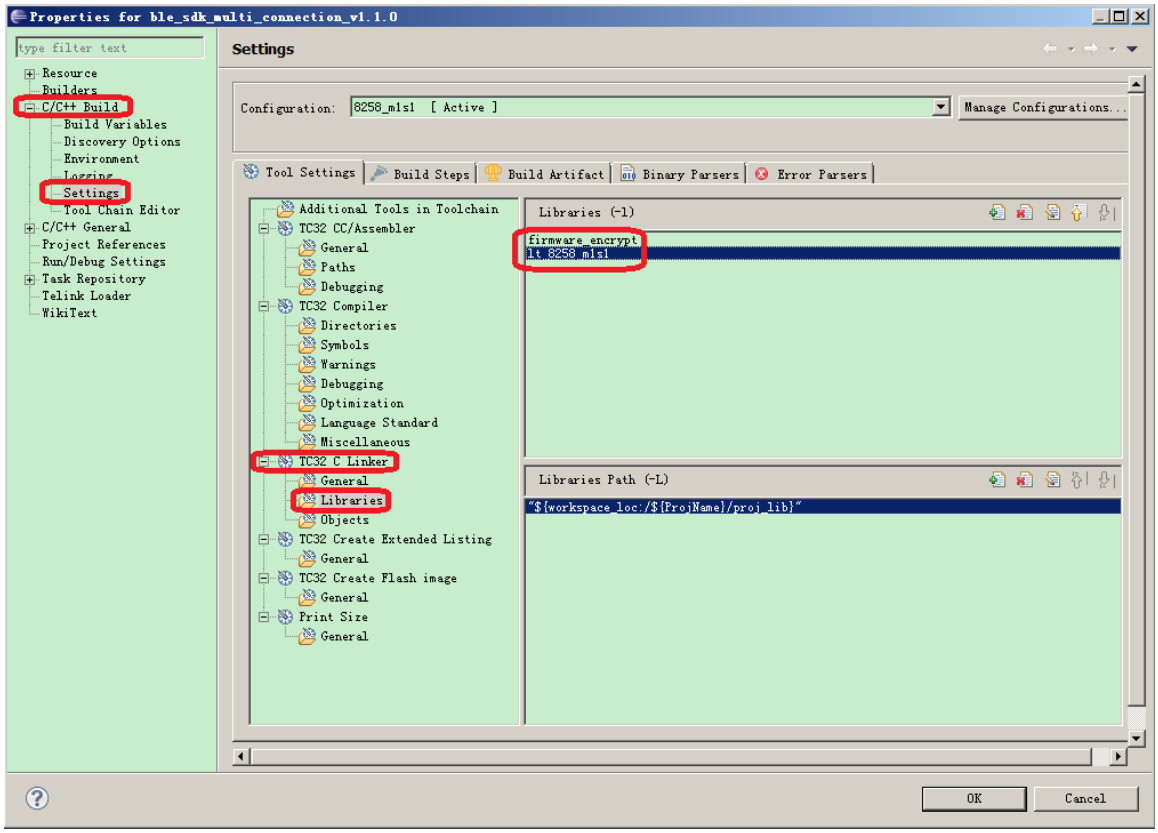
Figure 1-6 cstartup Option



1.4 Library Introduction

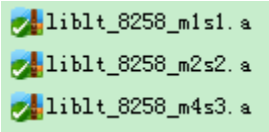
The following figure shows how to select the library corresponding to the project.

Figure 1-7 Project Library Option



The following figure shows current library provided by the SDK's proj_lib folder.

Figure 1-8 SDK Library



The following table shows the adaptation relationship between libraries and different MCUs.

Table 1-3 8x5x Corresponding Library

MCU \ lib	liblt_8258_m1s1	liblt_8258_m2s2	liblt_8258_m4s3
8251	√	√	×
8253	√	√	√
8258	√	√	√

1.5 Demo Introduction

Telink multi-connection BLE SDK provides multiple demo projects.

Users can observe the intuitive effect through the operation of the software and hardware demo. Users can also modify the demo code to develop their own applications.

1.5.1 M4S3 demo/M1S1 demo

Telink Multi-Connection BLE SDK provides the demo project 8258_m4s3 with 4 masters and 3 slaves (as shown in Figure 1-6). Users can choose two libraries: liblt_8258_m4s3 and liblt_8258_m2s2. At the same time, the demo project 8258_m1s1 with 1 master and 1 slave is provided, and users can only use the library file liblt_8258_m1s1.

Lib supports the number of connections with named numbers, such as: liblt_8258_m4s3 can support up to 4 master and 3 slave; liblt_8258_m2s2 can support up to 2 master and 2 slave. If the customer does not actually use so much, you can change it in app_config.h according to actual needs. such as:

*app_config.h

```
45
46
47  #define MASTER_MAX_NUM                  3
48  #define SLAVE_MAX_NUM                   2
49
```

The purpose of providing the liblt_8258_m2s2 library is that, if the user actually needs the number of master and slave connections to be no more than two, use the liblt_8258_m2s2 library will occupy less ram space.

Such as:

*app_config.h

```
45
46
47  #define MASTER_MAX_NUM                  2
48  #define SLAVE_MAX_NUM                   1
```

The above content will be introduced in detail in chapter 3.2.1 Connection Number & Connection Handle.

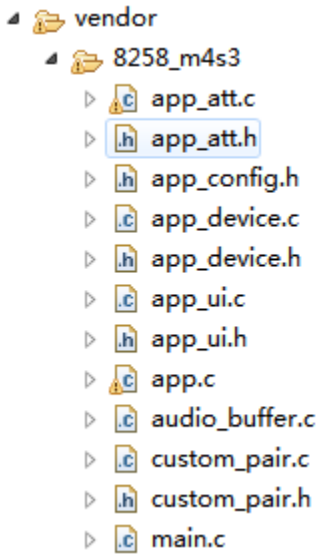
Taking liblt_8258_m4s3 as an example, the user can compile the project and download it to the development board. It should be noted that the 4 masters and 3 slaves mentioned here refers to the device that has burned the 8258_m4s3 project bin file and can connect 4 peer slaves and 3 peer masters at the same time. Therefore, in order to see the effect, the user needs to prepare additional 4 slaves (such as: remote control, etc.) and 3 masters (such as: mobile phone, etc.). For 8258_m1s1, one additional slave and one master can be prepared.

The Multiple SDK project supports multiple masters, but even if only one master is supported, due to resource constraints and other reasons, the master's SDP process cannot be fully implemented. Therefore, we only provide a simple SDP demo (see the SDP implementation in app_att.c). Of course, the SDK provides all the ATT APIs required for service discovery, and users can use these APIs to complete their own specific service discovery. The default SDP function in the SDK can be enabled or disabled through the macro BLE_MASTER_SIMPLE_SDP_ENABLE in app_config.h.

Also, telink multi-connection BLE SDK and the previous single master SDK have small changes in the structure of the app layer. The app layer of the previous single master SDK is composed of app + host, while the new multi-connection BLE SDK puts the host part of the app layer at the bottom to reduce the difficulty of use.

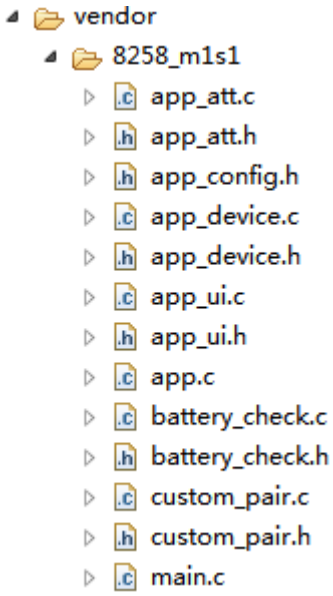
The 8258_m4s3 project does not support the low power consumption function, and the low voltage alarm is not added. (If the customer needs the battery detection function, please refer to the m1s1 project, and the subsequent m4s3 project will also add the battery detection code in the future).

Figure 1-9 M4S3 Demo Project



The 8258_m1s1 project supports low power consumption and battery detection functions.

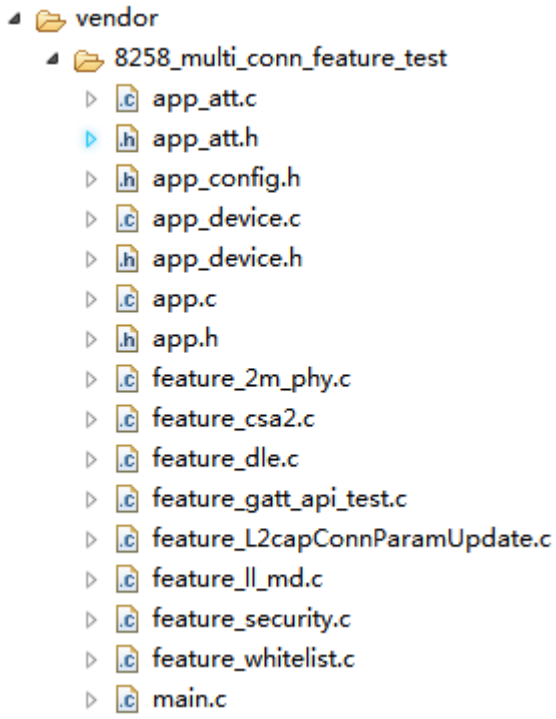
Figure 1-10 M1S1 Demo Project



1.5.2 Feature demo

Telink multi-connection BLE SDK also provides some demo projects 8258_multi_conn_feature_test (Figure 1-8) that demonstrate how to use common BLE related features. Users can refer to these demos to understand how to use various BLE functions in telink SDK to achieve their own function. Later chapters will detail the use of these features. Please be noted that multiple connection SDK no longer provides driver demo, users can refer to driver demo in Single connection SDK.

Figure 1-11 Feature Demo



Users can select different feature demos through the macro "FEATURE_TEST_MODE" in the app_config.h file in the 8258_multi_conn_feature_test project directory.

2. Basic Modules

2.1 MCU Address Space

2.1.1 MCU Address Space Allocation

Please refer to the corresponding section of the Single Connection SDK handbook. The Multiple Connection SDK is completely consistent with it, and will not be repeated here.

2.1.2 MCU Address Space Access

Please refer to the corresponding section of the Single Connection SDK handbook. The Multiple Connection SDK is completely consistent with it, and will not be repeated here.

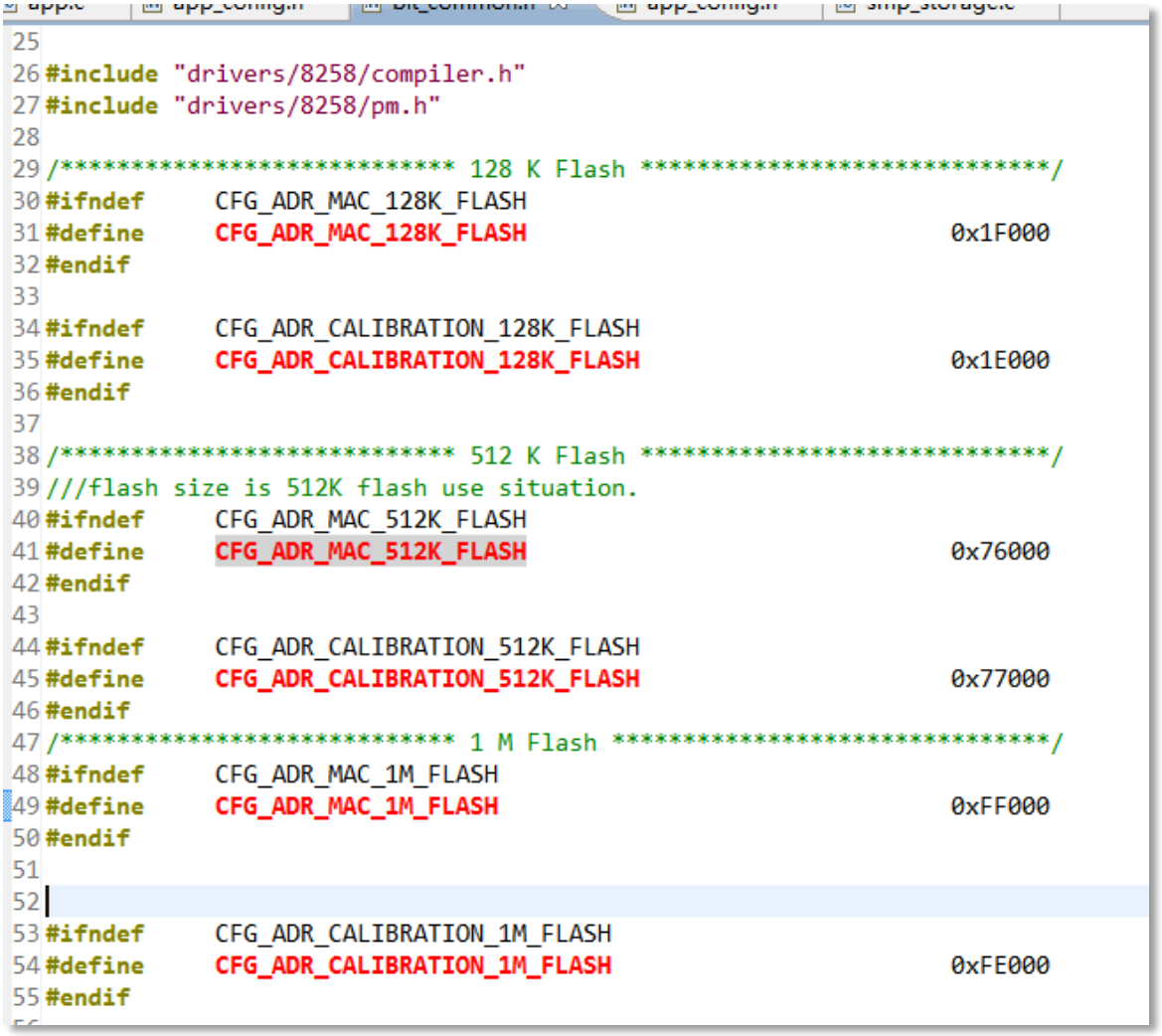
2.1.3 SDK FLASH Space Allocation

The basic FLASH storage unit is equal to the size of a sector (4K byte) , flash is erased by the sector (the erase function is flash_erase_sector), theoretically the same type of information needs to be stored in the same sector, different information needs to be stored in different sectors (to prevent other types of information from being erased by mistake when erasing the information). Therefore, it is recommended that users follow the principle of "putting different types of information in different sectors" when using FLASH to store customized information.

There are four types of information in the Telink multi-connection SDK that need to be stored in flash, namely MAC, calibration information, encrypted pairing information and SDP information. These parameters are allocated different flash space by default in the SDK.

The FLASH space for storing MAC and calibration information will vary with the size of the chip FLASH. By default, for 512K FLASH chips, the MAC is stored in the 4K FLASH space starting at 0x76000, and the calibration information is stored in the 4K FLASH starting at 0x77000. For the 1M FLASH chip, the MAC is stored in the 4K FLASH space starting at 0xFF000, and the calibration information is stored in the 4K FLASH space starting at 0xFE000; the SDK can automatically configure the corresponding MAC and FLASH according to the size of the user 's chip FLASH. The calibration value storage space no longer requires manual configuration by users who use the default MAC and calibration value storage space. Users can also modify the corresponding macros in vendor / common / blt_common.h (Figure 2-1) to modify the default MAC and calibration value storage space according to their own needs. At this time, it is necessary to modify telink mass production firmware write address accordingly.

Figure 2-1 MAC and Calibration Information Default FLASH Storage Address



```
25
26 #include "drivers/8258/compiler.h"
27 #include "drivers/8258/pm.h"
28
29 /*************************** 128 K Flash ***************************/
30 #ifndef     CFG_ADR_MAC_128K_FLASH
31 #define     CFG_ADR_MAC_128K_FLASH                          0x1F000
32 #endif
33
34 #ifndef     CFG_ADR_CALIBRATION_128K_FLASH
35 #define     CFG_ADR_CALIBRATION_128K_FLASH                  0x1E000
36 #endif
37
38 /*************************** 512 K Flash ***************************/
39 ///flash size is 512K flash use situation.
40 #ifndef     CFG_ADR_MAC_512K_FLASH
41 #define     CFG_ADR_MAC_512K_FLASH                          0x76000
42 #endif
43
44 #ifndef     CFG_ADR_CALIBRATION_512K_FLASH
45 #define     CFG_ADR_CALIBRATION_512K_FLASH                  0x77000
46 #endif
47 /*************************** 1 M Flash ***************************/
48 #ifndef     CFG_ADR_MAC_1M_FLASH
49 #define     CFG_ADR_MAC_1M_FLASH                            0xFF000
50 #endif
51
52
53 #ifndef     CFG_ADR_CALIBRATION_1M_FLASH
54 #define     CFG_ADR_CALIBRATION_1M_FLASH                    0xFE000
55 #endif
```

Encrypted pairing information and SDP information are also stored in independent FLASH space. The storage space of these information cannot be automatically adjusted with the FLASH size of the chip, and can only be set manually by the user. By default, encrypted pairing information is stored in the 16K FLASH space starting at 0x78000; and SDP information is stored in the 8K FLASH space starting at 0x7D000. Users can call blc_smp_configParingSecurityInfoStorageAddressAndSize () function to modify the starting address and size of encrypted pairing information storage; modify the starting address and size of SDP information storage by modifying the macros FLASH_SDP_ATT_ADRRESS and FLASH_SDP_ATT_MAX_SIZE in app_config.h.

2.2 Clock Module

Please refer to the corresponding section of the Single Connection SDK handbook.

The difference between the multi-connection SDK and the Single Connection SDK is that the multi-connection SDK can only use two system clocks, SYS_CLK_32M_Crystal and SYS_CLK_48M_Crystal. The other clocks are too slow to run the SDK.

2.3 GPIO Module

Please refer to the corresponding section of the Single Connection SDK handbook. The Multiple Connection SDK is completely consistent with it, and will not be repeated here.

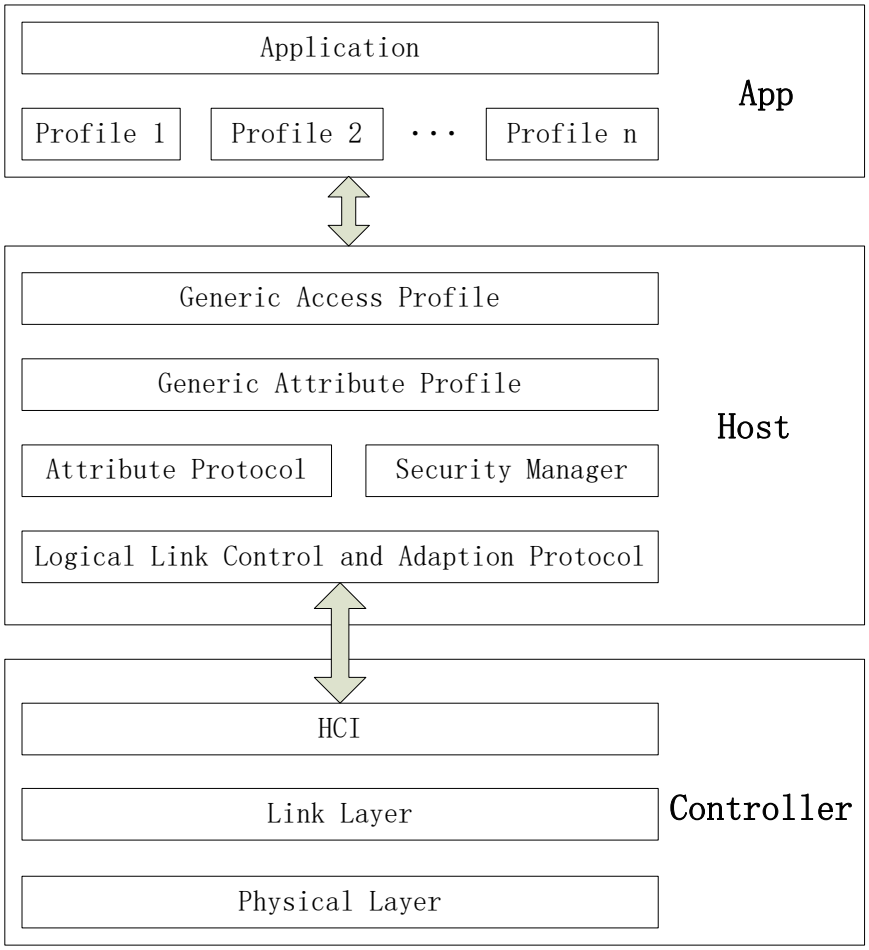
3. BLE Module

3.1 BLE SDK Software Architecture

3.1.1 Standard BLE SDK Software Architecture

According to BLE spec, a standard BLE SDK architecture is shown in the following figure:

Figure 3-1 BLE SDK Standard Architecture



In the architecture shown in the figure above, the BLE protocol stack is divided into two parts: Host and Controller.

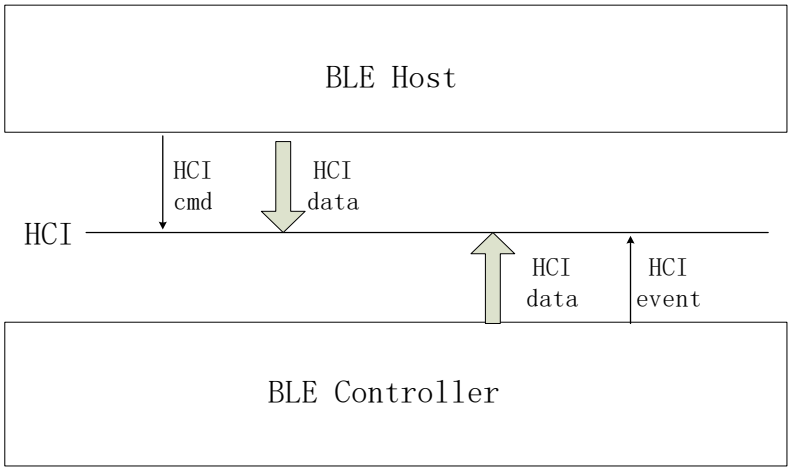
Controller is the underlying protocol of BLE, including Physical Layer (PHY) and Link Layer (LL). Host Controller Interface (HCI) is the only communication interface between Controller and Host, and all data interaction between Controller and Host is completed through this interface.

Host is the upper layer protocol of BLE, including Logic Link Control and Adaption Protocol (L2CAP), Attribute Protocol (ATT), Security Manager Protocol (SMP), and Profile includes Generic Access Profile (GAP) and Generic Attribute Profile (GATT).

The application layer (APP) contains the user's own application codes and profiles corresponding to various services. The user controls access to the host through GAP.

The Host executes data interaction with the Controller through HCI, as shown in the following figure:

Figure 3-2 HCI Data Interaction of Host and Controller



1. BLE Host uses HCI cmd to operate and set the controller. These HCI cmds correspond to the controller API that will be introduced later in this chapter
2. The Controller reports various HCI events to the host through HCI, which will also be introduced in this chapter.
3. The Host transmits the data that needs to be sent to the other device to the Controller through HCI, and the Controller directly throws the data to the Physical Layer for transmission.
4. The RF data received by the Controller at the Physical Layer first determines whether it is the Link Layer data or the Host data: if it is Link Layer data, the data is processed directly; if it is Host data, the data is transmitted to the Host through HCI.

3.1.2 Telink BLE SDK Software Architecture

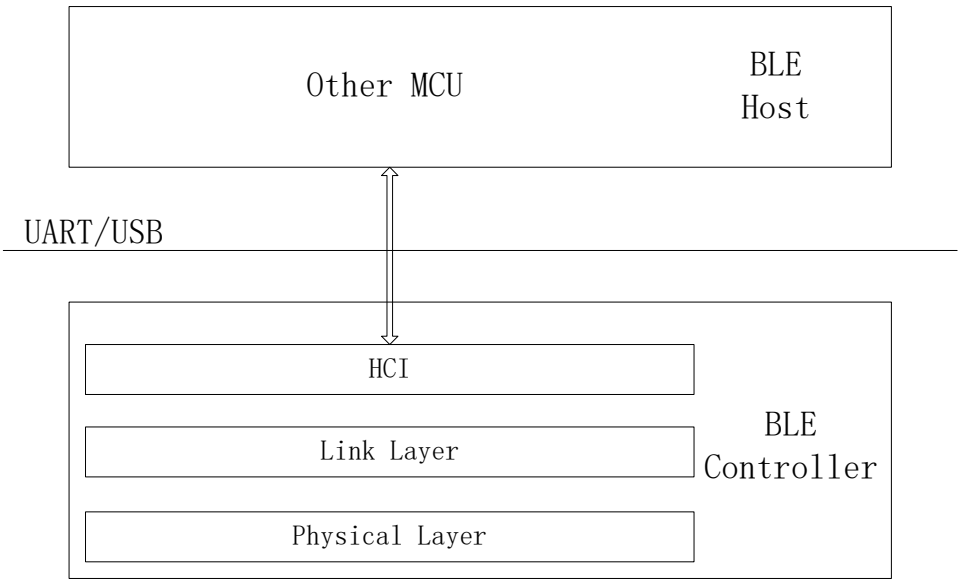
3.1.2.1 Telink BLE Multiple Connection Controller

Telink BLE Multi-Connection SDK supports standard BLE controllers, including HCI, PHY (Physical Layer) and LL (Link Layer). This part of the reference design is not yet available and will be added in the future SDK.

Telink BLE Multiple Connection SDK includes five standard states of Link Layer (standby, advertising, scanning, initiating, connection), and supports up to 4 Master roles and 3 Slave roles simultaneously in connection state.

The controller architecture diagram is as follows:

Figure 3-3 BLE Multiple Connection Controller Architecture

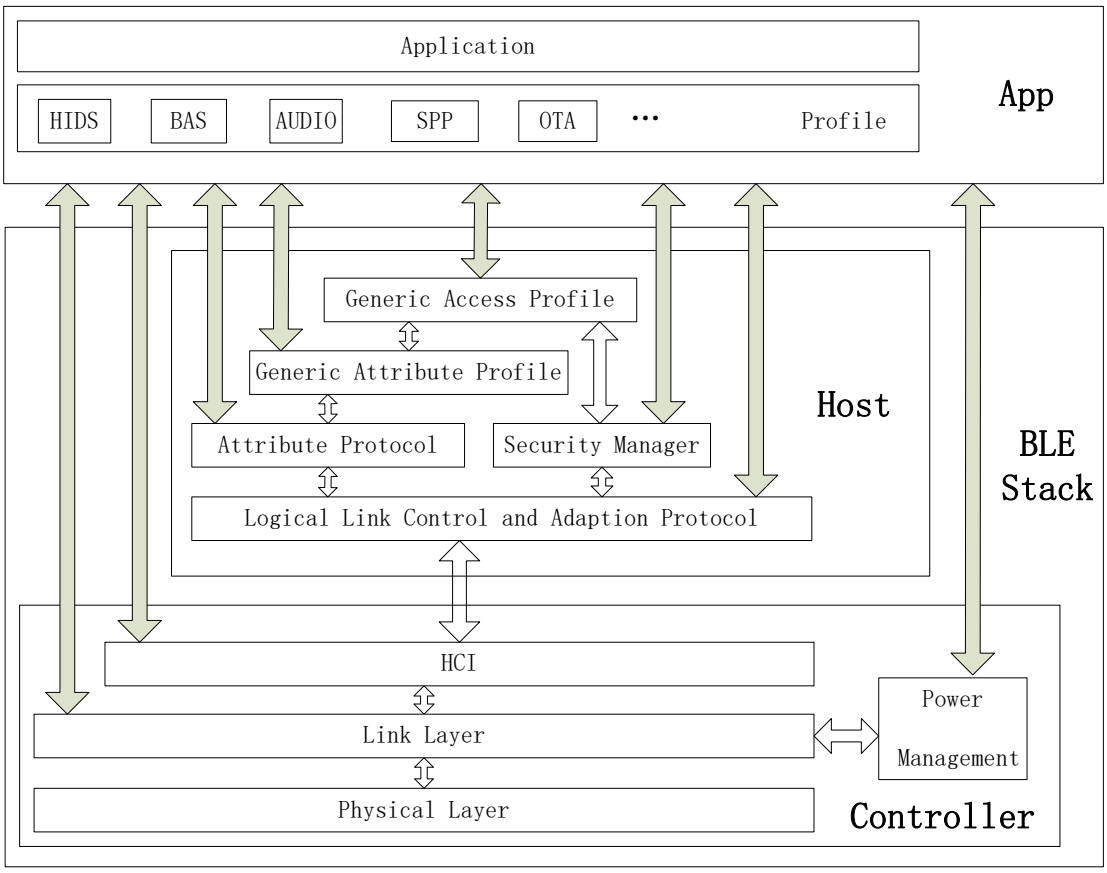


3.1.2.2 Telink BLE Multiple Connection Whole Stack (Controller + Host)

The Telink BLE Multiple Connection SDK provides a BLE Multiple Connection Whole Stack (Controller + Host) reference design, which cannot be fully supported only for Master SDP (service discovery), which will be introduced in the following chapters.

The Telink BLE stack architecture will simplify the above standard structure to minimize the system resource (including Sram, running time, power consumption, etc.) of the entire SDK. The architecture is shown in the following figure. 8258_m4s3 and 8258_m1s1 in the SDK are based on this architecture.

Figure 3-4 Telink BLE Multiple Connection Whole Stack Architecture



The data interaction shown by the solid arrows in the figure is that the user can operate and control through various interfaces, and the user API will be provided. The hollow arrow is the data interaction completed inside the protocol stack, and the user cannot participate.

HCI is the data communication interface between the Controller and the Host (interfacing with the L2CAP layer), but it is not the only interface. The APP application layer can also directly interact with the Link Layer for data interaction. The Power Management (PM) low-power management unit is embedded in the Link layer, and the application layer can call PM-related interfaces for power management settings.

Considering efficiency, the data interaction between the application layer and the host does not control access through GAP. The protocol stack provides related interfaces in ATT, SMP and L2CAP, which can directly interact with the application layer. But all Host events need to interact with the application layer through the GAP layer.

Based on the Attribute Protocol, the Host layer implements Generic Attribute Profile (GATT). Based on GATT, the application layer defines various profiles and services required by users themselves. The BLE SDK provides several basic profiles, including HIDS, BAS, OTA, etc.

Based on this architecture, we will give a basic introduction to each part of the 8x5x BLE protocol stack and give the user API of each layer.

The Physical Layer is completely controlled by the Link Layer, and does not require any involvement of the application layer, which is not described in this section.

Although part of the data interaction between Host and Controller is still done by HCI, it is basically done by the Host and Controller protocol stack. The application layer hardly participates. You only need to register the HCI data callback processing function at the L2CAP layer. HCI will not be described in this section.

3.2 Link Layer

3.2.1 Connection Number & Connection Handle

3.2.1.1 supportedMaxMasterNum & supportedMaxSlaveNum

The Multiple Connection SDK always refers to the maximum number of Connection master roles as supportedMaxMasterNum, and the maximum number of Connection Slave roles as supportedMaxSlaveNum. They are determined by the library, as shown in the following table:

Table 3-1 Maximum Supported Master/Slave Number of Libraries

library	supportedMaxMasterNum	supportedMaxSlaveNum
liblt_8258_m4s3	4	3
liblt_8258_m2s2	2	2
liblt_8258_m1s1	1	1

3.2.1.2 appMaxMasterNum & appMaxSlaveNum

If supportedMaxMasterNum and supportedMaxSlaveNum have been determined, users can set the maximum number of Masters and Slaves they want on their applications through the following APIs, which are called appMaxMasterNum and appMaxSlaveNum, respectively.

```
ble_sts_t  blc_llms_setMaxConnectionNumber(int max_master_num,
                int max_slave_num);
```

This API is only allowed to be called during initialization, that is, the number of related connections needs to be determined before the Link Layer runs, and it is not allowed to be modified later.

The user's appMaxMasterNum and appMaxSlaveNum must be less than or equal to supportedMaxMasterNum and supportedMaxSlaveNum.

The reference Demo design uses this API during initialization:

```
blc_llms_setMaxConnectionNumber( MASTER_MAX_NUM, SLAVE_MAX_NUM);
```

Users need to define their own appMaxMasterNum and appMaxSlaveNum in app_config.h, namely MASTER_MAX_NUM and SLAVE_MAX_NUM in SDK

```
#define MASTER_MAX_NUM                              4
#define SLAVE_MAX_NUM                               3
```

For example, appMaxMasterNum and appMaxSlaveNum in M3S3 Demo are 4 and 3 respectively; appMaxMasterNum and appMaxSlaveNum in M1S1 Demo are 1 and 1 respectively.

appMaxMasterNum and appMaxSlaveNum can save various resources of MCU, such as library for M4S3, if users only need to use M3S2, set MASTER_MAX_NUM and SLAVE_MAX_NUM to 3 and 2, respectively:

1. Save SRAM Space

Link Layer TX Master FIFO and TX Slave FIFO, L2CAP Master MTU buffer and L2CAP Slave MTU buffer are all allocated according to appMaxMasterNum and appMaxSlaveNum, so some Sram resources can be saved. For details, please refer to the relevant introduction in section 3.2.4.1 TX FIFO definition and setting.

2. Reduce Time and Power Comsumption

For M4S3, stack must wait until currentMasterNum is 4 to stop the Scan action, and must wait until currentSlaveNum is 3 to stop the Advertising action. For M3S2, Stack will stop Scan operation when currentMasterNum is 3, and will stop Advertising operation when currentSlaveNum is 2, so there is no unnecessary Scan and Advertising, which can save PHY layer bandwidth and reduce MCU power consumption.

3.2.1.3 currentMaxMasterNum & currentMaxSlaveNum

After user define appMaxMasterNum and appMaxSlaveNum, they determine the maximum number of Master and Slave created when the Link Layer runs. However, the number of Masters and Slaves at a certain moment is still uncertain. For example, when appMaxMasterNum is 4, the number of Masters may be 0,1,2,3,4 at any moment.

The SDK provides the following three APIs for users to query the number of Master and Slave on the current Link Layer in real time.

```
int         blc_llms_getCurrentConnectionNumber(void);//master + slave connection number
int   blc_llms_getCurrentMasterRoleNumber(void);//master role number
int   blc_llms_getCurrentSlaveRoleNumber(void);//slave   role number
```

3.2.1.4 Connection Handle

According to the BLE Spec, the Connection Handle is used to identify a specific connection, and its value is uncertain. In order to make the design and user development more convenient, the Telink Multiple Connection SDK simplifies the Connection Handle.

The range of the Connection Handle value is determined by supportedMaxMasterNum and supportedMaxSlaveNum.

The following table shows the value range of the Master / Slave Connection Handle corresponding to the 8258_m1s1 demo. The Master Connection Handle is always 0x0080, and the Slave Connection Handle is always 0x0041.

Table 3-2 m1s1 Connection Handle

library	Connection Handle	
	Master	Slave
libIt_8258_m1s1	0x0080	0x0041

The following table shows the value range of the Master / Slave Connection Handle corresponding to different libraries for the 8258_m4s3 demo:

Table 3-3 m4s3 Connection Handle

library	Connection Handle						
	Master1	Master2	Master3	Master4	Slave0	Slave1	Slave2
liblt_8258_m4s3	0x0080	0x0081	0x0082	0x0083	0x0044	0x0045	0x0046
liblt_8258_m2s2	0x0080	0x0081	×	×	0x0042	0x0043	×

It can be seen that the BIT (7: 6) in the Master Connection Handle value is always 0b'10 and the BIT (7: 6) in the Slave Connection Handle value is always 0b'01, which corresponds to the following macro definition in the SDK:

```
#define BLM_CONN_HANDLE BIT(7)
#define BLS_CONN_HANDLE BIT(6)
```

Therefore, a code similar to the following will appear at the application layer to determine whether the current connection is Master or Slave, and users can also use this method.

```
if(connHandle & BLM_CONN_HANDLE)  //master
if(connHandle & BLS_CONN_HANDLE)  //slave
```

The value range of the Master / Slave Connection Handle mentioned above is determined according to supportedMaxMasterNum and supportedMaxSlaveNum, and if the appMaxMasterNum and appMaxSlaveNum set by the user are relatively small, the actual range of the Master / Slave Connection Handle will be further reduced. Because M1S1 is too simple to introduce, there is no need to introduce it. Taking M4S3 as an example, assuming that the user's appMaxMasterNum and appMaxSlaveNum are 3 and 2, respectively, then the Master Connection Handle can only be 0x0080 / 0x0081 / 0x0082, 0x0083 is not possible, Slave Connection Handle is only It is 0x0044 / 0x0045, and 0x0046 is not possible.

Please analyze other appMaxMasterNum and appMaxSlaveNum values in similar way.

3.2.2 Link Layer State Machine

Users can refer to the introduction of Link Layer state machine in Single Connection SDK first. In the Single Connection SDK, the five basic states of Link Layer are supported. If the Connection state is further divided into Connection Slave role and Connection Master role, the Link Layer must be at any time and can only be one of the following six states: Standby, Advertising, Scanning, Initiating, Connection Slave role, Connection Master role.

For the Multiple Connection SDK, due to supporting multiple Master and Slave at the same time, Link Layer cannot be in a certain state at a time, and must be a combination of several states.

The Link Layer state machine of the Multiple Connection SDK is relatively complicated. The following is a general introduction to help users understand basic underlying layer and the use of the corresponding API.

3.2.2.1 Link Layer State Machine Initialization

The Multiple Connection SDK will design each basic state according to a modular design, and need to initialize modules in advance.

The initialization of the MCU is compulsory, the API is as follows:

```
void        blc_llms_initBasicMCU (void);
```

The API for adding the Standby module is as following. This is compulsory. All BLE applications need to be initialized.

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

The parameter public_adr is a pointer to BLE public mac address.

The initialization APIs of the corresponding modules in several other states (Advertising, Scanning, Multi_Master_Multi_Slave) are as follows:

```
void        blc_llms_initAdvertising_module(void);
void        blc_llms_initScanning_module(void);
void        blc_llms_initMultiMasterSingleSlave_module (void);
```

3.2.2.2 Link Layer State Combination

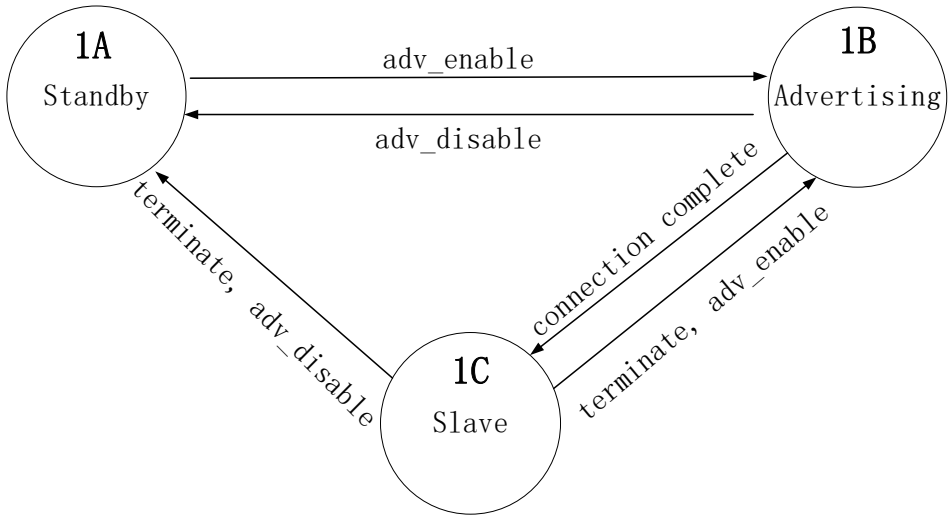
The Initiating state is relatively simple. When the Scan state needs to initiate a connection to a advertising device, the Link Layer enters the Initiating state. Within a certain period of time (this time is called create connection timeout), either the connection is established successfully, and an additional connection master role, or the connection establishment fails, and the Link Layer returns to the Scanning state again. In order to simplify the introduction of the Link Layer state machine and make it easier for users to understand, the following temporary states of initiating are ignored in the following introduction.

The Multiple Connection SDK Link Layer state machine can be described from two perspectives, one is the conversion of Advertising and Slave; the other is the conversion of Scanning and Master; these two angles do not affect each other.

We analyze from simple to complex, first analyze the situation of M1S1. In M1S1, supportedMaxMasterNum and supportedMaxSlaveNum are both 1, assuming that the user's appMaxMasterNum and appMaxSlaveNum are also 1.

The state machines for switching between M1S1 Advertising and Slave are as follows:

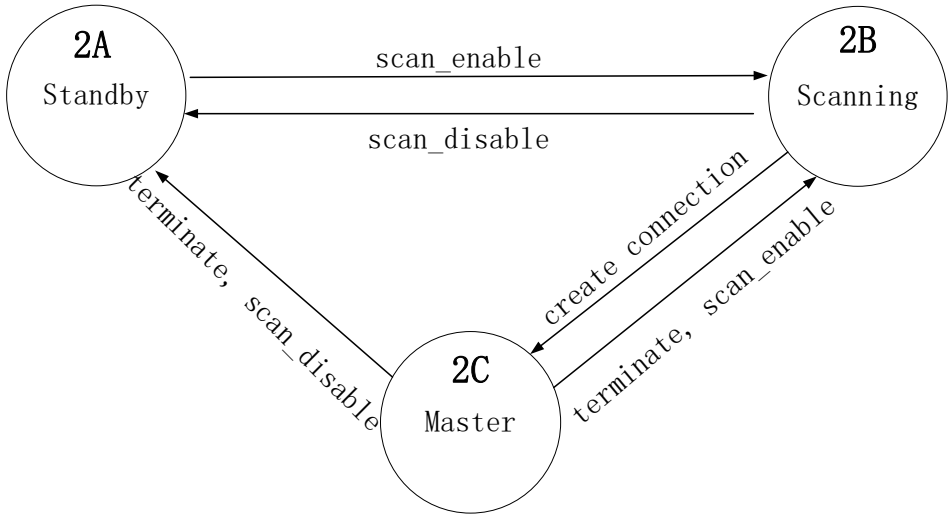
Figure 3-5 M1S1 Advertising and Slave Switching



In the figure, adv_enable and adv_disable refer to the state set by the user's last call to blc_llms_setAdvEnable (adv_enable) when the condition occurs.

The state machines for switching between M1S1 Scanning and Master are as follows:

Figure 3-6 M1S1 Scanning Master Switching



In the figure, scan_enable and scan_disable refer to the state set by the user when they call blc_llms_setScanEnable (scan_enable, filter_duplicate) for the last time.

Advertising and Slave, Scanning and Master each have three states. Since the logic between the two is completely independent and does not affect each other, then the final Link Layer combination state has 3 * 3 = 9, as shown in the following table:

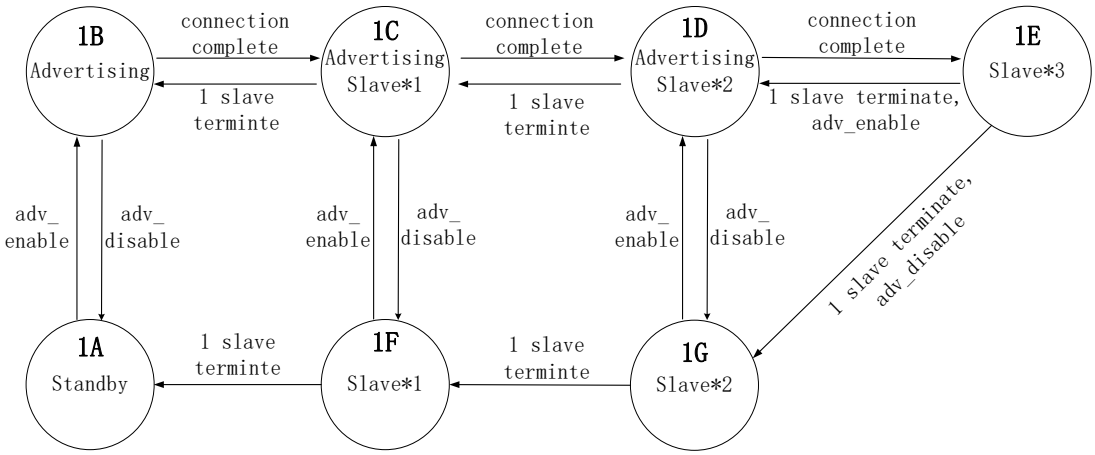
Table 3-4 M1S1 Link Player Status

	2A	2B	2C
1A	Standby	Scanning	Master
1B	Advertising	Advertising + Scanning	Advertising + Master
1C	Slave	Slave + Scanning	Slave + Master

For M4S3, assume that the user's appMaxMasterNum and appMaxSlaveNum are 4 and 3, which is the most complicated case.

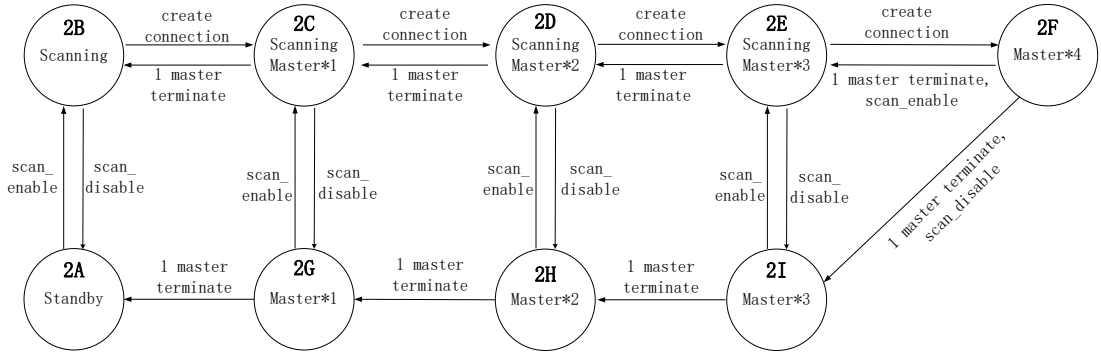
The state machines for switching between M4S3 Advertising and Slave are as follows:

Figure 3-7 M4S3 Advertising and Slave Switching



The state machines for switching between M4S3 Scanning and Master are as follows:

Figure 3-8 M4S3 Scanning and Master Switching



There are 7 possible states for Advertising and Slave, and 9 possible states for Scanning and Master. Since the logic between the two is completely independent and does not affect each other, then the final Link Layer combined state is 7 * 9 = 63, as shown in the following table.

Table 3-5 M4S3 Link Layer Status

	2A	2B	2C	2D	2E	2F	2G	2H	2I
1A	Standby	Scanning	Scanning Master*1	Scanning Master*2	Scanning Master*3	Master*4	Master*1	Master*2	Master*3
1B	Advertising	Advertising Scanning	Advertising Scanning Master*1	Advertising Scanning Master*2	Advertising Scanning Master*3	Advertising Master*4	Advertising Master*1	Advertising Master*2	Advertising Master*3

	2A	2B	2C	2D	2E	2F	2G	2H	2I
1C	Advertising Slave*1	Advertising Slave*1 Scanning	Advertising Slave*1 Scanning Master*1	Advertising Slave*1 Scanning Master*2	Advertising Slave*1 Scanning Master*3	Advertising Slave*1 Master*4	Advertising Slave*1 Master*1	Advertising Slave*1 Master*2	Advertising Slave*1 Master*3
1D	Advertising Slave*2	Advertising Slave*2 Scanning	Advertising Slave*2 Scanning Master*1	Advertising Slave*2 Scanning Master*2	Advertising Slave*2 Scanning Master*3	Advertising Slave*2 Master*4	Advertising Slave*2 Master*1	Advertising Slave*2 Master*2	Advertising Slave*2 Master*3
1E	Slave*3	Slave*3 Scanning	Slave*3 Scanning Master*1	Slave*3 Scanning Master*2	Slave*3 Scanning Master*3	Slave*3 Master*4	Slave*3 Master*1	Slave*3 Master*2	Slave*3 Master*3
1F	Slave*1	Slave*1 Scanning	Slave*1 Scanning Master*1	Slave*1 Scanning Master*2	Slave*1 Scanning Master*3	Slave*1 Master*4	Slave*1 Master*1	Slave*1 Master*2	Slave*1 Master*3
1G	Slave*2	Slave*2 Scanning	Slave*2 Scanning Master*1	Slave*2 Scanning Master*2	Slave*2 Scanning Master*3	Slave*2 Master*4	Slave*2 Master*1	Slave*2 Master*2	Slave*2 Master*3

Please analyze in the similar way if appMaxMasterNum / appMaxSlaveNum are not 4 and 3.

The previous 3.2.1 Connection Number & Connection Handle introduced the concepts of supportedMaxMasterNum / supportedMaxSlaveNum and appMaxMasterNum / appMaxSlaveNum, corresponding to the number of Master and Slave in the state machine combination table above, and then defining the two concepts currentMasterNum and currentSlaveNum, indicating the actual Link Layer The number of Masters and Slaves, for example, in the combined state of '1D2E' in the above table, currentMasterNum is 3 and currentSlaveNum is 2.

3.2.3 Link Layer timing

Link Layer timing is more complicated. Here only some basic knowledge is introduced, which is enough for users to understand and use related APIs reasonably.

Link Layer contains 5 basic single-state Standby, Advertising, Scanning, Initiating, Connection, ignore the brief Initiating that is only used when Master create connection, we only briefly introduce the timing of the remaining 4 states.

Each sub-state (Advertising, Master0 ~ Master3, Slave0 ~ Slave2, Scanning, UI task) will be indicated as the following figure:

Figure 3-9 Status Indicators

3.2.3.1 Timing for "Standby state"

This corresponds to the 1A2A state of M4S3 in Table 3-2.

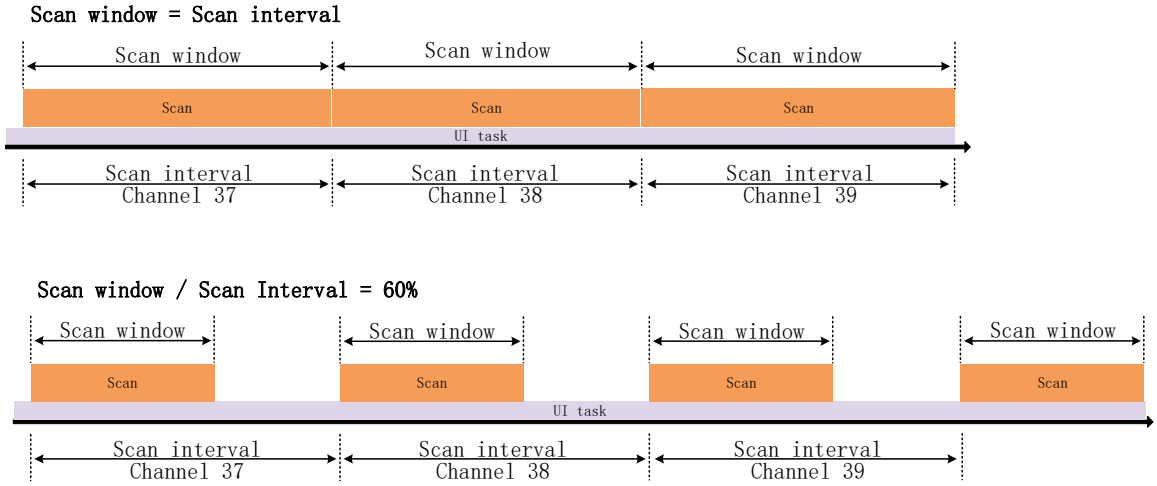
When the Link Layer is in the Idle state, the Link Layer and the Physical Layer do not have any tasks to deal with. The blt_llms_sdk_main_loop function does not work at all and does not generate any interruption. It can be considered that UI entry (UI task) occupies the entire main_loop time.

3.2.3.2 Timing for "Scanning only, no Adverting, no Connection"

This corresponds to the 1A2B state of M4S3 in Table 3-2.

At this time, only the Scanning state needs to be processed, and Scan has the highest efficiency. Link Layer switches channels 37/38/39 according to Scan interval. The timing diagram is as follows:

Figure 3-10 Timing Sequence of M4S3 1A2B



The actual Scan time is determined according to the size of the Scan window. If the Scan window is equal to the Scan interval, all the time is in Scan; if the Scan window is less than the Scan interval, select the time equal to the Scan window from the beginning to perform Scan in the Scan interval.

The Scan window shown in the figure is about 60% of the Scan interval. In the first 60% of the time, the Link Layer is in the Scanning state, and the PHY layer is receiving packets. At the same time, users can use this time to execute their own UI tasks in the main_loop . The last 40% of the time is not in the Scanning state,

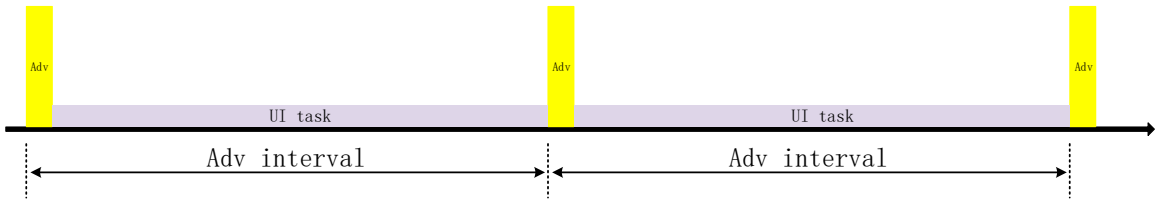
and the PHY layer stops working. Users can use this time to execute their own UI task in the main_loop. For the design of adding low power management to the M1S1 that will be introduced later, this time can also allow The MCU enters suspend to reduce the overall power consumption.

3.2.3.3 Timing for "Advertising only, no Scanning, no Connection"

This corresponds to the 1B2A state of M4S3 in Table 3-2.

According to the Adv interval, the Advertising Event is assigned to the time axis. The timing diagram is as follows:

Figure 3-11 Timing Sequence of M4S3 1B2A



For all the details of Adv Event, please refer to the detailed introduction of Adv Event in Single Connection SDK, the two are the same.

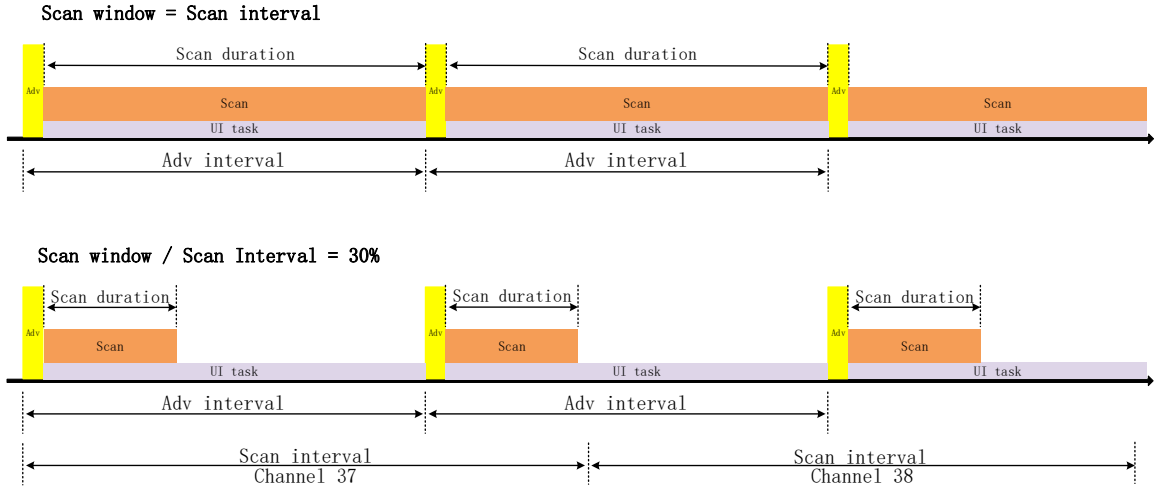
Users can use non-Adv time to execute their own UI task in main_loop.

3.2.3.4 Timing for "Advertising, Scanning, no Connection"

This corresponds to the 1B2B state of M4S3 in Table 3-2.

First, assign the Advertising Event to the time axis according to the Adv interval, and then assign Scanning. The timing diagram is as follows:

Figure 3-12 Timing Sequence of M4S3 1B2B



Since the application requires higher time accuracy for advertising than scanning, Adv Event has a higher priority. Assign the timing of Adv Event first, and then use the remaining time between Adv Events for Scan. Use this remaining time to execute your own UI task in main_loop. When the Scan window set by the user is equal to the Scan interval, the Scan duration in the figure will fill up the remaining time; when the Scan window set by the user is less than the Scan interval, the Link Layer will automatically calculate and obtain a Scan duration

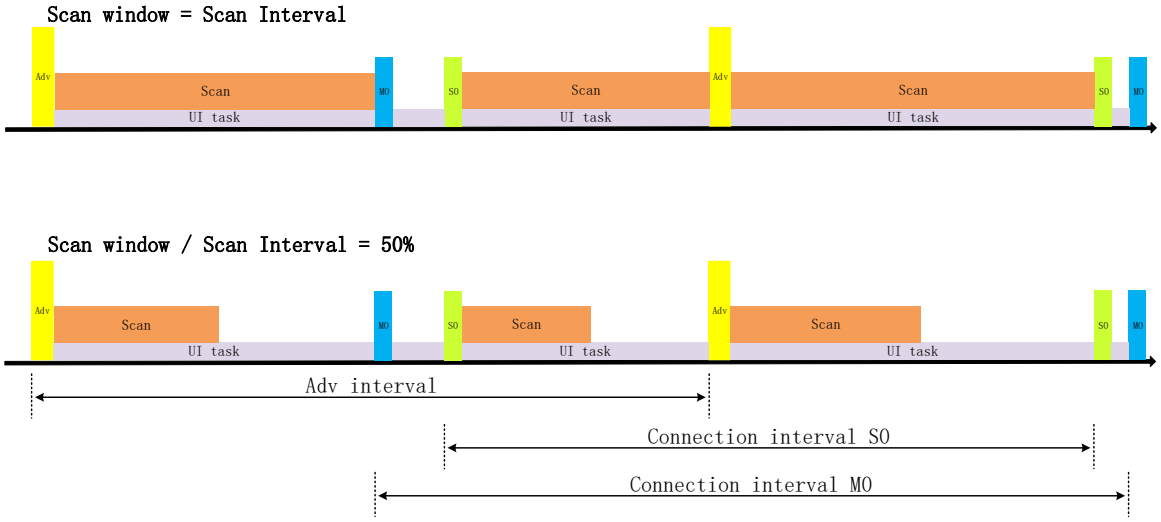
that meets the following conditions: Scan duration / (Adv interval + rand_dly) try to be equal to Scan window /Scan interval.

3.2.3.5 Timing for "Connection, Advertising, Scanning"

The number of Connection connections has not reached the set maximum, and there are still advertising and scanning states at this time.

The following figure corresponds to the 1C2C state of M4S3 in Table 3-2.

Figure 3-13 Timing Sequence of M4S3 1C2C



First of all, the assignment of connection tasks (whether master or slave) will be assigned in accordance with the timing of their respective connections. If multiple tasks occupy the same time period and conflict occurs, they will be allocated according to the priority level, and high priority will be preempted. Abandoned tasks will automatically increase the priority to ensure that they will not be discarded all the time.

Then carry on the assignment of adv task, the principle is:

1. The time interval from the last adv event is greater than the set minimum adv interval time.
2. The time between the next task is greater than a certain value (3.75ms), because the adv needs a certain time to complete.
3. The allocated time period is not occupied by other connection tasks

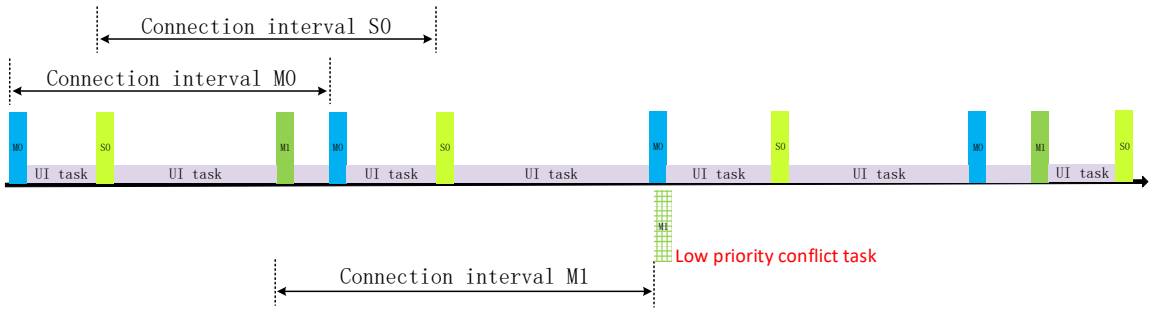
Finally, the principle of the scan task is allocated: as long as there is sufficient time between the two tasks (stack limits the minimum time, do not scan again if the time is too short), this time will be allocated to the scan task, the same Confirm the scan percentage according to Scan window / Scan interval.

3.2.3.6 Timing for "Connection, no Advertising, no Scanning"

The number of Connection connections has reached the set maximum, and there are no advertising and scanning states at this time.

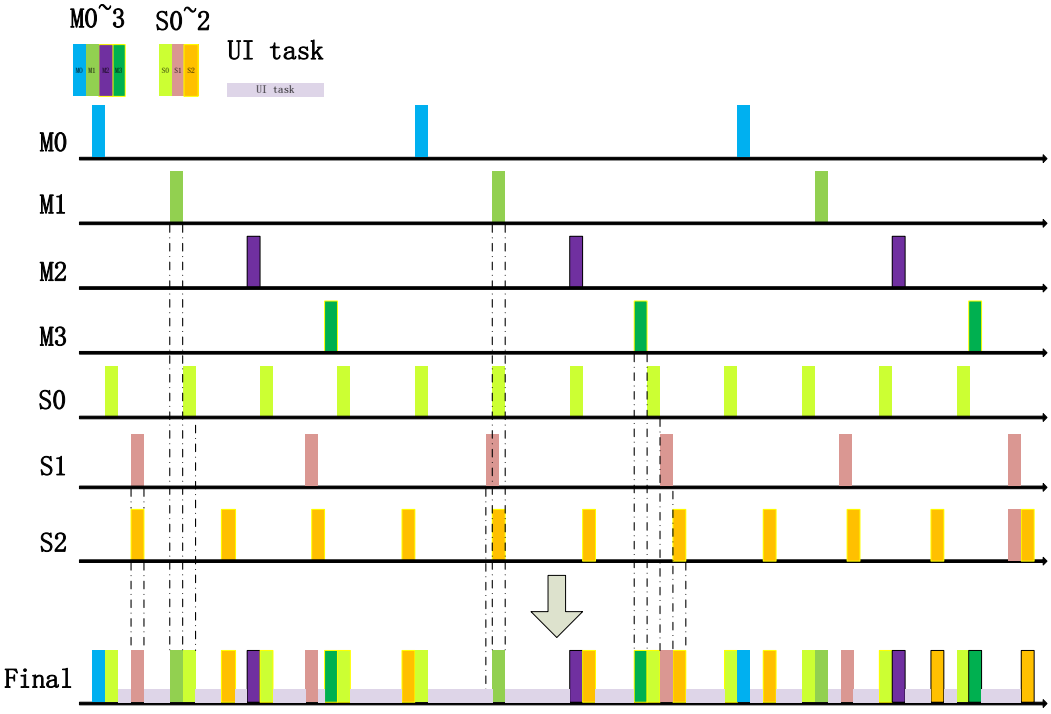
The following figure corresponds to the 1F2H state of M4S3 in Table 3-2.

Figure 3-14 Timing Sequence of M4S3 1F2H



The following figure corresponds to the 1E2F state of M4S3 in Table 3-2.

Figure 3-15 Timing Sequence of M4S3 1E2F



At this time, there are only connection tasks, and the tasks are allocated according to the timing of their connection. If a conflict occurs, the priority is assigned to the high and low tasks, and the high-priority task preempts. The abandoned task will automatically increase the priority and increase the probability of preemption in the next conflict.

3.2.4 Link Layer TX FIFO & RX FIFO

All data of the application layer and BLE Host will eventually need to send RF data through the Link Layer of the Controller. In the Link Layer, the corresponding TX FIFO is defined according to the number of connections set by the user.

3.2.4.1 TX FIFO Definition and Configuration

TX FIFO is defined as follows:

```
MULTI_CONN_FIFO_INIT(blt_m_txfifo, 40, 8, MASTER_MAX_NUM);
MULTI_CONN_FIFO_INIT(blt_s_txfifo, 40, 8, SLAVE_MAX_NUM);
```

This means: MASTER_MAX_NUM masters (default is 4), each master has 8 FIFOs, each FIFO is 40bytes; SLAVE_MAX_NUM slaves (default is 3), each slave has 8 FIFOs, each FIFO is 40bytes

The definition of TX FIFO is left at the application layer. The user is defined according to the actual situation, and the master TX FIFO and slave TX FIFO are defined separately. The purpose of this definition is to first cache the data for each connection in its own TX FIFO, and the TX data between each connection will not be mutually interference; second, you can also flexibly define the size of the TX FIFO according to the actual situation, and accordingly reduce the consumption of ram. Such as:

- Slave needs DLE function, but master does not need DLE, so you can define FIFO separately, saving ram space. For the explanation of DLE, please refer to chapter 3.2.6 MTU and DLE Concept and Usage.
- The customer actually uses 3 master and 2 slave, the customer can define 3 master tx fifo and 2 slave tx fifo, thereby reducing the use of ram and saving ram space, as follows: (note: lib still needs to use m4s3 lib) .

```
MULTI_CONN_FIFO_INIT(blt_m_txfifo, 40, 8, 3);
MULTI_CONN_FIFO_INIT(blt_s_txfifo, 40, 8, 2);
```

Below we describe the settings of TX FIFO in various states with a picture to give everyone a more intuitive understanding:

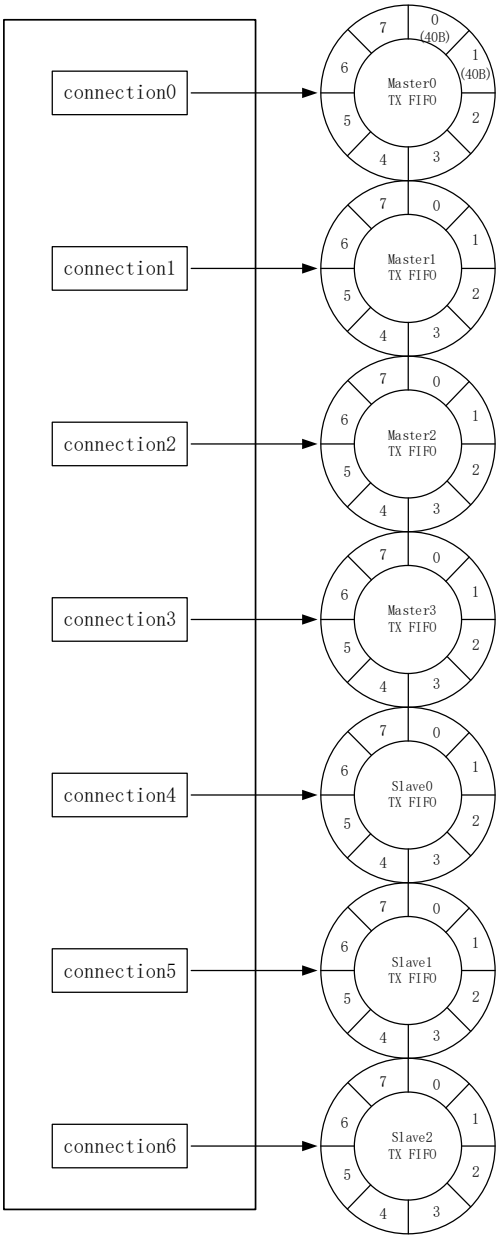
- The default setting of TX FIFO in SDK:

```
MULTI_CONN_FIFO_INIT(blt_m_txfifo, 40, 8, 4);
MULTI_CONN_FIFO_INIT(blt_s_txfifo, 40, 8, 3);
```

Shown as following:

Each connection corresponds to a tx fifo, and the number of each connection fifo is 8 (0 ~ 7), and the size of 0 ~ 7 is the same (40B):

Figure 3-16 Default Setting of TX FIFO

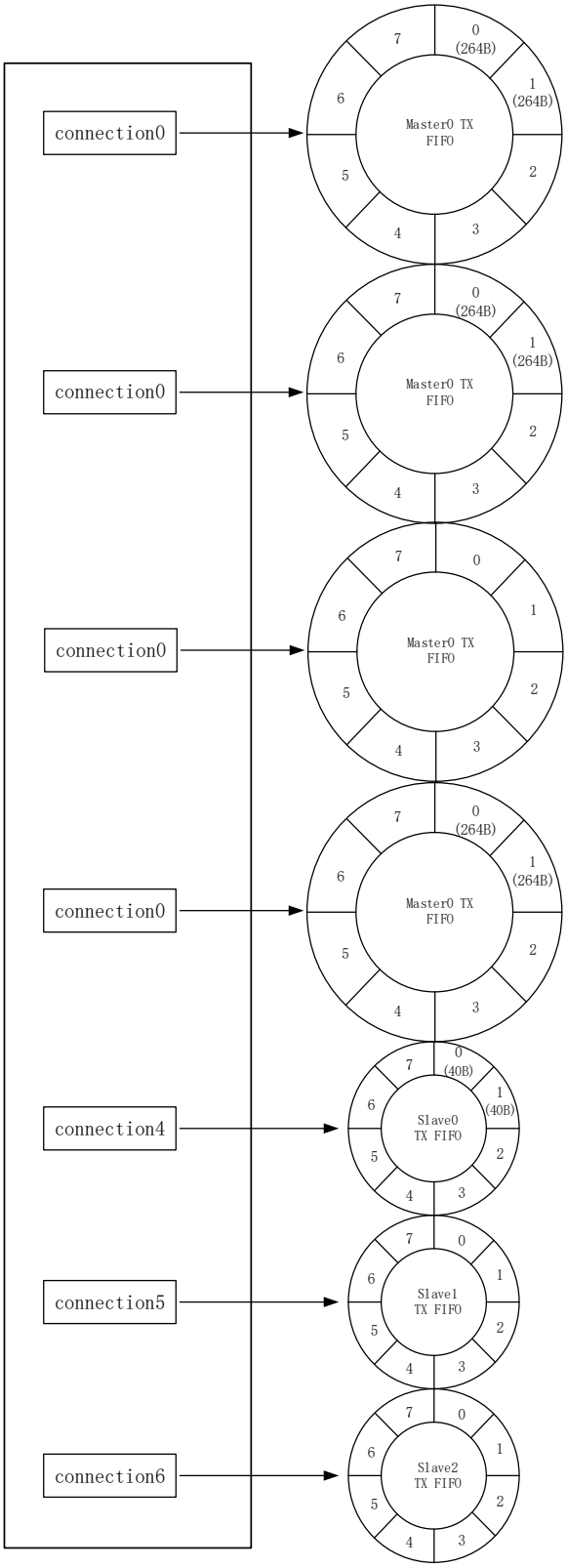


- The case where the master uses DLE and the slave does not use DLE, such as:

```
MULTI_CONN_FIFO_INIT(blt_m_txfifo, 264, 8, MASTER_MAX_NUM);
MULTI_CONN_FIFO_INIT(blt_s_txfifo, 40, 8, SLAVE_MAX_NUM);
```

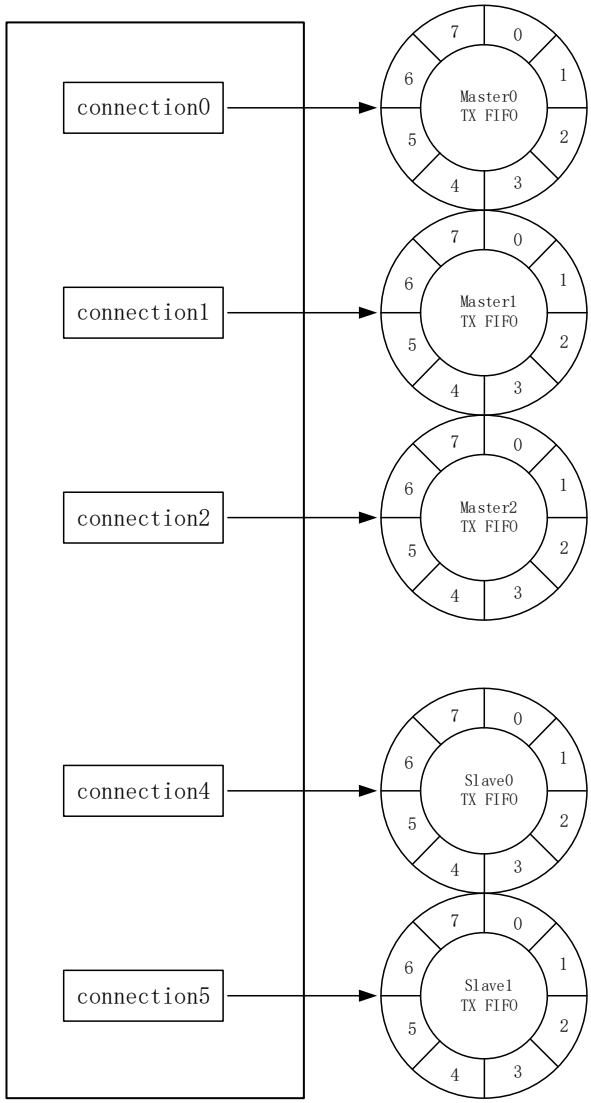
It can be seen from the figure that the number of fifo for each connection of master and slave is the same, which is 8 (0 ~ 7). But each master's fifo size is 264B, and the slave's fifo size is 40B.

Figure 3-17 Buffer Status of Master Using DLE while Slave not



- The customer only used 3 masters, and 2 slaves, shown as following:

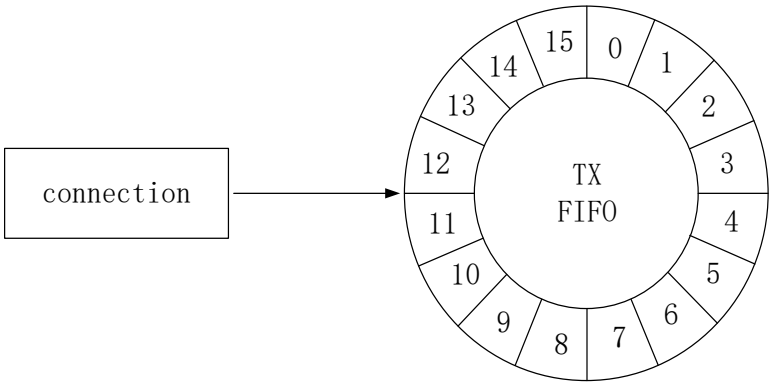
Figure 3-18 Buffer Status When Client using 3 Masters and 2 Slaves



Check the definition of TX FIFO in the single connection SDK:

```
u8 blt_txfifo_b[TX_FIFO_SIZE * TX_FIFO_NUM];
```

Figure 3-19 TX Buffer of Single Connection



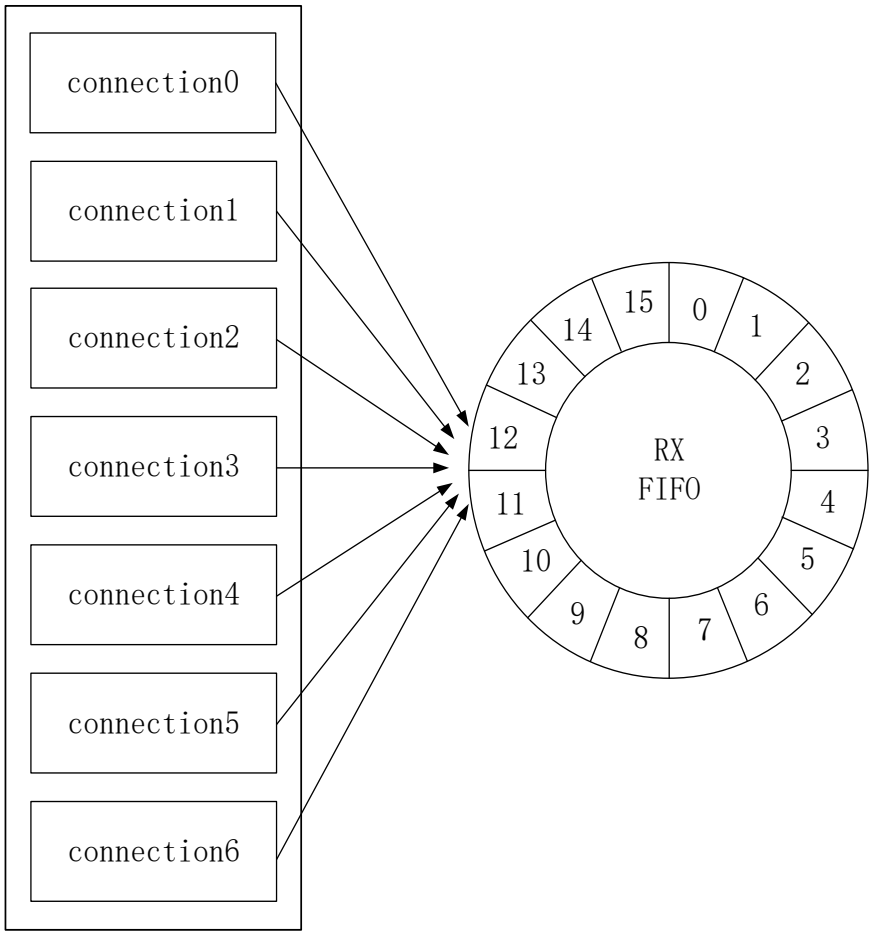
3.2.4.2 RX FIFO Definition and Configuration

For the RX FIFO, only one set of FIFO is currently defined in the SDK, that is, all advertisements, scans, masters, and slaves share the RX FIFO. It is also defined by the user at the application layer, and the Scan RX FIFO and Connection RX FIFO will be defined in the future, which can also save some ram space. For example: when using DLE in Connection, because the length of Scan Data is fixed, it will not Varies according to DLE.

Multiple SDK RX FIFO is defined by the user at the application layer:

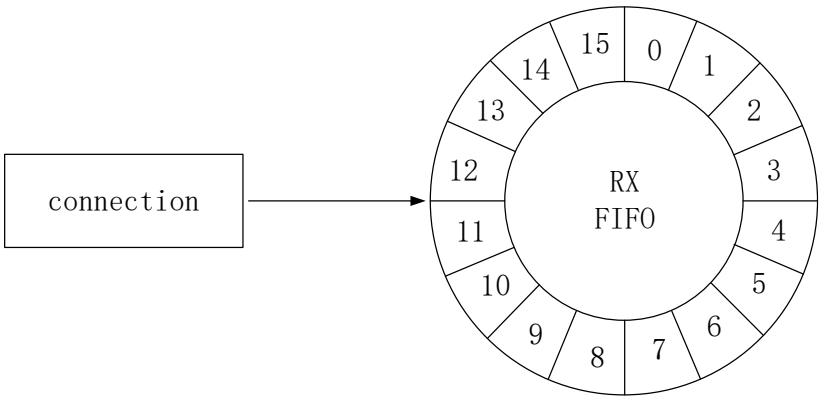
```
MYFIFO_INIT(blt_rxfifo, 64, 16);
```

Figure 3-20 RX Buffer Setting



Check definition of RX FIFO with single connection:

Figure 3-21 RX Buffer of Single Connection



The data of all peer devices received during Link Layer brx / btx will be stored in a BLE RX FIFO first, and then uploaded to the BLE Host or application layer for processing.

Among them, RX FIFO size defaults to 64, TX FIFO size defaults to 40, unless you need to use data length extension, otherwise it is not allowed to modify these two sizes.

3.2.4.3 RX overflow Analysis

No matter whether it is TX FIFO number or RX FIFO number, it must be set to a power of 2, that is, 2, 4, 8, 16 and other values. User can modify it slightly according to his needs.

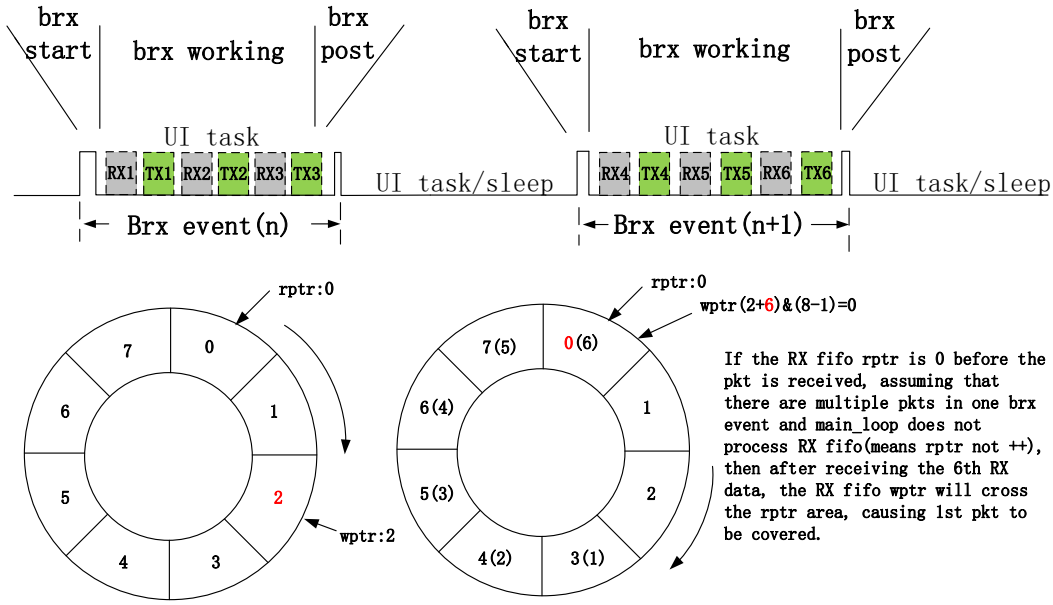
The default RX FIFO number is 16, which is a reasonable value, which can ensure that the bottom layer of the Link Layer caches up to 16 data packets. If the setting is too large, it will take up too much Sram. If the setting is too small, there may be a risk of data coverage: especially in the brx event (btx event can be controlled), Link Layer is likely to appear more data on an interval (MD) mode, continuous reception of multiple packets, if set to 8, it is likely that five or six packets will appear on an interval (such as OTA, voice data transmission, etc.), and due to multiple connections, Link Layer timing is relatively dense, it may be that the RX FIFO has cached multiple connected data packets, and the upper layer's response to these data is too late to process due to the longer decryption time, so there may be some data that is overflowed. For the description of RX overflow, please refer to the description of the relevant chapters in the single connection SDK. Here is a brief reference:

Here are a few examples of RX overflow, we have the following assumptions:

1. The number of RX FIFO number is 8 (defined as 8 is to facilitate the understanding of the RX overflow icon below)
2. Before brx_event (n) is turned on, the read and write pointers of the RX FIFO are 0 and 2 respectively
3. In the brx_event (n) and brx_event (n + 1) phases, there is a task block in the main_loop, and the RX FIFO is not taken in time; Note: brx_event (n) and brx_event (n + 1) may not be the same connection event, such as: brx_event (n) is slave0, and brx_event (n + 1) is slave1.
4. Both brx_event stages are multi-packet.

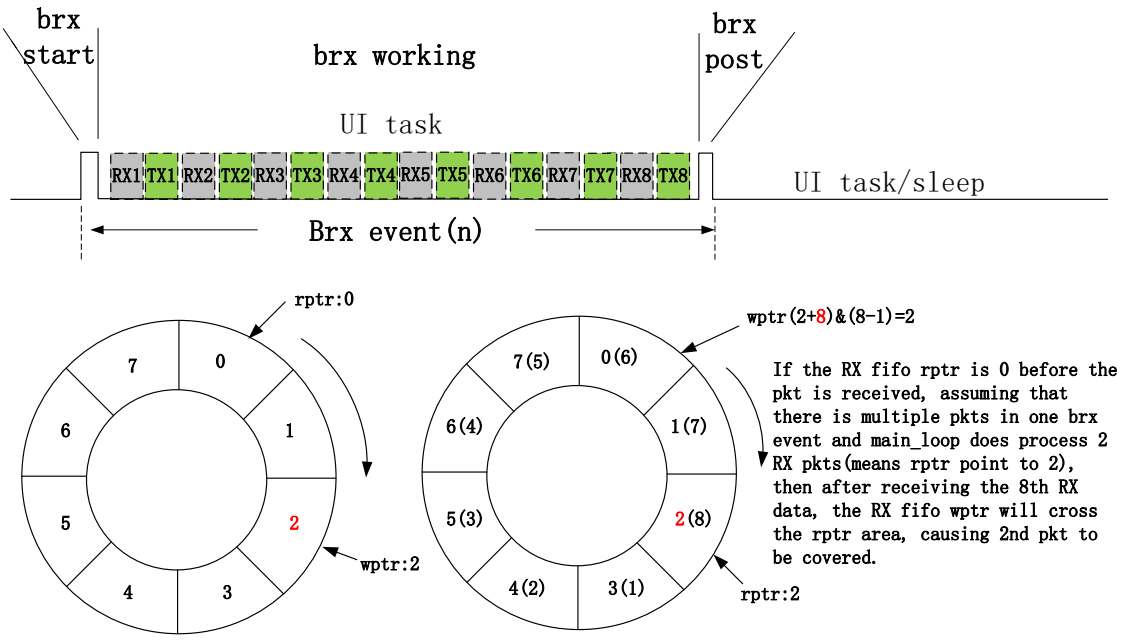
The BLE data packets received in the brx_working stage will only be copied to the RX FIFO (RX FIFO write pointer ++), and the real RX FIFO data is processed for processing operations in the main_loop stage (RX FIFO read pointer ++) , the sixth data will cover the read pointer 0 area. Please be noted here that the UI task time slot in the brx working stage is the time except for RX, TX, system timer and other interrupt processing.

Figure 3-22 RX Overflow Diagram 1



In the above example, because there is a connection interval, the task blocking time must be long enough, which is a bit extreme. The following RX overflow situation has a relatively higher probability: during a brx_event, the master writes multiple data to the slave For example, the number of multi-packets is 7 or 8. In this case, because the master sends a lot of data at once, the slave has no time to process it. As shown in the figure below, the read pointer has moved only 2 strokes, but the write pointer moved 8 strokes will also cause data overflow.

Figure 3-23 RX Overflow Diagram 2



Similarly, if there is an example of an interval with more than 8 valid data packets, the number of 8 is not enough.

Once the problem of data loss caused by overflow occurs, for the encryption system, there will be a problem of MIC failure disconnection. Therefore, users need to avoid that the peer device sends too much data in a connection interval, and the UI task processing time is also as short as possible to avoid blocking problems.

At present, the SDK has Rx overflow verification: check whether the difference between the current RX FIFO write pointer and read pointer is greater than the RX FIFO number in the brx event / btx event Rx IRQ. Once the RX FIFO is found to be full, let RF not ACK the other party. The BLE protocol will ensure data retransmission.

The default TX FIFO number of each connection is 8. If the setting is too large (such as 16), it will take up too much Sram.

In the TX FIFO, two SDK bottom stacks are used, and the rest are completely used by the application layer. When the TX FIFO is 8, the application layer can only use 6.

Before the user sends data in the application layer (such as calling blc_gatt_pushHandleValueNotify), it should check how many TX FIFOs are available in the current Link Layer.

The following API is used to determine how many TX FIFOs are currently occupied, noting how many are available.

```
u8    blc_llms_getTxFifoNumber (u16 connHandle);
```

For example, when the TX FIFO number defaults to 8, the user can use 6, so as long as the value returned by the API is less than 6, it is available: a return of 5 indicates that 1 is still available, and a return of 0 indicates that 6 are still available.

For TX FIFO, if customer checks how many FIFO is left first, then decides whether to directly push the data, a FIFO should be left to prevent various boundary problems.

Below is an extreme example, it is known that a long piece of data will be split into 5 packets and 5 TX FIFOs are required. In order to avoid abnormal conditions caused by the use of TX FIFOs (such as just catching up with the BLE stack to reply to the master command, a piece of data is inserted into the TX FIFO). At this time, the 6 FIFOs reserved for the application must not be occupied. The final code is as follows:

```
if (blc_llms_getTxFifoNumber(connHandle) < 1)
        {
            ......
        }
```

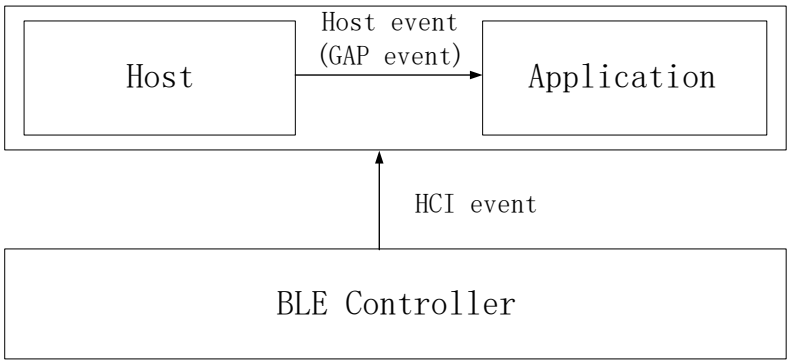
3.2.5 Controller event

In order to satisfy the user's recording and processing of key actions at the bottom of the multiple connection BLE stack at the application layer, the SDK provides two types of events: one is the standard HCI event defined by the BLE controller; the other is some protocol stack processes defined by the BLE host Interactive event notification type GAP event (also can be considered as host event, please refer to the "3.5 GAP" chapter of this document for specific introduction).

Note: On the single connection SDK, telink provides a set of self-defined controller events. Most of the HCI events specified in the spec are the same. In the multiple connection SDK, the repeated Telink-defined events are removed. The user can use the standard events.

The BLE SDK event architecture is shown in the following figure. As shown in the figure that the HCI event belongs to the Controller event, and the GAP event belongs to the BLE host event. The following section mainly introduces Controller event.

Figure 3-24 Controller Event

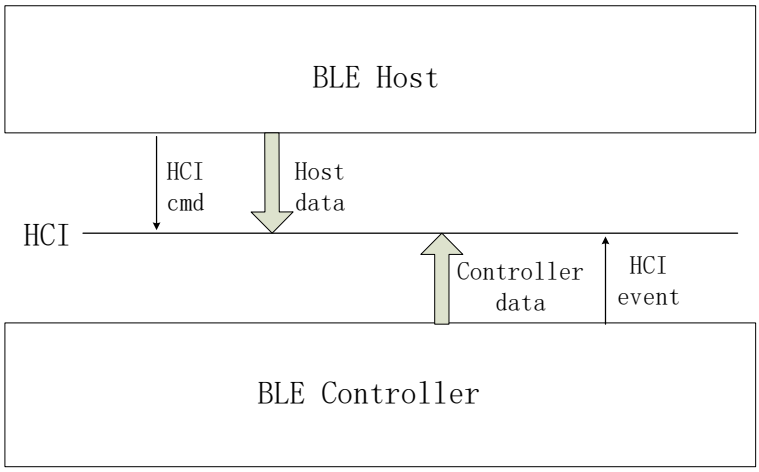


3.2.5.1 Controller Event Definition and Classification of Controller Events

Controller HCI event is designed according to BLE Spec standard.

As shown in the Host + Controller architecture below, the Controller HCI event reports all Controller events to the Host through HCI.

Figure 3-25 Host + Controller Structure



For the definition of Controller HCI event, please refer to "Core_v5.0" (Vol 2/Part E/7.7 "Events") for details. Among them, 7.7.65 "LE Meta Event" refers to HCI LE (low energy) Event, others are ordinary HCI events. According to the Spec definition, Telink multiple connection BLE SDK also divides Controller HCI events into two categories: HCI Event and HCI LE event. Since the Telink BLE SDK focuses on Bluetooth low energy, only the most basic HCI events are supported, and most of HCI LE events are supported.

For macro definitions and interface definitions related to Controller HCI event, please refer to the header files in the stack/ble/hci directory. If user needs to receive the Controller HCI event at the Host or App layer, first, register the callback function of the Controller HCI event, then, open the mask of the corresponding event. For the mask opening API, see the event analysis below.

The prototype and registration interface of the callback function of the Controller HCI event are:

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

The u32 h in the callback function prototype is a mark, frequently used in the underlying protocol stack. The user only needs to know the following one:

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

The HCI_FLAG_EVENT_BT_STD flag indicates that the current event is a Controller HCI event.

In the callback function prototype, *para and n represent the event data and data length, which are consistent with the definition in the BLE spec. User can refer to the following usage in m4s3 demo and the specific implementation of app_controller_event_callback function.

```
blc_hci_registerControllerEventHandler(app_controller_event_callback);
```

3.2.5.2 HCI Event

Some of HCI events are supported in the Telink BLE SDK. The following is a list of events that the user need to know.

```
#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

For details, please refer to "Core_v5.0" (Vol 2/Part E/7.7.5 "Disconnection Complete Event").

The total data length of the event is 7, param len is 4, as shown below, please refer to the BLE spec for the specific data meaning.

Figure 3-26 Packet Format of DISCONNECTION_COMPLETE

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

2. HCI_EVT_ENCRYPTION_CHANGE 和 HCI_EVT_ENCRYPTION_KEY_REFRESH

For details, please refer to "Core_v5.0" (Vol 2/Part E/7.7.8 & 7.7.39).

For Controller encryption, the specific processing is packed in the library, and details are not described here.

3. HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE

For details, please refer to "Core_v5.0" (Vol 2/Part E/7.7.12).

When the Host uses the HCI_CMD_READ_REMOTE_VER_INFO command, the Controller and the BLE peer device exchange version information, after receiving the version of the peer device, reports the event to the Host.

The total data length of the event is 11, param len is 8, as shown below, please refer to the BLE spec for the specific data meaning.

Figure 3-27 Packet Format of READ_REMOTE_VER_INFO_COMPLETE

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

4. HCI_EVT_LE_META

Indicates that it is a HCI LE event, and the specific event type is determined according to the sub event code behind.

Besides HCI_EVT_LE_META, all other HCI events must open the event mask via the following API.

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

The event mask definition is as follows:

```
#define HCI_EVT_MASK_DISCONNECTION_COMPLETE              0x0000000010
#define HCI_EVT_MASK_ENCRYPTION_CHANGE                   0x0000000080
#define HCI_EVT_MASK_READ_REMOTE_VERSION_INFORMATION_COMPLETE
                                                 0x0000000800
```

If the user does not set the HCI event mask through this API, the SDK only opens the mask corresponding to HCI_CMD_DISCONNECTION_COMPLETE by default, which ensures the reporting of the Controller disconnect event.

3.2.5.3 HCI LE Event

When the event code in the HCI event is HCI_EVT_LE_META, it is the HCI LE event. The subevent code is the most commonly used and the user may need to know as follows. Others will not be introduced.

```
#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
```

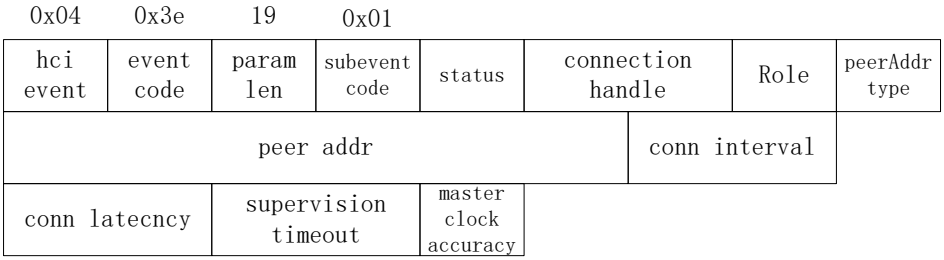
1. HCI_SUB_EVT_LE_CONNECTION_COMPLETE

For details, please refer to "Core_v5.0" (Vol 2/Part E/7.7.65.1 "LE Connection Complete Event").

When the controller link layer and the peer device establish connection, the event is reported.

The overall data length of this event is 22, and the param len is 19, as shown below. For the specific data meaning, please refer to BLE spec directly.

Figure 3-28 Packet Format of CONNECTION_COMPLETE



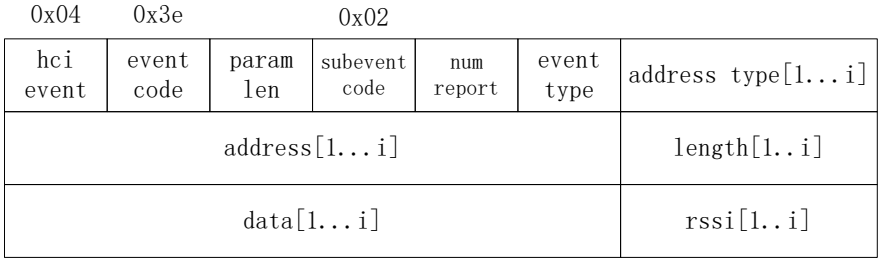
2. HCI_SUB_EVT_LE_ADVERTISING_REPORT

For details, please refer to "Core_v5.0" (Vol 2/Part E/7.7.65.2 "LE Advertising Report Event").

When the controller's Link Layer scan reaches the correct adv packet, it is reported to the Host via HCI_SUB_EVT_LE_ADVERTISING_REPORT.

The data length of this event is variable (depending on the payload of adv packet), as shown below, please refer to the BLE spec for the specific data meaning.

Figure 3-29 Packet Format of ADVERTISING_REPORT



Note: The LE Advertising Report Event in the Telink multiple connection BLE SDK only reports one adv packet at a time, that is, i in the figure above is 1.

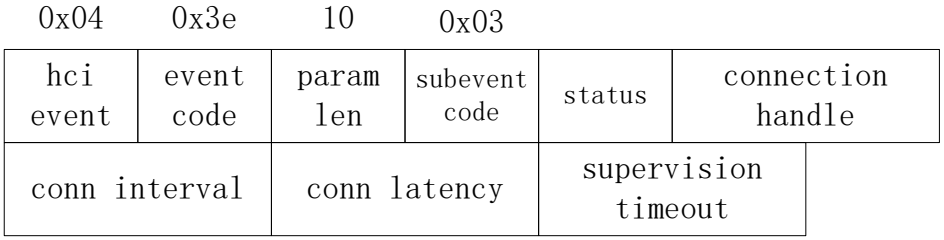
3. HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE

For details, please refer to "Core_v5.0" (Vol 2/Part E/7.7.65.3 "LE Connection Update Complete Event").

When the connection update on the Controller takes effect, report HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE to the Host.

The overall data length of this event is 13, param len is 10, as shown below, please refer to the BLE spec for the specific data meaning.

Figure 3-30 Packet Format of CONNECTION_UPDATE_COMPLETE



4. HCI_SUB_EVT_LE_CONNECTION_ESTABLISH

HCI_SUB_EVT_LE_CONNECTION_ESTABLISH is a supplement to HCI_SUB_EVT_LE_CONNECTION_COMPLETE, except the subevent code, all other parameters are the same.

This event is the only non-BLE spec standard event, which belongs to Telink's private definition.

The reason why Telink defines the event is explained in detail below.

When the BLE Controller in Initiating state detects the device adv packet that needs to be connected, it sends a connection request packet to it. At this time, regardless of whether the other party receives this connection request, it will unconditionally take for granted that the Connection completes and report LE Connection Complete to the Host Event, and Link Layer will then enter the Master role immediately.

Since this packet does not have an ack/retry mechanism, there is no guarantee that the Slave will receive it. If the Slave misses this connection request, it will not be able to enter the Slave role, nor will it enter the brx mode to send and receive packets.

When this happens, the processing mechanism on the Master Controller side is: after entering the Master role, check whether any slave packets have been received on the first 6 to 10 conn intervals (the CRC is correct or not is irrelevant at this time).

- If none of the packets are received, it is considered that the Slave has not received the connection request. On the premise that the LE Connection Complete Event has been reported before, it must quickly report a Disconnection Complete Event and indicate that the disconnect reason is 0x3E (HCI_ERR_CONN_FAILED_TO_ESTABLISH)
- If in the first 6 ~ 10 conn intervals, Slave packets are received, then it can be determined that connection is established (connection established), the process behind the Master can continue to proceed.

According to the above description, the processing method of BLE Host should be: after receiving the LE Connection Complete Event of the Controller, you can not take for granted that the connection has been Established, you must start a timer according to the conn interval (set longer time, more than 10 intervals, covering The longest time), within this timer, check whether there is a disconnect reason of 0x3E Disconnection Complete Event, if not, it can be regarded as connection established.

Because the process of BLE host is very complicated, it is easy to make mistakes, so the SDK defines HCI_SUB_EVT_LE_CONNECTION_ESTABLISH at the bottom layer. When the Host receives this event, it indicates that the Controller has determined that the connection on the slave side is OK. It can continue the following process.

HCI LE event needs to open the mask through the following API.

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

The definition of evtMask will also be decribed, other events users can find it in 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 the user does not set the HCI LE event mask through this API, all HCI LE events are disabled by default.

3.2.6 MTU and DLE Concept and Usage

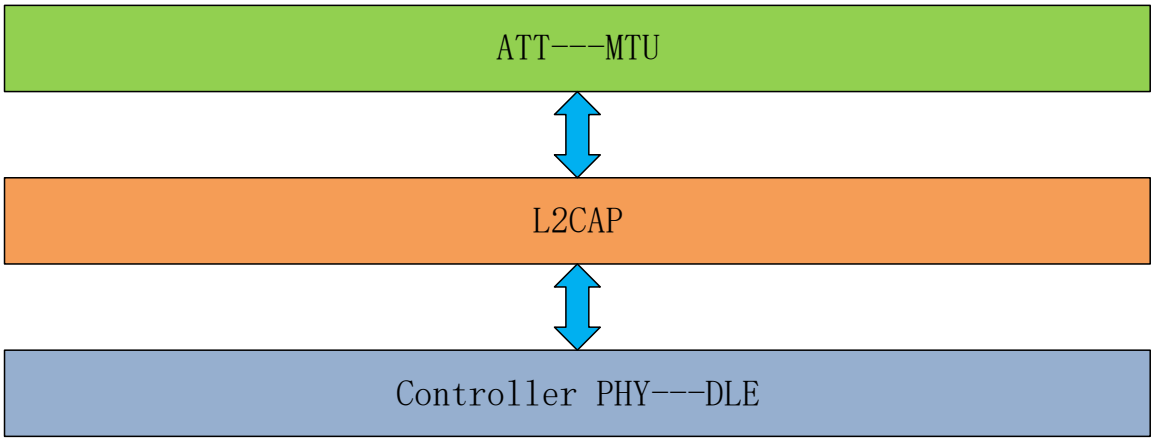
3.2.6.1 MTU and DLE Description of MTU and DLE

BLE Spec added data length extension (DLE) from core_4.2.

Multiple SDK Link Layer supports data length extension, and rf_len length supports the maximum length of 251 bytes on BLE spec. For details, please refer to "Core_v5.0" (Vol 6/Part B/2.4.2.21 "LL_LENGTH_REQ and LL_LENGTH_RSP").

MTU and DLE are shown in the following figure.

Figure 3-31 MTU and DLE



1. MTU means the length of ATT PDU.

   ATT packet format:

Figure 3-32 ATT Packet Format

Name	Size (octets)	Description
Attribute Opcode	1	The attribute PDU operation code bit7: Authentication Signature Flag bit6: Command Flag bit5-0: Method
Attribute Parameters	0 to (ATT_MTU - X)	The attribute PDU parameters X = 1 if Authentication Signature Flag of the Attribute Opcode is 0 X = 13 if Authentication Signature Flag of the Attribute Opcode is 1
Authentication Signature	0 or 12	Optional authentication signature for the Attribute Opcode and Attribute Parameters

MTU includes Opcode + Parameters + Authentication Signature (Optional), BLE stipulates the minimum MTU is 23 bytes.

2. For data length extension (DLE), the actual extension is the length of a single packet sent by the link layer RF. which is:

Figure 3-33 Link Layer Packet Format

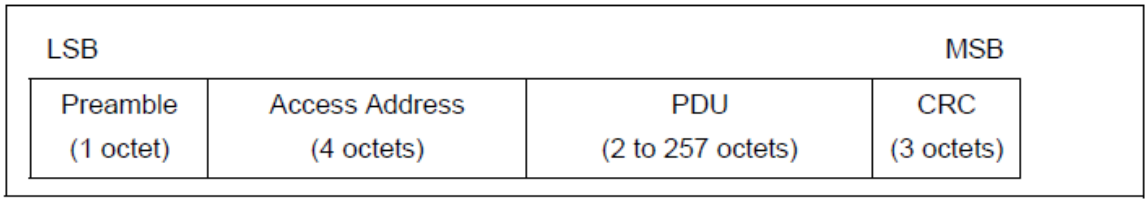


Figure 2.1: Link Layer packet format

Expand the PDU field, the structure is as follows:

Figure 3-34 Data Channel PDU

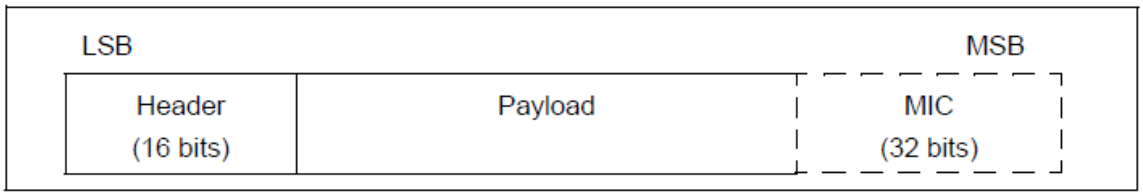


Figure 2.12: Data Channel PDU

As you can see from the link layer packet format above, the length of a single PDU can be from 2B to 257B. But by default, the maximum length of the above PDU sent by a single packet is 29B (27B L2CAP packet + 2B link layer header), that is, the payload length is 27B.

If you want to carry more data in a packet, you need to negotiate between the master and slave, through LL_LENGTH_REQ and LL_LENGTH_RSP interaction in the above figure, the length of the Payload field, i.e.: L2CAP packet.

The spec specifies that the minimum length of the Payload field is 27 bytes and the maximum is 251 bytes. 251bytes is because the rf length field is a byte, and the maximum length that can be expressed is 255. If it is an encrypted link, a 4 bytes MIC field is also required: 251 + 4 = 255.

3.2.6.2 How to use MTU and DLE

If the user needs to use the data length extension function, set it as below. The SDK also provides the corresponding MTU&DLE demo, refer to the 8258_multi_conn_feature_test project (feature_dle.c)

Define the macro in vendor/8258_multi_conn_feature_test/app_config.h

```
#define FEATURE_TEST_MODE       TEST_LL_DLE
```

1. Set the appropriate TX & RX FIFO size

Long TX and RX FIFO sizes are needed to send and receive long packets. Considering that these FIFOs will occupy a large amount of Sram space, the most appropriate value should be selected when setting the FIFO size to avoid Sram waste. You can refer to the explanation of TX FIFO & RX FIFO in chapter 3.2.4.

Sending long packets requires increasing the TX FIFO size. The TX FIFO size should be at least 12 larger than TX rf_len and must be aligned by 4 bytes. Such as:

```
TX rf_len = 251 byte:
MULTI_CONN_FIFO_INIT(blt_m_txfifo, 264, 8, MASTER_MAX_NUM);
MULTI_CONN_FIFO_INIT(blt_s_txfifo, 264, 16, SLAVE_MAX_NUM);
```

```
TX rf_len = 56 bytes：
MULTI_CONN_FIFO_INIT(blt_m_txfifo, 68, 8, MASTER_MAX_NUM);
MULTI_CONN_FIFO_INIT(blt_s_txfifo, 68, 16, SLAVE_MAX_NUM);
```

To receive long packets, you need to increase the RX FIFO size. RX FIFO size should be at least 24 larger than RX rf_len, and must be aligned by 16 bytes. Such as:

```
RX rf_len = 251 bytes:   MYFIFO_INIT(blt_rxfifo, 288, 8);
RX rf_len = 56 bytes：    MYFIFO_INIT(blt_rxfifo, 80,   8);
```

For example, the maximum length of TX and RX need to support up to 200 bytes, which can be set as follows:

```
MYFIFO_INIT(blt_rxfifo, 224,   8);
MULTI_CONN_FIFO_INIT(blt_m_txfifo, 212, 8, MASTER_MAX_NUM);
MULTI_CONN_FIFO_INIT(blt_s_txfifo, 212, 16, SLAVE_MAX_NUM);
```

2. data length exchange

Before sending and receiving long packets, make sure that the process of data length exchange on BLE connection has been completed.

The data length exchange process is the interaction process of the two packages LL_LENGTH_REQ and LL_LENGTH_RSP on the Link Layer. Either the slave or the master can initiate LL_LENGTH_REQ, and the other party responds to LL_LENGTH_RSP. After the interaction of these two packets, master and slave can know the maximum packet length of TX and RX of each other, and then take the smaller of the two maximum packet lengths to determine the maximum packet length of the current connection.

Regardless of which end initiates the LL_LENGTH_REQ, at the end of the data length exchange process, if blc_hci_registerControllerEventHandler (app_controller_event_callback) is registered, the SDK will generate an HCI event HCI_SUB_EVT_LE_DATA_LENGTH_CHANGE callback.

In this HCI_SUB_EVT_LE_DATA_LENGTH_CHANGE event processing interface, the final maximum TX packet length and RX packet length can be obtained.

In actual applications, the peer device may actively initiate LL_LENGTH_REQ, or may not initiate it. If the peer device does not actively initiate LL_LENGTH_REQ, it needs to be initiated by the local device. The SDK provides APIs for proactively initiating LL_LENGTH_REQ as follows:

```
ble_sts_t blc_llms_exchangeDataLength(u16 connHandle, u8 opcode, u16 maxTxOct);
```

ConnHandle in the API fills in the actual connection handle, such as slave handle 0x44, opcode fills in "LL_LENGTH_REQ", maxTxOct fills the maximum TX packet length supported by the current device, for example, when the maximum TX packet length is 200 bytes, set as following:

```
blc_llms_exchangeDataLength(0x44, LL_LENGTH_REQ , 200);
```

Since the local device does not know whether the peer Master/Slave device will actively send LL_LENGTH_REQ, we recommend a reference method: register for the BLT_EV_FLAG_DATA_LENGTH_EXCHANGE event callback, and start a software timer when the connection is established (such as 2 seconds), if it has not been triggered in 2 seconds after this callback, it means that the peer device has not actively initiated LL_LENGTH_REQ. At this time, the local device calls API blc_llms_exchangeDataLength to initiate LL_LENGTH_REQ.

3. MTU size exchange

In addition to the above data length exchange process, the MTU size exchange process must also be executed to ensure that the large MTU size takes effect to prevent the peer device from being able to process long packets at the BLE l2cap layer. The value of MTU size and the maximum packet length of TX & RX satisfy the following relationship: MTU size >= max(RX rf_len, TX rf_len)-4.

For the implementation of MTU size exchange, please refer to the detailed description in the "ATT & GATT" section of this document --- 3.4.4.7 Exchange MTU Request, Exchange MTU Response, or refer to the demo writing in the 8258_multi_conn_feature_test project (feature_dle.c).

4. The operation of sending and receiving long packets

Please refer to instructions in the "ATT & GATT" section of this document, including Handle Value Notification and Handle Value Indication, Write request and Write Command.

After the above three steps are completed correctly, you can start sending and receiving long packets.

To send a long packet, call the corresponding APIs of the Handle Value Notification and Handle Value Indication of the ATT layer, as shown below, and bring the data address and length to be sent into the following formal parameters "*p" and "len", respectively.

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

To receive long package, it is needed to handle the callback function "w" corresponding to Write request and Write Command. In the callback function, refer to the data pointed to by the formal parameter pointer.

3.2.7 2M PHY

2M PHY is a newly added feature of "Core_5.0", which greatly expands the application scenarios of BLE. 2M PHY (2Mbps) can be used as a data channel in the connected state, which greatly improves the BLE bandwidth.

Note: Coded PHY function is currently not supported.

3.2.7.1 2M PHY Demo Introduction

In the Multiple Connection SDK provided by Telink, in order to save Sram, the 2M PHY is closed by default. If you choose to use this feature, you can manually open it. For the opening method, please refer to the Demo provided by the 8258_multi_conn_feature_test project (feature_2m_phy.c).

For local device, please refer to Demo "8258_multi_conn_feature_test".

Define the macro in vendor/8258_multi_conn_feature_test/app_config.h

```
#define FEATURE_TEST_MODE       TEST_2M_PHY
```

peer Master/Slave device can refer to the single connection SDK Demo "8258_master_kma_dongle" project and "8258_feature_test" project (feature_2m_coded_phy_conn.c).

Users can also use equipment from other manufacturers, as long as they support 2M PHY and is compatible with Telink equipment.

If you use Telink single connection SDK, peer Master's 2M PHY is also turned off by default and needs to be turned on by the following method.

Add the following API to the function void user_init(void) in vendor/8258_master_kma_dongle/app.c (the SDK is closed by default): blc_ll_init2MPhyCodedPhy_feature();

3.2.7.2 2M PHY API Introduction

1. API blc_ll_init2MPhyCodedPhy_feature()

```
void blc_ll_init2MPhyCodedPhy_feature(void);
```

Used to enable 2M PHY, Coded PHY is not supported yet.

2. API blc_ll_setPhy()

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

BLE Spec standard interface, please refer to "Core_v5.0" (Vol 2/PartE/7.8.49 "LE Set PHY Command") for details.

connHandle: Master/Slave connHandle is filled according to the actual situation, refer to '3.2.1.4 Connection Handle'

For other parameters, please refer to the Spec definition, combined with the enumeration type definition and demo usage on the SDK to understand.

3.2.8 Channel Selection Algorithm #2

Channel Selection Algorithm #2 is a newly added Feature in "Core_5.0", which has stronger anti-interference ability. Please refer to "Core_5.0" (Vol 6/Part B/4.5.8.3 for the specific description of channel selection algorithm) Channel Selection Algorithm #2")

In the Multiple Connection SDK, CSA #2 is closed by default. If you choose to use this feature, you can manually open it. For the opening method, please refer to the Demo provided by the 8258_multi_conn_feature_test project (feature_csa2.c).

Define the macro in vendor/8258_multi_conn_feature_test/app_config.h

```
#define FEATURE_TEST_MODE TEST_CSA2
```

If you choose to use CSA #2, you need to enable the following API in user_init().

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

Only when local device and peer Master/Slave device both support CSA #2 (ChSel field is set to 1), CSA #2 can be used for connection.

3.2.9 Link Layer API

In the standard BLE protocol stack architecture, the application layer cannot directly communicate with the Link Layer of the Controller, and the data must be sent down through the Host, and the Host transmits the control commands to the Link Layer through HCI in the end. All Controller control commands issued by the Host through the HCI interface are strictly defined in the BLE spec "Core_v5.0". For details, please refer to "Core_v5.0" (Vol 2/Part E/Host Controller Interface Functional Specification).

The Whole Stack architecture of Telink BLE Multiple Connection SDK is shown in the Figure 3-4. The application layer directly sets the Link Layer across the Host, but the APIs used are strictly in accordance with

the HCI part of the Spec standard. The following API specific introduction will give the corresponding Host command on the Spec, users can refer to the specific instructions on the Spec to understand.

The declaration of the Controller API is in the header file in the stack/ble/llms directory, which is divided into llms.h, llms_adv.h, llms_scan.h, llms_init.h, llms_slave.h, llms_master.h according to the classification of the Link Layer state machine function, llms_conn.h, user can find according to the functions of the Link Layer, for example, APIs related to advertising functions should be declared in llms_adv.h.

The enumeration type ble_sts_t is defined in stack/ble/ble_common.h. This type is used as the return value type of most APIs in the SDK. Only when the setting parameters of the calling API are correct and accepted by the protocol stack, the BLE_SUCCESS (value 0) will be returned; all other non-zero values returned indicate a setting error, and each different value corresponds to an error type. The following API detailed description will list all possible return values of each API and explain the specific error reasons of each error return value.

The return value type ble_sts_t is not limited to the Link Layer API, but also applies to some APIs of the Host layer.

3.2.9.1 BLE MAC address initialization

The most basic types of BLE MAC address in this document include public address and random static address.

Call the following API to obtain the public address and random static address.

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

flash_addr only needs to fill in the address of the MAC address stored in the flash, please refer to the introduction of '2.1.3 SDK FLASH space allocation'. If you don't need random static address, just fill in "NULL" in mac_random_static above.

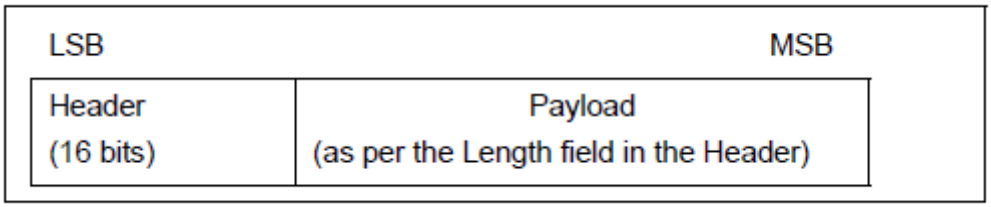
After the BLE public MAC address is successfully obtained, call the API initialized by the Link Layer and pass the MAC address into the BLE protocol stack:

```
blc_llms_initStandby_module (mac_public); //mandatory
```

3.2.9.2 blc_llms_setAdvData

For details, please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.7 "LE Set Advertising Data Command").

Figure 3-35 Protocol STACK BROADCAST PACKET FORMAT



In the BLE protocol stack, the format of the advertising packet is shown in the figure above. The first two bytes are the header, followed by the payload (PDU), with a maximum of 31 bytes.

The following API is used to set the data of the PDU part:

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

The data pointer points to the first address of the PDU, and len is the data length.

The possible results returned by the return type ble_sts_t are shown in the table below.

Table 3-6 Return Value of blc_llms_setAdvData

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	

The value ble_sts_t return is only BLE_SUCCESS, the API will not check the plausibility of the parameter, the user needs to pay attention to the rationality of the parameter setting.

The user can call the API to set the advertising data during initialization, or call the API at any time in the main_loop to modify the advertising data while the program is running.

The Adv PDU defined in the 8258_m4s3 project in the SDK is as following. For the meaning of each field, please refer to the specific description of the Data Type Specifcation in the document BLE Spec "CSS v6" (Core Specification Supplement v6.0).

```
const u8  tbl_advData[] = {
          0x09, 0x09, 'T','L','K','_','M','4','S','3',
          0x02, 0x01, 0x05,
          0x03, 0x19, 0x80, 0x01,
          0x05, 0x02, 0x12, 0x18, 0x0F, 0x18,
};
```

In the above advertising data, set the advertising device name to "TLK_M4S3".

3.2.9.3 blc_llms_setScanRspData

For details, please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.8 "LE Set Scan Response Data Command").

Similar to the setting of the advertising packet PDU above, the scan response PDU setting with the following API:

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

The data pointer points to the first address of the PDU, and len is the data length. The possible results returned by the return type ble_sts_t are shown in the table below.

Table 3-7 Return Value of blc_llms_setScanRspData

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	

The return ble_sts_t value is only BLE_SUCCESS, the API will not check the plausibility of the parameter, the user needs to pay attention to the rationality of the parameter setting.

The user can call the API to set the scan response data during initialization, or call the API at any time in the main_loop while the program is running to modify the scan response data.

The scan response data defined in the 8258_m4s3 project in the SDK is as follows, and the scan device name is "TLK_M4S3". For the meaning of each field, please refer to the specific description of the Data Type Specification in the document BLE Spec "CSS v6" (Core Specification Supplement v6.0).

```
const u8  tbl_scanRsp [] = {
      0x09, 0x09, 'T','L','K','_','M','4','S','3',
};
```

The same device name is set in the advertising data and scan response data above, and the peer master device will scan the same device name.

If the set device names are different, for example, advertising data is "A" and scan response data is "B", then when scanning a BLE device on mobile phone or IOS system, the device names you see may be different:

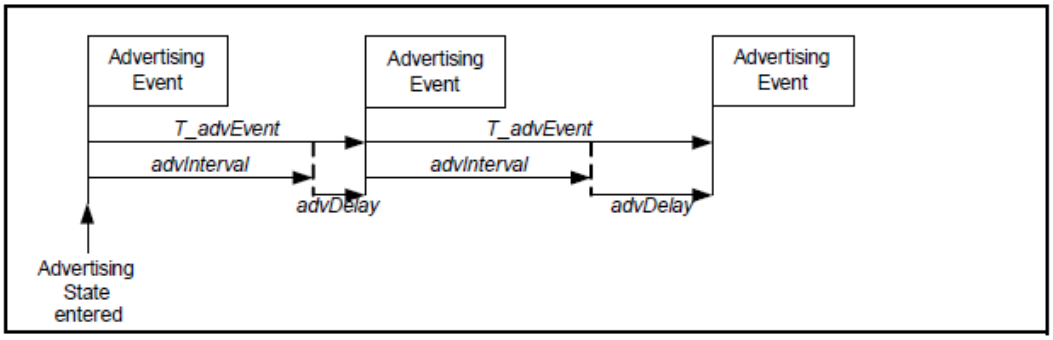
1. Some devices only watch advertising packets, then the device name is displayed as "A";
2. Some devices send scan request and read back the scan response after seeing the advertising, then the displayed device name may be "B".

In fact, after the device is connected by the peer Master device, the master will obtain the gap device name of the device when reading the Attribute Table of the device. After connecting to the device, the device name may be displayed according to the settings there.

3.2.9.4 blc_llms_setAdvParam

For details, please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.5 "LE Set Advertising Parameters Command").

Figure 3-36 Advertising Event in BLE Protocol Stack



The Advertising Event (Adv Event for short) in the BLE protocol stack is shown in the figure above, which means that each T_advEvent, slave performs a round of advertising, and sends a packet on each of the three advertising channels (channel 37, channel 38, channel 39).

The following API sets the parameters related to Adv Event.

```
    ble_sts_t blc_llms_setAdvParam( u16 intervalMin, u16 intervalMax,
                                    adv_type_t      advType,      own_addr_type_t ownAddrType,
                                    u8 peerAddrType, u8 *peerAddr,
adv_chn_map_t adv_channelMap,                                     adv_fp_type_t advFilterPolicy);
```

intervalMin 和 intervalMax intervalMin and intervalMax

Set the range of the advertising interval, 0.625ms as the basic unit, the range is between 20ms ~ 10.24S, and intervalMin is less than or equal to intervalMax.

The SDK uses intervalMin for advertising in connected state and intervalMax for non-connected state.

If the intervalMin is greater than intervalMax, intervalMin will be forced to be equal to intervalMax.

Depending on the type of advertising packet, the values of intervalMin and intervalMax have some restrictions, please refer to (Vol 6/Part B/ 4.4.2.2 "Advertising Events").

advType

Referring to BLE Spec, the four basic types of advertising events are as follows:

Figure 3-37 Four Broadcast Events of BLE Protocol Stack

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

The Allowable response PDUs for advertising event section in the above figure uses YES and NO to explain whether various types of advertising events respond to Scan Request and Connect Request of other devices, such as: the first Connectable Undirected Event can both Scan Request and Connect Request Response, and Non-connectable Undireted Event does not respond to them.

Note that the second Connectable Directed Event responds to Connect Request with a "*" in the upper right corner of the "YES", indicating that as long as it receives a matching Connect Request, it will respond without being filtered by the whitelist. The remaining three "YES" indicate that they can respond to the corresponding request, but actually need to depend on the settings of the whitelist, according to the filter conditions of the whitelist to decide whether to ultimately respond, the whitelist will be described in detail later.

Among the above four advertising events, Connectable Directed Event is divided into Low Duty Cycle Directed Advertising and High Duty Cycle Directed Advertising, so that a total of five types of advertising events can be obtained, as defined below (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;
```

The default most commonly used advertising type is ADV_TYPE_CONNECTABLE_UNDIRECTED.

ownAddrType

When specifying the advertising address type, the four optional values of ownAddrType are as follows.

```
    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;
```

Only the first two parameters are introduced here.

OWN_ADDRESS_PUBLIC means to use public MAC address when advertising, the actual address comes from the setting of API blc_initMacAddress(flash_sector_mac_address, mac_public, mac_random_static) when MAC address is initialized.

OWN_ADDRESS_RANDOM means to use random static MAC address when advertising, the address comes from the value set by the following API:

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

peerAddrType 和*peerAddr peerAddrType and*peerAddr

When advType is set to the direct advertising packet type directed adv (ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY and ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY), peerAddrType and *peerAddr are used to specify the type and address of the peer device MAC Address.

When advType is other types, the values of peerAddrType and *peerAddr are invalid and can be set to 0 and NULL.

adv_channelMap

Set advertising channel, you can choose any one or more of channel 37, 38, 39. The value of adv_channelMap can be set as the following 3 or any combination of them.

```
    typedef enum{
    BLT_ENABLE_ADV_37       =       BIT(0),
    BLT_ENABLE_ADV_38       =       BIT(1),
    BLT_ENABLE_ADV_39       =       BIT(2),
    BLT_ENABLE_ADV_ALL   =      (BLT_ENABLE_ADV_37 |
                                BLT_ENABLE_ADV_38 | BLT_ENABLE_ADV_39),
}adv_chn_map_t;
```

advFilterPolicy

It is used to set the filtering strategy adopted for scan request and connect request of other devices when sending advertising packets. The filtered addresses need to be stored in the whitelist in advance. It will be explained in detail in the whitelist introduction later.

The four types of filtering that can be set are as follows. If you do not need the whitelist filtering function, select ADV_FP_NONE.

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

```
} adv_fp_type_t;
```

The possible values and causes of the return value ble_sts_t are shown in the following table:

Table 3-8 Return Value of advFilterPolicy

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	

The return value ble_sts_t is only BLE_SUCCESS, the API will not check the plausibility of the parameter, the user needs to pay attention to the rationality of the parameter setting.

According to the design of the Host command in the BLE spec HCI part, Set Advertising parameters set the above 8 parameters at the same time. The idea of setting is also reasonable, because there are coupling relationships between some different parameters, such as advType and advInterval. Under different advType, the range limits of intervalMin and intervalMax will be different, so there will be different range checks If set advType and set advInterval are split into two different APIs, the range check between each other cannot be controlled.

3.2.9.5 blc_llms_setAdvEnable

For details, please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.9 "LE Set Advertising Enable Command").

```
    ble_sts_t  blc_llms_setAdvEnable(adv_en_t adv_enable);
```

When en is 1, Enable Advertising; when en is 0, Disable Advertising.

The state machine of Enable or Disable Advertising can refer to "3.2.2.2 Link Layer State Combination".

The possible values and causes of the return value ble_sts_t are shown in the following table:

Table 3-9 Return Value of blc_llms_setAdvEnable

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x0D	appMaxSlaveNum is 0, setting is not allowed

3.2.9.6 blc_llms_setAdvCustomedChannel

This API is used to customize special advertising channel & scanning channel, which is only meaningful for some very special applications, such as BLE mesh.

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

chn0/chn1/chn2 fill in the frequency points that need to be customized. The default standard frequency point is 37/38/39. For example, set three advertising channels to 2420MHz, 2430MHz, and 2450MHz, which can be called as follows:

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

Conventional BLE applications use this API to achieve such functions. If the user wants to use single-channel advertising & single-channel scan in some usage scenarios, such as fixing the advertising channel & scanning channel to 39, it can be called as follows:

```
blc_ll_setAdvCustomedChannel (39, 39, 39);
```

It should be noted that the API will change the advertising and scan channels at the same time.

3.2.9.7 rf_set_power_level_index

The SDK provides the BLE RF packet energy setting API:

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

The setting of the level value refers to the enumeration variable RF_PowerTypeDef defined in drivers/8258/rf_drv.h.

The RF packet energy set by this API is effective for both advertising packets and connection packets, and can be set anywhere in the program. The actual energy when the packet is sent is based on the most recent setting in time.

3.2.9.8 blc_llms_setScanParameter

For details, please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.10 "LE Set Scan Parameters Command").

```
ble_sts_t    blc_llms_setScanParameter (scan_type_t scan_type,
u16 scan_interval, u16 scan_window,
own_addr_type_t ownAddrType,
scan_fp_type_t scanFilter_policy);
```

Parameter analysis:

1) scan_type

You can choose between passive scan and active scan. The difference is that active scan will send scan_req on the basis of adv packet to get more information about device scan_rsp, and scan rsp package will also be passed to BLE Host through adv report event; passive scan will not send scan req.

```
typedef enum {
        SCAN_TYPE_PASSIVE     = 0x00,
        SCAN_TYPE_ACTIVE= 0x01,
} scan_type_t;
```

2) scan_inetrval/scan window

scan_interval sets the switching time of Scanning state time-frequency point, the unit is 0.625ms, and scan_window is the scan window time. If scan_window> scan_interval, the actual scan window is set to scan_interval.

The bottom layer will calculate scan_percent according to scan_window / scan_interval, which has achieved the purpose of reducing power consumption. For scanning specific details, please refer to "3.2.3.2" and "3.2.3.4" and "3.2.3.5".

3) ownAddrType

When specifying the scan req packet address type, ownAddrType has 4 optional values as follows:

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

OWN_ADDRESS_PUBLIC indicates that the public MAC address is used when scanning. The actual address comes from the setting of API blc_initMacAddress(int flash_addr, u8 *mac_public, u8 *mac_random_static) when the MAC address is initialized.

OWN_ADDRESS_RANDOM means to use random static MAC address when scanning, which is derived from the value set by the following API:

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

4) scanFilter_policy

```
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;
```

The currently supported scan filter policies are the following two:

SCAN_FP_ALLOW_ADV_ANY indicates that the Link Layer does not filter the adv packet received by the scan and directly reports to the BLE Host.

SCAN_FP_ALLOW_ADV_WL requires that the scanned adv packet must be in the whitelist before it is reported to the BLE Host.

The possible values and causes of the return value ble_sts_t are shown in the following table:

Table 3-10 Return Value of scanFilter_policy

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	

The return value ble_sts_t is only BLE_SUCCESS, the API will not check the plausibility of the parameter, the user needs to pay attention to the rationality of the parameter setting.

3.2.9.9 blc_llms_setScanEnable

For details, please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.11 "LE Set Scan Enable Command").

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

The scan_enable parameter type has the following 2 optional values:

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

When scan_enable is 1, Enable Scanning; when scan_enable is 0, Disable Scanning.

The state machine change of Enable/Disable Scanning can refer to "3.2.2.2 Link Layer State Combination".

The filter_duplicate parameter type has the following 2 optional values:

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

When filter_duplicate is 1, it means that duplicate packet filtering is enabled. At this time, for each different adv packet, the Controller only reports the adv report event to the Host once; when filter_duplicate is 0, the duplicate packet filtering is disabled, and the adv packet scanned will always be Report to Host.

The return value ble_sts_t is shown in the table below.

Table 3-11 Return Value of blc_llms_setScanEnable

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	

When scan_type is set to active scan (3.2.8.8 blc_llms_setScanParameter) and Enable Scanning, scan_rsp is read only once for each device and reported to the Host. Because after each Enable Scanning, the Controller will record the scan_rsp of different devices and store them in the scan_rsp list to ensure that the scan_req of the device will not be read again later.

If the user needs to report scan_rsp of the same device multiple times, it can be achieved by repeatedly setting Enable Scanning through blc_llms_setScanEnable, because the scan_rsp list of the device will be cleared every time Enable/Disable Scanning is enabled.

3.2.9.10 blc_llms_createConnection

For details, please refer to "Core_v5.0" (Vol 2/Part E/ 7.8.12 "LE Create Connection Command").

```
ble_sts_t blc_llms_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_interval/scan window

scan_interval/scan_window is temporarily not processed in this API. If you need to set these parameters, you can use "3.2.9.8 blc_llms_setScanParameter".

2) initiator_filter_policy

Specify the strategy of the currently connected device. The following two options are available:

```
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 that the connected device address is adr_type/mac behind;

INITIATE_FP_ADV_WL means to connect according to the devices in the whitelist, at this time adr_type/mac is meaningless.

3) adr_type/ mac

When the initiator_filter_policy is INITIATE_FP_ADV_SPECIFY, connect the device with the address type adr_type (BLE_ADDR_PUBLIC or BLE_ADDR_RANDOM) and the address mac[5...0].

4) own_adr_type

Specifies the MAC address type used by the master role to establish a connection. The 4 optional values of ownAddrType are as follows.

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

OWN_ADDRESS_PUBLIC means to use public MAC address when connecting. The actual address comes from the setting of API blc_llms_initStandby_module (mac_public) when MAC address is initialized.

OWN_ADDRESS_RANDOM means to use random static MAC address when connecting, the address comes from the value set by the following API:

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

5) conn_min/ conn_max/ conn_latency/ timeout

These 4 parameters stipulate the connection parameters of the master role after the connection is established, and these parameters will also be sent to the slave through the connection request, and the slave will also be the same connection parameters.

conn_min/conn_max specifies the range of conn interval, the unit is 1.25ms. If appMaxMasterNum > 1, the conn_min/conn_max parameter is invalid, and the Master role conn interval in the SDK is fixed to 25 by default (the actual interval is 31.25ms = 25 ×1.25ms). In this case, you can adjust the blc_llms_setMasterConnectionInterval before setting up the connection; if appMaxMasterNum Is 1, Master role conn interval in SDK directly uses the value of conn_max.

conn_latency specifies connection latency, which is generally set to 0.

timeout specifies the connection supervision timeout, the unit is 10ms.

6) ce_min/ ce_max

SDK has not processed ce_min/ ce_max.

Return value list:

Table 3-12 Return Value of blc_llms_createConnection

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x0D	The Link Layer is already in the Initiating state and no longer receives new create connection or the current device is in the Connection state

The API does not check the plausibility of the parameters, and users need to pay attention to the rationality of the parameters setting.

3.2.9.11 blc_llms_setCreateConnectionTimeout

```
ble_sts_t    blc_llms_setCreateConnectionTimeout(u32 timeout_ms);
```

The return value is BLE_SUCCESS, timeout_ms unit is ms.

After blc_llms_createConnection being triggered and enters the Initiating state, if the connection cannot be established for a long time, it will trigger the Initiate timeout and exit the Initiating state.

The SDK's default Initiate timeout is 5 seconds. If the user does not want to use the default time, you can call blc_llms_setCreateConnectionTimeout to set the Initiate timeout you need.

3.2.9.12 blc_llms_setMasterConnectionInterval

```
ble_sts_t    blc_llms_setMasterConnectionInterval(u16 conn_interval);
```

The return value is BLE_SUCCESS, and the unit of conn interval is 1.25ms.

If appMaxMasterNum> 1, the SDK master role conn interval defaults to 25 (the actual interval is 31.25ms = 25 ×1.25ms), the conn_min/conn_max parameter in blc_llms_createConnection is invalid, in this case, you can adjust blc_llms_setMasterConnectionInterval to change the setting before establishing a connection;

3.2.9.13 blc_llms_disconnect

```
ble_sts_t    blc_llms_disconnect(u16 connHandle, u8 reason);
```

Call this API to send a terminate on the Link Layer to the peer Master/Slave device to actively disconnect.

ConnHandle can refer to "3.2.1.4 Connection Handle".

Reason is the reason for disconnection. For the setting of reason, please refer to "Core_v5.0" (Vol 2/Part D/2 "Error Code Descriptions").

If it is not caused by abnormal system operation, the application layer generally specifies reason as

```
HCI_ERR_REMOTE_USER_TERM_CONN   = 0x13, blc_llms_disconnect(connHandle,
HCI_ERR_REMOTE_USER_TERM_CONN);
```

After calling the API to initiate the disconnection, the HCI_EVT_DISCONNECTION_COMPLETE event will be triggered. In the callback function of this event, you can see that the corresponding terminate reason is the same as the manually set reason.

In general, calling the API directly can successfully terminate and disconnect, but there are also some special circumstances that will cause the API call to fail. According to the return value ble_sts_t, you can understand the corresponding cause of the error. It is recommended that when the application layer calls this API, check whether the return value is BLE_SUCCESS.

The list of return values is as follows.

Table 3-13 Return Value of blc_llms_disconnect

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_UNKNOWN_CONN_ID	0x02	connHandle error or corresponding connection not found
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x3E	A large amount of data is being sent and the command cannot be accepted temporarily

3.2.9.14 Whitelist & Resolvinglist

As mentioned earlier, the Whitelist is involved in the filter_policy of the Advertising/Scanning/Initiating state, and the corresponding operations will be performed according to the devices in the Whitelist. The actual Whitelist concept contains two parts, Whitelist and Resolvinglist.

You can determine whether the peer device address type is RPA (resolvable private address) through peer_addr_type and peer_addr. Use the following macro to judge

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

Only non-RPA addresses can be stored in the whitelist. Currently, the SDK whitelist can store up to 4 devices:

```
#define   MAX_WHITE_LIST_SIZE                    4
```

Whitelist related APIs are as follows:

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

Reset whitelist, the return value is BLE_SUCCESS.

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

Add a device to the whitelist and return values are listed as following:

Table 3-14 Return Value when Adding Equipment to Whitelist

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Added successfully
HCI_ERR_MEM_CAP_EXCEEDED	0x07	whitelist is full, add failed

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

Delete the previously added device from the whitelist, the return value is BLE_SUCCESS.

For RPA (resolvable private address) devices, Resolvinglist is required. To save RAM usage, the SDK Resolvinglist currently stores up to 2 devices:

```
#define    MAX_WHITE_IRK_LIST_SIZE                2
```

Resolvinglist related APIs are as follows:

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

Reset Resolvinglist. The return value is BLE_SUCCESS.

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

Device address resolution is used. If you want to use Resolvinglist to resolve addresses, you must enable it. When no analysis is needed, it can be disabled.

```
ble_sts_t   ll_resolvingList_add(u8 peerIdAddrType, u8 *peerIdAddr,
u8 *peer_irk, u8 *local_irk);
```

Add the device using the RPA address, peerIdAddrType/peerIdAddr and peer-irk fill in the identity address and irk declared by the peer device, this information will be stored in the Flash during the pairing encryption process, the user can find the interface to obtain this information in "3.7 SMP" . For the local_irk SDk, there is no processing temporarily, just fill in NULL.

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

Remove the previously added device. The return value is BLE_SUCCESS.

For the use of Whitelist/Resolvinglist to implement address filtering, please refer to the 8258_multi_conn_feature_test project (feature_whitelist.c).

Define the macro in vendor/8258_multi_conn_feature_test/app_config.h

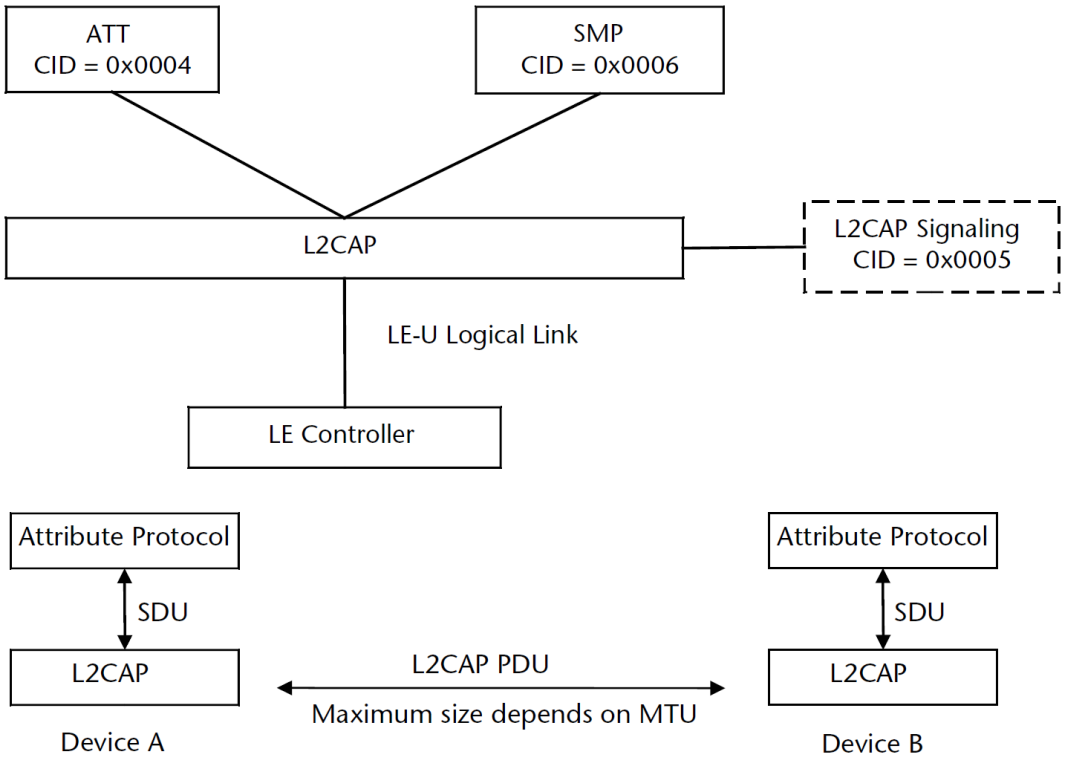
```
#define FEATURE_TEST_MODE       TEST_WHITELIST
```

3.3 L2CAP

The logical link control and adaptation protocol is usually referred to as L2CAP (Logical Link Control and Adaptation Protocol), which connects upward to the application layer and downward to the controller layer, and plays the role of an adapter between the host and the controller to enable the upper layer application to operate No need to care about the data processing details of the controller.

BLE's L2CAP layer is a simplified version of the classic Bluetooth L2CAP layer. In basic mode, it does not perform segmentation and reassembly, does not involve process control and retransmission mechanisms, and uses only fixed channels for communication. The simplified structure of L2CAP is shown in the following figure. Simply put, it is to sub-package the data of the application layer to the BLE controller, package the data received by the BLE controller into different CID data, and report it to the host layer.

Figure 3-38 BLE L2CAP Architecture and ATT Packet Module



L2CAP is designed according to the BLE Spec. The main function is to complete the data connection between the Controller and the Host. Most of them are completed at the bottom of the protocol stack, and there are few places that require user participation. The user can be set according to the following APIs.

3.3.1 Register L2CAP Data Processing Function

In the BLE multiple SDK architecture, the data of the Controller is connected to the Host through HCI. From HCI to Host data, it will first be processed at the L2CAP layer. Use the following API to register the processing function:

```
void        blc_hci_registerControllerDataHandler(void *p);
```

The functions of the multiple SDK L2CAP layer to process Controller data are:

```
int        blt_l2cap_pktHandler(u16 connHandle, u8 *raw_pkt);
```

This function has been implemented in the protocol stack. It parses the received data and transmits it to ATT, SIG, or SMP.

```
initialization：blc_hci_registerControllerDataHandler (blt_l2cap_pktHandler);
```

3.3.2 Update Connection Parameters

1. Slave requests to update connection parameters

In the BLE protocol stack, the slave applies for a new set of connection parameters to the master through the l2cap layer CONNECTION PARAMETER UPDATE REQUEST command. The format of this command is shown below. For details, please refer to "Core_v5.0" (Vol 3/Part A/ 4.20 "CONNECTION PARAMETER UPDATE REQUEST").

Figure 3-39 Connection Para update Req Format in BLE Protocol Stack

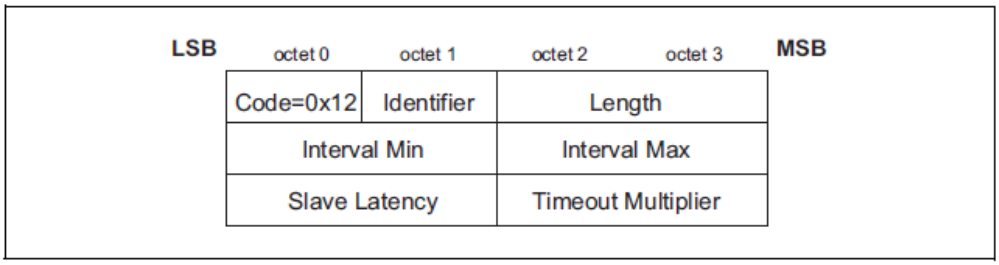


Figure 4.22: Connection Parameters Update Request Packet

The BLE SDK provides an API for slaves to actively apply for updating connection parameters on the L2CAP layer to send the above CONNECTION PARAMETER UPDATE REQUEST command to the master.

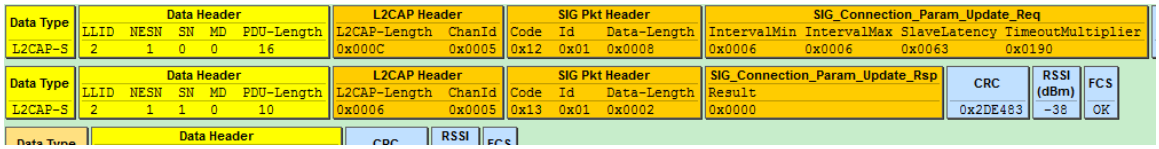
```
    void   bls_l2cap_requestConnParamUpdate (u16 min_interval,
                                             u16 max_interval,
                                          u16 latency, u16 timeout);
```

The implementation of this API is also limited to SLAVE. The above four parameters correspond to the four parameters in the data area of the CONNECTION PARAMETER UPDATE REQUEST. Note that the values of min_interval and max_interval are the actual interval time value divided by 1.25 ms (such as applying for a 7.5ms connection, the value is 6), and the timeout value is the actual supervision timeout time value divided by 10ms (such as timeout for 1s, the value is 100).

Application example: When the connection is established, apply to update the connection parameters. You can refer to TEST_L2CAP_CONN_PARAM_UPDATE in the feature test.

```
    void app_le_connection_complete_event_handle (u8 *p)
    {
......
bls_l2cap_requestConnParamUpdate (pCon->handle, 24, 24, 0, 400);
 //interval=30ms latency=0 timeout=4s
......
    }
```

Figure 3-40 conn para update request and response Information when Receiving Packets



Among them API:：

```
void bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate(u16 connHandle,int time_ms)
```

In the single connection SDK, it is used to set the slave device to execute API: bls_l2cap_requestConnParamUpdate after waiting for time_ms (unit: milliseconds) to update the connection parameters. If the user only calls bls_l2cap_requestConnParamUpdate after the connection is established, the slave device executes the connection parameter update request 1s after the connection is established. (Note: At present, this function has been incorrectly changed in the SDK, that is, the parameters set by the API are invalid, and the default 1s will always be valid. Need to be fixed in the next version)

In the application, the SDK provides the GAP Event---GAP_EVT_L2CAP_CONN_PARAM_UPDATE to obtain the result of the connection request. It is used to notify the user whether the connection parameter request applied by the slave is rejected or accepted by the master. As shown in the above figure, the master accepts the Connection_Param_Update_Req parameter of the slave. Users can use blc_gap_registerHostEventHandler(app_host_event_callback); to register GAP event callback function interface.

Process GAP_EVT_L2CAP_CONN_PARAM_UPDATE in the callback function, analyze the returned result, and determine whether the master has accepted the connection parameter request of the slave.

Reference slave initialization use case:

```
blc_gap_registerHostEventHandler( app_host_event_callback );
The app_host_event_callback function reference is as follows:
int app_host_event_callback(u32 h, u8 *para, int n)
{
u8 event = h & 0xFF;
      switch(event){
.......
case GAP_EVT_L2CAP_CONN_PARAM_UPDATE:
{
(rf_pkt_l2cap_sig_connParaUpRsp_t*) p= (rf_pkt_l2cap_sig_connParaUpRsp_t*) para;
If( p->result == CONN_PARAM_UPDATE_ACCEPT ){
//the LE master Host has accepted the connection parameters
}
else if ( p->result == CONN_PARAM_UPDATE_REJECT ){
//the LE master Host has rejected the connection parameter
}
}
Break;
......
}
      return 0;
}
```

2. The master responds to the update request

After the slave applies for new connection parameters, the master receives this command and returns the CONNECTION PARAMETER UPDATE RESPONSE command. For details, please refer to "Core_v5.0" (Vol 3/Part A/ 4.20 "CONNECTION PARAMETER UPDATE RESPONSE").

At present, in the 8258 multiple SDK, master does not accept the connection parameter update of the slave, because if the slave parameter update is accepted, this puts higher requirements on the master's timing allocation. Some appropriate connection parameters update will be accepted in the future, rather than rejecting all connection updates.

The following figure shows the command format and result meaning. When the result is 0x0000, the command is accepted, and when the result is 0x0001, the command is rejected.

Whether the actual Android and iOS devices accept the connection parameters applied by the user is related to the practices of BLE masters of various manufacturers. Basically, each one is different. There is no way to provide a unified standard. You can only rely on the user's usual master In the compatibility test, and summarize.

Figure 3-41 conn para update rsp Format in BLE Protocol Stack

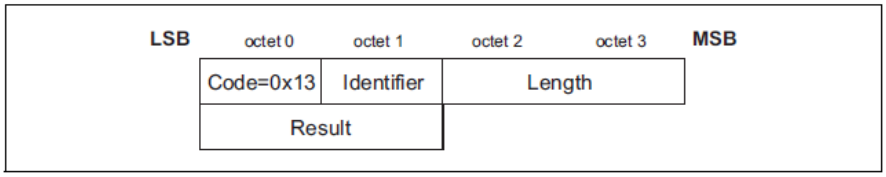


Figure 4.23: Connection Parameters Update Response Packet

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

Regardless of whether the Slave parameter application is accepted or not, the following API is used to reply to the application:

```
blc_l2cap_SendConnParamUpdateResponse(connHandle, req->id, connParaRsp);
```

connHandle specifies the current connection ID, and the following two options indicate acceptance and rejection.

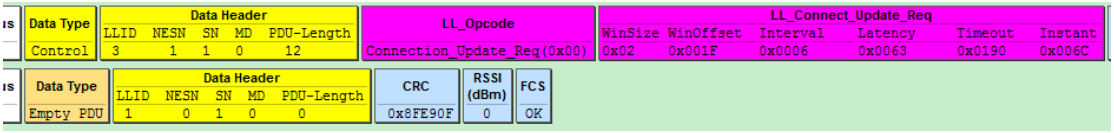
```
typedef enum{
        CONN_PARAM_UPDATE_ACCEPT = 0x0000,
        CONN_PARAM_UPDATE_REJECT = 0x0001,
}conn_para_up_rsp;
```

3. master updates the connection parameters on the Link Layer

Although currently master refuses the connection parameter request of the slave，future versions will consider accepting appropriate connection parameter requests. If the master accepts the update, there will be the following process.

After Slave sends conn para update req, and the master returns to conn para update rsp to accept the application, the master will send the LL_CONNECTION_UPDATE_REQ command of the link layer, as shown in the figure below.

Figure 3-42 ll conn update req information when Receiving Packet



After receiving this update request, the slave writes down the last parameter as the instant value of the master side. When the instant value of the slave side reaches this value, it updates to the new connection parameter and triggers the callback event BLT_EV_FLAG_CONN_PARA_UPDATE.

instant is the connection event count value maintained by the master and slave respectively, and the range is 0x0000~0xffff. In a connection, their values are always equal. When the master sends a conn_req application and connects to the slave, the master starts to switch its state (from the scanning state to the connection state), and clears the instant on the master side to 0. The slave receives conn_req, switches from the advertising state to the connected state, and clears the instant on the slave side to 0. Each connection packet of master and slave is a connection event. The first connection event at both ends after conn_req has an instant value of 1, and the second connection event has an instant value of 2, increasing in sequence.

When the master sends LL_CONNECTION_UPDATE_REQ, the last parameter instant refers to the connection event labeled instant, the master will use the corresponding values of the first few connection parameters in the LL_CONNECTION_UPDATE_REQ package. Because the instant values of slave and master are always equal, when it receives LL_CONNECTION_UPDATE_REQ, it uses the new connection parameters when its instant is equal to the instant connection event declared by the master. In this way, it can be ensured that both ends complete the switching of connection parameters at the same time point.

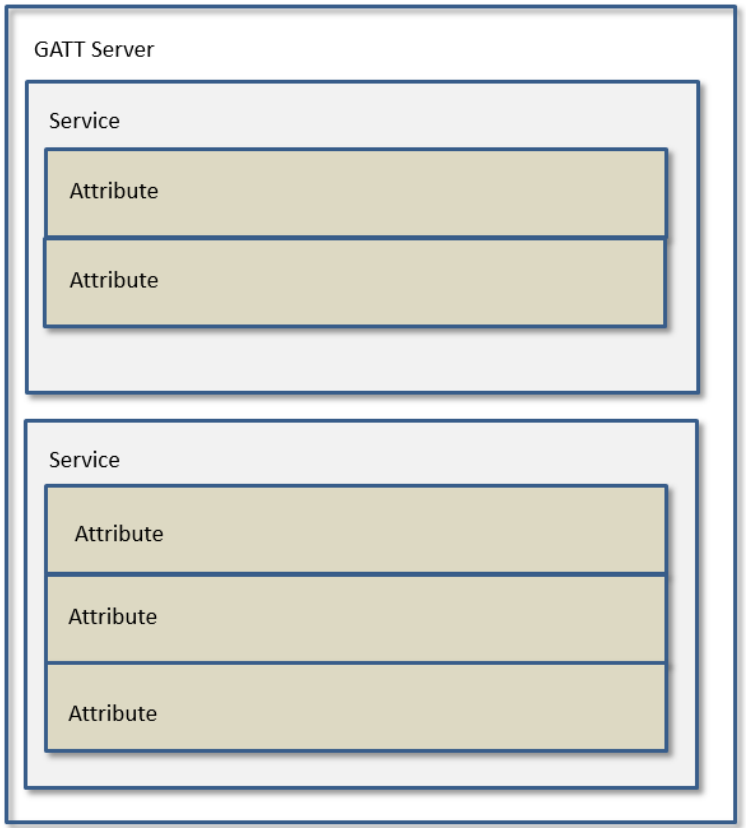
3.4 ATT & GATT

3.4.1 GATT basic unit Attribute

GATT defines two roles: Server and Client. In the BLE SDK, the slave device is a server, and the Android, iOS, or master device is a client. The Server needs to provide multiple services for the Client to access.

The essence of GATT service is composed of multiple Attributes, each Attribute has a certain amount of information, when multiple different types of Attribute are combined together, it can reflect a basic service.

Figure 3-43 Attributes Make GATT Service



The basic content and characteristics of an Attribute include the following:

Attribute Type：UUID

UUID is used to distinguish each attribute type, and its total length is 16 bytes. The UUID length in the BLE standard protocol is defined as 2 bytes. This is because the peer device devices all follow the same set of conversion methods and convert the UUID of 2 bytes to 16 bytes.

When the user directly uses the 2 byte UUID of the Bluetooth standard, the master device knows the device type represented by these UUIDs. Some standard UUIDs have been defined in the SDK and are distributed in the following files: stack/ble/service/hids.h, stack/ble/attr/gatt_uuid.h.

Some of Telink's private profiles (OTA, SPP, MIC, etc.), which are not supported in standard Bluetooth, define these private UUIDs in stack/ble/attr/gatt_uuid.h with a length of 16 bytes.

Attribute Handle

Service has multiple Attributes, which form an Attribute Table. In the Attribute Table, each Attribute has an Attribute Handle value, used to distinguish each different Attribute. After the connection between Slave and Master is established, the Master parses and reads the Slave Attribute Table through the Service Discovery process, and corresponds to each different Attribute according to the value of the Attribute Handle, so that as long as the data communication behind them brings the Attribute Handle, the other party will know which attribute is the data.

Attribute Value

Each Attribute has a corresponding Attribute Value, which is used as data for request, response, notification, and indication. In the SDK, Attribute Value is described by the pointer and the length of the area pointed to by the pointer.

3.4.2 Attribute and ATT Table

In order to realize the GATT service on the slave side, the SDK has designed an Attribute Table, which is composed of multiple basic Attributes. The basic Attribute is defined as:

```
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;
```

Combine the reference Attribute Table given by the SDK to explain the meaning of the above items. Attribute Table code see app_att.c, as shown in the following figure:

Figure 3-44 BLE SDK Attribute Table

```
static const attribute_t my_Attributes[] = {

    {ATT_END_H - 1, 0,0,0,0,0}, // total num of attribute


    // 0001 - 0007  gap
    {7,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID),  (u8*)(&my_gapServiceUUID), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_devNameCharVal),(u8*)(&my_characterUUID), (u8*)(my_devNameCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_devName), (u8*)(&my_devNameUUID), (u8*)(my_devName), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_appearanceCharVal),(u8*)(&my_characterUUID), (u8*)(my_appearanceCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof (my_appearance), (u8*)(&my_appearanceUUID),    (u8*)(&my_appearance), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_periConnParamCharVal),(u8*)(&my_characterUUID), (u8*)(my_periConnParamCharVal),
    {0,ATT_PERMISSIONS_READ,2,sizeof (my_periConnParameters),(u8*)(&my_periConnParamUUID),  (u8*)(&my_periConnParameter


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


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

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

Please note that the definition of Attribute Table is preceded by const:

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

The keyword of const will cause the compiler to store the data of this array in the flash to save ram space. All contents defined in this Attribute Table on Flash are read-only and cannot be rewritten.

attNum

attNum serves two purposes.

The first role of attNum is to represent the number of all valid Attributes in the current Attribute Table, that is, the maximum value of Attribute Handle, which is only used in the 0th invalid Attribute of the Attribute Table array: {57,0,0,0, 0,0}, // ATT_END_H - 1 = 57 in "8258_m4s3".

attNum = 57 means there are 57 Attributes in the current Attribute Table.

In BLE, the value of Attribute Handle starts from 0x0001, and increases by one, and the subscript of the array starts from 0. The above virtual Attribute is added to the Attribute Table, so that the subscript of each Attribute in the data is equal to The value of its Attribute Handle. When the Attribute Table is defined, you can know the current Attribute Handle value of the Attribute by counting the subscripts of the Attribute in the current Attribute Table array.

After counting all the Attributes in the Attribute Table, the last number is the number of effective Attributes in the current Attribute Table attNum, currently 57 in the SDK, if the user adds or deletes Attributes, you need to modify this attNum, you can refer to Vendor/8258_m4s3/app_att.h enumeration ATT_HANDLE.

The second role of attNum is to specify that the current service consists of several attributes.

The UUID of the first Attribute of each service must be GATT_UUID_PRIMARY_SERVICE (0x2800). The attNum on this Attribute specifies that a total of attNum Attributes from the current Attribute are part of the service.

As shown in the figure above, the attNum of the Attribute whose gap service UUID is GATT_UUID_PRIMARY_SERVICE is 7, then the 7 attributes of Attribute Handle 0x0001~ Attribute Handle 0x0007 belong to the description of the gap service.

Similarly, after the attNum of the first Attribute of the HID service in the above figure is set to 27, 27 consecutive Attributes starting from this Attribute belong to the HID service.

Except for the 0th Attribute and the first Attribute of each service, the value of attNum of all other Attributes must be set to 0.

perm

Perm is short for permission.

perm is used to specify the permission of the current Attribute to be accessed by the Client.

There are the following 10 types of permissions, and the permissions of each Attribute must be the following values or their combination.

```
#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: Currently, the SDK does not support authorized reading and authorized writing.

Uuid，uuidLen

As mentioned earlier, there are two types of UUIDs: the BLE standard 2 bytes UUID and Telink's private 16 bytes UUID. uuid and uuidLen can describe UUIDs at the same time.

uuid is a u8 type pointer, uuidLen indicates that the content of consecutive uuidLen bytes from the beginning of the pointer is the current UUID. Attribute Table is stored in flash, and all UUIDs are also stored in flash, so uuid is a pointer to flash.

1. 2 bytes UUID of BLE standard

   For example, the attribute of devNameCharacter of Attribute Handle = 0x0002, the relevant code is as follows:

```
#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 means character in BLE, uuid points to the address of my_devNameCharVal in flash, uuidLen is 2, when peer master reads this Attribute, UUID will be 0x2803.

2. Telink's private 16 bytes UUID:

   Such as OTA's Attribute, the relevant code:

```
#define TELINK_MIC_DATA
{0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x0}
const u8 my_OtaUUID[16]         = TELINK_SPP_DATA_OTA;
static u8 my_OtaData          = 0x00;
{0,3,16,1,(u8*)(&my_OtaUUID), (&my_OtaData), &otaMyWrite, &otaRead},
```

   uuid points to the address of my_OtaData in flash, uuidLen is 16, when the master reads this Attribute, the UUID will be 0x000102030405060708090a0b0c0d2b12.

pAttrValue、attrLen

Each Attribute will have a corresponding Attribute Value. pAttrValue is a u8 type pointer to the address of the RAM/Flash where the Attribute Value is located, attrLen is used to reflect the length of the data on the RAM/Flash. When the master reads the Attribute Value of a Slave Attribute, the BLE SDK starts from the area (RAM/Flash) pointed to by the pAttrValue pointer of the Attribute and fetches attrLen data back to the master.

UUID is read-only, so uuid is a pointer to flash; and Attribute Value may involve a write operation, if there is a write operation must be placed on RAM, so pAttrValue may point to RAM, as well as to Flash.

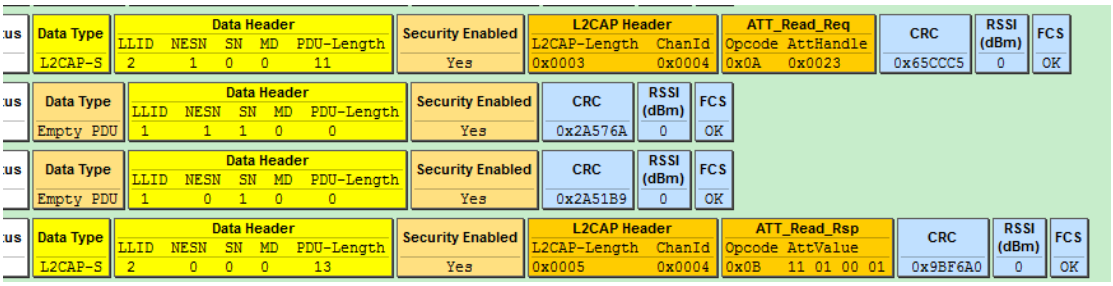
Attribute Handle=0x0027 Attribute of hid Information, related code:

```
    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 practical applications, the hidInformation 4 bytes 0x01 0x00 0x01 0x11 is read-only and does not involve write operations, so you can use the const keyword to store on the Flash when defining. pAttrValue points to the address of hidInformation on flash, attrlen takes the actual length of hidInformation at this time. When the master reads the Attribute, it returns 0x01000111 to the master according to pAttrValue and attrLen.

When the master reads the Attribute, the BLE captures the packet as shown below. The master uses the ATT_Read_Req command, assuming that AttHandle = 0x0023 = 35 to be read corresponds to the hid information in the Attribute Table in the SDK.

Figure 3-45 Master reads hidInformation's BLE packet capture



Attribute Handle=0x002C battery value Attribute, related code:

```
u8          my_batVal[1]      = {99};
{0,1,2,1,(u8*)(&my_batCharUUID),   (u8*)(my_batVal), 0},
```

In practical applications, the value of my_batVal, which reflects the current battery power, will change according to the power sampled by the ADC, and then be transmitted to the master through slave active notify or master active reading, so my_batVal should be placed in memory, at this time pAttrValue points to my_batVal in Address on RAM.

callback function w

The callback function w is a write function. Function prototype:

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

If the user needs to define a callback write function, it must follow the above format. The callback function w is optional. For a specific Attribute, the user can set the callback write function or not. (The null pointer 0 is used when the callback is not set.)

The trigger condition of the callback function w is: when the Attribute Opcode of the Attribute PDU received by the slave is the following three, the slave checks whether the callback function w is set:

1. opcode = 0x12, Write Request
2. opcode = 0x52, Write Command
3. opcode = 0x18, Execute Write Request

After the slave receives the above write command, if the callback function w is not set, the slave will automatically write the value passed by the master to the area pointed to by the pAttValue pointer. The length of the write is l2capLen-3 in the master packet format; if the user sets after the callback function w is received, the slave executes the user's callback function w after receiving the above write command, and no longer writes data to the area pointed to by the pAttrValue pointer at this time. These two write operations are mutually exclusive, and only one can take effect.

The user sets the callback function w to handle the Write Request, Write Command and Execute Write Request commands of the master in the ATT layer. If the callback function w is not set, it is necessary to evaluate whether the area pointed to by pAttrValue can complete the processing of the above commands (such as pAttrValue pointing Flash cannot complete the write operation; or the length of attrLen is not enough, the write operation of the master will cross the boundary, causing other data to be rewritten incorrectly).

Figure 3-46 Write Request in BLE Protocol Stack

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

Figure 3-47 Write Command in BLE Protocol 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

Figure 3-48 Execute Write Request in BLE Protocol 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

The void p pointer of the callback function w points to the specific value of the master write command. The actual p points to a piece of memory, and the value on the memory is shown in 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. The effective length of the written data is l2cap-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 above structure rf_packet_att_data_t is located at stack/ble/ble_format.h.

Callback function r

The callback function r is a reading function. Function prototype:

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

If the user needs to define a callback read function, it must follow the above format. The callback function r is optional. For a specific Attribute, the user can set the callback reading function or not. (The null pointer 0 is used when the callback is not set.)

The trigger condition of the callback function r is: when the Attribute Opcode of the Attribute PDU received by the slave is the following two, the slave checks whether the callback function r is set:

1） opcode = 0x0A, Read Request

2） opcode = 0x0C, Read Blob Request

- Read Response/Read Blob Response: If the user sets a callback read function, execute the function, and decide whether to reply Read Response/Read Blob Response according to the return value of the function:
- If the return value is 1, the slave does not reply Read Response/Read Blob Response to the master.
- If the return value is other values, the slave reads attrLen values from the area pointed to by the pAttrValue pointer and replies to the master with Read Response/Read Blob Response.
- If the user does not set a callback read function, the slave reads attrLen values from the area pointed to by the pAttrValue pointer and replies to the master with Read Response/Read Blob Response.

If the user wants to modify the content of the Read Response/Read Blob Response that will be returned after receiving the Master's Read Request/Read Blob Request, he can register the corresponding callback function r

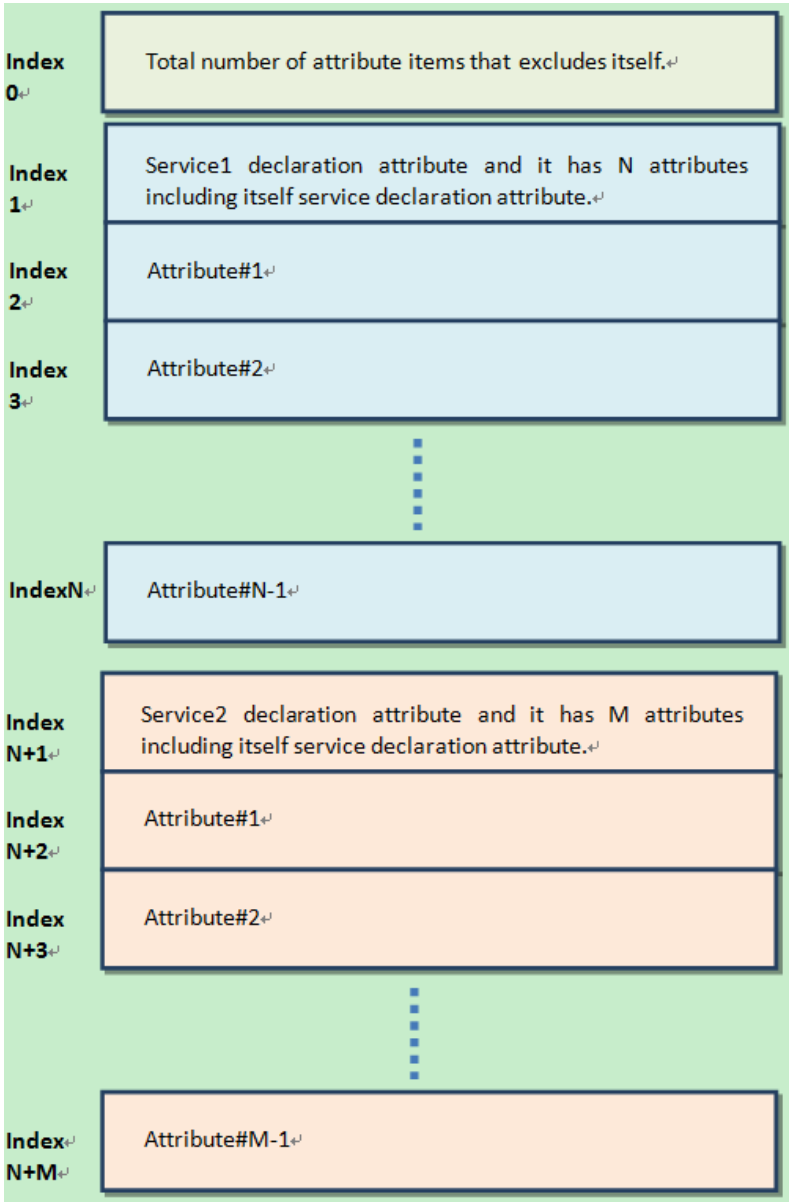
and modify the content of the ram pointed to by the pAttrValue pointer in the callback function. And the value of return can only be 0.

Attribute Table structure

According to the above detailed description of Attribute, use Attribute Table to construct Service structure as shown below. The attnum of the first Attribute is used to indicate the current number of ATT Table Attributes, the remaining Attributes are first grouped by Service, the first Attribute of each group is the declaration of the Service, and use attnum to specify how many Attributes that immediately follow belong to this Service specific description. Actually the first item of each group of Service is a Primary Service.

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

Figure 3-49 Service/Attribute Layout



ATT table Initialization

GATT & ATT initialization only needs to pass the pointer of the Attribute Table of the application layer to the protocol stack, the provided API:

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

p is the pointer of Attribute Table.

3.4.3 GATT Service Security

Before introducing GATT Service Security, users can first learn about SMP related content.

Please refer to the detailed introduction in the "SMP" chapter to understand the basic knowledge of LE pairing method, encryption level and so on.

The following figure is the mapping relationship between GATT service security level service requests given by BLE spec. For details, please refer to "core5.0" (Vol3/Part C/10.3 AUTHENTICATION PROCEDURE).

Figure 3-50 Local Device Responds to a Service Request

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

Table 10.2: Local device responds to a service request

Users can clearly see that the first column is related to whether the currently connected slave device is in an encrypted state, and the second column (local Device's Access Requirement for service) is related to the permission of the feature in the ATT table set by the user (Permission Access) The settings are as shown in the figure below. The third column is divided into 4 sub-columns, and these 4 sub-columns correspond to the four levels of the current LE security mode 1 (specifically, whether the current device pairing status is one of the following 4 types: 1, No authentication and no encryption; 2. Unauthenticated pairing with encryption; 3. Authenticated pairing with encryption; 4. Authenticated LE Secure Connections).

Figure 3-51 ATT Permission Definition

```
/** @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_C

//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
#define ATT_PERMISSIONS_AUTHEN_WRITE        (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)        //!< Write require
#define ATT_PERMISSIONS_AUTHEN_RDWR         (ATT_PERMISSIONS_RDWR  | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTHEN)        //!< Read & Write


#define ATT_PERMISSIONS_SECURE_CONN_READ    (ATT_PERMISSIONS_READ  | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTH
#define ATT_PERMISSIONS_SECURE_CONN_WRITE   (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTH
#define ATT_PERMISSIONS_SECURE_CONN_RDWR    (ATT_PERMISSIONS_RDWR  | ATT_PERMISSIONS_SECURE_CONN | ATT_PERMISSIONS_ENCRYPT | ATT_PERMISSIONS_AUTH
```

The final implementation of GATT service security is related to the parameter configuration during SMP initialization, including the highest security level setting supported, the feature permission setting in the ATT table, etc., and it is also related to the master. For example, the highest level that the SMP set by the slave can support is Authenticated pairing with encryption, but the highest security level that the master has is Unauthenticated pairing with encryption. At this time, if the permission of a write feature in the ATT table is ATT_PERMISSIONS_AUTHEN_WRITE, then when the master writes the feature, we will reply with an error that the encryption level is not enough.

Users can set the feature permissions in the ATT table to achieve the following applications:

For example, the highest security level supported by the slave device is Unauthenticated pairing with encryption, but you do not want to use the method of sending a Security Request to trigger the master to start pairing after connecting, then the customer can configure certain client characteristics with notify attribute (Client Characteristic Configuration, (CCC for short) The permission of the attribute is set to ATT_PERMISSIONS_ENCRYPT_WRITE, then after the master only writes the CCC, the slave will reply that its security level is not enough, which will trigger the master to start the pairing encryption process.

It should be noted that the security level set by the user only represents the highest security level that the device can support. As long as the permission of the feature in the ATT table (ATT Permission) does not exceed the actual maximum level, it can be controlled by GATT service security. For level 4 in LE security mode 1, if the user sets only one level of Authenticated LE Secure Connections, it means that the current setting supports LE Secure Connections only.

3.4.4 Attribute PDU & GATT API

According to the BLE Spec, the Attribute PDU currently supported by the BLE SDK has the following categories:

- Requests： Data request from client to server
- Response：The server sends a data reply after receiving the client's request.

- Commands：The command sent by the client to the server.
- Notification：The data sent by the server to the client.
- Indications：The data sent by the server to the client.
- Confirmations：The client confirms the server indication data.

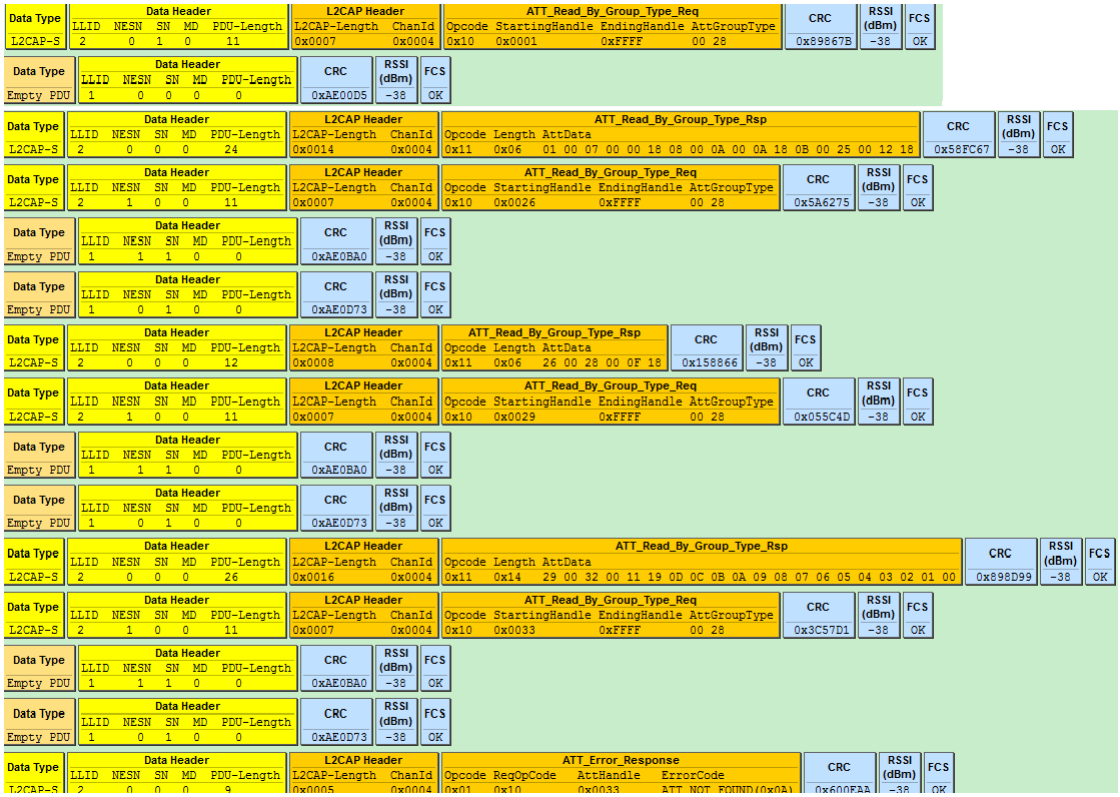
The following is an analysis of all ATT PDUs in the ATT layer in combination with the Attribute structure and Attribute Table structure introduced earlier.

3.4.4.1 Read by Group Type Request、Read by Group Type Response

For details of Read by Group Type Request and Read by Group Type Response, please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.9 and 3.4.4.10).

The master sends a Read by Group Type Request, specifies the start and end attHandle in this command, and specifies attGroupType. After receiving the Request, the slave traverses the current Attribute table, finds the Attribute Group that matches the attGroupType in the specified start and end attHandle, and responds to the Attribute Group information through Read by Group Type Response.

Figure 3-52 Read by Group Type Request/Read by Group Type Response Example



As shown in the figure above, the master queries the Attribute Group information of the primaryServiceUUID of the slave whose UUID is 0x2800:

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

Referring to the current "8258_m4s3" demo code, the following groups in the slave Attribute table meet the requirements:

A. AttHandle Attribute Group from 0x0001 ~ 0x0007, Attribute Value is:

SERVICE_UUID_GENERIC_ACCESS（0x1800）。

B. AttHandle Attribute Group from 0x0008 ~ 0x000B, Attribute Value is：

SERVICE_UUID_GENERIC_ATTRIBUTE（0x1801）。

C. AttHandle Attribute Group from 0x000C to 0x000E, Attribute Value is:

SERVICE_UUID_DEVICE_INFORMATION（0x180A）。

D. AttHandle Attribute Group from 0x000F to 0x0029, Attribute Value is:

SERVICE_UUID_HUMAN_INTERFACE_DEVICE（0x1812）。

E. AttHandle from 0x002A ~ 0x002D Attribute Group, Attribute Value is: SERVICE_UUID_BATTERY（0x180F）。

F. AttHandle from 0x002E ~ 0x0035 Attribute Group, Attribute Value is:

TELINK_SPP_UUID_SERVICE

（0x10,0x19,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00）。

G. attHandle Attribute Group from 0x0036 to 0x0039, Attribute Value is:

TELINK_OTA_UUID_SERVICE

（0x12,0x19,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00）。

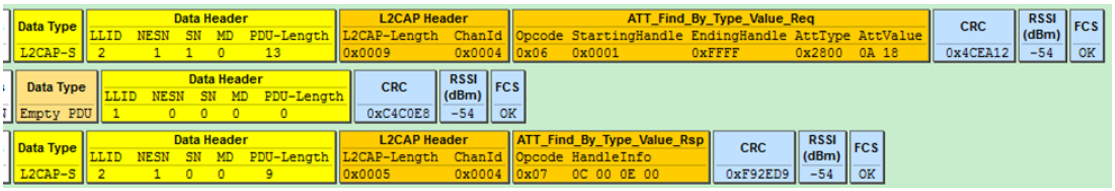
The slave replies the information of the above 7 GROUP's attHandle and attValue to the master through Read by Group Type Response. The last ATT_Error_Response indicates that all the Attribute Groups have been responded. The Response is over. The master will stop sending Read by Group when it sees this packet.

3.4.4.2 Find by Type Value Request、Find by Type Value Response

For details of Find by Type Value Request and Find by Type Value Response, please refer to "Core_v5.0" (Vol 3/Part F/3.4.3.3 and 3.4.3.4).

The master sends a Find by Type Value Request, specifying the start and end attHandle in this command, and specifying AttributeType and Attribute Value. After receiving the Request, the slave traverses the current Attribute table, finds the matching AttributeType and Attribute Value in the specified start and end attHandle, and returns the Attribute through Find by Type Value Response.

Figure 3-53 Find by Type Value Request/Find by Type Value Response

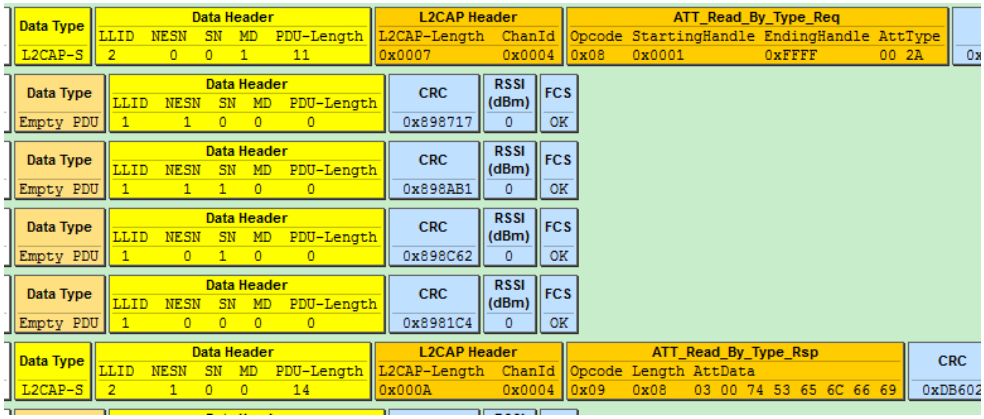


3.4.4.3 Read by Type Request、Read by Type Response

For details of Read by Type Request and Read by Type Response, please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.1 and 3.4.4.2).

The master sends a Read by Type Request, specifies the start and end attHandle in this command, and specifies the AttributeType. After receiving the Request, the slave traverses the current Attribute table, finds the Attribute that meets the AttributeType in the specified start and end attHandle, and responds to the Attribute by Read by Type Response.

Figure 3-54 Read by Type Request/Read by Type Response



As shown in the above figure, the master reads the Attribute with attType 0x2A00, and the Attribute Handle with 0x0003 in the 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},
```

The length in Read by Type response is 8, the first two bytes in attData are the current attHandle 0003, and the following 6 bytes are the corresponding Attribute Value.

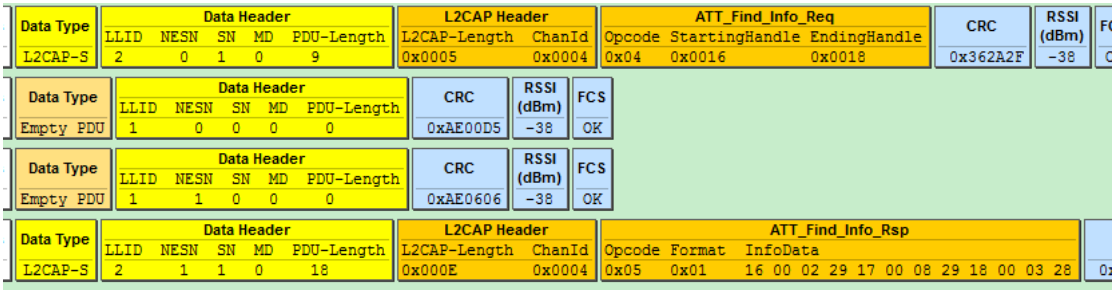
3.4.4.4 Find information Request、Find information Response

For details of Find information request and Find information response, please refer to "Core_v5.0" (Vol 3/Part F/3.4.3.1 and 3.4.3.2).

The master sends a Find information request, specifying the start and end attHandle. After receiving this command, the slave will reply the UUID of all the attHandle corresponding to the attribute at the beginning

and end to the master through Find information response. As shown in the figure below, the master requires information on the three Attributes of attHandle 0x0016 ~ 0x0018, and the slave replies the UUIDs of the three Attributes.

Figure 3-55 Find information request/Find information response

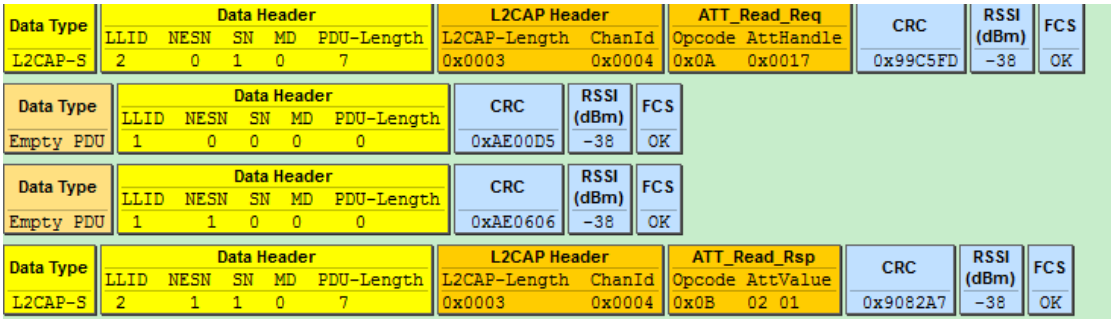


3.4.4.5 Read Request、Read Response

For details of Read Request and Read Response, please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.3 and 3.4.4.4).

The master sends a Read Request and specifies a certain attHandle as 0x0017. After the slave receives it, it responds to the Attribute Value of the specified Attribute by Read Response (if the callback function r is set, the function is executed), as shown in the following figure.

Figure 3-56 Read Request/Read Response



3.4.4.6 Read Blob Request、Read Blob Response

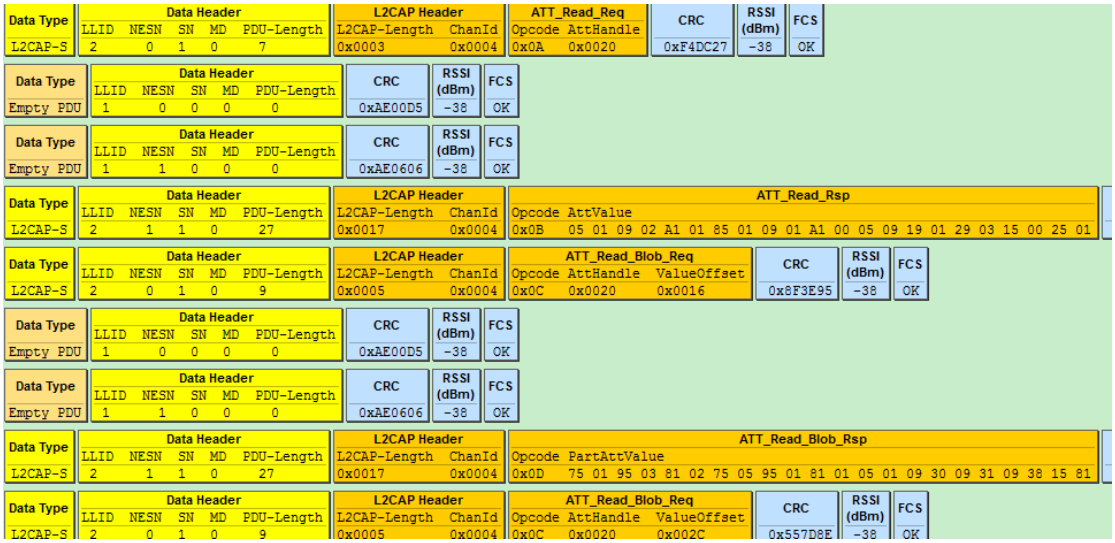
Please refer to "Core_v5.0" (Vol 3/Part F/3.4.4.5 and 3.4.4.6) for Read Blob Request and Read Blob Response.

When the length of the Attribute Value of a slave Attribute exceeds MTU_SIZE (currently the default is 23 in the SDK), the master needs to enable Read Blob Request to read the Attribute Value, so that the Attribute Value can be sent in packets. The master specifies attHandle and ValueOffset in the Read Blob Request. After receiving the command, the slave finds the corresponding Attribute, and responds to the Attribute Value through the Read Blob Response according to the ValueOffset value (if the callback function r is set, the function is executed).

As shown in the following figure, when the master reads the slave's HID report map (the report map is very large, far exceeding 23), it first sends a Read Request, and the slave returns a Read response, returning the

previous part of the report map to the master. After the master uses Read Blob Request, the slave returns the data to the master through Read Blob Response.

Figure 3-57 Read Blob Request/Read Blob Response

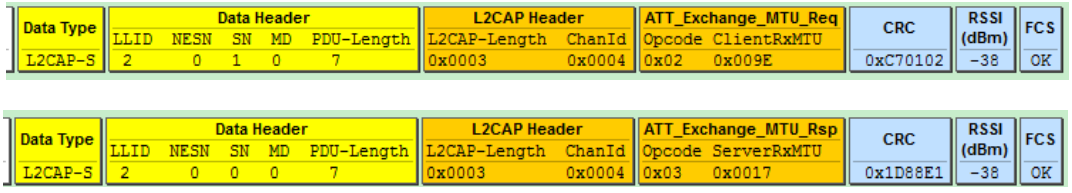


3.4.4.7 Exchange MTU Request、Exchange MTU Response

For details of Exchange MTU Request and Exchange MTU Response, please refer to "Core_v5.0" (Vol 3/Part F/3.4.2.1 and 3.4.2.2).

As shown below, the master and slave know each other's MTU size through Exchange MTU Request and Exchange MTU Response.

Figure 3-58 Exchange MTU Request/Exchange MTU Response



When data of more than one RF packet length appears in the data access process of the GATT layer, when the GATT layer is sub-packetized and packaged, it is necessary to interact with the peer master/slave in advance of the RX MTU size of both parties, which is the process of MTU size exchange . The purpose of MTU size exchange is to send and receive GATT layer long packet data.

A. Users can register the GAP event callback and enable eventMask:

B. Use GAP_EVT_MASK_ATT_EXCHANGE_MTU to obtain EffectiveRxMTU, where:

EffectiveRxMTU=min(ClientRxMTU, ServerRxMTU)。

The "GAP event" section of this document will introduce GAP events in detail.

C. 8258 Master/Slave GATT layer long packet data receiving and processing.

8258 multiple SDK ServerRxMTU and ClientRxMTU default to 23, the maximum ServerRxMTU/ClientRxMTU can support the same as the theoretical value (only limited by ram space). When you need to use sub-packaging and re-assembly in your application, use the following API to modify the Master RX size:

```
ble_sts_t    blm_l2cap_setRxMTUSize(u16 mtu_size);//set master
```

Use the following API to modify the slave RX size:

```
ble_sts_t    bls_l2cap_setRxMTUSize(u16 mtu_size);//set slave
```

Return value list:

Table 3-15 Return Value of Exchange MTU Request

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
GATT_ERR_INVALID_PARAMETER	See definition in SDK	Greater than the defined buffer size, namely: mtu_s_rx_fifo or mtu_m_rx_fifo

Note: The above two API settings are the MTU value when the ATT_Exchange_MTU_req/ATT_Exchange_MTU_rsp command interacts. This value cannot be greater than the buffer size actually defined, ie variables: mtu_m_rx_fifo[] and mtu_s_rx_fifo[ ], these two array variables are defined in app.c.

If there is long packet data in the peer master/slave GATT layer that needs to be sent to the slave/master, the peer master/slave will initiate ATT_Exchange_MTU_req. At this time, the slave/master will reply ATT_Exchange_MTU_rsp, where ServerRxMTU is the above API: blm_l2cap_setRxMTUSize / bls_l2cap_setRx. If the user registers for GAP event and eventMask: GAP_EVT_MASK_ATT_EXCHANGE_MTU is enabled, the user can obtain EffectiveRxMTU and ClientRxMTU on the peer master/slave side in the GAP event callback function.

D. 8258 Master/Slave GATT layer processing of long packet data.

When the 8258 Master/Slave needs to send long packet data at the GATT layer, it needs to obtain the RxMTU of the Master/Slave first, and the final data length cannot be greater than RxMTU.

Master first uses API blm_l2cap_setRxMTUSize to set its own ClientRxMTU, assuming it is set to 158;

```
blm_l2cap_setRxMTUSize(158);
```

Slave first uses the API bls_l2cap_setRxMTUSize to set its own ServerRxMTU, assuming it is set to 158.

```
blc_att_setRxMtuSize（158）;
```

Then call the following API to actively initiate an ATT_Exchange_MTU_req:

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

connHandle is the ID of the Master/Slave connection, and mtu_size is ServerRxMTU / ClientRxMTU.

blc_att_requestMtuSizeExchange(masterHandle, 158); or

blc_att_requestMtuSizeExchange(slaveHandle, 158);

After Peer Slave/Master receives ATT_Exchange_MTU_req, it responds to ATT_Exchange_MTU_rsp. After SDK receives rsp, it calculates EffectiveRxMTU. If the user registers for GAP event and eventMask is enabled: GAP_EVT_MASK_ATT_EXCHANGE_MTU, EffectiveRxMTU and ClientRxMTU will be reported to the user.

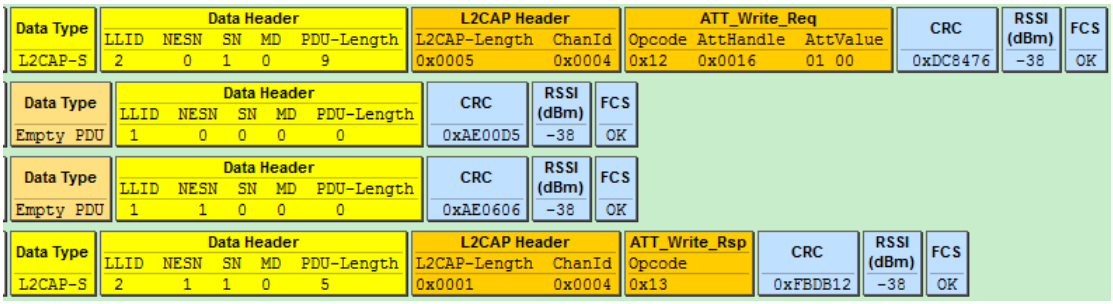
3.4.4.8 Write Request、Write Response

For Write Request and Write Response, please refer to "Core_v5.0" (Vol 3/Part F/3.4.5.1 and 3.4.5.2).

The master sends a Write Request, specifies an attHandle, and attaches relevant data. After receiving, the slave finds the specified Attribute, decides whether the data is processed using the callback function w or directly writes the corresponding Attribute Value according to whether the user has set the callback function w, and replies to Write Response.

The following figure shows that the master writes the Attribute Value to 0x0001 to the Attribute with attHandle of 0x0016. After the slave receives it, it executes the write operation and returns to Write Response.

Figure 3-59 Write Request/Write Response



3.4.4.9 Write Command

For Write Command, please refer to "Core_v5.0" (Vol 3/Part F/3.4.5.3).

The master sends a Write Command, specifies an attHandle, and attaches relevant data. After slave receives, the slave finds the specified Attribute and decides whether the data is processed using the callback function w or directly writes to the corresponding Attribute Value according to whether the user has set the callback function w, and does not reply any information.

3.4.4.10 Queued Writes

Queued Writes include ATT protocols such as Prepare Write Request/Response and Execute Write Request/Response. For details, please refer to "Core_v5.0" (Vol 3/Part F/3.4.6/Queued Writes).

Prepare Write Request and Execute Write Request support the following two functions:

A. Provide the function of writing long attribute values.

B. Allow multiple values to be written in a single atomic operation.

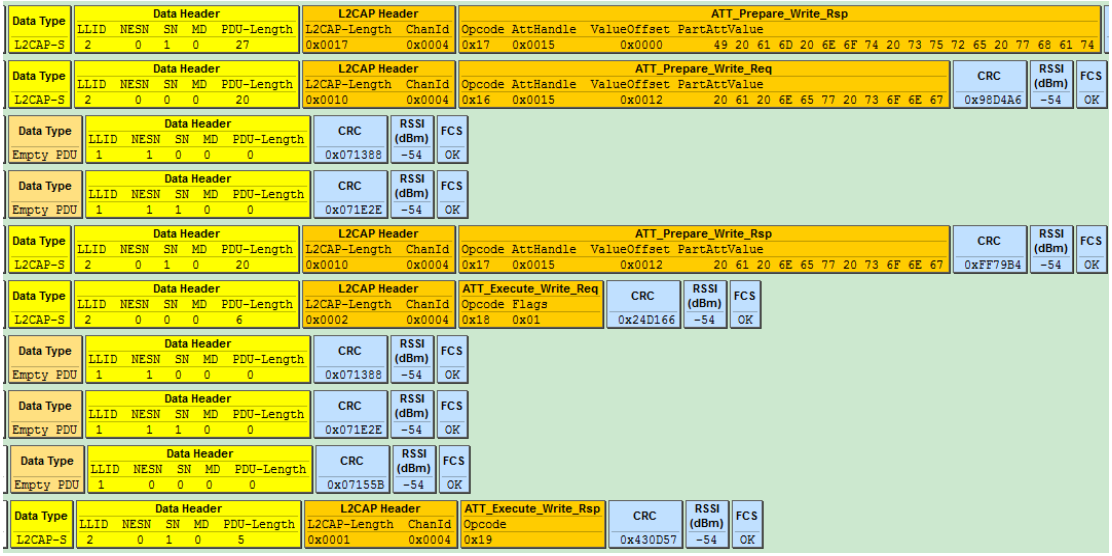
Prepare Write Request contains AttHandle, ValueOffset and PartAttValue, which is similar to Read_Blob_Req/Rsp. This shows that the Client can prepare multiple attribute values in the queue, or it can prepare various parts of a long attribute value. In this way, before actually executing the preparation queue, the Client can determine that all parts of a certain attribute can be written to the Server.

Remarks: The current SDK version only supports A. Long attribute value writing function, the maximum length of long attribute value is less than or equal to 244 bytes.

As shown in the following figure, when the master writes a long string to a certain feature of slave: "I am not sure what a new song" (the number of bytes is far more than 23, using the default MTU), first send Prepare Write Request , Offset 0x0000, write part of the data of "I am not sure what" to the slave, and the slave returns Prepare Write Response to the master. After that, the master sends a Prepare Write Request with an offset of 0x12, and writes the "a new song" part of the data to the slave. The slave returns the Prepare Write Response to the master. After the master writes all the long attribute values, it sends Execute Write Request to the slave. Flags is 1: the write takes effect immediately, and the slave replies to Execute Write Response. The entire Prepare write process ends.

Here we can see that Prepare Write Response also contains AttHandle, ValueOffset and PartAttValue in the request. The purpose of this is for the reliability of data transmission. The Client can compare the field values of Response and Request to ensure that the prepared data is received correctly.

Figure 3-60 Write Long Characteristic Values



3.4.4.11 Handle Value Notification

For details of Handle Value Notification, please refer to "Core_v5.0" (Vol 3/Part F/3.4.7.1)

Figure 3-61 Handle Value Notification in BLE Spec

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

The figure above shows the format of Handle Value Notification in BLE Spec.

The SDK provides an API for Handle Value Notification of an Attribute. The user calls this API to push the data they need to notify to the underlying BLE software fifo. The protocol stack will push the data of the software fifo to the hardware fifo when the nearest packet is sent and received, and finally send it out through RF.

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

connHandle is corresponding to the Connection state, attHandle is corresponding to the Attribute, p is the head pointer of the continuous memory data to be sent, and len specifies the number of bytes of the sent data. The API supports the automatic unpacking function (sub-packet processing according to EffectiveMaxTxOctets, that is, the smaller value of the maximum number of bytes of the link layer RF RX/TX, DLE may modify this value, the default is 27), can be a very long The data is split into multiple BLE RF packets and sent out, so len can support a lot.

When the Link Layer is in the Conn state, in general, directly calling the API can successfully push the data to the underlying software fifo, but there are some special circumstances that will cause the API call to fail. According to the return value ble_sts_t, you can understand the corresponding error reason.

When calling this API, it is recommended that the user check whether the return value is BLE_SUCCESS. If it is not BLE_SUCCESS, you need to wait a while and then push again.

The return values are listed as follows.

Table 3-16 Return Value of Handle Value Notification

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	
GAP_ERR_INVALID_PARAMETER	0xC0	Invalid parameter
SMP_ERR_PAIRING_BUSY	0xA1	In the Pairing stage
GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE	0xB5	len is greater than ATT_MTU-3, the length of the data to be sent exceeds the maximum data length

		supported by the ATT layer ATT_MTU
LL_ERR_CONNECTION_NOT_ESTABLISH	0x80	Link Layer is in None Conn state
LL_ERR_ENCRYPTION_BUSY	0x82	encryption stage，cannot send data
LL_ERR_TX_FIFO_NOT_ENOUGH	0x81	A task with a large amount of data is running, the software Tx fifo is not enough
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	0xB4	Cannot send data during traversal stage

3.4.4.12 Handle Value Indication

For details of Handle Value Indication, please refer to "Core_v5.0" (Vol 3/Part F/3.4.7.2).

Figure 3-62 Handle Value Indication in BLE Spec

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

The figure above shows the format of Handle Value Indication in BLE Spec.

The BLE SDK provides an API for Handle Value Indication of an Attribute. The user calls this API to push the data they need to indicate to the underlying BLE software fifo. The protocol stack will push the data of the software fifo to the hardware fifo when the nearest packet is sent and received, and finally send it out through RF.

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

connHandle is corresponding to the Connection state, attHandle is corresponding to the Attribute, p is the head pointer of the continuous memory data to be sent, and len specifies the number of bytes of the sent data. The API supports automatic unpacking function (sub-packet processing according to EffectiveMaxTxOctets, that is, the smaller value of the maximum number of bytes of the link layer RF RX/TX, DLE may modify this value, the default is 27, the following will introduce its replacement API , See remarks), a very long data can be split into multiple BLE RF packet and sent out, so len supports long data.

The BLE Spec stipulates that the data of each indicate must wait until the client confirms to consider the indicate to succeed. If it is not successful, the next indicate data cannot be sent.

When the Link Layer is in the Conn state, in general, directly calling the API can successfully push the data to the underlying software fifo, but there are some special circumstances that will cause the API call to fail. According to the return value ble_sts_t, you can understand the corresponding error reason.

When calling this API, it is recommended that the user check whether the return value is BLE_SUCCESS. If it is not BLE_SUCCESS, you need to wait a while and then push again.

The l return values are listed as follows.

Table 3-17 Return Value of Handle Value Indication

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	
GAP_ERR_INVALID_PARAMETER	0xC0	Invalid parameter
SMP_ERR_PAIRING_BUSY	0xA1	In the Pairing stage
GATT_ERR_DATA_LENGTH_EXCEED_MTU_SIZE	0xB5	len is greater than ATT_MTU-3, the length of the data to be sent exceeds the maximum data length supported by the ATT layer ATT_MTU
LL_ERR_CONNECTION_NOT_ESTABLISH	0x80	Link Layer is in None Conn state
LL_ERR_ENCRYPTION_BUSY	0x82	encryption stage，cannot send data
LL_ERR_TX_FIFO_NOT_ENOUGH	0x81	A task with a large amount of data is running, the software Tx fifo is not enough
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	0xB4	Cannot send data during traversal stage
GATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	0xB1	The previous indicate data has not been confirmed by the master

3.4.4.13 Handle Value Confirmation

For details of Handle Value Confirmation, please refer to "Core_v5.0" (Vol 3/Part F/3.4.7.3).

Every time the application layer calls blc_gatt_pushHandleValueIndicate, after sending the indicate data to the master, the master will reply with a confirmation, indicating the confirmation of this data, and then the slave can continue to send the next indicate data.

Figure 3-63 Handle Value Confirmation in BLE Spec

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

Table 3.36: Format of Handle Value Confirmation

As can be seen from the above figure, Confirmation does not specify which specific handle is confirmed, and all the indicate data on different handles are unified to reply to a Confirmation.

In order to let the application layer know whether the indicated data sent has been confirmed, users can obtain the Confirm event by registering the GAP event callback and enabling the corresponding event Mask: GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM. The "GAP event" section of this document will introduce the GAP event in detail.

3.4.4.14 Client GATT API

As a master role, the following GATT APIs are provided for simple service discovery or other data access functions.

```
void        att_req_find_info(u8 *dat,
u16 start_attHandle,
u16 end_attHandle);
```

The actual length of dat (byte): 11.

```
void        att_req_find_by_type ( u8 *dat, u16 start_attHandle,
u16 end_attHandle, u8 *uuid,
u8* attr_value, int len);
```

The actual length of dat (byte): 13 + attr_value length.

```
void        att_req_read_by_type (u8 *dat, u16 start_attHandle,
u16 end_attHandle, u8 *uuid,
int uuid_len);
```

The actual length of dat (byte): 11 + uuid length.

```
void        att_req_read (u8 *dat, u16 attHandle);
```

The actual length of dat (byte): 9.

```
void        att_req_read_by_group_type (u8 *dat, u16 start_attHandle,
u16 end_attHandle, u8 *uuid,
int uuid_len);
```

The actual length of dat (byte): 11 + uuid length.

The above API needs to define the memory space *dat in advance, then call the API for data assembly, and finally call blt_llms_pushTxfifo to send the dat to the Controller, and pay attention to whether the return value is TRUE. Taking att_req_find_info as an example, other interfaces can use similar methods.

```
u8   cmd[12];
```

```
att_req_find_info(cmd,   0x0001,   0x0003);
    if( blt_llms_pushTxfifo (BLM_CONN_HANDLE, cmd) ){
        //cmd send OK
    }
```

After sending the corresponding find info req, read req and other cmds to the slave using the method referenced above, you will soon receive the corresponding response information such as the find info rsp and read rsp reply from the slave. Int app_l2cap_handler (u16 conn_handle, u8 * raw_pkt) can be processed according to the following framework:

```
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
        }
    }
```

At the same time, another API format is also provided in the multiple SDK. These APIs do not need to define the memory space *dat in advance. The API internally calls blt_llms_pushTxfifo to send the data to the Controller to send:

```
ble_sts_t     blc_gatt_pushFindInformationRequest(u16 connHandle,
           u16 start_attHandle, u16 end_attHandle)
ble_sts_t    blc_gatt_pushFindByTypeValueRequest(u16 connHandle,
u16 start_attHandle, u16 end_attHandle,
                                          u16 uuid, u8 *attr_value, int len);
ble_sts_t   blc_gatt_pushReadByTypeRequest(u16 connHandle,
u16 start_attHandle, u16 end_attHandle,                                    u8 *uuid, int
uuid_len);
ble_sts_t   blc_gatt_pushReadRequest(u16 connHandle, u16 attHandle);
ble_sts_t    blc_gatt_pushReadByGroupTypeRequest(u16 connHandle,
 u16 start_attHandle,
u16 end_attHandle,
                                    u8 *uuid, int uuid_len);

void       blc_gatt_pushWriteCommand (u16 connHandle, u16 attHandle,
u8 *p, int len);

void       blc_gatt_pushWriteRequest (u16 connHandle, u16 attHandle,
u8 *p, int len);
```

3.5 GAP

3.5.1 GAP Initialization

In multiple SDK, because central and peripheral work at the same time in a device, there is no distinction between central and peripheral devices during initialization.

Initialization function:

```
void blc_gap_init(void);
```

As described above, the data interaction between the application layer and the host does not access control through GAP. The protocol stack provides related interfaces in ATT, SMP and L2CAP, which can directly interact with the application layer. At present, the GAP layer of the SDK mainly processes events on the host layer, and the GAP initialization is mainly to register the entry of the event processing function of the host layer.

3.5.2 GAP Event

GAP events are events generated during the interaction of host protocol layers such as ATT, GATT, SMP, and GAP. As described above, SDK events are currently divided into two categories: Controller events and GAP (host) events, controller events are divided into HCI events and LE HCI events.

The GAP event processing has been added to the Telink BLE SDK, to make the protocol stack event layering clearer, and the protocol stack more convenient for handling user-layer interaction events, especially SMP-related processing, such as Passkey input, notification of pairing results, etc.

If the user needs to receive the GAP event at the App layer, first need to register the callback function of the GAP event, and secondly need to enable the mask of the corresponding event.

GAP event's callback function prototype and registration interface are:

```
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 GAP event tag, which is used in many places in the underlying protocol stack.

Here are a few events that users 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
```

Para and n in the callback function prototype represent the event data and data length. The GAP events listed above will be described in detail below. User can refer to the following usage in demo code and the specific implementation of app_host_event_callback function.

```
    blc_gap_registerHostEventHandler( app_host_event_callback );
```

GAP event enable the mask with the following API.

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

The definition of eventMask is also described above. Other event mask users can find in 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)
```

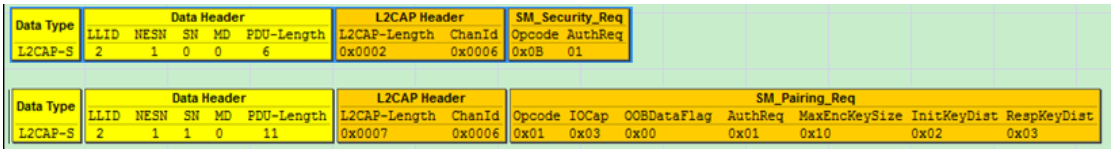
If the user does not set the GAP event mask through this API, the application layer will not be notified when the corresponding GAP event is generated.

Note: below GAP events, the GAP event callback is registered and the corresponding eventMask is enabled.

GAP_EVT_MASK_SMP_PARING_BEAGIN

Event trigger condition: When the slave and the master have just connected to enter the connection state, after the slave sends the SM_Security_Req command, the master sends the SM_Pairing_Req request to start pairing, and when the slave receives this pairing request command, the event is triggered, indicating that the pairing starts.

Figure 3-64 Trigger Event of SMP_PARING_BEAGIN



Data length n: 4.

Return pointer p: point to a piece of memory data, corresponding to the following structure:

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

connHandle indicates the current connection handle.

When secure_conn is 1, it means that the security encryption feature (LE Secure Connections) is used, otherwise LE legacy pairing will be used.

tk_method indicates what TK value method to use for pairing next: for example, JustWorks, PK_Init_Dsply_Resp_Input, PK_Resp_Dsply_Init_Input, Numric_Comparison, etc.

GAP_EVT_SMP_PARING_SUCCESS

Event trigger condition: This event is generated when the entire matching process is completed correctly. This stage is the key distribution phase 3 (Key Distribution, Phase 3) of the LE pairing phase. If a key needs to be distributed, wait for the key distribution of both parties to be completed, then trigger the pairing success event, otherwise directly trigger the pairing success event.

Data length n: 4.

Return pointer p: point to a piece of memory data, corresponding to the following structure:

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

connHandle indicates the current connection handle.

If bonding is 1, the bonding function is enabled, otherwise it is disabled.

bonding_result indicates the result of bonding: if bonding is not enabled, it is 0. If bonding is enabled, it is also necessary to check whether the encryption key is correctly stored in FLASH. If the storage success, the value is 1, otherwise it is 0.

GAP_EVT_SMP_PAIRING_FAIL

Event triggering condition: The pairing process is terminated due to an abnormal reason such as that one of the slave or master does not meet the standard pairing process, or an error occurs during communication.

Data length n: 2.

Return pointer p: point to a piece of memory data, corresponding to the following structure:

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

connHandle indicates the current connection handle.

Reason indicates the reason for pairing failure. Here are a few common pairing failure reason values. For other pairing failure reason values, please refer to the "stack/ble/smp/smp_const.h" file in the SDK directory.

The specific meaning of the pairing failure value can refer to "Core_v5.0" (Vol 3/Part H/3.5.5 "Pairing Failed").

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

GAP_EVT_SMP_CONN_ENCRYPTION_DONE

Event trigger condition: triggered when Link Layer encryption is completed (Link Layer receives a start encryption response from the master).

Data length n: 3.

Return pointer p: point to a piece of memory data, corresponding to the following structure:

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

connHandle indicates the current connection handle.

When re_connect is 1, it means fast reconnect (the LTK encrypted link that was previously distributed will be used). If the value is 0, it means that the current encryption is the first encryption.

GAP_EVT_SMP_TK_DISPALY

Event trigger condition: After the slave receives Pairing_Req sent by the master, according to the pairing parameter of the peer device and the pairing parameter configuration of the local device, we can know what TK (pincode) value method is used for pairing next. If the PK_Resp_Dsply_Init_Input is enabled (that is, the slave side displays the 6-bit pincode code, and the master side is responsible for entering the 6-bit pincode code), it will trigger immediately.

Data length n: 4.

Return pointer p: point to a u32 type variable tk_set, which is the 6-bit pincode code that the slave needs to notify the application layer, and the application layer needs to display the 6-bit code value.

User can also discard the 6-bit pincode code randomly generated by the bottom layer, and manually set a user-specified pincode code such as "123456".

```
		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// use pincode randomly generated by the bottom layer
			u32 pinCode = *(u32*)para;
#endif
		}
		break;
```

The user inputs the 6-digit pincode code seen on the slave to the master device (such as a mobile phone), completes the TK input, and the pairing process can be continued. If the user enters the wrong pincode or clicks cancel, the pairing process fails.

GAP_EVT_SMP_TK_REQUEST_PASSKEY

Event triggering conditions: When the slave device enables the Passkey Entry method and the PK_Init_Dsply_Resp_Input or PK_BOTH_INPUT pairing method is used, this event will be triggered to notify the user that the TK value needs to be entered. After receiving this event, the user needs to input the TK value through the IO input capability (the pairing fails if it has not been input within 30s after timeout). The API for entering the TK value: blc_smp_setTK_by_PasskeyEntry is explained in the "SMP Parameter Configuration" chapter.

Data length n: 0.

Return pointer p: NULL.

GAP_EVT_SMP_TK_REQUEST_OOB

Event triggering condition: When the slave device enables the traditional pairing OOB mode, the event will be triggered to inform the user that the 16-bit TK value needs to be input through the OOB mode. After receiving this event, the user needs to input a 16-bit TK value through the IO input capability (timeout 30s if the input fails to match), the API for entering the TK value: blc_smp_setTK_by_OOB is explained in the "SMP Parameter Configuration" chapter.

Data length n: 0.

Return pointer p: NULL.

GAP_EVT_SMP_NUMERIC_COMPARE

Event trigger condition: After the slave receives Pairing_Req sent by the master, we can know what TK (pincode) value method to use for the next pairing according to the pairing parameter of the peer device and the pairing parameter of the local device. If Numeric_Comparison is enabled, the event will be triggered immediately. (Numeric_Comparison method is numerical comparison. It belongs to smp4.2 security encryption. Both master and slave devices will display a 6-digit pincode and a "YES" and "NO" dialog box. The user needs to check whether the pincode displayed on both ends is consistent, and both ends need to confirm whether to click "YES" to confirm whether the TK check is passed).

Data length n: 4.

Return pointer p: point to a u32 variable pinCode. This value is the 6-bit pincode code that the slave needs to notify the application layer. The application layer needs to display the 6-bit code value and provide confirmation mechanisms for "YES" and "NO".

GAP_EVT_ATT_EXCHANGE_MTU

Event triggering conditions: Whether the master sends an Exchange MTU Request and the slave replies to an Exchange MTU Response, or the slave sends an Exchange MTU Request and the master replies to an Exchange MTU Response, both cases will trigger.

Data length n: 6.

Return pointer p: point to a piece of memory data, corresponding to the following structure:

```
typedef struct {
	u16	connHandle;
	u16	peer_MTU;
	u16	effective_MTU;
} gap_gatt_mtuSizeExchangeEvt_t;
```

connHandle indicates the current connection handle.

peer_MTU indicates the RX MTU value of the peer.

effective_MTU = min(CleintRxMTU, ServerRxMTU), CleintRxMTU represents the RX MTU size value of the client, and ServerRxMTU represents the RX MTU size value of the server. After the Master and the slave exchange each other's MTU size, the minimum value of the two is taken as the maximum MTU size of the interaction.

GAP_EVT_GATT_HANDLE_VALUE_CONFIRM

Event triggering condition: Every time the application layer calls bls_att_pushIndicateData (or calls blc_gatt_pushHandleValueIndicate), after sending the indicate data to the master, the master will reply with a confirm, indicating the confirmation of this data, which is triggered when the slave receives the confirmation.

Data length n: 0.

Return pointer p: NULL.

3.6 GATT Data processing

In this chapter, we will explain how and where the master and slave process the data after receiving it.

3.6.1 Master receiving ATT data processing

The entire processing flow of data received by the master is explained below.

The definition of master fifo has two parts:

1. The controller fifo is defined as follows. This is the link layer, that is, the fifo of a single packet received by the RF.

```
MYFIFO_INIT(blt_rxfifo, 64, 16);
```

2. ATT mtu fifo, defined as follows, this is the fifo that splices (if necessary) the data of the controller fifo into a complete ATT packet

```
u8 mtu_m_rx_fifo[MASTER_MAX_NUM * MTU_M_BUFF_SIZE_MAX];
```

The data storage to ATT mtu fifo is done automatically by the stack, and the user does not need to pay attention.

When the data reaches the ATT mtu fifo, it needs to be processed by the upper layer, and the processing of the master part is left to the user. User needs to configure the callback function for processing when user_init() is initialized. Take the 8258_m4s3 demo as an example, in user_init() we see the function: blc_gatt_register_data_handler(app_gatt_data_handler);

The app_gatt_data_handler callback function is to process all master ATT data, including notify, indicate, etc. As shown below:

```
int app_gatt_data_handler (u16 connHandle, u8 *pkt){
    if( connHandle & BLM_CONN_HANDLE)   //GATT data for Master
    {
        rf_packet_att_t *pAtt = (rf_packet_att_t*)pkt;

        if(pAtt->opcode == ATT_OP_HANDLE_VALUE_NOTI)  //slave handle notify
        {

        }
        else if (pAtt->opcode == ATT_OP_HANDLE_VALUE_IND)
        {

        }
        else if ()
        {

        }
    }
}
```

3.6.2 Slave receiving ATT data processing

The slave ATT data is also stored in the controller fifo through the link layer,

```
MYFIFO_INIT(blt_rxfifo, 64, 16);
```

After stack processing (packing, if it exists), the processed data will also be placed in ATT fifo.

```
u8 mtu_s_rx_fifo[SLAVE_MAX_NUM * MTU_S_BUFF_SIZE_MAX];
```

However, most of the data received by the slave is ATT reading and writing operations, such as: read by type, read blob, etc., the stack will automatically process and reply without user participation. If the user wants to perform an operation when the slave receives the master data, the user can assign the callback function interface to the corresponding att position in the ATT table. Let's take OTA reading and writing as an example. The callback usage is as follows:

Figure 3-65 callback Example

```
////////////////////////////////////// OTA //////////////////////////////////////////////////
// 0036 - 0039
{4,ATT_PERMISSIONS_READ, 2,16,(u8*)(&my_primaryServiceUUID),    (u8*)(&my_OtaServiceUUID), 0},
{0,ATT_PERMISSIONS_READ, 2, sizeof(my_OtaCharVal),(u8*)(&my_characterUUID), (u8*)(my_OtaCharVal), 0},
{0,ATT_PERMISSIONS_RDWR,16,sizeof(my_OtaData),(u8*)(&my_OtaUUID),   (&my_OtaData), &otaMyWrite, &otaRead},
{0,ATT_PERMISSIONS_READ, 2,sizeof (my_OtaName),(u8*)(&userdesc_UUID), (u8*)(my_OtaName), 0},
```

In this way, when the master writes the corresponding handle, the otaMyWrite function is triggered, and the user can perform corresponding processing in this callback function. When the master reads the corresponding handle, the otaRead function is triggered, and the user can perform the corresponding operation according to the actual situation.

User can refer to the demo mode above to implement on other att.

3.7 SMP

The main purpose of Security Manager (SM) in BLE is to provide LE devices with various keys required for encryption to ensure data confidentiality. Encrypted links can ensure that third-party "attackers" are prevented from intercepting, deciphering, or reading the original content of air data. For details of SMP, please refer to "Core_v5.0" (Vol 3/Part H/ Security Manager Specification).

At present, all connections of multiple SDK support the highest security level. The master and slave can be configured with different security levels, and each connection can use different security levels for communication-depending on the security level of the other party.

3.7.1 SMP Security Level

BLE 4.2 Spec has added a new method called LE Secure Connections pairing. The new pairing method has been further enhanced in terms of security, and the pairing method before BLE4.2 is called LE legacy pairing.

As described in the "GATT service Security" section, we can see that the local device's pairing status are as follows:

Figure 3-66 Local Device Pairing Status

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

These four states correspond to the four levels of LE security mode 1, for details, please refer to "Core_v5.0" (Vol 3//Part C/10.2 LE SECURITY MODES):

A. No authentication and no encryption (LE security **mode1 level1**);

B. Unauthenticated pairing with encryption (LE security **mode1 level2**);

C. Authenticated pairing with encryption (LE security **mode1 level3**);

D. Authenticated LE Secure Connections (LE security **mode1 level4**).

Note: The security level set by the local device only indicates the highest security level that the local device can reach. The desired security level depends on two factors: 1. The maximum security level set by the master peer> = the highest security level that can be supported set by slave end; 2. The local end and the opposite end correctly process the entire pairing process according to the SMP parameters set by them (if there is a pairing).

For example, even if the user sets the highest security level that the slave can support to be mode1 level3, but the master connected to the slave is set to not support pairing encryption (the highest only supports mode1 level1), the slave and master will not perform the pairing process after connection, the actual security level used by the slave is mode1 level1.

Users can set the highest security level that SM can support through the following APIs:

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

In multiple SDK, you can configure the security levels supported by master and slave separately, or you can configure the security levels of master and slave together.

If the master and slave use the same security level, you can use the first API blc_smp_setSecurityLevel() to set it. If the master and slave security levels want to be set differently, you can use API blc_smp_setSecurityLevel_master() to set the master separately; use API blc_smp_setSecurityLevel_slave() to set the slave.

Note: In the following chapters of SMP, if there is no special explanation, the function API will provide three forms, namely:

set both master and slave at the same time, such as AAA( ); set the master only, AAA_master( ); set the slave only, AAA_slave( );

The enumeration type le_security_mode_level_t is defined as follows:

```
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;
```

3.7.2 SMP Parameter Configuration

The introduction of SMP parameter configuration in the multiple BLE SDK mainly focuses on the configuration of the four security levels of SMP. The highest level that the SMP function of the master and slave support is LE security mode1 level4.

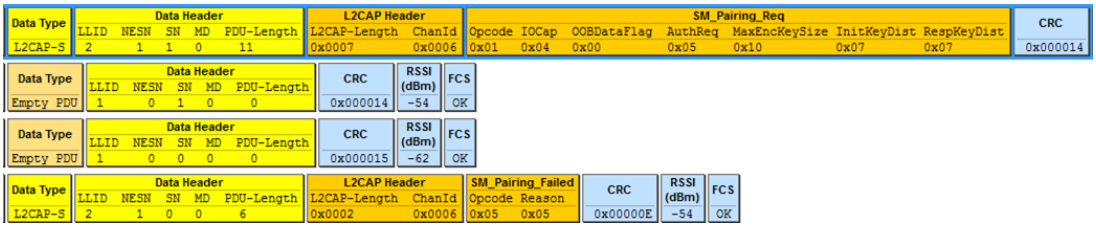
1. LE security mode1 level1

Security level 1 means that the device does not support encrypted pairing. If you need to disable the SMP function, call the following function at the initialization:

```
blc_smp_setSecurityLevel(No_Security);
( blc_smp_setSecurityLevel_slave(No_Security); blc_smp_setSecurityLevel_master(No_Security); )
```

It means that the device will not perform the pairing encryption process on the current connection. Even if the other party requests pairing encryption, the device side will refuse the pairing encryption. Generally used in the process that the current device does not support device encryption and pairing. As shown in the figure below, the master initiates a pairing request, and the slave replies SM_Pairing_Failed.

Figure 3-67 Pairing Disable in Packet Capturing



2. LE security mode1 level2

Security level 2 means that the device supports Unauthenticated_Paring_with_Encryption, such as the Just Works pairing mode in traditional pairing and secure connection pairing.

- As described in the basic concepts of SMP, SM pairing methods include traditional encryption (legacy pairing) and secure connection (secure connection) pairing. The SDK provides the following APIs to set whether to support the new BLE4.2 encryption feature:

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

The definition of the enumeration type paring_methods_t is introduced as follows:

```
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;
```

To use security level configurations other than LE security mode1 level1, the following API must be called to initialize the SMP parameter configuration, including the initial configuration of the binding area FLASH:

```
void blc_smp_smpParamInit(void);
```

Note: There is the only API, which does not distinguish between master and slave, applicable to both.

If only this API is called during the initialization phase, the SDK will use default parameters to configure SMP:

- ✧ The highest security level supported by default：Unauthenticated_Paring_with_Encryption；
- ✧ Default binding mode：Bondable_Mode(Store distributed encrypted key into Flash);
- ✧ The default IO capability is IO_CAPABILITY_NO_INPUT_NO_OUTPUT。

The above default parameters are configured according to the traditional Just Works mode, so the user only calls this API, which is equivalent to configuring LE security mode1 level2. Through A and B, we know that LE security mode1 level2 has two configurations:

A. The device has the initial configuration of Just works under traditional pairing:

```
blc_smp_smpParamInit();
```

B. The device has the initial configuration of Just works under a secure connection:

```
blc_smp_setParingMethods(LE_Secure_Connection); // If the master and slave need to be set separately, refer to the above funcation call to set the master and slave API。
blc_smp_smpParamInit();// Note: SMP parameter configuration must be placed before the API
```

3. LE security mode1 level3

Security level 3 means that the device supports up to Authenticated pairing with encryption, such as Passkey Entry and Out of Band in traditional pairing mode.

This level requires the device to support Authentication, which means that the identity of the pairing parties needs to be ensured by a certain method. BLE provides the following three Authentication methods:

- Methods need manual participation, for example, if the device has buttons or display capabilities, one side displays TK and the other enters the same TK (such as Passkey Entry);
- The two parties of the pair exchange some information through non-BLE RF transmission mode, and perform subsequent pairing operations (such as Out of Band, generally transmitting TK through NFC)
- The device negotiates TK by itself (such as Just Works, both ends use TK:0). It should be noted that the third method belongs to Unauthenticated, so the security level of Just works corresponds to LE security mode1 level2.

Authentication can ensure the legality of the identity of the pairing parties, and providing this method of protection can also be called MITM (Man in the Middle) middleman protection.

Devices with Authentication need to set their MITM flag or OOB flag. The SDK provides the following two APIs for setting the values of MITM and OOB flag:

```
void blc_smp_enableAuthMITM (int MITM_en);
 (blc_smp_enableAuthMITM_master();blc_smp_enableAuthMITM_slave();)
void blc_smp_enableOobAuthentication (int OOB_en);
 (blc_smp_enableOobAuthentication_master();
blc_smp_enableOobAuthentication_slave();)
```

The values of the parameters MITM_en and OOB_en are 0 or 1, 0 corresponds to disabled; 1 corresponds to enabled.

According to the introduction of the Authentication method, SM provides three types of authentication methods. The selection of these three types of methods depends on the IO capabilities of the pairing parties. Our SDK provides the following interfaces for configuring the IO capabilities of the current device:

```
void blc_smp_setIoCapability(io_capability_t ioCapablility);
void blc_smp_setIoCapability _master(io_capability_t ioCapablility);
void blc_smp_setIoCapability _ slave(io_capability_t ioCapablility);
```

The specific definition of enumeration type io_capability_t is introduced as follows:

```
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;
```

MITM and OOB flag usage rules in traditional pairing mode

Figure 3-68 Rules for Using Out-of-ban and MITM Flag for LE Legacy Pairing

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

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

The device determines whether to use the OOB method or the IO capability according to the OOB and MITM flag of the local device and the peer device. The following figure is the SDK to choose different KEY generation methods according to the IO capability mapping relationship (row and column parameter types io_capability_t)

Figure 3-69 Different Key Generating Methods Based on Different IO Referencing

```
// H: Initiator Capabilities
// V: Responder Capabilities
// See the Core_v5.0(Vol 3/Part H/2.3.5.1) for more information.
static const stk_generationMethod_t gen_method_legacy[5 /*Responder*/][5 /*Initiator*/] = {
    { JustWorks,                JustWorks,                PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { JustWorks,                JustWorks,                PK_Resp_Dsply_Init_Input, JustWorks, PK_Resp_Dsply_Init_Input },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_BOTH_INPUT,            JustWorks, PK_Init_Dsply_Resp_Input },
    { JustWorks,                JustWorks,                JustWorks,                JustWorks, JustWorks                },
    { PK_Init_Dsply_Resp_Input, PK_Init_Dsply_Resp_Input, PK_Resp_Dsply_Init_Input, JustWorks, PK_Init_Dsply_Resp_Input },
};

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

This part of the specific mapping relationship can refer to "core5.0" (Vol3/Part H/2.3.5.1 Selecting Key Generation Method), details will not be introduced.

As described in document mentioned above, LE security mode1 level3 has the following initial value configuration methods

A. The device has the initial configuration of OOB under traditional pairing:

```
blc_smp_enableOobAuthentication(1);
blc_smp_smpParamInit();//SMP parameter configuration must be placed before the API
```

Here, because it involves OOB transmission of TK values, the SDK provides related GAP events to users at the application layer. Please refer to the "GAP event" chapter. The API provided for users to set the OOB TK value is as follows:

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

The parameter oobData represents the head pointer of the 16-bit TK value array to be set.

B. The device has the initial configuration of Passkey Entry (PK_Resp_Dsply_Init_Input) under traditional pairing:

```
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_ONLY);
blc_smp_smpParamInit();//SMP parameter configuration must be placed before the API
```

C. The device has the initial configuration of Passkey Entry (PK_Init_Dsply_Resp_Input or PK_BOTH_INPUT) under traditional pairing:

```
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABLITY_KEYBOARD_ONLY);
blc_smp_smpParamInit();//SMP parameter configuration must be placed before the API
```

Here, because it involves the user input TK value, SDK provides related GAP event to the user at the application layer, please refer to the "GAP event" chapter. The TK value API provided for users to set Passkey Entry is as follows:

```
void blc_smp_setTK_by_PasskeyEntry (u32 pinCodeInput); // There is no API set by master and slave separately, the API is applicable to both.
```

The parameter pinCodeInput represents the set pincode value, the range is "0~999999". In the Passkey Entry mode, the master displays TK, and the slave needs to input TK to use.

The key generation method used by the final device is based on the SMP security level supported by the devices at both ends of the pairing connection. If the master only supports the security level LE security mode1 level1, then the slave will not enable the SMP function because the master does not Support pairing encryption.

4. LE security mode1 level4

Security level 4 indicates that the device supports Authenticated LE Secure Connections, such as Numeric Comparison, Passkey Entry, and Out of Band in secure connection pairing mode.

As described above, LE security mode1 level4 has the following initial value configuration methods:

A. The device has the initial configuration of Numeric Comparison under secure connection pairing:

```
blc_smp_setParingMethods(LE_Secure_Connection);
blc_smp_enableAuthMITM(1);
```

```
blc_smp_setIoCapability(IO_CAPABLITY_DISPLAY_YESNO);
```

Because it involves displaying numerical comparison values to users, the SDK provides related GAP events to users at the application layer. Please refer to the "GAP event" chapter. The APIs provided for users to set the numeric comparison result "YES" or "NO" value are as follows:

```
void blc_smp_setNumericComparisonResult(bool YES_or_NO);// There is no API set by master and slave separately, the API is applicable to both.
```

Parameter YES_or_NO: Under the value comparison and matching mode, it is used to confirm to the user whether the values displayed at both ends of the comparison are consistent. When the user confirms that the displayed 6-digit value is consistent with the opposite end, you can enter 1: "YES", otherwise 0: "NO".

B. The device has the initial configuration of Passkey Entry under secure connection pairing:

This part of the user initialization code is basically the same as LE security mode1 level3 configuration methods B and C (traditional pairing Passkey Entry). The only difference is that the pairing method needs to be set to "secure connection pairing" at the beginning of initialization:

```
blc_smp_setParingMethods(LE_Secure_Connection);
.....//Refer to LE security mode1 level3 configuration mode B, C
```

C. The device has the initial configuration of Out of Band under secure connection pairing:

This part of the SDK is not implemented yet, so details will not be introduced here.

1) Here are more APIs related to SMP parameter configuration:

A. The SDK provides the API to enable the binding function:

```
void blc_smp_setBondingMode(bonding_mode_t mode);
```

The definition of the enumeration type bonding_mode_t is as follows

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

For devices with a security level other than mode1 level1, the binding function must be enabled. The SDK is already enabled by default, so users generally do not need to call this API.

B. The SDK provides an API to enable the Key Press function

```
void blc_smp_enableKeypress (int keyPress_en);
```

Indicates whether it is possible to provide some necessary input status information for KeyboardOnly devices during Passkey Entry. Because the SDK does not support this function, the parameter must be set to 0.

C. Whether to enable Debug ellipse encryption key pair in the secure connection mode:

```
void blc_smp_setEcdhDebugMode(ecdh_keys_mode_t mode);
```

The definition of the enumeration type ecdh_keys_mode_t is as follows:

```
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 is only used in the case of secure connection pairing. Since the ellipse encryption algorithm is used in the case of secure connection pairing, it can effectively avoid eavesdropping, which is not so friendly to debugging and development. Users cannot grab BLE air packets through the sniffer tool, and For data analysis and debugging, the BLE spec also provides a set of elliptical encryption private/public key pairs for debugging. As long as this mode is turned on, the BLE sniffer tool can use known keys to decrypt the link.

D. Use the following API to set whether SM is bound, whether to enable the MITM flag, whether to support OOB, whether to support Keypress notification, and supported IO capabilities (the front of the document is a separate configuration API, for user convenience, SDK also provides Unified configuration API):

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

The meaning of each parameter has been introduced earlier, and will not be repeated here.

3.7.3 SMP security request configuration

SMP Security Request (Security Request) can only be sent by slave, so this part is only for slave devices.

Phase 1 of the pairing process has an optional security request package (Security Request), the purpose of this package is to enable the slave to actively trigger the start of the pairing process. The SDK provides the following APIs to flexibly configure whether the slave sends a Security Request to the master after connecting or immediately after re-connecting or pending_ms milliseconds, or not to send a Security Request to achieve different pairing trigger combinations:

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

The definition of the enumeration type secReq_cfg is as follows:

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

The meaning of each parameter is as follows:

SecReq_NOT_SEND：After the connection is established, the slave will not actively send a Security Request;

SecReq_IMM_SEND：After the connection is established, the slave will immediately send a Security Request;

SecReq_PEND_SEND：After the connection is established, the slave waits for pending_ms (in milliseconds) before deciding whether to send a Security Request (1. For the first connection, the slave receives the

master's Pairing_request packet before pending_ms milliseconds and will not send a Security Request; 2. During the reconnection phase, if pending_ms milliseconds before the master has sent LL_ENC_REQ encrypted backlink, then no longer send Security Request).

newConn_cfg：Used to configure a new device, reConn_cfg: used to configure the connected device. Here the SDK determine configuring whether to send a pairing request when connecting back: the paired and bound device, the next time you connect again (that is, connecting back), sometimes the master may not actively initiate LL_ENC_REQ to encrypt the link. If the slave sends a Security Request, it will trigger the master to actively encrypt the link, so the SDK provides reConn_cfg configuration, and the customer can configure it according to actual needs.

Note: The function can only be called before connecting. It is recommended to call it during initialization.

The input parameters of the function blc_smp_configSecurityRequestSending have the following 9 combinations:

Table 3-18 Input Parameter Combination of blc_smp_configSecurityRequestSending

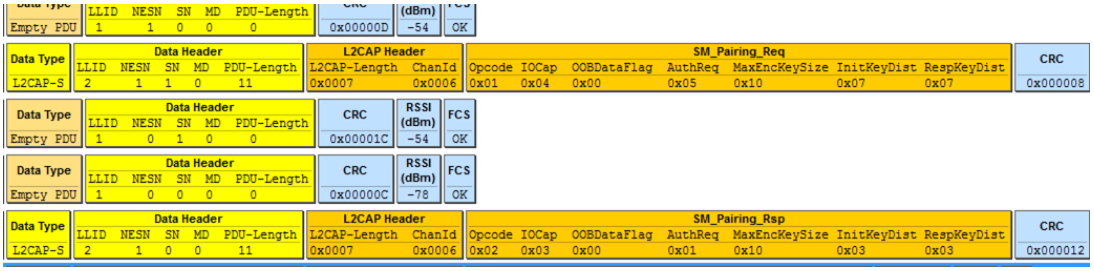
reConn_cfg / newConn_cfg	***SecReq_NOT_SEND***	***SecReq_IMM_SEND***	***SecReq_PEND_SEND***
SecReq_NOT_SEND	SecReq will not be sent for the first connection or connection-back (the parameter pending_ms is invalid)	SecReq will not be sent for the first connection, and SecReq will be sent immediately after reconnecting (parameter pending_ms is invalid)	SecReq will not be sent for the first connection, and SecReq will be sent after pending_ms milliseconds (*see the previous parameter description)for connection-back
SecReq_IMM_SEND	SecReq is sent immediately after the first connection, and SecReq is not sent for connection-back(the parameter pending_ms is invalid)	SecReq is sent immediately after the first connection and connetion-back (parameter pending_ms is invalid)	SecReq is sent immediately after the first connection, and SecReq is sent after pending_ms milliseconds (*see the previous parameter description)for connection-back
SecReq_PEND_SEND	SecReq is sent after the first connection	SecReq is sent after the first connection	SecReq is sent after pending_ms

	pending_ms milliseconds (*see the previous parameter description), SecReq is not sent for connection-back	pending_ms milliseconds (*see the previous parameter description), and SecReq is sent immediately after connection-back	milliseconds (*see the previous parameter description) for both the first connection and connection-back

We pick two of them to give a detailed description. The other combinations are similar, and will not be detailed here:

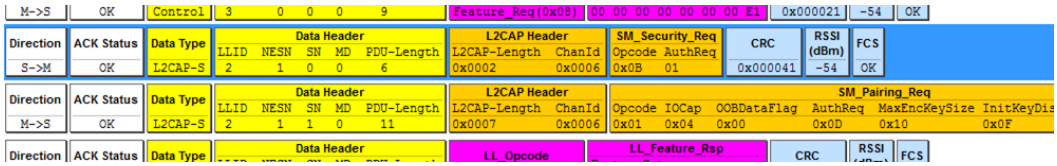
- newConn_cfg: SecReq_NOT_SEND,
- reConn_cfg: SecReq_NOT_SEND,
- pending_ms: This parameter is invalid at this time.
- newConn_cfg: SecReq_NOT_SEND means that the new device slave will not actively initiate Security Request, only responds to the other party's pairing request when the other party initiates a pairing request. If the other party does not sendpairing requests, encryption pairing will not be performed. As shown below, the master sends a pairing request packet SM_Pairing_Req, the slave will respond, but will not actively trigger the master to initiate a pairing request.

Figure 3-70 Paring Peer Trigger in Packet Capturing



- reConn_cfg: SecReq_NOT_SEND indicates that the device has been paired, and the slave device will not send a Security Request when connection-back.
- newConn_cfg：SecReq_IMM_SEND
- reConn_cfg: SecReq_NOT_SEND,
- pending_ms： This parameter is invalid at this time.
- newConn_cfg： SecReq_IMM_SEND means that the new device slave will actively send a Security Request to the master once connected to trigger the master to start the pairing process. As shown in the figure below, the slave actively sends SM_Security_Req to trigger the master to send a pairing request:

Figure 3-71 Paring ConnTrigger in Packet Capturing



- reConn_cfg: SecReq_NOT_SEND means that the slave will not send a Security Request when connection-back.

In addition, the SDK also provides an API to send the Security Request package separately for special applications. The application layer can call this API at any time to send the Security Request package:

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

It should be noted here that if users use blc_smp_configSecurityRequestSending to control the security pairing request packet, they should not use the blc_smp_sendSecurityRequest function.

3.7.4 SMP binding information description

For multiple SDK, it can save 8 slave pairing information as a master, and 4 master information as a slave. These 12 devices can be connected back successfully. The following interface is used to set the maximum number of master devices and the maximum number of slave devices currently stored. If the user does not set it, the default value is to save 8 slave messages and 4 master messages.

```
void blc_smp_setBondingDeviceMaxNumber ( int peer_slave_max,
int peer_master_max);
```

peer_slave_max indicates that the peer is slave, as the master, the maximum number of slave devices it can store.

peer_master_max indicates that the peer is master, as slaves, the maximum number of master devices it can store.

If blc_smp_setBondingDeviceMaxNumber(8, 4) is set, after pairing 8 slave devices, once the 9th slave device is paired, the code will automatically delete the pairing information of the oldest (1st) slave device, and then store the pairing information of the 9th slave device; when pairing After 4 master devices, once the 5th master device is paired, the code will automatically delete the oldest (1st) master device pairing information, and then store the 5th master device pairing information.

1） Binding information storage order

A concept related to BondingDeviceNumber is called index. If the current BondingDeviceNumber is 1, then there is only one bonding device, and its index is 0; if BondingDeviceNumber is 2, the indexes of the two devices are 0 and 1, respectively.

The multiple SDK provides an index update sequence method: the sequence is based on the device pairing time. This is explained below.

If it is a slave and BondingDeviceNumber is 2, the indexes of the two devices are 0 and 1, respectively. The index sequence is based on the pairing time sequence: suppose the slave is successfully paired with masterA first, and then successfully paired with masterB. At this time, masterA is index 0 and masterB is index1 on

the flash storage of the slave, and then slave and masterA are successfully connected back once. At this time, the index 0 device is still masterA, and the index1 device is masterB.

If BondingDeviceNumber is 4, the indexes of the four devices are 0, 1, 2, 3, 0 is the oldest paired device, and 3 is the latest paired device. As described above, if slaves continuously pair masterA, B, C, and D, then masterD is the index3 device, no matter what order slave and master A, B, C, and D are connected back during the period, index 0, 1, 2, 3 are still Corresponding to masterA, B, C, D respectively.

It should be noted that the device pairing exceeds 4: if masterA, B, C, D are paired consecutively, and then master E is paired, the slave will delete the oldest masterA; if after pairing masterA, B, C, D, return to masterA first Even once, the sequence is still A, B, C, D. If you pair masterE again, the slave will delete the pairing information of masterA.

2） Binding information format and related API description

Master device binding information is stored on flash, and its format is:

Figure 3-72 Bonding Information Format

```
typedef struct {
    //0x00
    u8      flag;
    u8      role_dev_idx;  //[7]:1 for master, 0 for slave;   [2:0] slave device index

    u8      peer_addr_type;  //address used in link layer connection
    u8      peer_addr[6];

    u8      peer_id_adrType; //peer identity address information in key distribution, used to identify
    u8      peer_id_addr[6];

    //0x10
    u8      local_peer_ltk[16];   //slave: local_ltk; master: peer_ltk

    //0x20
    u8      encryt_key_size;
    u8      local_id_adrType;
    u8      local_id_addr[6];

    u8      random[8];  //8


    //0x30
    u8      peer_irk[16];

    //0x40
    u8      local_irk[16];        // local_csrk can be generated based on this key, to save flash area (delete this

    //0x50
    u16     ediv;       //2
    u8      rsvd[14];   //14  peer_csrk info address if needed(delete this note at last, customers can not see it)
}smp_param_save_t;
```

The binding information is 96 bytes in total.

peer_addr_type and peer_addr are the address of the peer device on the link layer, which can be used when the device sends direct adv.

peer_id_adrType/peer_id_addr and peer_irk are the identity address and irk declared by the peer device address in the key distribution phase.

Multi-connection devices can only resolve the resolvable private address (RPA) of the peer device, but the address type of multi-master and multi-slave itself does not support RPA, because the current SDK configurable address type only supports static random address and public address.

Only when the peer_addr_type and peer_addr are RPA and users need to use address filtering, they need to add relevant information to the resolving list so that the local device can resolve the type of peer device (refer to the usage of TEST_WHITELIST in 8258_feature_test).

Other parameters user do not need to pay attention.

The following API uses index to get device information from flash.

```
u32 blt_smp_loadBondingInfoFromFlashByIndex(u8 isMaster, u8 slaveDevIdx,
                                            u8 index, smp_param_save_t* smp_param_load);
```

- param--- isMaster, 0 means slave, non-zero means master;
- param---slaveDevIdx, this parameter is currently set to 0.
- param--- index, it means to read the master or slave information of the pairing order index.
- param---smp_param_load, used to store the read data.

```
u32 blt_smp_loadBondingInfoByAddr(u8 isMaster, u8 slaveDevIdx,
u8 addr_type, u8* addr, smp_param_save_t* smp_param_load);
```

- param--- isMaster, 0 means slave, non-zero means master;
- param---slaveDevIdx, this parameter is currently set to 0.
- param--- index, it means to read the master or slave information of the pairing order index.
- param---smp_param_load, used to store the read data.

The following API is used by the slave device to clear all pairing binding information stored on the local flash:

```
void            blc_smp_eraseAllBondingInfo(void);
```

It should be noted that the user needs to ensure that the device is not connected before calling the API.

The following API can be used for the location where the slave device configuration binding information is stored in FLASH:

```
void blc_smp_configParingSecurityInfoStorageAddressAndSize (int address,
int size_byte);
```

The parameter addr can be modified according to actual needs. Before configuration, the user can refer to the "SDK FLASH Space Allocation" chapter in the document to determine the binding information to be placed in the appropriate area in FLASH.

3.8 Custom Pair

In multi-connection devices, if the master device disables SMP, the SDK cannot automatically complete the pairing and unpairing operations, and you need to add pairing management at the application layer. Based on this, Telink has customized a set of matching and unpairing solutions.

If the user needs to use custom pairing management, first initialize this function, with the following API:

```
        blc_smp_setSecurityLevel_master(No_Security);// disable SMP function
        user_master_host_pairing_management_init();//Customize
```

Flash storage method design

The default flash data sector used is 0x7C000-0x7CFFF, and the macro can be modified in app_config.h:

```
#define FLASH_ADR_CUSTOM_PAIRING                0x7C000
#define FLASH_CUSTOM_PAIRING_MAX_SIZE           4096
```

Every 8 bytes of flash 0x7C000 is divided into an area, called 8 bytes area. Each area can store a slave's mac address, where the first byte is the flag bit, the second byte is the address type, and the next six are 6 bytes of mac address.

```
typedef struct {
        u8 bond_mark;
        u8 adr_type;
        u8 address[6];
} macAddr_t;
```

In the flash storage process, the method of pushing 8 bytes area back in sequence is used. The first valid slave mac is stored at 0x7C000-0x7C007. The first byte flag bit of 0x7C000 is written as 0x5A, indicating that the current address is valid; the second valid mac address is stored at 0x7C008-0x7C00f, 0x7C008 is marked with 0x5A; the third valid mac address is stored at 0x7C010-0x7C017, and 0x7C010 is marked with 0x5A.

If you want to unpair a slave device, the multi-connection devices need to erase the MAC address of the device. You only need to write the flag bit of the 8 bytes area that stored the MAC address as 0x00; if you want to erase the first devices in the above three device, write 0x7C000 as 0x00.

The reason for using the above 8bytes extension method is that the program cannot call the flash_erase_sector function to erase the flash during operation, because it takes 20-200ms to erase an sector 4K flash in this operation. This time will cause BLE timing errors.

Use the 0x5A and 0x00 flag bits to indicate the paired storage and unpaired erasing of all slave MACs, as the 8 bytes area increasing, it may occupy the entire sector 4K flash and cause an error. Special treatment is added during initialization: read 8 bytes area information from 0x7C000, read all valid MAC addresses to the slave MAC table in RAM. In this process, check if there are too many 8 bytes area. If there are too many, erase the entire sector, and then write the slave MAC table maintained in RAM back to the 8 bytes area starting at 0x7C000.

Slave MAC table

Figure 3-73 Slave Mac Table

```
41
42 /* define pair slave max num,
43    if exceed this max num, two methods to process new slave pairing
44    method 1: overwrite the oldest one(telink demo use this method)
45    method 2: not allow paring unness unfair happend  */
46 #define USER_PAIR_SLAVE_MAX_NUM       4  //telink demo use max 4, you can change
47
48
49 typedef struct {
50     u8 bond_mark;
51     u8 adr_type;
52     u8 address[6];
53 } macAddr_t;
54
55
56 typedef struct {
57     u32 bond_flash_idx[USER_PAIR_SLAVE_MAX_NUM];  //mark paired slave mac addres
58     macAddr_t bond_device[USER_PAIR_SLAVE_MAX_NUM];  //macAddr_t alreay defined
59     u8 curNum;
60 } user_salveMac_t;
61
```

```
user_salveMac_t user_tbl_slaveMac;
```

Use the slave MAC table in RAM to maintain all paired devices with the above structure. Change the macro USER_PAIR_SLAVE_MAX_NUM to define the maximum number of pairs you want. The telink multi-connectionSDK defaults to 4, which refers to maintaining the pairing of 4 devices. , user can modify this value.

The curNum in user_tbl_slaveMac indicates that there are several valid slave devices recorded on the flash. The bond_flash_idx array records the offset of the effective address of the 8 bytes area on the flash relative to 0x7C000 (when unpairing this device, you can use this offset shift to find the flag bit of 8 bytes area, write it as 0x00), bond_device array records MAC address.

Related API

Based on the above FLASH storage design and the design of the slave MAC table in RAM, the following APIs can be called respectively.

- user_master_host_pairing_management_init

```
void user_master_host_pairing_management_init(void);
```

User-defined pairing management initialization function, which needs to be called when the custom mode is enabled.

- user_tbl_slave_mac_add

```
int user_tbl_slave_mac_add(u8 adr_type, u8 *adr);
```

Add a slave mac, return 1 means success, 0 means failure. This function needs to be called when a new device is paired.

The function first determines whether the device in the current flash and slave MAC table has reached the maximum value. If the maximum value is not reached, add it to the slave MAC table unconditionally and store it in an 8 bytes area of FLASH.

If it has reached the maximum value, the processing strategy is involved: whether to allow pairing or directly cover the oldest, the Telink demo method is to directly cover the oldest, first use user_tbl_slave_mac_delete_by_index(0) to delete the current device, and then add a new one to the slave mac table. User can modify the implementation of this function according to his own strategy.

- user_tbl_slave_mac_search

```
int user_tbl_slave_mac_search(u8 adr_type, u8 * adr)
```

According to the device address of the adv report, search whether the device is already in the slave MAC table, that is, determine whether the device that is currently sending advertising packets has been paired with the master before, and if the paired device can be directly connected.

- user_tbl_slave_mac_delete_by_adr

```
int user_tbl_slave_mac_delete_by_adr(u8 adr_type, u8 *adr)
```

Delete a paired device by specifying the address.

- user_tbl_slave_mac_delete_by_index

```
void user_tbl_slave_mac_delete_by_index(int index)
```

Delete the paired device by specifying the index. The Index value reflects the sequence of device pairing. If the maximum pairing number is 1, the index of the matched device is always 0; if the maximum pairing number is 2, the index of the first matched device is 0, and the index of the second matched device is 1.

- user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

Delete all paired devices.

Connection and pairing

When the master receives the advertising packet reported by the Controller, it will connect to the slave in the following two situations:

- Call the function user_tbl_slave_mac_search to check whether the current device has been paired with the master and has not been unpaired. If it has already been paired, it can be automatically connected.

```
master_auto_connect = user_tbl_slave_mac_search(pa->adr_type, pa->mac);
if(master_auto_connect) { create connection }
```

- If the current advertising device is not in the slave MAC table and can not be automatically connected, check whether the manual pairing conditions are met. Two manual pairing schemes are set by default in the SDK. Under the premise that the current advertising device is close enough, one is that the pairing key on the multi-connection device is pressed; the other is that the current advertising data is the paired advertising package data defined by Telink.
- Code:

```
//manual paring methods 2: special paring ADV data
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 }
```

- If the connection is triggered by manual pairing, after the connection is successfully established, that is, when HCI LE CONECTION ESTABLISHED EVENT reports, add the current device to the slave MAC table:

```
//manual paring, device match, add this device to slave mac table
if(blm_manPair.manual_pair && blm_manPair.mac_type == pCon->peer_adr_type && !memcmp(blm_manPair.mac, pCon->mac, 6)){
    blm_manPair.manual_pair = 0;

    user_tbl_slave_mac_add(pCon->peer_adr_type, pCon->mac);
}
```

- Unpairing

When the unpairing condition takes effect, the multi-connection device first calls blc_llms_disconnect to disconnect, and then calls the user_tbl_salve_mac_delete_by_adr function to delete the device.

3.9 Device Manage

BLE slave will have GATT service table, we can easily know the handle value of each "attribute" during development, but BLE master is not so easy, it needs to be obtained and maintained through SDP process. For the convenience of users, telink multi-connection SDK provides users with a complete set of peer slave service management solutions. The main idea of this solution is to bind the "attribute handle" and "connection handle" of the peer slave together and store them in FLASH and RAM. It is stored in FLASH to ensure that the SDP process will not be performed after reconnecting. It is stored in RAM to allow the software to run efficiently.

The following describes the peer slave service management solution in the telink multi-connection SDK. The solution is provided in the form of source code. Users can refer to the app_device.c/app_device.h file in the SDK. Users can also refer to the above solution to implement their own peer slave service management solution.

The Telink multi-connection SDK uses the following data structure to manage "attribute handle" and "connection handle".

```
typedef struct{
    u16  conn_handle;                    //connection handle
    u8       conn_state;
u8       char_handle_valid;
u8       rsvd[4];
u8       peer_adrType;
    u8       peer_addr[6];
    u8       peer_RPA;
    u16      char_handle[CHAR_HANDLE_MAX];//peer slave attribute handle
}dev_char_info_t;
```

In the SDK, use the array "conn_dev_list[]" to record and maintain the "attribute handle" of the peer slave device, as shown in Figure 3-1. Note: The array conn_dev_list[] records the attribute handle value of the multi-connectiondevice when work as master and slave. Although it is not necessary to record the attribute handle value when act as slave, but for ease of use and subsequent expansion, we still reserve space for the slave, but it has not been used yet and is only used when work as master. The SDK has also made detailed notes, users can refer to the relevant notes of app_device.c.

Figure 3-74 conn_dev_list array definition

```
/*
* Used for store information of connected devices.
*
* 0 ~ (MASTER_MAX_NUM - 1) is for master,  MASTER_MAX_NUM ~ (MASTER_MAX_NUM + SLAVE_MAX_NUM - 1) s for slave
*
* e.g.    MASTER_MAX_NUM   SLAVE_MAX_NUM          master                  slave
*               1               1            conn_dev_list[0]        conn_dev_list[1]
*               2               2            conn_dev_list[0..1]     conn_dev_list[2..3]
*               3               2            conn_dev_list[0..2]     conn_dev_list[3..4]
*               4               3            conn_dev_list[0..3]     conn_dev_list[4..6]
*/
dev_char_info_t conn_dev_list[DEVICE_CHAR_INFO_MAX_NUM];
```

When the multi-master and multi-slave device establishes a connection with the peer slave, in the connection completed event, by calling the dev_char_info_search_peer_att_handle_by_peer_mac() function, it is determined whether the slave's services attribute handle information exists on the FLASH (as shown in Figure 3-2). If it exists, no SDP process is required, and the "attribute handle" information is copied from FLASH to RAM for subsequent data interaction use; if it does not exist, the SDP process needs to be performed subsequently.

Figure 3-75 connection completed event handle

```
#if (BLE_MASTER_SIMPLE_SDP_ENABLE)
    u8  temp_buff[sizeof(dev_att_t)];
    dev_att_t *pdev_att = (dev_att_t *)temp_buff;

    if( dev_char_info_search_peer_att_handle_by_peer_mac(pCon->peer_adr_type, pCon->mac, pdev_att) ){
        #if(UI_AUDIO_ENABLE)
            cur_master_conn_device.char_handle[0] = pdev_att->char_handle[0];       //MIC
        #endif
        //cur_master_conn_device.char_handle[1] =                                   //Speaker
        cur_master_conn_device.char_handle[2] = pdev_att->char_handle[2];          //OTA
        cur_master_conn_device.char_handle[3] = pdev_att->char_handle[3];          //consume report
        cur_master_conn_device.char_handle[4] = pdev_att->char_handle[4];          //normal key repor
        //cur_master_conn_device.char_handle[6] =                                   //BLE Module, SPP
        //cur_master_conn_device.char_handle[7] =                                   //BLE Module, SPP

        dev_char_info_add_peer_att_handle(&cur_master_conn_device);  //add the peer device att_handle
    }
    else{
        master_sdp_pending = pCon->handle;  // mark this connection need SDP

        #if (BLE_MASTER_SMP_ENABLE)
            //service discovery initiated after SMP done, trigger it in "GAP_EVT_MASK_SMP_SECURITY_PR
        #else
            //No SMP, service discovery can initiated now
            app_register_service(&app_service_discovery);
        #endif
    }

#endif
```

The SDP process is implemented using the function app_service_discovery (), as shown in Figure 3-3. After the SDP is successful, the functions dev_char_info_add_peer_att_handle() and dev_char_info_store_peer_att_handle() will be called to store the "attribute handle" of the peer slave service in RAM and FLASH for subsequent use.

Figure 3-76 service discovery

```
void app_service_discovery (void)
{
    att_db_uuid16_t     db16[ATT_DB_UUID16_NUM];
    att_db_uuid128_t    db128[ATT_DB_UUID128_NUM];
    memset (db16, 0, ATT_DB_UUID16_NUM * sizeof (att_db_uuid16_t));
    memset (db128, 0, ATT_DB_UUID128_NUM * sizeof (att_db_uuid128_t));

    if ( master_sdp_pending && host_att_discoveryService (master_sdp_pending, db16, ATT_DB_UUID16_NUM, db128, ATT_DB_UUID
    {
        #if(UI_AUDIO_ENABLE)
            cur_master_conn_device.char_handle[0] = blm_att_findHandleOfUuid128 (db128, my_MicUUID);        //MIC
        #endif
        //cur_master_conn_device.char_handle[1] = blm_att_findHandleOfUuid128 (db128, my_SpeakerUUID);      //Speaker
        cur_master_conn_device.char_handle[2] = blm_att_findHandleOfUuid128 (db128, my_OtaUUID);           //OTA

        cur_master_conn_device.char_handle[3] = blm_att_findHandleOfUuid16 (db16, CHARACTERISTIC_UUID_HID_REPORT,
                    HID_REPORT_ID_CONSUME_CONTROL_INPUT | (HID_REPORT_TYPE_INPUT<<8));      //consume report(media key re

        cur_master_conn_device.char_handle[4] = blm_att_findHandleOfUuid16 (db16, CHARACTERISTIC_UUID_HID_REPORT,
                    HID_REPORT_ID_KEYBOARD_INPUT | (HID_REPORT_TYPE_INPUT<<8));             //normal key report

        //cur_master_conn_device.char_handle[6] = blm_att_findHandleOfUuid128 (db128, my_SppS2CUUID);       //BLE Module,
        //cur_master_conn_device.char_handle[7] = blm_att_findHandleOfUuid128 (db128, my_SppC2SUUID);       //BLE Module,


        //add the peer device att_handle value to conn_dev_list after service discovery is correctly finished
        dev_char_info_add_peer_att_handle(&cur_master_conn_device);

        //peer device att_handle value store in flash
        dev_char_info_store_peer_att_handle(&cur_master_conn_device);
```

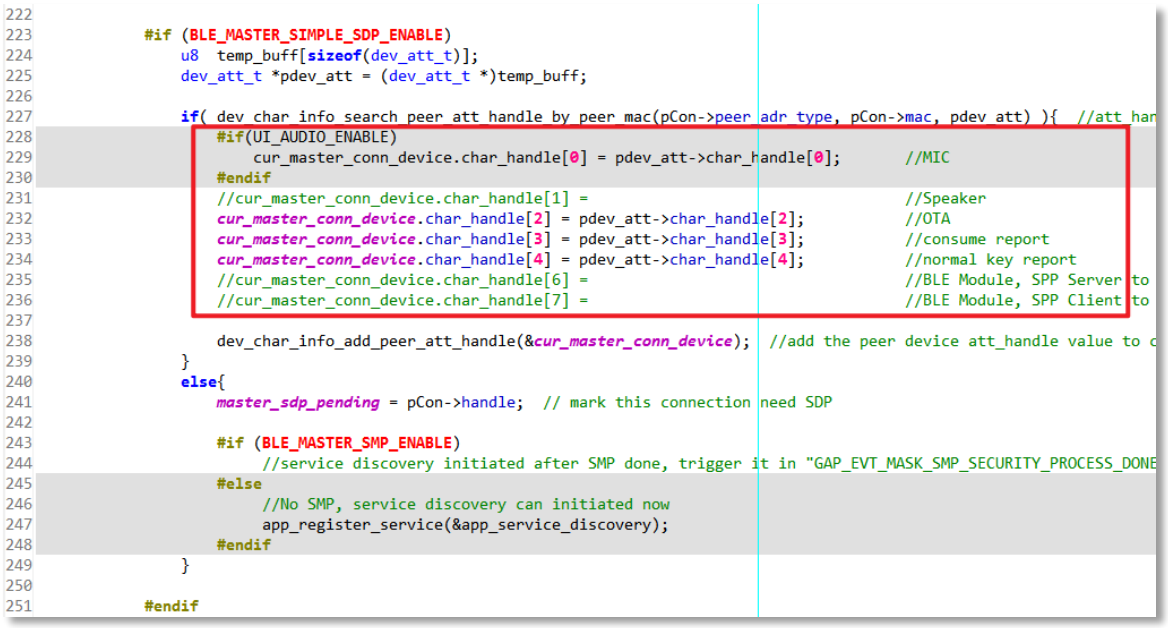
Please be noted: SDP is a very complicated part. For all telink SDKs, due to limited chip resources, SDP is not as complicated as that of mobile phones. Here is a simple reference.

In order to use the peer slave attribute handle, the user needs to call the dev_char_info_search_by_connhandle() function, which will search for the attribute handle data corresponding to it in the array conn_dev_list according to the connection handle given by the user and return the dev_char_info_t structure pointer, after which the user can get the desired attribute handle, as shown below.

```
dev_char_info_t* dev_info = dev_char_info_search_by_connhandle (connHandle);
```

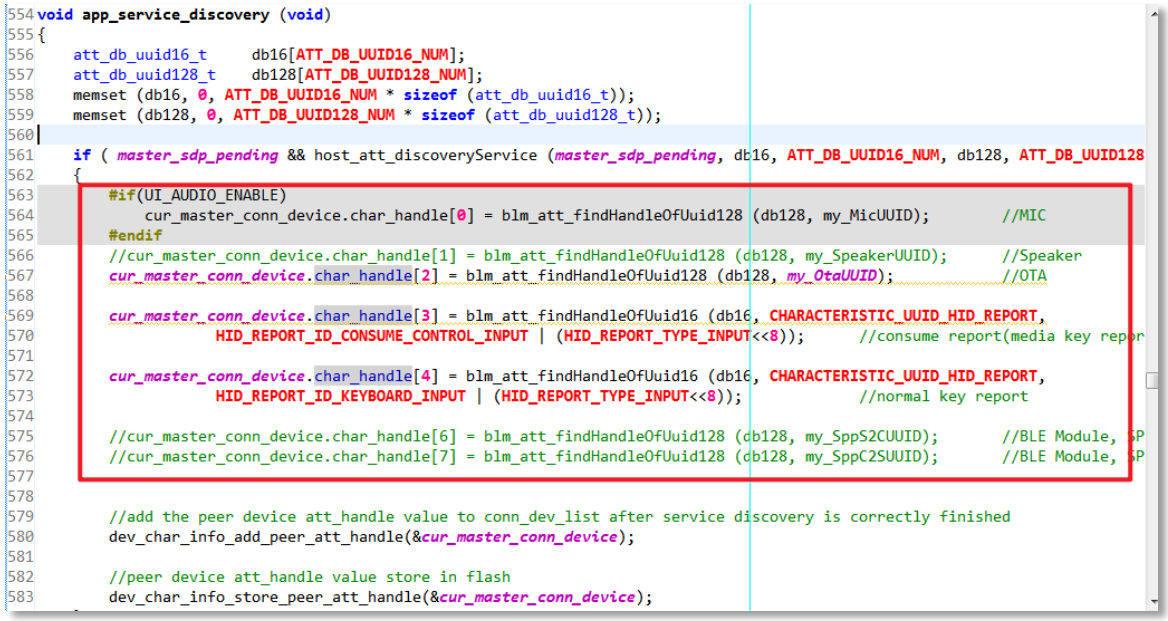
Please be noted that the index stored in the peer slave attribute handle needs to be modified according to the actual situation of the user and must be consistent, as shown in Figure 3-4, Figure 3-5, and Figure 3-6.

Figure 3-77 Connection Complete Event Processing char_handle in Functions



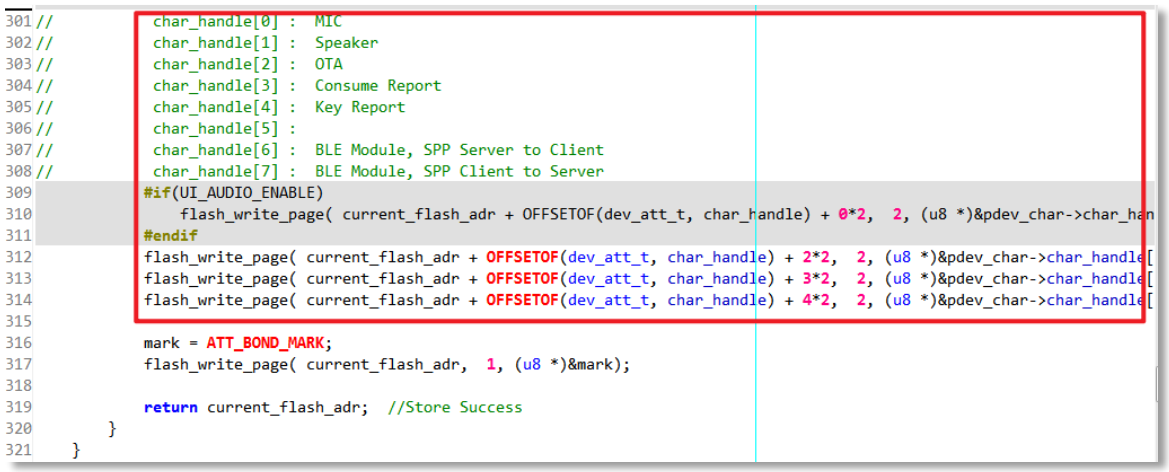
```
222
223         #if (BLE_MASTER_SIMPLE_SDP_ENABLE)
224             u8  temp_buff[sizeof(dev_att_t)];
225             dev_att_t *pdev_att = (dev_att_t *)temp_buff;
226
227             if( dev_char_info_search_peer_att_handle_by_peer_mac(pCon->peer_adr_type, pCon->mac, pdev_att) ){  //att_han
228                 #if(UI_AUDIO_ENABLE)
229                     cur_master_conn_device.char_handle[0] = pdev_att->char_handle[0];       //MIC
230                 #endif
231                 //cur_master_conn_device.char_handle[1] =                                   //Speaker
232                 cur_master_conn_device.char_handle[2] = pdev_att->char_handle[2];          //OTA
233                 cur_master_conn_device.char_handle[3] = pdev_att->char_handle[3];          //consume report
234                 cur_master_conn_device.char_handle[4] = pdev_att->char_handle[4];          //normal key report
235                 //cur_master_conn_device.char_handle[6] =                                   //BLE Module, SPP Server to
236                 //cur_master_conn_device.char_handle[7] =                                   //BLE Module, SPP Client to
237
238                 dev_char_info_add_peer_att_handle(&cur_master_conn_device);  //add the peer device att_handle value to c
239             }
240             else{
241                 master_sdp_pending = pCon->handle;  // mark this connection need SDP
242
243                 #if (BLE_MASTER_SMP_ENABLE)
244                     //service discovery initiated after SMP done, trigger it in "GAP_EVT_MASK_SMP_SECURITY_PROCESS_DONE
245                 #else
246                     //No SMP, service discovery can initiated now
247                     app_register_service(&app_service_discovery);
248                 #endif
249             }
250
251         #endif
```

Figure 3-78 char_handle in app_service_discovery（）



```
554 void app_service_discovery (void)
555 {
556     att_db_uuid16_t     db16[ATT_DB_UUID16_NUM];
557     att_db_uuid128_t    db128[ATT_DB_UUID128_NUM];
558     memset (db16, 0, ATT_DB_UUID16_NUM * sizeof (att_db_uuid16_t));
559     memset (db128, 0, ATT_DB_UUID128_NUM * sizeof (att_db_uuid128_t));
560
561     if ( master_sdp_pending && host_att_discoveryService (master_sdp_pending, db16, ATT_DB_UUID16_NUM, db128, ATT_DB_UUID128
562     {
563         #if(UI_AUDIO_ENABLE)
564             cur_master_conn_device.char_handle[0] = blm_att_findHandleOfUuid128 (db128, my_MicUUID);        //MIC
565         #endif
566         //cur_master_conn_device.char_handle[1] = blm_att_findHandleOfUuid128 (db128, my_SpeakerUUID);      //Speaker
567         cur_master_conn_device.char_handle[2] = blm_att_findHandleOfUuid128 (db128, my_OtaUUID);          //OTA
568
569         cur_master_conn_device.char_handle[3] = blm_att_findHandleOfUuid16 (db16, CHARACTERISTIC_UUID_HID_REPORT,
570                     HID_REPORT_ID_CONSUME_CONTROL_INPUT | (HID_REPORT_TYPE_INPUT<<8));      //consume report(media key repor
571
572         cur_master_conn_device.char_handle[4] = blm_att_findHandleOfUuid16 (db16, CHARACTERISTIC_UUID_HID_REPORT,
573                     HID_REPORT_ID_KEYBOARD_INPUT | (HID_REPORT_TYPE_INPUT<<8));             //normal key report
574
575         //cur_master_conn_device.char_handle[6] = blm_att_findHandleOfUuid128 (db128, my_SppS2CUUID);       //BLE Module, SP
576         //cur_master_conn_device.char_handle[7] = blm_att_findHandleOfUuid128 (db128, my_SppC2SUUID);       //BLE Module, SP
577
578
579         //add the peer device att_handle value to conn_dev_list after service discovery is correctly finished
580         dev_char_info_add_peer_att_handle(&cur_master_conn_device);
581
582         //peer device att_handle value store in flash
583         dev_char_info_store_peer_att_handle(&cur_master_conn_device);
```

Figure 3-79 char_handle in dev_char_info_store_peer_att_handle()



```
//          char_handle[0] :  MIC
//          char_handle[1] :  Speaker
//          char_handle[2] :  OTA
//          char_handle[3] :  Consume Report
//          char_handle[4] :  Key Report
//          char_handle[5] :
//          char_handle[6] :  BLE Module, SPP Server to Client
//          char_handle[7] :  BLE Module, SPP Client to Server
            #if(UI_AUDIO_ENABLE)
                flash_write_page( current_flash_adr + OFFSETOF(dev_att_t, char_handle) + 0*2,  2, (u8 *)&pdev_char->char_han
            #endif
            flash_write_page( current_flash_adr + OFFSETOF(dev_att_t, char_handle) + 2*2,  2, (u8 *)&pdev_char->char_handle[
            flash_write_page( current_flash_adr + OFFSETOF(dev_att_t, char_handle) + 3*2,  2, (u8 *)&pdev_char->char_handle[
            flash_write_page( current_flash_adr + OFFSETOF(dev_att_t, char_handle) + 4*2,  2, (u8 *)&pdev_char->char_handle[

            mark = ATT_BOND_MARK;
            flash_write_page( current_flash_adr,  1, (u8 *)&mark);

            return current_flash_adr;  //Store Success
        }
    }
```

4. Low Power Management

Low Power Management can also be called Power Management, in this document will be referred to as PM.

4.1 Low Power Driver

4.1.1 Low Power Mode

The 8x5x MCU is in working mode when executing the program normally, and the working current is between 3~7mA. If you need to save power, you need to enter a low-power mode. Currently in the multiple SDK, only the m1s1 project has low-power management (there is no power management in other projects) and only uses the suspend mode.

In the current SDK, users should use suspend carefully, because suspend is controlled by stack, improper use will disrupt the timing. Users can use deepsleep. Below we explain about suspend and deepsleep.

Low power mode (low power mode), also known as sleep mode, the following two types: suspend mode and deepsleep mode will be explained.

Table 4-1 Low Power Mode

module \ sleep mode		suspend	deepsleep
Sram		100% keep	100% lost
register	digital register	99% keep	100% lost
	analog register	100% keep	99% lost

The above table is the statistical description of the state saving of Sram, digital register and analog register in 2 sleep modes.

1） suspend mode

sleep mode 1

The program stops running at this time, similar to a pause function. Most hardware modules of the MCU are powered off, and the PM module maintains normal operation. At this time, the IC current is between 60~70uA. When suspend is woken up, the program continues to execute.

In suspend mode, all SRAM and analog registers can save state, and most digital registers keep state. There are a few of the digital register that will be powered down, including:

a) A small number of digital registers in the baseband circuit. The user needs to pay attention to the registers set by the API rf_set_power_level_index. As mentioned earlier in this document, this API needs to be called again after each suspend wake up.

b) Digital register that controls the Dfifo state. Corresponding to the relevant APIs in drivers/8258/dfifo.h, users must ensure that they are reset after each suspend wake_up when using these APIs.

2） deepsleep mode

sleep mode 2

At this time, the program operation stops, most of the hardware modules of the MCU are powered off, and the PM hardware module maintains its work. In deepsleep mode, the IC current is less than 1uA. If the standby current of the built-in flash appears around 1uA, it may result in a deepsleep of 1~2uA. When deepsleep mode wake_up, MCU will restart, similar to the effect of power-on, the program will restart the initialization.

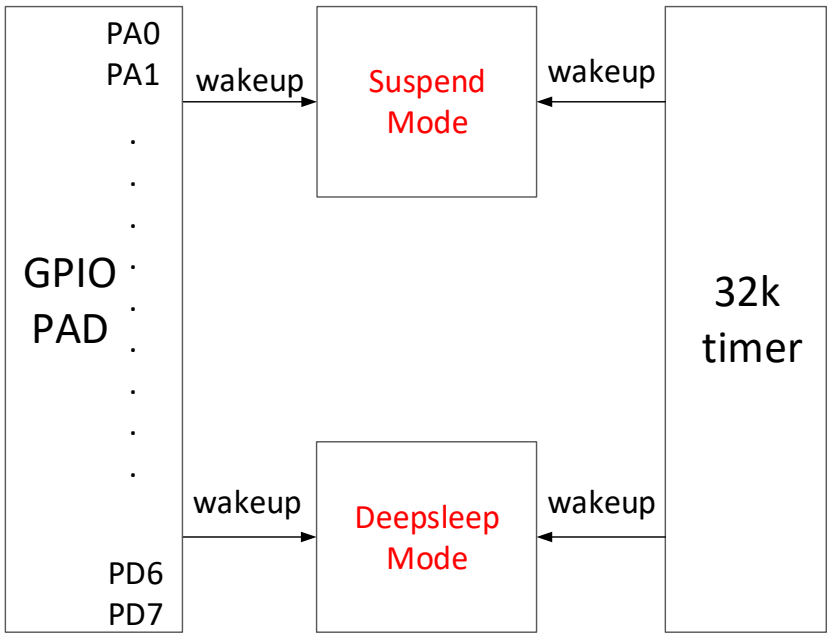
In deepsleep mode, except for a few registers on the analog register that can save the state, other SRAMs, digital registers, and analog registers are all powered down and lost.

4.1.2 Low-power wake-up source

The diagram of the low-power wake-up source of 8x5x MCU is as follows. Both suspend/deepsleep can be woken up by GPIO PAD and timer. In the BLE SDK, only two wake-up sources are concerned, as shown below (note that the two definitions of PM_TIM_RECOVER_START and PM_TIM_RECOVER_END in the code are not wake-up sources):

```
typedef enum {
	PM_WAKEUP_PAD	= BIT(4),
	PM_WAKEUP_TIMER = BIT(6),
}SleepWakeupSrc_TypeDef;
```

Figure 4-1 8x5x MCU Hardware Wake-up Source



As shown in the figure above, MCU suspend/deepsleep has 2 wake-up sources on the hardware: TIMER, GPIO PAD.

- The wake-up source PM_WAKEUP_TIMER comes from the hardware 32k timer (32k RC timer or 32k Crystal timer). The 32k timer has been properly initialized in the SDK. The user does not need any configuration when using it, just set the wake-up source in cup_sleep_wakeup().
- The wake-up source PM_WAKEUP_PAD comes from the GPIO module. All GPIOs (PAx/PBx/PCx/PDx) except the MSPI 4 pins have a wake-up function.

API to configure GPIO PAD to wake up 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 is defined as GPIO.

pol is the definition of wakeup polarity: Level_High means high level wakeup, Level_Low means low level wakeup.

en: 1 means enable, 0 means disable.

For example:

```
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 1); //GPIO_PC2 PAD wake on, high level wake
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 0); //GPIO_PC2 PAD wake up is off
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 1); //GPIO_PB5 PAD wake on, low level wake up
```

```
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 0); //GPIO_PB5 PAD wakes off
```

4.1.3 Low-power mode entry and wake-up

At present, in m1s1, suspend is controlled by stack. It is not recommended that customers set their own entry to suspend. However, the user can set to enter deepsleep mode.

The API for setting the MCU to sleep and wake up is:

```
int cpu_sleep_wakeup (SleepMode_TypeDef sleep_mode,
SleepWakeupSrc_TypeDef wakeup_src,
unsigned int wakeup_tick);
```

The first parameter sleep_mode: set the sleep mode, the current customer can only choose one: deepsleep mode. (Suspend is controlled by stack)

```
typedef enum {
        ......
        DEEPSLEEP_MODE                  = 0x80,
      ......
}SleepMode_TypeDef;
```

The second parameter wakeup_src: set the current wakeup source of deepsleep, the parameter can only be one or more of PM_WAKEUP_PAD, PM_WAKEUP_TIMER. If wakeup_src is 0, you cannot wake up after entering low-power sleep mode.

The third parameter wakeup_tick: When PM_WAKEUP_TIMER is set in wakeup_src, wakeup_tick needs to be set to determine when the timer will wake up the MCU. If PM_WAKEUP_TIMER is not set, this parameter is meaningless.

The value of wakeup_tick is an absolute value, set according to the System Timer tick introduced earlier in this document. When the value of System Timer tick reaches this set wakeup_tick, sleep mode is woken up. The value of wakeup_tick needs to be based on the current System Timer tick value plus the absolute time converted from the time needed to sleep to effectively control the sleep time. If you do not consider the current System Timer tick, directly set wakeup_tick, the time point of wake-up cannot be controlled.

Because wakeup_tick is an absolute time, it must be within the range that the 32-bit System Timer tick can represent, so the maximum sleep time that this API can represent is limited. The current design is that the maximum sleep time is 7/8 of the corresponding time of the maximum System Timer tick that can be expressed by 32bit. The maximum system timer tick can represent about 268S, then the maximum sleep time is 268*7/8=234 S, that is, the following delta_Tick cannot exceed 234 S.

```
cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_TIMER, clock_time() + delta_tick);
```

The return value is the set of wake-up sources in the current sleep mode. The wake-up sources corresponding to each bit of the return value are:

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

- *WAKEUP_STATUS_TIMER is 1,* indicating that the current sleep mode is awakened by Timer.
- *WAKEUP_STATUS_PAD* is 1, indicating that the current sleep mode is awakened by GPIO PAD
- WAKEUP_STATUS_TIMER and WAKEUP_STATUS_PAD are both 1, indicating that Timer and GPIO PAD two wake-up sources are effective at the same time.

- STATUS_GPIO_ERR_NO_ENTER_PM is a relatively special state, indicating that a GPIO wakeup error has occurred: for example, when a GPIO PAD is set to wake up high, and when this GPIO is high, try to call cpu_sleep_wakeup to enter suspend, and set PM_WAKEUP_PAD wake source. At this time, there will be no way to enter suspend, and the MCU immediately exits the cpu_sleep_wakeup function, giving the return value STATUS_GPIO_ERR_NO_ENTER_PM.

The following API is used to control the sleep time:

```
cpu_sleep_wakeup (DEEPSLEEP_MODE , PM_WAKEUP_TIMER,
clock_time() + delta_Tick);
```

delta_Tick is a relative time (e.g., 100* CLOCK_16M_SYS_TIMER_CLK_1MS), and the current clock_time() becomes the absolute time.

To illustrate the usage of cpu_sleep_wakeup:

1) cpu_sleep_wakeup (*SUSPEND_MODE* , *PM_WAKEUP_PAD*, 0);

   When the program executes this function, it enters suspend mode and can only be woken up by GPIO PAD.

2) cpu_sleep_wakeup (*DEEPSLEEP_MODE* , *PM_WAKEUP_TIMER*, clock_time() + 10* *CLOCK_16M_SYS_TIMER_CLK_1MS*;

   When the program executes this function, it enters deepsleep mode and can only be woken up by Timer. The wakeup time is the current time plus 10 ms, so the deepsleep time is 10 ms.

3) cpu_sleep_wakeup (*DEEPSLEEP_MODE* , *PM_WAKEUP_PAD* | *PM_WAKEUP_TIMER*,

   clock_time() + 50* *CLOCK_16M_SYS_TIMER_CLK_1MS*);

   When the program executes this function, it enters deepsleep mode and can be woken up by GPIO PAD and Timer. The timer wake-up time is set to 50ms. If the GPIO wake-up action is triggered before the end of 50ms, the MCU will be awakened by the GPIO PAD; if there is no GPIO action within 50ms, the MCU will be awakened by the Timer.

4) cpu_sleep_wakeup (*DEEPSLEEP_MODE*, *PM_WAKEUP_PAD*, 0);

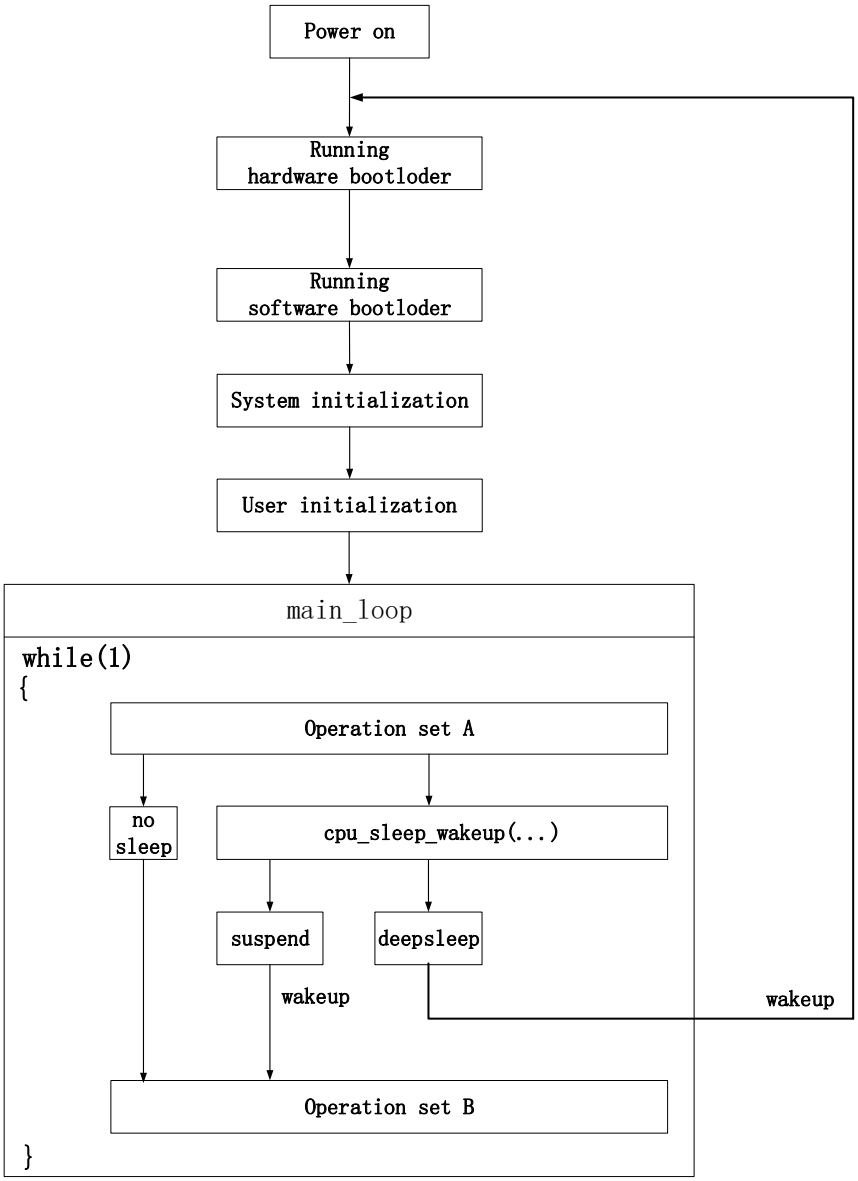
   When the program executes this function, it enters deepsleep mode and can be woken up by GPIO PAD.

4.1.4 Process after low power consumption wake-up

When the user calls the API cpu_sleep_wakeup, the MCU enters sleep mode; when the wake-up source triggers the MCU to wake up, the MCU software operation process is different for different sleep modes.

The MCU running process after suspend and deepsleep sleep modes are woken up is described in detail below. Please refer to the picture below.

Figure 4-2 Sleep Mode Wakeup Work Flow



Process after the MCU is powered on (Power on):

1) Run hardware bootloader

Some fixed actions are performed on the MCU hardware. These actions are solidified on the hardware and cannot be modified by the software.

Give a few examples to explain these actions, for example: read the boot boot flag of flash, determine whether the current firmware should be stored on flash address 0, or flash address 0x20000 (related to OTA); read the corresponding location of flash The value of, determine how much data needs to be copied from flash to Sram, as the data of resident memory (refer to Chapter 2 for the introduction of Sram allocation).

Running the hardware bootloader involves copying data from flash to sram, and the execution time is long. For example, copying 10K data takes about 5ms.

2) Run software bootloader

After the hardware bootloader finishes running, the MCU starts to run the software bootloader. Software bootloader is the vector segment introduced earlier (corresponding to the assembler in cstartup_8258_16K_RET.S).

Software bootloader is to set memory environment for the operation of the C language program behind, which can be understood as the initialization of the entire memory.

3) System initialization

System initialization corresponds to the initialization of each hardware module (including cpu_wakeup_init, rf_drv_init, gpio_init, clock_init) in the main function before cpu_wakeup_init to user_init, and sets the digital/analog register status of each hardware module.

4) User initialization

User initialization corresponds to the functions user_init or user_init_normal/ user_init_deepRetn in the SDK.

5) main_loop

After User initialization completes, enter main_loop controlled by while(1). A series of operations in main_loop before entering sleep mode is called "Operation Set A", and a series of operations after sleep wakes up is called "Operation Set B".

As illustration, the sleep mode process is detailed as below:

1) no sleep

If there is no sleep mode, the operation flow of MCU is to loop in while(1), and repeatedly execute "Operation Set A" -> "Operation Set B".

2) suspend

If the cpu_sleep_wakeup function is called to enter suspend mode, when suspend is awakened, it is equivalent to the normal exit of the cpu_sleep_wakeup function, and the MCU runs to "Operation Set B".

suspend is the cleanest sleep mode. During the suspend period, all Sram data can remain unchanged, and all digital/analog registers remain unchanged (with only a few special exceptions); after suspend wakes up, the program runs at the original location , Hardly need to consider any recovery of sram and register state. The disadvantage of suspend is the high power consumption.

3) deepsleep

If cpu_sleep_wakeup function is called to enter deepsleep mode, after deepsleep is woken up, the MCU will return to Run hardware bootloader.

It can be seen that the process of deepsleep wake_up and Power on are almost the same, and all software and hardware must be re- initialized.

After the MCU enters deepsleep, all Sram and digital/analog registers (with the exception of a few analog registers) will be powered down, so the power consumption is very low, and the MCU current is less than 1uA.

4.2 Low Power Management

4.2.1 BLE PM Initialization

If the low power consumption mode is used, the BLE PM module needs to be initialized and the following API can be called.

```
void        blc_ll_initPowerManagement_module(void);
```

If the low-power mode is not required and this API is not called, PM-related codes and variables will not be compiled into the program, which can save firmware size and sram size.

4.2.2 BLE PM for Link Layer

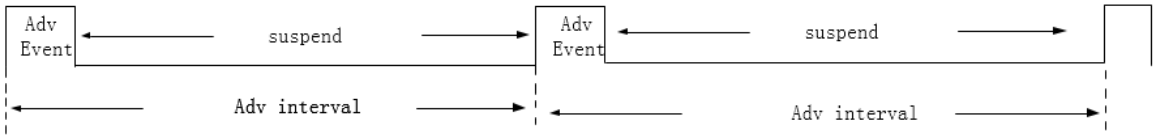
The SDK does low-power management for Advertising state, Scan state, connection master, connection slave, and connection master/slave.

Please be noted that currently the SDK does not use latency, that is, every interval will send and receive packets. Even if the slave accepts the connection parameters of the other party's master, where latency is not 0, the SDK will send and receive data according to the latency of 0.

4.2.2.1 suspend for advertise “only advertise”

When the m1s1 project only enables advertising and turns off the scan function, that is, when the Link Layer is in the Advertise state, the timing is as follows:

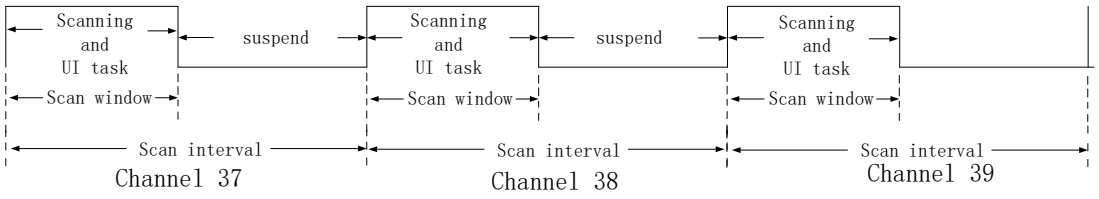
Figure 4-3 Timing Sequence of M1S1 in ADV Status



When the advertise time is reached, it will wake up from suspend and then process the advertising event. After the processing is completed, the stack will determine the difference between the next adv time point and the current time. If the conditions are met, it will enter suspend to reduce power consumption. The time consumed by Adv Event is related to the specific situation, for example: the user sets only 37 channel; the length of ADV packet is relatively small; scan req or conn req is received on channel 37 or 38, etc.

4.2.2.2 suspend for scan “only scan”

Figure 4-4 M1S1 Suspend for Scan for Only Scan



The actual Scan time is determined according to the size of the Scan window. If the Scan window is equal to the Scan interval, all the time is in Scan; if the Scan window is less than the Scan interval, select the time equal to the Scan window from the beginning to perform Scan in the Scan interval.

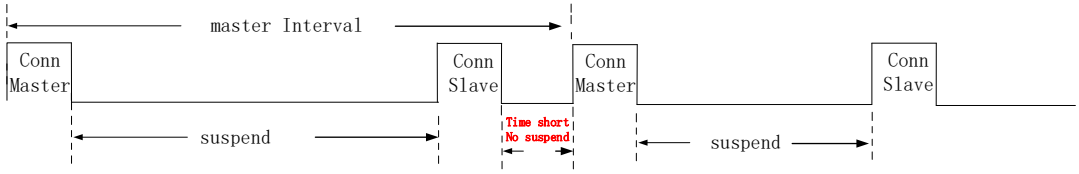
The Scan window shown in the figure is about 40% of the Scan interval. In the first 40% of the time, the Link Layer is in the Scanning state, the PHY layer is receiving packets, and users can use this time to execute their own UI tasks in the main_loop . In the later 60% of the time, the MCU enters suspend to reduce the power consumption of the whole machine.

The API for setting the ratio is as follows:

```
blc_llms_setScanParameter(SCAN_TYPE_PASSIVE,SCAN_INTERVAL_200MS,
SCAN_WINDOW_50MS,OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
```

4.2.2.3 suspend for connection

Figure 4-5 suspend for connection



Conditions to enter suspend are:

- The time interval between the next task and the end of the current task;
- Whether there is data unprocessed in the RX FIFO;
- The execution of BRX POST and BTX POST is completed;
- The device itself has no event pending.

If the time interval from the next task is relatively large, and there is no data in the RX FIFO, after the BRX POST or BTX POST is executed, the stack will let the MCU enter suspend. When the next task arrives, the timer wakes up the MCU to start the task.

4.2.3 API blc_pm_setSuspendMask

API for configuring low power management:

```
    void        blc_pm_setSuspendMask (u8 mask);
```

Use blc_pm_setSuspendMask to set blmsPm.suspend_mask (default is USPEND_DISABLE).

The source codes of these two APIs are:

```
    void        blc_pm_setSuspendMask (u8 mask)
{
        blmsPm.suspend_mask = mask;
......
}
```

For the setting of blmsPm.suspend_mask, you can select one of the following values, or select the "or operation" of multiple values.

```
typedef enum {
        PM_SUSPEND_DISABLE        = 0,
```

```
        PM_SUSPEND_ADV          = BIT(0),
        PM_SUSPEND_SCAN         = BIT(1),
        PM_SUSPEND_SLAVE        = BIT(2),
        PM_SUSPEND_MASTER       = BIT(3),
}pm_mask_t;
```

SUSPEND_DISABLE indicates sleep disable and does not allow the MCU to enter suspend.

SUSPEND_ADV and SUSPEND_SCAN are used to control the MCU to enter suspend when Advertising state and Scan state, respectively.

SUSPEND_SLAVE and SUSPEND_MASTER are used to control the Slave role and Master role when the MCU enters suspend.

The two most common situations of this API are as follows:

1) blc_pm_setSuspendMask(SUSPEND_DISABLE);

The MCU is not allowed to enter suspend

2）blc_pm_setSuspendMask(PM_SUSPEND_ADV | PM_SUSPEND_SCAN | PM_SUSPEND_SLAVE | PM_SUSPEND_MASTER);

The MCU is allowed to enter suspend in the Advertising state, Scan state, master role and slave role.

Please be noted that in M1S1, adv will be converted to slave after connection. At this time, the adv task will be removed from the stack and converted to slave task. Similarly, after scan connection, it will be converted to master. At this time, the stack will remove the scan task and convert to master task.

If user_init() is initialized, we set adv enable and scan enable. If you want to enter suspend, you must set both adv and scan to enable suspend. If you only set adv or scan, you will not enter suspend. The same is for the master and slave. If only the master is enabled or only the slave is enabled in suspend, it will not enter suspend in the connected state. which is:

```
bls_pm_setSuspendMask(PM_SUSPEND_ADV | PM_SUSPEND_SCAN）
```

You can think of adv and scan as a whole, and master and slave after successful connection as a whole. There will be pseudo code in the explanation of the PM software process.

4.2.4 API blc_pm_setWakeupSource

The user sets the MCU to enter sleep mode (suspend or deepsleep) through the above blc_pm_setSuspendMask, and can set the wake source of sleep mode through the following API.

```
    void        blc_pm_setWakeupSource(u8 source);
```

Source can choose wake up source PM_WAKEUP_PAD.

The API sets the underlying variable bltPm.wakeup_src. The source code in the SDK is:

```
void blc_pm_setWakeupSource (u8 src)
{
        blmsPm.wakeup_src = src;
}
```

When the MCU is in suspend or deepsleep mode, the actual wakeup source is:

blmsPm.wakeup_src | *PM_WAKEUP_TIMER*

That is, PM_WAKEUP_TIMER will definitely exist and does not depend on the user's settings. This is to ensure that the MCU must wake up at a specific time point to process the next ADV task, SCAN task, master task, and slave task.

Each time blc_pm_setWakeupSource is called to set the wakeup source, once the MCU enters sleep mode and is woken up, blmsPm.wakeup_src will be cleared to 0.

4.2.5 PM Software Processing Flow

The software processing flow of low power consumption management will be described below using a combination of code and pseudocode. The purpose is to let the user understand all the logical details of the processing flow.

4.2.5.1 blt_llms_sdk_main_loop

In the SDK, blt_llms_sdk_main_loop is called repeatedly in a while (1) structure.

```
while(1)
{
    ////////////////////// BLE entry ////////////////////////
    blt_llms_sdk_main_loop ();
/////////////
////////// UI entry ////////////////////////
//UI  task
////////////////////// user PM mask setting ////////////////////////
blc_pm_setSuspendMask( PM_SUSPEND_ADV | PM_SUSPEND_SCAN | PM_SUSPEND_SLAVE |
PM_SUSPEND_MASTER);
}
```

The blt_llms_sdk_main_loop function is continuously executed in while(1), and the BLE low-power management code is in the blt_llms_sdk_main_loop function, so the low-power management code is also being executed all the time.

The following is the implementation of low power management logic in the blt_llms_sdk_main_loop function.

```
int   blt_llms_sdk_main_loop (void)
{
    ……
if(blmsPm. suspend_mask == SUSPEND_DISABLE && blmsPm.sleep_tick < BLMS_PM_ALLOWED_TIMING_MARGIN )
{
        return  0; // SUSPEND_DISABLE, can not enter sleep mode;sleep time                //too
short, can not enter sleep mode.
    }
//////////////////////////
if( bltSlot.task_mask && (blmsPm.suspend_mask & bltSlot.task_mask) !=    bltSlot.task_mask   )
 // Is there a task (adv, scan, master, slave)
// Whether the suspend mask allows this state (adv, scan, master, slave) to enter suspend.
{
        return 0;
    }
//////////////////////
   If ( suspend_allowed & (brx_post | btx_post | adv_post | scan_post) == 0 )
{
return 0; // Allow suspend only after each task is completed
}
    else
    {
        blt_brx_sleep (); //process sleep & wakeup
}
```

```
}
```

1) When bltmsPm. suspend_mask is SUSPEND_DISABLE, exit directly without executing blt_brx_sleep function. Therefore, when the user uses bls_pm_setSuspendMask (SUSPEND_DISABLE), the logic of low power management will be completely invalidated, the MCU will not enter low power, and the loop of while(1) has been executing.

2) If the sleep time is too short, it will not enter suspend.

3) When there are tasks, such as adv task, scan task, master task, slave task, if the suspend_mask of the corresponding task is not enabled, it will not enter the low power mode. In M1S1, adv will be converted to slave after connection. At this time, the adv task will be removed from the stack and converted to slave task. Similarly, after scan connection, it will be converted to master. At this time, the stack will remove the scan task and convert to master task.

4) If the Adv Event or Scan Event or Bnx Event of Conn state Master role or Brx Event of Conn state Slave role is being executed, the blt_brx_sleep function will not be executed, because the RF task is running at this time, the SDK needs Make sure to enter sleep mode after Adv Event/Brx Event ends.

Only when the above conditions are met, the blt_brx_sleep function will be executed.

4.2.5.2 blt_brx_sleep

The logical implementation of the blt_brx_sleep function is shown below.

```
void blt_brx_sleep (void)
{
......
      if(blmsPm.next_slot_task & SLOT_TASK_CONN){
         current_wakeup_tick = blmsPm.next_slot_tick - margin;  //Calculate the next wake-up time point
      }
     ......

blmsPm.current_wakeup_tick = current_wakeup_tick;//Record wake-up time
//Execute BLT_EV_FLAG_SUSPEND_ENTER callback function
blt_p_event_callback (BLT_EV_FLAG_SUSPEND_ENTER, NULL, 0);
//Enter low power function
cpu_sleep_wakeup (SUSPEND_MODE, (PM_WAKEUP_TIMER | blmsPm.wakeup_src | blmsPm.pm_border_flag),
current_wakeup_tick);
......
//Execute BLT_EV_FLAG_SUSPEND_EXIT callback function
 blt_p_event_callback (BLT_EV_FLAG_SUSPEND_EXIT, NULL, 0);
......
blmsPm.wakeup_src = 0;
}
```

The above is the brief flow of the blt_brx_sleep function. Here we see the timing of the execution of several suspend related event callback functions: BLT_EV_FLAG_SUSPEND_ENTER, BLT_EV_FLAG_SUSPEND_EXIT.

In suspend mode, the API cpu_sleep_wakeup in the driver is finally called:

```
cpu_sleep_wakeup (      SUSPEND_MODE,
PM_WAKEUP_TIMER | blmsPm.wakeup_src,        T_wakeup);
```

The wake-up source is PM_WAKEUP_TIMER | blmsPm.wakeup_src. Timer is unconditionally effective to ensure that the MCU wakes up before the next task arrives.

When the blt_brx_sleep function exits, the value of blmsPm.wakeup_src is reset, so it is necessary to pay attention to the API blc_pm_setWakeupSource to set the life cycle of the wakeup source. The value set each time is only valid for the sleep mode to be entered last time.

4.2.6 API blc_pm_getSystemWakeupTick

The following API is used to obtain the suspend wake up time (System Timer tick) of the low power management calculation, namely T_wakeup.

```
u32 blc_pm_getSystemWakeupTick(void);
```

The calculation of T_wakeup is before the cpu_sleep_wakeup function is processed, and the application layer can only get accurate T_wakeup in the BLT_EV_FLAG_SUSPEND_ENTER event callback function.

Suppose the user needs to wake up by pressing the key when the suspend time is relatively long. Below we explain the setting method.

We need to use the BLT_EV_FLAG_SUSPEND_ENTER event callback function and blc_pm_getSystemWakeupTick.

The callback registration method of BLT_EV_FLAG_SUSPEND_ENTER is as follows:

```
blc_llms_registerTelinkControllerEventCallback(     BLT_EV_FLAG_SUSPEND_ENTER,
                    &ble_set_sleep_wakeup);
void ble_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( ((u32)(blc_pm_getSystemWakeupTick() - clock_time())) >
                    50 * CLOCK_SYS_CLOCK_1MS){
        blc_pm_setWakeupSource(PM_WAKEUP_PAD);
    }
}
```

In the example above, if the suspend time exceeds 50ms, add GPIO to wake up. User can adjust according to the actual situation. However, currently multiple SDK does not use latency, and every interval will send and receive packets, so the longest suspend time is only related to the interval between master and slave.

Here only provides an interface, the customer decides whether to use according to the actual situation.

4.3 Precautions for GIPO Wakeup

Can not Enter Sleep mode when the wake-up level is valid

Because the 8x5x GPIO wakes up by high and low levels instead of rising and falling edges, when GPIO PAD is configured to wake up, such as setting a GPIO PAD high level to suspend, make sure that the MCU calls cpu_wakeup_sleep to enter suspend at this time, the current level read by this GPIO cannot be high. If the current level is already high, it actually enters the cpu_wakeup_sleep function, which is invalid when suspend is triggered, and it will immediately exit, that is, it does not enter suspend at all.

Users should pay attention to avoid this problem when using Telink's GPIO PAD to wake up.

If the application layer does not avoid this problem, when the cpu_wakeup_sleep function is called, the GPIO PAD wakeup source has taken effect. In order to prevent the program from entering unpredictable logic, the PM driver has made some improvements:

1) suspend

If it is suspend, it will quickly exit the function cpu_wakeup_sleep, the return value given by this function may appear in two cases:

- The GPIO PAD valid status is detected on the PM module, and it returns WAKEUP_STATUS_PAD
- The GPIO PAD valid status is not detected on the PM module, and it returns STATUS_GPIO_ERR_NO_ENTER_PM

2) deepsleep mode

If it is in deepsleep mode, the PM driver will automatically reset the MCU at the bottom (the reset at this time is the same as the watchdog reset effect), and the program returns to "Run hardware bootloader" to start running again.

5. Low Battery Detect

Battery power detect/check other names may appear in the Telink BLE SDK and related documents, including: battery power detect/check, low battery detect/check, low power detect/check, battery detect/check, etc. For example, related files and functions in the SDK appear named battery_check, battery_detect, battery_power_check, etc.

This document uses the name "low battery detect" to explain. In addition, in the multi-master multi-slave SDK, currently only M1S1 supports battery detection.

5.1 Importance of low power detect

For battery-powered products, due to the gradual decline in battery power, when the voltage is reduced to a certain value, it will cause many problems:

1) The operating voltage range of 8x5x is 1.8V~3.6V. When the voltage is lower than 1.8V, 8x5x can no longer guarantee stable operation.

2) When the battery voltage is low, due to the instability of the power supply, the Flash "write" and "erase" operations may have the risk of error, causing the program firmware and user data to be abnormally modified, which eventually leads to product failure. According to previous mass production experience, we set this low-pressure threshold that may be at risk to 2.0V.

As described above, a battery-powered product must set a safe voltage value (secure voltage), only when the voltage is higher than this safe voltage, the MCU is allowed to continue working; once the voltage is lower than the safe voltage, the MCU stops running, it needs to be shut down immediately (using the SDK to enter deepsleep mode to achieve).

Before the MCU is shut down, some behaviors of the UI (for example: the rapid flashing of the LED light) can be used to inform the product user. This UI behavior is called low-voltage alarm. When the user of the product sees the behavior of the low-voltage alarm, he understands that the battery is currently in a low-power state, and can charge or replace the battery.

The safety voltage is also called the alarm voltage. This voltage value is 2.0V by default in the SDK. If the user has an unreasonable design in the hardware circuit, resulting in the deterioration of the stability of the power network, the safe voltage value needs to be increased, such as 2.1V, 2.2V, etc.

For products developed and implemented by the Telink BLE SDK, as long as battery power is used, low power detect must be a real-time task for the entire life cycle of the product to ensure product stability.

5.2 Implementation of low battery detect

Low power detect requires the use of an ADC to measure the power supply voltage. For the user, please refer to the document "8258 Datasheet" and the relevant documentation of the ADC driver, understanding the 8x5x ADC module.

The implementation of low battery detect is explained with the implementation given in the SDK demo "8258_m1s1". Refer to the files battery_check.h and battery_check.c.

You must ensure that the macro "BATT_CHECK_ENABLE" in the app_config.h file is enabled. This macro is enabled by default, and the user should not modify it

```
#define BATT_CHECK_ENABLE                   1    //must enable
```

5.2.1 Precautions for low battery detect

Low battery detect is a basic ADC sampling task. There are some issues that need to be noted when implementing ADC sampling power supply voltage, as described below.

5.2.1.1 GPIO input channel must be used

Telink's previous generation 8267/8269 IC supports ADC sampling of the power supply voltage on the "VCC/VBAT" input channel. This design is also retained on the 8x5x ADC input channel, corresponding to the last "VBAT" in the variable ADC_InputPchTypeDef below.

However, due to some special reasons, 8x5x "VBAT" channel cannot be used, so Telink stipulates that: "VBAT" input channel is not allowed, and GPIO input channel must be used.

The available GPIO input channels are the input channels corresponding to PB0~PB7, PC4, and 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 the power supply voltage using the GPIO input channel.

- In the hardware circuit design, the power supply is directly connected to the GPIO input channel. When the ADC is initialized, set the GPIO to a high-impedance state (ie, oe, and output are all set to 0). At this time, the voltage on the GPIO is equal to the power supply voltage, and ADC sampling can be performed directly.
- The hardware circuit does not require power supply and GPIO input channel connection. Use GPIO output high level to measure. The 8x5x internal circuit structure design can ensure that the GPIO output high-level voltage value and the power supply voltage value are always equal.Then the high level of the GPIO output can be used as the power supply voltage, and ADC sampling is performed through the GPIO input channel.

At present, the GPIO input channel selected by "8258_m1s1" is PB7, and the second "power source does not connect with the GPIO input channel" method is used.

Select PB7 as the GPIO input channel, PB7 as the ordinary GPIO function, all states (ie, oe, output) can use the default state during initialization, without special modification.

```
#define GPIO_VBAT_DETECT            GPIO_PB7
#define PB7_FUNC                    AS_GPIO
#define PB7_INPUT_ENABLE            0
#define ADC_INPUT_PCHN              B7P
```

When ADC sampling is required, PB7 outputs high level:

```
gpio_set_output_en(GPIO_VBAT_DETECT, 1);
gpio_write(GPIO_VBAT_DETECT, 1);
```

Please be noted that the GPIO selected for battery detection cannot be multiplexed. GPIO can only do battery detect.

5.2.1.2 Only differential mode can be used

Although 8x5x ADC input mode supports both Single Ended Mode and Differential Mode, for some specific reasons, Telink stipulates that only differential mode can be used, and single-ended mode is not allowed.

The input channels in differential mode are divided into positive input channel and negative input channel. The measured voltage value is the positive input channel voltage minus the negative input channel voltage.

If there is only one input channel sampled by the ADC, when using the differential mode, set the current input channel to the positive input channel and set GND to the negative input channel. In this way, the voltage difference between the two is equal to the positive input channel voltage.

The low-voltage detection in the SDK uses the differential mode, and the code is as follows. "#If 1" and "#else" branch are the same function settings, "#if 1" is just to make the code run faster to save time. It can be understood by looking at "#else". In the adc_set_ain_channel_differential_mode API, PB7 was selected as the positive input channel, and GND as the 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 mode to obtain ADC sample value

Telink's previous generation 826x series of ICs used the way of reading registers to obtain ADC sampling results. For 8x5x, Telink stipulates: use only DFIFO mode to read the ADC sample value. Refer to the implementation of the following functions in dirver.

```
unsigned int adc_sample_and_get_result(void);
```

5.2.1.4 Different ADC tasks need to be switched

As described in "8258 Datasheet", ADC state machine includes Left, Right, Misc and other channels. For some special reasons, these state channels cannot work at the same time. Telink stipulates that the channels in the ADC state machine must run independently and cannot work at the same time.

As a basic ADC sampling, low voltage detection uses Misc channel. If users need other ADC tasks than low voltage detection, they also need to use Misc channel. Amic Audio uses Left channel. Low-voltage detection cannot run simultaneously with Amic Audio and other ADC tasks, and must be implemented by switching.

5.2.2 API Low Battery Detect API

5.2.2.1 ADC Initialization

The sequence of ADC initialization must meet the following procedure: first power off (power down) the SAR ADC, then configure other parameters, and finally power on (power on) the SAR ADC. All ADC sampling initialization must follow this process.

```
void adc_vbat_detect_init(void)
{
    /******power off sar adc********/
    adc_power_on_sar_adc(0);

    //add ADC configuration
    /******power on sar adc********/
    //note: this setting must be set after all other settings
    adc_power_on_sar_adc(1);
}
```

Sar adc power on and power off the previous configuration, the user should try not to modify, use these default settings. If the user selects a different GPIO input channel, directly modify the definition of the macro "ADC_INPUT_PCHN". If the user's hardware circuit adopts the design of "power connected to GPIO input channel", the operation of "GPIO_VBAT_DETECT" output high level needs to be removed.

The code called by the adc_vbat_detect_init initialization function in app_battery_power_check is:

```
    if(!adc_hw_initialized){
        adc_hw_initialized = 1;
        adc_vbat_detect_init();
}
```

A variable adc_hw_initialized is used here. Only when the variable is 0, the initialization is called once and set to 1; when the variable is 1, it is no longer initialized. adc_hw_initialized will also be operated in the following API.

```
void battery_set_detect_enable (int en)
{
    lowBattDet_enable = en;
    if(!en){
        adc_hw_initialized = 0;    //need initialized again
    }
}
```

The functions that can be realized by the design using adc_hw_initialized are:

1) Without considering the effect of sleep mode (suspend/deepsleep retention), we only analyze the switching between low-power detection and other ADC tasks.

Because of the need to consider switching between low-power detection and other ADC tasks, adc_vbat_detect_init may be executed multiple times, so it cannot be written to user intilialization and must be implemented in main_loop.

When the app_battery_power_check function is executed for the first time, adc_vbat_detect_init is executed, and it will not be executed repeatedly.

Once the "ADC other task" needs to be executed, the ADC's right to use will be snatched to ensure that the "ADC other task" must call battery_set_detect_enable(0) when it is initialized. At this time, adc_hw_initialized will be cleared to 0.

After "ADC other task" is completed, hand over the right to use ADC. app_battery_power_check is executed again. Since the value of adc_hw_initialized is 0, adc_vbat_detect_init must be executed again to ensure that the low power detect will be re-initialized every time it is switched back.

2) Adaptive processing of suspend and deepsleep retention consider sleep mode.

The variable adc_hw_initialized must be defined as a variable in the "data" segment or "bss" segment, and cannot be defined in retention_data. Defined in the "data" section or "bss" can ensure that this variable will be re-initialized to 0 each time the software bootloader (ie cstartup_xxx.S) is executed after deepsleep retention wake_up; this variable can remain unchanged after sleep wake_up.

The common feature of the register configured in the adc_vbat_detect_init function is that it does not power down in suspend mode and can save the state; it will power down in deepsleep retention mode.

If the MCU enters suspend mode and executes app_battery_power_check again after waking up, the value of adc_hw_initialized is the same as before suspend, and there is no need to re-execute the adc_vbat_detect_init function.

If the MCU enters deepsleep retention mode, adc_hw_initialized is 0 after waking up, you must re-execute adc_vbat_detect_init, ADC related register state needs to be reconfigured.

The state of the register set in the adc_vbat_detect_init function can be kept during the suspend without power down.

Refer to the description of suspend mode in the "Low Power Management" section of the document. The Dfifo related registers will be powered down in suspend mode, so the following two codes are not placed in the adc_vbat_detect_init function, but in the app_battery_power_check function to ensure that each low Reset all before electrical detection.

```
adc_config_misc_channel_buf((u16 *)adc_dat_buf,ADC_SAMPLE_NUM<<2);
dfifo_enable_dfifo2();
```

The keyword "_attribute_ram_code_" has been added to the adc_vbat_detect_init function in the SDK to set it as ram_code. The final purpose is to optimize the power consumption of the long sleep connection state. For example, for a typical long sleep connection of 10ms * (99+1) = 1S, wake up every 1S, and the middle long sleep uses deepsleep retention mode, then adc_vbat_detect_init will be re-executed after each wake up The execution speed will become faster after ram_code.

This "_attribute_ram_code_" is not necessary. In the application of the product, the user can decide whether to put this function into the ram_code according to the usage of the deepsleep retention area and the result of the power consumption test.

5.2.2.2 Low battery detect processing

In main_loop, call the app_battery_power_check function to implement low-power detect. The relevant codes are as follows:

```
_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);
	}
```

The default value of lowBattDet_enable is 1, low power detect is enabled by default, and the low power detect starts immediately after the MCU is powered on. This variable needs to be set to retention_data to ensure that deepsleep retention cannot modify its state.

The value of lowBattDet_enable can only be changed when other ADC tasks need to preempt the right to use the ADC: when other ADC tasks start, battery_set_detect_enable(0) is called, and app_battery_power_check function will not be called in main_loop; after other ADC tasks, battery_set_detect_enable is called (1), hand over the right to use ADC, at this time you can call app_battery_power_check function in main_loop.

The variable lowBattDet_tick controls the frequency of low power detect. The low power detect is performed once every 500mS in the Demo. User can modify this time value according to your needs.

The specific implementation of the app_battery_power_check function looks more complicated, involving the initialization of low power detect, preparation of Dfifo, data acquisition, data processing, low power alarm processing, etc.

Because the use of ADC is more complicated, and there are some special restrictions on the hardware circuit, it is difficult for the user to understand all the details. The processing of every detail in this part of the processing flow (this document will not introduce every detail) is very particular, so users should not try to modify, try to use the original demo code. There are only a few places that can be modified, this document will clearly point out; please do not modify places with clear indication that can be modified.

The acquisition of ADC sampling data uses Dfifo mode, Dfifo samples 8 data by default, and calculates the average value after removing the maximum and minimum values. In the adc_vbat_detect_init function, you can see that each adc sampling period is 10.4uS, so the data acquisition process is about 83us.

You can see that the macro "ADC_SAMPLE_NUM" in the Demo can be modified to 4 to shorten the ADC sampling time to 41uS. The method of using 8 data is recommended, the calculation result will be more accurate.

```
#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
```

The app_battery_power_check function is placed on the ram_code, referring to the description of the "adc_vbat_detect_init" ram_code above, it is also to save running time and optimize power consumption.

This "_attribute_ram_code_" is not necessary. In the application of the product, the user can decide whether to put this function into the ram_code according to the usage of the deepsleep retention area and the result of the power consumption test.

```
_attribute_ram_code_ int app_battery_power_check(u16 alram_vol_mv);
```

5.2.2.3 Low voltage alarm

The parameter alram_vol_mv of app_battery_power_check specifies the low voltage detection alarm voltage in mV. According to the previous introduction, the default setting in the SDK is 2000 mV. In the low voltage detection of main_loop, when the power supply voltage is lower than 2000mV, it enters the low voltage range.

The demo code for handling low voltage alarms is shown below. The MCU must be shut down after low voltage, and no other work can be done.

"8258_m1s1" uses the way to enter deepsleep to implement shutdown MCU. In addition to shutdown, the user can modify other alarm behaviors in the processing of low-voltage alarms.

In the following code, the LED light flashes 3 times to inform the product user that the battery needs to be charged or replaced.

```
if(batt_vol_mv < alram_vol_mv){
        #if (1 && BLT_APP_LED_ENABLE)   //led indicate
            gpio_set_output_en(GPIO_LED, 1);   //output enable
            for(int k=0;k<3;k++){
                gpio_write(GPIO_LED, LED_ON_LEVAL);
                sleep_us(200000);
                gpio_write(GPIO_LED, !LED_ON_LEVAL);
                sleep_us(200000);
            }
        #endif

        analog_write(DEEP_ANA_REG2, LOW_BATT_FLG);   //mark
        cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);
}
```

The SDK will quickly perform a low-power detect during user initialization, instead of waiting for the main_loop test. The reason for this processing is to avoid application errors. The following are examples:

If the LED flashes to remind the product user when the low battery alarm occurs, and then enters deepsleep and is awakened, from the processing of main_loop, it takes at least 500mS to perform the low battery detection. Before 500mS, the slave advertising package has been sent for a long time, and it may be connected to the master. In this case, there will be a bug that the device that has been alarmed by low battery continues to work.

For this reason, the SDK must do low-power detection in advance during user initialization, and this must be prevented at this step. So during user initialization, add low battery detection:

```
    if(analog_read(DEEP_ANA_REG2) ==   LOW_BATT_FLG){
        app_battery_power_check(VBAT_ALRAM_THRES_MV + 200);   //2.2 V
}
```

According to the value of the DEEP_ANA_REG2 analog register, you can determine whether the low battery alarm shutdown has been awakened. At this time, a fast low battery detection is performed, and the previous 2000mV alarm voltage is increased to 2200mV (called recovery voltage). The reason for the increase of 200mV is:

There will be some errors in low-voltage detection, which cannot guarantee the accuracy and consistency of the measurement results. For example, if the error is 20mV, it may be that the first detected voltage is 1990mV and it enters shutdown mode, and then the voltage value detected again during user initialization after wake-up is 2005mV. If the alarm voltage is still 2000mV, the bug described above cannot be prevented.

Therefore, it is necessary to increase the alarm voltage slightly when the rapid low-power detection after the wake-up in shutdown mode is performed, and the amplitude of the adjustment is slightly larger than the maximum error of the low-power detection.

Only when a low-voltage detection finds that the voltage is lower than 2000mV and enters the shutdown mode, the recovery voltage of 2200mV will appear, so the user does not need to worry that this 2200mV will falsely report low voltage to products with actual voltages of 2V~2.2V. After the user of the product sees the low-voltage alarm indication, after recharging or replacing the battery, it meets the requirements for voltage recovery and the product resumes normal use.

5.2.2.4 Low power detection debug mode

In the "8258_m1s1" Demo code, two debug-related macros are reserved for users to debug.

```
#define DBG_ADC_ON_RF_PKT              0
#define DBG_ADC_SAMPLE_DAT             0
```

Only when debugging is it possible to open the above two "macro".

After "DBG_ADC_SAMPLE_DAT" is turned on, the intermediate result of ADC sampling can be stored on Sram.

When "DBG_ADC_ON_RF_PKT" is enabled, the ADC sampling result information will be displayed on the advertising packet and the data packet of the key value in the connection state. Note: At this time, the advertising package and key data are modified, so it can only be used for debugging.

When "DBG_ADC_SAMPLE_DAT" is enabled, the intermediate result of ADC sampling can be stored on Sram.

6. Audio

For this part, please refer to the 825x single connection SDK handbook first, only the differences are introduced later.

Only the Master Role of this SDK supports decompressing 4-bit ADPCM Audio data reported by the peer Slave device (RCU voice remote control) into pcm data, and then transferring the data to the host via USB. .

6.1 Audio Initialization

Currently only the 8258_m4s3 project supports this feature, which is enabled by default.

If users need to use it, open the following definition macro in vendor/8258_m4s3/app_config.h:

```
#define APPLICATION_DONGLE                          1
#define UI_AUDIO_ENABLE                             1
```

6.2 Audio Data Processing

The original sound data sampled by ACU/Dmic by the RCU voice remote controller is in pcm format, and compressed into the adpcm format using the pcm to adpcm algorithm. The compression rate is 25% to reduce the amount of BLE RF data. The local master devic will decompress and restore the received data in adpcm format to pcm format data.

Regarding the volume of voice-related data, define relevant macros in vendor/8258_m4s3/app_config.h:

```
////////////////////// Audio ////////////////////////////////////
#define MIC_RESOLUTION_BIT      16
#define MIC_SAMPLE_RATE             16000
#define MIC_CHANNLE_COUNT           1
#define   MIC_ENOCDER_ENABLE        0

//////////////////////////// MIC BUFFER ////////////////////////////
#define   MIC_ADPCM_FRAME_SIZE      128
#define   MIC_SHORT_DEC_SIZE        248

#define   MIC_ADPCM_FRAME_SIZE_NUM    4
#define   MIC_SHORT_DEC_SIZE_NUM      4
```

Define the abuf_mic of the ADPCM data buffer reported from the peer slave device (RCU voice remote control):

```
    u8          abuf_mic[MIC_ADPCM_FRAME_SIZE * MIC_ADPCM_FRAME_SIZE_NUM];
//128 * 4 = 512 bytes
```

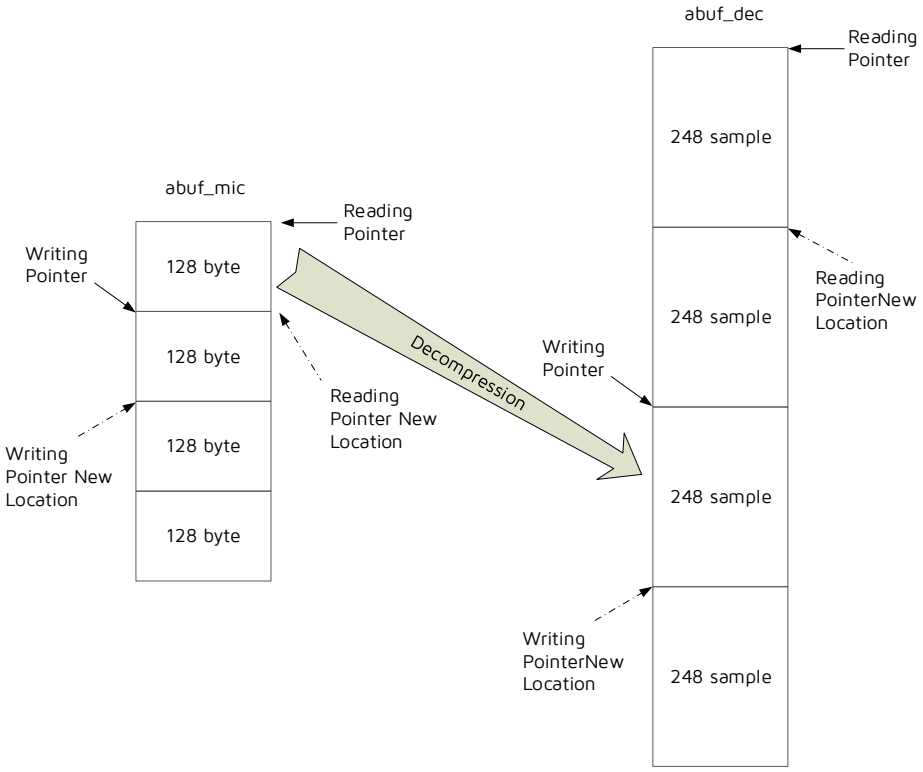
Up to 4 ADPCM data can be cached.

Define the data cache abuf_mic after decompressing ADPCM into pcm:

```
#define DEC_BUFFER_SIZE (MIC_SHORT_DEC_SIZE *
 MIC_SHORT_DEC_SIZE_NUM)
s16         abuf_dec[DEC_BUFFER_SIZE]; //248 * 4 * 2(s16 占 2 个 bytes) = 1984 bytes
```

Up to 4 PCM data can be cached.

Figure 6-1 Data Decompression



The figure above shows the method of data decompression processing.

When the software detects that there is a difference between the abuf_mic write pointer and the read pointer, it starts to call the decompression processing function, extracts 128 bytes of data from the read pointer and compresses it to 248 samples, and moves the read pointer to a new position on the map, indicating that the latest unread data starts at a new location. This cycle goes back and forth.

Similarly, abuf_dec stores the decompressed pcm data, which is also maintained by reading and writing pointers and transmitted to the host via USB.

6.3 Decompression algorithm

The function called by the decompression algorithm are:

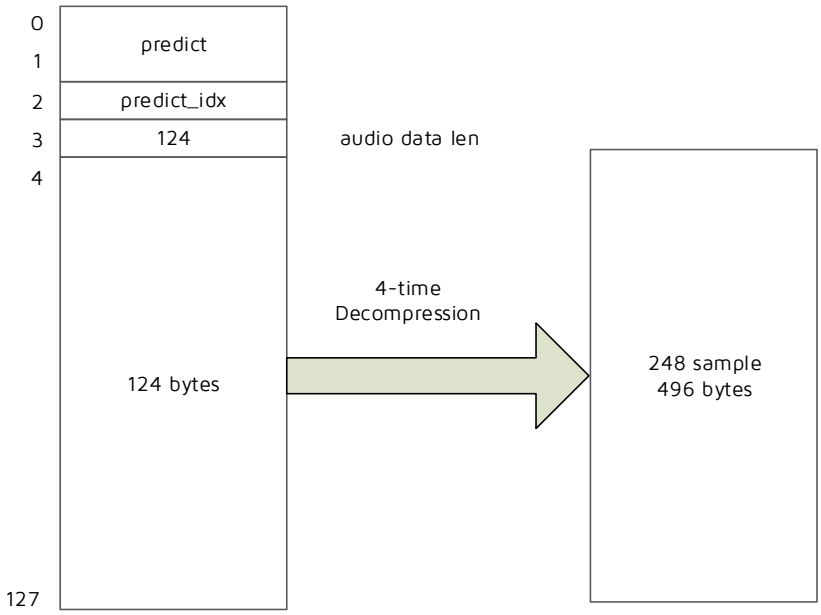
```
void adpcm_to_pcm (signed short *ps, signed short *pd, int len)
```

ps: points to the first address of the data memory before decompression, corresponding to the position of the read pointer of abuf_mic in the data decompression process in Figure 6-1, that is, points to 128 bytes of data in the adpcm format.

pds: points to the first address of the decompressed data memory, corresponding to the position of the abuf_dec write pointer in the data compression process in Figure 6-1, that is, the address that points to the beginning of the 496 bytes pcm format audio data memory restored after decompression.

len: Take MIC_SHORT_DEC_SIZE (248), which means 248 samples.

Figure 6-2 Decompression Algorithm Data



As shown in the figure above: when decompressing, the data read from the first two bytes is predict, the third byte is predict_idx, the fourth is the effective length of audio data 124, and the following 124 bytes are converted to ADPCM data in 496bytes pcm format

7. OTA

For OTA, the entire process is exactly the same as single connection. You can refer to the 825x single connection SDK handbook.

However, there will be a limitation: OTA can only be performed in a slave connected state.

If all are master roles, firmware update can be performed via USB, UART, etc.

If you have a slave role, you can use OTA.

8. Button Scan

Please refer to 825x single connection SDK handbook.

9. LED Management

Please refer to 825x single connection SDK handbook.

10. BLT Software timer

Please refer to 825x single connection SDK handbook.

Please be noted that the software timer cannot be used in m1s1 with low power consumption control. It can be used in other projects, but the current software timer code has not been cleaned up, and several functions have been deleted. These functions have no effect in projects without low-power management and can be removed directly. The user can delete the mentioned functions first, and we will clean up this part in the future.

11. IR

Please refer to 825x single connection SDK handbook.

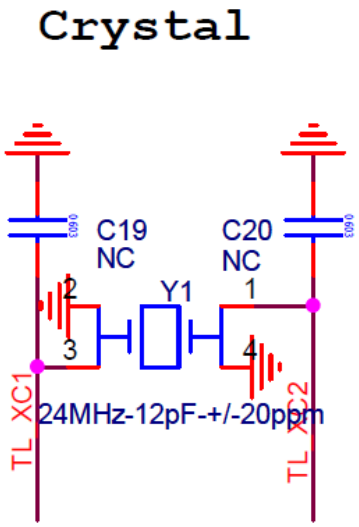
12. Other Modules

12.1 24M crystal external capacitor

Refer to the position C19/C20 of the 24M crystal matching capacitor in the figure below.

By default, the SDK uses 8x5x internal capacitors (that is, caps corresponding to ana_8a<5:0>) as the matching capacitors for 24M crystals. At this time, C19/C20 do not need to be soldered. The advantage of using this solution is that the capacitance can be measured and adjusted on the Telink firmware, so that the frequency value of the final application product can be optimized.

Figure 12-1 24M Crystal Schematics



If you need to use an external soldering capacitor as the matching capacitor of the 24M crystal (C19/C20 soldering capacitor), just call the following API at the beginning of the main function (before the cpu_wakeup_init function):

```
static inline void blc_app_setExternalCrystalCapEnable(u8   en)
{
        blt_miscParam.ext_cap_en = en;
}
```

As long as the API is called before cpu_wakeup_init, the SDK will automatically handle all the settings, including turning off the internal matching capacitor and no longer reading the frequency offset correction value.

12.2 32K clock source selection

The SDK uses the internal 32kRC oscillation circuit of the MCU by default, referred to as 32k RC. The error of 32k RC is relatively large, so for applications with longer suspend or deep retention time, the time accuracy will be worse. At present, the maximum long connection supported by the 32k RC by default cannot exceed 3s (the current SDK also limits the external 32k crystal). Once this time is exceeded, ble_timing will make an

error, resulting in inaccurate packet reception time, which is prone to receive and send packet retry. The consumption increases, and even disconnection occurs.

If users need to achieve lower connection power consumption, including more accurate clock timing in low-power sleep, they can choose to use an external 32k crystal, referred to as 32k Pad, which is currently supported by the SDK.

The user only needs to call one of the following two APIs at the beginning of the main function (before the cpu_wakeup_init function):

```
void blc_pm_select_internal_32k_crystal(void);
void blc_pm_select_external_32k_crystal(void);
```

They are the API for selecting 32k RC and 32k Pad respectively. The SDK calls the 32k RC selected by blc_pm_select_internal_32k_crystal by default. If you need to use 32k Pad, replace it with blc_pm_select_external_32k_crystal.

12.3 PA

If you need to use RF PA, please refer to drivers/8258/rf_pa.c and rf_pa.h.

First open the following macro, which is closed by default.

```
#ifndef PA_ENABLE
#define PA_ENABLE                   0
#endif
```

Call the initialization of PA when the system is initialized.

```
void rf_pa_init(void);
```

Refer to the code implementation. In this initialization, set PA_TXEN_PIN and PA_RXEN_PIN to GPIO output mode, and the initial state is output 0. Users need to define the GPIO 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
```

In addition, the void app_rf_pa_handler(int type) is registered as the callback processing function of the PA. With reference to the implementation of this function, it actually handles the following three PA states: PA off, TX TX on, and RX PA on.

```
    #define PA_TYPE_OFF                 0
#define PA_TYPE_TX_ON               1
#define PA_TYPE_RX_ON               2
```

User only needs to call the above rf_pa_init, app_rf_pa_handler is registered to the underlying callback, BLE will automatically call app_rf_pa_handler for processing in various states.

12.4 PhyTest

PhyTest, or PHY test, refers to the test of RF performance of BLE controller.

For details, please refer to "Core_v5.0" (Vol 2/Part E/7.8.28~7.8.30) and "Core_v5.0" (Vol 6/Part F "Direct Test Mode").

12.4.1 PhyTest API

The source code of PhyTest is encapsulated in the library file, and provides related APIs for users to use. Please refer to the stack/ble/phy/ble_test.h file.

```
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, call blc_phy_initPhyTest_module to set up the PhyTest module.

After the application layer triggers PhyTest, call blc_phy_setPhyTestEnable(1) to start the PhyTest mode.

When the SDK demo "8258_feature_test" is initialized, it directly triggers the start of phytest;

In the SDK demo "8258 ble remote", a key combination is set to trigger. Only when the user presses this group of keys the system will enter PhyTest mode.

PhyTest is a special mode and mutually exclusive with normal BLE function. Once it enters PhyTest mode, advertising and connection are no longer available. Therefore, PhyTest cannot be triggered when the normal BLE function is running.

After PhyTest is finished, either directly power on again, or call blc_phy_setPhyTestEnable(0), then the MCU will automatically reboot.

Use blc_phy_isPhyTestEnable to determine whether the current PhyTest is triggered. You can see that the API is used in the code to achieve low power management. PhyTest mode cannot enter low power consumption.

When PhyTest uses uart two-wire mode (PHYTEST_MODE_THROUGH_2_WIRE_UART), the initialization is as follows:

```
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 implements the analysis and execution of the cmd delivered by the host computer, and phy_test_2_wire_tx_to_uart implements the corresponding results and data feedback to the host computer.

12.4.2 PhyTest demo

Please refer to 825x single connection SDK handbook.

12.5 EMI

12.5.1EMI Test

When testing EMI Test, you need to call rfdrv related interfaces, such as rf_drv_init(), and these operation interfaces are encapsulated in the library. You can see the API declaration in rf_drv.h.

EMI Test has four test modes: carrier only mode (single carrier mode), continue mode (sending mode with data on the carrier, continuous transmission), RX mode, and three TX burst modes (different types of data packet payloads sent). As shown in the following definition:

```
Struct test_list_sate_list[] = {
                {0x01,emicarrieronly},//单载波模式 Single carrier 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 settings

1) Before conducting EMI test, first call rf_drv_init() function to complete rf initialization:

```
void        rf_drv_init (RF_ModeTypeDef rf_mode);
```

The parameter rf_mode is used to select rf mode, but in the 8258 ble SDK, only RF_MODE_BLE_1M is temporarily supported.

2) After setting rf initialization, call app_emi_init() function, which will initialize the host computer interface command.

```
write_reg32(0x408,0x29417671 );//rf access code
      write_reg8(0x840005,tx_cnt);// tx_cnt is initialized to 0
      write_reg8(0x840006,run);// run command 1: start test item, 0: end test item
      write_reg8(0x840007,cmd_now);//cmd: test item settings
      write_reg8(0x840008,power_level);//power_level: send power initialization
      write_reg8(0x840009,chn);//chn: RF channel initialization
      write_reg8(0x84000a,mode);// mode: RF mode initialization,
// Only BLE 1M mode is supported in BLE SDK
      write_reg8(0x840004,0); // 4bytes RSSI statistical average is initialized to 0
write_reg32(0x84000c,0); //4bytes rx packet statistics receiving number is initialized to 0
```

3) App_rf_emi_test_start() is called in main_loop to poll test items.

12.5.1.2 Power level and Channel

During the test, you can configure the rf power level and rf channel to set the packet sending power and packet sending channel.

RF Power: You can set different power values according to RF_PowerTypeDef rf_power_Level_list[60].

RF Channel: The set frequency value is equal to (2400+chn) MHz. (0≤chn≤100)

Among them, when setting the power level, it should be noted that the transmit power is based on the actual value, because the power output by different boards or different antenna matching values will be slightly different. The user can achieve the power setting by calling the following 2 functions:

1. `static void rf_set_power_level_index_singletone (RF_PowerTypeDef level); // Power level Adjust power level in single carrier and continuous packet sending modes`
2. `void rf_set_power_level_index (RF_PowerTypeDef level);// Adjust power level setting in mode tx burst`

The parameter level can be set according to the enumeration type RF_PowerTypeDef.

When setting chn, the range of chn is 0~100. For example, if the user wants to set a 2405MHz channel, set chn to 5. Users can call the following functions:

```
void rf_set_channel (signed char chn, unsigned short set);
```

Among them, the parameter chn can refer to the RF_channel setting, and the parameter set is set to 0.

12.5.1.3 Emi Carrier Only

Carrier mode is EMI Test single carrier transmission mode, users can directly call the emicarrieronly () function, no other settings are required.

```
void emicarrieronly(RF_ModeTypeDef rf_mode, RF_PowerTypeDef pwr,signed char rf_chn)
```

Among them, the parameter rf_mode is RF_MODE_BLE_1M, and the parameters pwr and rf_chn can be set according to the setting method described above.

12.5.1.4 emi_con_prbs9

The continue mode is a data transmission mode with continuous modulation on the EMI Test carrier. The data on the carrier is updated by the rf_continue_mode_loop() function to ensure that the data on the carrier is a series of random numbers.

The user directly calls the emi_con_prbs9 () function to enter the continue mode, no other settings are required.

When setting the continue mode, the emi_con_prbs9 () function will call the rf_emi_tx_continue_setup() function to complete the setting of the continue mode, such as rf_mode, power level, chn, etc. The rf_continue_mode_loop() function is also called to update the data on the carrier.

```
void emi_con_prbs9(RF_ModeTypeDef rf_mode,RF_PowerTypeDef pwr,signed char rf_chn)
```

Among them, the parameter rf_mode, power level and parameter rf_chn can be set according to the previous introduction.

12.5.1.5 Emi TX Burst

Tx Burst mode can send three types of data packets: PRBS9 packet payload, 00001111b packet payload, 10101010b packet payload. Users can select different TX modes through cmd.

The user can directly call one of the functions of emitxprbs9(), emitx55(), emitx0f() to enter TX Burst mode without any other settings.

```
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);
```

Among them, the parameter rf_mode, power level and parameter rf_chn can be set according to the previous introduction.

The emitxprbs9(), emitx55(), emitx0f() functions will call the rf_emi_tx_brust_setup function to complete the tx burst initialization setting. After the TX initialization is completed, the rf_emi_tx_brust_loop() function will be combined to trigger the package sending and update the payload content.

```
void rf_emi_tx_brust_setup(RF_ModeTypeDef rf_mode,unsigned char power_level,signed char rf_chn,unsigned char
pkt_type)
```

Among them, the parameter rf_mode, power level and parameter rf_chn can be set according to the previous introduction. The parameter pkt_type 0 is the packet sending payload PRBS9, 1 is 00001111b, and 2 is 10101010b.

12.5.1.6 EMI RX

Enter rx mode by calling emirx(), call rf_emi_rx_loop() in main_loop() to poll whether RX received data, and count and RSSI statistics of the received RX data.

```
void emirx(RF_ModeTypeDef rf_mode,RF_PowerTypeDef pwr,signed char rf_chn) ;
void rf_emi_rx_loop(void);
```

Among them, the parameters rf_mdoe, pwr and parameter rf_chn can be set with reference to the previous introduction.

12.5.1.7 Upper computer configuration parameter setting

Run:

0	Default	1	Start test

Cmd:

1	CarrierOnly	2	ContinuePRBS9	3	RX
4	TXBurst(PRBS9)	5	TXBurst(0x55)	6	TXBurst(0x0f)

Power and channel have been introduced earlier.

Mode:

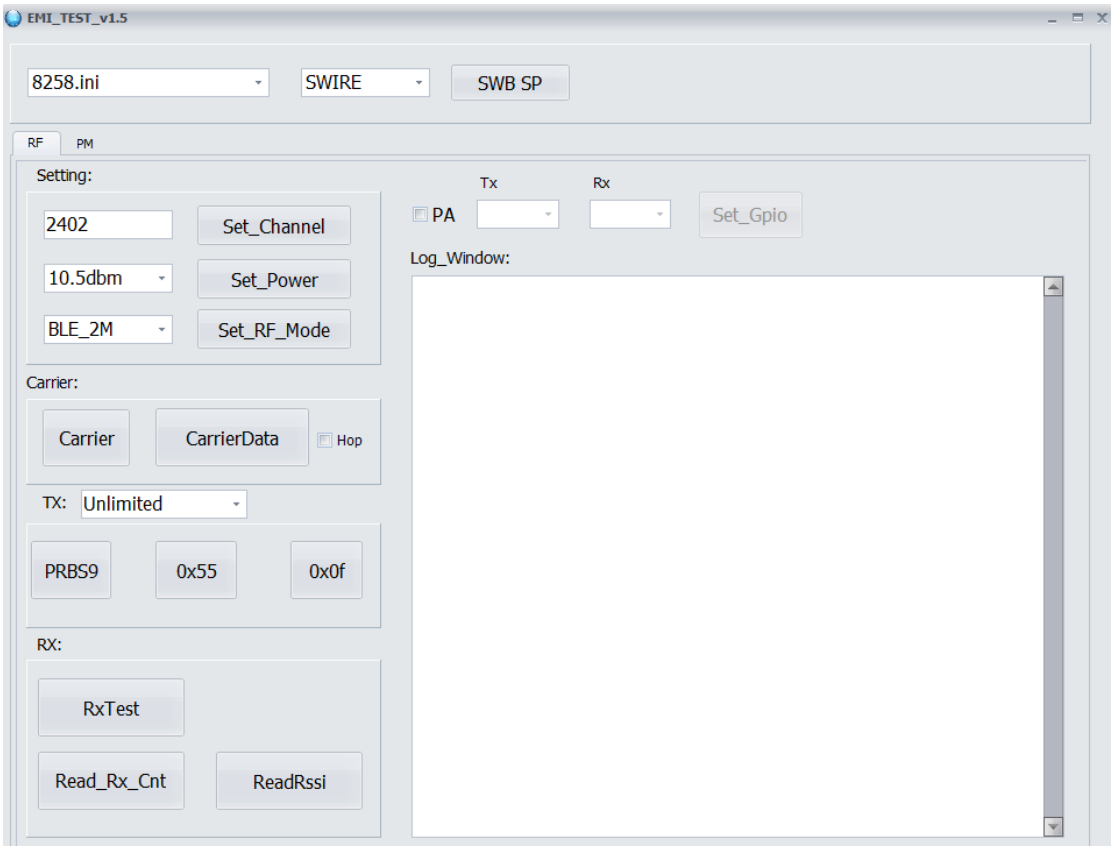
0	Reserve	1	Ble_1M

The default power-on state of these parameters is (mode=1; power=0; channel=2; cmd=1), that is, a single carrier is transmitted with a transmit power of 10.4dbm in ble_1M mode at 2402MHz.

12.5.2 EMI Test Tool

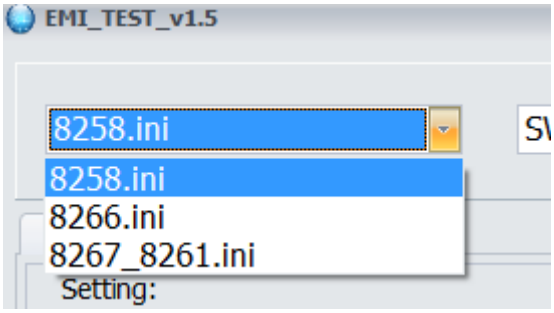
In order to facilitate testing, users can combine the EMI Test Tool tool for EMI testing. The tool interface is shown below:

Figure 12-2 EMI test tool



Step 1 Select the chip model

Figure 12-3 Choose SoC



Step 2 The user can choose the way to connect with the hardware. When selecting Swire, if the system clock is 16MHz or less, you need to use the SWB SP of the WTCDB tool to ensure normal communication.

Figure 12-4 Choose Data Bus

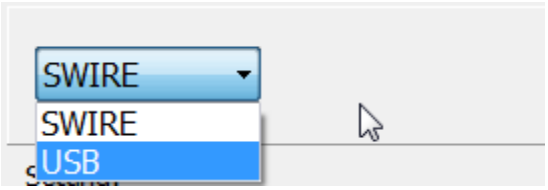
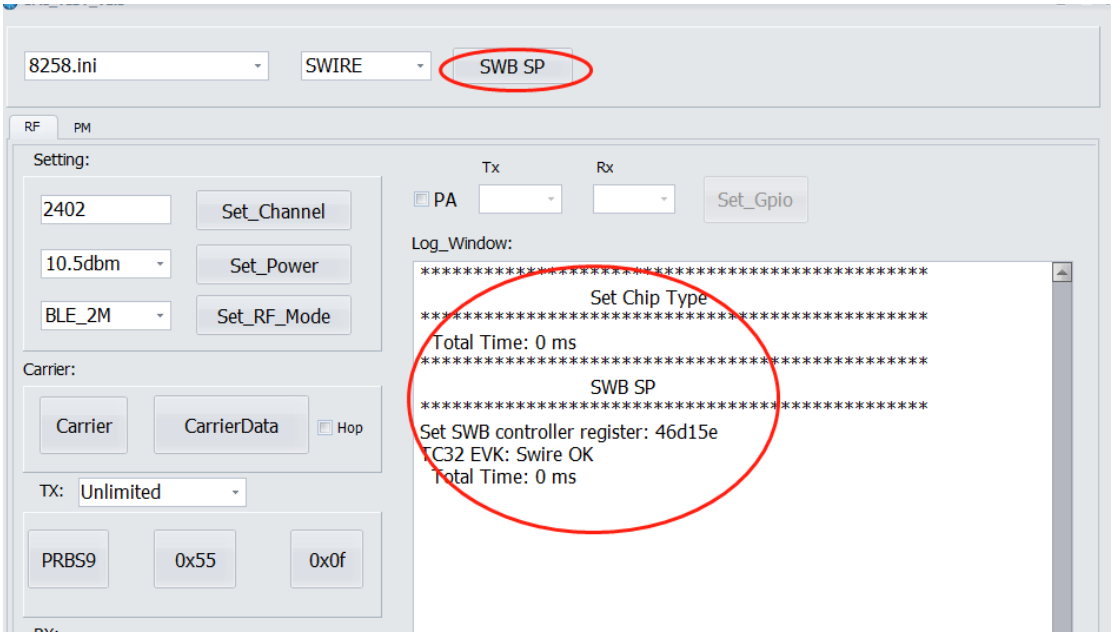
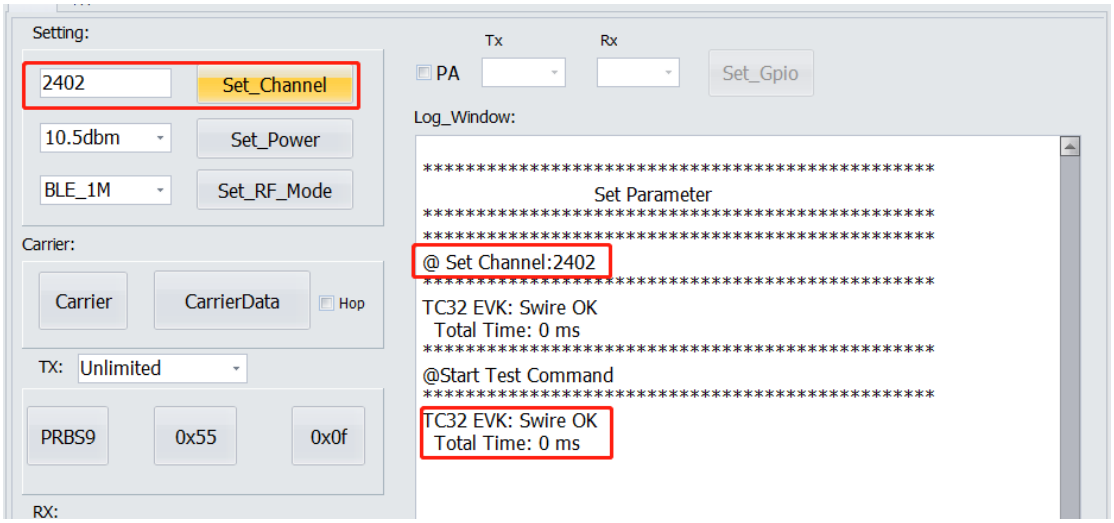


Figure 12-5 Swire SP



Step 3 To set chn, you can enter it directly in the input box, and then click Set_Channel. If the communication is normal, Swire ok will be displayed, as shown in the figure below.

Figure 12-6 Set Channel



Step 4 You can select different power level and ble mode through the drop-down box. After selecting, click the set button on the right ("Set_Power"/"Set_RF_Mode") to complete the setting.

Figure 12-7 Set RF Mode

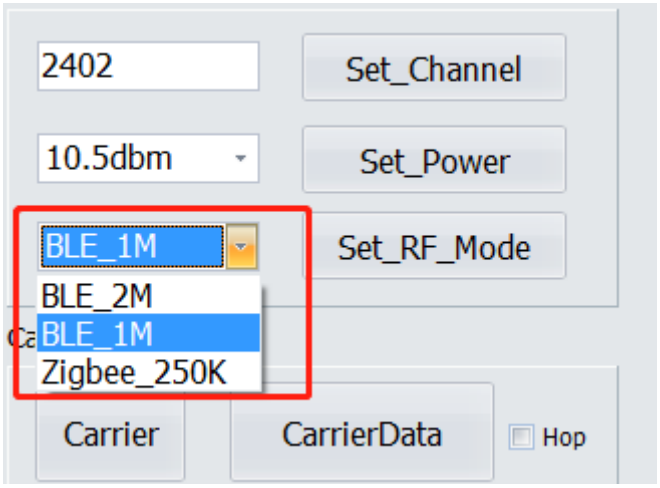
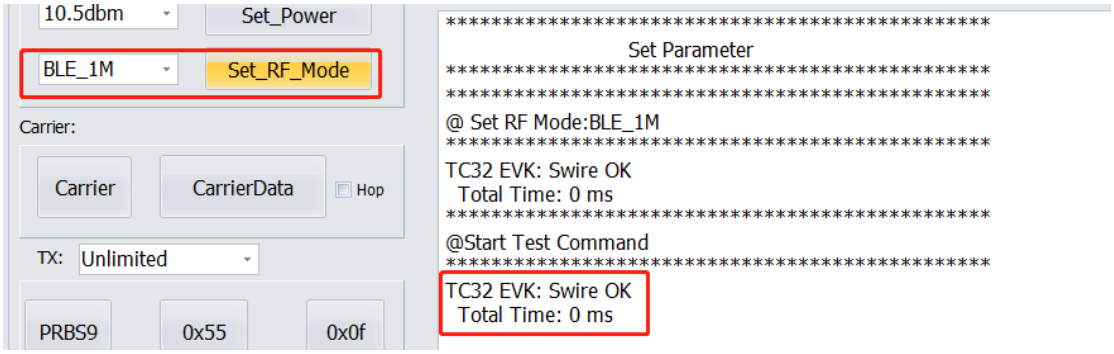
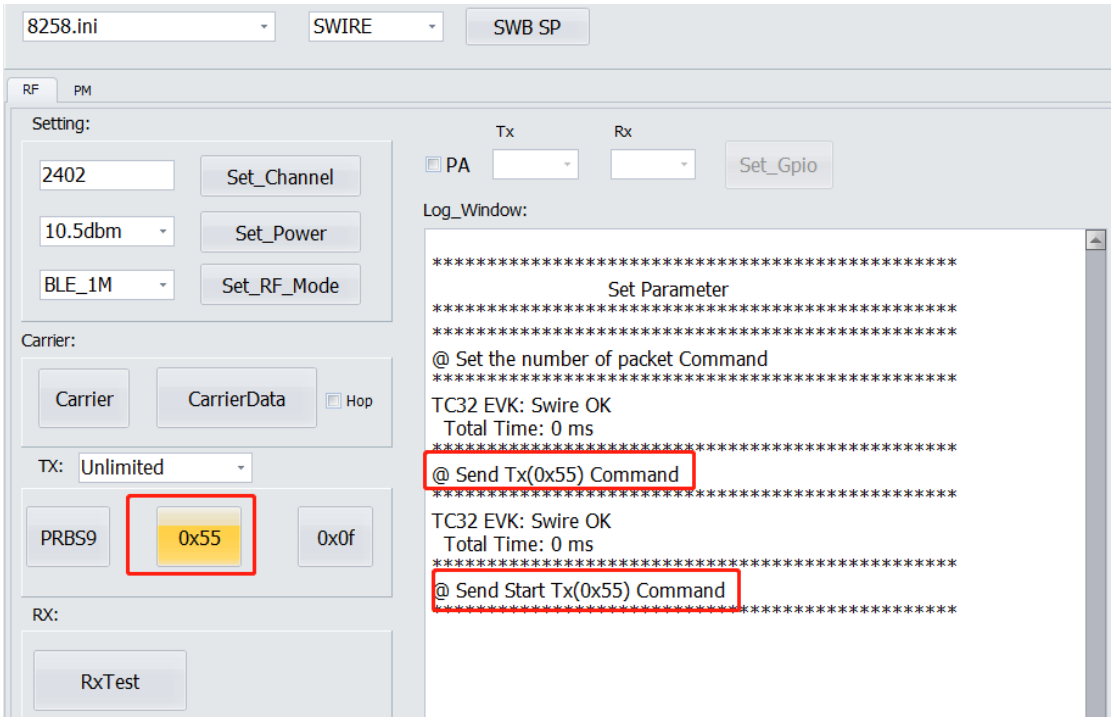


Figure 12-8 Set RF Mode Interface



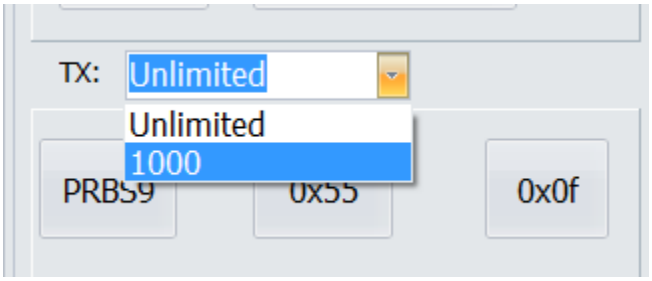
Step 5 Click Carrier, CarrierData, RXTest, PRBS9, 0x55, 0x0f to enter different modes.

Figure 12-9 Set Test Mode



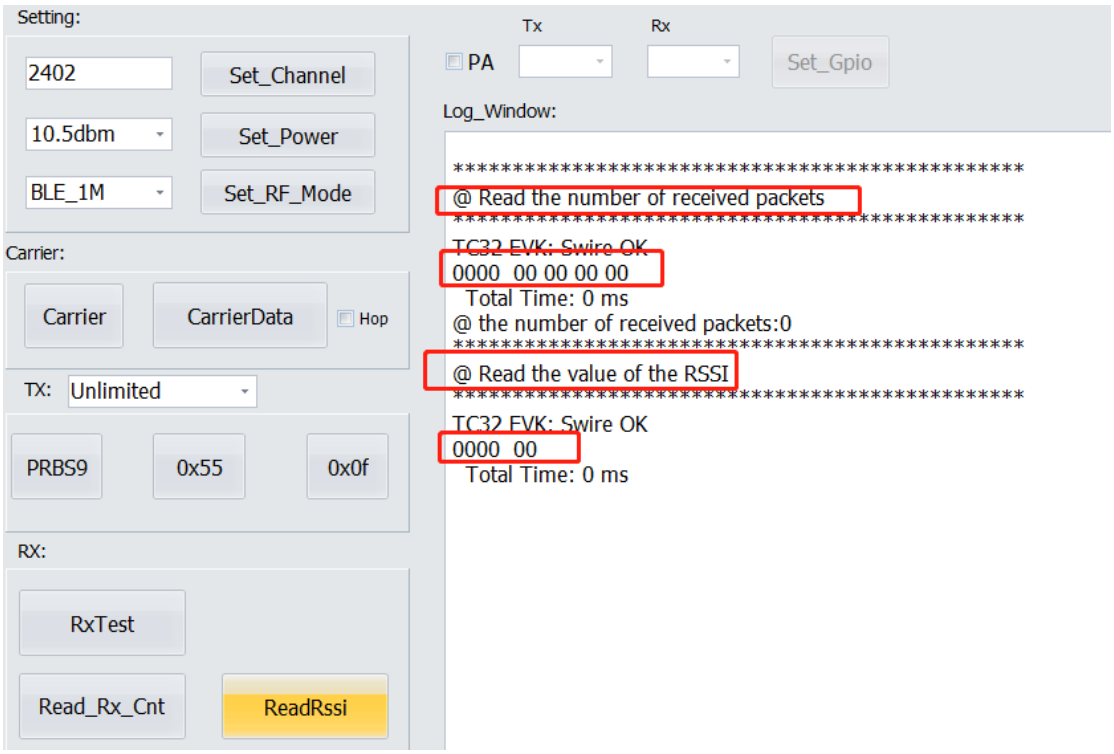
Step 6 In TX mode, you can choose to send 1000 packets or send unlimited packets.

Figure 12-10 Set TX Packet Number



Step 7 In RX mode, you can click Read_Rx_Cnt to read the number of received packets, and click ReadRssi to get the current RSSI, as shown in the following figure.

Figure 12-11 RX Packet Number and RSSI



13. Appendix

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;
}
```