Telink BLE Multiple Connection

SDK Developer Handbook

AN-22063000-E1

Ver1.0.0

2022.06.30

Keyword

Multiple connection, Bluetooth LE, Master/Central, Slave/Perpheral, SMP, PM, Host, Controller, HCI

Brief

This document is a development guide for the Telink BLE Multiple Connection SDK version V4.x.x.x for the B91/ B85m series.

Published by
Telink Semiconductor

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

Legal Disclaimer

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

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:

telinksales@telink-semi.com

telinksupport@telink-semi.com

Revision History

Version	Release Date	Change Description
V1.0.0	2022-06	Initial release

Contents

List of Figures

List of Tables

1 SDK Overview

Telink BLE Multiple Connection SDK provides reference code for BLE multiple connection applications based on the B85m series and B91 series MCUs, and users can develop their own applications on this basis.

The content of the two SDKs is basically the same, and they are called Telink B85m BLE Multiple Connection SDK and Telink B91 BLE Multiple Connection SDK respectively. If there is any inconsistency between the two SDKs during the introduction of this document, we will explain.

Except for the BLE Stack part, other functional modules (such as Flash, Clock, GPIO, IR, etc.) in the Telink BLE Multiple Connection SDK are the same as the Telink BLE Single Connection SDK. In order to avoid repeated introduction of these modules, users should prepare the corresponding Telink BLE Single Connection SDK Handbook at the same time. This document will specify whether the current module is completely consistent with the Telink BLE Single Connection SDK Handbook in the introduction of each module later, and if not, it will also introduce the differences in detail.

The Telink BLE Single Connection SDK Handbook that need to be prepared includes (click the Handbook link to open it):

- Telink B85m BLE Single Connection SDK Handbook
- Telink B91 BLE Signle Connection SDK Handbook

Multiple in Telink BLE Multiple Connection SDK refers to the coexistence of multiple (greater than or equal to 1) Master or Slave roles, for example, it acts as 4 Masters and 3 Slaves at the same time (M4S3 for short).

Note:

> These designations are named after the role played by the local device itself.

1.1 Applicable IC Introduction

Telink BLE Multiple Connection SDK is available for the following MCU models, B85m series (8253/8258/8273/8278) and B91 series. The Flash size and SRAM size are different between them, as shown below. Please select the corresponding boot files when using different MCUs (boot files are explained in section 1.3).

Table 1.1: MCU resources

MCU	Flash size	SRAM size
8253	512 KB	48 KB
8258	512 KB / 1 MB	64 KB
8273	512 KB	64 KB
8278	1 MB	64 KB
B91	1 MB / 2 MB	128 KB + 128 KB

1.2 Software Architecture

Telink BLE Multiple Connection SDK software architecture includes two parts: application layer and BLE stack.

1.2.1 File Structure

After importing the Telink BLE Multiple Connection SDK in the IDE, the displayed file structure is shown in the figure below (take Telink B85m BLE Multiple Connection SDK as an example). There are 8 toplayer folders: algorithm, application, boot, common, drivers, proj_lib, stack and vendor.

> Officially recommended IDE: B85m series use Telink IDE, B91 series use Andes Telink_DRS.

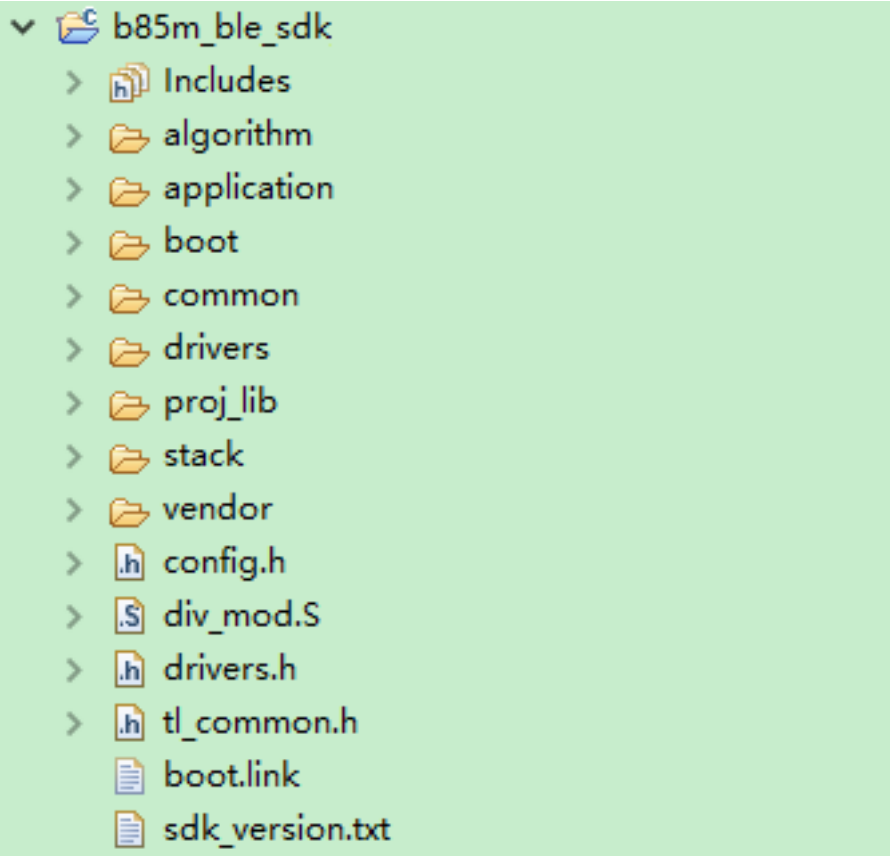


Figure 1.1: SDK file structure

algorithm: This folder contains some general algorithms, such as aes_ccm. The C files corresponding to most algorithms are supplied in the form of library files, leaving only the corresponding header files.

application: This folder contains general application program, e.g. print, keyboard, and etc.

boot: This folder contains software bootloader for MCU, i.e., assembly code after MCU power on or deep-sleep 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 MCU peripheral drivers, e.g. Clock, Flash, I2C, USB, GPIO, UART.

proj_lib: This folder contains library files necessary for SDK running. BLE stack, RF driver, PM driver, etc. are supplied in the form of library files.

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 demo code or user's own code.

Telink B85m BLE Multiple Connection SDK provides 6 demos: b85m_demo, b85m_controller, b85m_feature, b85m_m1s1, b85m_master_dongle, and b85m_slave. These demo projects are in the vendor folder, as shown in the following figure.

> b85m_m1s1 and b85m_slave support suspend mode and deepsleep retention mode, other demos support suspend mode but not deepsleep retention mode.

Figure 1.2: Demo project of B85m multiple connection SDK

Telink B91 BLE Multiple Connection SDK provides 5 demos: B91_demo, B91_controller, B91_feature, B91_master_dongle, B91_slave. These demo projects are in the vendor folder, as shown in the following figure.

> The default configuration of B91_slave supports suspend mode and deepsleep retention mode, and the default configuration of other demos supports suspend mode. If you want to support deepsleep retention mode, please refer to B91_slave.

Figure 1.3: Demo project of B91 multiple connection SDK

Taking b85m_demo as an example to explain the demo file structure, the file composition is shown in the following figure:

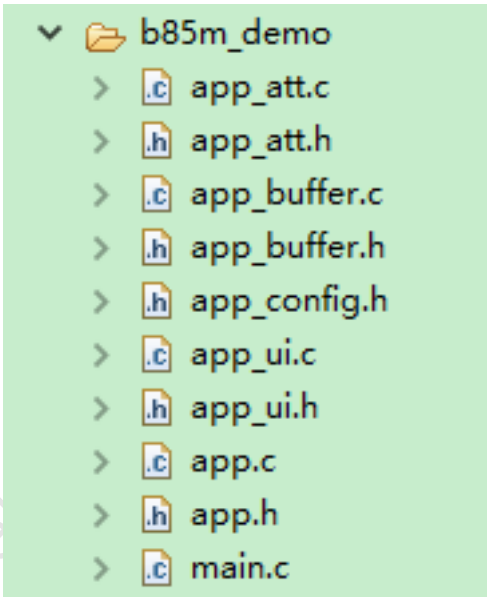


Figure 1.4: B85m_demo file structure

1.2.2 main.c

The main.c file contains the main function and the interrupt handler. The main function is the entry point of program execution, and contains the configuration requied for the system to work properly. It is recommended that users do not make any modifications to it. The interrupt handler is the entry function when the system triggers an interrupt.

```
_attribute_ram_code_ int main(void)
{
    #if (BLE_APP_PM_ENABLE)
        blc_pm_select_internal_32k_crystal();
    #endif
```

```
//MCU's most basic hardware initialization
    #if (MCU_CORE_TYPE == MCU_CORE_825x)
        cpu_wakeup_init();
    #elif (MCU_CORE_TYPE == MCU_CORE_827x)
        cpu_wakeup_init(DCDC_MODE, EXTERNAL_XTAL_24M);
    #endif

    /* detect if MCU is wake_up from deep retention mode */
    int deepRetWakeUp = pm_is_MCU_deepRetentionWakeup();  //MCU deep retention wakeUp

    clock_init(SYS_CLK_TYPE); //clock initialization

    rf_drv_init(RF_MODE_BLE_1M); //RF initialization

    gpio_init(!deepRetWakeUp); //GPIO initialization

    if( deepRetWakeUp ){ //MCU wake_up from deepSleep retention mode
        user_init_deepRetn ();
    }
    else{ //MCU power_on or wake_up from deepSleep mode
        user_init_normal ();
    }

    /* load customized freq_offset cap value.*/
    blc_app_loadCustomizedParameters();

    irq_enable();

    while(1)
    {
        main_loop (); //Including BLE sending and receiving packet processing and user tasks
    }
    return 0;
}
```

1.2.3 app_config.h

The user configuration file "app_config.h" servers to configure parameters of the whole system (e.g. BLE parameters, GPIO configuration, low power enable/disable, encryption enable/disable, etc.).

Parameter details of each module will be illustrated in the following sections.

1.2.4 Application File

app.c: User main file for BLE protocol stack initialization, data processing and low power management.

app_att.c: This file provides GATT service table and profile file. GATT sevice table has provided the standard GATT service, standard GAP service, standard HID sevice and some private service. Users can refer to these to add their own service and profile.

app_ui.c: This file provides keystroke function.

app_buffer.c: This file is used to define the buffers used by each layer of the stack, such as: LinkLayer TX & RX buffer, L2CAP layer MTU TX & RX buffer, HCI TX & RX buffer, etc.

1.2.5 Common File

The path is vendor/common.

blt_soft_timer.c: This file provides an implementation of the software timer.

custom_pair.c: This file provides a set of pair solution customized by Telink.

device_manage.c: This file is mainly for connection device information management (e.g., connection handle, attribute handle, BLE device address, address type, etc.). This information is used when developing applications.

simple_sdp.c: This file provides a simple SDP (Service Discovery Protocol) implementation of the Master role.

1.2.6 BLE Stack Entry

The BLE interrupt handler entry function is blc_sdk_irq_handler().

The BLE logic and data handling entry function is blc_sdk_main_loop(), which is responsible for handling data and events related to the BLE stack.

1.3 Software Bootloader Introduction

In Telink BLE Multiple Connection SDK, different MCU models correspond to different bootloader files. The software bootloader files are stored in the boot folder. Telink's bootloader file is composed of two parts, link file and cstartup.S assembly file.

1.3.1 Bootloader File for B85m Multiple Connection

The bootloader file of Telink B85m BLE Multiple Connection SDK is shown in the figure below.

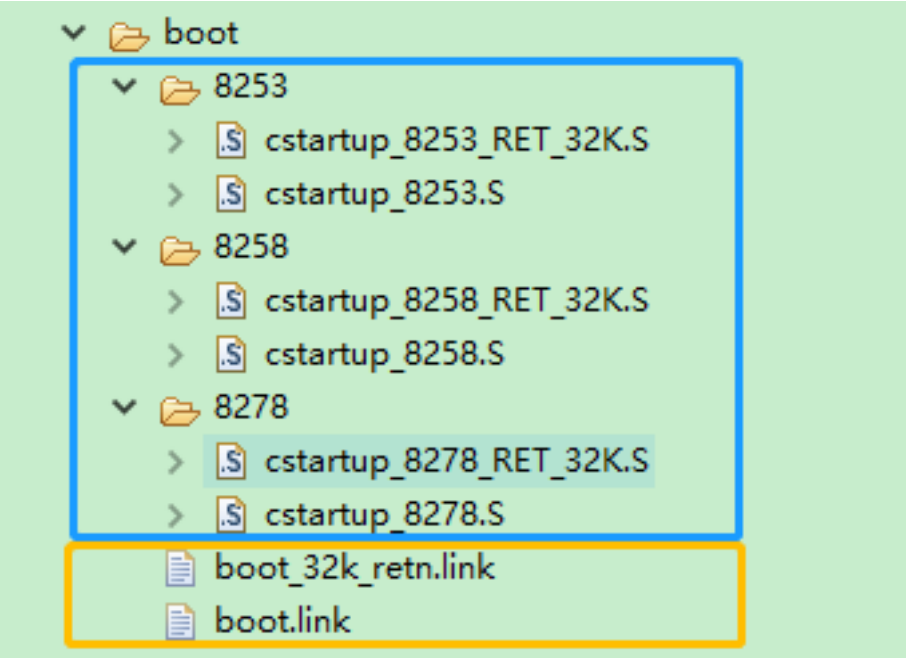


Figure 1.5: Bootloader file for B85m multiple connection SDK

There are two link files in Telink B85m BLE Multiple Connection SDK, boot.link (supports suspend mode) and boot_32k_retn.link (supports suspend mode and deepsleep retention mode).

Note:

- When compiling the project, the link file that actually takes effect is b85m_ble_sdk/boot.link.
- Only b85m_slave and b85m_m1s1 support deepsleep retention mode. When using these two demos, you need to copy the content of b85m_ble_sdk/boot/boot_32k_retn.link to b85m_ble_sdk/boot.link.
- When using other demos, you need to copy the content of b85m_ble_sdk/boot/boot.link to b85m_ble_sdk/boot.link.

The cstartup.S file in Telink B85m BLE Multiple Connection SDK varies depending on the size of the chip's SRAM resources. Each MCU corresponds to 2 cstartup.S files, "RET_32K" means it supports deepsleep retention 32K mode. Since the SRAM of 8273 and 8278 are both 64 KB, their cstartup.S files are exactly the same, and the configuration of 8278 is used uniformly.

Taking b85m_demo, MCU selects 8258 as an example, the method of setting the corresponding cstartup.S is as follows:

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 project properties window as shown in the figure below.

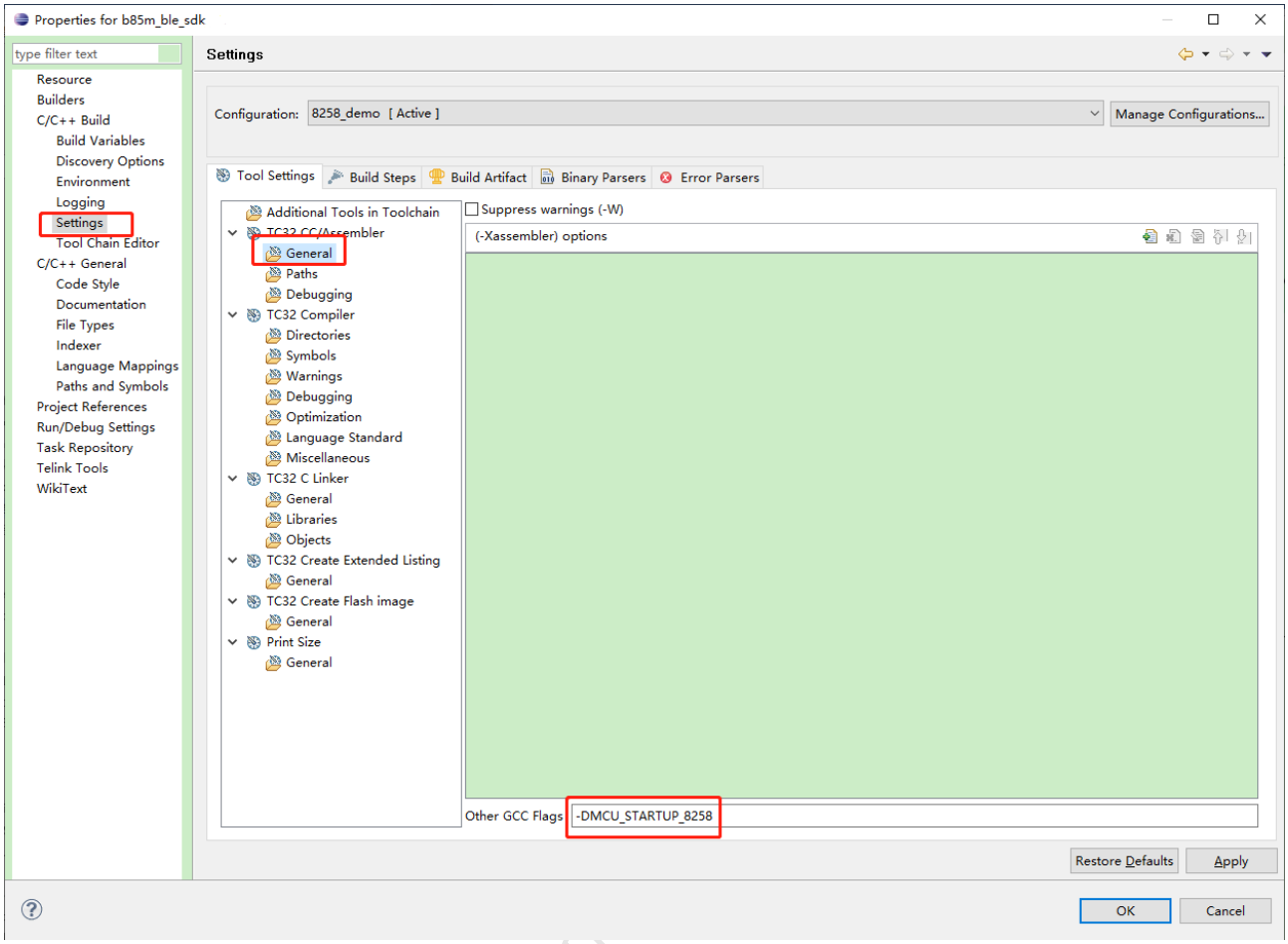


Figure 1.6: Cstartup selection

The relationship of bootloader file selection for different MCUs of B85m series is shown in the following table.

Table 1.2: B85m series MCU bootloader file selection

MCU	b85m_demo, b85m_controller, b85m_feature, b85m_master_dongle	b85m_m1s1、b85m_slave
8253	boot.link cstartup_8253.S	boot_32k_retn.link cstartup_8253_RET_32K.S
8258	boot.link cstartup_8258.S	boot_32k_retn.link cstartup_8258_RET_32K.S
8273/8278	boot.link cstartup_8278.S	boot_32k_retn.link cstartup_8278_RET_32K.S

1.3.2 Bootloader File for B91 Multiple Connection

The bootloader file for Telink B91 BLE Multiple Connection SDK is shown below.

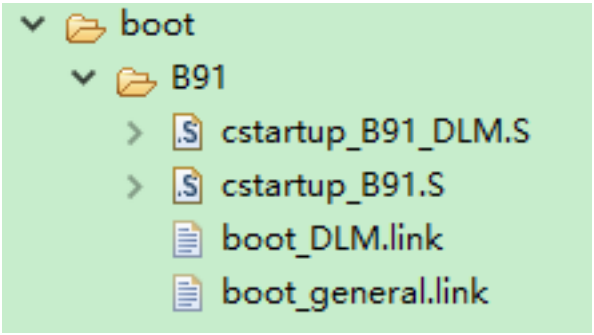


Figure 1.7: Bootloader file for B91 multiple connection SDK

The cstartup_B91.S and boot_general.link files are run by default. At this time, the SDK will occupy the ISRAM and D-SRAM space. I-SRAM includes retention_reset, aes_data, retention_data, ramcode, and unused I-SRAM area. D-SRAM includes data, sbss, bss, heap, unused D-SRAM area and stack.

If you want to leave all the 128K D-SRAM space for users, you need to make the following changes.

(1) Change the #if 1 at the beginning of the cstartup_B91.S file to #if 0.

(2) Change #if 0 at the beginning of the cstartup_B91_DLM.S file to #if 1.

(3) Copy the content in eagle_ble_sdk/boot/B91/boot_DLM.link to eagle_ble_sdk/boot.link.

Note:

> Telink B91 BLE Multiple Connection SDK is configured with deepsleep retention 64K mode by default.

1.4 Library

1.4.1 B85m Multiple Connection Library

The library corresponding to the Telink B85m BLE Multiple Connection SDK configuration project is shown in the figure below.

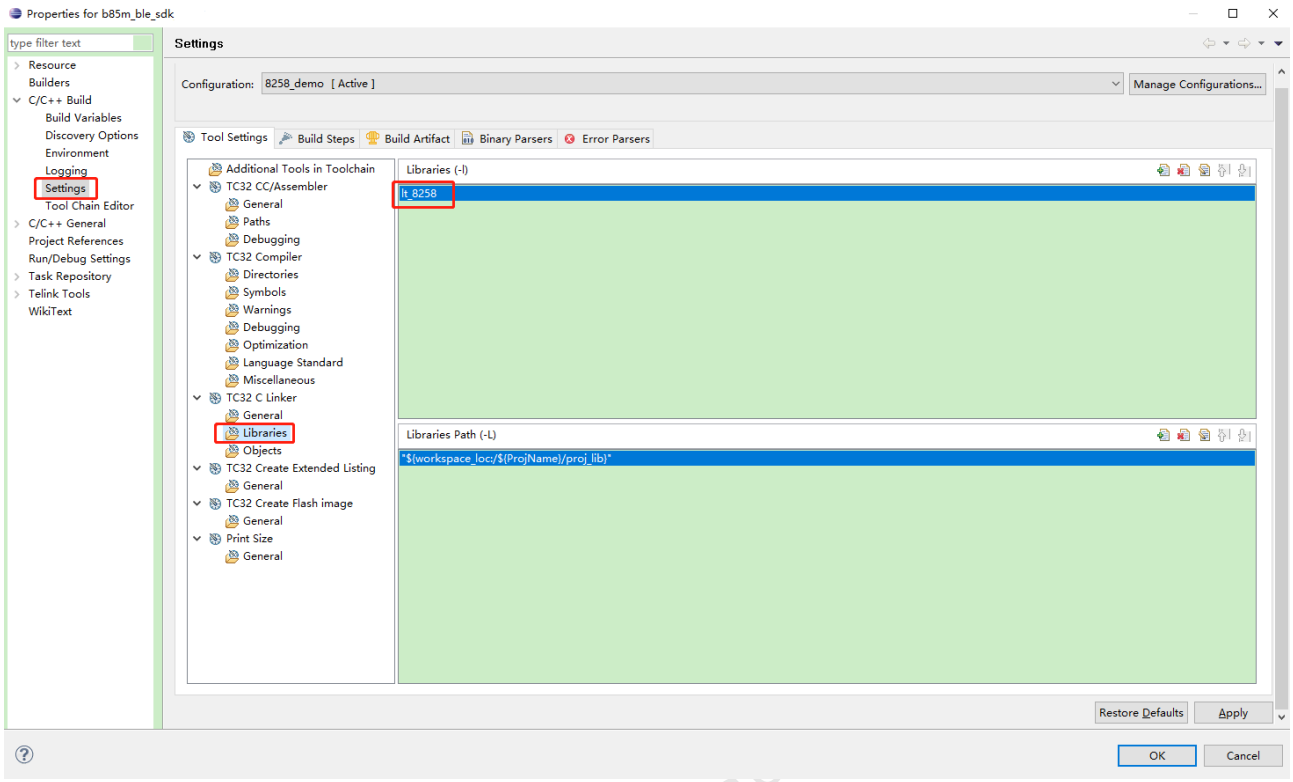


Figure 1.8: B85m multiple connection SDK selects the project corresponding library

The following picture shows the library currently provided in the proj_lib folder of the Telink B85m BLE Multiple Connection SDK.

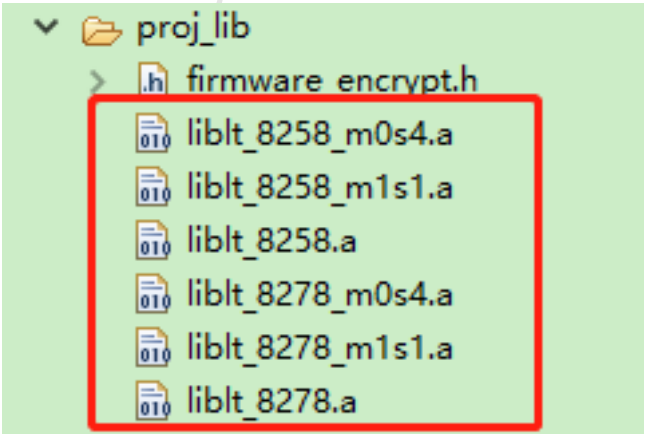


Figure 1.9: B85m multiple connection SDK library

The following table shows the adaptation relationship between different MCUs, demos and library of Telink B85m BLE Multiple Connection SDK.

Table 1.3: B85m multiple connection SDK different MCU, demo and library adaptation relationship

MCU	b85m_demo, b85m_controller, b85m_feature, b85m_master_dongle	b85m_m1s1	b85m_slave
8253	liblt_8258.a	liblt_8258_m1s1.a	liblt_8258_m0s4.a
8258	liblt_8258.a	liblt_8258_m1s1.a	liblt_8258_m0s4.a
8273	liblt_8278.a	liblt_8278_m1s1.a	liblt_8278_m0s4.a
8278	liblt_8278.a	liblt_8278_m1s1.a	liblt_8278_m0s4.a

1.4.2 B91 Multiple Connection Library

The following picture shows the library currently provided in the proj_lib folder of the Telink B91 BLE Multiple Connection SDK.

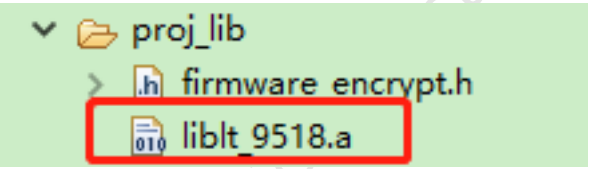


Figure 1.10: B91 multiple connection SDK library

All demos of Telink B91 BLE Multiple Connection SDK use the same library.

2 MCU Basic Modules

2.1 MCU Address Space

2.1.1 MCU Address Space Allocation

Please refer to the introduction in the corresponding chapter of Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x) and B91 series.

2.1.2 SRAM Space Allocation

Please refer to the introduction in the corresponding chapter of Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x) and B91 series.

For Telink B85m BLE Multiple Connection SDK, only b85m_m1s1 and b85m_slave support the deepsleep retention. Please refer to Telink B85m BLE Single Connection SDK for the SRAM space allocation of b85m_m1s1 and b85m_slave. The rest of the demos only use the suspend and deepsleep function, the SRAM space allocation that does not support deepsleep retention is as shown below.

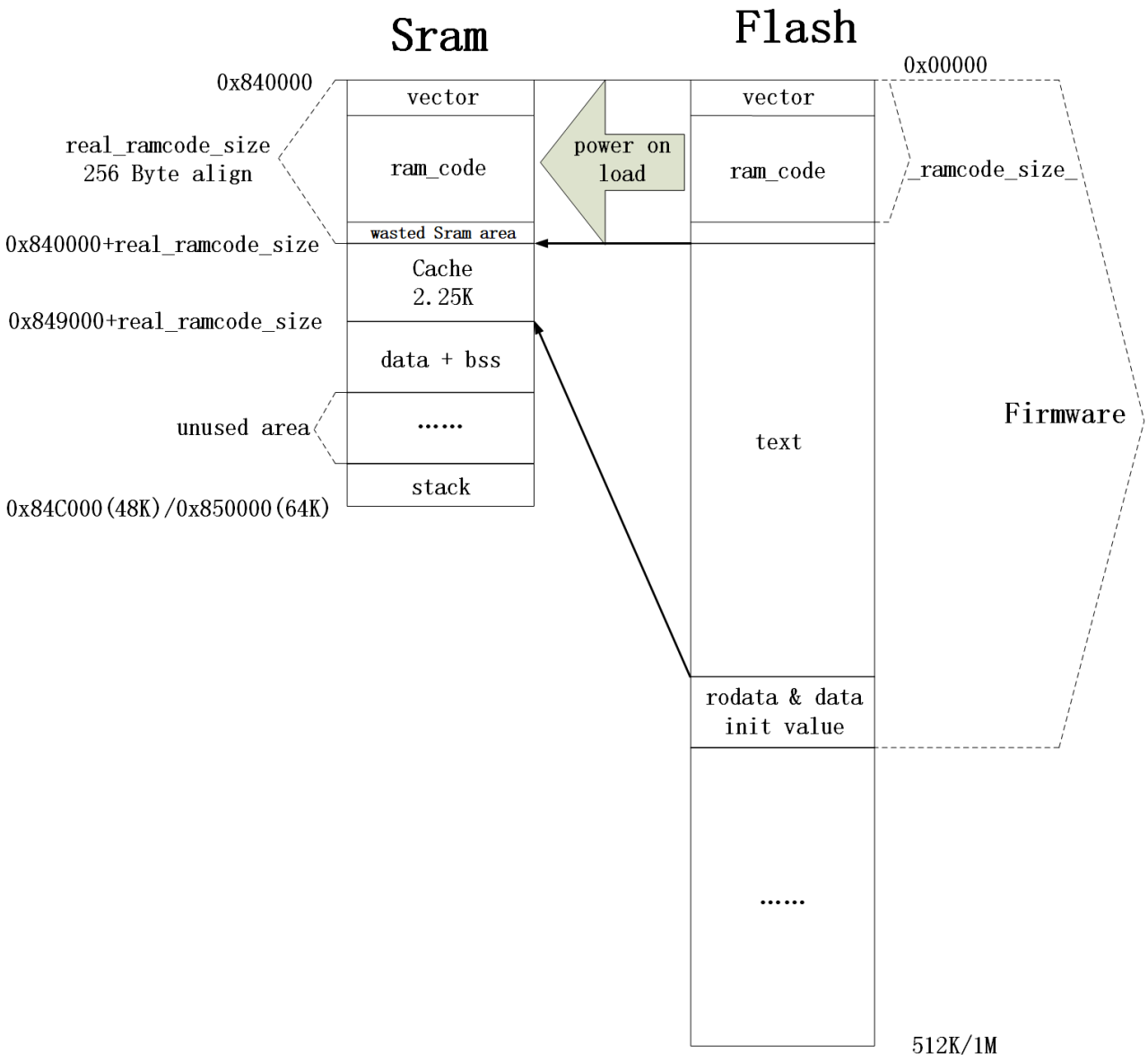


Figure 2.1: SRAM and firmware space allocation not support deepSleep retention

Firmware in Flash includes vector, ramcode, text, rodata and data initial value.

SRAM includes vector, ramcode, Cache, data, bss, unused sram area and stack.

The vector/ramcode in SRAM is a copy of the vector/ramcode in Flash.

2.1.3 MCU Address Space Access

Please refer to the introduction in the corresponding chapter of Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x) and B91 series.

2.1.4 SDK Flash Space Allocation

Multiple connection uses more flash storage areas due to the relatively complex functions, and the method of automatically reading the Flash size for Flash configuration is not adopted, but require users to manually configure Flash size according to the actual situation of their products. The app_config.h in all demos only reserves the default configuration, **Users must check to confirm or even modify the definition of "FLASH_SIZE_CONFIG"**.

```
#define FLASH_SIZE_512K                 0x80000
#define FLASH_SIZE_1M                   0x100000
#define FLASH_SIZE_2M                   0x200000

#define FLASH_SIZE_CONFIG               FLASH_SIZE_1M
```

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

There are four types of information that need to be stored in Flash in Telink BLE Multiple Connection SDK, namely MAC, calibration information, encrypted pairing information and SDP information. These parameters are allocated different Flash space in the SDK by default.

The Flash space for MAC and calibration information storage varies with size of the chip Flash. By default, for the 512K Flash B85m series (8253/8258/8273) chips, the MAC is stored in the 4K Flash space starting at 0x76000 and the calibration information is stored in the 4K Flash space starting at 0x77000. For the 512K Flash B91 series chips, the MAC is stored in the 4K Flash space starting at 0x7F000 and the calibration information is stored in the 4K Flash space starting at 0x7E000. For the 1M Flash chips (8258/8278/B91 series), 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. For the 2M Flash B91 series chips, the MAC is stored in the 4K Flash space starting at 0x1FF000 and the calibration information is stored in the 4K Flash space starting at 0x1FE000.

```
/**
 * @brief   512 K Flash MAC address and calibration data area
 */
#if (MCU_CORE_TYPE == MCU_CORE_9518)
    #define CFG_ADR_MAC_512K_FLASH                  0x7F000
    #define CFG_ADR_CALIBRATION_512K_FLASH          0x7E000
#else
    #define CFG_ADR_MAC_512K_FLASH                  0x76000
    #define CFG_ADR_CALIBRATION_512K_FLASH          0x77000
#endif

/**
 * @brief   1 M Flash MAC address and calibration data area
 */
#define CFG_ADR_MAC_1M_FLASH                        0xFF000
#define CFG_ADR_CALIBRATION_1M_FLASH                0xFE000

/**
 * @brief   2 M Flash MAC address and calibration data area
 */
#define CFG_ADR_MAC_2M_FLASH                        0x1FF000
#define CFG_ADR_CALIBRATION_2M_FLASH                0x1FE000
```

Figure 2.2: Flash space for MAC and calibration information storage

The encrypted pairing information and SDP information are also stored in a separate Flash space, and the SDK can automatically configure the corresponding default location according to the Flash size set by users. By default, for the 512K Flash chips, the encrypted pairing information is stored in the 16K Flash space starting at 0x78000, the custom binding ionformation is stored in the 4K Flash space starting at 0x7C000 and the SDP information is stored in the 8K Flash space starting at 0x7D000. For the 1M Flash chips, the encrypted pairing information is stored in the 16K Flash space starting at 0xFA000, the custom binding ionformation is stored in the 4K Flash space starting at 0xF8000 and the SDP information is stored in the 8K Flash space starting at 0xF6000. For the 2M Flash chips, the encrypted pairing information is stored in the 16K Flash space starting at 0x1FA000, the custom binding ionformation is stored in the 4K Flash space starting at 0x1F8000 and the SDP information is stored in the 8K Flash space starting at 0x1F6000.

The SDK can automatically configure the corresponding MAC and calibration value storage space according to the Flash size set by users. Users can also modify the corresponding macros in vendor/common/blt_common.h to modify the default MAC and calibration storage space according to their needs, and it should be noted that the burn-in address on the telink mass production tool should be modified accordingly.

```
#if(FLASH_SIZE_CONFIG == FLASH_SIZE_512K)

    /* MAC and calibration data area */
    #define     CFG_ADR_MAC                                 CFG_ADR_MAC_512K_FLASH          //can not change this value
    #define     CFG_ADR_CALIBRATION                         CFG_ADR_CALIBRATION_512K_FLASH  //can not change this value

    #if (MCU_CORE_TYPE == MCU_CORE_9518)
        //TODO
    #else
        /* SMP paring and key information area */
        #ifndef FLASH_ADR_SMP_PAIRING
        #define FLASH_ADR_SMP_PAIRING                       0x78000
        #endif

        #ifndef FLASH_SMP_PAIRING_MAX_SIZE
        #define FLASH_SMP_PAIRING_MAX_SIZE                  (2*4096)  //normal 8K + backup 8K = 16K
        #endif


        /* bonding slave information for custom pair area */
        #ifndef FLASH_ADR_CUSTOM_PAIRING
        #define FLASH_ADR_CUSTOM_PAIRING                    0x7C000
        #endif

        #ifndef FLASH_CUSTOM_PAIRING_MAX_SIZE
        #define FLASH_CUSTOM_PAIRING_MAX_SIZE               4096
        #endif


        /* bonding slave GATT service critical information area */
        #ifndef FLASH_SDP_ATT_ADRRESS
        #define FLASH_SDP_ATT_ADRRESS                       0x7D000    //for master: store peer slave device's ATT handle
        #endif

        #ifndef FLASH_SDP_ATT_MAX_SIZE
        #define FLASH_SDP_ATT_MAX_SIZE                      (2*4096)   //8K flash for ATT HANLDE storage
        #endif
    #endif
```

Figure 2.3: 512K Flash default Flash storage address

```
#elif(FLASH_SIZE_CONFIG == FLASH_SIZE_1M)

    /* MAC and calibration data area */
    #define CFG_ADR_MAC                                 CFG_ADR_MAC_1M_FLASH            //can not change this value
    #define CFG_ADR_CALIBRATION                         CFG_ADR_CALIBRATION_1M_FLASH    //can not change this value

    /* SMP paring and key information area */
    #ifndef FLASH_ADR_SMP_PAIRING
    #define FLASH_ADR_SMP_PAIRING                       0xFA000
    #endif

    #ifndef FLASH_SMP_PAIRING_MAX_SIZE
    #define FLASH_SMP_PAIRING_MAX_SIZE                  (2*4096)  //normal 8K + backup 8K = 16K
    #endif


    /* bonding slave information for custom pair area */
    #ifndef FLASH_ADR_CUSTOM_PAIRING
    #define FLASH_ADR_CUSTOM_PAIRING                    0xF8000
    #endif

    #ifndef FLASH_CUSTOM_PAIRING_MAX_SIZE
    #define FLASH_CUSTOM_PAIRING_MAX_SIZE               4096
    #endif


    /* bonding slave GATT service critical information area */
    #ifndef FLASH_SDP_ATT_ADRRESS
    #define FLASH_SDP_ATT_ADRRESS                       0xF6000    //for master: store peer slave device's ATT handle
    #endif

    #ifndef FLASH_SDP_ATT_MAX_SIZE
    #define FLASH_SDP_ATT_MAX_SIZE                      (2*4096)   //8K flash for ATT HANLDE storage
    #endif
```

Figure 2.4: 1M Flash default Flash storage address

```
#elif(FLASH_SIZE_CONFIG == FLASH_SIZE_2M)
/* MAC and calibration data area */
    #define CFG_ADR_MAC                                 CFG_ADR_MAC_2M_FLASH            //can not change this value
    #define CFG_ADR_CALIBRATION                         CFG_ADR_CALIBRATION_2M_FLASH    //can not change this value

    /* SMP paring and key information area */
    #ifndef FLASH_ADR_SMP_PAIRING
    #define FLASH_ADR_SMP_PAIRING                       0x1FA000
    #endif

    #ifndef FLASH_SMP_PAIRING_MAX_SIZE
    #define FLASH_SMP_PAIRING_MAX_SIZE                  (2*4096)  //normal 8K + backup 8K = 16K
    #endif


    /* bonding slave information for custom pair area */
    #ifndef FLASH_ADR_CUSTOM_PAIRING
    #define FLASH_ADR_CUSTOM_PAIRING                    0x1F8000
    #endif

    #ifndef FLASH_CUSTOM_PAIRING_MAX_SIZE
    #define FLASH_CUSTOM_PAIRING_MAX_SIZE               4096
    #endif


    /* bonding slave GATT service critical information area */
    #ifndef FLASH_SDP_ATT_ADRRESS
    #define FLASH_SDP_ATT_ADRRESS                       0x1F6000    //for master: store peer slave device's ATT handle
    #endif

    #ifndef FLASH_SDP_ATT_MAX_SIZE
    #define FLASH_SDP_ATT_MAX_SIZE                      (2*4096)   //8K flash for ATT HANLDE storage
    #endif
#endif
```

Figure 2.5: 2M Flash default Flash storage address

2.2 Clock Module

Please refer to the introduction in the corresponding chapter of Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x) and B91 series.

The difference between the multiple connection SDK and the single connection SDK is that the multiple connection SDK can only use two system clocks, CLK_32M and CLK_48M, and the other clocks are too slow to meet the operation of the multiple connection SDK.

2.3 GPIO Module

Please refer to the introduction in the corresponding chapter of Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x).

2.4 Interrupt Nesting

Telink B85m BLE Multiple Connection SDK does not support interrupt nesting. All interrupts have a uniform priority and are responded to on a first-come, first-served basis. please refer to the introduction in the corresponding chapter of Telink B85m Single Connection SDK Handbook for details.

Telink B91 BLE Multiple Connection SDK supports interrupt nesting. please refer to the introduction in the corresponding chapter of Telink B91 BLE Single Connection SDK Handbook for details.

3 BLE Module

This handbook refers to Bluetooth Core Specification version 5.3.

3.1 BLE SDK Software Architecture

3.1.1 Standard BLE SDK Software Architecture

According to Bluetooth Core Specification, a standard BLE SDK architecture is shown in the figure below.

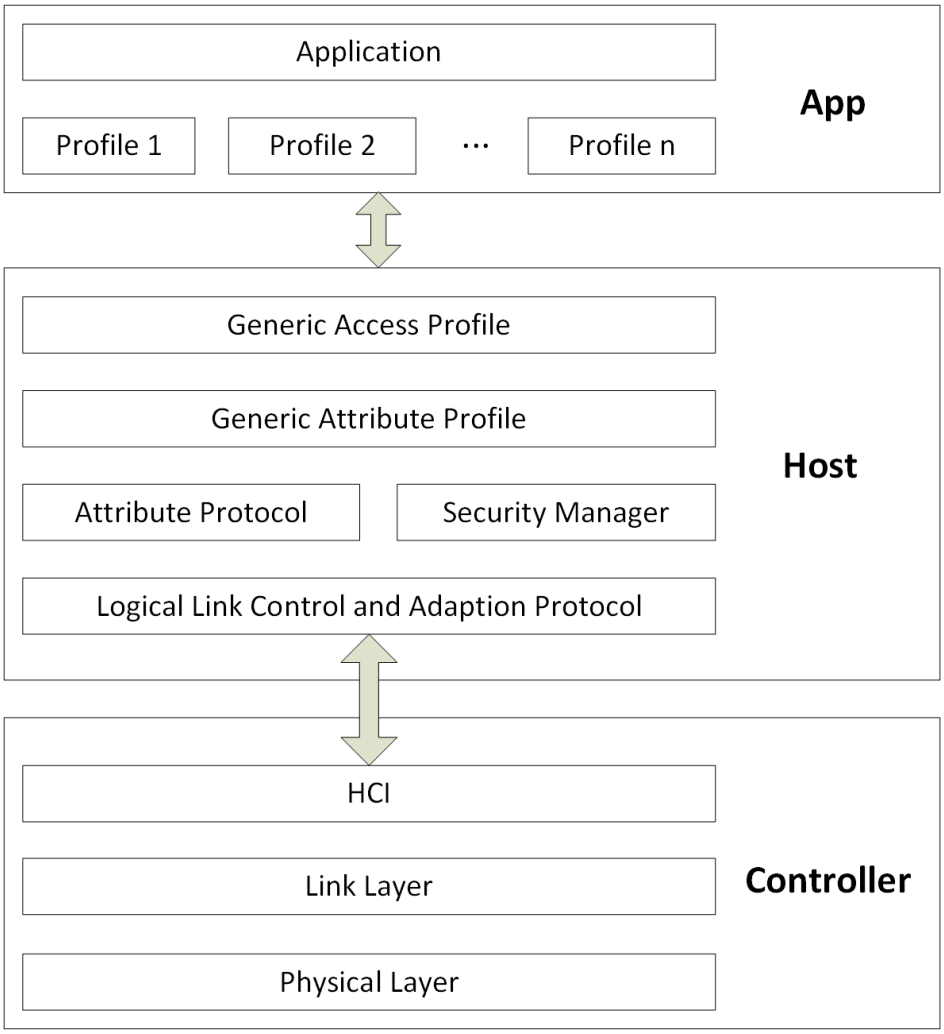


Figure 3.1: BLE SDK software architecture

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

- Controller as the lower layer 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 as BLE upper layer protocol, the protocol has Logic Link Control and Adaption Protocol (L2CAP), Attribute Protocol (ATT), Security Manager Protocol (SMP), Profile includes Generic Access Profile (GAP) and Generic Attribute Profile (GATT).
- The application layer (APP) contains the user's own related application code and various service corresponding profiles, and the user controls access to the Host through the GAP. The Host completes data interaction with the Controller through the HCI, as shown in the following figure.

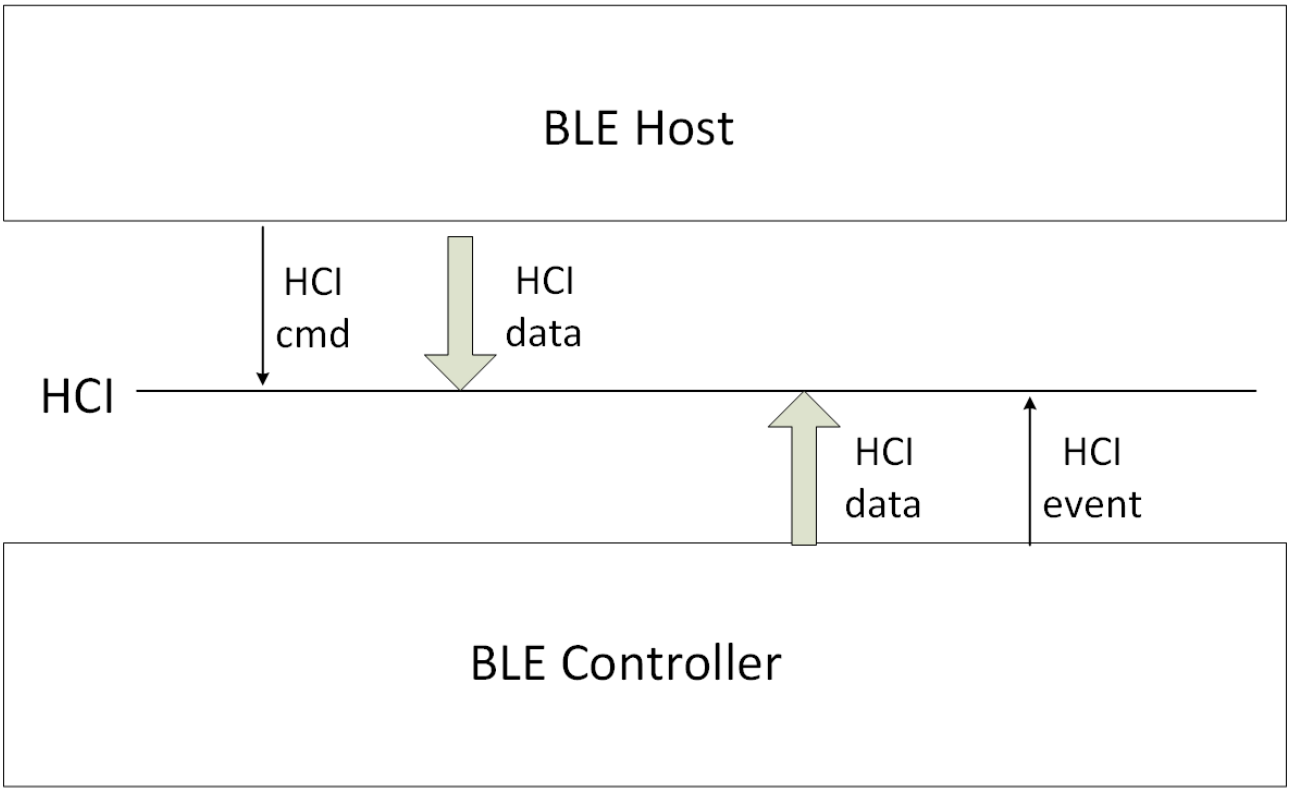


Figure 3.2: HCI data interaction between Host and Controller

(1) BLE Host operates and sets the Controller through HCI cmd, these HCI cmd 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 detail in this chapter.

(3) The Host transmits the data that needs to be sent to the other party device to the Controller through HCI, and the Controller drops the data directly to the Physical Layer for sending.

(4) Controller in the Physical Layer received RF data, first determine whether the data belongs to the Link Layer or Host: If it is Link Layer data, process the data directly; If it is the data of the Host, the data will be 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 Multiple Connection SDK supports standard BLE controller, including HCI, PHY (Physical Layer) and LL (Link Layer).

Telink BLE Multiple Connection SDK contains the five standard states of the Link Layer (standby, advertising, scanning, initiating, connection), The connection state supports up to 4 Master roles and 4 Slave roles at the same time.

The controller architecture is as follows:

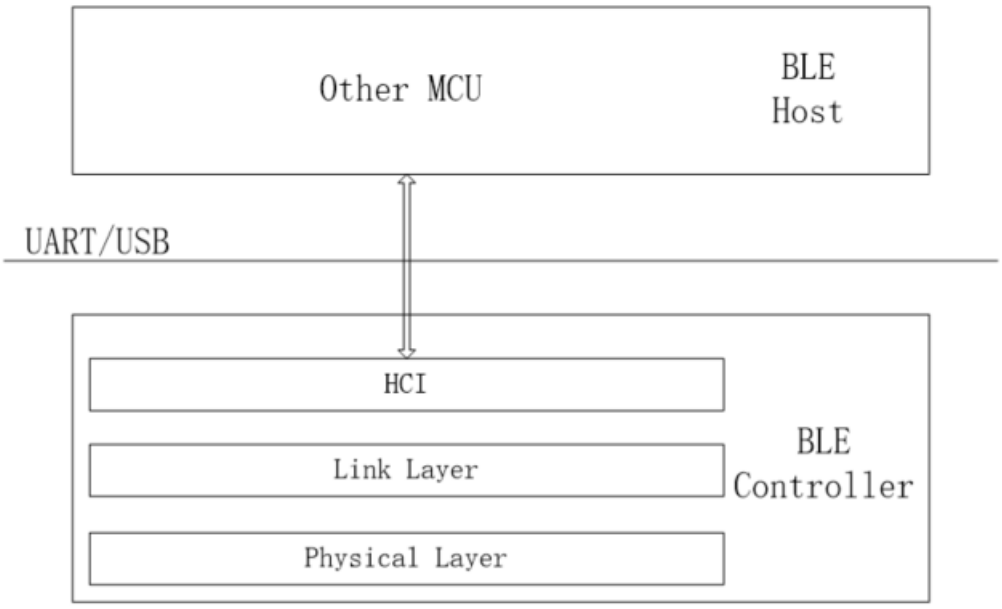


Figure 3.3: Telink HCI architecture

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

Telink BLE Multiple Connection SDK provides BLE multiple connection Whole Stack (Controller + Host) reference design, Only for Master SDP (service discovery) can not be fully supported, which will be introduced in detail in the following chapters.

Telink BLE stack architecture will do some simplification to the standard structure above, so that the system resource overhead (including Sram, Running time, Power consumption, etc.) of the whole SDK is minimized, and its architecture is shown in the figure below. The b85m_demo, b85_m1s1, b91_demo and b91_slave in the SDK are based on this architecture.

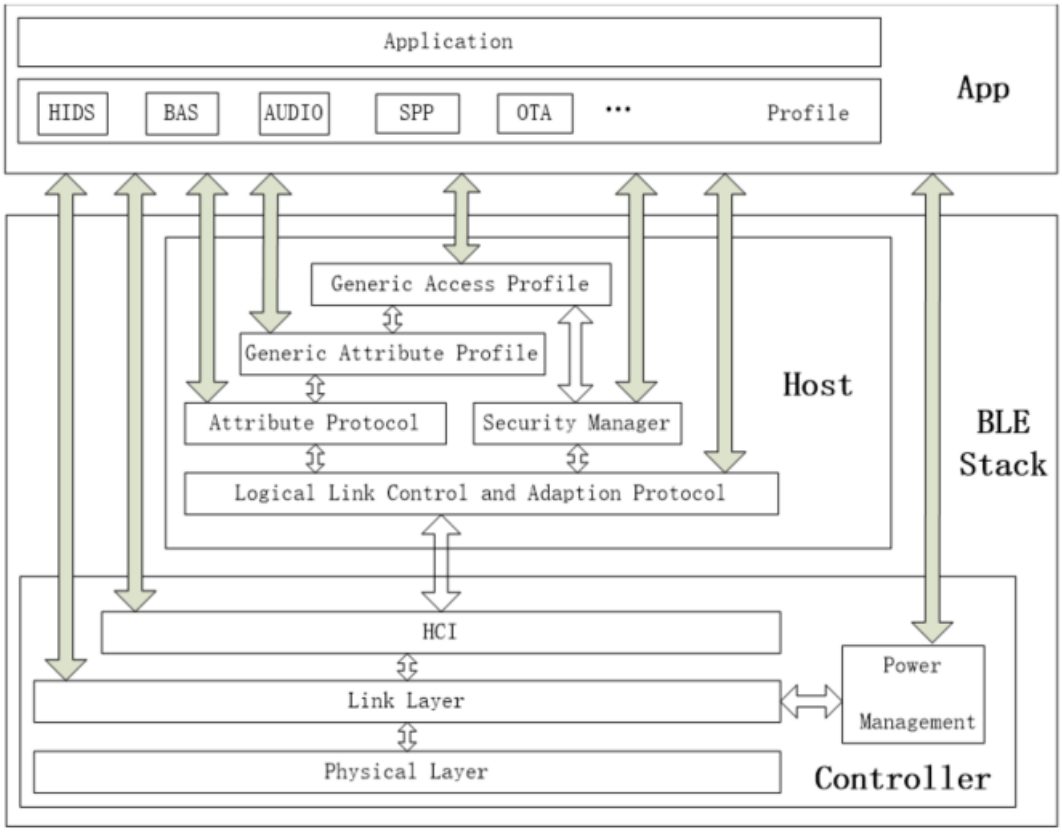


Figure 3.4: Telink BLE Multiple Connection Whole Stack Structure

The data interaction shown by the solid arrow in the figure can be operated and controlled by the user through various interfaces, and the user API will be provided. The hollow arrow is the data interaction completed within the protocol stack, and the user cannot participate.

HCI is the data communication interface between Controller and Host (docking with L2CAP layer), but it is not the only interface. APP application layer can also directly interact with Link Layer for data. The Power Management (PM) low power management unit is embedded into the Link layer, and the application layer can call the relevant interfaces of PM to set the power management.

Considering the efficiency, the data interaction between the application layer and Host is not controlled through GAP to access, the protocol stack provides relevant interfaces in ATT, SMP and L2CAP, which can interact directly with the application layer. However, all Events of Host need to interact with the application layer through the GAP layer.

The Host layer is based on Attribute Protocol and implements Generic Attribute Profile (GATT). The application layer is based on GATT and defines the various profiles and services that the user needs. This BLE SDK provides several basic profiles, including HIDS, BAS, OTA, etc.

Based on this architecture, the following is a basic introduction to each part of the BLE multiple connection 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 participation from the application layer, so this part is not introduced.

Although the Host and Controller part of the data interaction or rely on HCI to complete, basically completed

by the host and controller protocol stack, the application layer is almost not involved, only needs to register the HCI data callback processing function in the L2CAP layer, so the HCI part is not introduced.

3.2 Controller

3.2.1 Connection Number configuration

3.2.1.1 supportedMaxMasterNum & supportedMaxSlaveNum

The Telink BLE Multiple Connection SDK 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: Support the correspondence between the maximum number of master-slave and Library

IC l	ibrary	supportedMaxMaster-Num	supportedMaxSlaveNum
B91	liblt_9518	4 4	
825x lib	lt_8258	4 4	
lib	lt_8258_m0s4	0 4	
lib	lt_8258_m1s1	1 1	
827x	liblt_8278	4 4	
lib	lt_8278_m0s4	0 4	
lib	lt_8278_m1s1	1 1	

The SDK can query the number of Master and Slave supported by the current stack through the following API.

```
int blc_ll_getSupportedMaxConnNumber(void);
```

3.2.1.2 appMaxMasterNum & appMaxSlaveNum

With the premise that supportedMaxMasterNum and supportedMaxSlaveNum have been determined, users can set the maximum number of Master and Slave they want in their applications through the following API, which are called appMaxMasterNum and appMaxSlaveNum respectively.

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

This API is only allowed to be called during initialization, that is, the number of relevant 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 example design uses this API during initialization:

```
blc_ll_setMaxConnectionNumber( MASTER_MAX_NUM, SLAVE_MAX_NUM);
```

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

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

appMaxMasterNum and appMaxSlaveNum can save various resources of MCU, for example, for the M4S4 library, if the user only needs to use M3S2, after setting MASTER_MAX_NUM and SLAVE_MAX_NUM to 3 and 2 respectively:

(1) Save SRAM resources

Link Layer TX Master FIFO and TX Slave FIFO, L2CAP Master MTU buffer and L2CAP Slave MTU buffer are allocated according to appMaxMasterNum and appMaxSlaveNum, so it can save some Sram resources. For details, please refer to the TX FIFO related introduction at the back of the document.

(2) Save time resources and power consumption

For M4S4, Stack must wait until currentMasterNum is 4 to stop the Scan action, and must wait until currentSlaveNum is 4 to stop the Advertising action. For M3S2, Stack waits until the currentMasterNum is 3 to stop the Scan action, and the currentSlaveNum is 2 to stop the Advertising action, so that there is less unnecessary Scan and Advertising, which can save the PHY layer bandwidth and reduce MCU power consumption.

3.2.1.3 currentMaxMasterNum & currentMaxSlaveNum

After the user defines appMaxMasterNum and appMaxSlaveNum, the maximum number of Master and Slave created at Link Layer runtime is determined. However, the number of Master and Slave at a given moment is still uncertain, for example, if appMaxMasterNum is 4, the number of Master at any moment may be 0,1,2,3,4.

The SDK provides the following 3 APIs for users to query the current number of Master and Slave on the Link Layer in real time.

```
int blc_ll_getCurrentConnectionNumber(void);//Master + Slave connection number
int blc_ll_getCurrentMasterRoleNumber(void);//Master role number
int blc_ll_getCurrentSlaveRoleNumber(void);//Slave  role number
```

3.2.2 Link Layer State Machine

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

For Telink BLE Multiple Connection SDK, because to support multiple Master and Slave at the same time, Link Layer can't be in only one state at a certain moment, it must be a combination of several states.

The Link Layer state machine of the Telink BLE Multiple Connection SDK is complex. Just a general introduction can meet the user's basic understanding of the lower layer and the use of the corresponding API.

3.2.2.1 Link Layer State Machine Initialization

Telink BLE Multiple Connection SDK designs each basic state according to modularity, and the modules that need to be used need to be initialized in advance.

MCU initialization is necessary. The API is as follows:

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

The Add API for the Standby module is as follows, this is necessary and all BLE applications need to be initialized.

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

The actual parameter public_adr is the pointer to BLE public mac address.

The initialization APIs of the corresponding modules for several other states (Advertising, Scanning, ACL Master, ACL Slave) are as follows:

```
void   blc_ll_initLegacyAdvertising_module(void);
void   blc_ll_initLegacyScanning_module(void);
void  blc_ll_initAclConnection_module(void);
void   blc_ll_initAclMasterRole_module (void);
void   blc_ll_initAclSlaveRole_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, and within a certain period of time (this time is called create connection timeout) either the connection is established successfully and there is an additional Connection Master role, or the connection is established unsuccessfully and the Link Layer returns to the Scanning state. In order to simplify the introduction of Link Layer state machine and make it easier for users to understand, the brief temporary state of Initiating is ignored in the following introductions.

The Link Layer state machine in the Telink BLE Multiple Connection SDK can be described from two perspectives: one is the transition between Advertising and Slave; the second is the transition between Scanning and Master; these two perspectives do not affect each other.

Take M1S1 as an example analysis, assuming that the user's appMaxMasterNum and appMaxSlaveNum are both 1. The state machine for M1S1 Advertising and Slave switching is as follows:

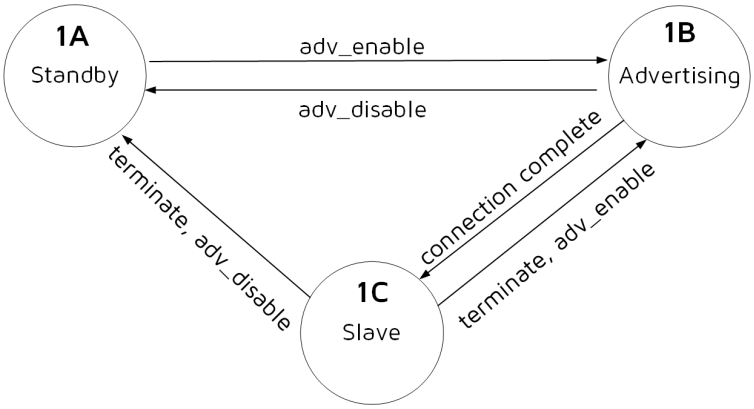


Figure 3.5: M1S1 advertising and slave switching

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

The state machine for M1S1 Scanning and Master switching is as follows:

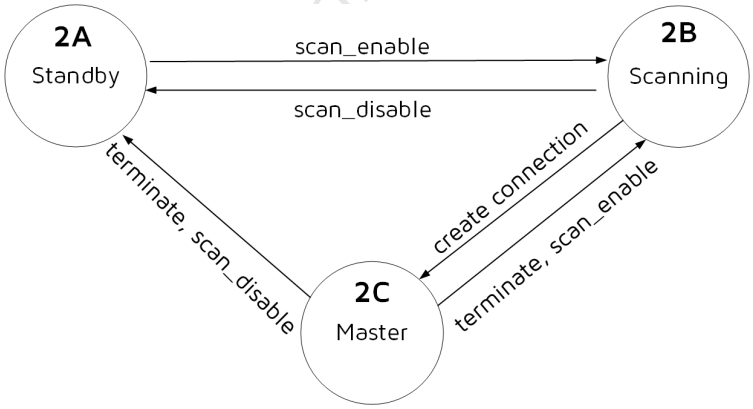


Figure 3.6: M1S1 scanning and master switching

In the figure, scan_enable and scan_disable refer to the state set by the user's last call to blc_ll_setScanEnable (scan_enable, filter_duplicate) when the condition occurs.

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

Table 3.2: M1S1 Link Layer combination status

	2A 2	B 2C	
1A S	tandby S	canning Mast	er
1B A	dvertising A	dvertising + Scanning A	dvertising + Master
1C S	lave Sla	ve + Scanning Slave	+ Master

Take a more complex M4S4 analysis, assuming that the user's appMaxMasterNum and appMaxSlaveNum are 4 and 4 respectively, the state machine for M4S4 Advertising and Slave switching is as follows:

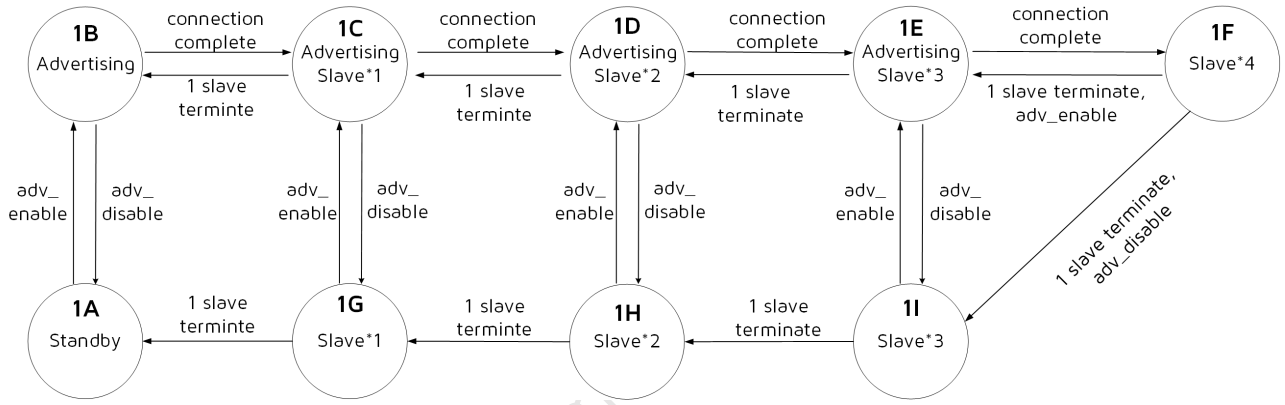


Figure 3.7: M4S4 advertising and slave switching

The state machine for M4S4 Scanning and Master switching is as follows:

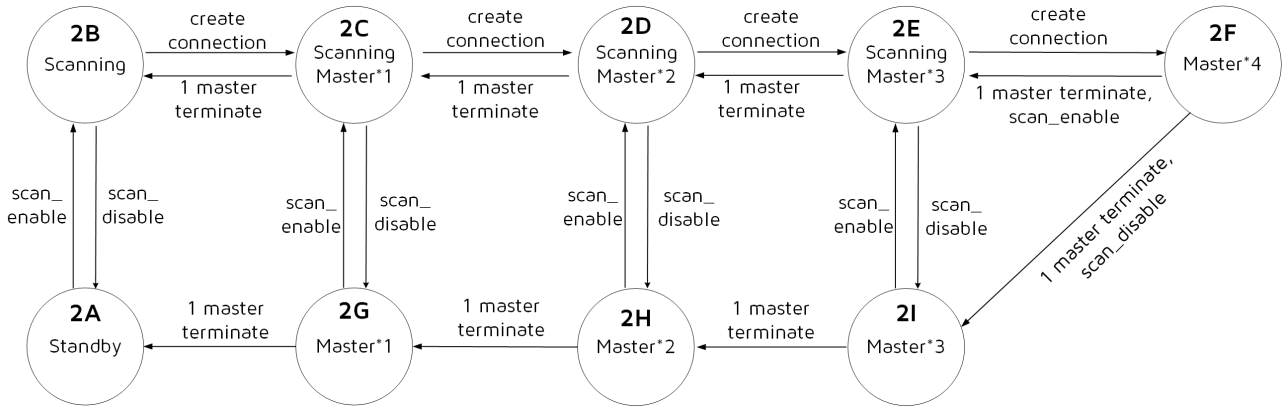


Figure 3.8: M4S4 scanning and master switching

Advertising and Slave have 9 possible states, and Scanning and Master have 9 possible states. Since the logic between the two is completely independent and does not affect each other, then the final Link Layer combination state has a total of 9*9=81 kinds, as shown in the following table:

Table 3.3: M4S4 Link Layer combination status

2A	2B	2C	2D	2E	2F	2G	2H	2I	
1A S	tandby S	canning Sca	nning Sca Mas- ter*1	nning Sca Mas- ter*2	nning Mas Mas- ter*3	ter*4 Mas	ter*1 Mas	ter*2 Mas	ter*3
1B A	dv Adv	Adv Scan- ning Sc	Adv anning Sc Mas- ter*1	Adv anning Sc Mas- ter*2	Adv anning Ma Mas- ter*3	Adv ster*4 Ma	Adv ster*1 Ma	Adv ster*2 Ma	ster*3
1C A	dv A Slave*1 Sla	dv A ve*1 Sla Scan- ning Mas- ter*1	dv A ve*1 Sla Scan- ning Mas- ter*2	dv A ve*1 Sla Scan- ning Mas- ter*3	dv A ve*1 Sla Scan- ning	dv A ve*1 Sla Mas- ter*4	dv A ve*1 Sla Mas- ter*1	dv A ve*1 Sla Mas- ter*2	dv ve*1 Mas- ter*3
1D A	dv A Slave*2	dv A Slave*2 Scan- ning	dv A Slave*2 Scan- ning Mas- ter*1	dv A Slave*2 Scan- ning Mas- ter*2	dv A Slave*2 Scan- ning Mas- ter*3	dv A Slave*2 Mas- ter*4	dv A Slave*2 Mas- ter*1	dv A Slave*2 Mas- ter*2	dv Slave*2 Mas- ter*3
1E A Sla	dv A ve*3 Slave	dv A *3 Slave Scanning Sc	dv A *3 Slave anning Sc Mas- ter*1	dv A *3 Slave anning Sc Mas- ter*2	dv A *3 Slave anning Ma Mas- ter*3	dv A *3 Slave ster*4 Ma	dv A *3 Slave ster*1 Ma	dv A *3 Slave ster*2 Ma	dv *3 ster*3
1F S	lave*4 Slav	e*4 Slav Scan- ning Sc	e*4 Slav anning Sc Mas- ter*1	e*4 Slav anning Sc Mas- ter*2	e*4 Slav anning Ma Mas- ter*3	e*4 Slav ster*4 Ma	e*4 Slav ster*1 Ma	e*4 Slav ster*2 Ma	e*4 ster*3
1G S	lave*1 Slav	e*1 Slav Scan- ning Sc	e*1 Slav anning Sc Mas- ter*1	e*1 Slav anning Sc Mas- ter*2	e*1 Slav anning Ma Mas- ter*3	e*1 Slav ster*4 Ma	e*1 Slav ster*1 Ma	e*1 Slav ster*2 Ma	e*1 ster*3
1H S	lave*2 Slav	e*2 Slav Scan- ning Sc	e*2 Slav anning Sc Mas- ter*1	e*2 Slav anning Sc Mas- ter*2	e*2 Slav anning Ma Mas- ter*3	e*2 Slav ster*4 Ma	e*2 Slav ster*1 Ma	e*2 Slav ster*2 Ma	e*2 ster*3

2A	2B	2C	2D	2E	2F	2G	2H	2I	
1I S	lave*3 Slav	e*3 Slav Scan- ning Sc	e*3 Slav anning Sc Mas- ter*1	e*3 Slav anning Sc Mas- ter*2	e*3 Slav anning Ma Mas- ter*3	e*3 Slav ster*4 Ma	e*3 Slav ster*1 Ma	e*3 Slav ster*2 Ma	e*3 ster*3

If the user's appMaxMasterNum appMaxSlaveNum is not M1S1 or M4S4, please analyze according to the above analysis method.

The concepts of supportedMaxMasterNum/ supportedMaxSlaveNum and appMaxMasterNum/ appMaxSlaveNum were introduced earlier, corresponding to the number of Master and Slave in the above state machine combination table, two concepts currentMasterNum and currentSlaveNum were defined to indicate the actual number of Master and Slave in the Link Layer at the current moment, for example, in the "1D2E" combination state in the above table, currentMasterNum is 3 and currentSlaveNum is 2.

3.2.3 Link Layer Timing

Link Layer timing is complex, here is only the introduction of some of the most basic knowledge, enough for users to understand, and rational use of related APIs.

The five basic single states of Link Layer are Standby, Advertising, Scanning, Initiating, and Connection. Ignoring the short Initiating, which is only used when the Master create connection, we only briefly introduce the timing of the remaining four states.

In this section, we use the M4S4 state as an example to illustrate, assuming that appMaxMasterNum and appMaxSlaveNum are 4 and 4, respectively.

Each sub-state (Advertising, Master0 ~ Master3, Slave0 ~ Slave3, Scanning, UI task) will be indicated by the following figure:

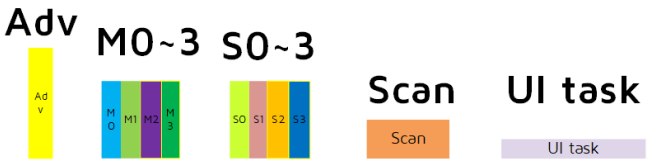


Figure 3.9: Each sub state indication

3.2.3.1 Standby state Timing

Corresponds to the 1A2A state of M4S4 in table 3.3.

When the Link Layer is in Idle state, the Link Layer and Physical Layer do not have any tasks to process, and the blc_sdk_main_loop function does not work at all and does not produce any interrupts. It can be considered that UI entry (UI task) occupies the whole main_loop time.

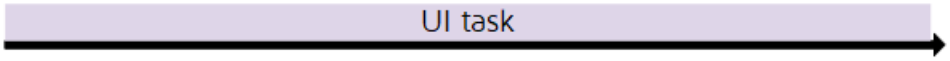


Figure 3.10: Standby statu timing

3.2.3.2 Scanning only, no Adverting, no Connection Timing

Corresponds to the 1A2B state of M4S4 in Table 3.3.

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

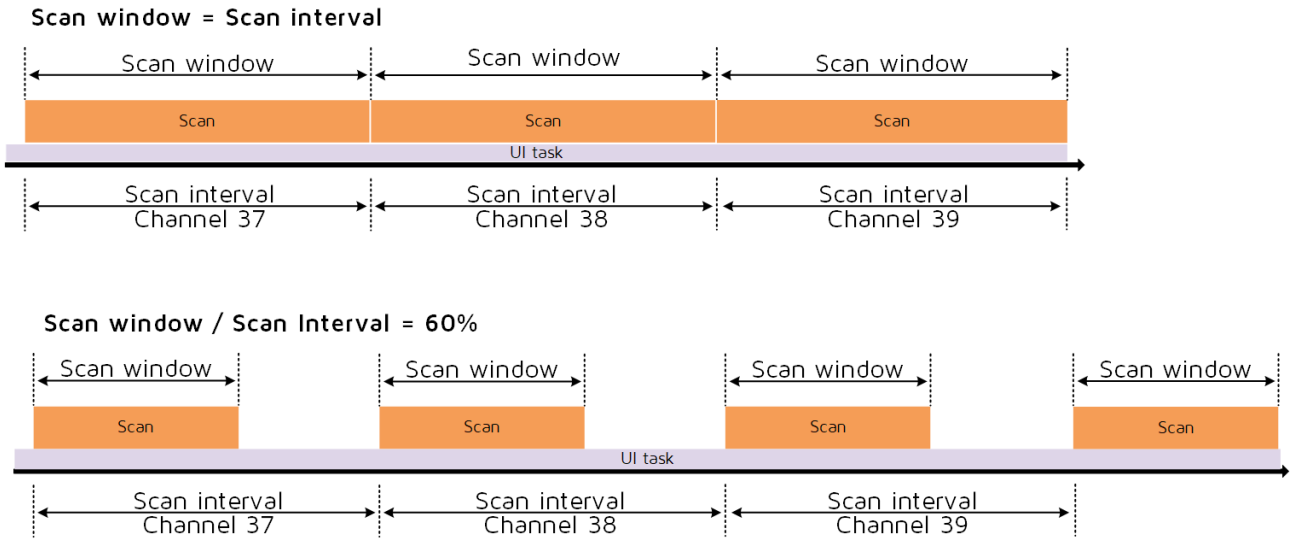


Figure 3.11: Scanning state timing allocation

According to the size of the Scan window to determine the true Scan time, if the Scan window is equal to the Scan interval, all the time in the Scan; if the Scan window is less than the Scan interval, select the time equal to the Scan window from the front in the Scan interval to Scan.

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, the user can use this time to execute their UI task in main_loop. The last 40% of the time is not in the Scanning state, the PHY layer stops working, the user can use this time to execute their UI task in main_loop. For the design of the low power management to be described later, this time also allows the MCU to enter into suspend to reduce the power consumption of the whole machine.

3.2.3.3 Advertising only, no Scanning, no Connection Timing

Corresponds to the 1B2A state of M4S4 in Table 3.3.

The Advertising Event is allocated to the timeline according to the Adv interval and the timing diagram is as follows:

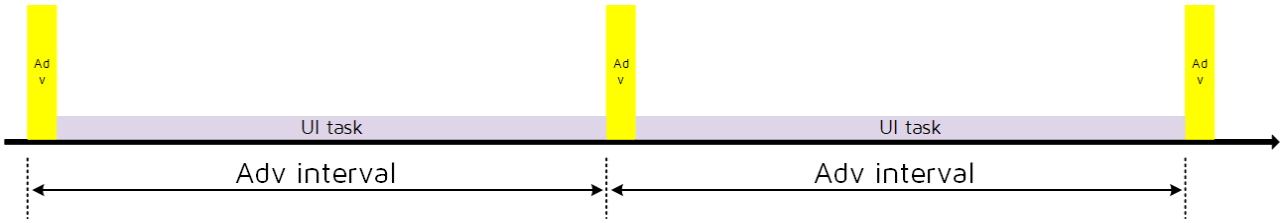


Figure 3.12: Advertising state timing allocation

For all the details of Adv Event, just refer to the details of Adv Event in the Telink BLE Single Connection SDK, both of them are the same.

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

3.2.3.4 Advertising, Scanning, no Connection Timing

Corresponds to the 1B2B state of M4S4 in Table 3.3.

First, according to the Adv interval, the Advertising Event is first allocated to the timeline, and then the Scanning is allocated, the timing diagram is as follows:

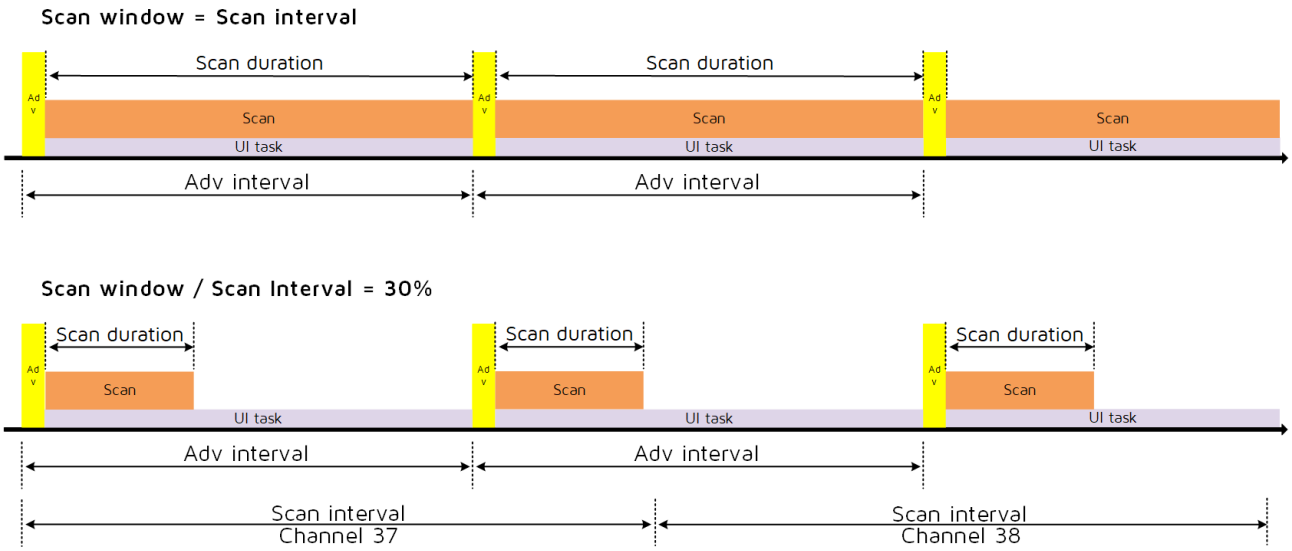


Figure 3.13: Advertising + Scanning state timing allocation

Since the application requires higher timing accuracy for advertising than scan, Adv Event has higher priority at this time, first allocate the timing of Adv Event, then use the remaining time between Adv Event for Scan, while the users can use this remaining time to execute their 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 the remaining time; when the Scan window set by the user is less than the Scan interval, the Link Layer will automatically calculate and get a Scan duration that satisfies the following condition: Scan duration/(Adv interval + rand_dly) should be equal to Scan window/Scan interval as much as possible.

3.2.3.5 Connection, Advertising, Scanning Timing

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

The following figure corresponds to the 1C2C state of M4S4 in Table 3.3.

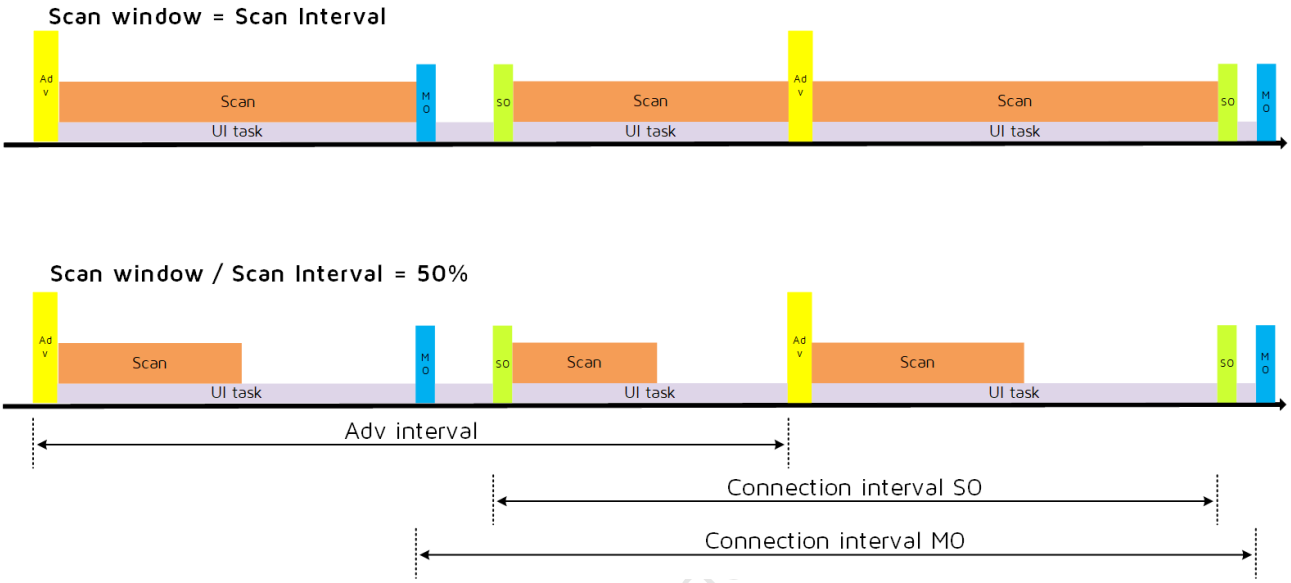


Figure 3.14: M4S4 1C2 state timing allocation

The allocation of connection tasks (whether Master or Slave) is done first and will be allocated according to the timing of the respective connections. If multiple tasks occupy the same time slot and conflict occurs, they will be allocated according to the priority and preempted with high priority. Abandoned tasks are automatically increased in priority to ensure that they are not always discarded.

Then the adv task is allocated, the principle is:

(1) The time interval from the last adv event should be greater than the set minimum adv interval time.

(2) The time between and the next task is greater than a certain value (3.75ms), because adv takes a certain amount of time to complete.

(3) The allocated time period is not occupied by other connection tasks.

Finally the allocation of scan tasks, the principle is: as long as there is more than enough time between the two tasks (stack limits the minimum time, too short to no longer scan), this time is allocated to the scan tasks, and the percentage of the scan is also confirmed according to the scan window/Scan interval.

3.2.3.6 Connection, no Advertising, no Scanning Timing

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 M4S4 in Table 3.3.

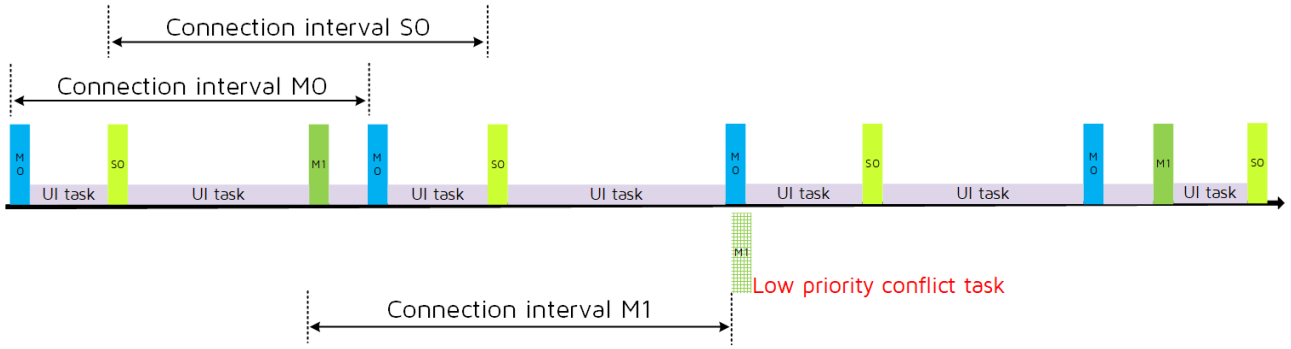


Figure 3.15: M4S4 1F2H state timing allocation

The following figure corresponds to the 1E2F state of M4S4 in Table 3.3.

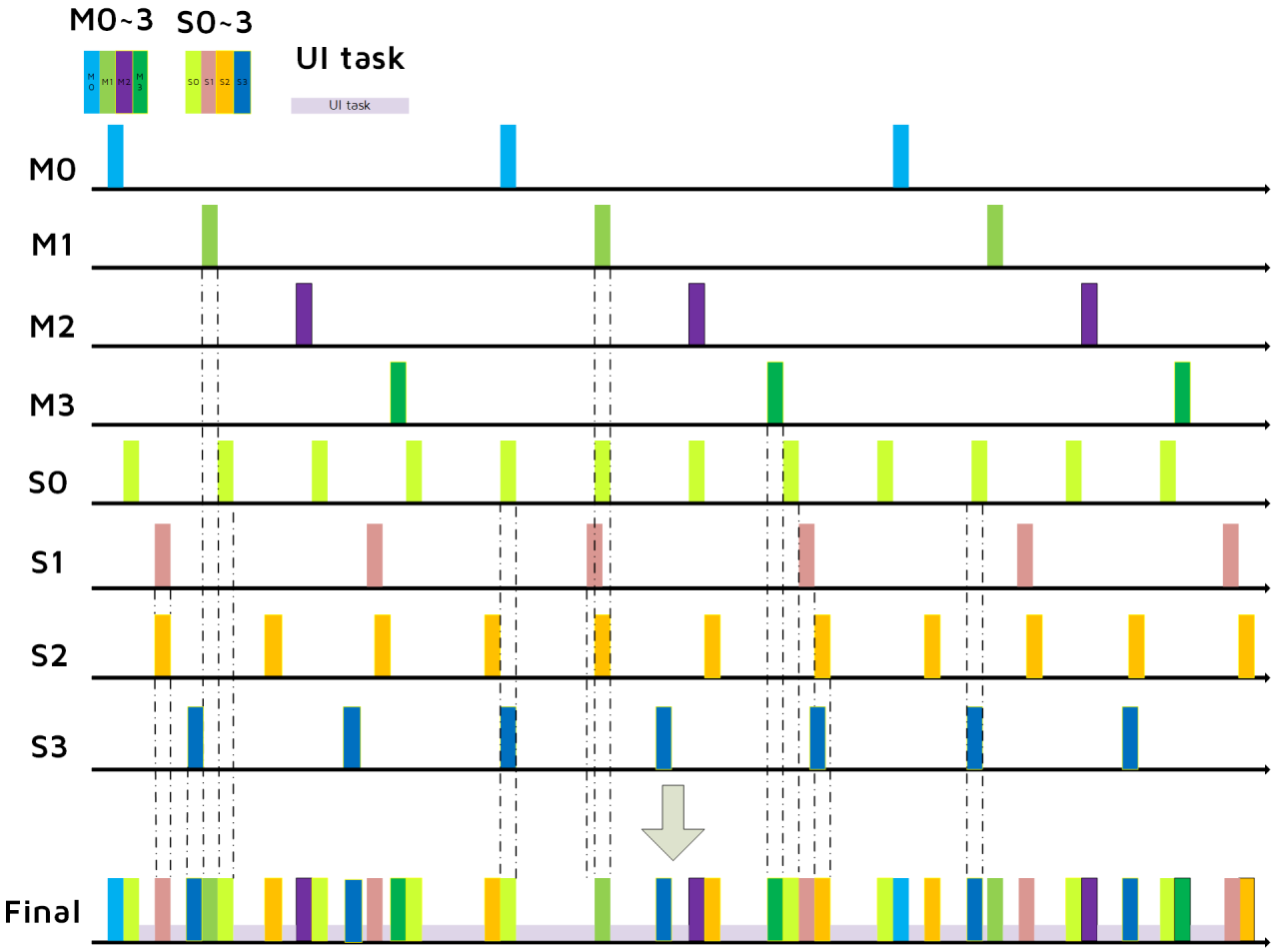


Figure 3.16: M4S4 1E2F state timing allocation

At this time, only connected tasks are assigned according to the respective connected timings. If a conflict occurs, it will be allocated according to the priority, the high priority tasks will preempt, abandoned tasks are automatically given increased priority, increasing the chances of grabbing them in the next conflict.

3.2.4 ACL TX FIFO & ACL RX FIFO

3.2.4.1 ACL TX FIFO Definition and Setting

Application layer and BLE Host all the data eventually need to complete the RF data sent through the Controller's Link Layer, in the Link Layer according to the number of connections set by the user, the corresponding TX FIFO is defined.

The ACL TX FIFO is defined as follows:

```
u8 app_acl_mstTxfifo[ACL_MASTER_TX_FIFO_SIZE * ACL_MASTER_TX_FIFO_NUM * MASTER_MAX_NUM] = {0};
u8 app_acl_slvTxfifo[ACL_SLAVE_TX_FIFO_SIZE * ACL_SLAVE_TX_FIFO_NUM * SLAVE_MAX_NUM] = {0};
```

The TX FIFO of ACL Master and ACL Slave are defined separately. Take ACL Slave as an example, ACL Master principle is the same, and so on.

The size of the array app_acl_slvTxfifo is related to three macros:

(1) SLAVE_MAX_NUM is the maximum number of connections, i.e. appMaxSlaveNum. Users can modify this value in app_config.h as needed.

(2) ACL_SLAVE_TX_FIFO_SIZE is the size of each sub_buffer, which is related to the possible maximum value of data sent by the ACL Slave. Use the following macro to define the implementation in the SDK.

```
#define ACL_SLAVE_TX_FIFO_SIZE  CAL_LL_ACL_TX_FIFO_SIZE(ACL_SLAVE_MAX_TX_OCTETS)
```

Among them, CAL_LL_ACL_TX_FIFO_SIZE is a formula, which is related to the implementation of MCU. Different MCU calculation methods may be different. Please refer to the notes in app_buffer.h for details.

ACL_SLAVE_MAX_TX_OCTETS is the user-defined slaveMaxTxOctets. if the client uses DLE (Data Length Extension), this value needs to be modified accordingly; the default value is 27, which corresponds to the minimum value when DLE is not used, which is used to save Sram. For details, please refer to the comments in app_buffer.h and the detailed description in Bluetooth Core Specification.

(3) ACL_SLAVE_TX_FIFO_NUM, the number of sub_buffer, please refer to the comments in app_buffer.h for the selection of this value. This value has a certain relationship with the amount of data sent in the client application, if the amount of data sent is large and the real-time requirement is high, you can consider a larger number.

The user defines the TX FIFO according to the actual situation, and the Master TX FIFO and the Slave TX FIFO are defined separately, so that one is for each connection's data are cached in their own TX FIFO, the TX data between each connection will not interfere with each other; the second is also according to the actual situation, flexible definition TX FIFO size, corresponding to reduce the consumption of ram. For example:

- The Slave needs the DLE function, while the Master does not need DLE, so that the FIFOs can be defined separately to save ram space. For the explanation of DLE, please refer to "3.2.5 MTU and DLE".
- For example, if the customer is actually using 3 Master and 2 Slave, the customer can define only 3 Master tx fifo and 2 Slave tx fifo, thus reducing the use of ram and saving ram space:

```
#define MASTER_MAX_NUM      3
#define SLAVE_MAX_NUM       2
```

Below we describe the settings of the TX FIFO in various states with a diagram, so that you can have a more intuitive understanding. Here we Take B85m as an example, other chips (such as B91) is the same principle, and so on.

(1) M4S4, ACL Master, ACL Slave do not use DLE

Assume that the relevant definitions are as follows:

```
#define MASTER_MAX_NUM     4
#define SLAVE_MAX_NUM      4
#define ACL_MASTER_MAX_TX_OCTETS  27
#define ACL_SLAVE_MAX_TX_OCTETS   27

#define ACL_MASTER_TX_FIFO_SIZE      CAL_LL_ACL_TX_FIFO_SIZE(ACL_MASTER_MAX_TX_OCTETS)
#define ACL_MASTER_TX_FIFO_NUM      8
#define ACL_SLAVE_TX_FIFO_SIZE      CAL_LL_ACL_TX_FIFO_SIZE(ACL_SLAVE_MAX_TX_OCTETS)
#define ACL_SLAVE_TX_FIFO_NUM      8
```

Then the TX FIFO is defined as the following value:

```
u8 app_acl_mstTxfifo[40 * 8 * 4] = {0};
u8 app_acl_slvTxfifo[40 * 8 * 4] = {0};
```

The figure is as follows:

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):

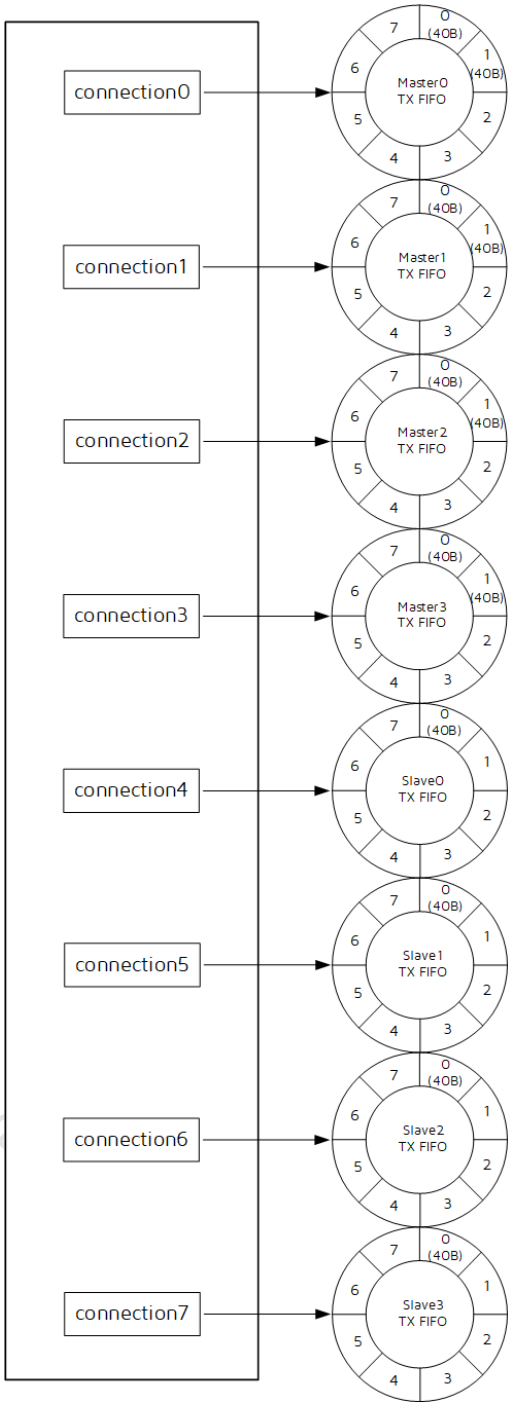


Figure 3.17: TX FIFO default setting

(2) M4S4, ACL Master uses DLE and MasterMaxTxOctets is maximum 251, ACL Slave does not use DLE.

Assume that the relevant definitions are as follows:

```
#define MASTER_MAX_NUM      4
#define SLAVE_MAX_NUM       4
#define ACL_MASTER_MAX_TX_OCTETS   251
#define ACL_SLAVE_MAX_TX_OCTETS    27
```

```
#define ACL_MASTER_TX_FIFO_SIZE      CAL_LL_ACL_TX_FIFO_SIZE(ACL_MASTER_MAX_TX_OCTETS)
#define ACL_MASTER_TX_FIFO_NUM      8
#define ACL_SLAVE_TX_FIFO_SIZE      CAL_LL_ACL_TX_FIFO_SIZE(ACL_SLAVE_MAX_TX_OCTETS)
#define ACL_SLAVE_TX_FIFO_NUM      8
```

Then the TX FIFO is defined as the following value:

```
u8 app_acl_mstTxfifo[264 * 8 * 4] = {0};
u8 app_acl_slvTxfifo[40  * 8 * 4] = {0};
```

As can be seen from the figure, the number of FIFOs for each connection of Master and Slave is the same, which is 8 (0 ~ 7). However, each FIFO size of the Master is 264b, while the FIFO size of the Slave is 40B.

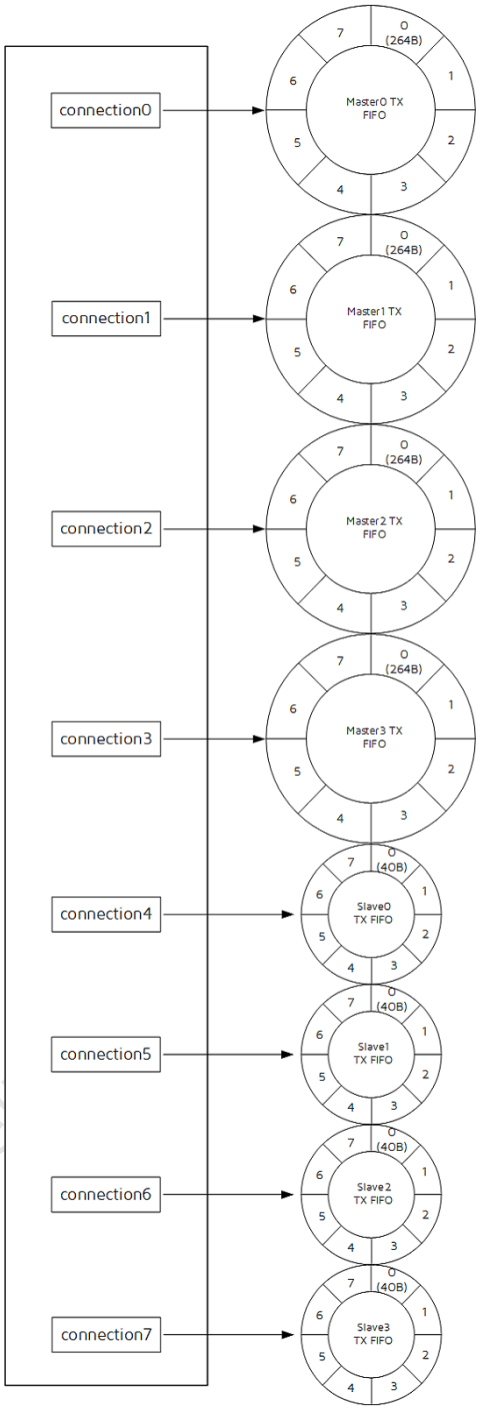


Figure 3.18: Master uses DLE and Slave does not use DLE's buffer

(3) M3S2, ACL Master, ACL Slave do not use DLE

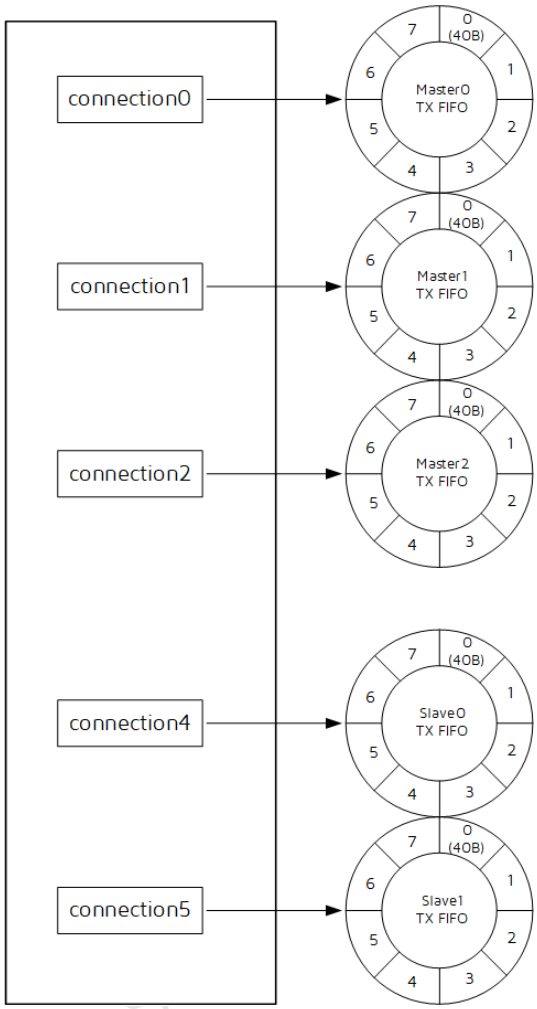


Figure 3.19: 3 Master and 2 Slave buffer situation

3.2.4.2 ACL RX FIFO Definition and Setting

The ACL RX FIFO is defined as follows:

```
u8 app_acl_rxfifo[ACL_RX_FIFO_SIZE * ACL_RX_FIFO_NUM] = {0};
```

The ACL master and ACL slave share the ACL RX FIFO. Take ACL slave as an example, ACL master principle is the same, and so on.

The size of the array app_acl_rxfifo is related to two macros:

(1) ACL_RX_FIFO_SIZE is buffer size, which is related to the possible maximum value of data received by the ACL Slave. Use the following macro definitions in the SDK to implement.

```
#define ACL_RX_FIFO_SIZE    CAL_LL_ACL_RX_FIFO_SIZE(ACL_CONN_MAX_RX_OCTETS)
```

Among them, CAL_LL_ACL_RX_FIFO_SIZE is a formula, which is related to the MCU implementation, different MCU calculation methods may be different, you can refer to the notes in app_buffer.h for details.

ACL_CONN_MAX_RX_OCTETS is the user-defined slaveMaxRxOctets. If the client uses DLE (Data Length Extension), this value needs to be modified accordingly; the default value is 27, which corresponds to the minimum value when DLE is not used, which is used to save Sram. For details, please refer to the comments in app_buffer.h and the detailed description in Bluetooth Core Specification.

(2) ACL_RX_FIFO_NUM, buffer number, please refer to the comments in app_buffer.h for the selection of this value. This value is related to the amount of data received in the client applications, if the amount of data received is large and the real-time requirement is high, you can consider a larger number.

Assume that the M4S4 ACL RX FIFO is defined as follows:

```
#define MASTER_MAX_NUM     4
#define SLAVE_MAX_NUM      4

#define ACL_CONN_MAX_RX_OCTETS    27
#define ACL_RX_FIFO_SIZE        CAL_LL_ACL_RX_FIFO_SIZE(ACL_CONN_MAX_RX_OCTETS)
#define ACL_RX_FIFO_NUM      16
```

The corresponding ACL RX FIFO allocation is shown below:

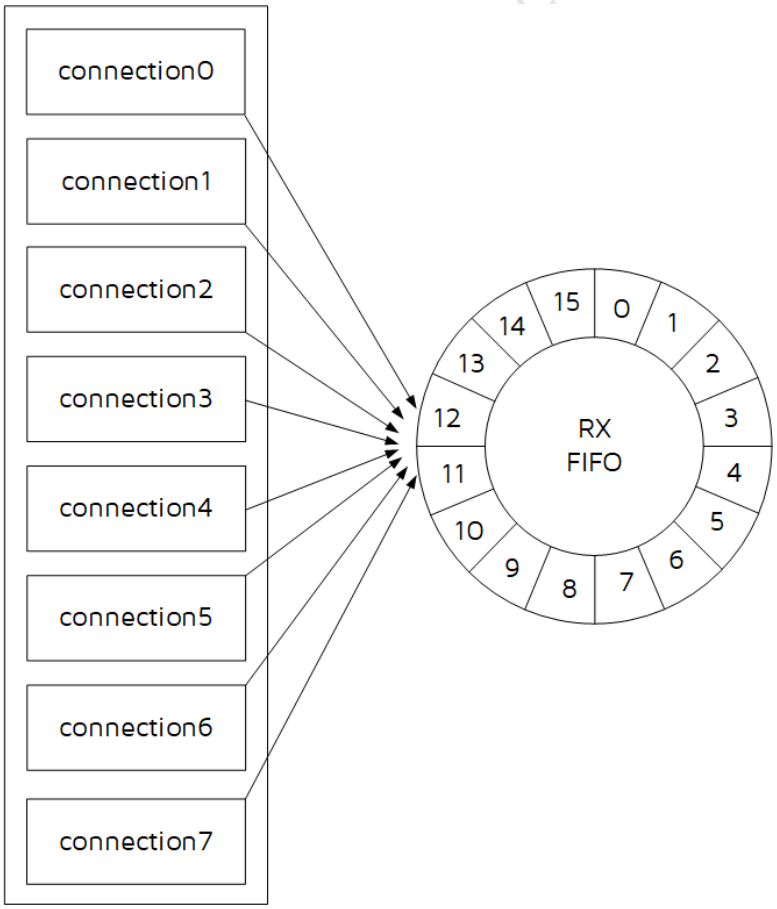


Figure 3.20: ACL RX buffer

3.2.4.3 RX overflow analysis

Refer to the introduction in Telink BLE Single Connection Handbook, the principle is the same.

3.2.5 MTU and DLE Concepts and Usage

3.2.5.1 MTU and DLE concepts description

Bluetooth Core Specification adds data length extension (DLE) from core_4.2.

The Link Layer in the Telink BLE Multiple Connection SDK supports data length extension and rf_len length supports the maximum length of 251 bytes in the Bluetooth Core Specification. For details, please refer to Bluetooth Core Specification V5.3, Vol 6, Part B, 2.4.2.21 LL_LENGTH_REQ and LL_LENGTH_RSP.

Before the specific explanation, we need to figure out what the concepts of MTU and DLE are. First look at a picture:

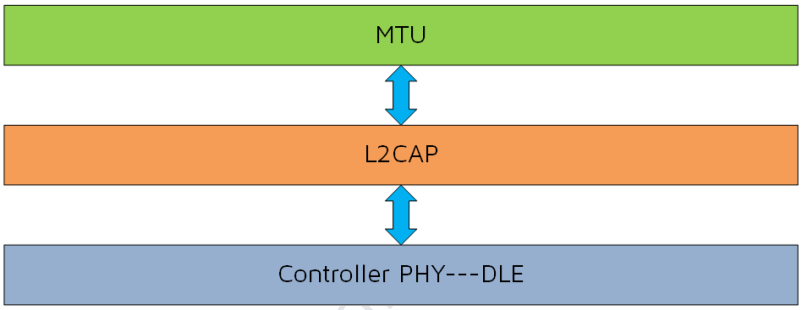


Figure 3.21: Concepts of MTU and DLE

The following figure shows the specific contents of MTU and DLE:

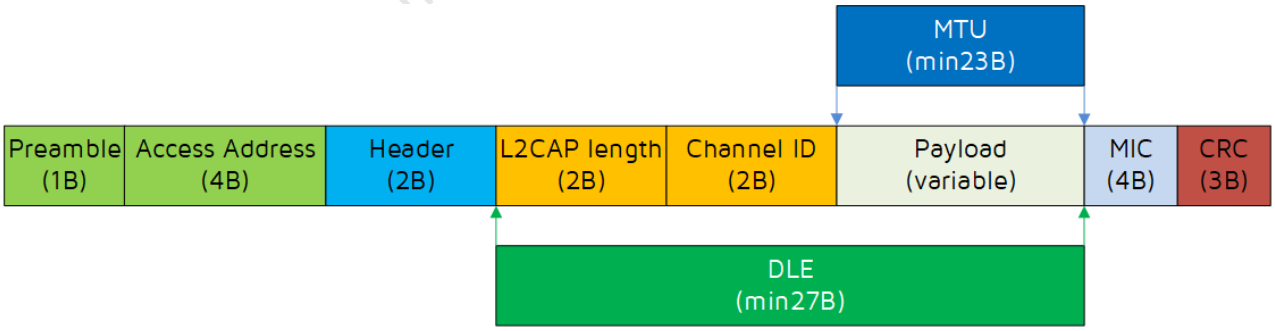


Figure 3.22: Contents of MTU and DLE

- MTU stands for Maximum Transmission Unit and is used in computer networking to define the maximum size of a Protocol Data Unit (PDU) that can be sent by a specific protocol.
- The Attribute MTU (ATT_MTU as defined by the specification) is the largest size of an ATT payload that can be sent between a client and a server. The Bluetooth Core Specification specifies that the minimum value of MTU is 23 bytes.

The DLE is data length extension. The BLE spec specifies that the minimum value of DLE is 27 bytes.

To carry more data in a packet requires negotiation between Master and Slave, interacting the DLE size through LL_LENGTH_REQ and LL_LENGTH_RSP.

The Bluetooth Core Specification specifies that the minimum length of DLE is 27 bytes and the maximum is 251 bytes. 251 bytes is because the rf length field is a byte, and the maximum length that can be represented is 255, If it is an encrypted link, it also requires 4 bytes MIC field: 251 + 4 = 255.

3.2.5.2 MTU and DLE automatic interaction method

If User needs to use the data length extension function, set it as follows. The SDK also provides the corresponding MTU&DLE usage demo, refer to feature_dle in the b85m_feature/b91_feature project.

Define macro in vendor/b85m_feature/feature_config.h or vendor/b91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE TEST_LL_DLE
```

(1) MTU size exchange

First, you need to interact with MTU, just modify ATT_MTU_MASTER_RX_MAX_SIZE and ATT_MTU_SLAVE_RX_MAX_SIZE, which represent the RX MTU size of ACL Master and ACL Slave respectively. The default is the minimum value of 23, just change it to the desired value. CAL_MTU_BUFF_SIZE is a fixed calculation formula and can not be modified.

The process of MTU size exchange ensures that the minimum MTU size of both parties takes effect, preventing the peer device from being unable to process long packets at the BLE L2cap layer and greater than or equal to 23.


```
#define ATT_MTU_MASTER_RX_MAX_SIZE    23
#define  MTU_M_BUFF_SIZE_MAX    CAL_MTU_BUFF_SIZE(ATT_MTU_MASTER_RX_MAX_SIZE)

#define ATT_MTU_SLAVE_RX_MAX_SIZE     23
#define MTU_S_BUFF_SIZE_MAX       CAL_MTU_BUFF_SIZE(ATT_MTU_SLAVE_RX_MAX_SIZE)
```

Then the following API is called inside the initialization to set the RX MTU size of ACL Master and ACL Slave respectively. **Note:** These two APIs need to be placed in blc_gap_init() to take effect.

```
blc_att_setMasterRxMTUSize(ATT_MTU_MASTER_RX_MAX_SIZE);
blc_att_setSlaveRxMTUSize(ATT_MTU_SLAVE_RX_MAX_SIZE);
```

For the implementation of MTU size exchange, please refer to the detailed description in the "ATT & GATT" section of this document —3.4.2.4.7 Exchange MTU Request, Exchange MTU Response, and also refer to the writing method of feature_dle in the b85m_feature/b91_feature project.

(2) Set connMaxTxOctets and connMaxRxOctets

Secondly, you need to set the size of the DLE of the Master and Slave, refer to the introduction of ACL TX FIFO and ACL RX FIFO, you only need to modify the following macros, and change 27 to the desired value.

```
#define ACL_CONN_MAX_RX_OCTETS    27
#define ACL_MASTER_MAX_TX_OCTETS  27
#define ACL_SLAVE_MAX_TX_OCTETS   27
```

Then the following API is called in the initialization to set the rx DLE size and tx DLE size of ACL Master and ACL Slave respectively.

```
ble_sts_t blc_ll_setAclConnMaxOctetsNumber(u8 maxRxOct, u8 maxTxOct_master, u8 maxTxOct_slave)
```

(3) Operation of sending and receiving long packets

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

On the basis that all the above 3 steps are completed correctly, you can start sending and receiving long packets.

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

```
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 packets, just needs to process the callback function "w" corresponding to Write request and Write Command, and in the callback function, reference the data pointed to by the formal parameter pointer.

3.2.5.3 MTU and DLE manual interaction method

If the user does not want the stack to automatically interact with MTU/DLE due to some special circumstances, the SDK also provides the corresponding API, and the user decides when to interact with MTU/DLE according to the specific situation. Most of the settings for manual interaction are the same as for automatic interaction, with the differences handled as described below.

For MTU, call API BLC_Att_SetMasterRRXMTUSIZE(23) and BLC_ATT_SETSLAVERXMTUSIZE(23) when initialization, set the initial MTU to the minimum value of 23, and no automatic interaction is performed after stack comparison. When the user needs to interact with the MTU, call these two APIs (blc_att_setMasterRxMTUSize and blc_att_setSlaveRxMTUSize) to set the MTU to the actual value, and then just call the API blc_att_requestMtuSizeExchange() to trigger the MTU interaction.

For DLE, you can use API blc_ll_setAutoExchangeDataLengthEnable(0) to disable automatic interaction during initialization, and then call API blc_ll_sendDateLengthExtendReq() to trigger DLE interaction when you need to interact. Refer to the "3.2.8 Controller API" for specific descriptions of these two APIs.

3.2.6 Coded PHY/2M PHY

Coded PHY and 2M PHY are new features added in Core_v5.0, which largely extend the application scenarios of BLE, Coded PHY includes S2 (500 kbps) and S8 (125 kbps) to adapt to longer distance applications, and

2M PHY (2 Mbps) greatly improves the BLE bandwidth. 2M PHY/Coded PHY can be used in the advertising channel or the data channel in the connected state.

3.2.6.1 Coded PHY/2M PHY Demo Introduction

In the Telink BLE Multiple Connection SDK, Coded PHY/2M PHY is turned off by default to save Sram, and users can turn it on manually if they choose to use this feature.

- Refer to b85m_feature/feature_2M_coded_phy or b91_feature/feature_2M_coded_phy for local device.

Define macro in feature_config.h:

```
#define FEATURE_TEST_MODE      TEST_2M_CODED_PHY_CONNECTION
```

- The peer device can choose to use a cell phone or other standard BLE Master device; it can also use any Telink BLE SDK that supports Coded PHY/2M PHY to compile a BLE Master device to run, such as xxx_kma_master_dongle compiled with Telink BLE Single Connection SDK or xxx_master_dongle compiled with Telink BLE Multiple Connection SDK.

3.2.6.2 Code PHY/2M PHY API Introduction

(1) API blc_ll_init2MPhyCodedPhy_feature()

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

Used to enable Coded PHY/2M PHY.

(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);
```

Bluetooth Core Specification standard interface, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.49 LE Set PHY command. Used to trigger the local device to actively apply for PHY exchange. If the determination result of PHY exchange is that the new PHY can be used, the Master device will trigger a PHY update and the new PHY will take effect soon. BLUETOOTH CORE SPECIFICATION, Vol 4, Part E, 7.8.49 LE Set PHY command

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

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

3.2.7 Channel Selection Algorithm #2

Channel Selection Algorithm #2 is a new Feature added in Bluetooth Core Specification V5.0, which has stronger anti-interference capability. For the specific instructions of the channel selection algorithm, please refer to Bluetooth Core Specification V5.3, Vol 6, Part B, 4.5.8.3 Channel Selection algorithm #2.

In the Telink BLE Multiple Connection SDK, CSA #2 is disabled by default and can be enabled manually if the user chooses to use this Feature, which needs to call the following API enable in user_init().

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

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

Demo reference b85m_feature/feature_misc, define the macro in feature_config.h:

```
#define FEATURE_TEST_MODE  TEST_MISC_FUNC
```

Define it in app.c of the corresponding project:

```
#define MISC_FUNC_TEST_MODE  TEST_CSA2
```

3.2.8 Controller API

In the standard BLE protocol stack architecture shown, the application layer cannot data interact directly with the Link Layer of Controller, the data must be sent down through the Host, and finally the Host transmits the control commands to the Link Layer through the HCI. All Controller control commands issued by the Host through the HCI interface are strictly defined in the Bluetooth Core Specification. For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E: Host Controller Interface Functional Specification.

The Telink BLE Multiple Connection SDK adopts the Whole Stack architecture, and the application layer directly operates and sets the Link Layer across the Host, but the APIs used are strictly in accordance with the HCI part of the Bluetooth Core Specification standard.

The declaration of the Controller API is in the header file in the stack/ble/controller directory and is classified according to the Link Layer state machine functions as ll.h, leg_adv.h, leg_scan.h, leg_init.h, acl_slave.h, acl_master.h, acl_conn.h, etc. Users can search according to the function of Link Layer, for example, APIs related to legacy advertising should be declared in leg_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 for most of the APIs in the SDK, BLE_SUCCESS (value 0) is returned only when the setup parameters of the API call are all correct and accepted by the protocol stack; all other non-zero values returned indicate setting errors, and each different value corresponds to an error type. In the following API specific description, all possible return values of each API will be listed, and the specific error reasons of each error return value will be explained.

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

3.2.8.1 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 public address and random static address:

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

The flash_addr can fill in the address of the MAC address stored on the flash, refer to the document in front of the introduction "2.1.4 SDK Flash Space Allocation". If you do not need random static address, the mac_random_static obtained above can be ignored.

After the BLE public MAC address is successfully obtained, the API for Link Layer initialization is called and the MAC address is passed into the BLE protocol stack:

```
blc_ll_initStandby_module (mac_public); //mandatory
```

3.2.8.2 bls_ll_setAdvData

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.7 LE Set Advertising Data command.

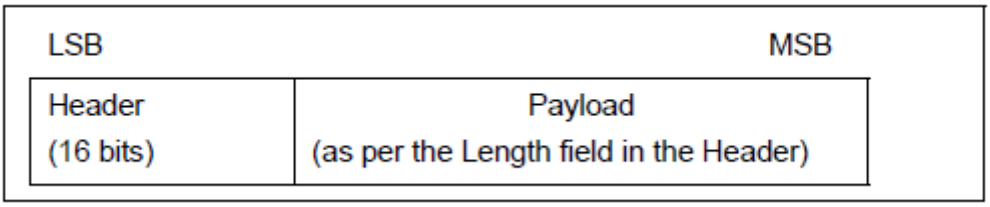


Figure 3.23: Protocol stack advertising package format

In the BLE stack, the format of the advertising packet is shown above, the first two bytes are the Header, followed by the Advertising PDU, up to 37 bytes, including AdvA (6B) and AdvData (up to 31B).

The following API is used to set the data in the AdvData section:

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

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

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

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success

The return value ble_sts_t is only BLE_SUCCESS, the API will not carry out parameter reasonableness check, the user needs to pay attention to the reasonableness of setting parameters.

The user can call this API to set the advertising data during initialization, and can also call this API at any time in the main_loop to modify the advertising data when the program is running.

3.2.8.3 bls_ll_setScanRspData

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.8 LE Set Scan Response Data command.

Similar to the advertising packet PDU settings above, the scan response PDU settings use the API:

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

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

The return type ble_sts_t may return the results shown in the table below:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success

The return value ble_sts_t is only BLE_SUCCESS, the API will not carry out parameter reasonableness check, the user needs to pay attention to the reasonableness of setting parameters.

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

3.2.8.4 bls_ll_setAdvParam

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.5 LE Set Advertising Parameters command.

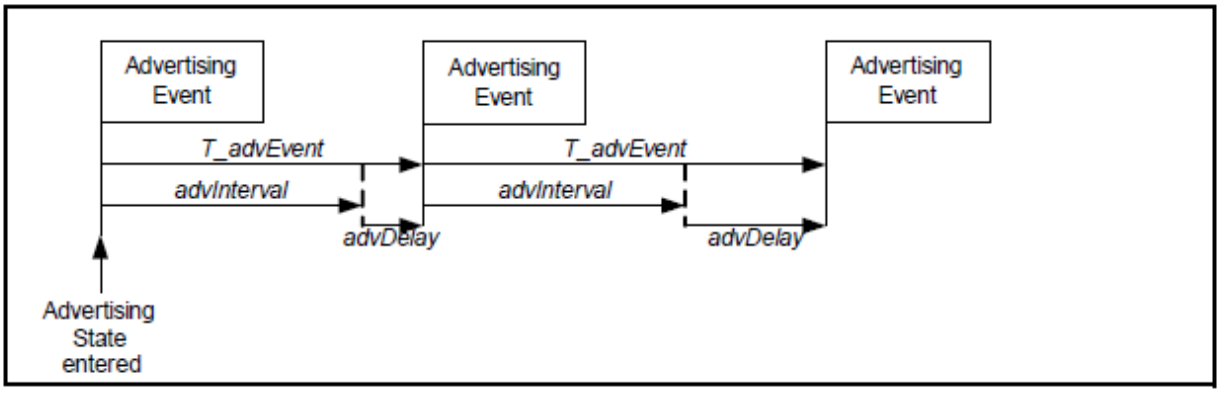


Figure 3.24: Advertising Event in BLE stack

The Advertising Event (Adv Event) in the BLE stack is shown in the figure above, which means that at each T_advEvent, the Slave performs a advertising round, sending 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_ll_setAdvParam( adv_inter_t intervalMin, adv_inter_t 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);
```

(1) intervalMin & intervalMax

Set the range of advertising interval adv interval, with 0.625 ms 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 intervals in the connected state and intervalMax for advertising intervals in the non-connected state.

If intervalMin > intervalMax is set, intervalMin will be forced to equal intervalMax.

According to different advertising packet types, the values of intervalmin and intervalmax are limited. Please refer to (Vol 6/Part B/ 4.4.2.2 "Advertising Events").

(2) advType

Referring to Bluetooth Core Specification, the four basic types of advertising events are as follows:

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

Figure 3.25: BLE protocol stack four advertising events

In the above figure, the Allowable response PDUs for advertising event section uses YES and NO to indicate whether the various types of advertising events respond to Scan request and Connect Request from other devices, for example: the first Connectable Undirected Event can respond to both Scan request and Connect Request, while the Non-connectable Undireted Event does not respond to either of them.

Note that the second Connectable Directed Event adds a "*" sign in the upper right corner of the "YES" response to the Connect Request, indicating that it will definitely respond as long as it receives a matching Connect Request, and will not be whitelist filter. The remaining three "YES" means that you can respond to the corresponding request, but the actual need to rely on the settings of the whitelist, according to the whitelist filter conditions to determine whether the final response, the latter will be described in detail in the whitelist.

Among the above four advertising events, Connectable Directed Event is further 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.

(3) ownAddrType

When specifying the advertising address type, 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;
```

Only the first two parameters are described here.

OWN_ADDRESS_PUBLIC means the public MAC address is used 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 indicates the use of random static MAC address when advertising, which is derived from the value set by the following API:

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

(4) peerAddrType & *peerAddr

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

(5) adv_channelMap

Set the advertising channel, you can select any one or more of channel 37, 38, 39. The value of Adv_Channelmap can be set as follows 3 or arbitrarily or combined with 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;
```

(6) advFilterPolicy

It is used to set the filtering policy 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 is explained in detail later in the whitelist introduction.

The four filter types that can be set are as follows, if you do not need the whitelist filter 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 reasons for the return value ble_sts_t are shown in the following table:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success

The return value ble_sts_t is only BLE_SUCCESS, the API will not carry out parameter reasonableness check, the user needs to pay attention to the reasonableness of setting parameters.

According to the design of the Host command in the HCI part of the Bluetooth Core Specification, Set Advertising parameters set the above 8 parameters at the same time. The idea of setting them at the same time is reasonable, because there is a coupling relationship between some different parameters, such as advType and advInterval, the range limits for intervalMin and intervalMax will be different under different advType, so there will be different range checks, if set advType and set advInterval are split into two different APIs, the range checks between each other cannot be controlled.

3.2.8.5 bls_ll_setAdvEnable

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.9 LE Set Advertising Enable command.

```
ble_sts_t blc_ll_setAdvEnable(adv_en_t adv_enable);
```

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

For the state machine changes of Enable or Disable Advertising, please refer to "3.2.2.2 Link Layer State Combination".

The possible values and reasons for the return value ble_sts_t are shown in the following table:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x0D	appMaxSlaveNum is 0, not allowed to set

3.2.8.6 blc_ll_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_ll_setAdvCustomedChannel(u8 chn0, u8 chn1, u8 chn2);
```

For chn0/chn1/chn2, fill in the frequency points that need to be customized, the default standard frequency points are 37/38/39, for example, set 3 advertising channel for 2420 MHz, 2430 MHz, 2450 MHz respectively, can be called as follows.

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

Regular BLE applications can use this API to achieve such a function, if the user wants to use single-channel advertising & single-channel scanning in some usage scenarios, for example, fixing the advertising channel & scanning channel to 39, you can call as follows:

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

Note that this API will change both the advertising and scan channels.

3.2.8.7 blc_ll_setScanParameter

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.10 LE Set Scan Parameters command.

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

Parameter description:

(1) scan_type

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

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

(2) scan_interval/scan window

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

The bottom layer will calculate scan_percent according to scan_window / scan_interval to achieve the purpose of reducing power consumption. For details of Scanning, 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, the ownAddrType has four 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 during Scan, and the actual address comes from the setting of the API blc_initMacAddress(int flash_addr, u8 mac_public, u8mac_random_static) when the MAC address is initialized.

OWN_ADDRESS_RANDOM indicates that the random static MAC address is used during Scan, which is derived from the value set by the following API:

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

(4) scan filter 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 Link Layer does not filter the scanned adv packet and reports it directly to the BLE Host.

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

The possible values and reasons for the return value ble_sts_t are shown in the following table:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success

The return value ble_sts_t is only BLE_SUCCESS, the API will not carry out parameter reasonableness check, the user needs to pay attention to the reasonableness of setting parameters.

3.2.8.8 blc_ll_setScanEnable

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.11 LE Set Scan Enable command.

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

The scan_enable parameter type has the following two 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.

For the state machine changes of Enable/Disable Scanning, please refer to "3.2.2.2 Link Layer State Combination".

The filter_duplicate parameter type has the following two 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 will only report the adv report event to the Host once; when filter_duplicate is 0, duplicate packet filtering is not enabled, and the adv packet scanned is always reported to the Host.

The return value ble_sts_t is shown in the following table:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success

After setting scan_type to active scan (3.2.8.7 blc_ll_setScanParameter) and Enable Scanning, read scan_rsp once for each device and report it to 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 the scan_rsp of the same device multiple times, you can set Enable Scanning repeatedly by blc_ll_setScanEnable, because each Enable/Disable Scanning, the scan_rsp list of the device will be cleared to 0.

3.2.8.9 blc_ll_createConnection

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.8.12 LE Create Connection command.

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

(1) scan_inetrval/scan window

The scan_interval/scan_window is not handled in this API for now. If you need to set it, you can use "3.2.8.7 blc_ll_setScanParameter".

(2) initiator_filter_policy

Specifies the policy 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 followed by adr_type/mac.

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

(4) own_adr_type

Specifies the type of MAC address used by the Master role that establishes the connection. 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;
```

OWN_ADDRESS_PUBLIC indicates that the public MAC address is used when connecting, and the actual address comes from the setting of the API blc_llms_initStandby_module(mac_public) when the MAC address is initialized.

OWN_ADDRESS_RANDOM indicates that the random static MAC address is used when connecting, which is derived 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 four parameters specify the connection parameters of the Master role after the connection is established, at the same time, these parameters will also be sent to the Slave through the connection request, and the Slave will be the same connection parameters.

The conn_min/conn_max specifies the range of conn interval in 1.25 ms. If appMaxMasterNum > 1, the conn_min/conn_max parameter is invalid, the Master role conn interval in the SDK is fixed to 25 by default(the actual interval is 31.25 ms = 25 × 1.25 ms), in which case the setting can be changed by calling blc_ll_setAclMasterConnectionInterval before establishing the connection; if appMaxMasterNum is 1, the value of conn_max is used directly for the Master role conn interval in the SDK.

The conn_latency specifies connection latency, generally set to 0.

The timeout specifies connection supervision timeout with a unit of 10 ms.

(6) ce_min/ ce_max

Telink BLE Multiple Connection SDK does not handle ce_min/ce_max yet.

Return value list:

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

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

3.2.8.10 ble_ll_setCreateConnectionTimeout

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

The return value is BLE_SUCCESS, and the unit of timeout_ms is ms.

After blc_ll_createConnection is triggered to enter the Initiating state, if the connection cannot be established for a long time, it will trigger Initiate timeout and exit the Initiating state.

The default Initiate timeout of Telink BLE Multiple Connection SDK is 5 seconds. If the User does not want to use this default time, they can call blc_ll_setCreateConnectionTimeout to set the initiate timeout they need.

3.2.8.11 blc_ll_setAclMasterConnectionInterval

```
ble_sts_t   blc_ll_setAclMasterConnectionInterval(u16 conn_interval);
```

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

This API sets the master base connection interval benchmark, through which the timing of multiple masters can be staggered, ensuring that the timing does not conflict when multiple masters are connected at the same time, and improving the efficiency of data transmission. The actual master connection interval in effect is 1/2/3/4/6/8/12 times this benchmark.

3.2.8.12 blc_ll_setAutoExchangeDataLengthEnable

```
void   blc_ll_setAutoExchangeDataLengthEnable(int auto_dle_en)
```

auto_dle_en: 0, Disables the stack to automatically interact with DLE; 1, stack actively interacts with DLE.

By default, if the length of the initialized DLE is not 27, stack will automatically perform DLE interaction after connection. If users do not want stack active interaction, he can use this API to turn it off. When interaction is needed, the user calls the relevant API blc_ll_sendDateLengthExtendReq to interact.

Note:

> If you want to prevent stack automatic interaction DLE, you need to call this API when initialization.

3.2.8.13 blc_ll_sendDateLengthExtendReq

```
ble_sts_t blc_ll_sendDateLengthExtendReq (u16 connHandle,  u16 maxTxOct)
```

connHandle: specify the connection that needs to update the connection parameters.

maxTxOct: the length of the DLE to be set.

The possible results returned by the return type ble_sts_t are shown in the following table:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success

The return value ble_sts_t is only BLE_SUCCESS, the API will not carry out parameter reasonableness check, the user needs to pay attention to the reasonableness of setting parameters.

3.2.8.14 blc_ll_setDataLengthReqSendingTime_after_connCreate

```
void blc_ll_setDataLengthReqSendingTime_after_connCreate(int time_ms)
```

Set the pending time.

Used to set the interaction to perform DLE after waiting time_ms (unit: milliseconds) after connecting.

3.2.8.15 blc_ll_disconnect

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

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

The connHandle is the handle value of the current connection.

Reason is the reason for disconnection, please refer to Bluetooth Core Specification V5.3, Vol 1, Part F, 2 Error code descriptions for details of the reason setting.

If termination is not caused by system operation exception, the application layer generally specifies the reason as HCI_ERR_REMOTE_USER_TERM_CONN = 0x13，blc_ll_disconnect(connHandle, HCI_ERR_REMOTE_USER_TERM_CONN).

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

In general, a direct call to this API can successfully send terminate and disconnect, but there are some special cases that will cause the API call to fail, according to the return value ble_sts_t can understand the corresponding error cause. It is recommended that when the application layer calls this API, check whether the return value is BLE_SUCCESS.

The return value is as follows:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
HCI_ERR_UNKNOWN_CONN_ID	0x02	connHandle error or cannot find the corresponding connection
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x3E	A large amount of data is being sent, the command cannot be accepted at the moment

3.2.8.16 rf_set_power_level_index

Telink BLE Multiple Connection SDK provides an API for setting the energy of BLE RF packet, different chips have different function prototypes.

The API prototype of b85m is as follows:

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

The level value setting of B85m refers to the enumeration variable RF_PowerTypeDef defined in drivers/8258(8278)/rf_drv.h.

The API prototype of b91 is as follows:

```
void rf_set_power_level_index (rf_power_level_index_e idx);
```

The idx setting of B91 refers to the enumeration variable rf_power_level_index_e defined in drivers/B91/rf.h.

This RF packet-sending energy set by this API is valid for both advertising packets and connection packets, and can be set anywhere in the program, the actual packet-sending energy is based on the most recent setting in time.

Note:

> The rf_set_power_level_index function internally sets some registers related to MCU RF, and once the MCU enters sleep (including suspend/deepsleep retention), the values of these registers will be lost. So the user needs to pay attention that this function has to be set again after each sleep wake-up. For example, the BLT_EV_FLAG_SUSPEND_EXIT event callback is used in the SDK demo to ensure that the rf power is reset every time the suspend wakes up.

3.2.8.17 Whitelist & Resolvinglist

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

Whether the peer device address type is RPA (resolvable private address) can be determined by peer_addr_type and peer_addr. Use the following macro to determine.

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

Only non-RPA addresses can be stored in whitelist. Currently, the whitelist in the Telink BLE Multiple Connection SDK 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 whitelist, return a list of values:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
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 Whitelist, and the return value is BLE_SUCCCESS.

RPA (resolvable private address) device needs to use ResolvingList. In order to save ram usage, currently

the Resolvinglist in the Telink BLE Multiple Connection SDK can store 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 parsing and use, if you want to use Resolvelist to parse the address, you must open the enable. You can disable it when you do not need to parse.

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

Add the device that uses the RPA address, and fill in the identity address and irk declared by the peer device for peerIdAddrType/ peerIdAddr and peer-irk, this information will be stored in Flash during the pairing encryption process, the user can find the interface to get this information in "3.4.4 SMP", for local_irk, the SDk is not handled for now, just fill in NULL.

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

Delete the previously added device. Return value BLE_SUCCESS.

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

Define macros in vendor/b85m_feature/app_config.h:

```
#define FEATURE_TEST_MODE TEST_WHITELIST
```

3.3 Host Controller Interface

HCI (Host Controller Interface) is the bridge between Host and Controller, it defines the various types of data that Host and Controller interact with, such as CMD, Event, ACL, SCO, ISO, etc. HCI makes it possible to implement Bluetooth Host and Controller on different hardware platforms. For details of the HCI, see Bluetooth Core Specification V5.3, Vol 4: Host Controller Interface.

HCI Transport is the transport layer of HCI and is responsible for the transport of data of the various data types of HCI. HCI Transport defines the Type Indicator for the different data types of HCI, as shown in the figure below.

HCI packet type	HCI packet indicator
HCI Command Packet	0x01
HCI ACL Data Packet	0x02
HCI Synchronous Data Packet	0x03
HCI Event Packet	0x04

Figure 3.26: HCI Packet Type Indicator

3.3.1 HCI Software Structure

The architecture of the HCI software in the Telink BLE Multiple Connection SDK is shown in the figure below; HCI Transport is the software implementation of the HCI Transport Layer, the source code of which is completely open to users; Controller HCI mainly implements the parsing and processing of HCI CMD and HCI ACL, and generates HCI Events, and provides functional interfaces for HCI Transport. This section will describe the use of Telink HCI in detail around the HCI Transport and Controller HCI interfaces.

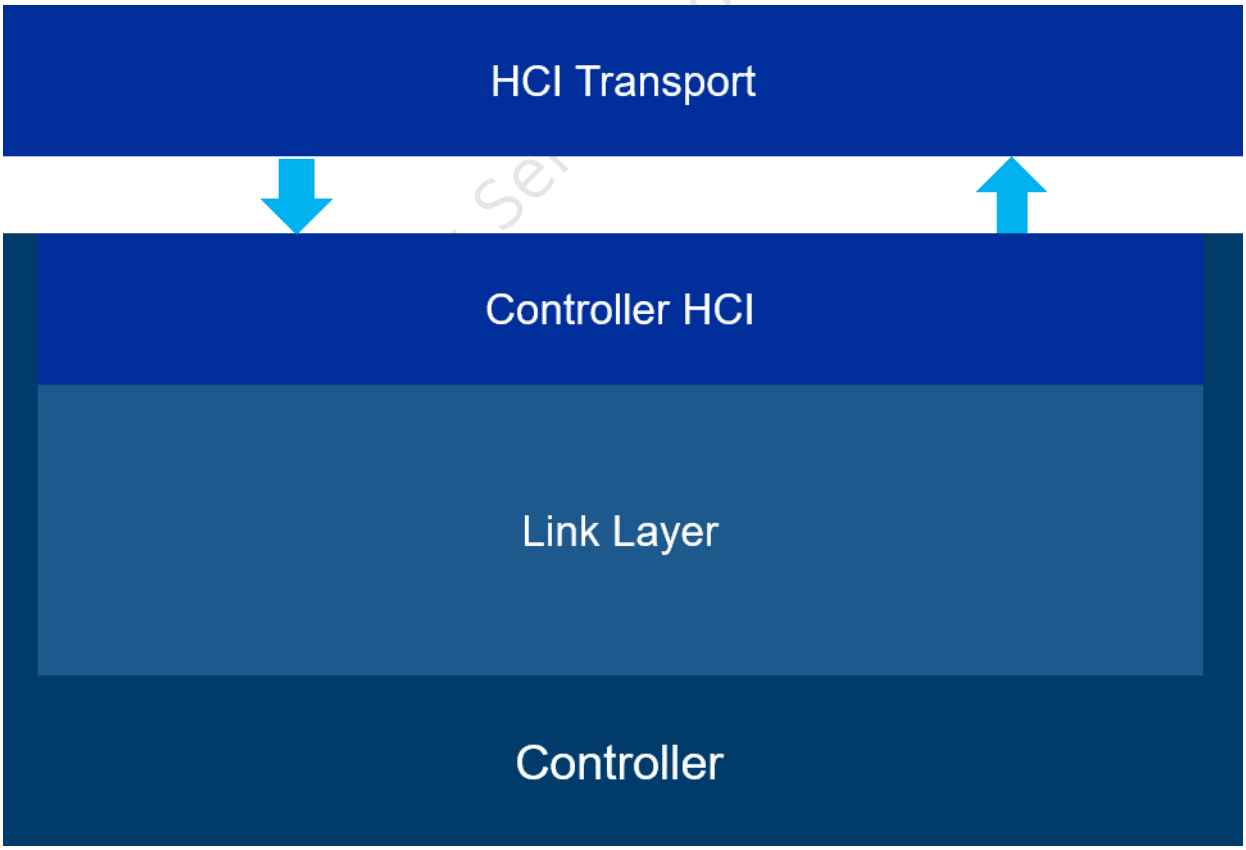


Figure 3.27: Telink HCI Software Structure

HCI Transport is used to transport HCI protocol packets, it does not need to parse HCI protocol packets, it

only needs to receive HCI protocol packets according to HCI Type and then hand them over to Controller HCI for processing. HCI Transport supports various hardware interfaces, such as USB, UART, SDIO, etc., among which UART is commonly used. The Telink BLE Multiple Connection SDK currently only provides HCI Transport for the UART interface, the software implementation of Telink BLE HCI transport can be found in the vendor/common/hci_transport folder of the SDK.

HCI UART Transport supports two protocols, H4 and H5, both of which are supported by the Telink BLE Multiple Connection SDK and are available as open source code. The software architecture of the Telink SDK HCI Transport is shown in the figure below.

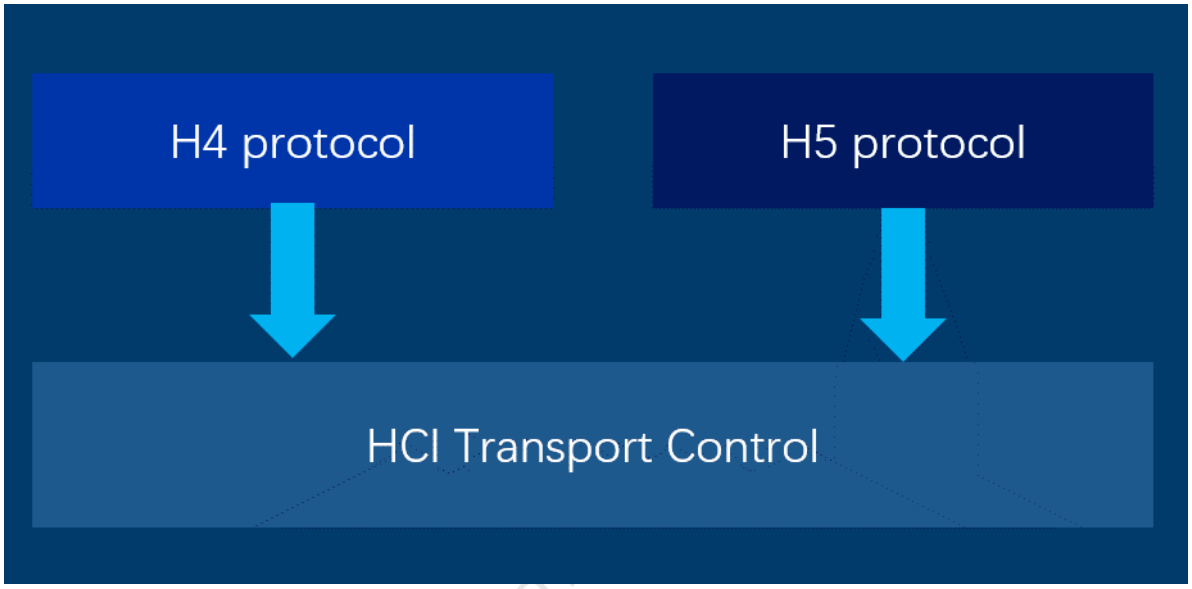


Figure 3.28: Telink HCI Transport Software Structure

H4 Protocol is the software implementation of the HCI UART Transport H4 protocol; H5 Protocol is the software implementation of the HCI UART Transport H5 protocol; HCI Transport Control is the configuration management layer of HCI Transport and provides everything a user needs to use HCI Transport, so users using HCI Transport need only focus on this layer.

3.3.1.1 H4 Protocol

3.3.1.1.1 H4 PDU

The H4 PDU format is shown in the figure below.

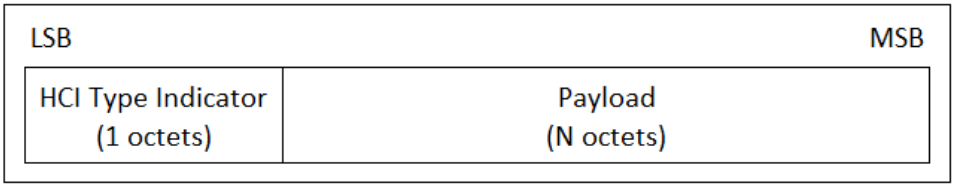


Figure 3.29: H4 PDU format

The H4 PDU consists of the HCI Type Indicator and Payload. The HCI Type Indicator specifies the content of the Payload, and the value of the HCI Type Indicator as shown below; the Payload can be Protocol packets such as HCI CMD, HCI ACL, HCI Event, etc.

The Host sends H4 PDUs to the Controller, and the H4 Protocol software implements the receiving and parsing of H4 PDUs, which can be found in the hci_tr_h4.c and hci_tr_h4.h files in the vendor/common/hci_transport folder.

The user needs to configure the UART Rx Buffer size and the number of buffers according to the requirements when using the H4 protocol software. This can be configured via the macros HCI_H4_TR_RX_BUF_SIZE and HCI_H4_TR_RX_BUF_NUM in the hci_tr_h4.h file. In fact, the SDK calculates the H4 UART Buffer Size automatically for the user's convenience. the user only needs to configure the macro HCI_TR_RX_BUF_SIZE in the hci_tr.h file. HCI_H4_TR_RX_BUF_NUM does not normally need to be modified by the user, unless there is a special requirement.

3.3.1.1.2 H4 API

H4 Protocol provides 3 API:

```
void HCI_Tr_H4Init(hci_fifo_t *pFifo);
void HCI_Tr_H4RxHandler(void);
void HCI_Tr_H4IRQHandler(void);
```

void HCI_Tr_H4Init(hci_fifo_t *pFifo)

Function: This function is the initialization of H4, including UART, RX Buffer, etc.

Parameter:

Parameter	Description
pFifo	Points to the Controller HCI Rx FIFO, which is used to store the received and parsed HCI protocol packets for processing by the Controller. This parameter is provided by the Controller HCI.

Description: This function is not normally called by the user, it is called by the HCI Transport Control layer.

void HCI_Tr_H4RxHandler(void)

Function: This function implements the parsing and processing of H4 PDUs.

Parameter: none.

Description: This function is not normally called by the user, it is called by the HCI Transport Control layer.

void HCI_Tr_H4IRQHandler(void)

Function: This function implements the UART RX/TX interrupt handling.

Parameter: none.

Description: This function is not normally called by the user, it is called by the HCI Transport Control layer.

3.3.1.2 H5 Protocol

H5, also known as 3wire UART, supports software flow control and retransmission mechanisms. H5 has higher reliability than H4, but is not as efficient as H4. See Bluetooth Core Specification V5.3, Vol 4, Part D: Three-wire UART Transport Layer for details.

H5 PDUs (Protocol Data Units) need to be encoded before transmission and decoded before the H5 PDUs can be parsed, the encoding and decoding of H5 PDUs is done by Slip Layer.

The Telink H5 software architecture is shown as follows: UART is used for sending and receiving data; Slip layer implements the encoder and decoder of H5 PDUs, which is responsible for encoding and decoding H5 PDUs; H5 Handler implements the parsing and processing of H5 PDUs, creation of H5 Link, and traffic control and retransmission control. Users can view the H5 implementation in the hci_tr_h5.c, hci_slip.c and hci_h5.c files in the vendor/common/hci_transport folder.

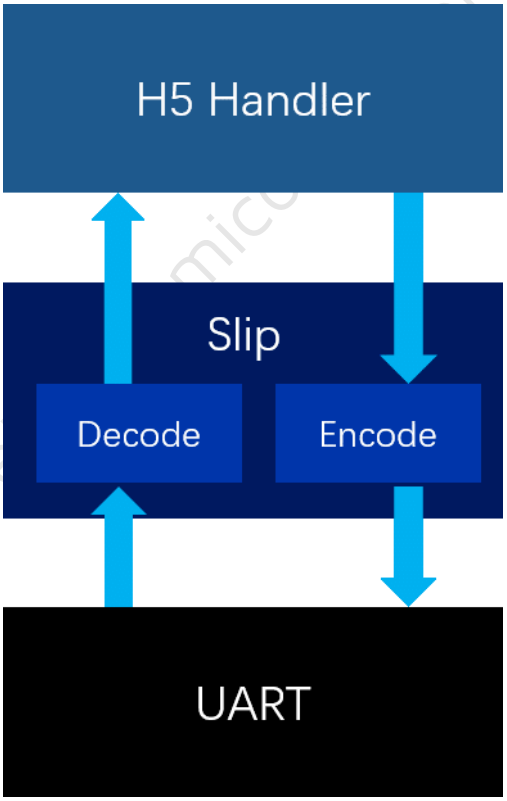


Figure 3.30: H5 Software Architecture

3.3.1.2.1 Slip Layer

(1) Slip Encode

Slip layer encoding will place a byte of C0 at the beginning and end of each H5 packet, all C0s appearing in the H5 packet will be encoded as DB DC sequences; all DBs appearing in the H5 packet will be encoded as DB

DD sequences; here DB DCs and DB DDs are called Slip's escape sequences, and all Slip's escape sequences start with DBs. The table of Slip's escape sequences is shown below.

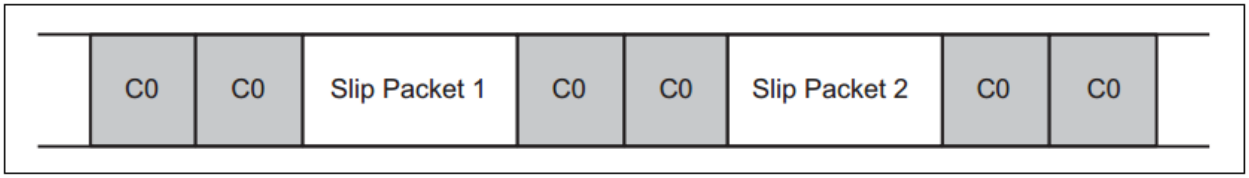


Figure 3.31: HCI Slip packet

SLIP Escape Sequence	Unencoded form	Notes
0xDB 0xDC	0xC0	
0xDB 0xDD	0xDB	
0xDB 0xDE	0x11	Only valid when OOF Software Flow Control is enabled
0xDB 0xDF	0x13	Only valid when OOF Software Flow Control is enabled

Figure 3.32: HCI Slip Sequence list

The encoding of the Slip in the Telink BLE Multiple Connection SDK is done via the API HCI_Slip_EncodePacket(u8 *p, u32 len). The encoded data will be stored in the Slip Encode buffer. The Encode Buffer Size of the Slip can be set by the macro HCI_SLIP_ENCODE_BUF_SIZE, which is automatically calculated by the SDK for the convenience of the user. It does not need to be configured by the user. The user only needs to configure the macro HCI_TR_RX_BUF_SIZE in the vendor/common/hci_transport/hci_tr.h file.

(2) Slip Decode

Once the Slip packet is received, a complete Slip packet is obtained by using the Slip packet start and end flags C0. Then the DB DC and DB DD sequences are converted to C0 and DB according to the Slip escape byte table, so that the Slip is decoded and the H5 PDU is parsed next.

The decoding of the Slip in Telink BLE Multiple Connection SDK is implemented by void HCI_Slip_DecodePacket(u8 *p, 32 len) and the decoded data is stored in the Slip Decode Buffer. The size of the Slip Decode Buffer can be set by the macro HCI _SLIP_DECODE_BUF_SIZE. For user convenience, the SDK has implemented automatic calculation of HCI_SLIP_DECODE_BUF_SIZE, which does not require user configuration. The user only needs to configure the macro HCI_TR_RX_BUF_SIZE in the vendor/common/hci_transport/hci_tr.h file.

3.3.1.2.2 H5 Handle

The H5 Handler implements the parsing and processing of H5 PDUs, H5 Link creation and traffic control and retransmission control.

(1) H5 PDU

H5 PDU (Protocol Data Unit). The H5 PDU contains 3 segments, Packet Header, Payload and an optional Data Integrity Check.

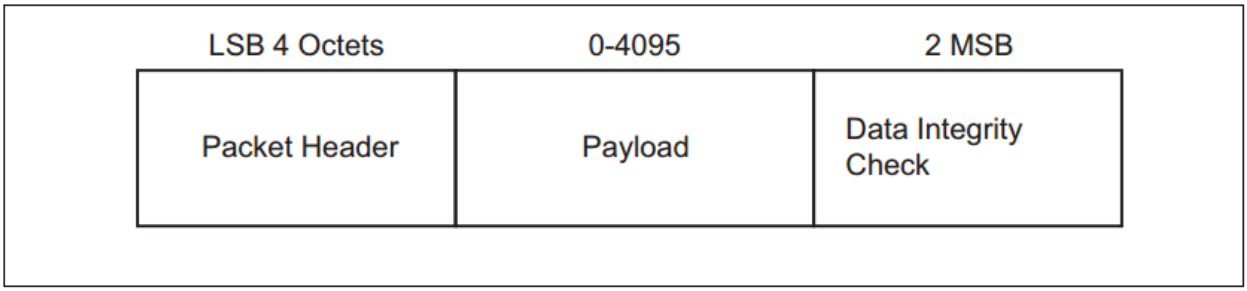


Figure 3.33: H5 PDU

The H5 PDU Header is constructed as follows.

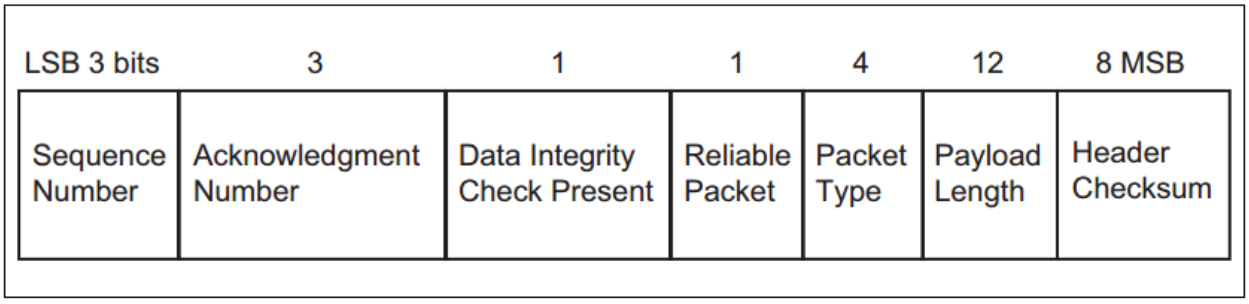


Figure 3.34: H5 PDU Header

Sequence Number(SEQ): For unreliable packets, SEQ should be set to 0. For reliable packets, SEQ indicates the serial number of the packet. For each new packet received, SEQ should be added by 1 . The range of SEQ is 0 ~ 7. SEQ remains unchanged to indicate retransmission.

Acknowledgment Number(ACK): ACK should be set to the next packet sequence number expected by the device, in the range 0 ~ 7.

Data Integrity Check Present: Set to 1 to indicate that the PDU's Payload segment needs to be CRC-checked, that is, the Data Integrity Check in the PDU is present, otherwise it is not present.

Reliable Packet: Set to 1, use reliable transmission and SEQ will take effect.

Packet Type: H5 defines 8 packet types.

HCI Packet Type	Packet Type
Acknowledgment packets	0
HCI Command packet	1
HCI ACL Data packet	2
HCI Synchronous Data packet	3
HCI Event packet	4
HCI ISO Data packet	5
Vendor Specific	14
Link Control packet	15
Reserved for future use	All other values

Figure 3.35: H5 Packet Type

Payload length: The length of the payload segment in the PDU.

Header CheckSum: The sum check value of the H5 PDU Header.

H5 Handler implements the parsing and processing of H5 PDUs through the function void HCI_H5_PacketHandler(u8 *p, u32 len), which is a function used internally by H5 and does not need to be called by the user.

(2) H5 Link establish and configuration information exchange

The Host and Controller need to establish an H5 connection and negotiate configuration information before exchanging H5 packets. The H5 link is established using the SYNC message, SYNC_RSP message, CONFIG message and CONFIG_RSP message.

Initially, the H5 is in the Uninitialized state and keeps sending SYNC messages. When the SYNC_RSP message is received, the H5 enters the Initialized state and keeps sending the CONFIG message. When the CONFIG_RSP message is received, the H5 enters the Active state. At this point the H5 link is successfully established and data can be sent and received.

Telink H5 initially sends a SYNC message at 250 ms intervals and enters the Initialized state when the CONFIG_RSP message is received and sends a CONFIG message at 250 ms intervals and enters the Connected state when the CONFIG_RSP message is received.

The CONFIG message and the CONFIG_RSP message contain the connection parameters used by the Host and the Controller, with the common parts being the final connection parameters. The configuration information for H5 connections mainly includes Sliding Window Size, OOF Flow Control, Data Integrity Check Type and Version.

Sliding Window Size: Sets the maximum number of packets that do not require an immediate ACK. When Sliding Window Size = 1, it means that after the Controller sends a packet, it must wait for the Host ACK before transmitting the next packet. When Sliding Window Size = N (N>1), the Controller can send N packets without waiting for an ACK from the Host, but after the Controller sends the Nth packet, it must wait for an ACK from the Host before sending other packets. The purpose of Sliding Window Size is to improve the

efficiency of H5 transmission. Currently the Telink SDK only supports the case where Sliding Window = 1, and the case where Sliding Window > 1 is easily extended.

OOF Flow Control: Enabling software flow control, this is not commonly used and therefore will not be detailed.

Data Integrity Check Type: Set to 1, the segments associated with Data Integrity Check in the H5 PDU will all be enabled.

Version: Set the H5 (3 wire UART) version. Currently it is v1.0.

In Telink BLE Multiple Connection SDK, users can configure the H5 connection parameters with the following macros.

```
#define HCI_H5_SLIDING_WIN_SIZE         1
#define HCI_H5_OOF_FLW_CTRL             HCI_H5_OOF_FLW_CTRL_NONE
#define HCI_H5_DATA_INTEGRITY_LEVEL     HCI_H5_DATA_INTEGRITY_LEVEL_NONE
#define HCI_H5_VERSION                  HCI_H5_VERSION_V1_0
```

(3) H5 data interaction and retransmission

Once an H5 connection has been established between the Host and Controller, the packets can be exchanged between them. H5 supports flow control and retransmission mechanisms, which are implemented through the SEQ and ACK fields in the H5 PDU Header. The following diagram shows an example of H5 data interaction and retransmission.

Device A sends a SEQ of 6 and an ACK of 3 to Device B. A SEQ of 6 means that Device B expects packet number 6 and an ACK of 3 means that Device A expects the next packet number to be 3. Device B receives a packet from device A and sends a SEQ of 3 and an ACK of 7 to device A. The SEQ of device B is 3 because device A expects a packet of 3; the ACK of device B is 7 because device B has received a packet with serial number 6 and expects a new packet of 7 and so on.

Device A sends a packet with SEQ 0 and ACK 5 to Device B, but for some reason Device B does not receive this packet. Device B will retransmit the previous packet after some time because it has not received the packet from Device A.

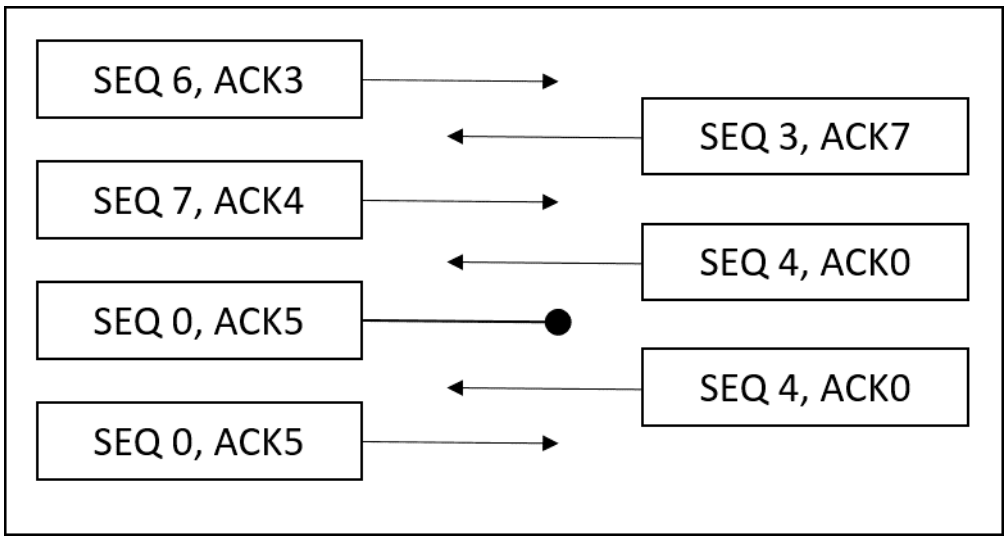


Figure 3.36: H5 Data flow

Flow control:

After the Host and Controller have established an H5 connection, the main interaction is between Data packets and pure ACK packets, which are packets with an H5 Packet Type of 0. Under normal circumstances, the device on the other end sends a Data packet, the local device replies with a Data packet, and then the process continues, but there is always a time when the device on the other end and the local device have no Data packets to send, so the device can send a pure ACK packet instead of a Data packet. For example, if the host enables scan, the controller will keep reporting adv reports, the controller will have a constant stream of Data packets, but the Host does not have a large number of Data packets to send, if the controller sends data packets to the host, but does not receive a reply from the host, then it will cause the controller to keep retransmitting, the host will never receive a new adv report. The host needs to send a pure ACK packet instead of a Data packet, which is equivalent to a reply to the controller, so that the host will keep receiving new adv reports.

Retransmission:

There are several cases of retransmission: after the local device sends a data packet, if it does not receive a reply (data packet or pure ACK packet) from the other device before the timeout is reached, the local device will resend; if the ACK value in the data packet or pure ACK packet received from the other device indicates that a local retransmission is required, the local device will resend. The SEQ should remain the same.

3.3.1.2.3 H5-related APIs

H5 has two important APIs.

```
void HCI_H5_Init(hci_fifo_t *pHciRxFifo, hci_fifo_t *pHciTxFifo);
void HCI_H5_Poll(void);
```

void HCI_H5_Init(hci_fifo_t *pHciRxFifo, hci_fifo_t *pHciTxFifo)

Function: This function is used to initialize H5.

Parameters

Parameter	Description
pHciRxFifo	Point to Controller HCI Rx FIFO
pHciTxFifo	Ppint to Controller HCI Tx FIFO，H5 will take over the HCI Tx FIFO and does not need to be managed by the user, reducing the ease of use. Note: This function does not normally need to be called by the user, it is called by the HCI Transport Control layer.

Description: This function is not normally called by the user, it is called by the HCI Transport Control layer.

void HCI_H5_Poll(void)

Function: This function is used to manage the parsing, sending, resend and flow control of H5 packets.

Parameter: none.

Description: This function is not normally called by the user, it is called by the HCI Transport Control layer.

3.3.1.3 HCI Transport Control

HCI Transport Control is the centralized management layer of Telink HCI Transport, it is the bridge between HCI Transport and Controller HCI, and it also provides the macros and APIs needed for the user to configure and use HCI Transport. For those using the Telink Controller project, it is only necessary to focus on the HCI Transport Control layer. The macros and APIs provided by HCI Transport Control can be found in hci_tr.h in the Controller project.

3.3.1.3.1 HCI Transport Configuration

HCI Transport Control provides a range of configuration macros, which are described in detail below.

The user can select the transport protocol to be used by using the following macros. The Telink BLE Multiple Connection SDK uses HCI_TR_H4 by default.

```
/*! HCI transport layer protocol selection. */
#define HCI_TR_H4            0
#define HCI_TR_H5            1
#define HCI_TR_USB           2 /*!< Not currently supported */
#define HCI_TR_MODE          HCI_TR_H4
```

The user can set the maximum Size of the UART buffer on the Transport Rx Patch and Tx Path with the following macros. HCI_TR_RX_BUF_SIZE should be set to the size of the maximum possible HCI packet to be received; HCI_TR_TX_BUF_SIZE should be set to the size of the maximum possible HCI packet to be sent, which applies for both H4 and H5. For example, if the maximum Rx HCI ACL is 27B and the maximum Rx HCI Cmd Payload is 65B, then HCI_TR_RX_BUF_SIZE should be set to 1B (HCI Type length) + 4B (HCI ACL Header Length) + MAX(27, 65) = 70B, and HCI_TR_TX _BUF_SIZE is calculated in the same way.

```
#define HCI_TR_RX_BUF_SIZE    HCI_RX_FIFO_SIZE
#define HCI_TR_TX_BUF_SIZE    HCI_TX_FIFO_SIZE
/*! HCI UART transport pin define */
#define HCI_TR_RX_PIN         GPIO_PB0
#define HCI_TR_TX_PIN         GPIO_PA2
#define HCI_TR_BAUDRATE       (1000000)

/*! HCI UART transport Flow Control. */
#define HCI_TR_FLOW_CTRL_EN  0
#if HCI_TR_FLOW_CTRL_EN
/*** RTS/CTS Pin ***/
#define HCI_TR_RTS_PIN        UART_RTS_PB3
#define HCI_TR_CTS_PIN        UART_CTS_PA3
#endif
```

HCI_TR_RX_PIN, HCI_TR_TX_PIN are used to set the UART Tx/Rx pins. HCI_TR_BAUDRATE is used to set the UART baud rate. HCI_TR_FLOW_CTRL_EN is used to enable the UART hardware flow control.

Regarding the UART baud rate selection, it should be noted that since BLE can use 1M and 2M, the UART baud rate should be matched accordingly, otherwise there may not be enough buffer when transmitting a large amount of ACL data. In addition, when the baud rate is low, increasing the buffer and the number of buffer will consume more RAM, so a good baud rate matching is needed. We recommend that when the BLE rate is 1M, the UART baud rate should be greater than or equal to 1M; when the BLE rate is 2M, the UART baud rate should be greater than or equal to 2M.

3.3.1.3.2 HCI Transport API

In order to make it easier for users, HCI Transport eventually leaves the necessary APIs available to users, who only need to call them when actually using it.

```
void HCI_TransportInit(void);
void HCI_TransportPoll(void);
void HCI_TransportIRQHandler(void);
```

void HCI_TransportInit(void)

Function: This function is a wrapper for the initialisation of the various Transport protocols. This function is required to initialise HCI Transport before the user can use the HCI Transport function.

Parameter: none.

void HCI_TransportPoll(void)

Function: This function is a wrapper around the various Transport protocol task handlers and needs to be called by the user in the main loop.

Parameter: none.

void HCI_TransportIRQHandler(void)

Function: This function is a wrapper for the various Transport protocols using interrupts. The user needs to call this API from within an interrupt.

Parameter: none.

3.3.1.4 Controller HCI

The Controller HCI interface implements the parsing and processing of HCI protocol packets and generates HCI Events, see Bluetooth Core Specification V5.3 Vol4 for details of HCI. This section will explain a few important APIs.

The Controller HCI provides the necessary APIs for users to use.

```
ble_sts_t blc_ll_initHciRxFifo(u8 *pRxbuf, int fifo_size, int fifo_number);
ble_sts_t blc_ll_initHciTxFifo(u8 *pTxbuf, int fifo_size, int fifo_number);
ble_sts_t blc_ll_initHciAclDataFifo(u8 *pAclbuf, int fifo_size, int fifo_number);
int blc_hci_handler(u8 *p, int n);
```

blc_ll_initHciRxFifo(u8 *pRxbuf, int fifo_size, int fifo_number)

Function: This function is used to initialize the HCI Rx FIFO. The HCI Rx FIFO can manage multiple sets of Rx buffer. The HCI Rx FIFO is used to store incoming HCI packets. HCI Rx Buffer needs to be defined by the user at the application level and registered by this function. HCI Rx Buffer is defined in the controller project, you can refer to the app_buffer.c and app_buffer.h files in the project.

Parameter:

Parameter	Description
pRxbuf	Point to Rx buffer
fifo_size	Size of each buffer in the Rx FIFO, must be 16 bytes aligned.
fifo_number	Number of buffer in Rx FIFO, must be exponential power of 2.

Description: The user needs to call when initialization.

blc_ll_initHciTxFifo(u8 *pTxbuf, int fifo_size, int fifo_number)

Function: This function is used to initialize the HCI Tx FIFO, which can manage multiple Tx buffer groups, and to store the HCI Evt and HCI ACL that the controller will send to the Host. The HCI Tx Buffer needs to be defined by the user at the application level and registered with this function. The HCI Tx Buffer is already defined in the controller project, you can refer to the app_buffer.c and app_buffer.h files in the project.

Parameter:

Parameter	Description
pTxbuf	Point to Tx buffer
fifo_size	Size of each buffer in the Tx FIFO, must be 4 bytes aligned.
fifo_number	Number of buffer in Tx FIFO, must be exponential power of 2.

Description: The user needs to call when initialization.

blc_ll_initHciAclDataFifo(u8 *pAclbuf, int fifo_size, int fifo_number)

Function: This function is used to initialize the HCI ACL FIFO, which can manage multiple ACL buffer groups. The HCI ACL FIFO is used to store the ACL data sent from the Host to the Controller. The HCI ACL buffer

needs to be defined by the user at the application level and registered by this function. The HCI ACL Buffer is already defined in the controller project, you can refer to the app_buffer.c and app_buffer.h files in the project.

Parameter:

Parameter	Description
pAclbuf	Point to Acl buffer
fifo_size	Size of each buffer in the Acl FIFO, must be 4 bytes aligned.
fifo_number	Number of buffer in Acl FIFO, must be exponential power of 2.

Description: The user needs to call when initialization.

int blc_hci_handler(u8 *p, int n)

Function: This function is the Controller HCI packet processor and implements the parsing and processing of HCI CMD and ACL packets.

Parameter:

Parameter	Description
p	Point to incoming HCI protocol packets (using H4 PDU format)
n	Not used because of the length information contained in the HCI protocol package.

Description: This function is called by the HCI Transport Control layer and does not normally need to be called by the user.

3.3.2 Controller Project Introduction

The Telink BLE Multiple Connection SDK provides a Controller project for users to interface with other Hosts. This document takes the Controller project in the Telink B85m BLE Multiple Connection SDK as an example to introduce the project file structure as shown below.

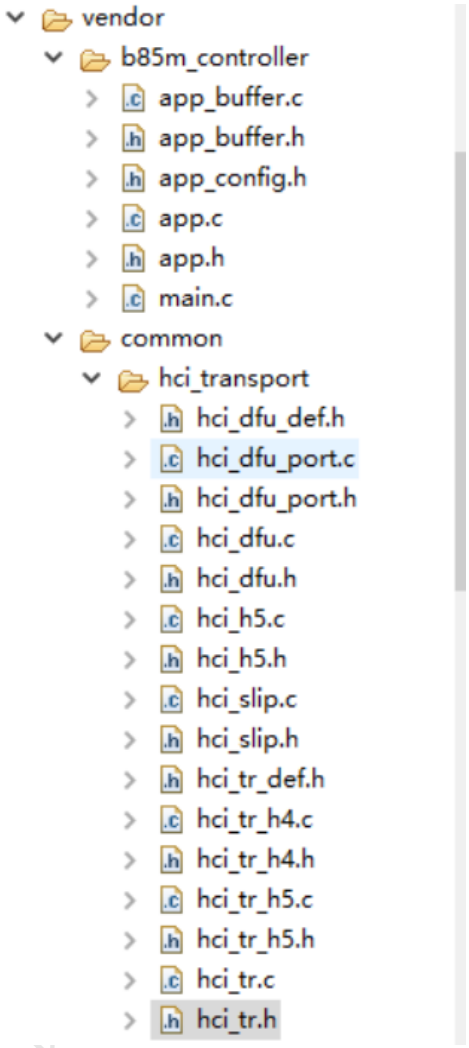


Figure 3.37: Controller project file structure

The Controller project has the same code as the rest of the demo project, except for the HCI part. The demo project is described in the previous sections, this section focuses on the HCI related code.

(1) main.c

The main.c provides the program entry and the global interrupt function entry. The program entry main function implements the initialisation of the hardware and software modules and the while(1) loop; the global interrupt function calls the interrupt handler function of ble stack and the interrupt handler function of HCI Transport, as follows.

```
void irq_handler(void)
{
    blc_sdk_irq_handler ();

 HCI_TransportIRQHandler();
}
```

(2) app.c

The initialization procedures for the Controller HCI interface and HCI Transport are called in the user_init_normal() function in the app.c file, and the HCI Transport task processing is called in the main_loop() function, as follows.

```
void user_init_normal(void)
{
     ......

    /* HCI RX FIFO */
 blc_ll_initHciRxFifo(app_bltHci_rxfifo, HCI_RX_FIFO_SIZE, HCI_RX_FIFO_NUM);
 /* HCI TX FIFO */
 blc_ll_initHciTxFifo(app_bltHci_txfifo, HCI_TX_FIFO_SIZE, HCI_TX_FIFO_NUM);
 /* HCI RX ACL FIFO */
 blc_ll_initHciAclDataFifo(app_hci_aclDataFifo,HCI_ACL_DATA_FIFO_SIZE, HCI_ACL_DATA_FIFO_NUM);

 /* HCI Data && Event call-back */
 blc_hci_registerControllerDataHandler (blc_hci_sendAC
    blc_hci_registerControllerEventHandler(blc_hci_send_data);

 //bluetooth event
 blc_hci_setEventMask_cmd (HCI_EVT_MASK_DISCONNECTION_COMPLETE);
 //bluetooth low energy(LE) event, all enable
 blc_hci_le_setEventMask_cmd( 0xFFFFFFFF );
 blc_hci_le_setEventMask_2_cmd( 0x7FFFFFFF );

     ......
     /* HCI Transport initialization */
 HCI_TransportInit();
 blc_ll_register_user_irq_handler_cb(HCI_Tr_IRQHandler);
}

void main_loop(void)
{
    HCI_TransportPoll();

    ......
}
```

(3) app_buffer.c and app_buffer.h

These two files mainly implement the definition of all the buffer used by the SDK. There are three buffer used by the Controller HCI section, namely app_bltHci_rxfifo[], app_bltHci_txfifo[] and app_hci_aclDataFifo[], the size definitions are controlled by the following macros.

```
#define HCI_TX_FIFO_SIZE    ((2+1+4 + 130 + 4)& ~3)
#define HCI_TX_FIFO_NUM    16
```

```
#define HCI_RX_FIFO_SIZE    ((1+4+ 130 +16) & ~15)
#define HCI_RX_FIFO_NUM    4

#define LE_ACL_DATA_PACKET_LENGTH      (28)
#define HCI_ACL_DATA_FIFO_SIZE  CALCULATE_HCI_ACL_DATA_FIFO_SIZE(LE_ACL_DATA_PACKET_LENGTH)
#define HCI_ACL_DATA_FIFO_NUM   8
```

HCI_TX_FIFO_SIZE: It is used to set the size of HCI Tx Buffer. It should be set to the largest of the maximum possible HCI Event payload and HCI ACL payload plus 4 Bytes of HCI ACL header, 1 Byte of HCI type and 2Bytes of memory used internally by Telink, then 4 Byte alignment. For example: max HCI Event payload = 65B, max HCI ACL payload = 27B, then HCI_TX_FIFO_SIZE = align4(2B(telink) + 1B(HCI Type) + 4B(HCI ACL Header) + MAX(27, 65)) = 72B.

HCI_TX_FIFO_NUM: It is used to set the number of HCI Tx buffer, which must be an exponential power of 2, such as 2, 4, 8...

HCI_RX_FIFO_SIZE: It is used to set the size of HCI Rx buffer. It should be set to the largest of the maximum possible HCI Cmd payload and HCI ACLpaylaod received plus 4Bytes for the HCI ACL header and 1Byte for the HCI Type, then 16 Bytes aligned. For example: max HCI Cmd payload = 65B, max HCI ACL payload = 27B, then HCI_RX_FIFO_SIZE = align16(1B(HCI Type) + 4B(HCI ACL Header) + MAX(27, 65)) = 80B.

HCI_RX_FIFO_NUM: It is used to set the number of HCI Rx buffer, which must be an exponential power of 2, such as 2, 4, 8...

LE_ACL_DATA_PACKET_LENGTH: Used to set the maximum HCI ACL payload, which cannot exceed 252.

HCI_ACL_DATA_FIFO_SIZE: It is used to set the ACL Data Buffer Size, which is automatically calculated by the user by calling the macro CALCULATE_HCI_ACL_DATA_ FIFO_SIZE(LE_ACL_DATA_PACKET_LENGTH) directly.

HCI_ACL_DATA_FIFO_NUM: It is used to set the number of HCI ACL DATA buffer, which must be an exponential power of 2, such as 2, 4, 8...

3.3.3 Controller Event

In order to satisfy the recording and processing of some key actions on the bottom layer of Multiple Connection Ble Stack in the application layer, the SDK provides two types of events as shown below: one is the standard HCI event defined by the BLE controller; the second is the BLE host defined by some protocol stack process interaction event notification type GAP event (can also be considered as host event, please refer to the "3.4.3.2 GAP event" of this document for details). This section mainly introduces Controller event.

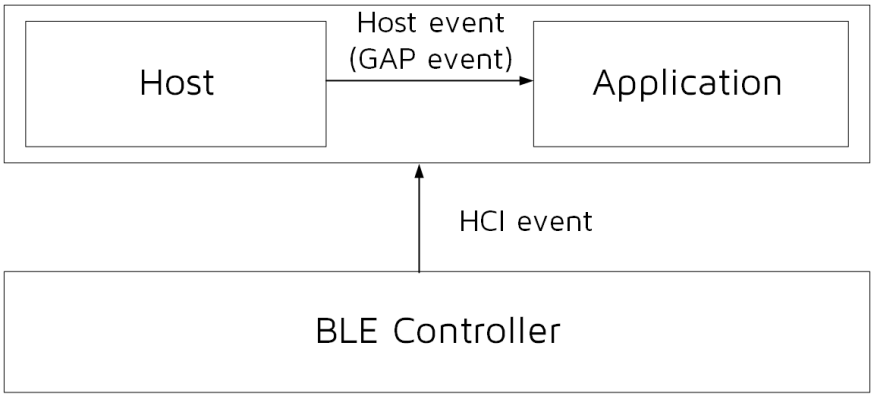


Figure 3.38: BLE SDK Event Structure

Note:

> In the Telink BLE Single Connection SDK, telink provides a set of controller event defined by itself, which are mostly the same as the HCI event specified in the Bluetooth Core Specification, and in the Telink BLE Multiple Connection SDK, the event defined by telink in the repeated part is removed, and the user can use the standard event.

3.3.3.1 Controller HCI Event Classification

The Controller HCI event is designed according to the Bluetooth Core Specification standard.

The Host + Controller architecture is shown in the figure below, the Controller HCI event reports all events of the Controller to the Host through HCI.

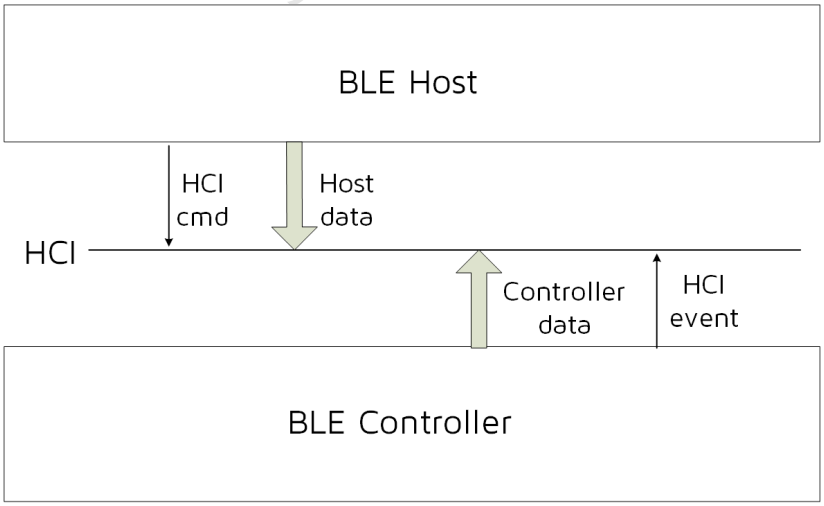


Figure 3.39: Host + Controller architecture

For the definition of Controller HCI event, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.7 Events for details. Among them, the 7.7.65 LE Meta Event refers to HCI LE(low energy) Event, and the others are ordinary HCI events. Telink BLE Multiple Connection SDK also divides Controller HCI event into two categories: HCI Event and HCI LE event. Since Telink BLE Multiple Connection SDK mainly focuses on

low-power Bluetooth, only a few basic HCI events are supported, while the vast majority of HCI LE events are supported.

Controller HCI event related macro definitions, interface definitions, etc. please refer to the header file in the stack/ble/hci directory.

If the user needs to receive Controller HCI event in Host or App layer, first need to register the callback function of Controller HCI event, and then open the mask of the corresponding event, mask open API see below event analysis.

The callback function prototype and registration interface 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 marker, which is used in many places in the lower layer 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 data and data length of the event, which is consistent with that defined in the Bluetooth Core Specification. Users can refer to the following usage in the code and the specific implementation of the app_controller_event_callback function.

```
blc_hci_registerControllerEventHandler(app_controller_event_callback);
```


3.3.3.2 Common Controller HCI event

Most HCI events are supported in the Telink BLE Multiple Connection SDK, and the following are the events that clients may use.

```
#define HCI_EVT_DISCONNECTION_COMPLETE                  0x05
#define HCI_EVT_LE_META                                 0x3E
```

(1) HCI_EVT_DISCONNECTION_COMPLETE

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.7.5 Disconnection Complete event. The data structure pointed to by the callback pointer is as follows:

```
typedef struct {
 u8         status;
 u16        connHandle;
 u8         reason;
} hci_disconnectionCompleteEvt_t;
```

(2) HCI_EVT_LE_META

It indicates that the current event is HCI LE event, and the specific event type is determined according to the following sub event code.

In HCI event, except for HCI_EVT_LE_META (HCI_EVT_LE_META uses blc_hci_le_setEventMask_cmd to open the event mask), all others have to open the event mask through the following API.

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

The definition of event mask is shown below:

```
#define HCI_EVT_MASK_DISCONNECTION_COMPLETE          0x0000000010
```

If the user does not set the HCI event mask through this API, the SDK only opens the mask corresponding to HCI_EVT_MASK_DISCONNECTION_COMPLETE by default, that is, to ensure the reporting of Controller disconnect event.

3.3.3.3 Common 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 understand the following, the 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_PHY_UPDATE_COMPLETE                  0x0C
```

(1) HCI_SUB_EVT_LE_CONNECTION_COMPLETE

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.7.65.1 LE Connection Complete event. The data structure pointed to by the callback pointer is as follows:

```
typedef struct {
 u8         subEventCode;
 u8         status;
 u16        connHandle;
 u8         role;
 u8         peerAddrType;
 u8         peerAddr[6];
 u16        connInterval;
 u16        slaveLatency;
 u16        supervisionTimeout;
 u8         masterClkAccuracy;
} hci_le_connectionCompleteEvt_t;
```

(2) HCI_SUB_EVT_LE_ADVERTISING_REPORT

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.7.65.2 LE Advertising Report event. The data structure pointed to by the callback pointer is as follows:

```
typedef struct {
 u8 subcode;
 u8 nreport;
 u8 event_type;
 u8 adr_type;
 u8 mac[6];
 u8 len;
 u8 data[1];
} event_adv_report_t;
```

After the Link Layer scan of the controller reaches the correct adv packet, it is reported to the Host through HCI_SUB_EVT_LE_ADVERTISING_REPORT.

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

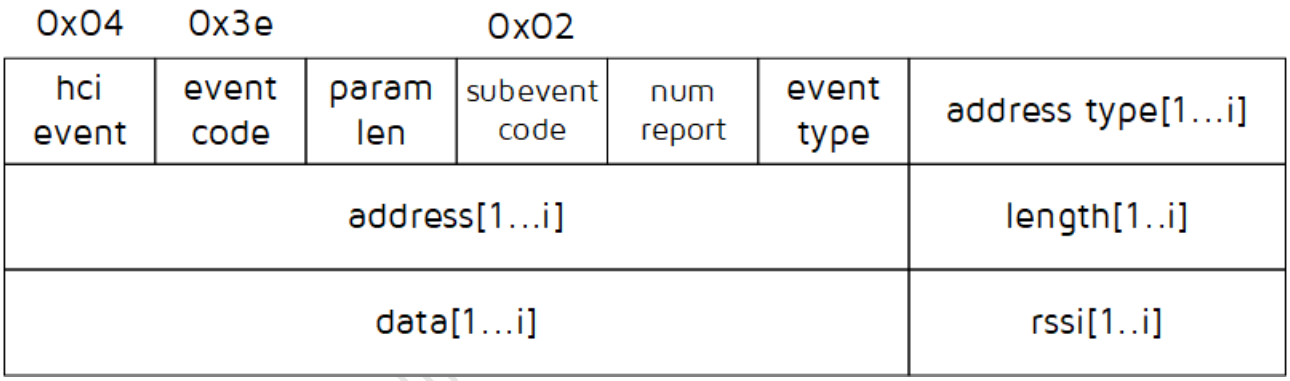


Figure 3.40: ADVERTISING_REPORT event packet format

Note:

> The LE Advertising Report Event in the Telink BLE Multiple Connection SDK only reports one adv packet at a time, that is, i is 1 in the above figure.

(3) HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.7.65.3 LE Connection Update Complete event.

When the connection update on Controller takes effect, report HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE to the Host. The data structure pointed to by the callback pointer is as follows:

```
typedef struct {
 u8         subEventCode;
 u8         status;
 u16        connHandle;
```

```
 u16        connInterval;
 u16        connLatency;
 u16        supervisionTimeout;
} hci_le_connectionUpdateCompleteEvt_t;
```

(4) HCI_SUB_EVT_LE_PHY_UPDATE_COMPLETE

For details, please refer to Bluetooth Core Specification V5.3, Vol 4, Part E, 7.7.65.12 LE PHY Update Complete event.

The data structure pointed to by the callback pointer is as follows:

```
typedef struct {
 u8         subEventCode;
 u8         status;
 u16        connHandle;
 u8       tx_phy;
 u8      rx_phy;
} hci_le_phyUpdateCompleteEvt_t;
```

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

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

The definition of evtMask also corresponds to some given above, and other event users can check 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
```

If the user does not set the HCI LE event mask through this API, all HCI LE events are not open by the SDK by default.

3.4 Host

3.4.1 L2CAP

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

The L2CAP layer of BLE is a simplified version of the classic Bluetooth L2CAP layer, in the basic mode, it does not perform segmentation and recombination, does not involve process control and retransmission mechanism, and only uses fixed channels for communication. The simplified structure of L2CAP is shown

in the figure below, which simply means that the application layer data is subpackaged and sent to the BLE controller, and the data received by the BLE controller is packetized into different CID data and reported to the host layer.

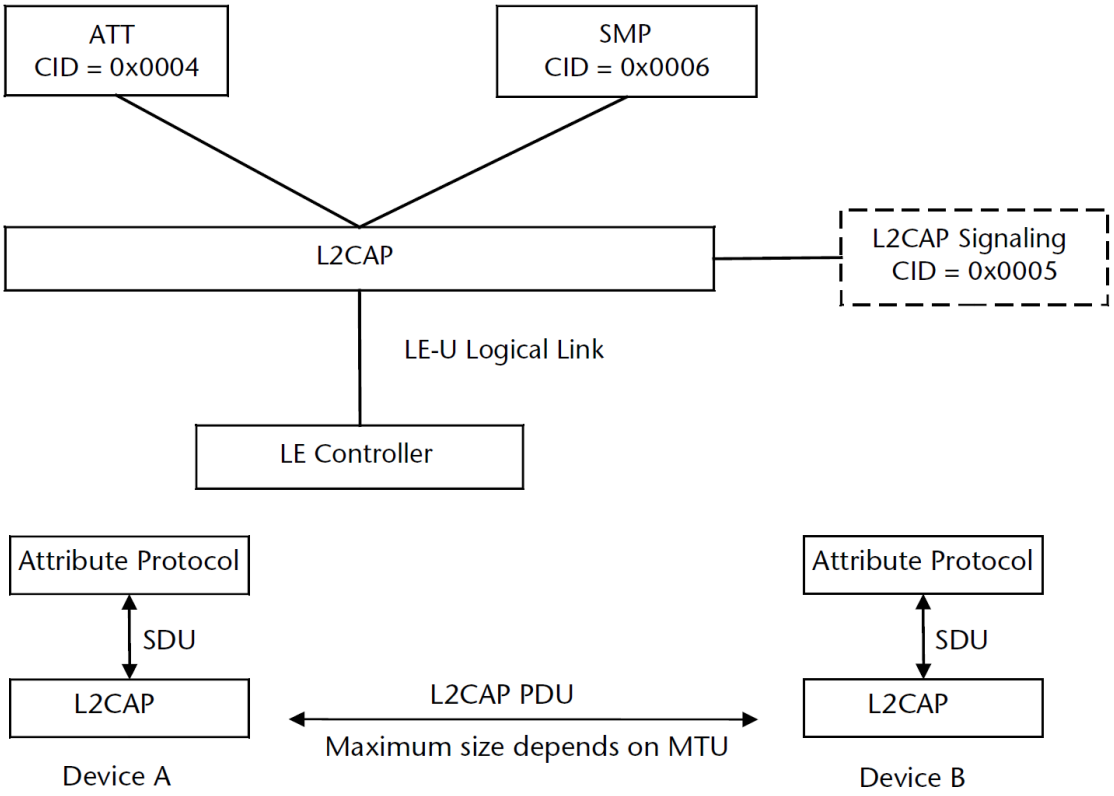


Figure 3.41: BLE L2CAP structure and ATT packet assembly model

L2CAP is designed according to the Bluetooth Core Specification, the main function is to complete the data docking of Controller and Host, most of which is done at the bottom of the protocol stack, and there are very few places where user participates. The user can set it according to the following APIs.

3.4.1.1 Register L2CAP data processing function

In the BLE multiple SDK architecture, the data of the Controller is interfaced with Host through HCI, and data from HCI to Host is first processed at the L2CAP layer, using the following API to register this processing function:

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

The functions of the 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 transmit it upwards to ATT, SIG or SMP.

Initialization:

```
blc_hci_registerControllerDataHandler (blt_l2cap_pktHandler);
```

3.4.1.2 Update connection parameters

(1) Slave request to update connection parameters

In the BLE protocol stack, The Slave applies a set of new connection parameters to the Master through the L2CAP layer CONNECTION PARAMETER UPDATE REQUEST command. The command format is shown below, please refer to Bluetooth Core Specification V5.3, Vol 3, Part A, 4.20 L2CAP_CONNECTION_PARAMETER_UPDATE_REQ (code 0x12) for details.

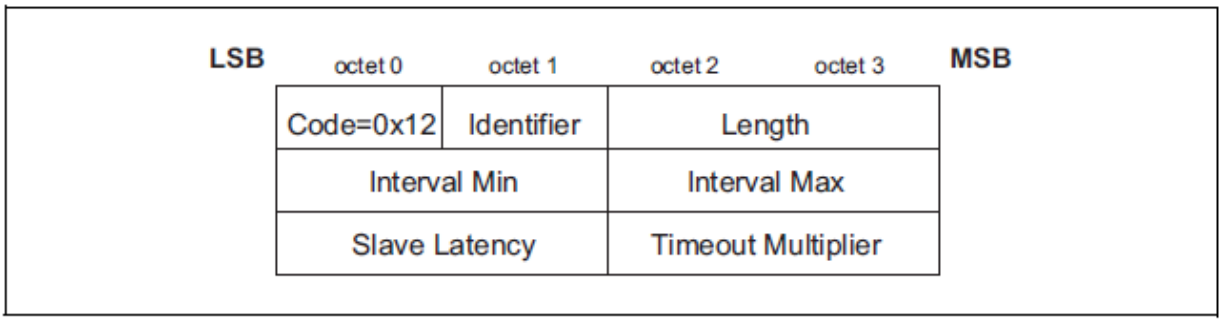


Figure 4.22: Connection Parameters Update Request Packet

Figure 3.42: Connection para update Req format in BLE protocol stack

The Telink BLE Multiple Connection SDK provides an API for the Slave to actively apply for updating connection parameters, which is used to send the CONNECTION PARAMETER UPDATE REQUEST command to the Master.

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

This API is only used by Slave. The unit of min_interval and max_interval is 1.25 ms, and the unit of timeout is 10 ms.

The Telink BLE Multiple Connection SDK provides an API for the Slave to set the time to send requests to update connection parameters:

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

Taking the connection establishment moment as the time reference point, the connection parameter update request will be sent out after time_ms has passed, this API is not called, and the default setting is 1000 ms.

If the API bls_l2cap_requestConnParamUpdate is called after time_ms after the connection is established, the connection parameter update request is sent immediately.

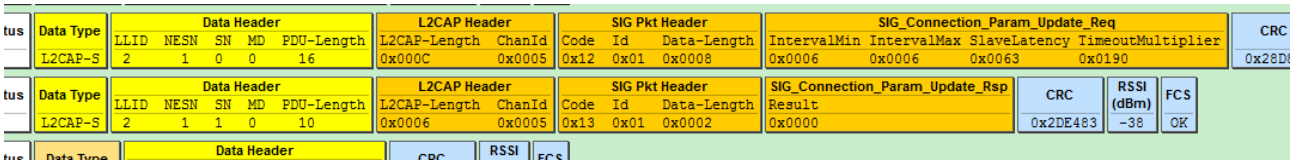


Figure 3.43: BLE sniffer packet sample conn para update reqeust and response

In the application, the SDK provides a gap event "GAP_EVT_L2CAP_CONN_PARAM_UPDATE_RSP" to obtain the connection request result, which is used to notify the user whether the connection parameter request applied by the Slave is accepted or not, as shown in the figure above, the Master accepts the Connection_Param_Update_Req parameter of the Slave.

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_RSP:
        {
            (gap_l2cap_connParamUpdateRspEvt_t*) p= (gap_l2cap_connParamUpdateRspEvt_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) Master responds to update requests

After the peer Slave applies for new connection parameters, the Master receives the command and returns the CONNECTION PARAMETER UPDATE RESPONSE command. For details, please refer to Bluetooth Core Specification V5.3, Vol 3, Part A, 4.21 L2CAP_CONNECTION_PARAMETER_UPDATE _RSP (code 0x13).

Regarding whether the actual Android and iOS devices accept the connection parameters applied by the user, it has to do with the practice of each manufacturer's BLE Master, and the standard is not uniform among them.

In the Telink BLE Multiple Connection SDK, regardless of whether the Slave's parameter request is accepted or not, the following API is used to reply to this request:

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

This API is only available to Master. connHandle specifies the current connection handle, req->id is the Identifier value in the connection parameter update request, and connParaRsp reference is as follows:

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

In the Telink BLE Multiple Connection SDK, after the Master determines the appropriate connection parameter request, if the user has registered GAP_EVT_L2CAP_CONN_PARAM_UPDATE_REQ, determines whether to agree to the connection parameter update to be performed by the user in the event callback, if not registered, the Master will directly enter the connection parameter update process at the bottom layer.

When GAP_EVT_L2CAP_CONN_PARAM_UPDATE_REQ takes effect, if the user does not agree to the connection parameter request, blc_l2cap_SendConnParamUpdateResponse needs to be called in the callback and the third parameter set to CONN_PARAM_UPDATE_REJECT. If the user agrees to the connection parameter request, you need to first call blc_l2cap_SendConnParamUpdateResponse in the callback and set the third parameter to CONN_PARAM_UPDATE_ACCEPT, and then call blm_l2cap_processConnParamUpdatePending to enter the connection parameter update process.

```
void blm_l2cap_processConnParamUpdatePending(u16 connHandle, u16 min_interval, u16 max_interval,
 ↪  u16 latency, u16 timeout);
```

(3) update connection parameters on Link Layer

The Master can perform connection parameter updates directly, or the Slave can send conn para update req, and after the Master replies the conn para update rsp to accept the request, there will be the following process.

The Master will send the LL_CONNECTION_UPDATE_REQ command of the link layer layer, as shown in the following figure.

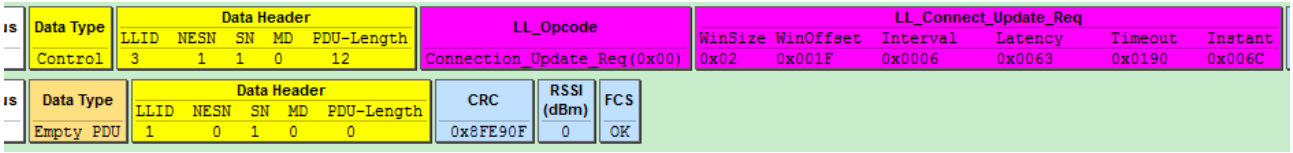


Figure 3.44: Sniffer packet display ll conn update req

After the Slave receives the update request, it updates the connection parameters. Both the Master and the Slave will trigger the HCI event of HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE.

3.4.2 ATT & GATT

3.4.2.1 GATT Basic Unit Attribute

GATT defines two roles: Server and Client. In the Telink BLE Multiple Connection SDK, the Slave device is the Server, and the Android, iOS or Master device is the Client. Server needs to provide multiple services for Client to access.

The essence of GATT's service is composed of multiple Attributes, each of which has a certain amount of information, When multiple Attributes of different kinds are combined together, a basic service can be reflected.

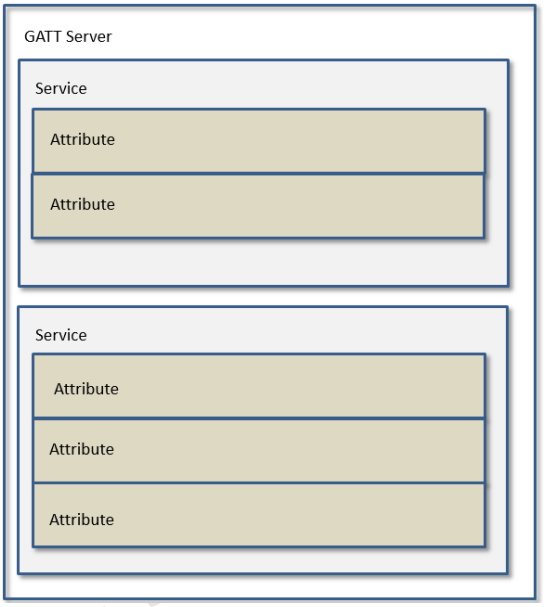


Figure 3.45: Attribute constitutes GATT service

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

(1) Attribute Type: UUID

UUID is used to distinguish the type of each attribute, and its full length is 16 bytes. The UUID length in the BLE standard protocol is defined as 2 bytes, because the peer devices follow the same set of conversion methods to convert the UUID of 2 bytes into 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 distributed in the following files: tack/ble/service/hids.h, stack/ble/attr/gatt_uuid.h

Telink private some profiles (OTA, SPP, MIC, etc.) are not supported in standard Bluetooth. These private UUID are defined in stack/ble/attr/gatt_uuid.h with a length of 16 bytes.

(2) Attribute Handle

The service has multiple Attributes, and these Attributes form an Attribute Table. In the Attribute Table, each Attribute has an Attribute Handle value, which is used to distinguish each different Attribute. After the Slave and Master establish connection, the Master parses and reads the Attribute Table of the Slave through

the Service Discovery process, and corresponds to each different Attribute according to the value of the Attribute Handle, so that the data communication behind them as long as they bring Attribute Handle, the other party will know which Attribute's data.

(3) Attribute Value

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

3.4.2.2 Attribute and ATT Table

In order to implement the Slave's GATT service, the SDK designs an Attribute Table, which consists 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;
```

Combined with the current SDK reference Attribute Table to illustrate the meaning of the above. The Attribute Table code can be found in app_att.c, as shown in the following screenshot:

```
static const attribute_t my_Attributes[] = {

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


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


    // 0008 - 000b gatt
    {4,ATT_PERMISSIONS_READ,2,2,(u8*)(&my_primaryServiceUUID),  (u8*)(&my_gattServiceUUID), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof(my_serviceChangeCharVal),(u8*)(&my_characterUUID),        (u8*)(my_serviceChangeCharVal), 0},
    {0,ATT_PERMISSIONS_READ,2,sizeof (serviceChangeVal), (u8*)(&serviceChangeUUID),     (u8*)(&serviceChangeVal), 0},
    {0,ATT_PERMISSIONS_RDWR,2,sizeof (serviceChangeCCC),(u8*)(&clientCharacterCfgUUID), (u8*)(serviceChangeCCC), 0},


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

Figure 3.46: BLE SDK attribute table

Please note that const is added before the definition of attribute table:

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

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

(1) attNum

attNum has two functions.

The first role of attNum is to indicate the number of all valid Attributes in the current Attribute Table, i.e., the maximum value of Attribute Handle, which is only used in the invalid Attribute in the item 0 of the Attribute Table array:

```
{57,0,0,0,0,0}, // ATT_END_H - 1 = 57
```

attNum = 57 indicates that there are 57 attributes in the current Attribute Table.

In BLE, the Attribute Handle value starts from 0x0001 and increases by one, while the subscript of the array starts from 0, adding the above virtual attribute in Attribute Table makes the subscript number of each attribute in the data equal to the value of its Attribute Handle. After defining the Attribute Table, count the subscript of the Attribute in the current Attribute Table array to know the current Attribute Handle value of the Attribute.

After counting all the Attribute in Attribute Table, the last number is the number attNum of valid Attributes in the current Attribute Table, which is currently 57 in the SDK. If the user adds or deletes Attribute, this attNum needs to be modified, you can refer to the enumeration ATT_HANDLE of vendor/b85m_demo/app_att.h.

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), and the attNum on this Attribute specifies the count from the current Attribute, and there are a total of attNum Attributes that belong to the component of the service.

As shown in the figure above, the gap service UUID is the first Attribute of GATT_UUID_PRIMARY_SERVICE, and its attNum 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 graph is set to 27, the 27 consecutive Attributes from this Attribute are HID services.

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

(2) perm

The perm is the abbreviation of permission.

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

There are 10 permissions as follows, and the permissions of each Attribute must be the following value 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, Telink BLE Multiple Connection SDK does not support authorized read and authorized write.

(3) uuid and uuidLen

As mentioned earlier, there are two types of UUIDs: BLE standard 2 bytes UUID and Telink proprietary 16 bytes UUID. Both UUIDs can be described simultaneously by uuid and uuidLen.

uuid is a u8 type pointer, uuidLen means the content of consecutive uuidLen byte from the beginning of the pointer is the current UUID. Attribute Table is existed on flash, all the UUID is also existed on flash, so uuid is a pointer to flash.

a) BLE standard 2 bytes UUID:

If Attribute Handle = devNameCharacter attribute of 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 my_devNameCharVal address in flash, uuidLen is 2, when peer Master comes to read this Attribute, UUID will be 0x2803.

b) Telink's private 16 bytes UUID:

Such as OTA's Attribute, related 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},
```

The uuid points to the address of my_OtaData in flash, uuidLen is 16, when Master comes to read this Attribute, UUID will be 0x000102030405060708090a0b0c0d2b12.

(4) pAttrValue and attrLen

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

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

Attribute Handle=0x0027 hid Information's Attribute, relevant 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, hidInformation 4 bytes 0x01 0x00 0x01 0x11 are read-only and do not involve write operations, so it can be stored on Flash using the const keyword when defined. pAttrValue points to the address of hidInformation on the flash, at this time attrlen takes the value of the actual length of hidInformation. When Master reads the Attribute, it returns 0x01000111 to Master based on pAttrValue and attrLen.

When the Master reads the Attribute, the BLE sniffer packet as shown in the figure below, the Master uses the ATT_Read_Req command, assuming that AttHandle = 0x0023 = 35 is set to be read, which corresponds to the hid information in the Attribute Table in the SDK.

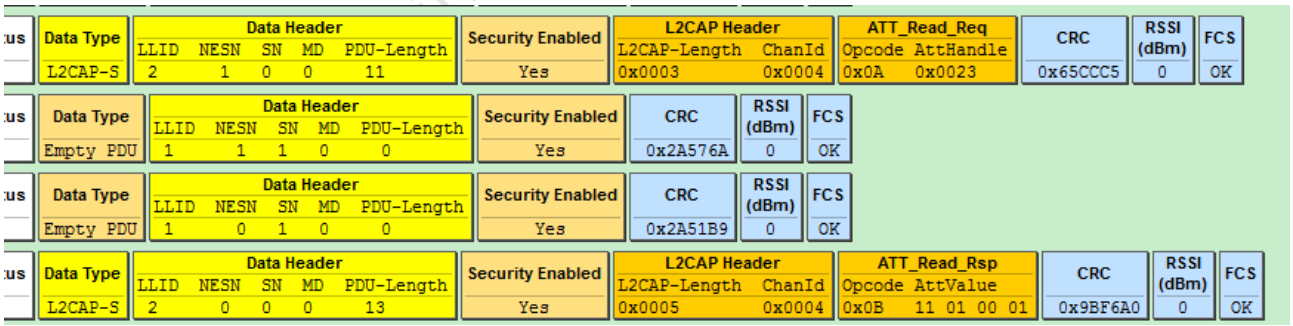


Figure 3.47: BLE sniffer packet sample when Master reads hidInformation

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

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

In practical applications, the my_batVal value which reacts to the current battery level will change according to the power level sampled by the ADC, and then transmitted to Master by Slave active notify or Master

active read, so my_batVal should be placed in the memory, at this time pAttrValue points to the address of my_batVal on RAM.

(5) callback function w

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

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

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

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

a) opcode = 0x12, Write Request.

b) opcode = 0x52, Write Command.

c) 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, and the length of the write is l2capLen-3 in the master data packet format; if the user sets the callback function w, 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. These two write operations are mutually exclusive, and only one can take effect.

The user sets the callback function w to process the Master's Write Request, Write Command and Execute Write Request commands at 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 (for example, pAttrValue points to flash cannot complete the write operation; or the length of attrLen is not enough, the Master's write operation will be out of bounds, causing other data to be incorrectly rewritten).

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 3.48: Write request in BLE stack

3.4.5.3 Write Command

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

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

Figure 3.49: Write command in BLE stack

3.4.6.3 Execute Write Request

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

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

Figure 3.50: Execute write request in BLE stack

The void-type p pointer to 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{
 u8 type;
 u8 rf_len;  //User do not use this member, because it may be changed by stack layer.
 u16 l2capLen;
 u16 chanId;

 u8 opcode;
 u16 handle;

 u8 dat[20];
}rf_packet_att_t;
```

p points to the first element type. The valid length of the written data is l2cap - 3, and the first valid data is dat[0].

```
int my_WriteCallback(u16 connHandle, void * p)
{
 rf_packet_att_t *pw = (rf_packet_att_t *)p;
 int len = pw->l2capLen - 3;
 //add your code
 //valid data is pw->dat[0] ~ pw->dat[len-1]
 return 1;
}
```

The location of the above structure rf_packet_att_t is stack/ble/ble_format.h.

Note:

> The rf_len in the structure rf_packet_att_t should not be used by the user, rf_len may be rewritten when assembling the package, please use l2capLen conversion before use.

(6) callback function r

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

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

If user needs to define a callback read function, it needs to follow the above format. The callback function r is optional, for a specific Attribute, the user can set the callback read function, or not set the callback (when no callback is set, it is represented by a null pointer 0).

The callback function r trigger condition is: when the Attribute Opcode of the Attribute PDU received by Slave is the following two, the Slave will check whether the callback function R is set:

a) opcode = 0x0A, Read Request.

b) opcode = 0x0C, Read Blob Request.

After the Slave receives the above read command:

a) If the user sets the 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, Slave does not reply Read Response/Read Blob Response to 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.

b) If the user does not set the 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 replied to 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.

(7) Attribute Table structure

According to the above detailed description of Attribute, use Attribute Table to construct the Service structure as shown in the following figure. The attnum of the first Attribute is used to indicate the current number of ATT Table Attributes, the remaining Attributes are firstly grouped by Service, and the first Attribute of each group is the declaration of the Service, and the attnum is used to specify how many of the immediately following Attribute belongs to the specific description of the Service. The first one of each group of services is a Primary Service.

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

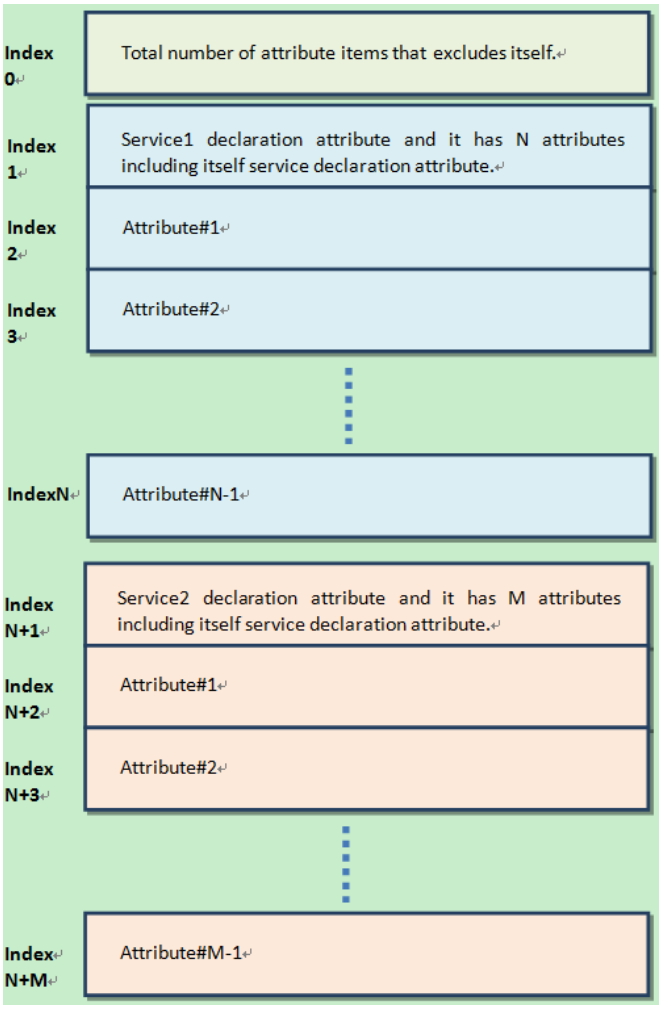


Figure 3.51: Service attribute layout

(8) ATT table Initialization

GATT & ATT initialization only needs to transmit the pointer of the Attribute Table of the application layer to the protocol stack. The API provided:

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

p is the pointer of Attribute Table.

3.4.2.3 GATT Service Security

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

Please refer to the relevant detailed introduction in the "3.4.4 SMP" to understand the basic knowledge of LE pairing method and encryption level, etc.

The picture below is a mapping relationship between the GATT service security level service request given by Bluetooth Core Specification, you can refer to core5.0 (Vol3/Part C/10.3 AUTHENTICATION PROCEDURE) for details.

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

Figure 3.52: Service request response mapping relationship

Users can clearly see:

- The first column is related to whether the currently connected Slave device is in the encrypted state or not;

- The second column (local Device's Access Requirement for service) is related to the permission (Permission Access) setting of the characteristic in the ATT table set by the user, as shown in the following figure;
- The third column is divided into four sub-columns, and these four 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 four):

a) No authentication and no encryption

b) Unauthenticated pairing with encryption

c) Authenticated pairing with encryption

d) Authenticated LE Secure Connections

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

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


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


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


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


#define ATT_PERMISSIONS_AUTHOR_READ         (ATT_PERMISSIONS_READ | ATT_PERMISSIONS_AUTHOR)        //!< Read requires Authorization
#define ATT_PERMISSIONS_AUTHOR_WRITE        (ATT_PERMISSIONS_WRITE | ATT_PERMISSIONS_AUTHEN)       //!< Write requires Authorization
#define ATT_PERMISSIONS_AUTHOR_RDWR         (ATT_PERMISSIONS_RDWR | ATT_PERMISSIONS_AUTHOR)        //!< Read & Write requires Authorization
```

Figure 3.53: ATT permission definition

The final implementation of GATT service security is related to the parameter configuration during SMP initialization, including the highest supported security level setting, the characteristic permission settings 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 of Master is Unauthenticated pairing with encryption, at this time if the authority of a writing characteristic in the ATT table is ATT_PERMISSIONS_AUTHEN_WRITE, then when the Master writes this characteristic, we will reply to the mistake of insufficient encryption level.

The user can set characteristic 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 don 't want to use the method of sending a Security Request to trigger the Master to start pairing after connecting, then the client can set the permissions of some client characteristic configuration (CCC) attributes that have notify attributes to ATT_PERMISSIONS_ENCRYPT_WRITE, then 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.

Note:

> The security level set by the user only represents the highest security level that the device can support, as long as the permissions of the characteristic (ATT Permission) in the ATT table does not exceed the highest level that is actually in effect, it can be controlled by GATT service security. For level 4 in LE security mode 1, if the user only sets one level of Authenticated LE Secure Connections, it means that the current setting supports LE Secure Connections only.

3.4.2.4 Attribute PDU and GATT API

According to the Bluetooth Core Specification, the Telink BLE Multiple Connection SDK currently supports Attribute PDUs in the following categories:

- Requests: the data request sent by the client to the server.
- Responses: the data reply sent by the server after receiving the client's request.。
- Commands: The commands sent by the client to the server.
- Notifications: the data sent by the server to the client.
- Indications: the data sent by the server to the client.
- Confirmations: the confirmation of the server Indication data by the client.

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

34.24.1 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 Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.4.9 ATT_READ_BY_GROUP_TYPE_REQ/ 3.4.4.10 ATT_READ_BY_GROUP_TYPE_RSP.

The Master sends Read by Group Type Request, specify the initial and ending attHandle in this command, and specify attGroupType. After the Slave receives the Request, it iterates through the current Attribute table, finds the Attribute Group that conforms the attGroupType in the specified starting and ending attHandle, and replies to the Attribute Group information through Read by Group Type Response

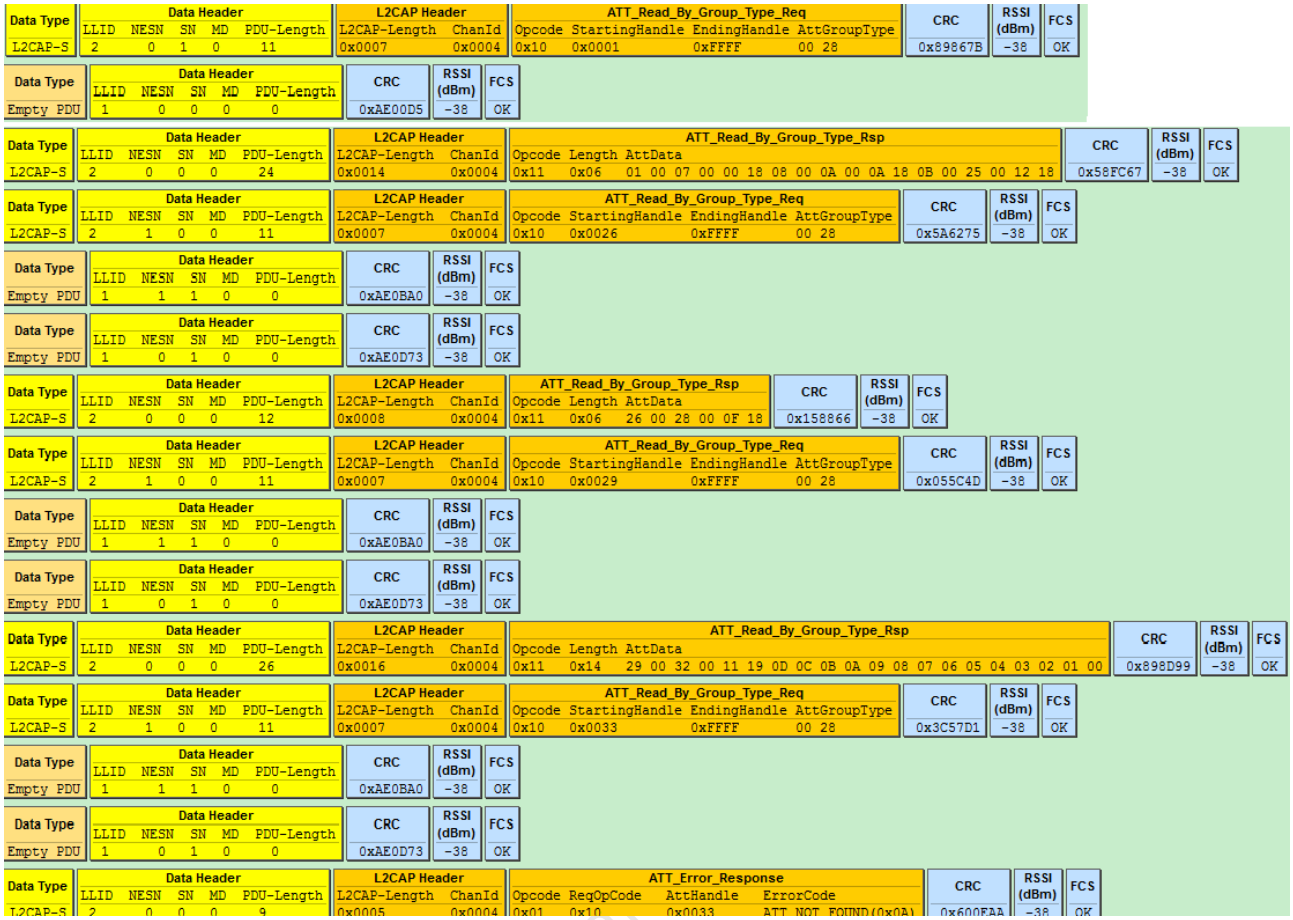


Figure 3.54: Read by group type request and read by group type response

As shown in the figure above, the Master queries the Slave for the Attribute Group information of the primaryServiceUUID whose UUID is 0X2800.

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

Referring to the current demo code, the Attribute table has the following groups that meet this requirement:

(1) attHandle is the Attribute Group from 0x0001 ~ 0x0007, and the Attribute Value is SERVICE_UUID_GENERIC _ACCESS(0x1800).

(2) attHandle is the Attribute Group from 0x0008 ~ 0x000B, and the Attribute Value is SERVICE_UUID_GENERIC _ATTRIBUTE(0x1801).

(3) attHandle is the Attribute Group from 0x000C ~ 0x000E, and the Attribute Value is SERVICE_UUID_DEVICE_ INFORMATION(0x180A).

(4) attHandle is the Attribute Group from 0x000F ~ 0x0029, and the Attribute Value is SERVICE_UUID_HUMAN_ INTERFACE_DEVICE(0x1812).

(5) attHandle is the Attribute Group from 0x002A ~ 0x002D, and the Attribute Value is SERVICE_UUID_BATTERY (0x180F).

(6) attHandle is the Attribute Group from 0x002E ~ 0x0035, and the Attribute Value is TELINK_SPP_UUID_SERVICE(0x10,0x19,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,0x01,0x00).

(7) attHandle is the Attribute Group from 0x0036 ~ 0x0039, and the 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 attHandle and attValue of the above 7 groups to the Master through Read by Group Type Response, the last ATT_Error_Response indicates that all Attribute Groups have been replied, the Response is over, and the Master will also stop sending Read by Group Type Request when it sees this packet.

Use the following API to implement Read by Group Request:

```
ble_sts_t  blc_gatt_pushReadByGroupTypeRequest(u16 connHandle, u16 start_attHandle, u16
↪  end_attHandle, u8 *uuid, int uuid_len);
```

The data of Read by Group Response can be read and processed in the app_gatt_data_handler function.

34.24.2 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 Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.3.3 ATT_FIND_BY_TYPE_VALUE_REQ/3.4.3.4 ATT_FIND_BY_TYPE_VALUE_RSP.

The Master sends Find by Type Value Request, specifying the starting and ending attHandle, AttributeType and Attribute Value in this command. After the Slave receives the Request, it iterates through the current Attribute table and finds the Attribute that matches the AttributeType and Attribute Value in the specified starting and ending attHandle, and reply Attribute with Find by Type Value Response.

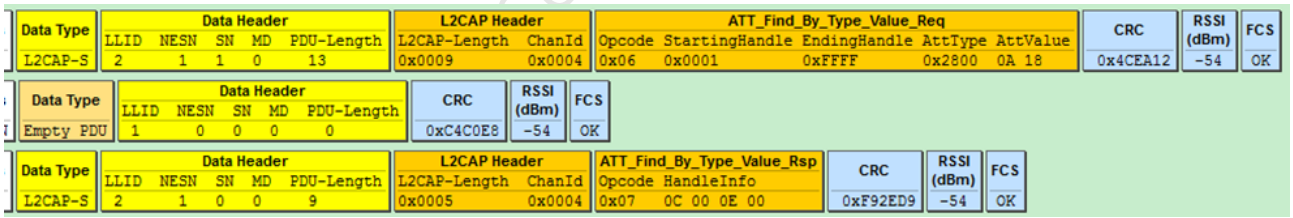


Figure 3.55: Find by type value request and find by type value response

Use the following API to implement Find by Type Value Request:

```
ble_sts_t   blc_gatt_pushFindByTypeValueRequest(u16 connHandle, u16 start_attHandle, u16
↪  end_attHandle, u16 uuid, u8 *attr_value, int len);
```

The data of Find by Type Value Response can be read and processed in the app_gatt_data_handler function.

34.24.3 Read by Type Request, Read by Type Response

For details of Read by Type Request and Read by Type Response, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.4.1 ATT_READ_BY_TYPE_REQ/3.4.4.2 ATT_READ_BY_TYPE_RSP.

The Master sends Read by Type Request, specifying the start and end attHandle in the command, and specifies AttributeType. After Slave receives the Request, it traverses the current Attribute table, finds the Attribute that matches AttributeType in the specified starting and ending attHandle, and replies to Attribute through Read by Type Response.

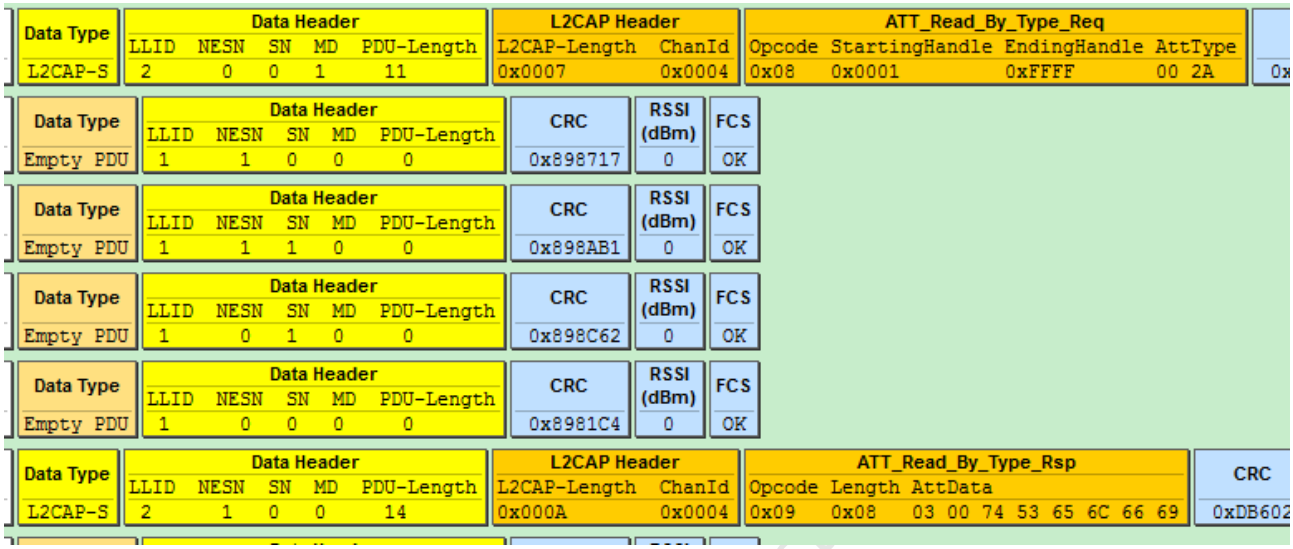


Figure 3.56: Read by type request and read by type response

As shown in the figure above, Master reads Attribute with attType 0x2A00, Attribute Handle in Slave is 0x0003 Attribute:

```
const u8 my_devName [] = {'t', 'S', 'e', 'l', 'f', 'i'};
   #define GATT_UUID_DEVICE_NAME            0x2a00
   const u16 my_devNameUUID = GATT_UUID_DEVICE_NAME;
   {0,1,2, sizeof (my_devName),(u8*)(&my_devNameUUID),              (u8*)(my_devName), 0},
```

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

Use the following API to implement Read by Type Request:

```
ble_sts_t   blc_gatt_pushReadByTypeRequest(u16 connHandle, u16 start_attHandle, u16
↪  end_attHandle, u8 *uuid, int uuid_len);
```

The data of Read by Type Response can be read and processed in the app_gatt_data_handler function.

34.244 Find information Request, Find information Response

For details of Find information request and Find information response, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.3.1 ATT_FIND_INFORMATION_REQ/3.4.3.2 ATT_FIND_INFORMATION_RSP.

The Master sends Find information request, specifying the starting and ending attHandle. After the Slave receives this command, it will reply to the Master with the UUID of the Attribute corresponding to all attHandle

at the beginning and ending via Find information response. As shown in the following figure, Master requires to obtain information of attHandle 0x0016 ~ 0x0018 three Attributes, and Slave responds to UUID of these three Attributes.

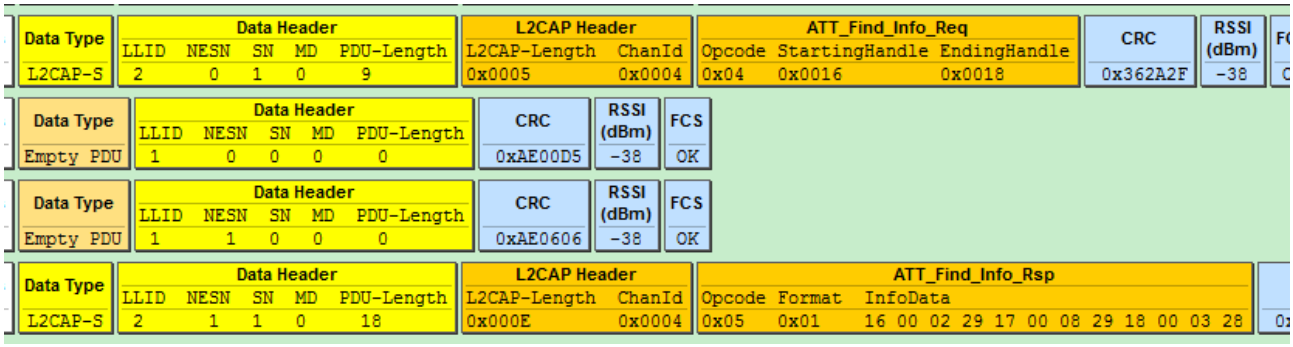


Figure 3.57: Find information request and find information response

34.24.5 Read Request, Read Response

For details of Read Request and Read Response, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.4.3 ATT_READ_REQ/3.4.4.4 ATT_READ_RSP.

The Master sends Read Request, specifying an attHandle as 0x0017, after receiving it, the Slave replies the Attribute Value of the specified Attribute through the Read Response (if the callback function r is set, execute this function), as shown in the figure below.

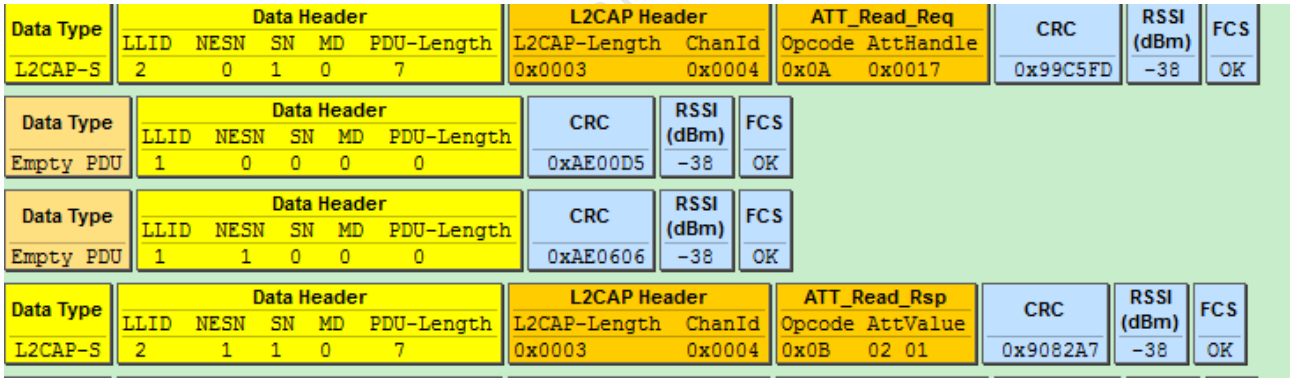


Figure 3.58: Read request and read response

Use the following API to implement Read Request:

```
ble_sts_t   blc_gatt_pushReadRequest(u16 connHandle, u16 attHandle);
```

The data of Read Response can be read and processed in the app_gatt_data_handler function.

34.24.6 Read Blob Request, Read Blob Response

For details of Read Blob Request and Read Blob Response, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.4.5 ATT_READ_BLOB_REQ/3.4.4.6 ATT_READ_BLOB_RSP.

When the length of the Attribute Value of an Attribute of the Slave exceeds MTU_SIZE (the default value is 23 in the current Telink BLE Multiple Connection SDK), the Master needs to enable Read Blob Request to read this Attribute Value, so that the Attribute Value can be sent by subpacketing. The Master specifies attHandle and ValueOffset in the Read Blob Request, after receiving the command, the Slave finds the corresponding Attribute, and replies the Attribute Value through the Read Blob Response according to the ValueOffset value (if the callback function r is set, execute it).

As shown in the figure below, when the Master reads the HID report map of the Slave (the report map is large, far exceeding 23), it first sends the Read Request, the Slave returns the Read response, and returns the previous part of the report map to the Master. After that, the Master uses the Read Blob Request, and the Slave returns data to the Master through Read Blob Response.

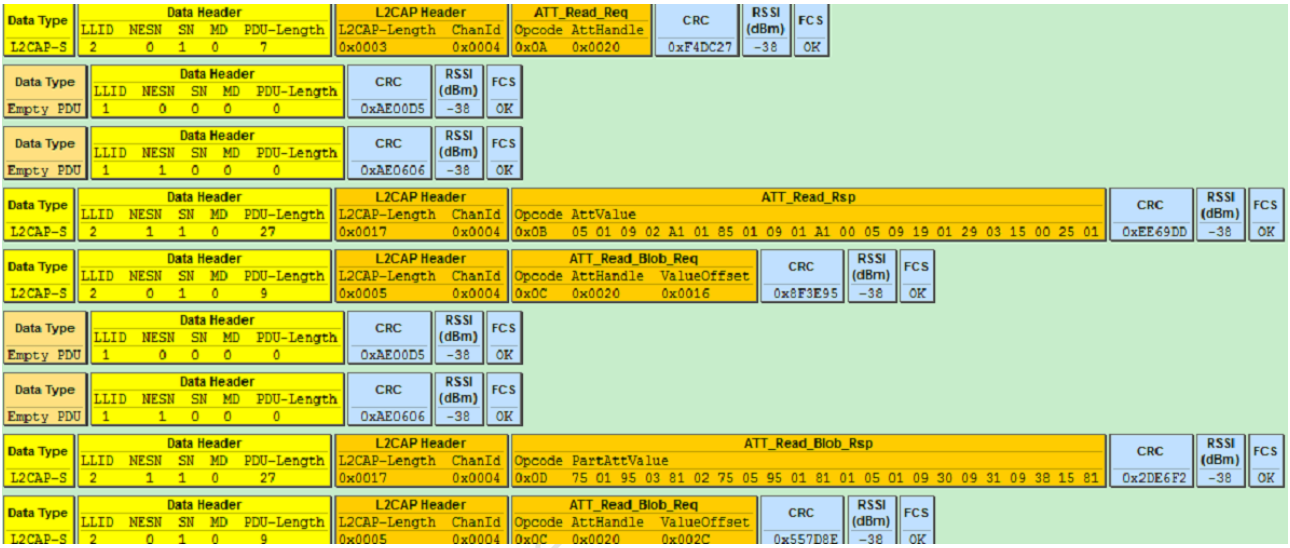


Figure 3.59: Read blob request and Read blob response

Use the following API to implement Read Blob Request:

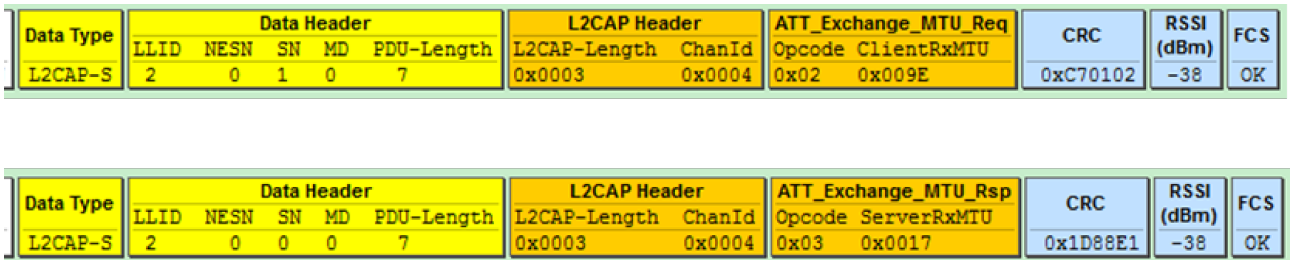
```
ble_sts_t   blc_gatt_pushReadBlobRequest(u16 connHandle, u16 attHandle, u16 offset);
```

The data of Read Blob Response can be read and processed in the app_gatt_data_handler function.

34.24.7 Exchange MTU Request, Exchange MTU Response

For details of Exchange MTU Request and Exchange MTU Response, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.2.1 ATT_EXCHANGE_MTU_REQ/3.4.2.2 ATT_EXCHANGE_MTU_RSP.

As shown below, Master and Slave learn each other's MTU size through Exchange MTU Request and Exchange MTU Response.



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

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

Figure 3.60: Exchange MTU request and exchange MTU response

When the data access process of the GATT layer appears more than one RF packet length data, involves the GATT layer subpacket and parcels, it is necessary to exchange the RX MTU size of both parties with the peer Master/Slave in advance, that is, the process of MTU size exchange. The purpose of the MTU size exchange is to enable the transceiver of long packet data at the GATT layer.

(1) The user can obtain EffectiveRxMTU by registering GAP event callback and turning on eventMask: Gap_evt_Mask_ATT_EXCHANGE_MTU, where:

```
EffectiveRxMTU=min(ClientRxMTU, ServerRxMTU)。
```

GAP event are described in detail in the "3.4.3.2 GAP event" of this document.

(2) The GATT layer receives long packet data for processing.

The default value of ServerRxMTU and ClientRxMTU is 23, and the maximum ServerRxMTU/ClientRxMTU can support the same as the theoretical value (only limited by RAM space). When the application needs to use subcontracting and reassembly, use the following API to modify the RX size on the master:

```
ble_sts_t blc_att_setMasterRxMTUSize(u16 master_mtu_size);
```

Use the following API to modify the RX size on the Slave:

```
ble_sts_t blc_att_setSlaveRxMTUSize(u16 slave_mtu_size);
```

Return value list:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
GATT_ERR_INVALID_PARAMETER	See the definition in SDK	Larger than the defined buffer size, i.e.: mtu_s_rx_fifo or mtu_m_rx_fifo

Note:

> The above two API settings are the MTU values when the ATT_Exchange_MTU_req/ATT_Exchange_MTU_rsp commands interact. The value cannot be greater than the actually defined buffer size, that is, the variable: mtu_m_rx_fifo[ ] and mtu_s_rx_fifo[ ], these two array variable are defined in the app_buffer.c.

As long as the MTU set using the above API is not the default value of 23, after the connection is established, the SDK will actively initiate the interaction process of MTU. By registering the Host event GAP_EVT_ATT_EXCHANGE_MTU, you can see the result of MTU interaction in the callback function.

34.24.8 Write Request, Write Response

For details of Write Request and Write Response, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.5.1 ATT_WRITE_REQ/3.4.5.2 ATT_WRITE_RSP.

The Master sends Write Request, specifies an attHandle, and comes with relevant data. After the Slave receives it, it finds the specified Attribute, and determines whether the data is processed by the callback function w or directly written to the corresponding Attribute Value according to whether the user has set the callback function w, and replies with Write Response.

As shown in the figure below, the Master writes the Attribute Value of 0x0001 to the Attribute whose attHandle is 0x0016, and the Slave executes the write operation after receiving it and replies with a Write Response.

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

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

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

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

Figure 3.61: Write request and write response

Use the following API to implement Write Request:

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

The data of Write Response can be read and processed in the app_gatt_data_handler function.

34.24.9 Write Command

For details of Write Command, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.5.3 ATT_WRITE_CMD.

The Master sends Write Command, specifies an attHandle, and comes with relevant data. After the Slave receives it, it finds the specified Attribute, and determines whether the data is processed by the callback

function w or directly written to the corresponding Attribute Value according to whether the user has set the callback function w, and does not reply any information.

Use the following API to implement Write Command:

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

34.24.10 Queued Writes

Queued Writes includes ATT protocols such as Prepare Write Request/Response and Execute Write Request/Response, for details, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.6 Queued writes.

Note:

> When using Queued Writes, the API blc_att_setPrepareWriteBuffer needs to be called at initialization time to allocate the storage buffer for the prepare write, which is not initially set by default in order to save ram.

Prepare Write Request and Execute Write Request can implement the following two functions:

a) Provides write functionality for long attribute value.

b) Allows multiple values to be written in a single atomic operation.

Prepare Write Request contains AttHandle, ValueOffset and PartAttValue, which is similar to the Read_Blob_Req/Rsp. This means that the Client can either prepare multiple attribute values in the queue, or prepare each part of a long attribute value. In this way, before actually executing the prepare queue, the client can be sure that all parts of an attribute can be written to the server.

Note:

> The current version of the Telink BLE Multiple Connection SDK only supports a) long attribute value write function, and the maximum length of long attribute value is less than or equal to 244 bytes.

As shown in the figure below, when Master writes long string to a characteristic of the Slave:"I am not sure what a new song" (The number of bytes is much more than 23, using the default MTU case), first send a Prepare Write Request with an offset of 0x0000, write the "I am not sure what" part of the data to the Slave, and the Slave returns a Prepare Write Response to the Master. After that, the Master sends a Prepare Write Request with an offset of 0x12, and writes the data of "a new song" to the Slave, and the Slave returns a Prepare Write Response to the Master. When the Master has finished writing all the long attribute values, it sends an Execute Write Request to the Slave, and Flags is 1: it means that the write takes effect immediately, the Slave replies with an Execute Write Response, and the entire Prepare write process ends.

Here we can know that Prepare Write Response also includes AttHandle, ValueOffset and PartAttValue in the request, and 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.

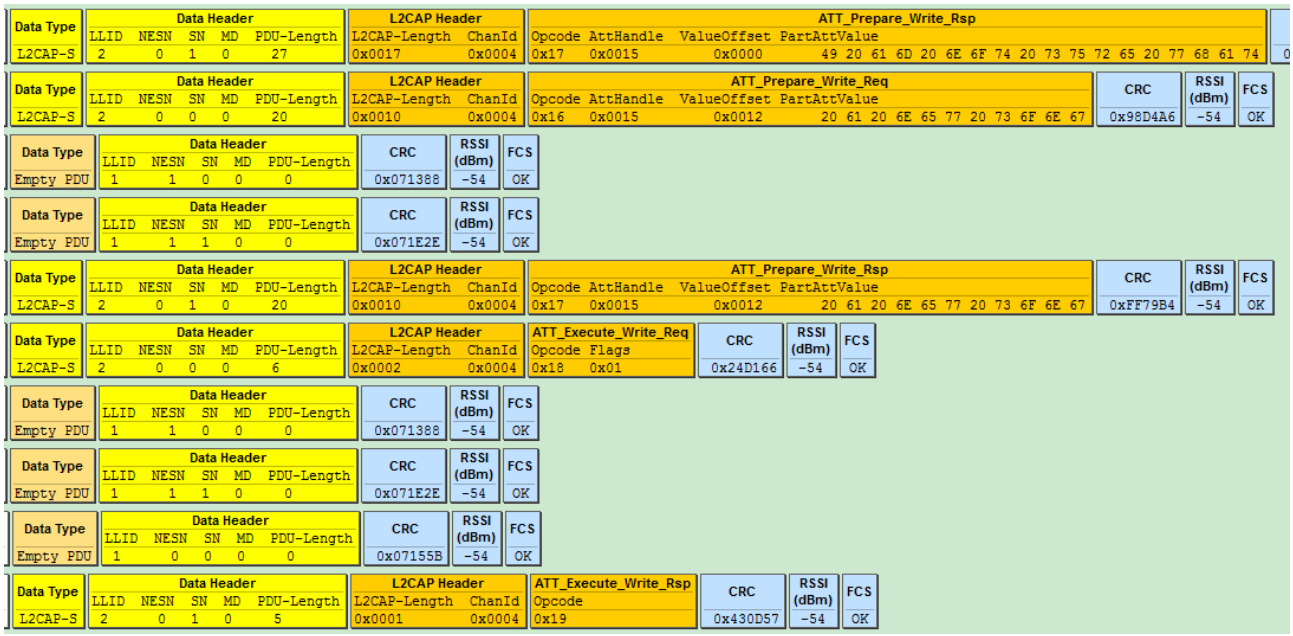


Figure 3.62: Example for write long characteristic values

34.24.11 Handle Value Notification

For details of Handle Value Notification, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.7.1 ATT_HANDLE_VALUE_NTF.

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

Figure 3.63: Handle value notification in Bluetooth Core Specification

The figure above shows the format of Handle Value Notification in Bluetooth Core Specification.

The Telink BLE Multiple Connection SDK provides an API for Handle Value Notification of an Attribute. The user calls this API to push the data that it needs to notify to the lower layer BLE software fifo, the protocol stack will push the data of the software fifo to the hardware fifo at the nearest transceiver packet interval, and finally send it out through RF.

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

The connHandle is the connHandle corresponding to the Connection state, attHandle is the attHandle 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 data to be sent. The API supports automatic unpacking function (Perform sub-packet handling according to EffectiveMaxTxOctets, that is, the smaller value of the maximum number of transceiver bytes in the link layer RF RX/TX, DLE may modify this value, and the default value is 27), which can split a long data into multiple BLE RF Packets and sent out, so len can support very large.

When Link Layer is in Conn state, generally calling this API directly can successfully push data to the lower layer software fifo, but there are some special cases that may cause the API call to fail, you can learn the corresponding error cause according to the return value ble_sts_t.

When calling this API, it is recommended that the user check whether the return value is BLE_SUCCESS, if it is not BLE_SUCCESS, need to wait for a period of time and push again.

The list of return values is as follows:

ble_sts_t	Value	ERR reason
BLE_SUCCESS	0	Success
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 ATT_MTU supported by ATT layer
LL_ERR_CONNECTION_NOT_ESTABLISH	0x80	Link Layer is in None Conn state
LL_ERR_ENCRYPTION_BUSY	0x82	In the encryption stage, and cannot send data
LL_ERR_TX_FIFO_NOT_ENOUGH	0x81	There are large data volume tasks running, and the software Tx fifo is not enough
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	0xB4	In the traversal service stage, and cannot send data

34.24.12 Handle Value Indication

For details of Handle Value Indication, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.7.2 ATT_HANDLE_VALUE_IND.

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

Figure 3.64: Handle value indication in Bluetooth Core Specification

The figure above shows the format of Handle Value Indication in Bluetooth Core Specification.

The Telink BLE Multiple Connection SDK provides an API for Handle Value Indication of an Attribute. The user calls this API to push the data that it needs to indicate to the lower layer BLE software fifo, the protocol stack will push the data of the software fifo to the hardware fifo at the nearest transceiver packet interval, and finally send it out through RF.

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

The connHandle is the connHandle corresponding to the Connection state, attHandle is the attHandle 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 data to be sent. The API supports automatic unpacking function (Perform sub-packet handling according to EffectiveMaxTxOctets, that is, the smaller value of the maximum number of transceiver bytes in the link layer RF RX/TX, DLE may modify this value, and the default value is 27, its replacement API will be introduced below, see remarks), which can split a long data into multiple BLE RF Packets and sent out, so len can support very large.

The Bluetooth Core Specification stipulates that each indicate data should wait until the client's confirmation to consider the indicate successful, if it is unsuccessful, the next indicate data cannot be sent.

When Link Layer is in Conn state, generally calling this API directly can successfully push data to the lower layer software fifo, but there are some special cases that may cause the API call to fail, you can learn the corresponding error cause according to the return value ble_sts_t.

When calling this API, it is recommended that the user check whether the return value is BLE_SUCCESS, if it is not BLE_SUCCESS, need to wait for a period of time and push again.

The list of return values is as follows:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Success
GAP_ERR_INVALID_PARAMETER	0xC0	Invalid parameters
SMP_ERR_PAIRING_BUSY	0xA1	In the pairing stage

ble_sts_t	Value	ERR Reason
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 ATT_MTU supported by ATT layer
LL_ERR_CONNECTION_NOT_ESTABLISH	0x80	Link Layer is in None Conn state
LL_ERR_ENCRYPTION_BUSY	0x82	In pairing or encryption stage, and cannot send data
LL_ERR_TX_FIFO_NOT_ENOUGH	0x81	There are large data volume tasks running, and the software Tx fifo is not enough
GATT_ERR_DATA_PENDING_DUE_TO_SERVICE_DISCOVERY_BUSY	0xB4	In the traversal service stage, and cannot send data
GATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	0xB1	The previous indicate data has not yet been confirmed by the master

34.24.13 Handle Value Confirmation

For details of Handle Value Confirmation, please refer to Bluetooth Core Specification V5.3, Vol 3, Part F, 3.4.7.3 ATT_HANDLE_VALUE_CFM.

Each time the application layer 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, and then the Slave can continue to send the next indicate data.

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

Table 3.36: Format of Handle Value Confirmation

Figure 3.65: Handle value confirmation in BLE

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

In order to let the application layer know whether the sent indicate data has been confirmed, the user can register the GAP event callback and enable the corresponding eventMask: GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM to obtain the confirm event, this document "3.4.3.2 GAP event" will introduce GAP event in detail.

34.24.14 blc_att_setServerDataPendingTime_upon_ClientCmd

The bottom layer of Telink BLE Multiple Connection SDK does not allow notify and indicate operations during the SDP process and the data pending time (default 300 ms) after the SDP. If the user needs to change the data pending time, this API can be used.

```
void   blc_att_setServerDataPendingTime_upon_ClientCmd(u8 num_10ms);
```

The parameter is in 10 ms units, e.g. if the parameter is substituted with 30, it means 30 * 10 ms, that is, 300 ms.

3.4.3 GAP

3.4.3.1 GAP Initialization

In the Telink BLE Multiple Connection SDK, because central and peripheral play at the same time in one device, the central and peripheral devices are not distinguished during initialization.

Initialization function:

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

From the foregoing we know that, the data interaction between the application layer and host is not controlled through GAP, the protocol stack provides relevant interfaces in ATT, SMP and L2CAP, and 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 registered with the host layer event processing function entry.

3.4.3.2 GAP Event

GAP event is an event generated during the interaction of ATT, GATT, SMP, GAP host protocol layers. From the previous article, we can know that the current SDK events are mainly divided into two categories: Controller event and GAP (host) event, in which controller event is divided into HCI event and LE HCI event.

Telink BLE SDK has added GAP event handling, which mainly makes the protocol stack event layering clearer and the protocol stack processing user layer interaction events more convenient, especially SMP related processing, such as Passkey input, pairing result notification to the user, etc.

If the user needs to receive the GAP event at the App layer, the callback function of the GAP event needs to be registered first, and then the mask of the corresponding event needs to be opened.

The callback function prototype and registration interface of GAP event 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 lower layer protocol stack.

The following lists several events that users may use:

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

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

```
blc_gap_registerHostEventHandler( app_host_event_callback );
```

GAP event need to open the mask through the following API.

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

The definition of eventMask also corresponds to some of those given above, other event masks can be found by the user in ble/gap/gap_event.h.

```
#define GAP_EVT_MASK_SMP_PAIRING_BEGIN          (1<<GAP_EVT_SMP_PAIRING_BEGIN)
#define GAP_EVT_MASK_SMP_PAIRING_SUCCESS        (1<<GAP_EVT_SMP_PAIRING_SUCCESS)
#define GAP_EVT_MASK_SMP_PAIRING_FAIL            (1<<GAP_EVT_SMP_PAIRING_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, then the application layer will not be notified when the GAP corresponding event is generated.

Note:

> When GAP event is discussed below, all are set to register the GAP event callback, and the corresponding eventMask is opened.

(1) GAP_EVT_SMP_PAIRING_BEGIN

Event trigger conditions: When the Slave and Master are just connected and enter the connection state, after the Slave sends the SM_Security_Req command, the Master sends the SM_Pairing_Req request to

start pairing, when the Slave receives the pairing request command, this event is triggered, indicating that pairing begins.

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

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

Figure 3.66: SMP_PAIRING_BEGIN event trigger conditions

Data length n: 4.

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

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

The connHandle indicates the current connection handle.

The secure_conn is 1 to use the secure encryption characteristic (LE Secure Connections), otherwise LE legacy pairing will be used.

The tk_method indicates what TK value method is used for the next pairing: such as JustWorks, PK_Init_Dsply_ Resp_Input, PK_Resp_Dsply_Init_Input, Numric_Comparison, etc.

(2) GAP_EVT_SMP_PAIRING_SUCCESS

Event trigger conditions: The event is generated when the whole pairing process is completed correctly, and this stage is the key distribution stage 3 (Key Distribution, Phase 3) of the LE pairing stage, if there is a key to be distributed, the pairing success event will be triggered after the key distribution of both parties is completed, otherwise, the pairing success event will be triggered directly.

Data length n: 4.

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

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

The connHandle indicates the current connection handle.

The bonding is 1 indicates the bonding function is enabled, otherwise it is not enabled.

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

(3) GAP_EVT_SMP_PAIRING_FAIL

Event trigger conditions: Due to the termination of the pairing process due to one of the Slave or Master not conforming to the standard pairing process, or the abnormal reasons such as reporting errors during communication.

Data length n: 2.

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

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

The connHandle indicates the current connection handle.

The reason indicates the reason for the pairing failure, here are several common pairing failure cause values, For other pairing failure cause values, we can refer to the "stack/ble/smp/smp_const.h" file in the SDK directory.

For the specific meaning of the pairing failure value, please refer to Bluetooth Core Specification V5.3, Vol 3, Part H, 3.5.5 Pairing Failed.

```
#define PAIRING_FAIL_REASON_CONFIRM_FAILED     0x04
#define PAIRING_FAIL_REASON_PAIRING_NOT_SUPPORTED  0x05
#define PAIRING_FAIL_REASON_DHKEY_CHECK_FAIL     0x0B
#define PAIRING_FAIL_REASON_NUMUERIC_FAILED     0x0C
#define PAIRING_FAIL_REASON_PAIRING_TIEMOUT     0x80
#define PAIRING_FAIL_REASON_CONN_DISCONNECT     0x81
```

(4) GAP_EVT_SMP_CONN_ENCRYPTION_DONE

Event trigger conditions: Triggered when Link Layer encryption is complete (Link Layer receives start encryption response from Master).

Data length n: 3.

Return pointer p: Points 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;
```

The connHandle indicates the current connection handle.

The re_connect is 1, it means fast reconnection (the previously distributed LTK encrypted link will be used), if this value is 0, it means that the current encryption is the first encryption.

(5) GAP_EVT_SMP_TK_DISPALY

Event trigger conditions: After Slave receives the Pairing_Req sent by Master, according to the pairing parameters of the peer device and the pairing parameter configuration of the local device, we can know what TK (pincode) value method is used for the next pairing. If the PK_Resp_Dsply_Init_Input (that is, The Slave displays the 6 bit pincode code, and the Master is responsible for inputting the 6 bit pincode code) mode is enabled, it will be triggered immediately.

Data length n: 4.

Return pointer p: Points to a u32 variable tk_set, the value 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, the reference code is as follows:

```
case GAP_EVT_SMP_TK_DISPALY:
{
 char pc[7];
 u32 pinCode = *(u32*)para;
 sprintf(pc, "%d", pinCode);
 printf("TK display:%s\n", pc);
}
break;
```

The pincode can be set during initialization through the following API, such as setting to 123456:

```
blc_smp_setDefaultPinCode(123456);
```

If the value is set to 0, or if the above API is not called to set it, the Pincode value is random.

Users input the 6 bit pincode code seen on Slave to Master device (such as mobile phone), complete TK input, and the pairing process can be continued. If the user input pincode error or click cancel, the pairing process fails.

(6) GAP_EVT_SMP_TK_REQUEST_PASSKEY

Event trigger conditions: When the Passkey Entry method is enabled on the Slave device and the PK_Init_Dsply_Resp_Input or PK_BOTH_INPUT pairing mode is used, this event will be triggered to notify user that TK value needs to be entered. After receiving the event, the user needs to input TK value through IO input ability (if the timeout is 30s, it has not been input, the pairing will fail), the API for inputting TK value: blc_smp_setTK_by_PasskeyEntry is described in the "SMP parameter configuration" section.

Data length n: 0.

Return pointer p: NULL.

(7) GAP_EVT_SMP_TK_REQUEST_OOB

Event trigger conditions: When the traditional pairing OOB method is enabled on the Slave device, this event will be triggered to notify the user that the 16 bit TK value needs to be input through the OOB method. After receiving the event, the user needs to input 16 bit TK value through IO (if the timeout is 30s, it has not been input, the pairing will fail), the API for inputting TK value: blc_smp_setTK_by_OOB is described in "3.4.4.2 SMP parameter configuration".

Data length n: 0.

Return pointer p: NULL.

(8) GAP_EVT_SMP_TK_NUMERIC_COMPARE

Event trigger conditions: After Slave receives the Pairing_Req sent by Master, according to the pairing parameters of the peer device and the pairing parameter configuration of the local device, we can know what TK (pincode) value method is used for the next pairing, if the Numeric_Comparison method is enabled, it will be triggered immediately. (Numeric_Comparison method, i.e. numeric comparison, belongs to smp4.2 secure encryption, both Master and Slave devices will pop up the 6 bit pincode code and the "YES" and "No" dialog box, users need to check whether the pincode displayed on both devices is the same, and needs to confirm whether to click "YES" on both devices to confirm whether the TK checksum is passed).

Data length n: 4.

Return pointer p: Points to a u32 variable pinCode, the value 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 and provide the confirmation mechanism of "YES" and "NO".

(9) GAP_EVT_ATT_EXCHANGE_MTU

Event trigger conditions: Whether the Master sending Exchange MTU Request, Slave reply Exchange MTU Response, or the Slave sending Exchange MTU Request, Master reply Exchange MTU Response, both cases will trigger.

Data length n: 6.

Return pointer p: Points 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;
```

The connHandle indicates the current connection handle.

The peer_MTU indicates the RX MTU value of the peer.

The effective_MTU = min(CleintRxMTU, ServerRxMTU), CleintRxMTU indicates the RX MTU size value for the client and ServerRxMTU indicates the RX MTU size value for the server. After the Master and Slave have interacted with each other's MTU size, take the minimum value of the two as the maximum MTU size value of each other's interaction.

(10) GAP_EVT_GATT_HANDLE_VLAUE_CONFIRM

Event trigger condition: Each 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 the data, which is triggered when the Slave receives the confirm.

Data length n: 0.

Return pointer p: NULL.

3.4.4 SMP

Security Manager (SM) provides LE devices with various keys required for encryption to ensure data confidentiality. The encrypted link can prevent third-party "attackers" from intercepting, deciphering or tampering with the original content of air data. For SMP details, please refer to Bluetooth Core Specification V5.3, Vol 3, Part H: Security Manager Specification.

3.4.4.1 SMP Security Level

Bluetooth Core Specification V4.2 adds a new pairing method called LE Secure Connections, which further enhances security. The previous pairing method is collectively referred to as LE Legacy Pairing.

Telink BLE Mulitple Connection SDK provides the following 4 security levels, refer to Bluetooth Core Specification V5.3, Vol 3, Part C, 10.2 LE security modes:

a) No authentication and no encryption (LE security Mode 1 Level 1)

b) Unauthenticated pairing with encryption (LE security Mode 1 Level 2)

c) Authenticated pairing with encryption (LE security Mode 1 Level 3)

d) Authenticated LE Secure Connections (LE security Mode 1 Level 4)

Note:

- All connections are supported to the highest security level, the master and slave can configure different security levels, and each connection can use different security levels to communicate;
- The security level set by the local device only indicates the highest security level that the local device may reach, and want to achieve the set security level is related to two factors:
  (a) peer device set the highest security level that can support >= local device set the highest security level that can support;
  (b) local device and peer device process the entire pairing process correctly according to the SMP parameters set for each (if a pairing exists).

For example, if the user sets the highest security level that the Slave can support is Mode 1 Level 3, but the Master connecting to the Slave is set to not support pairing encryption (only Mode 1 Level 1 is supported at the highest), then Slave and Master will not be paired after connection, and the actual security level used by the Slave is Mode 1 Level 1.

Use the following API to set the highest security level that the local device can support:

```
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);
```

Description:

In the Telink BLE Multiple Connection SDK, the API for configuring SMP-related parameters, unless otherwise specified, will have the following three configuration forms:

- API(...) for unified configuration of Master role and Slave role parameters;
- API_master(...) for configuring all Master role parameters individually;
- API_slave(...) for configuring all Slave role parameters individually.

3.4.4.2 SMP Parameter Configuration

When the initialization of GAP is called, the SMP is initialized, and the parameters of the SMP are initialized to default values:

- The highest security level supported by default: Unauthenticated_Paring_with_Encryption, which is Mode 1 Level 2;
- Default bonding mode: Bondable_Mode (refer to blc_smp_setBondingMode() API description);
- Default IO Capability: IO_CAPABILITY_NO_INPUT_NO_OUTPUT;
- Default pairing method: Legacy Pairing Just Works.

After the initialization is completed, first configure the SMP parameters through the API of SMP parameter configuration, and then use the following API to bring the parameters configured at the application layer into the bottom layer for initial configuration.

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

The following describes the related APIs for SMP parameter configuration.

```
void blc_smp_setPairingMethods(pairing_methods_t  method); //_slave()/_master()
```

This set of APIs is used to configure the SMP pairing method, Legacy or Secure Connections.

Note:

> The secure pairing method of Secure Connection requires MTU >= 65.

```
void blc_smp_setIoCapability(pairing_methods_t  method); //_slave()/_master()
```

This set of APIs is used to configure the SMP IO capability (see the figure below) and determine the key generation method. Refer to Bluetooth Core Specification V5.3, Vol 3, Part H, 2.3.5.1 Selecting Key Generation Method.

```
// 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 IOCap*/][5 /*Initiator IOCap*/] = {
    { 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 },
};

static const stk_generationMethod_t gen_method_sc[5 /*Responder IOCap*/][5 /*Initiator IOCap*/] = {
    { 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 },
};
```

Figure 3.67: Mapping KEY generation method according to different IO capabilities

```
void    blc_smp_enableAuthMITM(int MITM_en);  //_slave()/_master()
```

This set of APIs is used to configure the MITM (Man in the Middle) flag of SMP to provide Authentication. When the security level is Mode 1 Level 3 and above, this parameter is required to be 1. The value of parameter MITM_en is 0 corresponding to disabled; 1 corresponding to enable.

```
void    blc_smp_enableOobAuthentication(int OOB_en); //_slave()/_master()
```

This set of APIs is used to configure the OOB flag of SMP, which requires the security level to be Mode 1 Level 3 and above. The value of parameter OOB_en is 0 corresponding to disabled; 1 corresponding to enable.

The device will decide whether to use the OOB mode or the IO capability according to the OOB and MITM flag of the local device and the peer device, refer to Bluetooth Core Specification V5.3, Vol 3, Part H, 2.3.5.1 Selecting Key Generation Method.

```
void    blc_smp_setBondingMode(bonding_mode_t mode); //_slave()/_master()
```

This set of APIs is used to configure whether to store the Key generated by the SMP process in the Flash, If it is set to Bondable_Mode, the user can use SMP information for automatic reconnection, and will not re-pairing during reconnection; if it is set to Non_Bondable_Mode, the generated Key will not be stored in Flash, and will not be able to reconnection automatically after disconnection, and re-pairing is required.

```
void    blc_smp_enableKeypress(int keyPress_en); //_slave()/_master()
```

This set of APIs is used to configure whether to enable the Key Press function. The value of the parameter keyPress_en is 0 corresponding to disabled; 1 corresponding to enable.

```
void  blc_smp_setSecurityParameters(bonding_mode_t mode, int MITM_en, pairing_methods_t method,
↪  int OOB_en, int keyPress_en, io_capability_t ioCapablility); //_slave()/_master()
```

This set of APIs is used to configure the aforementioned SMP parameters as a whole. The corresponding relationship between each parameter and the above APIs is as follows:

parameter	API
mode	void blc_smp_setBondingMode(bonding_mode_t mode);
MITM_en	void blc_smp_enableAuthMITM(int MITM_en);
method	void blc_smp_setPairingMethods(pairing_methods_t method);
OOB_en	void blc_smp_enableOobAuthentication(int OOB_en);
keyPress_en	void blc_smp_enableKeypress(int keyPress_en);
ioCapablility	void blc_smp_setIoCapability(pairing_methods_t method);

```
void blc_smp_setEcdhDebugMode(ecdh_keys_mode_t mode); //_slave()/_master()
```

This set of APIs is used to configure whether Security Connections enables the Debug key pair for elliptical encryption keys. Using the elliptic encryption algorithm in the case of secure connection pairing can effectively avoid eavesdropping, but users cannot parse BLE air packets through the sniffer tool, so the Bluetooth Core Specification provides a set of elliptical encryption private/public key pairs for Debug, as long as this mode is opened, some BLE sniffer tools can use this known key to decrypt the link.

Note:

> Slave and Master only allow one party's key to be configured as a Debug key pair, otherwise the connection is not secure and the meaning of pairing is lost, and the protocol stipulates that it is illegal.

```
void blc_smp_setDefaultPinCode(u32 pinCodeInput); //_slave()/_master()
```

This set of APIs is used to configure the default Pincode displayed by the Display device in Passkey Entry or Numeric Comparison pairing mode. The parameter range is in [0,999999].

```
u8 blc_smp_setTK_by_PasskeyEntry (u16 connHandle, u32 pinCodeInput); //connHandle distinguishes
↪  connection links
```

This API is used to input the TK value of the Input device in Passkey Entry pairing mode. The return value 1 means the setting is successful, and 0 means that the Input device is not currently required to input the TK value.

Description:

Here is an explanation of the relationship between TK, Passkey, and Pincode. TK (Temporary Key), as the most basic original key in the SMP process, can be generated in various ways: e.g. Just Works generates TK = 0 by default; the Passkey Entry method enters the TK value, which is called Pincode in the application layer; the OOB method generates the TK through OOB data. It can be simply understood that Pincode generates Passkey, and Passkey generates TK, but this "generation" does not necessarily change its value.

```
u8 blc_smp_setTK_by_OOB (u16 connHandle, u8 *oobData); //connHandle distinguishes connection
↪  links
```

This API is used to set the OOB data of the device in OOB pairing mode. The parameter oobData indicates the head pointer of the 16 bit OOB data array to be set. The return value 1 means the setting is successful, and 0 means that the Input device is not currently required to input the TK value.

```
u8 blc_smp_isWaitingToSetTK(u16 connHandle); //connHandle distinguishes connection links
```

This API is used to obtain whether the Input device is waiting for TK input in Passkey Entry or OOB pairing mode. Returns 1 to indicate waiting for input.

```
void    blc_smp_setNumericComparisonResult(u16 connHandle, bool YES_or_NO); //connHandle
↪  distinguishes connection links
```

This API is used to set YES or NO for device input in Numeric Comparison pairing mode under Security Connections. When the user confirms that the displayed 6 bit value is consistent with the peer device, he can enter 1 ("YES"), and if it is inconsistent, enter 0 ("NO").

```
u8   blc_smp_isWaitingToCfmNumericComparison(u16 connHandle); //connHandle distinguishes
↪  connection links
```

This API is used to get whether the device is waiting for Yes or No input in Numeric Comparison pairing mode under Security Connections. Returns 1 to indicate waiting for input.

```
int  blc_smp_isPairingBusy(u16 connHandle); //connHandle distinguishes connection links
```

This API is used to query whether a connection is being paired. Return value 0 indicates not being paired, and 1 indicates being paired.

3.4.4.3 SMP Process Configuration

- The SMP Security Request can only be sent by the Slave, and is used to actively request the peer Master to perform the pairing process, which is an optional process of SMP.
- The SMP Pairing Request can only be sent by the Master to notify the Slave to start the pairing process.

344.3.1 SMP Security Request

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

This API is used to flexibly configure the timing of Slave sending Security Request.

Note:

> The call is effective only before the connection is established, and it is recommended to configure it during initialization.

The enumeration type secReq_cfg is defined as follows:

```
typedef enum {
 SecReq_NOT_SEND = 0,   //After the connection is established, Slave will not actively send
↪  Security Request
 SecReq_IMM_SEND = BIT(0),  //After the connection is established, the Slave will immediately
↪  send a Security Request
```

```
 SecReq_PEND_SEND = BIT(1),  //After the connection is established, the Slave waits pending_ms
 ↪  (in milliseconds) before deciding whether to send a Security Request
}secReq_cfg;
```

newConn_cfg: Used to configure new connections. If the Slave is configured as SecReq_PEND_SEND and receives the Pairing Request packet from the Master before pending_ms, it will not send the Security Request.

reConn_cfg: Used to configure a reconnection. Pairing the bound device, the next time it connects (ie, reconnection), Master may not necessarily initiate LL_ENC_REQ to encrypt the link, at this time, if the Slave sends a Security Request, it can trigger the Master to encrypt the link. If the Slave is configured as SecReq_PEND_SEND and has received the LL_ENC_REQ packet from the Master before pending_ms, the Security Request will not be sent again.

pending_ms: This parameter only works when either newConn_cfg or reConn_cfg is configured as SecReq_PEND_SEND.

3.4.4.3.2 SMP Pairing Request

```
void blc_smp_configPairingRequestSending( PairReq_cfg newConn_cfg,  PairReq_cfg reConn_cfg);
```

This API is used to flexibly configure the timing of Pairing Requests sent by the Master.

Note:

> It can only be called before connection and is recommended to be configured during initialisation.

The enumeration type PairReq_cfg is defined as follows:

```
typedef enum {
 PairReq_SEND_upon_SecReq = 0,  // Master sending Pairing Request is dependent on receiving
 ↪  Security Request from Slave
 PairReq_AUTO_SEND = 1,    // The Master will automatically send a Pairing Request as soon as it
 ↪  is connected.
}PairReq_cfg;
```

3.4.4.4 SMP Pairing Method

SMP pairing method mainly focuses on the configuration of four security levels of SMP.

3.4.4.4.1 Mode 1 Level 1

The device does not support encrypted pairing, that is, disable the SMP function, and initialize the configuration:

```
blc_smp_setSecurityLevel(No_Security);
```

3444.2 Mode 1 Level 2

The device supports up to Unauthenticated_Paring_with_Encryption, such as Just Works pairing mode in Legacy Pairing and Secure Connections pairing modes.

- Initial configuration for LE Legacy Just works:

```
//blc_smp_setPairingMethods(LE_Legacy_Pairing);  //Default
//blc_smp_setSecurityLevel_master(Unauthenticated_Pairing_with_Encryption); //Default
blc_smp_smpParamInit();
```

- Initial configuration for LE Security Connections Just works:

```
blc_smp_setPairingMethods(LE_Secure_Connection);
blc_smp_smpParamInit();
```

3444.3 Mode 1 Level 3

The device supports up to Authenticated pairing with encryption, such as Passkey Entry and Out of Band of Legacy Pairing.

This level requires the device to support Authentication, which can ensure the legitimacy of the identity of both parties.

- Initial configuration of LE Legacy Passkey Entry mode Display device:

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABILITY_DISPLAY_ONLY);
//blc_smp_setDefaultPinCode(123456);
blc_smp_smpParamInit();
```

or

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_setSecurityParameters(Bondable_Mode, 1, LE_Legacy_Pairing, 0, 0,
 ↪  IO_CAPABILITY_DISPLAY_ONLY);
blc_smp_smpParamInit();
```

Here it concerns the display of TK's GAP event: GAP_EVT_SMP_TK_DISPALY, please refer to "3.4.3.2 GAP event".

- Initial configuration of LE Legacy Passkey Entry mode Input device:

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABLITY_KEYBOARD_ONLY);
blc_smp_smpParamInit();
```

or

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_setSecurityParameters(Bondable_Mode, 1, LE_Legacy_Pairing, 0, 0,
 ↪  IO_CAPABLITY_KEYBOARD_ONLY);
blc_smp_smpParamInit();
```

This concerns the GAP event for requesting TK: GAP_EVT_SMP_TK_REQUEST_PASSKEY, please refer to "3.4.3.2 GAP event".

The user calls the following API to set up TK:

```
void blc_smp_setTK_by_PasskeyEntry (u16 connHandle, u32 pinCodeInput);
```

- Initial configuration of LE Legacy OOB:

```
blc_smp_setSecurityLevel(Authenticated_Pairing_with_Encryption);
blc_smp_enableOobAuthentication(1);
blc_smp_smpParamInit();
```

or

```
blc_smp_setSecurityLevel_slave(Authenticated_Pairing_with_Encryption);
blc_smp_setSecurityParameters_slave(Bondable_Mode, 1, LE_Legacy_Pairing, 1, 0,
 ↪  IO_CAPABILITY_KEYBOARD_DISPLAY);
blc_smp_smpParamInit();
```

Here the GAP event for requesting OOB data is involved: GAP_EVT_SMP_TK_REQUEST_OOB, please refer to "3.4.3.2 GAP event".

34444 Mode 1 Level 4

The device supports up to Authenticated LE Secure Connections, such as Numeric Comparison, Passkey Entry, Out of Band of Secure Connections.

- Initial configuration of Secure Connections Passkey Entry mode:

It is basically the same as Legacy Pairing Passkey Entry, the only difference is that the pairing method needs to be set to "secure connection pairing" at the very beginning of initialization:

```
blc_smp_setSecurityLevel(Authenticated_LE_Secure_Connection_Pairing_with_Encryption);
blc_smp_setParingMethods(LE_Secure_Connection);
...//Refer to Mode 1 Level 3 configuration method
```

- Initial configuration of Secure Connections Numeric Comparison:

```
blc_smp_setSecurityLevel(Authenticated_LE_Secure_Connection_Pairing_with_Encryption);
blc_smp_setParingMethods(LE_Secure_Connection);
blc_smp_enableAuthMITM(1);
blc_smp_setIoCapability(IO_CAPABLITY_DISPLAY_YESNO);
blc_smp_smpParamInit();
```

or

```
blc_smp_setSecurityLevel_master(Authenticated_LE_Secure_Connection_Pairing_with_Encryption);
blc_smp_setSecurityParameters_master(Bondable_Mode, 1, LE_Secure_Connection, 0, 0,
 ↪  IO_CAPABLITY_DISPLAY_YESNO);
blc_smp_smpParamInit();
```

This concerns the GAP event for requesting Yes/No: GAP_EVT_SMP_TK_NUMERIC_COMPARE, please refer to "3.4.3.2 GAP event".

- Secure Connections OOB method, not supported by SDK at this time.

3.4.4.5 SMP Storage

Whether the device is a Master or a Slave, after SMP bonding with another device, some SMP-related information needs to be stored in Flash, so that it can be automatically reconnected after the device is powered on again. This process is called SMP Storage.

344.5.1 SMP Storage Area

The area in Flash for storing SMP bonding information is called the SMP Storage area.

For the Telink BLE Multiple Connection SDK, the starting position of SMP Storage area is specified by macro FLASH_ADR_SMP_PAIRING (default 0xFA000 for 1M Flash, 0x78000 for 512K Flash). The SMP Storage area is divided into 2 areas, called area A and area B, which occupy the same space and are specified by the macro FLASH_SMP_PAIRING_MAX_SIZE (the default is 0x2000, which is 8K, so the total SMP Storage area size is 16K). The (FLASH_SMP_PAIRING_MAX_SIZE-0x10) offset (default is 0x1FF0) position of each zone is the "zone valid Flag", 0x3C means valid, 0xFF means not valid. Users can reconfigure the SMP Storage area using the following API:

```
void blc_smp_configPairingSecurityInfoStorageAddressAndSize (int address, int size_byte);  //
 ↪  address and size must be 4K aligned
```

- address: The starting address of the SMP Storage area (also the starting address of area A);

- size_byte: The size of each SMP area, area A and area B are equal in size.

The following API is used to get the starting address of the current SMP Storage valid area:

```
u32 blc_smp_getBondingInfoCurStartAddr(void);
```

- Return value: The starting address of the current valid area in SMP Storage, such as 0xFC000.

After pairing, the SMP bonding information is stored in the SMP Storage area A by default. When the amount of bonding information in the area A reaches the warning line (8KB 3/4 = 96 Bytes 64, that is, a maximum of 64 bonding information can be stored) , will migrate the valid binding information to area B, set "area valid Flag" to 0x3C, and clear area A. Similarly, when the bonding information in area B reaches the warning line, switch to area A and clear area B. The following APIs can be used to confirm whether the information volume of the current SMP Storage effective area has reached the warning line:

```
bool blc_smp_isBondingInfoStorageLowAlarmed(void);
```

- Return value: 0 means not to the warning line, 1 means has reached the warning line.

If you need to clear the information in the SMP Storage and reset the SMP bonding information, it is recommended that you call the following API in a non-connected state:

```
void blc_smp_eraseAllBondingInfo(void);
```

3.4.4.5.2 Bonding Info

Each set of SMP bonding information stored in SMP Storage is called a Bonding Info block. By default, SMP Storage fills in Bonding Info into the SMP Storage area according to the pairing sequence. Refer to its structure smp_param_save_t to get:

- A Bonding Info block of size 96 bytes (0x60);
- The first Byte of the Bonding Info block, i.e. the flag member, represents the status of the Bonding Info block, and if flag & 0x0F == 0x0A, it means that the SMP bonding information is valid; if the flag member value of Master's Bonding Info block is 0x00, it means that the device has been unbound; if the bit7 of the flag is 0, it means that RPA is supported. For details, please refer to the RPA function section (this function is not yet fully released in the SDK);
- The second Byte of the Bonding Info block, role_dev_idx, represents the role played by itself, if bit7 is 1, it represents itself as the Master, if bit7 is 0, it represents the role of Slave in the connection;
- The peer Id Address and local/peer IRK obtained by SMP are stored in the Bonding Info block.

The following figure is a reference to the content of SMP Storage, which indicates that the Bonding Info block is valid and the device is the Master:

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000FA430:	65	98	B3	EE	D3	BD	CB	A4	DE	AA	F3	82	4B	B4	0A	BF
000FA440:	10	00	11	11	22	77	66	55	4C	11	2C	42	34	5B	21	55
000FA450:	BF	0A	B4	53	2A	5F	05	32	82	67	79	28	79	2D	B1	82
000FA460:	D2	10	CA	86	B4	FE	9F	98	29	A5	9D	97	DE	70	62	AA
000FA470:	9A	67	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
000FA480:	9A	80	00	04	00	00	77	66	55	00	04	00	00	77	66	55
000FA490:	EE	E0	65	1D	C2	2C	A6	45	71	28	6A	5F	30	AE	73	89
000FA4A0:	10	00	11	11	22	77	66	55	9A	E2	3D	D3	59	BA	8E	D7
000FA4B0:	89	73	AE	EC	6F	4C	B9	FC	D6	19	1B	5C	34	DC	1D	03
000FA4C0:	D2	10	CA	86	B4	FE	9F	98	29	A5	9D	97	DE	70	62	AA
000FA4D0:	11	1F	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
000FA4E0:	9A	80	00	05	00	00	77	66	55	00	05	00	00	77	66	55
000FA4F0:	DE	E1	8D	4D	DC	07	25	7F	11	C9	0E	4D	63	AF	CD	5F
000FA500:	10	00	11	11	22	77	66	55	72	B2	23	F8	DA	80	EE	36
000FA510:	5F	CD	AF	B9	DA	A6	E5	EB	B8	A5	00	FC	F8	AD	ED	3D
000FA520:	D2	10	CA	86	B4	FE	9F	98	29	A5	9D	97	DE	70	62	AA

Figure 3.68: SMP Storage Bonding Info Master

The following API can be used to obtain its Bonding Info through the MAC address of peer device :

```
u32 blc_smp_loadBondingInfoByAddr(u8 isMaster, u8 slaveDevIdx, u8 addr_type, u8* addr,
↪  smp_param_save_t* smp_param_load);
```

- isMaster: Its own role, 0 means Slave, non-0 means Master;
- slaveDevIdx: When the multi-address function is not involved, this parameter is 0;
- addr_type: The address type of the peer device, refer to BLE_ADDR_PUBLIC and BLE_ADDR_RANDOM;
- addr: MAC address of peer device;
- smp_param_load: The output parameter, points to the Bonding Info block corresponding to the peer device.
- return value: the first address in Flash of the Bonding Info block corresponding to the peer device.

For the convenience of the application layer, an API is provided for Master role to obtain its pairing state according to peer Slave's MAC :

```
u32 blc_smp_searchBondingSlaveDevice_by_PeerMacAddress( u8 peer_addr_type, u8* peer_addr);
```

- peer_addr_type: The address type of the peer Slave, refer to BLE_ADDR_PUBLIC and BLE_ADDR_RANDOM;
- peer_addr: The MAC address of the peer slave;
- return value: The first address in Flash of the Bonding Info block of the Bonding Device found; 0 means that no valid bonding information was found.

Use the following API to remove the corresponding Bonding Info through the MAC address of the peer device (actually, it is not removed, but it is invalidated by setting the flag):

```
int blc_smp_deleteBondingSlaveInfo_by_PeerMacAddress(u8 peer_addr_type, u8* peer_addr);
```

- peer_addr_type: The address type of the peer Slave, refer to BLE_ADDR_PUBLIC and BLE_ADDR_RANDOM;
- peer_addr: The MAC address of the peer slave;
- return value: Find and delete the first address of the Bonding Info block of Bonding Device in Flash; 0 means no valid bonding information is found.

344.5.3 Max Bonding Quantity

For Telink BLE Multiple Connection SDK, the SMP information of up to 8 valid peer Slave and the SMP information of 4 valid peer Master can be saved by default ("Valid" means that the device can be reconnected successfully, that is, the flag member of the Bonding Info block indicates that the current state is valid), which are called the maximum bonding number of Master and Slave in SDK (Bonding Device Max Number). Users can also reconfigure the maximum bonding number of SMP Storage through the following API :

```
void blc_smp_setBondingDeviceMaxNumber ( int peer_slave_max,
int peer_master_max);
```

- peer_slave_max: The maximum number of peer Slave bonding for itself as the Master;
- peer_master_max: The maximum number of peer Master bonding for itself as the Slave.

When the maximum number of bonding is reached, the next bound device will replace the earliest bound device of the currently valid devices in the same role. Specifically, the Bonding Info of the new device will continue to be written into the Flash, the flag will be set as valid, and the flag of the first device in the valid Bonding Info of the same role will be set as invalid.

For example, if the BLC_SMP_SETBONDICEVICEMAXNUMBER (8, 4) is set, when 8 peer Slaves are bound, once the 9th peer Slave is bound, the Bonding Info of the oldest (first) peer Slave will fail, and the Bonding Info of the 9th peer Slave device continues to be stored in Flash.

The user can get the current Slave or Master binding quantity through the following API:

```
u8 blc_smp_param_getCurrentBondingDeviceNumber(u8 isMasterRole, u8 slaveDevIdx);
```

- isMasterRole: Its own role, 0 means Slave, non-0 means Master;
- slaveDevIdx: When the multi-address function is not involved, this parameter is 0;
- return value: The number of valid bound devices. When isMasterRole is 0, it indicates the number of valid peer Master bound; when isMasterRole is not 0, it indicates the number of valid peer Slave bound.

344.54 SMP Bonding Info Index

The bonding information of each Bonding Device is assigned a serial number in SMP, which is called Bonding Info Index, the value of Bonding Info Index is assigned in Bonding Device Max Number by default according to the order of bonding. For example, if the Master's Bonding Device Max Number is 2, the Bonding Info Index of the two peer Slave pairs will be 0 and 1 respectively.

In this way, in addition to obtaining Bonding Info by peer device MAC address as described above, you can also obtain the Bonding Info through the Bonding Info Index when the device Bonding Info Index is known:

```
u32 blc_smp_loadBondingInfoFromFlashByIndex(u8 isMaster, u8 slaveDevIdx, u8 index,
↪  smp_param_save_t* smp_param_load);
```

- isMaster: Its own role, 0 means Slave, non-0 means Master;
- slaveDevIdx: When the multi-address function is not involved, this parameter is 0;
- index: Bonding Info Index representing the Master or Slave information to read.
- smp_param_load: The output parameter, points to the Bonding Info block corresponding to the peer device.
- return value: the first address in Flash of the Bonding Info block corresponding to the peer device.

The following API is used to set the assignment principle of the Bonding Info Index, which is not yet released in the SDK and is only used for the specific needs of some users:

```
void blc_smp_setBondingInfoIndexUpdateMethod(index_updateMethod_t method);
```

- method: Referring to index_updateMethod_t, it can be set to assign Bonding Info Index values according to the order in which connections are established or in the order of device bonding.

3.4.5 Custom Pair

In multiple Master and multiple Slave devices, if the Master device disables SMP, the SDK cannot automatically complete the pairing and unpairing operation and needs to add pairing management in the application layer. Based on this, Telink customizes a set of pairing and unpairing schemes.

If the user needs to use a custom pairing management, the function needs to be initialized first, call the following API :

```
blc_smp_setSecurityLevel_master(No_Security);//disable SMP function
user_master_host_pairing_management_init();//self-defining mode
```

(1) Flash storage method design

The flash data area sector used by default 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
```

Starting from flash 0x7C000, every 8 bytes 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 last 6 are 6 bytes mac addresses.

```
typedef struct {
  u8 bond_mark;
  u8 adr_type;
  u8 address[6];
} macAddr_t;
```

Flash stored procedures use a method that pushes back 8 bytes area in sequence, The first valid Slave mac is stored in 0x7C000-0x7C007, and write the first byte flag of 0x7C000 as 0x5A to indicate that the current address is valid; when the second valid mac address is stored in 0x7C008-0x7C00f, 0x7C008 is marked with 0x5A; when the third valid mac address is stored in 0x7C010-0x7C017, 0x7C010 is marked with 0x5A.

If you want a Slave device to unpairing, multiple Master and multiple Slave devices need to erase the MAC address of that device, Simply write the flag bit of the 8 bytes area where the MAC address was previously stored to 0x00; for example, to erase the first device of the three devices above, write 0x7C000 to 0x00.

The reason for using the above 8 bytes extension method is that the program cannot call the flash_erase_sector function to erase the flash memory while it is running, because the operation takes between 20-200ms to erase a sector 4K flash, this time can cause BLE timing errors.

Use the 0x5A and 0x00 flags to indicate paired storage and unpaired erasure of all Slave MACs, when the 8 bytes area becomes more and more, it may fill up the whole sector 4K flash and cause an error, special processing is added during initialization: reading the 8 bytes area information from 0x7C000 and read all valid MAC addresses to the Slave MAC table in RAM. In this process, check if the 8 bytes area is too much, if it is too much, erase the whole sector, and then rewrite the Slave MAC table maintained in RAM back to the 8 bytes area starting at 0x7C000.

(2) 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
```

Figure 3.69: Slave MAC table

```
user_salveMac_t user_tbl_slaveMac;
```

Maintain all matching devices in RAM using Slave MAC table with the above structure, Change macros USER_PAIR_SLAVE_MAX_NUM can define the maximum number of pairs you want to allow, The default of 4 in the Telink BLE Multiple Connection SDK refers to maintaining the pairing of 4 devices, user can modify this value.

The curNum in user_tbl_slaveMac indicates how many valid Slave devices are recorded on the current flash, the bond_flash_idx array records the offset of the starting address of the 8 bytes area of the effective address on the flash relative to 0x7C000 (when unpairing this device, the flag bit of 8 bytes area can be found by this offset and writing it as 0x00), the bond_device array records the MAC address.

(3) Related API description

Based on the above FLASH storage design and the design of the Slave MAC table in RAM, the following APIs can be called respectively.

a. 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 self-defined method is enabled.

b. user_tbl_slave_mac_add

```
int user_tbl_slave_mac_add(u8 adr_type, u8 *adr);
```

Add a Slave mac, return 1 for success, 0 for 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 it does not reach the maximum value, it will be added to the Slave MAC table unconditionally and stored in an 8 bytes area of FLASH.

If it has reached the maximum value. It involves the strategy of processing: whether do not allow pairing or directly overwrite the oldest one, the Telink demo method is to directly overwrite the oldest one. 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.

c. 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, to determine whether the device currently sending the advertising packet has been paired with the Master before, and if the device has been paired, it can be directly connected.

d. 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.

e. user_tbl_slave_mac_delete_by_index

```
void user_tbl_slave_mac_delete_by_index(int index)
```

Delete paired devices by specifying index. The Index value reflects the order in which the devices were paired. If the maximum number of pairings is 1, the index of the paired device is always 0; if the maximum number of pairings is 2, the index of the first paired device is 0, and the index of the second paired device is 1; and so on.

f. user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

Delete all paired devices.

(4) Connection and Pairing

When the Master receives the advertising packet reported by the Controller, it will connect with the Slave in the following two cases:

a. Call the function user_tbl_slave_mac_search to check whether the current device has been paired with Master and not unpairing, if it has been paired, it can be connected automatically.

```
master_auto_connect = user_tbl_slave_mac_search(pa->adr_type, pa->mac);
if(master_auto_connect) { create connection }
```

b. If the current advertising device is not in the Slave MAC table and does not meet the automatic connection, 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 matching key on the multiple master and multiple slave device is pressed; the second is that the current advertising data is a pairing 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);
}
```

Figure 3.70: Connecting pairing code 1

```
if(user_manual_pairing) { create connection }
```

If the connection is established by manual pairing, after the connection is successfully established, that is, when the HCI LE CONECTION ESTABLISHED EVENT is reported, the current device is added 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);
}
```

Figure 3.71: Connecting pairing code 2

c. unpairing

When the unpairing condition takes effect, the multiple master and multiple slave device first calls blc_llms_disconnect to disconnect, and then calls the user_tbl_salve_mac_delete_by_adr function to delete this device.

3.4.6 Device Manage & Simple SDP

As described above for GATT, in BLE, the Slave, in its role as GATT Server, maintains a table of GATT Services, with each Attribute in the table corresponding to an Attribute handle value. For the Master, to obtain this information for the Slave, it is necessary to obtain it through the SDP process and maintain it for use when needed.

For ease of use, the Telink BLE Multiple Connection SDK provides the user with an implementation of Device Manage as a connected device management solution and a simple implementation for the Master to do SDP to get the GATT Service table of the peer Slave. Not only is it possible to manage the GATT Service table of a peer Slave for the Master, but it can also be used to retrieve other information about the peer device at any time by using some of the information from that device. The solution is provided in source code form and users can refer to the vendor/common/device_manage.\and vendor/common/simple_sdp.\ files in the SDK.

The Telink BLE Multiple Connection SDK uses the following data structure to manage "Attribute handle" and "Connection handle".

```
typedef struct
{
 u16  conn_handle;
 u8  conn_role;    // 0: master; 1: slave
 u8   conn_state;    // 1: connect;  0: disconnect

 u8  char_handle_valid;      // 1: peer device's attHandle is available;   0: peer device's
 ↪  attHandle not available
 u8  rsvd[3];      // for 4 Byte align

 u8  peer_adrType;
 u8  peer_addr[6];
 u8  peer_RPA;         //RPA: resolvable private address

 u16  char_handle[CHAR_HANDLE_MAX];
}dev_char_info_t;
```

In the SDK, the array "conn_dev_list[]" is used to record and maintain the "attribute handle" of the peer device, as shown in the following figure.

vice_manage.h | device_manage.c

```
/*
 * 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
 *                0               1                none              conn_dev_list[0]
 *                0               2                none              conn_dev_list[0..1]
 *                0               3                none              conn_dev_list[0..2]
 *                0               4                none              conn_dev_list[0..3]
 *
 *                1               0           conn_dev_list[0]              none
 *                1               1           conn_dev_list[0]        conn_dev_list[1]
 *                1               2           conn_dev_list[0]        conn_dev_list[1..2]
 *                1               3           conn_dev_list[0]        conn_dev_list[1..3]
 *                1               4           conn_dev_list[0]        conn_dev_list[1..4]
 *
 *                2               0           conn_dev_list[0..1]           none
 *                2               1           conn_dev_list[0..1]     conn_dev_list[2]
 *                2               2           conn_dev_list[0..1]     conn_dev_list[2..3]
 *                2               3           conn_dev_list[0..1]     conn_dev_list[2..4]
 *                2               4           conn_dev_list[0..1]     conn_dev_list[2..5]
 *
 *                3               0           conn_dev_list[0..2]           none
 *                3               1           conn_dev_list[0..2]     conn_dev_list[3]
 *                3               2           conn_dev_list[0..2]     conn_dev_list[3..4]
 *                3               3           conn_dev_list[0..2]     conn_dev_list[3..5]
 *                3               4           conn_dev_list[0..2]     conn_dev_list[3..6]
 *
 *                4               0           conn_dev_list[0..3]           none
 *                4               1           conn_dev_list[0..3]     conn_dev_list[4]
 *                4               2           conn_dev_list[0..3]     conn_dev_list[4..5]
 *                4               3           conn_dev_list[0..3]     conn_dev_list[4..6]
 *                4               4           conn_dev_list[0..3]     conn_dev_list[4..7]
 */
_attribute_ble_data_retention_  dev_char_info_t conn_dev_list[DEVICE_CHAR_INFO_MAX_NUM];
```

Figure 3.72: conn_dev_list definition

When a connection is established with another device, the identity information of the peer device is stored in the conn_dev_list[] by calling dev_char_info_insert_by_conn_event() in the connection complete event.

```
int app_le_connection_complete_event_handle(u8 *p)
{
    hci_le_connectionCompleteEvt_t *pConnEvt = (hci_le_connectionCompleteEvt_t *)p;

    if(pConnEvt->status == BLE_SUCCESS){

        dev_char_info_insert_by_conn_event(pConnEvt);

        if( pConnEvt->role == LL_ROLE_MASTER ) // master role, process SMP and SDP if necessary
        {
            #if (BLE_MASTER_SMP_ENABLE)
            #else
                //manual pairing, device match, add this device to slave mac table
                if(blm_manPair.manual_pair && blm_manPair.mac_type == pConnEvt->peerAddrType && !memcmp(blm_manPair.mac, pConnEvt->peerAddr, 6)){
                    blm_manPair.manual_pair = 0;
                    user_tbl_slave_mac_add(pConnEvt->peerAddrType, pConnEvt->peerAddr);
                }
            #endif
```

Figure 3.73: Connection completed event handle

If you are the Master and Simple SDP is enabled, you will first check if the GATT Service table of the peer device is already in Flash via dev_char_info_search_peer_att_handle_by_peer_mac(), and if so, retrieve it directly from Flash Place in conn_dev_list[] via dev_char_info_add_peer_att_handle().

app.c

```
                    user_tbl_slave_mac_add(pConnEvt->peerAddrType, pConnEvt->peerAddr);
                }
            #endif



            #if (BLE_MASTER_SIMPLE_SDP_ENABLE)
                memset(&cur_sdp_device, 0, sizeof(dev_char_info_t));
                cur_sdp_device.conn_handle = pConnEvt->connHandle;
                cur_sdp_device.peer_adrType = pConnEvt->peerAddrType;
                memcpy(cur_sdp_device.peer_addr, pConnEvt->peerAddr, 6);

                u8  temp_buff[sizeof(dev_att_t)];
                dev_att_t *pdev_att = (dev_att_t *)temp_buff;

                /* att_handle search in flash, if success, load char_handle directly from flash, no need SDP again */
                if( dev_char_info_search_peer_att_handle_by_peer_mac(pConnEvt->peerAddrType, pConnEvt->peerAddr, pdev_att) ){
                    //cur_sdp_device.char_handle[1] =                           //Speaker
                    cur_sdp_device.char_handle[2] = pdev_att->char_handle[2];   //OTA
                    cur_sdp_device.char_handle[3] = pdev_att->char_handle[3];   //consume report
                    cur_sdp_device.char_handle[4] = pdev_att->char_handle[4];   //normal key report
                    //cur_sdp_device.char_handle[6] =                           //BLE Module, SPP Server to Client
                    //cur_sdp_device.char_handle[7] =                           //BLE Module, SPP Client to Server

                    /* add the peer device att_handle value to conn_dev_list */
                    dev_char_info_add_peer_att_handle(&cur_sdp_device);
                }
                else
                {
                    master_sdp_pending = pConnEvt->connHandle;  // mark this connection need SDP

                    #if (BLE_MASTER_SMP_ENABLE)
                    #else
                        app_register_service(&app_service_discovery);  //No SMP, service discovery can initiated now
                    #endif
                }
            #endif
        }
    }

    return 0;
}
```

Figure 3.74: simpleSDP info in Flash

If not, it will be fetched via app_service_discovery(). Once obtained, the functions dev_char_info_add_peer_att_handle() and dev_char_info_store_peer_att_handle() will be called to store the peer Slave GATT Service table into RAM and FLASH respectively for subsequent use. This is shown in the figure below.

app.c simple_sdp.c

```
/**
 * @brief   SDP handler.
 *          !!! Note: This is a simple SDP processing implemented by telink.
 * @param   none.
 * @return  none.
 */
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_UUID128_NUM) == BLE_SUCCESS)   // service discovery OK
    {
        cur_sdp_device.char_handle[2] = blm_att_findHandleOfUuid128 (db128, my_OtaUUID);            //OTA
        cur_sdp_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 report)
        cur_sdp_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_sdp_device.char_handle[6] = blm_att_findHandleOfUuid128 (db128, TelinkSppDataServer2ClientUUID);      //BLE Module, SPP Server to Client
        //cur_sdp_device.char_handle[7] = blm_att_findHandleOfUuid128 (db128, TelinkSppDataClient2ServerUUID);      //BLE Module, SPP Client to Server

        /* 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_sdp_device);

        /* peer device att_handle value store in flash */
        dev_char_info_store_peer_att_handle(&cur_sdp_device);

        //my_dump_str_data(APP_DUMP_SDP_EN,"OTA handle", (u8*)&cur_sdp_device.char_handle[2], 2);
        //my_dump_str_data(APP_DUMP_SDP_EN,"CMKEY handle", (u8*)&cur_sdp_device.char_handle[3], 2);
        //my_dump_str_data(APP_DUMP_SDP_EN,"KBKEY handle", (u8*)&cur_sdp_device.char_handle[4], 2);
    }

    master_sdp_pending = 0;  //service discovery finish
}
```

Figure 3.75: Service discovery

Note:

> The SDP is a very complicated part. For Telink BLE Multiple Connection SDK, due to limited chip resources, SDP cannot be as complicated as a mobile phone. Given here is a simple reference.

The user can fetch the Attribute handle from the GATT Service table by following the connHandle dev_char_info_search_by_connhandle(), whose return value is a pointer to the conn_dev_list[index] structure, pointing to that element of the conn_dev_list[] array to which the connHandle corresponds.

4 Low Power Management

Low Power Management is also called Power Management, or PM as referred by this document.

4.1 Low Power Driver

4.1.1 Low Power Mode

When MCU works in normal mode, or working mode, current is about 3 ~ 7mA. To save power consumption, MCU should enter low power mode.

There are three low power modes, or sleep modes: Suspend mode, Deepsleep mode, and Deepsleep retention mode.

Table 4.1: Low power mode

Module	suspend	deepsleep retention	deepsleep
Sram	100% keep	first 16K/32K/64K keep, others lost	100% lost
digital register	99% keep	100% lost	100% lost
analog register	100% keep	99% lost	99% lost

The table above illustrates statistically data retention and loss for SRAM, digital registers and analog registers during each sleep mode.

(1) Suspend mode (sleep mode 1)

In this mode, program execution pauses, most hardware modules of MCU are powered off, and the PM module still works normally. In this mode, the IC current of B85m is about 60-70uA and B91 is about 40-50uA. Program execution continues after wakeup from suspend mode

In suspend mode, data of the SRAM, all analog registers and most digital registers are maintained. A few digital registers will power down, involving:

a) A small number of digital registers in the baseband circuit. User should pay close attenton to the registers configured by the API "rf_set_power_level_index()". This API needs to be invoked after each wakeup from suspend mode.

b) The B85m IC also has the digital register that controls the state of the Dfifo. Corresponding to the related APIs in drivers/dfifo.h. When using these APIs, the user must ensure that they are reset after each suspend wake_up.

(2) Deepsleep mode (sleep mode 2)

In this mode, program execution pauses, vast majority of hardware modules are powered off, and the PM module still works. In this mode, IC current is less than 1uA, but if flash standby current comes up at 1uA or so, total current may reach 1~2uA. When deepsleep mode wake_up, the MCU will restart, similar to the effect of power-on, and the program will restart to initialize.

In deepsleep mode, except a few retention analog registers, data of all registers (analog & digital) and SRAM are lost.

(3) Deepsleep retention mode (sleep mode 3)

In deepsleep mode, current is very low, but all SRAM data are lost; while in suspend mode, though SRAM and most registers are non-volatile, current is increased.

Deepsleep with SRAM retention (deepsleep retention or deep retention) mode is designed in the B85m and B91 family, so as to achieve application scenes with low sleep current and quick wakeup to restore state, e.g. maintain BLE connection during long sleep. Corresponding to the size of the SRAM retention area, B85m family involves deepsleep retention 16K Sram and deepsleep retention 32K Sram, B91 family involves deepsleep retention 32K Sram and deepsleep retention 64K Sram,.

Deepsleep retention mode is also a kind of deepsleep. Most of the hardware modules of the MCU are powered off, and the PM hardware modules remain working. Power consumption is the power consumed by retention Sram plus that of deepsleep mode, and the current is between 2~3uA. When deepsleep retention mode wake_up, the MCU will restart and the program will restart to initialize.

Deepsleep retention mode and deepsleep mode are consistent in register state, almost all of them are powered off. Compare with in deepsleep mode, in deepsleep retention mode, the first 16K (or the first 32K) of Sram can be kept without power-off, and the remaining Sram is powered off.

4.1.2 Low Power Wake-up Source

The low-power wake-up source diagram of B85m and B91 MCU is shown below, suspend/ deepsleep/ deepsleep retention can all be awakened by GPIO PAD and timer. In Telink BLE Multiple Connection SDK, only two types of 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;
```

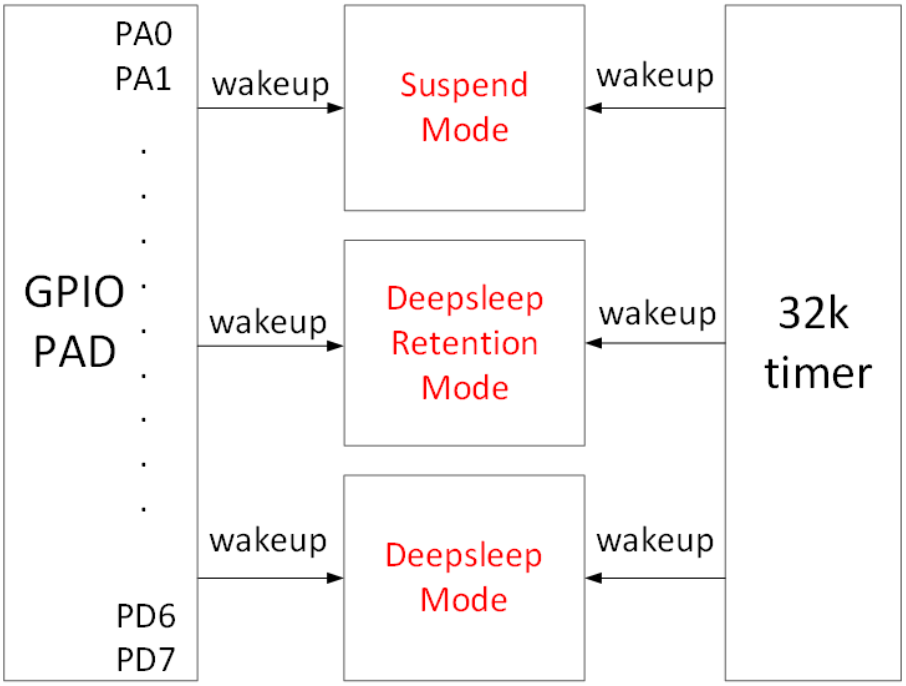


Figure 4.1: MCU HW wakeup source

As shown above, there are two hardware wakeup sources: TIMER and GPIO PAD.

- The "PM_WAKEUP_TIMER" comes from 32k HW timer (32k RC timer or 32k Crystal timer). Since 32k timer is correctly initialized in the SDK, no configuration is needed except setting wakeup source in the "cpu_sleep_wakeup ()".
- The "PM_WAKEUP_PAD" comes from GPIO module. Except 4 MSPI pins, all GPIOs (PAx/PBx/PCx/PDx) support high or low level wakeup.

The API below serves to configure GPIO PAD as wakeup source for sleep mode.

```
typedef enum{
    Level_Low=0,
    Level_High,
} GPIO_LevelTypeDef;
void cpu_set_gpio_wakeup (GPIO_PinTypeDef pin, GPIO_LevelTypeDef pol, int en);
```

- pin: GPIO pin
- pol: wakeup polarity, Level_High: high level wakeup, Level_Low: low level wakeup
- en: 1-enable, 0-disable.

Examples:

```
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 1);  //Enable GPIO_PC2 PAD high level wakeup
cpu_set_gpio_wakeup (GPIO_PC2, Level_High, 0);  //Disable GPIO_PC2 PAD wakeup
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 1);  //Enable GPIO_PB5 PAD low level wakeup
cpu_set_gpio_wakeup (GPIO_PB5, Level_Low, 0);  //Disable GPIO_PB5 PAD wakeup
```

4.1.3 Sleep and Wake-up from Low Power Mode

The stack controls suspend and deepsleep retention in Telink BLE Multiple Connection SDK. It is no recommended for users to set suspend/deepsleep retention by themselves, but users can set deepsleep entry mode.

The API below serves to configure MCU sleep and wakeup.

```
int cpu_sleep_wakeup (SleepMode_TypeDef sleep_mode, SleepWakeupSrc_TypeDef wakeup_src,
unsigned int wakeup_tick);
```

- sleep_mode: This para serves to set sleep mode. Currently, users can only choose deepsleep mode. (suspend and deepsleep retention are controlled by stack.)

```
typedef enum {
    ......
    DEEPSLEEP_MODE          = 0x80,
    ......
}SleepMode_TypeDef;
```

- wakeup_src: This para serves to set wakeup source for suspend/deep retention/deepsleep as one or combination of PM_WAKEUP_PAD and PM_WAKEUP_TIMER. If set as 0, MCU wakeup is disabled for sleep mode.
- wakeup_tick: if PM_WAKEUP_TIMER is assigned as wakeup source, the "wakeup_tick" serves to set MCU wakeup time. If PM_WAKEUP_TIMER is not assigned, this para is negligible.

The "wakeup_tick" is an absolute value, which equals current value of System Timer tick plus intended sleep duration. When System Timer tick reaches the time defined by the wakeup_tick, MCU wakes up from sleep mode. Without taking current System Timer tick value as reference point, wakeup time is uncontrollable.

Since the wakeup_tick is an absolute time, it follows the max range limit of 32bit System Timer tick. In current SDK, 32bit max sleep time corresponds to 7/8 of max System Timer tick. Since max System Timer tick is 268s or so, max sleep time is 268*7/8=234s, which means the "delta_Tick" below should not exceed 234s.

```
cpu_sleep_wakeup(SUSPEND_MODE, PM_WAKEUP_TIMER, clock_time() + delta_tick);
```

The return value is an ensemble of current wakeup sources. Following shows wakeup source for each bit of the return value.

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

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

a) If WAKEUP_STATUS_TIMER bit = 1, wakeup source is Timer.

b) If WAKEUP_STATUS_PAD bit = 1, wakeup source is GPIO PAD.

c) If both WAKEUP_STATUS_TIMER and WAKEUP_STATUS_PAD equal 1, wakeup source is Timer and GPIO PAD.

d) STATUS_GPIO_ERR_NO_ENTER_PM is a special state indicating GPIO wakeup error. E.g. Suppose a GPIO is set as high level PAD wakeup (PM_WAKEUP_PAD). When MCU attempts to invoke the "cpu_sleep_wakeup" to enter suspend, if this GPIO is already at high level, MCU will fail to enter suspend and immediately exit the "cpu_sleep_wakeup" with return value STATUS_GPIO_ERR_NO_ENTER_PM.

Sleep time is typically set in the following way:

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_TIMER,  clock_time() + delta_Tick);
```

The "delta_Tick", a relative time (e.g. 100* CLOCK_16M_SYS_TIMER_CLK_1MS), plus "clock_time()" becomes an absolute time.

Some examples on cpu_sleep_wakeup:

```
cpu_sleep_wakeup (SUSPEND_MODE , PM_WAKEUP_PAD, 0);
```

When it's invoked, MCU enters suspend, and wakeup source is GPIO PAD.

```
cpu_sleep_wakeup (DEEPSLEEP_MODE , PM_WAKEUP_TIMER, clock_time() + 10*
↪  CLOCK_16M_SYS_TIMER_CLK_1MS);
```

When it's invoked, MCU enters deepsleep, wakeup source is timer, and wakeup time is current time plus 10 ms, so the deepsleep duration is 10 ms.

```
cpu_sleep_wakeup (DEEPSLEEP_MODE , PM_WAKEUP_PAD | PM_WAKEUP_TIMER,
                  clock_time() + 50* CLOCK_16M_SYS_TIMER_CLK_1MS);
```

When it's invoked, MCU enters deepsleep, wakeup source includes timer and GPIO PAD, and timer wakeup time is current time plus 50 ms. If GPIO wakeup is triggered before 50 ms expires, MCU will be woke up by GPIO PAD in advance; otherwise, MCU will be woke up by timer.

```
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 Low Power Wake-up Procedure

When user calls the API cpu_sleep_wakeup(), the MCU enters the sleep mode; when the wake-up source triggers the MCU to wake up, the MCU software operation flow is inconsistent for different sleep modes.

The following is a detailed description of the MCU operating process after the suspend, deepsleep, and deepsleep retention three sleep modes are awakened. Please refer to the figure below.

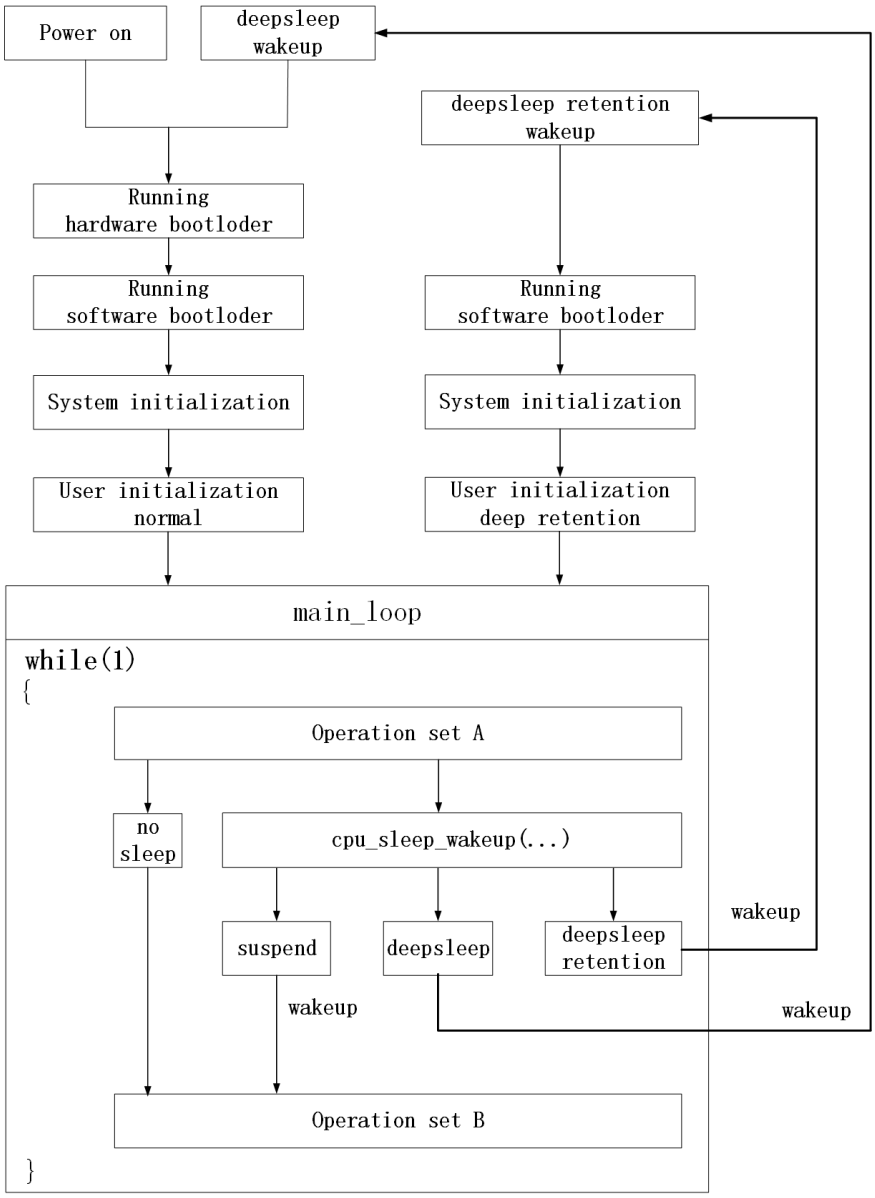


Figure 4.2: Sleep mode wakeup work flow

Detailed process after the MCU is powered on (Power on) is introduced as following:

(1) Run hardware bootloader

It is pure MCU hardware operation without involvement of software.

Couple of examples:

Read the boot flag of flash to determine whether the firmware that should be run currently is stored on flash address 0 or on flash address 0x20000 (related to OTA); read the value of the corresponding location of flash to determine how much data currently needs to be copied from flash to Sram as resident memory data (refer to the introduction of Sram allocation in Chapter 2).

The part of running the hardware bootloader involves copying data from flash to sram, which generally takes a long time to execute. For example, it takes about 5ms to copy 10K data.

(2) Run software bootloader

After the hardware bootloader finishes running, the MCU starts to run the software bootloader. Software bootloader is the vector end introduced earlier.

Software bootloader serves to set up memory environment for C program execution, so it can be regarded as memory initialization.

(3) System initialization

System initialization corresponds to the initialization of each hardware module (including cpu_wakeup_init, rf_drv_init, gpio_init, clock_init) from cpu_wakeup_init to user_init in the main function, and sets the digital/ analog register status of each hardware module.

(4) User initialization

User initialization corresponds to user_init, or user_init_normal/ user_init_deepRetn in the SDK.

(5) main_loop

After User initialization, program enters main_loop inside while(1). The operation is called "Operation Set A" before main_loop enters sleep mode, and called "Operation Set B" after wakeup from sleep.

Analyze the sleep mode flow from the above figure.

(6) no sleep

Without sleep mode, MCU keeps looping inside while(1) between "Operation Set A" -> "Operation Set B".

(7) suspend

If the cpu_sleep_wakeup function is called to enter the suspend mode, when the suspend is woken up, 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 suspend, all Sram data can remain unchanged, and all digital/ analog register states also remain unchanged (with a few special exceptions); after suspend wakes up, the program continues to run in its original position , hardly any sram and register state restoration needs to be considered. The disadvantage of suspend is the high power consumption.

(8) deepsleep

If the cpu_sleep_wakeup function is called to enter deepsleep mode, when deepsleep is woken up, 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 initializations have to be redone.

After the MCU enters deepsleep, all Sram and digital/analog registers (except a few analog registers) are power down, so the power consumption is very low and the MCU current is less than 1uA.

(9) deepsleep retention

If the cpu_sleep_wakeup function is called to enter deepsleep retention mode, when deepsleep retention is woken up, MCU will return to Run software bootloader.

The deepsleep retention is an intermediate sleep mode between suspend and deepsleep.

In suspend, the current is high because it needs to save all sram and register states; deepsleep retention does not need to save the register state, sram only retains the first 16K/32K/64K without power down, so the power consumption is much lower than suspend, only about 2uA.

After deepsleep wake_up, all processes need to be run again, while deepsleep retention can skip the step of "Run hardware bootloader", this is because the data on the first 16K/32K/64K of SRAM is not lost, no need re-copy from flash. However, due to the limited retention area on SRAM, "run software bootloader" cannot be skipped and must be executed; since deepsleep retention cannot save the register state, system initilization must be executed, and the initialization of registers needs to be reset. User initialization deep retention after deepsleep retention wake_up can be optimized and improved to distinguish User initialization normal after processing MCU power on/deepsleep wake_up.

4.1.5 API pm_is_MCU_deepRetentionWakeup

As can be seen from the above figure "sleep mode wakeup work flow", MCU power on, deepsleep wake_up and deepsleep retention wake_up all need to go through Running software bootloader, System initialization, and User initialization. When running system initialization and user initialization, user needs to know whether the current MCU is woke up from deepsleep retention, so as to differentiate from power on and deepsleep wake_up. PM driver provides API for judging whether deepsleep retention wake_up is:

```
int pm_is_MCU_deepRetentionWakeup(void);
```

Return value: 1 - deepsleep retention wake_up; 0 - power on or deepsleep wake_up.

4.2 BLE Low Power Management

4.2.1 BLE PM Initialization

For applications with low power mode, BLE PM module needs to be initialized by following API.

```
void    blc_ll_initPowerManagement_module(void);
```

If low power is not required, DO NOT use this API, so as to skip compiling of related code and variables into program and thus save FW and SRAM space.

4.2.2 BLE PM for Link Layer

Telink BLE Multiple Connection SDK applies low power management to Legacy advertising state, Scanning state, ACL connection master and ACL connection slave.

It should be noted that the SDK currently has restrictions on the use of latency by slave. If the restrictions are not met, packets will be sent and received at each interval. Even if it accepts the connection parameters of the opposite master as a slave, and the latency is not 0, the SDK will send and receive RF data according to the latency of 0.

Restrictions on the use of latency by slave:

(1) Only Legacy advertising and ACL slave tasks.

(2) ACL slave has only 1 connection (later the SDK will be optimized to support more slave connections).

(3) If there is Legacy advertising, the minimum advertising interval needs to be greater than 195 ms.

The SDK does not apply low power management to Idle state either. In Idle state, since there is no RF activity, i.e. the "blt_sdk_main_loop" function is not valid, user can use PM driver for certain low power management.

4.2.2.1 Sleep for Advertising "only advertising"

When only advertising is enabled and scan is turned off, i.e., the Link Layer is in the advertising state, the timing is as follows.

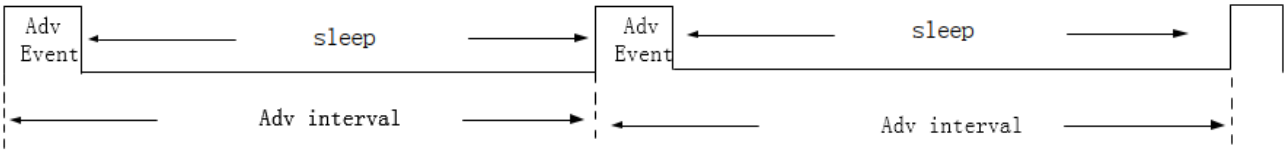


Figure 4.3: Sleep for advertising timing

When the advertising time is reached, it will wake up from sleep and then process the advertising event. After processing, stack will determine the difference between the time point of the next Adv Event and the current time, and if the condition is met, it will go into sleep to reduce power consumption. The time consumed by the Adv Event is related to the specific situation, for example: the users only set 37channel, ADV packet length is relatively small, in channel 37 or 38 received SCAN_REQ or CONNECT_IND, etc.

4.2.2.2 Sleep for scanning "only scanning"

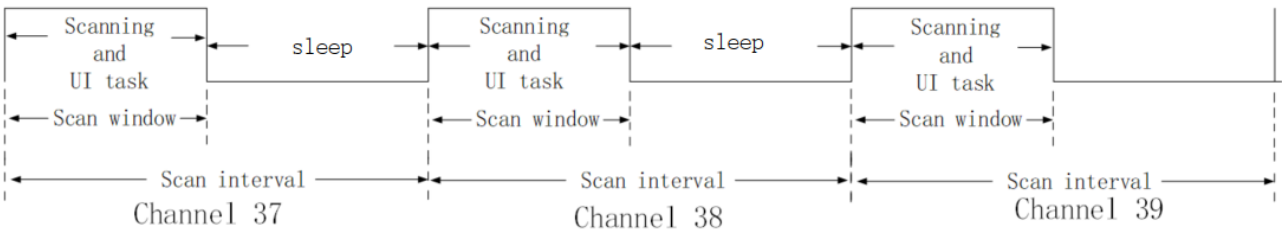


Figure 4.4: Sleep for scanning for only scanning

The actual scanning 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 scanning; if the Scan window is less than the Scan interval, from the front part of the Scan interval to allocate time to scanning, equivalent time reference Scan window.

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 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. During the last 60% of the time, the MCU enters sleep to reduce the power consumption of the whole machine.

The API for setting the percentage is as follows.

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_200MS, SCAN_WINDOW_50MS,
↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
```

4.2.2.3 Sleep for connection

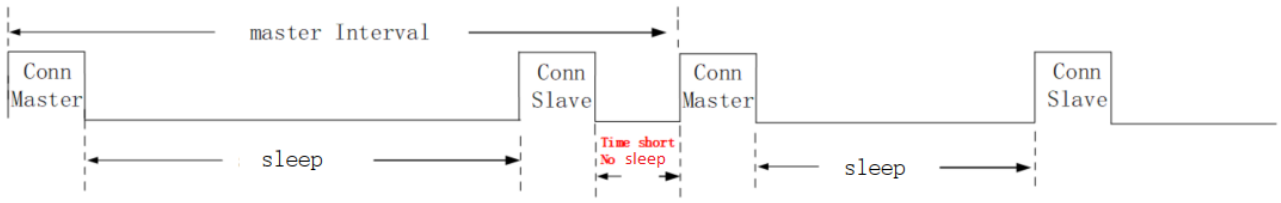


Figure 4.5: Sleep for connection

The conditions for entering sleep are:

(1) The time interval between the next task and the end of the current task;

(2) Whether there is unprocessed data in the RX FIFO;

(3) The execution of BRX POST and BTX POST is completed;

(4) The device itself does not have any event pending.

If the time interval from the next task is relatively large, and there is no data in the RX FIFO, and no event pending, when the BRX POST or BTX POST is executed, the bottom layer will let the MCU enter sleep. When the next task is about to come, the timer wakes up the MCU to start the task.

4.2.3 BLE PM Variables

The variables in this section are helpful to understand BLE PM software flow.

The struct "st_ll_pm_t" is defined in Telink BLE Multiple Connection SDK. Following lists some variables of the struct which will be used by PM APIs.

```
typedef struct {
    u8  deepRt_en;
    u8  deepRet_type;
```

```
    u8  wakeup_src;
    u16 sleep_mask;
    u16 user_latency;
    u32 deepRet_thresTick;
    u32 deepRet_earlyWakeupTick;
    u32 sleep_taskMask;
    u32 next_task_tick;
    u32 current_wakeup_tick;
}st_llms_pm_t;

st_llms_pm_t  blmsPm;
```

Note:

> The above structure variable is encapsulated in the library. The definitions given here are only for the convenience of the following introduction. Users are not allowed to perform any operations on this structure variable.

Variables like "blmsPm.sleep_mask" will appear frequently in the following introduction.

4.2.4 API blc_pm_setSleepMask

The APIs below serve to configure low power management.

```
void blc_pm_setSleepMask (sleep_mask_t mask);
```

The "blmsPm.sleep_mask" is set by the "blc_pm_setSleepMask" and its default value is PM_SLEEP_DISABLE.

Following shows source code of the API.

```
void    blc_pm_setSleepMask (sleep_mask_t mask)
{
    u32 r = irq_disable();
    ......
    blmsPm.sleep_mask = mask;
    ......
    u32 r = irq_disable();
}
```

The "blmsPm.sleep_mask" can be set as any one or the "or-operation" of following values:

```
typedef enum {
    PM_SLEEP_DISABLE        = 0,
    PM_SLEEP_LEG_ADV        = BIT(0),
    PM_SLEEP_LEG_SCAN       = BIT(1),
```

```
    PM_SLEEP_ACL_SLAVE      = BIT(2),
    PM_SLEEP_ACL_MASTER     = BIT(3),
}sleep_mask_t;
```

PM_SLEEP_DISABLE means sleep is disabled which stops MCU to enter sleep.

PM_SLEEP_LEG_ADV and PM_SLEEP_LEG_SCAN decide whether MCU at Legacy advertising state and Scanning state can enter sleep.

PM_SLEEP_ACL_SLAVE and PM_SLEEP_ACL_MASTER decide whether MCU at ACL connection slave and ACL connection master can enter sleep.

Following shows 2 typical use cases:

(1) blc_pm_setSleepMask(PM_SLEEP_DISABLE);

MCU will not enter sleep。

(2) blc_pm_setSleepMask(PM_SLEEP_LEG_ADV | PM_SLEEP_LEG_SCAN | PM_SLEEP_ACL_SLAVE | PM_SLEEP_ ACL_MASTER);

At Legacy advertising state, Scanning state, ACL connection slave and ACL connection master, MCU can enter sleep.

4.2.5 API blc_pm_setWakeupSource

User can set the blc_pm_setSleepMask to enable MCU to enter sleep mode (suspend or deepsleep retention), and use the following API to set wakeup source.

```
void  blc_pm_setWakeupSource (SleepWakeupSrc_TypeDef wakeup_src)
{
    blmsPm.wakeup_src = (u8)wakeup_src;
}
```

wakeup_src: Wakeup source, can be set as PM_WAKEUP_PAD.

This API sets the bottom-layer variable "blmsPm.wakeup_src".

When MCU enters sleep mode at Legacy advertising state, Scanning state, ACL connection master and ACL connection slave, its actual wakeup source is:

```
blmsPm.wakeup_src | PM_WAKEUP_TIMER
```

So PM_WAKEUP_TIMER is mandatory, not depending on user setup. This guarantees that MCU will wake up at specified time to handle ADV task, SCAN task, master task and slave task.

Everytime wakeup source is set by the "blc_pm_setWakeupSource", after MCU wakes up from sleep mode, the blmsPm.wakeup_src is set to 0.

4.2.6 API blc_pm_setDeepsleepRetentionType

Deepsleep retention further separates into 16K/32K/64 sram retention. When the deepsleep retention mode in sleep mode takes effect, the SDK will enter the corresponding deepsleep retention mode according to the settings.

Only two modes available for B85m, 16K and 32K. The default deepsleep retention mode of the SDK is DEEPSLEEP_MODE_RET_SRAM_LOW32K:

```
typedef enum {
    DEEPSLEEP_MODE_RET_SRAM_LOW16K      = 0x43,
    DEEPSLEEP_MODE_RET_SRAM_LOW32K      = 0x07,
}SleepMode_TypeDef;
```

Only two modes available for B91, 32K and 64K. The default deepsleep retention mode of the SDK is DEEPSLEEP_MODE_RET_SRAM_LOW64K:

```
typedef enum {
    DEEPSLEEP_MODE_RET_SRAM_LOW32K      = 0x21,
    DEEPSLEEP_MODE_RET_SRAM_LOW64K      = 0x03,
}SleepMode_TypeDef;
```

When entering deepsleep retention mode, the following API can be set to decide which sub-mode to enter. Since the current SDK uses the maximum retention sram size by default, users basically do not use it.

```
void blc_pm_setDeepsleepRetentionType(SleepMode_TypeDef sleep_type)
{
    blmsPm.deepRet_type = sleep_type;
}
```

Note:

> The API must be called after the blc_ll_initPowerManagement_module to take effect.

4.2.7 API blc_pm_setDeepsleepRetentionEnable

This API is used to enable deepsleep retention mode.

```
typedef enum {
    PM_DeepRetn_Disable = 0x00,
    PM_DeepRetn_Enable  = 0x01,
} deep_retn_en_t;

void blc_pm_setDeepsleepRetentionEnable (deep_retn_en_t en)
{
    blmsPm.deepRt_en = en;
}
```

4.2.8 API blc_pm_setDeepsleepRetentionThreshold

In the presence of a BLE task, suspend will be automatically switched to deepsleep retention if the following conditions are met:

```
//Determine whether sleep mode is suspend mode or deepsleep retention mode
SleepMode_TypeDef  sleep_M = SUSPEND_MODE;
if( blmsPm.deepRt_en && (u32)(blmsPm.current_wakeup_tick - clock_time() -
 ↪  blmsPm.deepRet_thresTick) < BIT(30) ){
    sleep_M = (SleepMode_TypeDef)blmsPm.deepRet_type;
}
```

The first condition, blmsPm.deepRt_en, needs to be enabled by calling the API blc_pm_ setDeepsleepRetentionEnable, which has been introduced earlier.

The second condition (u32)(blmsPm.current_wakeup_tick - clock_time() - blmsPm.deepRet_thresTick) < BIT(30), which means that the duration of sleep (ie wakeup time minus real-time time) exceeds a specific time threshold (ie blmsPm .deepRet_thresTick), the sleep mode of the MCU will automatically switch from suspend to deepsleep retention.

The API blc_pm_setDeepsleepRetentionThreshold is used to set the time threshold for suspend to switch to the deepsleep retention trigger condition. This design is to pursue lower power consumption.

```
void blc_pm_setDeepsleepRetentionThreshold(u32 thres_ms)
{
    blmsPm.deepRet_thresTick = thres_ms * SYSTEM_TIMER_TICK_1MS;
}
```

4.2.9 PM Software Processing Flow

The software processing flow of low power management is described below using a combination of code and pseudo-code, in order to let the user understand all the logical details of the processing flow.

4.2.9.1 blc_sdk_main_loop

In Telink BLE Multiple Connection SDK, "blc_sdk_main_loop" is called repeatedly in a while(1) structure.

```
while(1)
{
    ////////////////////////////////// BLE entry ////////////////////////////////
    blc_sdk_main_loop();

    ////////////////////////////////// UI entry ////////////////////////////////
    // UI  task

    ////////////////////////////////// PM entry ////////////////////////////////
```

```
    blc_pm_setSleepMask( PM_SUSPEND_ADV | PM_SUSPEND_SCAN | PM_SUSPEND_SLAVE |
↪   PM_SUSPEND_MASTER);
}
```

The blc_sdk_main_loop function is executed continuously in while(1), and the code for BLE low-power management is in the blc_sdk_main_loop function, so the code for low-power management is also executed all the time.

Following shows the implementation of BLE PM logic inside the "blc_sdk_main_loop".

```
void blc_sdk_main_loop (void)
{
    ......
    if( blmsPm.sleep_mask == PM_SLEEP_DISABLE )
    {
        return; // PM_SLEEP_DISABLE, can not enter sleep mode;sleep time
    }

    if( !tick1_exceed_tick2(blmsPm.next_task_tick, clock_time() + PM_MIN_SLEEP_US) )
    {
        return; //too short, can not enter sleep mode.
    }

    if( bltSche.task_mask && (blmsPm.sleep_taskMask & bltSche.task_mask) != bltSche.task_mask )
    //Whether there is a task (adv, scan, master, slave)
    //Whether sleep_taskMask allows this state (adv, scan, master, slave) to enter sleep
    {
        return;
    }

    if ( (brx_post | btx_post | adv_post | scan_post) == 0 )
    {
        return; //Sleep can only be allowed after each task is completed
    }
    else
    {
        blt_sleep_process(); //process sleep & wakeup
    }
    ......
}
```

(1) When the "bltmsPm.sleep_mask" is PM_SLEEP_DISABLE, the SW directly exits without executing the "blt_sleep_process" function. So when using the "blc_pm_setSleepMask(PM_SLEEP_DISABLE)", PM logic is completely ineffective; MCU will never enter sleep and the SW always execute while(1) loop.

(2) If the sleep time is too short, it will not enter sleep.

(3) When there are tasks, such as adv task, scan task, master task, slave task, but if the sleep_taskMask of the corresponding task is not enabled, it will not enter low power mode.

(4) If the Adv Event or Scan Event or Btx Event of Conn state Master role or Brx Event of Conn state Slave role is being executed, the "blt_sleep_process" function will not be executed, this is because RF task is running at this time, and the SDK needs to ensure that the sleep mode can only be entered after the Adv Event/Scan Event/Btx Event/Brx Event ends.

Only when both cases above are valid, the blt_sleep_process will be executed.

4.2.9.2 blt_sleep_process

Following shows logic implementation of the "blt_sleep_process" function.

```
void blt_sleep_process (void)
{
    ......
    blmsPm.current_wakeup_tick = blmsPm.next_task_tick;//Record wake-up time

    //Execute the BLT_EV_FLAG_SLEEP_ENTER callback function
    blt_p_event_callback (BLT_EV_FLAG_SLEEP_ENTER, NULL, 0);

    //Enter low power function
    u32 wakeup_src = cpu_sleep_wakeup (sleep_M, PM_WAKEUP_TIMER | blmsPm.wakeup_src,
↪   blmsPm.current_wakeup_tick);

    //Execute the BLT_EV_FLAG_SUSPEND_EXIT callback function
    blt_p_event_callback (BLT_EV_FLAG_SUSPEND_EXIT, (u8 *)&wakeup_src, 1);

    blmsPm.wakeup_src = 0;
    ......
}
```

The above is a brief flow of the blt_sleep_process function. Here we see the execution timing of the two sleep-related event callback functions: BLT_EV_FLAG_SLEEP_ENTER, BLT_EV_FLAG_SUSPEND_EXIT.

Regarding how to enter sleep mode, the API cpu_sleep_wakeup in the driver is finally called:

```
cpu_sleep_wakeup(SleepMode_TypeDef sleep_mode,  SleepWakeupSrc_TypeDef wakeup_src, unsigned int
↪   wakeup_tick);
```

This API sets wakeup source as PM_WAKEUP_TIMER | blmsPm.wakeup_src, so Timer wakeup is mandatory to guarantee MCU wakeup before next task.

When exiting the "blt_sleep_process" function, the "blmsPm.wakeup_src" reset. So the API "blc_pm_ setWakeupSource" is only effective for the latest sleep mode.

4.2.10 API blc_pm_getWakeupSystemTick

The following API is used to get the sleep wakeup time point (System Timer tick) for low-power management calculations, i.e. T_wakeup.

```
u32 blc_pm_getWakeupSystemTick (void);
```

The calculation of T_wakeup is close to the processing of the cpu_sleep_wakeup function, and the application layer can only get the accurate T_wakeup in the BLT_EV_FLAG_SLEEP_ENTER event callback function.

Suppose the user needs to press the key to wake up when the sleep time is relatively long. Below we explain the setting method.

We need to use the BLT_EV_FLAG_SLEEP_ENTER event callback function and blc_pm_getWakeupSystemTick.

The callback registration method of BLT_EV_FLAG_SLEEP_ENTER is as follows:

```
blc_ll_registerTelinkControllerEventCallback (BLT_EV_FLAG_SLEEP_ENTER, &app_set_kb_wakeup);
_attribute_ram_code_ void app_set_kb_wakeup (u8 e, u8 *p, int n)
{
    /* sleep time > 100ms. add GPIO wake_up */
    if(((u32)(blc_pm_getWakeupSystemTick() - clock_time())) > 100 * SYSTEM_TIMER_TICK_1MS){
        blc_pm_setWakeupSource(PM_WAKEUP_PAD);  //GPIO PAD wake_up
    }
}
```

For the above example, if the sleep time exceeds 100 ms, add GPIO wakeup. User can adjust it according to the actual situation.

Here just provides an interface, and the user decides whether to use it according to the actual situation.

4.3 Issues in GPIO Wake-up

4.3.1 Fail to enter sleep mode when wake-up level is valid

In Telink MCU, GPIO wakeup is level triggered instead of edge triggered, so when GPIO PAD is configured as wakeup source, for example, suspend wakeup triggered by GPIO high level, MCU needs to make sure when MCU invokes cpu_sleep_wakeup to enter suspend, that the wakeup GPIO is not at high level. Otherwise, once entering cpu_sleep_wakeup, it would exit immediately and fail to enter suspend.

If the above situation occurs, it may cause unexpected problems, for example, it was intended to enter deepsleep and be woken up and the program re-executed, but it turns out that the MCU cannot enter deepsleep, resulting in the code continuing to run, not in the state we expected, and the whole flow of the program may be messed up.

User should pay attention to avoid this problem when using Telink's GPIO PAD to wake up.

If the APP layer does not avoid this problem, and GPIO PAD wakeup source is already effective at invoking of cpu_sleep_wakeup, PM driver makes some improvement to avoid flow mess:

(1) suspend & deepsleep retention mode

For both suspend and deepsleep retention mode, the SW will fast exit cpu_sleep_wakeup with two potential return values:

- Return WAKEUP_STATUS_PAD if the PM module has detected effective GPIO PAD state.
- Return STATUS_GPIO_ERR_NO_ENTER_PM if the PM module has not detected effective GPIO PAD state.

(2) deepsleep mode

For deepsleep mode, PM diver will reset MCU automatically in bottom layer (equivalent to watchdog reset). The SW restarts from "Run hardware bootloader".

4.4 Timer Wake-up by Application Layer

BLE task exists and without GPIO PAD wakeup, once MCU enters sleep mode, it only wakes up at T_wakeup pre-determined by SDK. User can not wake up MCU at an earlier time which might be needed at certain scenario. To provide more flexibility, application layer wakeup and associated callback function are added in the SDK:

Application layer wakeup API:

```
void blc_pm_setAppWakeupLowPower(u32 wakeup_tick, u8 enable);
```

"wakeup_tick" is wakeup time at System Timer tick value.

"enable": 1-wakeup is enabled; 0-wakeup is disabled.

Registered call back function blc_pm_registerAppWakeupLowPowerCb is executed at application layer wakeup:

```
typedef void (*pm_appWakeupLowPower_callback_t)(int);
pm_appWakeupLowPower_callback_t  pm_appWakeupLowPowerCb = NULL;
void blc_pm_registerAppWakeupLowPowerCb(pm_appWakeupLowPower_callback_t cb)
{
    pm_appWakeupLowPowerCb = cb;
}
```

Take Conn state Slave role as an example:

When the user uses blc_pm_setAppWakeupLowPower to set the app_wakeup_tick for the application layer to wake up regularly, the SDK will check whether app_wakeup_tick is before T_wakeup before entering sleep.

(1) If app_wakeup_tick is before T_wakeup, as shown in the figure below, it will trigger sleep in app_wakeup_tick to wake up early;

(2) If app_wakeup_tick is after T_wakeup, MCU will still wake up at T_wakeup.

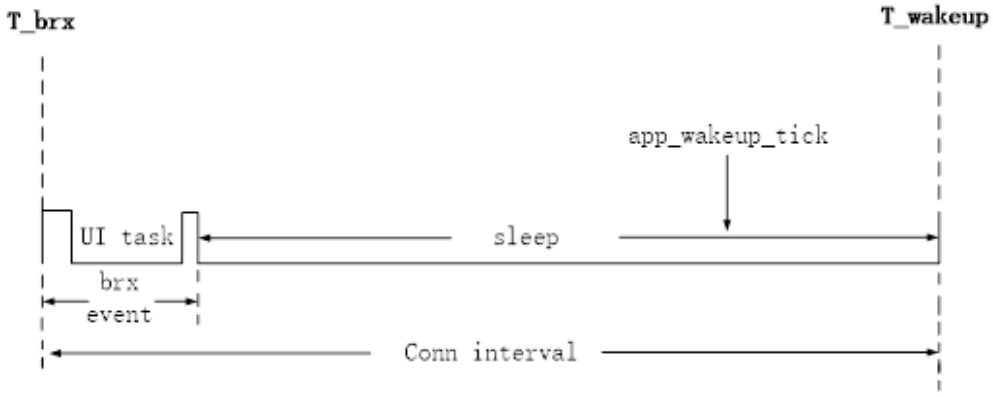


Figure 4.6: Early wake_up at app_wakup_tick

5 Low Battery Detect

The low power detection function is not yet open in Telink BLE Multiple Connection SDK.

6 OTA

Regarding OTA, the whole process is exactly the same as single connection SDK, please refer to the introduction of the corresponding chapters in Telink BLE Single Connection SDK Handbook, including B85m series (825x + 827x) and B91 series.

Compared with Telink BLE Single Connection SDK, Telink BLE Multiple Connection SDK adds a restriction: OTA is only allowed on one slave connection at the same time.

7 Key Scan

This section can refer to the introduction of the corresponding chapters in the Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x) and the B91 series.

8 LED Management

This section can refer to the introduction of the corresponding chapters in the Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x) and the B91 series.

9 Software Timer

This section can refer to the introduction of the corresponding chapters in the Telink BLE Single Connection SDK Handbook, including the B85m series (825x + 827x) and the B91 series.

10 Feature Demos

This section will introduce the usage and phenomenon of each function under sdk/vendor/xx_feature, and users can refer to the code implementation to add the required function to their own code.

10.1 feature_backup

- **Function**: Demonstration of BLE basic functions. This includes advertising, passive scanning, connectivity, and etc. The demo also serves as a relatively "clean" base version for users developing BLE applications (with SMP, SDP, etc. disabled by default).
- **Main hardware**: B91 development board x 2

Take Telink B91 BLE Multiple Connection SDK as an example, the application layer code is under eagle_ble_sdk/vendor/B91_feature/feature_backup, you need to modify the definition in eagle_ble_sdk/vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                   TEST_FEATURE_BACKUP
```

to activate this part of the code. The code sets the advertising parameters, advertising content, scan response content, scan parameters, and the configuration to enable advertising and enable scan, refer to the following in the initialization code.

```
//////////////////////////// User Configuration for BLE application ////////////////////////////
    blc_ll_setAdvData( (u8 *)tbl_advData, sizeof(tbl_advData) );
    blc_ll_setScanRspData( (u8 *)tbl_scanRsp, sizeof(tbl_scanRsp));
    blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED, OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_NONE);
    blc_ll_setAdvEnable(BLC_ADV_ENABLE);  //ADV enable

    blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS, OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
    blc_ll_setScanEnable (BLC_SCAN_ENABLE, DUP_FILTER_DISABLE);
}
```

Figure 10.1: Configure advertising scanning parameters in the initialization

Compile and burn the generated firmware into each of the two development boards. After powering up, the two devices named "feature" can be scanned by scanning the advertising packets, which can be connected by other Master or Slave devices, respectively. To interconnect the two devices, press SW4 on one of the boards to initiate the connection, and read the list of variable values on the left side through the Tdebug tab of the BDT tool (see the Debug method chapter for details on how to use it), and see that the value of conn_master_num for that board is 1.

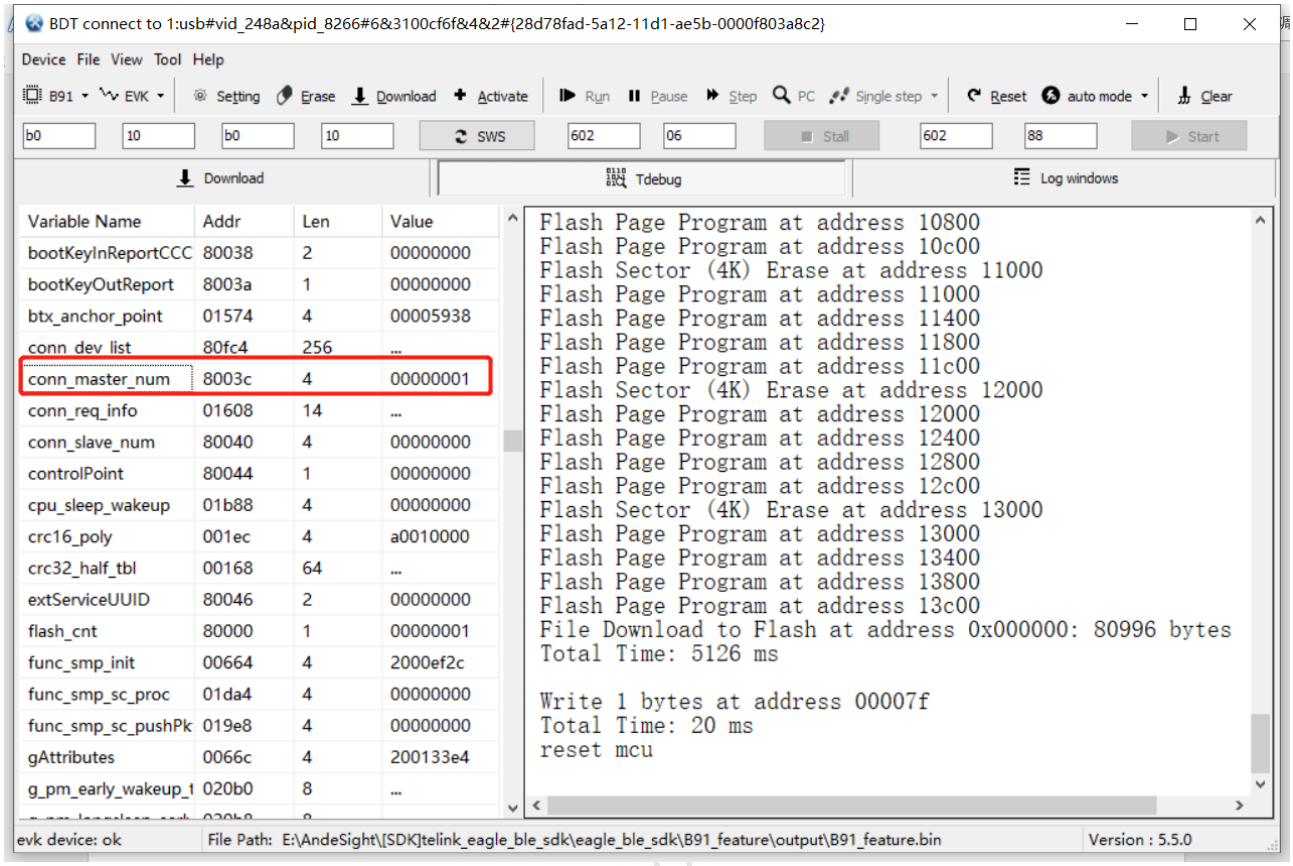


Figure 10.2: conn_master_num=1 in Tdebug

Check the value of the conn_slave_num variable on the other development board, which also has a value of 1.

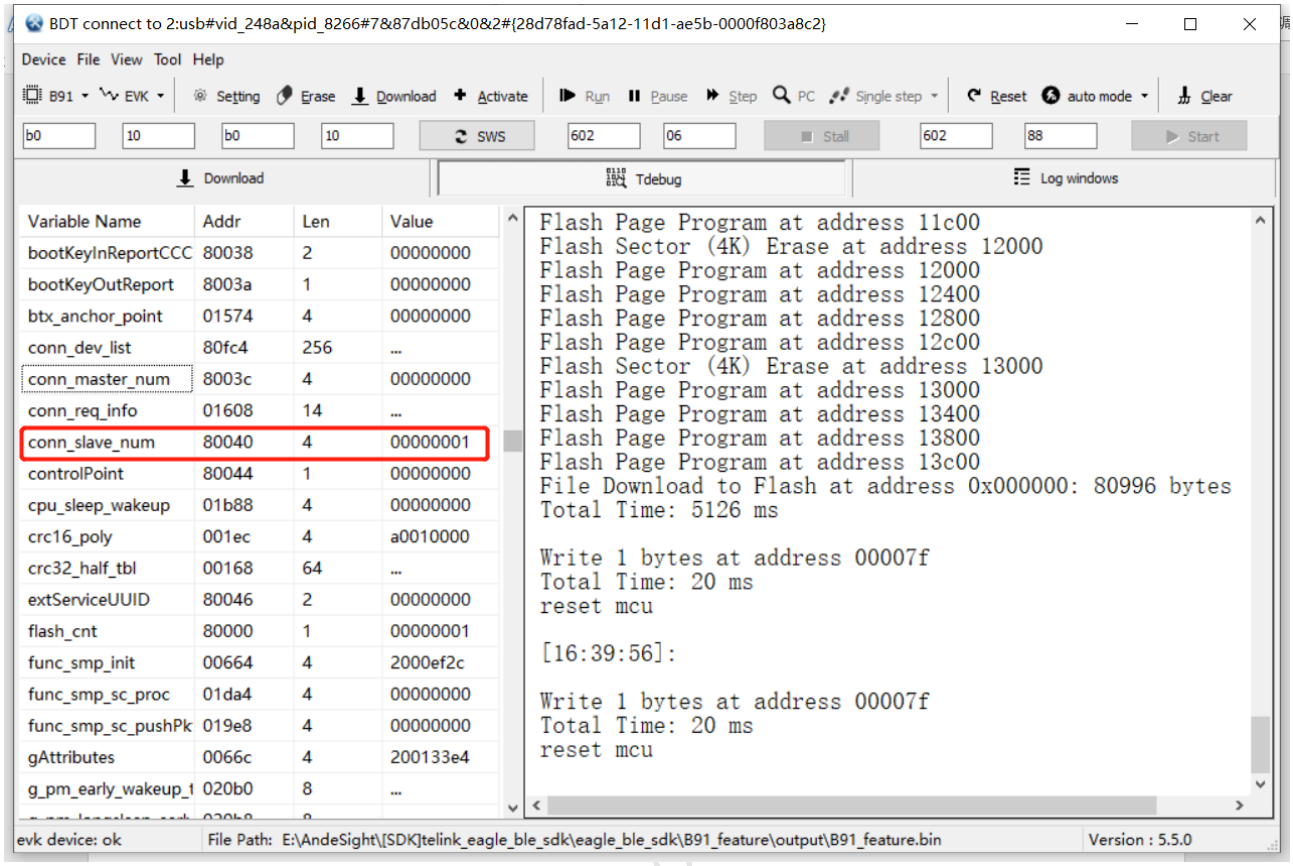


Figure 10.3: conn_slave_num=1 in Tdebug

means the connection between the two is successful.

10.2 feature_2M_coded_phy

- **Function:** BLE 1M/2M/Coded PHY function demonstration.
- **Main hardware:** B91 development board x 2

Take Telink B91 BLE Multiple Connection SDK as an example, the application layer code is under eagle_ble_sdk/ vendor/ B91_feature/ feature_2M_coded_phy, you need to modify the definition in eagle_ble_sdk/vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                TEST_2M_CODED_PHY_CONNECTION
```

to activate this part of the code to do a demo of dynamically switching PHYs.

Initialization call blc_ll_init2MPhyCodedPhy_feature() to enable the PHY switching function. Dynamic switching of PHYs is implemented in feature_2m_phy_test_mainloop() in main_loop().

(1) After establishing the connection for the Master, do a WriteCmd to each Slave every 1s, with a valid data length of 8 bytes (the actual sending interval is also related to the Connection Interval).

(2) After establishing the connection for the Slave, notify each Master every 1s, with a valid data length of 8 bytes (the actual sending interval is also related to the Connection Interval).

(3) Do a PHY switch every 10s for each connection, in the following order: Coded_S8 → 2M → 1M → Coded_S8...

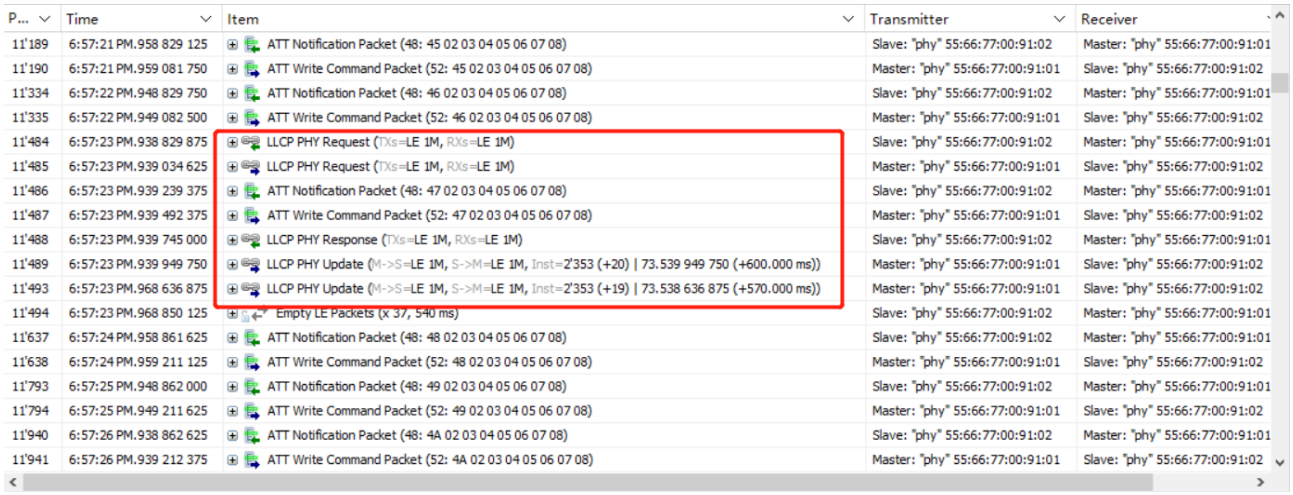
The actual packet capture is as follows.

P...	Time	Item	Transmitter	Receiver
80'690	7:05:24 PM.315 799 125	LLCP Reserved (0x68)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
80'727	7:05:24 PM.616 258 750	ATT Write Command Packet (52: B5 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
80'739	7:05:24 PM.706 029 250	ATT Notification Packet (48: B6 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
80'853	7:05:25 PM.606 259 375	ATT Write Command Packet (52: B6 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
80'872	7:05:25 PM.696 029 625	ATT Notification Packet (48: B7 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'003	7:05:26 PM.626 260 125	ATT Write Command Packet (52: B7 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'013	7:05:26 PM.686 030 000	ATT Notification Packet (48: B8 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'144	7:05:27 PM.616 260 500	ATT Write Command Packet (52: B8 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'154	7:05:27 PM.676 260 625	LLCP PHY Request (TXs=LE Coded, RXs=LE Coded)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'161	7:05:27 PM.706 030 500	LLCP PHY Response (TXs=LE Coded, RXs=LE Coded)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'163	7:05:27 PM.706 513 625	ATT Notification Packet (48: B9 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'166	7:05:27 PM.736 260 625	LLCP PHY Update (M->S=LE Coded, S->M=LE Coded, Inst=14'688 (+19) | 557.306 260 625 (+570.000...	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'167	7:05:27 PM.736 530 000	Empty LE Packets (x 37, 539 ms)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'274	7:05:28 PM.607 538 875	ATT Write Command Packet (52: B9 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'290	7:05:28 PM.696 669 500	ATT Notification Packet (48: BA 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'422	7:05:29 PM.627 539 375	ATT Write Command Packet (52: BA 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
81'432	7:05:29 PM.688 408 750	ATT Notification Packet (48: BB 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
81'571	7:05:30 PM.617 539 875	ATT Write Command Packet (52: BB 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02

Figure 10.4: feature_2M_coded_phy Coded Req

P...	Time	Item	Transmitter	Receiver
83'506	7:05:44 PM.627 547 750	ATT Write Command Packet (52: C9 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'516	7:05:44 PM.686 678 375	ATT Notification Packet (48: CA 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'656	7:05:45 PM.617 548 250	ATT Write Command Packet (52: CA 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'670	7:05:45 PM.706 679 125	ATT Notification Packet (48: CB 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'789	7:05:46 PM.607 548 750	ATT Write Command Packet (52: CB 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'804	7:05:46 PM.696 679 375	ATT Notification Packet (48: CC 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'937	7:05:47 PM.627 549 500	ATT Write Command Packet (52: CC 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'943	7:05:47 PM.657 549 500	LLCP PHY Request (TXs=LE 2M, RXs=LE 2M)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'949	7:05:47 PM.686 680 000	LLCP PHY Response (TXs=LE 2M, RXs=LE 2M)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'951	7:05:47 PM.688 610 500	ATT Notification Packet (48: CD 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
83'955	7:05:47 PM.717 549 625	LLCP PHY Update (M->S=LE 2M, S->M=LE 2M, Inst=15'354 (+19) | 577.287 549 625 (+570.000 ms))	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
83'956	7:05:47 PM.718 738 875	Empty LE Packets (x 37, 538 ms)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
84'078	7:05:48 PM.616 204 375	ATT Write Command Packet (52: CD 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
84'086	7:05:48 PM.706 011 125	ATT Notification Packet (48: CE 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
84'208	7:05:49 PM.606 205 000	ATT Write Command Packet (52: CE 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
84'225	7:05:49 PM.696 011 750	ATT Notification Packet (48: CF 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
84'356	7:05:50 PM.626 205 375	ATT Write Command Packet (52: CF 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
84'364	7:05:50 PM.686 012 375	ATT Notification Packet (48: D0 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01

Figure 10.5: feature_2M_coded_phy 2M Req



P...	Time	Item	Transmitter	Receiver
11'189	6:57:21 PM.958 829 125	ATT Notification Packet (48: 45 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'190	6:57:21 PM.959 081 750	ATT Write Command Packet (52: 45 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'334	6:57:22 PM.948 829 750	ATT Notification Packet (48: 46 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'335	6:57:22 PM.949 082 500	ATT Write Command Packet (52: 46 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'484	6:57:23 PM.938 829 875	LLCP PHY Request (TXs=LE 1M, RXs=LE 1M)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'485	6:57:23 PM.939 034 625	LLCP PHY Request (TXs=LE 1M, RXs=LE 1M)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'486	6:57:23 PM.939 239 375	ATT Notification Packet (48: 47 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'487	6:57:23 PM.939 492 375	ATT Write Command Packet (52: 47 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'488	6:57:23 PM.939 745 000	LLCP PHY Response (TXs=LE 1M, RXs=LE 1M)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'489	6:57:23 PM.939 949 750	LLCP PHY Update (M->S=LE 1M, S->M=LE 1M, Inst=2'353 (+20) | 73.539 949 750 (+600.000 ms))	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'493	6:57:23 PM.968 636 875	LLCP PHY Update (M->S=LE 1M, S->M=LE 1M, Inst=2'353 (+19) | 73.538 636 875 (+570.000 ms))	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'494	6:57:23 PM.968 850 125	Empty LE Packets (x 37, 540 ms)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'637	6:57:24 PM.958 861 625	ATT Notification Packet (48: 48 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'638	6:57:24 PM.959 211 125	ATT Write Command Packet (52: 48 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'793	6:57:25 PM.948 862 000	ATT Notification Packet (48: 49 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'794	6:57:25 PM.949 211 625	ATT Write Command Packet (52: 49 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02
11'940	6:57:26 PM.938 862 625	ATT Notification Packet (48: 4A 02 03 04 05 06 07 08)	Slave: "phy" 55:66:77:00:91:02	Master: "phy" 55:66:77:00:91:01
11'941	6:57:26 PM.939 212 375	ATT Write Command Packet (52: 4A 02 03 04 05 06 07 08)	Master: "phy" 55:66:77:00:91:01	Slave: "phy" 55:66:77:00:91:02

Figure 10.6: feature_2M_coded_phy 1M Req

10.3 feature_gatt_api

- **Function**: BLE GATT command function demonstration and API usage. These commands are used in the reference implementation of the SDP process for some of the examples in the BLE Multiple Connection SDK, and users can use this demo code for single instruction testing.
- **Main hardware**: B91 development board x 2

Take Telink B91 BLE Multiple Connection SDK as an example, the application layer code is under eagle_ble_sdk/vendor/B91_feature/feature_gatt_api, you need to modify the definition in eagle_ble_sdk/vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                TEST_GATT_API
```

to activate this part of the code. In app.c, test the different GATT commands by modifying the definition of TEST_API.

Compile and burn the generated firmware into the two development boards. When powered on, the red light on the development board toggles on and off every 2s. The connection is triggered by pressing the SW4 button on one of the boards (as Master), and by capturing the packets, we can see that every 2s, the Master sends a test command and the Slave replies accordingly. The implementation of the reply refers to the function app_gatt_data_handler().

The following is the packet capture when TEST_API is defined separately for different command test definitions.

P...	Time	Item	Payl...	Transmitter	Receiver
1	5:02:05 PM.005 955 125	Connectable ("gatt" 55:66:77:00:91:01, Initiator "gatt" 55:66:77:00:9...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
4	5:02:05 PM.019 693 125	Connectable ("gatt" 55:66:77:00:91:02, 1.59 min)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
3'242	5:02:23 PM.139 076 625	SMP Security Request (Bonding)	2 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
3'245	5:02:23 PM.155 342 250	Connectable ("gatt" 55:66:77:00:91:01, 1.29 min)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
3'288	5:02:23 PM.326 346 625	ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
3'294	5:02:23 PM.357 366 375	ATT Read By Group Type Response Packet (12 03 00 00 2A › 02 05 00 ...	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
3'297	5:02:23 PM.388 387 125	Empty LE Packets (x 124, 2 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
3'659	5:02:25 PM.326 337 750	ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
3'666	5:02:25 PM.357 357 500	ATT Read By Group Type Response Packet (12 03 00 00 2A › 02 05 00 ...	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
3'671	5:02:25 PM.388 378 250	Empty LE Packets (x 124, 2 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'018	5:02:27 PM.326 328 875	ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'024	5:02:27 PM.357 348 625	ATT Read By Group Type Response Packet (12 03 00 00 2A › 02 05 00 ...	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
4'031	5:02:27 PM.388 369 375	Empty LE Packets (x 125, 1 retry, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'398	5:02:29 PM.326 320 000	ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'408	5:02:29 PM.357 339 625	ATT Read By Group Type Response Packet (12 03 00 00 2A › 02 05 00 ...	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
4'412	5:02:29 PM.388 360 375	Empty LE Packets (x 122, 4 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'782	5:02:31 PM.326 310 625	ATT Read By Group Type Request Packet (1 - Max Handle, Characterist...		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
4'789	5:02:31 PM.357 330 625	ATT Read By Group Type Response Packet (12 03 00 00 2A › 02 05 00 ...	18 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
4'793	5:02:31 PM.388 351 750	Empty LE Packets (x 118, 7 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 10.7: GATT API test TEST_READ_BY_GROUP_TYPE_REQ

P...	Time	Item	Payl...	Transmitter	Receiver
1	5:10:45 PM.000 336 750	Connectable ("gatt" 55:66:77:00:91:02, 9.77 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
297	5:10:47 PM.745 293 125	Connectable ("gatt" 55:66:77:00:91:01, Initiator "gatt" 55:66:77:00:9...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
314	5:10:47 PM.839 464 625	SMP Security Request (Bonding)	2 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
315	5:10:47 PM.854 043 250	Connectable ("gatt" 55:66:77:00:91:01, 6.92 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
322	5:10:47 PM.870 485 250	ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
323	5:10:47 PM.870 788 125	Empty LE Packets (x 119, 8 retries, 2 s)		Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
704	5:10:49 PM.870 476 375	ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
710	5:10:49 PM.901 496 375	ATT Find Information Response Packet (Primary Service › Characteristic...	20 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
714	5:10:49 PM.932 517 125	Empty LE Packets (x 118, 8 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'064	5:10:51 PM.870 467 750	ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'070	5:10:51 PM.901 487 500	ATT Find Information Response Packet (Primary Service › Characteristic...	20 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'074	5:10:51 PM.932 508 250	Empty LE Packets (x 121, 5 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'451	5:10:53 PM.870 459 125	ATT Find Information Request Packet (1 - Max Handle)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'458	5:10:53 PM.901 478 625	ATT Find Information Response Packet (Primary Service › Characteristic...	20 byte...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02

Figure 10.8: GATT API test TEST_FIND_INFO_REQ

P...	Time	Item	Payl...	Transmitter	Receiver
1	5:17:54 PM.023 646 375	Connectable ("gatt" 55:66:77:00:91:02, 14.7 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
21	5:17:54 PM.223 858 750	Connectable ("gatt" 55:66:77:00:91:01, Initiator "gatt" 55:66:77:00:9...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
42	5:17:54 PM.351 704 375	SMP Security Request (Bonding)	2 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
43	5:17:54 PM.364 521 250	Connectable ("gatt" 55:66:77:00:91:01, 14.2 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
48	5:17:54 PM.382 265 500	Empty LE Packets (x 88, 1.38 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
331	5:17:55 PM.757 718 625	ATT Find By Type Value Request Packet (1 - Max Handle, PnP ID, Sourc...	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
337	5:17:55 PM.788 738 250	ATT Find By Type Value Response Packet (14 - 14)	4 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
342	5:17:55 PM.819 759 000	Empty LE Packets (x 125, 1 retry, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
723	5:17:57 PM.757 709 375	ATT Find By Type Value Request Packet (1 - Max Handle, PnP ID, Sourc...	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
729	5:17:57 PM.788 729 250	ATT Find By Type Value Response Packet (14 - 14)	4 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
733	5:17:57 PM.819 750 000	Empty LE Packets (x 123, 5 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'104	5:17:59 PM.757 700 125	ATT Find By Type Value Request Packet (1 - Max Handle, PnP ID, Sourc...	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'111	5:17:59 PM.788 720 375	ATT Find By Type Value Response Packet (14 - 14)	4 bytes ...	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'187	5:18:00 PM.163 488 875	LLCP LE Power Control Response (Delta=-82 dB, Tx Power=-95 dBm)	33 byte...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'192	5:18:00 PM.194 739 250	Empty LE Packets (x 101, 1.56 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'502	5:18:01 PM.757 691 500	ATT Find By Type Value Request Packet (1 - Max Handle, PnP ID, Sourc...	7 bytes ...	Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 10.9: GATT API test TEST_FIND_BY_TYPE_VALUE_REQ

P...	Time	Item	Payload	Transmitter	Receiver
1	5:19:31 PM.029 298 125	Connectable ("gatt" 55:66:77:00:91:01, Initiator...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
225	5:19:33 PM.484 844 750	Connectable ("gatt" 55:66:77:00:91:02, 30.2 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
254	5:19:33 PM.650 559 500	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
258	5:19:33 PM.663 049 375	Connectable ("gatt" 55:66:77:00:91:01, 30 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
260	5:19:33 PM.681 121 125	Empty LE Packets (x 118, 1.81 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
605	5:19:35 PM.494 072 125	ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
612	5:19:35 PM.525 091 750	ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
616	5:19:35 PM.556 112 625	Empty LE Packets (x 124, 2 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
963	5:19:37 PM.494 063 125	ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
969	5:19:37 PM.525 083 000	ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
974	5:19:37 PM.556 103 625	Empty LE Packets (x 118, 8 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'332	5:19:39 PM.494 054 125	ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'338	5:19:39 PM.525 074 000	ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'342	5:19:39 PM.556 094 625	Empty LE Packets (x 117, 9 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'687	5:19:41 PM.494 044 875	ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'697	5:19:41 PM.525 064 750	ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'701	5:19:41 PM.556 085 750	Empty LE Packets (x 119, 6 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
2'037	5:19:43 PM.494 036 000	ATT Read Request Packet (3)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
2'042	5:19:43 PM.525 056 000	ATT Read Response Packet ("feature")	7 bytes (66 65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
2'047	5:19:43 PM.556 076 750	Empty LE Packets (x 117, 9 retries, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 10.10: GATT API test TEST_READ_REQ

P...	Time	Item	Payload	Transmitter	Receiver
1	5:28:50 PM.004 054 625	Connectable ("gatt" 55:66:77:00:91:01, Initiator...		Master: "gatt" 55:66:77:00:91:01	Slave: "gatt" 55:66:77:00:91:02
202	5:28:52 PM.364 372 625	Connectable ("gatt" 55:66:77:00:91:02, 9.64 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "Scanning Device"
213	5:28:52 PM.442 587 875	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
214	5:28:52 PM.457 181 375	Connectable ("gatt" 55:66:77:00:91:01, 9.54 s)		Master: "gatt" 55:66:77:00:91:01	Slave: "Scanning Device"
423	5:28:53 PM.473 145 125	L2CAP SDU (Basic)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
595	5:28:54 PM.379 850 125	ATT Read Blob Request Packet (3 @1)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
601	5:28:54 PM.410 869 750	ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
608	5:28:54 PM.441 890 625	Empty LE Packets (x 126, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
961	5:28:56 PM.379 841 125	ATT Read Blob Request Packet (3 @1)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
969	5:28:56 PM.410 860 875	ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
974	5:28:56 PM.441 881 500	Empty LE Packets (x 125, 1 retry, 1.94 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'320	5:28:58 PM.379 832 000	ATT Read Blob Request Packet (3 @1)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'326	5:28:58 PM.410 851 625	ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'332	5:28:58 PM.441 872 250	Empty LE Packets (x 113, 11 retries, 1.91 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01
1'659	5:29:00 PM.379 823 000	ATT Read Blob Request Packet (3 @1)		Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'665	5:29:00 PM.410 842 750	ATT Read Blob Response Packet (6 bytes)	6 bytes (65 61 74 75 72 65)	Slave: "gatt" 55:66:77:00:91:01	Master: "gatt" 55:66:77:00:91:02
1'673	5:29:00 PM.441 863 375	Empty LE Packets (x 85, 7 retries, 1.47 s)		Master: "gatt" 55:66:77:00:91:02	Slave: "gatt" 55:66:77:00:91:01

Figure 10.11: GATT API test TEST_READ_BLOB_REQ

10.4 feature_ll_more_data

- **Function:** demonstration of BLE MD=1. MD, More Data, is a flag bit MD flag in the data channel PDU Header. At the same time, the demo also provides users to do throughput testing.
- **Main hardware:** B91 development board x 2

Take Telink B91 BLE Multiple Connection SDK as an example, the application layer code is under eagle_ble_sdk/vendor/B91_feature/feature_ll_more_data, you need to modify the definition in eagle_ble_sdk/vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                TEST_LL_MD
```

to activate this part of the code.

Compile and burn the generated firmware into each of the two development boards. Power on the board and press SW4 on one of the boards (as Master) to trigger the connection. When the connection is successful, the red light will be on (if more than one Slave is connected, the red, white, green and blue lights will be on respectively). By pressing SW3 of the Master, you can see from the packet capture that the Master keeps sending WriteCmd to the Slave and the MD flag in its packet is set to 1, which means the next packet is ready to be sent.

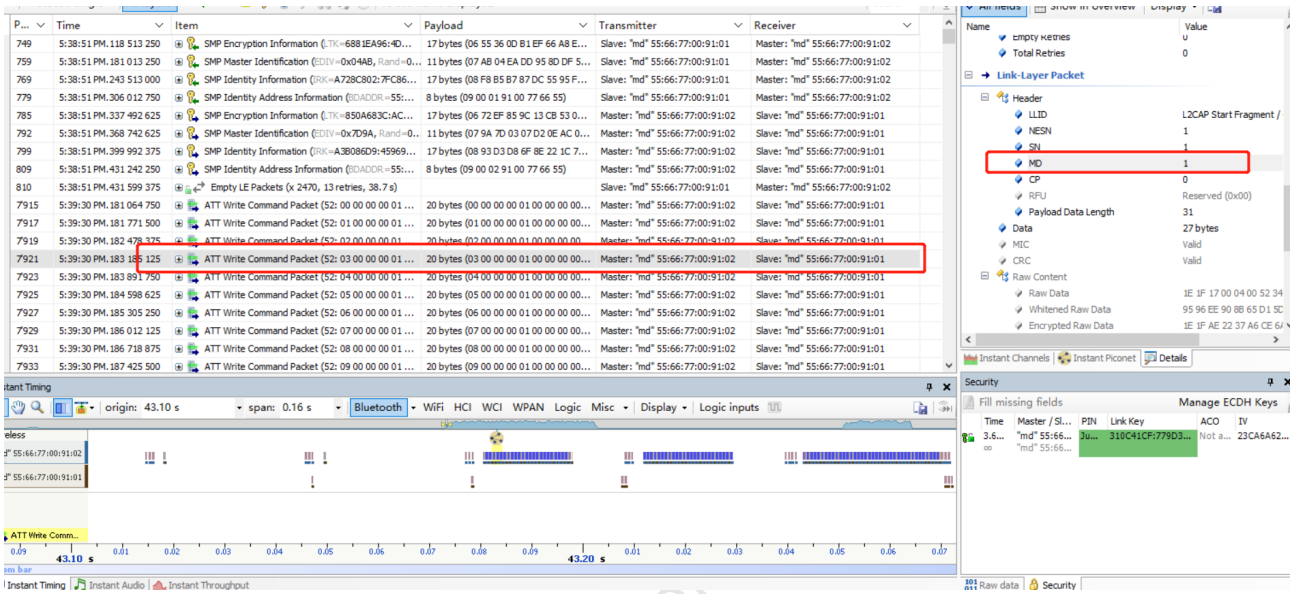


Figure 10.12: feature_ll_more_data WirteCmd

Press the Master board's key SW2 to stop sending.

Press the SW3 on the Slave board, you can see from the packet capture that the Slave board keeps sending Notify packets to the Master board, where the MD flag is set to 1.

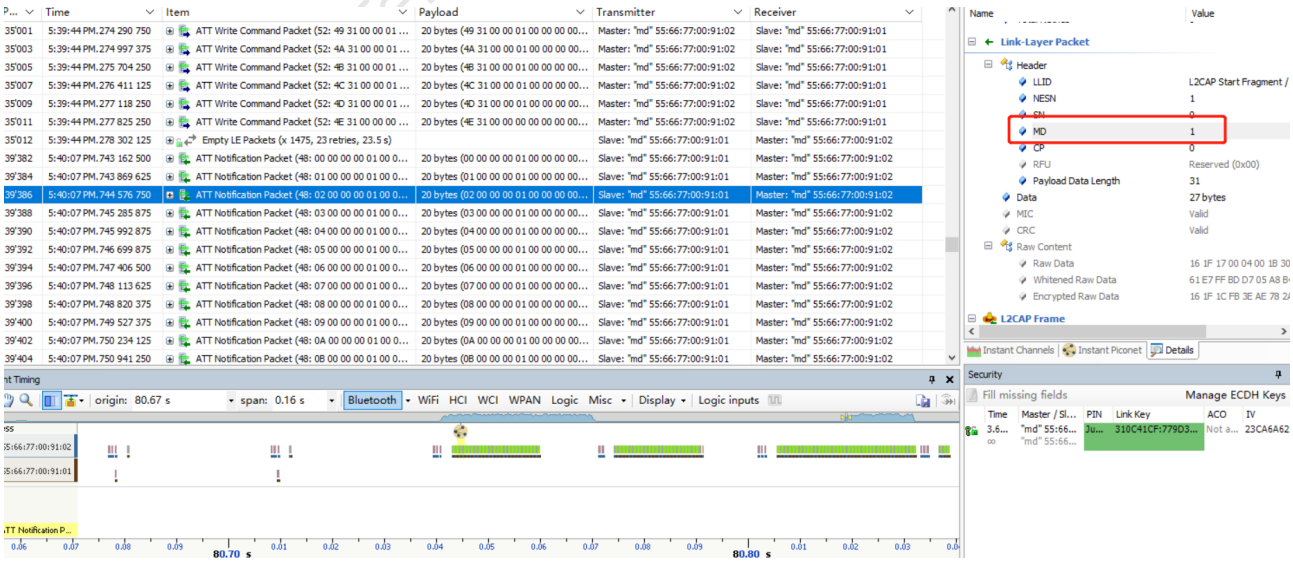


Figure 10.13: feature_ll_more_data Notify

Press key SW2 of the Slave board to stop transmitting.

10.5 feature_dle

- **Function**: Demonstration of BLE DLE (Data Length Extension) and MTU Exchange and L2CAP packet splitting and grouping features.
- **Main hardware**: B91 development board x 2

Take Telink B91 BLE Multiple Connection SDK as an example, the application layer code is under eagle_ble_sdk/vendor/B91_feature/feature_dle, you need to modify the definition in eagle_ble_sdk/vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                   TEST_LL_DLE
```

to activate this part of the code and do a demonstration of DLE and MTU.

Modify DataLength by modifying the definition of DLE_LENGTH_SELECT, as demonstrated by:

(1) The test is triggered by pressing two buttons at the same time.

(2) After the Master board triggers the test, each connection sends a WriteCmd every 1s.

(3) After the Slave board triggers the test, each connection sends a Notify every 1s.

The packet capture is as follows:

P...	Time	Item	Payl...	Transmitter	Receiver
186	7:29:39 PM.776 199 250	SMP Pairing Confirm (Ca=1183DCCD:F3676F57:F030F5AF:A1B32D05)	17 byte...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
191	7:29:39 PM.807 219 125	SMP Pairing Confirm (Cb=2C9954BC:FA9032AE:60809925:9064F84F)	17 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
196	7:29:39 PM.838 699 250	SMP Pairing Random (Na=A12FD88F:DEB97684:C2677EF1:AB6C5596)	17 byte...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
201	7:29:39 PM.869 719 000	SMP Pairing Random (Nb=E08992DA:6A7356BF:6F203F79:A1FC6519)	17 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
206	7:29:39 PM.901 201 000	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, S...	23 byte...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
213	7:29:39 PM.932 221 750	LLCP Encryption Response (SKDs=0x51A60FAF408D8DB8, IVs=0x49E9...	13 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
224	7:29:40 PM.025 969 250	LLCP Start Encryption Request	1 byte (...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
228	7:29:40 PM.057 449 750	LLCP Start Encryption Response	1 byte (...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
232	7:29:40 PM.088 469 250	LLCP Start Encryption Response	1 byte (...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
234	7:29:40 PM.088 968 375	LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes,...	9 bytes ...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
236	7:29:40 PM.089 531 625	ATT Exchange MTU Request Packet		Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
241	7:29:40 PM.119 949 250	LLCP Length Request (MaxRx=200 bytes, 1.71 ms, MaxTx=200 bytes,...	9 bytes ...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
243	7:29:40 PM.120 512 625	ATT Exchange MTU Request Packet		Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
245	7:29:40 PM.121 059 500	LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 byte...	9 bytes ...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
247	7:29:40 PM.121 623 000	ATT Exchange MTU Response Packet		Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
252	7:29:40 PM.150 969 375	SMP Encryption Information (LTK=B5DCC78F:3F2603EA:3A756A2C:F4...	17 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
254	7:29:40 PM.151 628 875	LLCP Length Response (MaxRx=200 bytes, 1.71 ms, MaxTx=200 byte...	9 bytes ...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
256	7:29:40 PM.152 192 250	ATT Exchange MTU Response Packet		Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
262	7:29:40 PM.213 469 375	SMP Master Identification (EDIV=0xC4F7, Rand=0xA82178CB488BD234)	11 byte...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01

Figure 10.14: feature_dle DLE & MTU Exchange

P...	Time	Item	Payl...	Transmitter	Receiver
371	7:29:41 PM. 119 949 625	ATT Write Command Packet (52: E3 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
379	7:29:41 PM. 150 969 875	ATT Notification Packet (48: 2C 00 00 00 00 00 00 00 00 00 00 00 0...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
492	7:29:42 PM. 119 950 250	ATT Write Command Packet (52: E4 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
500	7:29:42 PM. 150 970 250	ATT Notification Packet (48: 2D 00 00 00 00 00 00 00 00 00 00 00 0...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
610	7:29:43 PM. 119 950 625	ATT Write Command Packet (52: E5 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
618	7:29:43 PM. 150 970 750	ATT Notification Packet (48: 2E 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
729	7:29:44 PM. 119 951 125	ATT Write Command Packet (52: E6 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
734	7:29:44 PM. 150 971 500	ATT Notification Packet (48: 2F 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
836	7:29:45 PM. 119 951 875	ATT Write Command Packet (52: E7 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
843	7:29:45 PM. 150 971 875	ATT Notification Packet (48: 30 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
957	7:29:46 PM. 119 952 375	ATT Write Command Packet (52: E8 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
964	7:29:46 PM. 150 972 125	ATT Notification Packet (48: 31 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'078	7:29:47 PM. 119 953 000	ATT Write Command Packet (52: E9 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
1'085	7:29:47 PM. 150 972 875	ATT Notification Packet (48: 32 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'197	7:29:48 PM. 119 953 125	ATT Write Command Packet (52: EA 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
1'205	7:29:48 PM. 150 973 250	ATT Notification Packet (48: 33 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'317	7:29:49 PM. 119 954 000	ATT Write Command Packet (52: EB 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
1'323	7:29:49 PM. 150 973 750	ATT Notification Packet (48: 34 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:01
1'434	7:29:50 PM. 119 954 375	ATT Write Command Packet (52: EC 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02

Figure 10.15: Master and Slave make WriteCmd and Notify respectively

Since the DLE is smaller than the MTU, you can see the effect of packet splitting.

970 250	ATT Notification Packet (48: 2D 00 00 00 00 00 00 00 00 00 00 00 0...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:0
9 950 625	ATT Write Command Packet (52: E5 00 00 00 00 00 00 00 00 00 00 00 ...	244 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
9 950 625	L2CAP SDU (Basic, Service=ATT)	247 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
9 950 625	L2CAP B-Frame (Service=ATT)	247 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
9 950 625	Start/Complete LE-U Transfer	200 byt...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
2 042 000	Continuation LE-U Transfer	51 byte...	Master: "dle" 55:66:77:00:91:01	Slave: "dle" 55:66:77:00:91:02
0 970 750	ATT Notification Packet (48: 2E 00 00 00 00 00 00 00 00 00 00 00 00...	244 byt...	Slave: "dle" 55:66:77:00:91:02	Master: "dle" 55:66:77:00:91:0

Figure 10.16: Packet-split sending

10.6 feature_smp

- **Function:** function demonstration of BLE SMP (Security Manager Protocol)
- **Main hardware:** B91 development board x 2

Take Telink B91 BLE Multiple Connection SDK as an example, the application layer code is under eagle_ble_sdk/vendor/B91_feature/feature_smp, you need to modify the definition in eagle_ble_sdk/vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                TEST_SMP
```

to activate this part of the code, by default the SMP function is not enabled for both Slave and Master.

Description:

(1) Since we demonstrate multiple encryption configurations under SMP, for user's reference, we define the SMP_TEST_MODE macro in feature_smp/app_config.h. Users only need to modify the definition of this macro to implement the demo code for different encryption configurations.

(2) When configuring SMP related parameters, if the Master and Slave configurations are the same, use the API without suffix, if different, use the API with _master/_slave suffix as required. The demo code

uses the API with suffix for the SMP encryption enablement demonstration in order to facilitate the user to configure it separately for testing.

(3) To facilitate the observation of the phenomenon, the code includes the display of LED indicators, defined as follows:

a) Green: ON: Master connected; OFF: Master disconnected.

b) Red: ON: Slave connected; OFF: Slave disconnected.

c) Blue: ON: Pair Succeeded; OFF: Disconnected, no Pair process and off.

d) White: ON: Encryption succeeded; OFF: Disconnected.

(4) For the convenience of debugging, the code uses the DEBUG mode of GPIO analog serial port to output Log by default, and PA0(Tx) needs to be connected to the input pin of the display device, and the baud rate is configured to 115200.

(5) **Tips:** The pairing method for Security Connections requires MTU >= 65.

10.6.1 Both Slave and Master do not enable SMP

To disable the demonstration of the SMP function for Slave and Master, define it in feature_smp/app_config.h (default):

```
#define SMP_TEST_MODE           SMP_TEST_NOT_SUPPORT
```

Compile B91_feature，burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards. Press SW4 (Start Connection) on one of the development boards (as Master) and you will see green and red lights on the two development boards respectively, which means the connection is successful.

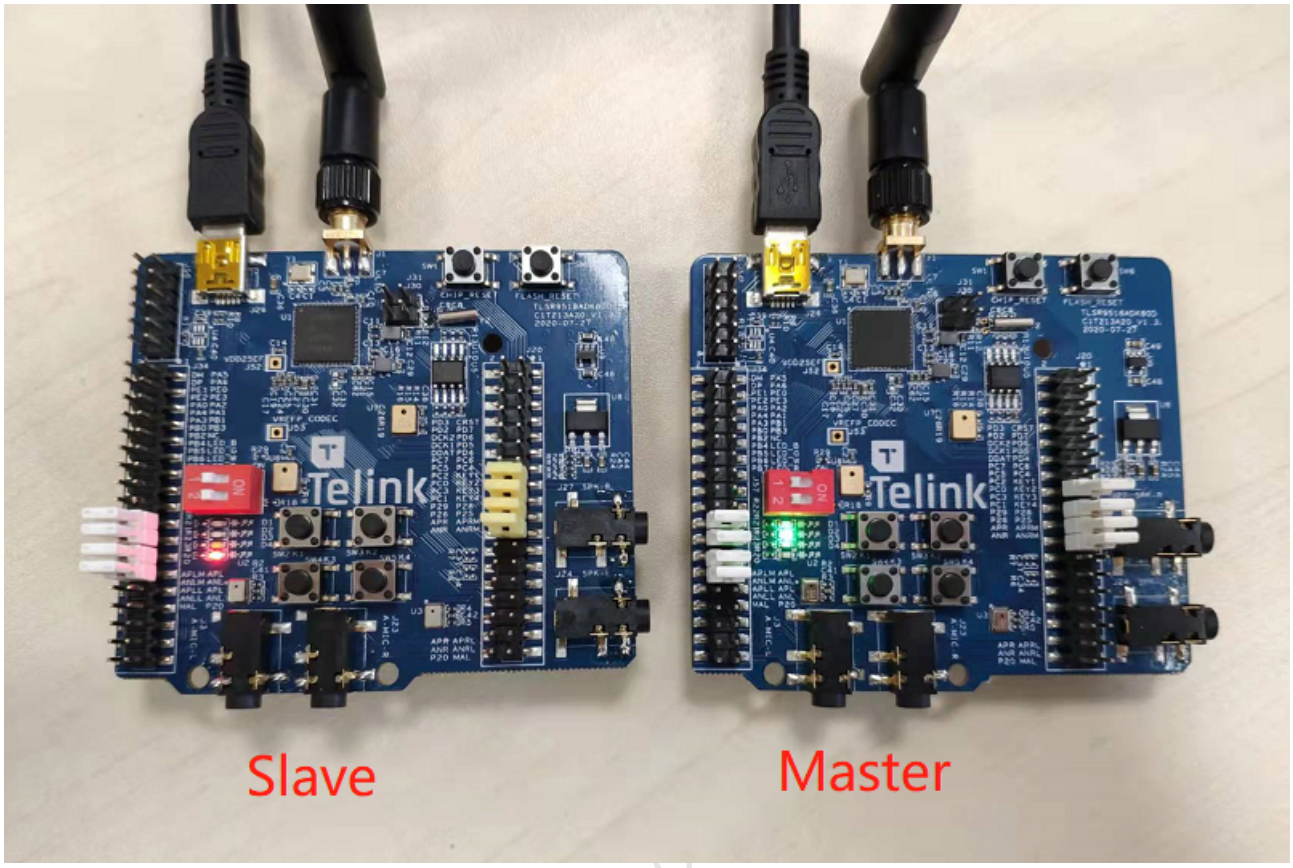


Figure 10.17: Slave and Master connect successfully when SMP is disabled

The packet capture is as follows:

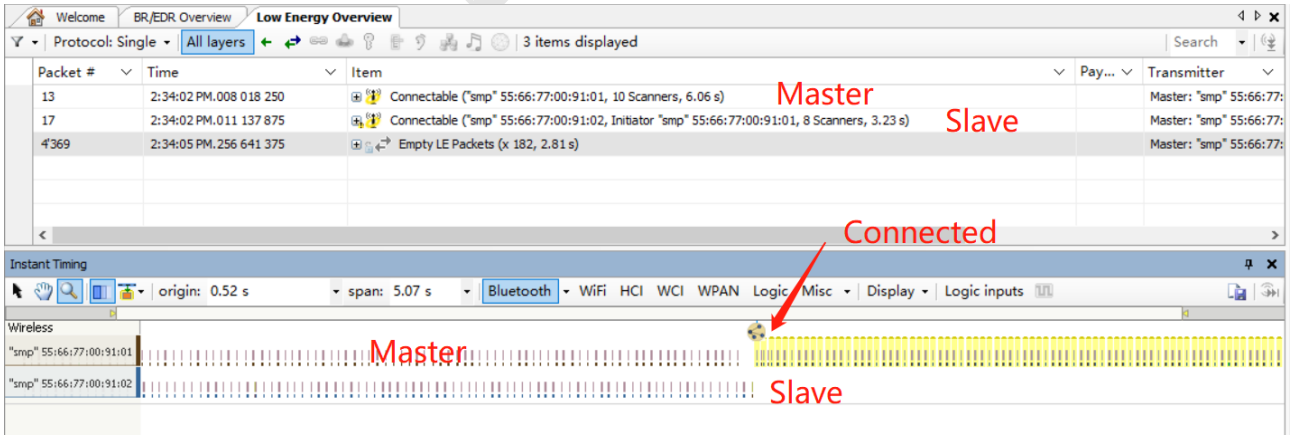


Figure 10.18: Packet capture of successful Slave and Master connection when SMP is disabled

Both of them advertising separately first, and after the connection is established, the device working as the master stops sending advertisings.

Description:

Initialize the device with appMaxSlaveNum set to 1. When currentMaxSlaveNum is equal to app-

MaxSlaveNum, it will stop advertising. If you want the connection to be established and still advertising, modify the following parameters in the initialization.

```
blc_ll_setMaxConnectionNumber( 1, 1);
blc_ll_setAclConnMaxOctetsNumber(ACL_CONN_MAX_
```

Figure 10.19: Modify appMaxSlaveNum

And the packet capture is as follows:

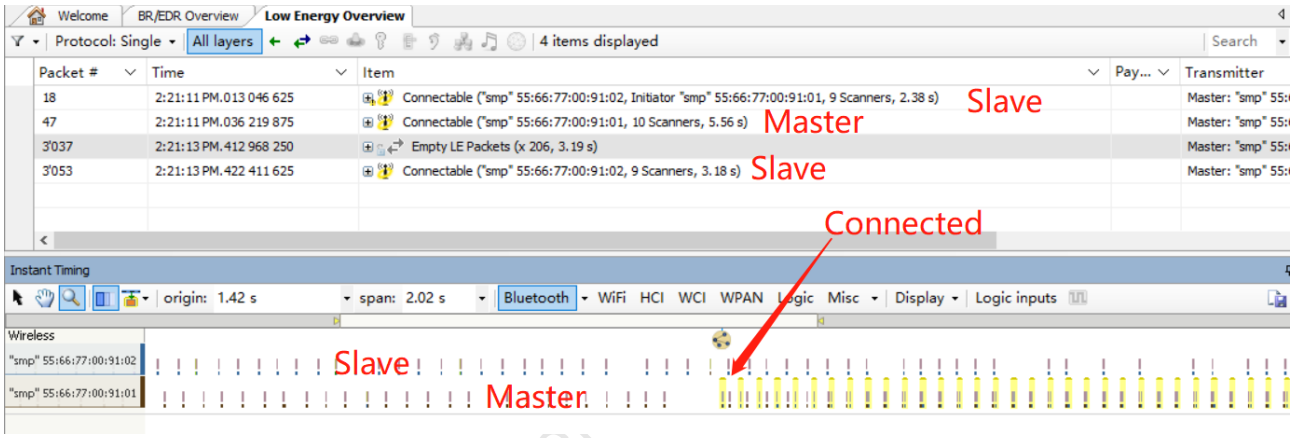


Figure 10.20: The packet capture when appMaxSlaveNum>1 connection successful

10.6.2 Enable Slave SMP only

Demo for Slave only when SMP is enabled. The demo needs to be defined in feature_smp/app_config.h:

```
#define SMP_TEST_MODE              SMP_TEST_ONLY_SLAVE
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards. Press SW4 (Start Connection) on one of the development boards (as Master), the light effect is the same as if both sides did not enable SMP. However, by capturing the packet, we can see that after establishing the connection, the Slave sends a Security Request, but since the Master does not support SMP, the Mater does not respond, as follows:

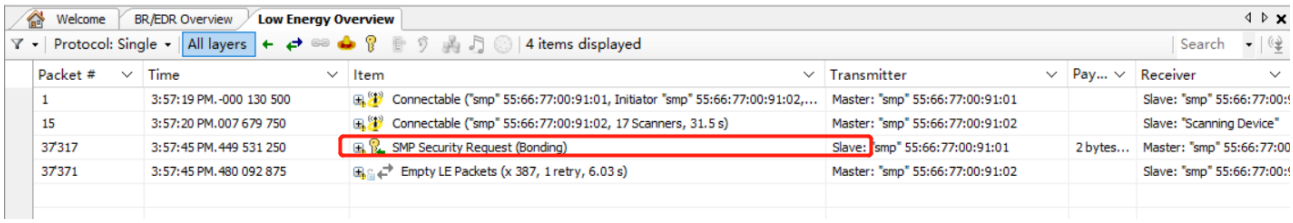


Figure 10.21: Packet capture when Slave SMP only

Description:

If the user does not want the Slave to send a Security Request after the connection is established, the following should be added to the initialization code (comment out by default).

```
blc_smp_configSecurityRequestSending( SecReq_NOT_SEND,  SecReq_NOT_SEND, 0);
```

This API is only for Slave (only Slave sends Security Request), so there is no corresponding API with _master/_slave suffix.

10.6.3 Enable Master SMP only

Demonstration only when Master board enables the SMP feature. This demo is defined in feature_smp/app_config.h:

```
#define SMP_TEST_MODE           SMP_TEST_ONLY_MASTER
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Press SW4 (Start Connection) on one of the boards (as Master), the light effect is the same as if both sides did not enable SMP. However, we can see through the packet capture that the Master sends Pairing Request after establishing the connection, but since the Slave does not support SMP, the Slave responds Pairing Failed, and the Error Code corresponds to the definition in the SDK as PAIRING_FAIL_REASON_PAIRING_NOT_ SUPPORTED. The packet capture is as follows.

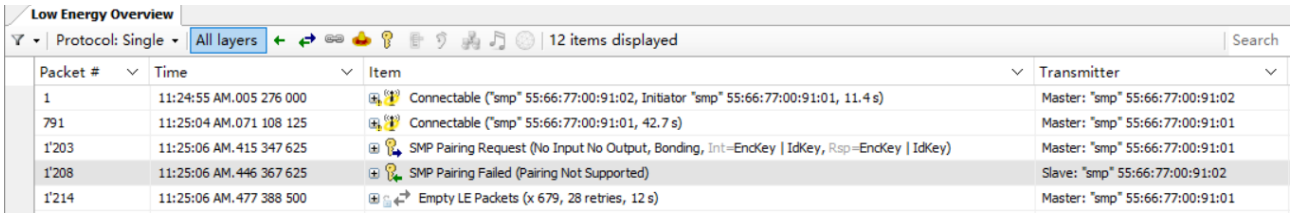


Figure 10.22: Packet capture when Master SMP only

10.6.4 Legacy Just Works

Demonstration of Master and Slave enabling SMP functionality, paired as Legacy Just Works. This demo is defined in feature_smp/app_config.h:

```
#define SMP_TEST_MODE              SMP_TEST_LEGACY_JUST_WORKS
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Press SW4 (start connection) on one of the development boards (as Master), after successful connection, you can see the Master lights up green, white and blue, and the Slave lights up red, white and blue, which means they are successful in Pair, and the packet capture is as follows.

P...	Time	Item	Transmitter	Payl...
4	3:04:02 PM.009 471 875	Connectable ("smp" 55:66:77:00:91:01, 5 Scanners, 48 s)	Master: "smp" 55:66:77:00:91:01	
854	3:04:06 PM.082 387 125	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 4 Scanners, 2.38 s)	Master: "smp" 55:66:77:00:91:02	
1'606	3:04:08 PM.478 718 875	SMP Pairing Request (No Input No Output, Bonding, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	7 bytes ...
1'607	3:04:08 PM.509 826 250	SMP Security Request (Bonding)	Slave: "smp" 55:66:77:00:91:02	2 bytes ...
1'609	3:04:08 PM.510 335 875	SMP Pairing Response (No Input No Output, Bonding, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Slave: "smp" 55:66:77:00:91:02	7 bytes ...
1'619	3:04:08 PM.541 219 250	SMP Pairing Confirm (Ca=E8A0D0A3:13196DC1:251D7A6D:9C413836)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'632	3:04:08 PM.572 238 875	SMP Pairing Confirm (Cb=37CB4C8C:722C5B6A:C7833FE0:731711D2)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'641	3:04:08 PM.603 718 875	SMP Pairing Random (Na=47132D6A:8F9B4F69:E0BCB4D9:97A9515C)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'651	3:04:08 PM.634 739 125	SMP Pairing Random (Nb=CE2B5F91:EAC9D347:54F2040F:45EEDDCA)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'657	3:04:08 PM.666 219 250	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x4FC0E6552EFC57...	Master: "smp" 55:66:77:00:91:01	23 byte...
1'664	3:04:08 PM.697 239 250	LLCP Encryption Response (SKDs=0xEACAEF93276E5AE6, IVs=0x80DEC572)	Slave: "smp" 55:66:77:00:91:02	13 byte...
1'690	3:04:08 PM.790 989 125	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	1 byte (...
1'699	3:04:08 PM.822 469 375	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	1 byte (...
1'713	3:04:08 PM.853 489 250	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	1 byte (...
1'728	3:04:08 PM.915 989 375	SMP Encryption Information (LTK=9B7E0AC4:BF9C8612:01A7515A:10B8889F)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'742	3:04:08 PM.978 489 750	SMP Master Identification (EDIV=0x63B4, Rand=0x22A540EEEECF6798)	Slave: "smp" 55:66:77:00:91:02	11 byte...
1'758	3:04:09 PM.040 989 625	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'777	3:04:09 PM.103 489 625	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	8 bytes ...
1'788	3:04:09 PM.134 969 875	SMP Encryption Information (LTK=1246783F:DACE1A3C:B5E9E18C:C2FC0409)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'804	3:04:09 PM.166 219 875	SMP Master Identification (EDIV=0xD5E2, Rand=0xFDE22565FD06C8BC)	Master: "smp" 55:66:77:00:91:01	11 byte...
1'813	3:04:09 PM.197 469 875	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'826	3:04:09 PM.228 720 000	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	8 bytes ...
1'827	3:04:09 PM.229 077 375	Empty LE Packets (x 854, 136 retries, 15.5 s)	Slave: "smp" 55:66:77:00:91:02	

Figure 10.23: Packet capture of Legacy Just Works

At this time, press the SW1 button on the Master or Slave board to re-power the board, because the Master will automatically connect back when it scans the Slave device that has been paired (refer to the master_auto_connect variable in the demo code), you can see that the light turns on again immediately after it goes off, but this time the blue light does not turn on, because the reconnection does not go through the Pair process, but directly through the LTK encryption process.

Description:

Sometimes, we need to keep running the pairing process and do not want the Master and Slave to skip the Pair process after re-powering, we just need the Bonding Flag of either the Master or Slave to be set to 0. Two methods are given here.

- Method 1: Initially configure the Security Parameters with the Bonding Flag set to 0. Take the Slave as an example (the default is commented out).

```
blc_smp_setSecurityParameters_slave(Non_Bondable_Mode, 0, LE_Legacy_Pairing,    0, 0,
↪  IO_CAPABILITY_NO_INPUT_NO_OUTPUT);
```

The bonding_mode on the Slave side is set to Non-Bondable mode, and the other settings are set to the Just Works default configuration, as is the Master side. If the API is not called, the initial values of the SecurityParameters are set in blc_gap_init() for Slave and Master by default.

```
mode_level = Unauthenticated_Pairing_with_Encryption;
bond_mode = Bondable_Mode;
MITM_en = 0;
method = LE_Legacy_Pairing;
OOB_en = 0;
keyPress_en = 0;
ioCapablility = IO_CAPABILITY_NO_INPUT_NO_OUTPUT;
ecdh_debug_mode = non_debug_mode;
passKeyEntryDftTK = 0;
```

After calling this API, when pairing, you will see Bonding Flags=0 in the Pairing Response from the Slave, so that neither the Master nor the Slave will be bonding, which is no pairing information will be stored in Flash. When the Master or Slave device reboots, they are still "brand new" and do not "know" each other.

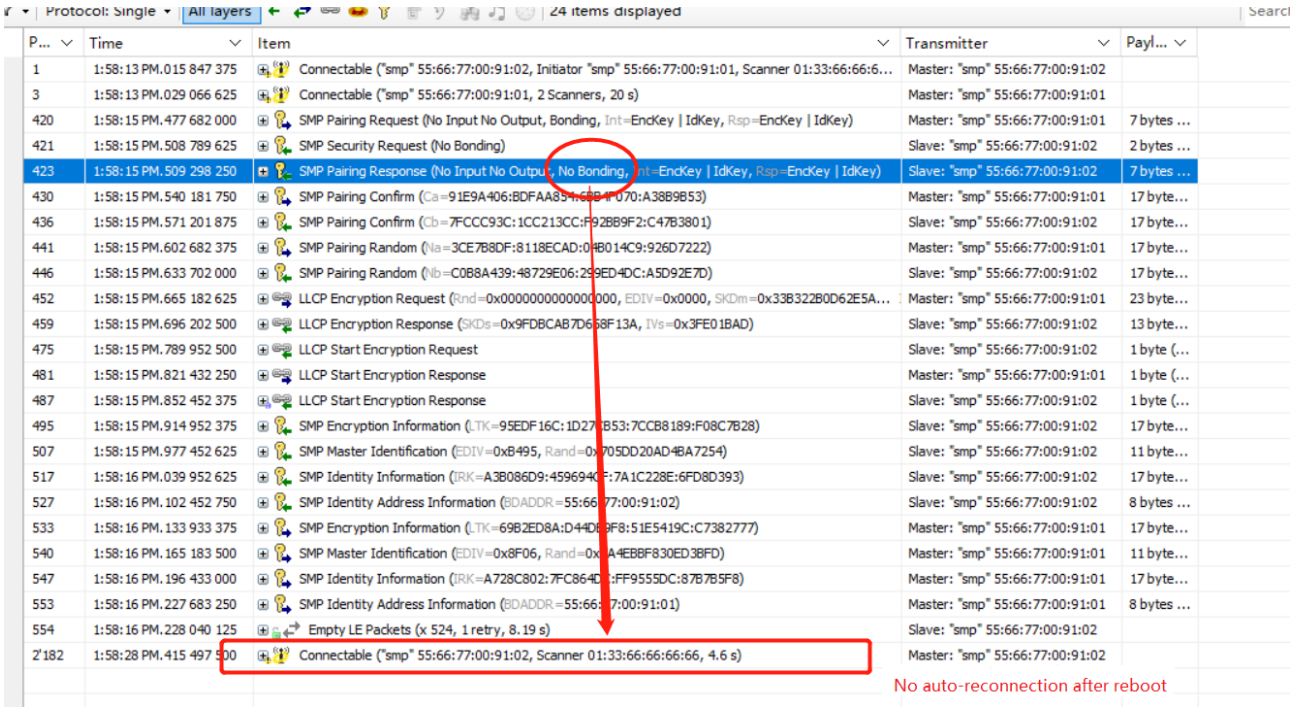
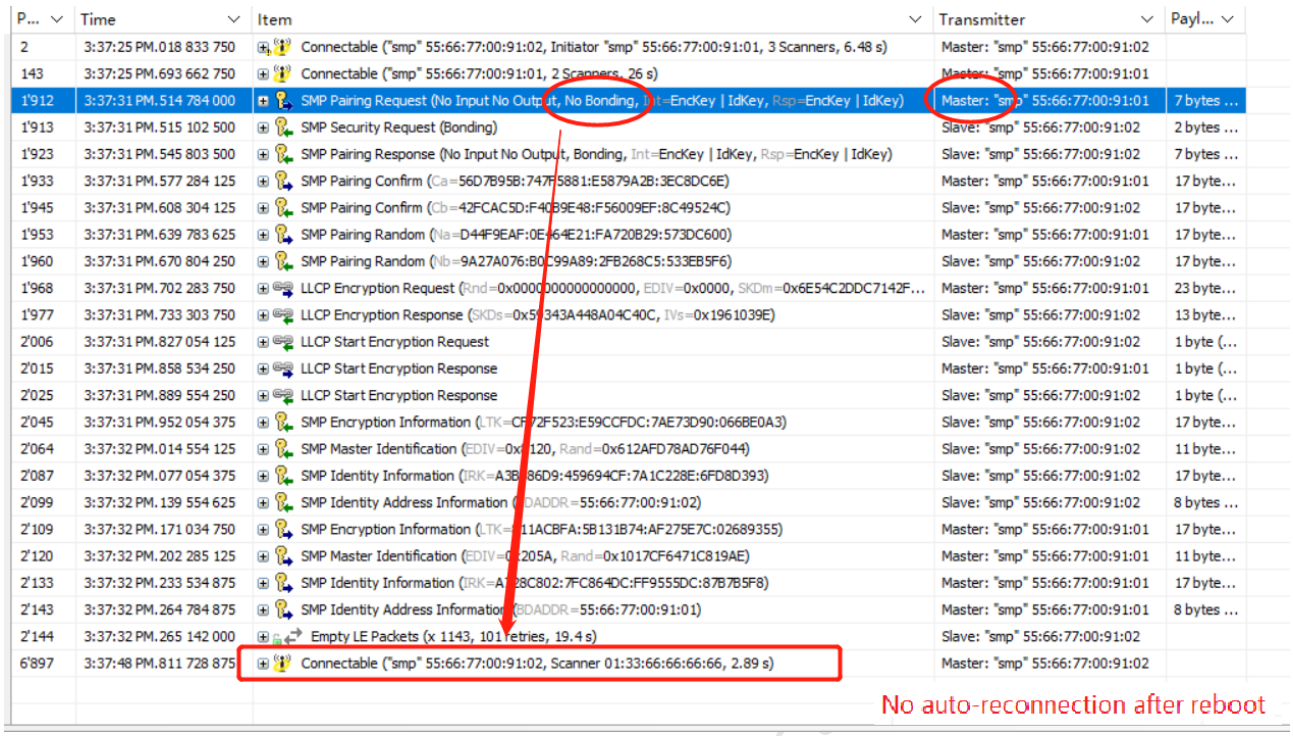


Figure 10.24: Packet capture of Legacy Just Works Slave NoBonding

- Method 2: Initially configure the Security Parameters and configure Bonding Mode separately, taking Master as an example:

```
blc_smp_setBondingMode_master(Non_Bondable_Mode);
```

Here, the bonding_mode is set to Non-Bondable mode on the Master side, and the same on the Slave side. The packet capture shows the same effect as method 1 above.



P...	Time	Item	Transmitter	Payl...
2	3:37:25 PM.018 833 750	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 3 Scanners, 6.48 s)	Master: "smp" 55:66:77:00:91:02	
143	3:37:25 PM.693 662 750	Connectable ("smp" 55:66:77:00:91:01, 2 Scanners, 26 s)	Master: "smp" 55:66:77:00:91:01	
1'912	3:37:31 PM.514 784 000	SMP Pairing Request (No Input No Output, No Bonding, Ini=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	7 bytes ...
1'913	3:37:31 PM.515 102 500	SMP Security Request (Bonding)	Slave: "smp" 55:66:77:00:91:02	2 bytes ...
1'923	3:37:31 PM.545 803 500	SMP Pairing Response (No Input No Output, Bonding, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Slave: "smp" 55:66:77:00:91:02	7 bytes ...
1'933	3:37:31 PM.577 284 125	SMP Pairing Confirm (Ca=56D7B95B:747F5881:E5879A2B:3EC8DC6E)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'945	3:37:31 PM.608 304 125	SMP Pairing Confirm (Cb=42FCAC5D:F40B9E48:F56009EF:8C49524C)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'953	3:37:31 PM.639 783 625	SMP Pairing Random (Na=D44F9EAF:0E464E21:FA720B29:573DC600)	Master: "smp" 55:66:77:00:91:01	17 byte...
1'960	3:37:31 PM.670 804 250	SMP Pairing Random (Nb=9A27A076:B0C99A89:2FB268C5:533EB5F6)	Slave: "smp" 55:66:77:00:91:02	17 byte...
1'968	3:37:31 PM.702 283 750	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x6E54C2DDC7142F...	Master: "smp" 55:66:77:00:91:01	23 byte...
1'977	3:37:31 PM.733 303 750	LLCP Encryption Response (SKDs=0x5[illegible]3A448A04C40C, IVs=0x1961039E)	Slave: "smp" 55:66:77:00:91:02	13 byte...
2'006	3:37:31 PM.827 054 125	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	1 byte (...
2'015	3:37:31 PM.858 534 250	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	1 byte (...
2'025	3:37:31 PM.889 554 250	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	1 byte (...
2'045	3:37:31 PM.952 054 375	SMP Encryption Information (LTK=CF[illegible]2F523:E59CCFDC:7AE73D90:066BE0A3)	Slave: "smp" 55:66:77:00:91:02	17 byte...
2'064	3:37:32 PM.014 554 125	SMP Master Identification (EDIV=0x[illegible]120, Rand=0x612AFD78AD76F044)	Slave: "smp" 55:66:77:00:91:02	11 byte...
2'087	3:37:32 PM.077 054 375	SMP Identity Information (IRK=A3B[illegible]86D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	17 byte...
2'099	3:37:32 PM.139 554 625	SMP Identity Address Information ([illegible]DADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	8 bytes ...
2'109	3:37:32 PM.171 034 750	SMP Encryption Information (LTK=[illegible]11ACBFA:5B131B74:AF275E7C:02689355)	Master: "smp" 55:66:77:00:91:01	17 byte...
2'120	3:37:32 PM.202 285 125	SMP Master Identification (EDIV=0[illegible]205A, Rand=0x1017CF6471C819AE)	Master: "smp" 55:66:77:00:91:01	11 byte...
2'133	3:37:32 PM.233 534 875	SMP Identity Information (IRK=A[illegible]28C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	17 byte...
2'143	3:37:32 PM.264 784 875	SMP Identity Address Information ([illegible]DADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	8 bytes ...
2'144	3:37:32 PM.265 142 000	Empty LE Packets (x 1143, 101 retries, 19.4 s)	Slave: "smp" 55:66:77:00:91:02	
6'897	3:37:48 PM.811 728 875	Connectable ("smp" 55:66:77:00:91:02, Scanner 01:33:66:66:66:66, 2.89 s)	Master: "smp" 55:66:77:00:91:02	

No auto-reconnection after reboot

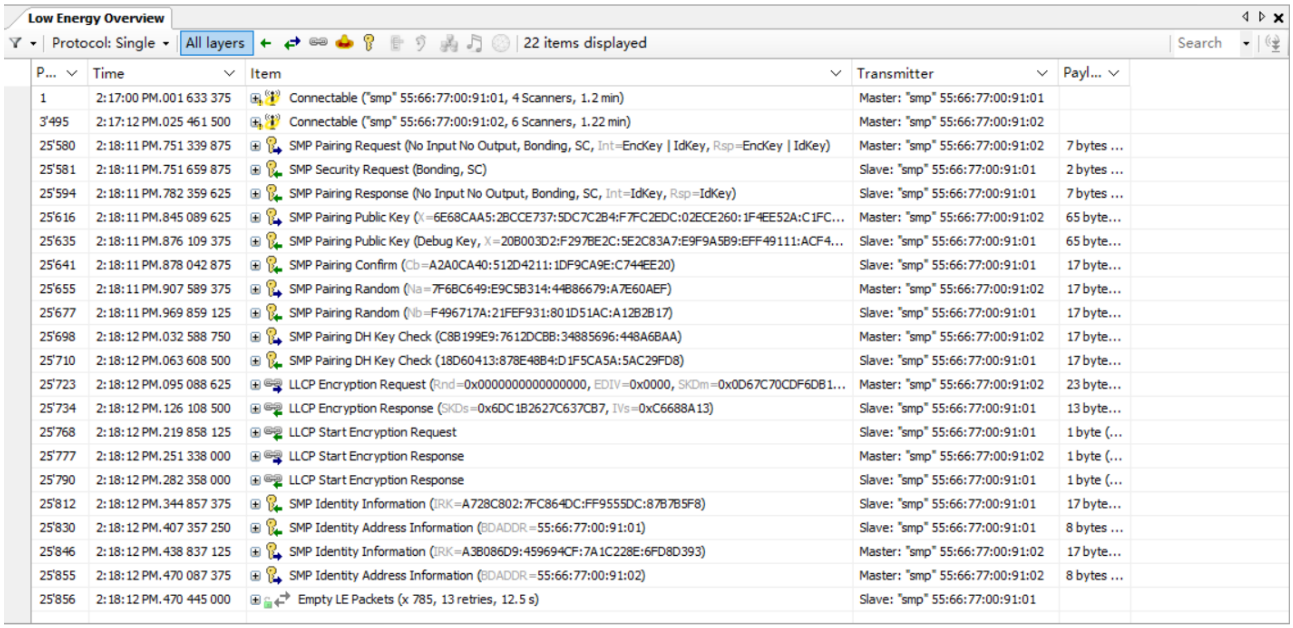
Figure 10.25: Packet capture of Legacy Just Works Master NoBonding

10.6.5 Secure Connections Just Works

Demonstration of Master and Slave enabling SMP functionality to pair as Secure Connections Just Works. This demo is defined in feature_smp/app_config.h:

```
#define SMP_TEST_MODE              SMP_TEST_SC_JUST_WORKS
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Press SW4 (start connection) on one of the development boards (as Master), after successful connection, you can see the Master lights up green, white and blue, and the Slave lights up red, white and blue, which means they are successful in Pair, and the packet capture is as follows.



P...	Time	Item	Transmitter	Payl...
1	2:17:00 PM.001 633 375	Connectable ("smp" 55:66:77:00:91:01, 4 Scanners, 1.2 min)	Master: "smp" 55:66:77:00:91:01	
3'495	2:17:12 PM.025 461 500	Connectable ("smp" 55:66:77:00:91:02, 6 Scanners, 1.22 min)	Master: "smp" 55:66:77:00:91:02	
25'580	2:18:11 PM.751 339 875	SMP Pairing Request (No Input No Output, Bonding, SC, Ini=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:02	7 bytes ...
25'581	2:18:11 PM.751 659 875	SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
25'594	2:18:11 PM.782 359 625	SMP Pairing Response (No Input No Output, Bonding, SC, Ini=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01	7 bytes ...
25'616	2:18:11 PM.845 089 625	SMP Pairing Public Key (X=6E68CAA5:2BCCE737:5DC7C2B4:F7FC2EDC:02ECE260:1F4EE52A:C1FC...	Master: "smp" 55:66:77:00:91:02	65 byte...
25'635	2:18:11 PM.876 109 375	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4...	Slave: "smp" 55:66:77:00:91:01	65 byte...
25'641	2:18:11 PM.878 042 875	SMP Pairing Confirm (Cb=A2A0CA40:512D4211:1DF9CA9E:C744EE20)	Slave: "smp" 55:66:77:00:91:01	17 byte...
25'655	2:18:11 PM.907 589 375	SMP Pairing Random (Na=7F6BC649:E9C5B314:44B86679:A7E60AEF)	Master: "smp" 55:66:77:00:91:02	17 byte...
25'677	2:18:11 PM.969 859 125	SMP Pairing Random (Nb=F496717A:21FEF931:801D51AC:A12B2B17)	Slave: "smp" 55:66:77:00:91:01	17 byte...
25'698	2:18:12 PM.032 588 750	SMP Pairing DH Key Check (C8B199E9:7612DCBB:34885696:448A6BAA)	Master: "smp" 55:66:77:00:91:02	17 byte...
25'710	2:18:12 PM.063 608 500	SMP Pairing DH Key Check (18D60413:878E48B4:D1F5CA5A:5AC29FD8)	Slave: "smp" 55:66:77:00:91:01	17 byte...
25'723	2:18:12 PM.095 088 625	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x0D67C70CDF6DB1...	Master: "smp" 55:66:77:00:91:02	23 byte...
25'734	2:18:12 PM.126 108 500	LLCP Encryption Response (SKDs=0x6DC1B2627C637CB7, IVs=0xC6688A13)	Slave: "smp" 55:66:77:00:91:01	13 byte...
25'768	2:18:12 PM.219 858 125	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:01	1 byte (...
25'777	2:18:12 PM.251 338 000	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:02	1 byte (...
25'790	2:18:12 PM.282 358 000	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:01	1 byte (...
25'812	2:18:12 PM.344 857 375	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Slave: "smp" 55:66:77:00:91:01	17 byte...
25'830	2:18:12 PM.407 357 250	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Slave: "smp" 55:66:77:00:91:01	8 bytes ...
25'846	2:18:12 PM.438 837 125	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Master: "smp" 55:66:77:00:91:02	17 byte...
25'855	2:18:12 PM.470 087 375	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Master: "smp" 55:66:77:00:91:02	8 bytes ...
25'856	2:18:12 PM.470 445 000	Empty LE Packets (x 785, 13 retries, 12.5 s)	Slave: "smp" 55:66:77:00:91:01	

Figure 10.26: Packet capture of SC Just Works

Refer to the SMP section for Debug Mode, which can be configured by the user as follows, depending on the requirements.

Table 10.1: Debug Mode configuration options

Master	Slave	Description
debug mode disabled	debug mode enabled	Demo code default configuration
debug mode enabled	debug mode disabled	The effect is similar to the demo code
debug mode disabled	debug mode disabled	The encrypted packets cannot be parsed by packet capture tools

Note:

> The combination of Slave debug mode enable + Master debug mode enable is not allowed, the SMP Pairing Failed event will occur.

P...	Time	Item	Transmitter	Payl...
5	2:44:54 PM.031 107 250	Connectable ("smp" 55:66:77:00:91:01, Initiator "smp" 55:66:77:00:91:02, 7 Scanners, 17.1 s)	Master: "smp" 55:66:77:00:91:01	
908	2:44:57 PM.293 257 000	Connectable ("smp" 55:66:77:00:91:02, 5 Scanners, 21.4 s)	Master: "smp" 55:66:77:00:91:02	
5'933	2:45:11 PM.136 815 750	SMP Pairing Request (No Input No Output, Bonding, SC, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:02	7 bytes ...
5'934	2:45:11 PM.137 133 250	SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
5'943	2:45:11 PM.167 835 500	SMP Pairing Response (No Input No Output, Bonding, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01	7 bytes ...
5'957	2:45:11 PM.199 315 625	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDDB:...	Master: "smp" 55:66:77:00:91:02	65 byte...
5'977	2:45:11 PM.230 335 375	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDDB:...	Slave: "smp" 55:66:77:00:91:01	65 byte...
5'983	2:45:11 PM.232 264 500	SMP Pairing Confirm (Cb=33A2614E:2B77C2B7:C295C14D:B191735E)	Slave: "smp" 55:66:77:00:91:01	17 byte...
5'993	2:45:11 PM.261 815 500	SMP Pairing Failed (Invalid Parameters)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
5'995	2:45:11 PM.262 323 625	SMP Pairing Failed (Unspecified Reason)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
5'996	2:45:11 PM.262 600 625	Empty LE Packets (x 472, 1 retry, 7.41 s)	Slave: "smp" 55:66:77:00:91:01	

Figure 10.27: Packet capture when both Debug Modes Pair Failed

Important note!

> Prior to B91 Multi-Connection SDK V4.0.1.0 and versions, the following API calls are required (commented out by default in the code).

```
void    blc_smp_setPairingMethods{_master/_slave}(pairing_methods_t  method);
```

Set up Secure Connection to avoid the situation where the Public Key is all zeros.

P...	Time	Item	Transmitter	Payl...
4	3:13:35 PM.024 392 250	Connectable ("smp" 55:66:77:00:91:01, Initiator "smp" 55:66:77:00:91:02, 3 Scanners, 26.1 s)	Master: "smp" 55:66:77:00:91:01	
7	3:13:35 PM.027 956 875	Connectable ("smp" 55:66:77:00:91:02, 5 Scanners, 29.1 s)	Master: "smp" 55:66:77:00:91:02	
8'265	3:14:01 PM.138 787 500	SMP Pairing Request (No Input No Output, Bonding, SC, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:02	7 bytes ...
8'266	3:14:01 PM.139 106 250	SMP Security Request (Bonding, SC)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
8'277	3:14:01 PM.169 807 375	SMP Pairing Response (No Input No Output, Bonding, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01	7 bytes ...
8'291	3:14:01 PM.201 287 250	SMP Pairing Public Key (X=00000000:00000000:00000000:00000000:00000000:00000000:00000000:00...	Master: "smp" 55:66:77:00:91:02	65 byte...
8'307	3:14:01 PM.232 306 875	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDDB:...	Slave: "smp" 55:66:77:00:91:01	65 byte...
8'313	3:14:01 PM.234 242 000	SMP Pairing Confirm (Cb=83826038:524EA192:FD6C7DAB:FD9DDB15)	Slave: "smp" 55:66:77:00:91:01	17 byte...
8'326	3:14:01 PM.263 787 000	SMP Pairing Random (Na=7372B849:A0423073:D1A2543F:D1FF1A06)	Master: "smp" 55:66:77:00:91:02	17 byte...
8'338	3:14:01 PM.294 806 750	SMP Pairing Failed (Invalid Parameters)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
8'342	3:14:01 PM.295 314 250	SMP Pairing Failed (Unspecified Reason)	Slave: "smp" 55:66:77:00:91:01	2 bytes ...
8'345	3:14:01 PM.295 821 875	SMP Pairing Random (Nb=78CA1B3B:47C7BC7F:51029039:E67AED08)	Slave: "smp" 55:66:77:00:91:01	17 byte...
8'354	3:14:01 PM.326 286 875	SMP Pairing Failed (Invalid Parameters)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
8'357	3:14:01 PM.326 794 375	SMP Pairing Failed (Confirm Value Failed)	Master: "smp" 55:66:77:00:91:02	2 bytes ...
8'359	3:14:01 PM.327 071 375	Empty LE Packets (x 177, 2.78 s)	Slave: "smp" 55:66:77:00:91:01	

Figure 10.28: Packet capture when PublicKey=0

10.6.6 Legacy Passkey Entry MDSI

Demonstration of Master and Slave enabling SMP functionality, paired with Legacy Passkey Entry, via the MDSI method. MDSI stands for Master(Initiator) Displays Slave(Responder) Inputs, and the Inputs demonstrated here are Keyboard Input, which is defined in feature_smp/app_config.h:

```
#define SMP_TEST_MODE       SMP_TEST_LEGACY_PASSKEY_ENTRY_MDSI
```

The demo code provides two methods for Responder to enter Passkey, here is a demonstration of the direct way to set the default Passkey to 123456, refer to the definition:

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_BY_DEFAULT
```

Compile B91_feature，burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards. Press SW4 (start connection) of one of the development board (as Master), after successful connection, you can see the Master lights up green and the Slave lights up red, after a few seconds, the white and blue lights of both of them light up one after another, representing the successful Pair of both of them, grab the packet as follows (the boxed part is waiting for the Slave to input Passkey)

P...	Time	Item	Transmitter	Receiver
1	7:38:56 AM.017 320 250	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 4.99 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
22	7:38:56 AM.249 404 500	Connectable ("smp" 55:66:77:00:91:01, 20.4 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
829	7:39:01 AM.031 142 625	SMP Pairing Request (Display Only, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
830	7:39:01 AM.031 461 500	SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
834	7:39:01 AM.062 164 125	SMP Pairing Response (Keyboard Only, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
839	7:39:01 AM.093 642 125	SMP Pairing Confirm (Ca=0BA90EF6:59A1A768:F639C5D7:022A7D7C)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
840	7:39:01 AM.094 040 000	Empty LE Packets (x 412, 6.44 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'897	7:39:07 AM.530 919 375	SMP Pairing Confirm (Cb=62554804:0B1CC38D:519B6ED2:D46C7112)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'903	7:39:07 AM.562 397 750	SMP Pairing Random (Na=40AF1741:FD2B700C:97FC66DB:085D8D5B)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'909	7:39:07 AM.593 419 875	SMP Pairing Random (Nb=F330447D:EEED74E5:C434F3A6:F395063E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'916	7:39:07 AM.624 897 625	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x68EB75150229EC31, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'923	7:39:07 AM.655 919 625	LLCP Encryption Response (SKDs=0x1B506FB7DD5D292E, IVs=0x90B6AF02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'938	7:39:07 AM.749 669 875	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'944	7:39:07 AM.781 148 000	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'950	7:39:07 AM.812 169 750	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'960	7:39:07 AM.874 670 000	SMP Encryption Information (LTK=A6651128:BBB821B0:9161A6F3:A6C0536B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'972	7:39:07 AM.937 170 000	SMP Master Identification (EDIV=0xF673, Rand=0x15DF9B2BD156054B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'983	7:39:07 AM.999 670 000	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'992	7:39:08 AM.062 170 000	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'999	7:39:08 AM.093 648 125	SMP Encryption Information (LTK=15FA4214:A87E2559:C2A9338E:5D08D80E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'007	7:39:08 AM.124 898 500	SMP Master Identification (EDIV=0x246F, Rand=0x0ADF331C9DD870E0)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'014	7:39:08 AM.156 148 500	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'021	7:39:08 AM.187 398 250	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'022	7:39:08 AM.187 755 625	Empty LE Packets (x 541, 8.44 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 10.29: Packet capture of Legacy Passkey Entry MDSI

The code uses blc_smp_setDefaultPinCode() to set the default Passkey of the Display device during initialization, here it is set to 123456. If this statement is not called to set the default Passkey, the system will generate a random 6-digit Passkey in decimal 000000~999999.

Thereafter, the Input device polls blc_smp_isWaitingToSetTK() in the main_loop to get whether it is currently waiting for the input Passkey, and when it gets the demand, it sets the Passkey via blc_smp_setTK_by_PasskeyEntry(), which should be equal to the default Passkey set by the The default Passkey set by the Display device. Note that the Passkey set by calling blc_smp_setTK_by_PasskeyEntry() before the demand for blc_smp_isWaitingToSetTK() is obtained is invalid.

In addition, calling blc_smp_setTK_by_PasskeyEntry() in app_host_event_callback() when the GAP_EVT_SMP_TK_REQUEST_PASSKEY GAP event is fetched will also successfully set the Passkey of the Input device.

10.6.7 Legacy Passkey Entry MISD

A demonstration of the Master and Slave enabling SMP functionality, paired with Legacy Passkey Entry, via the MISD method. MISD is similar to the previous demo MDSI, except that the roles of display and input devices have been swapped, MISD stands for Master(Initiator) Inputs Slave(Responder) Displays. To use this feature, define it in feature_smp/app_config.h:

```
#define SMP_TEST_MODE        SMP_TEST_LEGACY_PASSKEY_ENTRY_MISD
```

The demo code provides two methods for the Initiator to input Passkey, here is a demonstration of the method to input Passkey via UART, which is more relevant to the actual usage requirements, refer to the definition:

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_MANUALLY
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Connect the UART port PD2(Tx) and PD3(Rx) of one of the development boards (as Master) to the PC serial port and configure the baud rate 115200. Press SW4 (start connection) on the Master board, after successful connection, you can see the green light on the Master and red light on the Slave, at this time, the log output of the Slave Display Pincode, fill the output number into the Master's UART serial port, the SDK default timeout is 30s, Passkey must be input successfully within the time, you can see the SMP process continues, the white and blue lights are on one after another, representing the successful Pair of the two. The packet capture is as follows (the boxed part is waiting for the Master to enter Passkey).

P...	Time	Item	Transmitter	Receiver
1	6:08:39 PM.012 992 500	Connectable ("smp" 55:66:77:00:91:01, Scanner 70:DC:AB:F7:A7:FC (Resolvable), 28.8 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
4	6:08:39 PM.018 719 375	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, Scanner 70:DC:AB:F7:A7:F...	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
750	6:08:43 PM.297 228 625	SMP Pairing Request (Keyboard Only, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
751	6:08:43 PM.297 545 250	SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
756	6:08:43 PM.328 250 375	SMP Pairing Response (Keyboard Display, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
761	6:08:43 PM.359 271 375	Empty LE Packets (x 924, 4 retries, 14.4 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'177	6:08:57 PM.797 238 375	SMP Pairing Confirm (Ca=C626E58E:2642B915:83B24D2C:0CD3EF7A)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'183	6:08:57 PM.828 260 375	SMP Pairing Confirm (Cb=17978F39:639F692C:532DB6B7:A0AB2DA6)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'189	6:08:57 PM.859 739 000	SMP Pairing Random (Na=CC95CAFF:1EE76639:5DEF99D5:8B41F3D5)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'195	6:08:57 PM.890 760 625	SMP Pairing Random (Nb=5178A0CC:F8A85D18:D91318D1:FA21786B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'201	6:08:57 PM.922 238 625	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x7EA7876E7735A3C3, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'207	6:08:57 PM.953 260 875	LLCP Encryption Response (SKDs=0xC84BB63FF3758AD2, IVs=0x2F59F10D)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'224	6:08:58 PM.047 011 250	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'230	6:08:58 PM.078 489 125	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'236	6:08:58 PM.109 511 125	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'246	6:08:58 PM.172 010 875	SMP Encryption Information (LTK=042DF599:ADFD084D:8C464D84:AF742D3E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'257	6:08:58 PM.234 511 000	SMP Master Identification (EDIV=0x7B41, Rand=0xFB223EEA324C7128)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'266	6:08:58 PM.297 011 250	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'276	6:08:58 PM.359 511 250	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
3'282	6:08:58 PM.390 989 250	SMP Encryption Information (LTK=99C09FAA:4BB2336C:08BACC80:DE14A680)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'290	6:08:58 PM.422 239 000	SMP Master Identification (EDIV=0xCA34, Rand=0x2C2907786762F5D4)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'298	6:08:58 PM.453 489 500	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'305	6:08:58 PM.484 739 375	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
3'306	6:08:58 PM.485 096 500	Empty LE Packets (x 598, 9.34 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 10.30: Packet capture of Legacy Passkey Entry MISD

10.6.8 Legacy Passkey Entry Both Input

Demonstration of Master and Slave enabling SMP functionality, paired with Legacy Passkey Entry, via Both Input method. Both Input is implemented by the user entering the same Pincode on both sides. The use of this feature is defined in feature_smp/app_config.h:

```
#define SMP_TEST_MODE        SMP_TEST_LEGACY_PASSKEY_ENTRY_BOTH_INPUT
```

The demo code provides two methods of entering Passkey, here is a demonstration of setting the default Passkey to 123456 directly, refer to the definition below:

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_BY_DEFAULT
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Press SW4 (start connection) of one of the development board (as Master), after successful connection, you can see the Master lights up green and Slave lights up red, after a few seconds, the white and blue lights of both of them light up one after another, representing the successful Pair of them, the packet capture is as follows (the boxed part is waiting for both sides to input Passkey)

P...	Time	Item	Transmitter	Receiver
1	12:37:08 PM.001 769 750	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 1.65 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
4	12:37:08 PM.005 029 125	Connectable ("smp" 55:66:77:00:91:01, 15.6 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
318	12:37:09 PM.668 638 125	SMP Pairing Request (Keyboard Only, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
319	12:37:09 PM.668 957 750	SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
327	12:37:09 PM.699 660 125	SMP Pairing Response (Keyboard Only, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
331	12:37:09 PM.730 681 000	Empty LE Packets (x 364, 5.66 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'302	12:37:15 PM.387 391 875	SMP Pairing Confirm (Ca=F97DAD23:70D9D0B0:6FBEE76B:4ABF50FB)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'303	12:37:15 PM.387 788 750	Empty LE Packets (x 252, 3.94 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'979	12:37:19 PM.324 666 000	SMP Pairing Confirm (Cb=BE2A022C:C137F8E1:F2716704:035A8220)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'986	12:37:19 PM.356 143 750	SMP Pairing Random (Na=575D9442:A21676C7:D893D0F7:A5BB4616)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'993	12:37:19 PM.387 165 625	SMP Pairing Random (Nb=5647EFBC:69CFC73A:D10786D1:18DAC55B)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'998	12:37:19 PM.418 644 000	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0xA324E72A74871E57, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'004	12:37:19 PM.449 665 750	LLCP Encryption Response (SKDs=0xCD89EBF03B33A59B, IVs=0x3C2FAC4F)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'019	12:37:19 PM.543 415 625	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'029	12:37:19 PM.574 893 625	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'036	12:37:19 PM.605 915 875	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'045	12:37:19 PM.668 415 750	SMP Encryption Information (LTK=0312BAE9:3C9A926F:8452D384:4D8F900E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'054	12:37:19 PM.730 915 875	SMP Master Identification (EDIV=0x8F2A, Rand=0x906F3EC0AE5E4609)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'065	12:37:19 PM.793 415 750	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'079	12:37:19 PM.855 915 875	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'084	12:37:19 PM.887 394 125	SMP Encryption Information (LTK=0208C117:F7432392:8DC685A2:F0EE1343)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'090	12:37:19 PM.918 644 000	SMP Master Identification (EDIV=0x1051, Rand=0x12B943A4A065C5A9)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'097	12:37:19 PM.949 894 250	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'104	12:37:19 PM.981 144 125	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'105	12:37:19 PM.981 501 125	Empty LE Packets (x 235, 3.66 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 10.31: Packet capture of Legacy Passkey Entry Both Input

10.6.9 Secure Connections Passkey Entry

Demonstration of Master and Slave enabling the SMP function to pair in Secure Connections Passkey Entry. Corresponding to Legacy, Secure Connections has three implementations of Passkey Entry matching: MDSI, MISD, and Both Input. For cross demonstration, the MISD method of Passkey Entry under Secure Connections is given in the demo code as By_Default method to enter Passkey, and the MDSI and BothInput methods are given as Manually method to enter Passkey.

- For MDSI, you need define it in feature_smp/app_config.h:

```
#define SMP_TEST_MODE        SMP_TEST_SC_PASSKEY_ENTRY_MDSI
```

The Passkey input refers to below definition:

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_MANUALLY
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Connect the UART ports PD2(Tx) and PD3(Rx) of one of the development boards (as Slave) to the PC serial port and configure the baud rate 115200. Press SW4 (start connection) on the other board (Master), after successful connection, you can see the green light on Master and red light on Slave. At this time, the Master's Log output Display Pincode, the output number will be filled into the Master's UART serial port within 30s, you can see the SMP process continues, white and blue lights up one after another, representing the successful Pair of the two, the packet capture is as follows (the highlighted part is waiting for the Slave input Passkey).

P...	Time	Item	Transmitter	Receiver
1	6:25:23 PM.000 271 000	Connectable ("smp" 55:66:77:00:91:02, Scanner 72:C0:97:F5:A6:85 (Resolvable), 1.43 min)	Master: "smp" 55:66:77:00:91:02	Slave: "Scanning Device"
5	6:25:23 PM.025 245 875	Connectable ("smp" 55:66:77:00:91:01, Initiator "smp" 55:66:77:00:91:02, Scanner 72:C0:97:F5:A6:8...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'580	6:25:31 PM.296 809 000	SMP Pairing Request (Display Only, Bonding, MITM, SC, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
1'581	6:25:31 PM.297 129 000	SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
1'591	6:25:31 PM.359 080 375	SMP Pairing Response (Keyboard Only, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
1'597	6:25:31 PM.390 558 250	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDD...	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
1'607	6:25:31 PM.421 580 125	SMP Pairing Public Key (X=8BA53ADB:1E3DC32D:97582578:2E9343AA:99454744:9718B3AB:6D5E7A7C...	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
1'616	6:25:31 PM.453 058 125	SMP Pairing Confirm (Ca=73E18614:E690315D:097B644E:0F7E5477)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
1'617	6:25:31 PM.453 455 500	Empty LE Packets (x 454, 7.09 s)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'813	6:25:38 PM.546 547 000	SMP Pairing Confirm (Cb=1C9167F8:D7C28965:B11F13CF:93693F82)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'819	6:25:38 PM.578 024 875	SMP Pairing Random (Na=11C771C1:9BC749A5:9D422608:E9EBE503)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'830	6:25:38 PM.640 296 625	SMP Pairing Random (Nb=3EDDF357:62A33DD7:8EEDC0B5:72EBAEDA)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'842	6:25:38 PM.703 024 125	SMP Pairing Confirm (Ca=3A77F3E1:464B2D75:99A0C7D8:093BFC94)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'848	6:25:38 PM.734 045 875	SMP Pairing Confirm (Cb=4E3FCC81:3BD3E1AF:DE51995E:DB16FDAA)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'855	6:25:38 PM.765 524 375	SMP Pairing Random (Na=085460E2:ACEED819:D338FCAB:4A9EE62C)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'862	6:25:38 PM.796 545 625	SMP Pairing Random (Nb=F751F547:3EF4E62C:55FC6872:F2C09816)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'868	6:25:38 PM.828 023 500	SMP Pairing Confirm (Ca=C265DE01:8BBA1848:1F20391A:A73E55C0)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'874	6:25:38 PM.859 045 375	SMP Pairing Confirm (Cb=E98E2EA3:0F7C407F:3759FC6E:C12D0122)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'880	6:25:38 PM.890 523 375	SMP Pairing Random (Na=56CFAF11:F9BB8D03:48288261:0944237F)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'887	6:25:38 PM.921 545 125	SMP Pairing Random (Nb=3A861EAA:3FC00EB7:C9E63736:422DB8A7)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'893	6:25:38 PM.953 022 875	SMP Pairing Confirm (Ca=A6823F39:14E5FAA8:B36AE266:01B2D08B)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'899	6:25:38 PM.984 044 875	SMP Pairing Confirm (Cb=513383BD:1E64CCAC:A08AB92F:4E11E6E5)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'905	6:25:39 PM.015 522 750	SMP Pairing Random (Na=6596634E:DF6668C2:22EA8AC9:DDCB86AB)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'911	6:25:39 PM.046 544 625	SMP Pairing Random (Nb=52F1015A:913CB0AC:9A249E85:7C3377F0)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'917	6:25:39 PM.078 022 625	SMP Pairing Confirm (Ca=D18328F5:70557D4B:BB03E4C0:987565B8)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'923	6:25:39 PM.109 044 125	SMP Pairing Confirm (Cb=AE80A5C8:04F63BB6:BD014FA4:EC26EC6A)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'929	6:25:39 PM.140 522 250	SMP Pairing Random (Na=7EE38584:2CA71F81:A6376237:039AB495)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
2'935	6:25:39 PM.171 544 000	SMP Pairing Random (Nb=67374C69:4E8B7EFD:2A90AB68:3C075F51)	Slave: "smp" 55:66:77:00:91:01	Master: "smp" 55:66:77:00:91:02
2'941	6:25:39 PM.203 021 875	SMP Pairing Confirm (Ca=718A1F43:0EE8E9E4:F6DEC7E8:09D8E0A6)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01

Figure 10.32: Packet capture of SC Passkey Entry MDSI

- For MISD, you need define it in feature_smp/app_config.h:

```
#define SMP_TEST_MODE        SMP_TEST_SC_PASSKEY_ENTRY_MISD
```

The Passkey input refers to below definition:

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_BY_DEFAULT
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Press SW4 (start connection) on one of the development boards (as Master), after

successful connection, you can see the Master lights up green and the Slave lights up red. After a few seconds, the white and blue lights of both of them light up one after another, which means the pair is successful, and the packet capture is as follows (the boxed part is waiting for the Master to input Passkey)

P...	Time	Item	Transmitter	Receiver
1	6:47:54 PM.000 102 250	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, Scanner 76:E7:D0:69:66:D...	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
637	6:48:00 PM.189 301 750	Connectable ("smp" 55:66:77:00:91:01, 12.7 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
1'528	6:48:04 PM.654 787 875	SMP Pairing Request (Keyboard Only, Bonding, MITM, SC, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'529	6:48:04 PM.655 104 625	SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'540	6:48:04 PM.717 059 875	SMP Pairing Response (Keyboard Display, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'546	6:48:04 PM.748 538 000	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDD...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'558	6:48:04 PM.779 559 750	SMP Pairing Public Key (X=BC55B8C9:99D8BFA3:6FCC97B9:9E23638F:2A5EB048:515891C1:50FBDF66...	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'567	6:48:04 PM.810 580 625	Empty LE Packets (x 90, 1.38 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'823	6:48:06 PM.186 039 000	SMP Pairing Confirm (Ca=DE1D9ABC:235E1745:338774AC:8B6E4B6E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'830	6:48:06 PM.217 061 000	SMP Pairing Confirm (Cb=15730C94:FF5E39FA:B38DD4DB:EACE17E5)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'837	6:48:06 PM.248 539 250	SMP Pairing Random (Na=D14D480F:1A4427A4:8A997B60:333F492B)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'850	6:48:06 PM.310 810 500	SMP Pairing Random (Nb=24AE7FA0:C5EE0CBD:EDA7EF0C:97DC19A9)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'863	6:48:06 PM.373 539 250	SMP Pairing Confirm (Ca=6291D2EC:DFC1D5AE:7CFF8E5F:56DCA82E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'869	6:48:06 PM.404 560 875	SMP Pairing Confirm (Cb=9868FE16:6DDD927E:30258FB7:2F2AEA1A)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'876	6:48:06 PM.436 039 000	SMP Pairing Random (Na=070C6FF6:36FD84C6:54B6757E:FEEEA972)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'884	6:48:06 PM.467 061 000	SMP Pairing Random (Nb=E3306A36:252CD99E:6C9F0839:51763F41)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'891	6:48:06 PM.498 538 750	SMP Pairing Confirm (Ca=DA693016:B3749A74:A24F97D5:39A13AC2)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'897	6:48:06 PM.529 561 000	SMP Pairing Confirm (Cb=7AD6378C:2FF1827A:8B4C51AD:1D2D1A10)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'904	6:48:06 PM.561 039 000	SMP Pairing Random (Na=BC690F50:0761CA8D:EAE58958:3AB6BEAD)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'911	6:48:06 PM.592 060 875	SMP Pairing Random (Nb=DE678F41:84A576A5:A9A0EF9F:1DDC7A7E)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'918	6:48:06 PM.623 539 250	SMP Pairing Confirm (Ca=30F2EE32:23AFD266:BF0840E4:C93080B1)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'925	6:48:06 PM.654 561 250	SMP Pairing Confirm (Cb=64388E1D:176E49A0:9EC34968:1859D49F)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'931	6:48:06 PM.686 039 250	SMP Pairing Random (Na=92A54638:EBC8B871:BFF53688:2072EEAB)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'938	6:48:06 PM.717 060 875	SMP Pairing Random (Nb=5AE201FA:2BCF6202:8726A6F4:A0958BD3)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'945	6:48:06 PM.748 539 500	SMP Pairing Confirm (Ca=E5A7E4F2:3964BADD:ECC3B259:4325589F)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'951	6:48:06 PM.779 561 000	SMP Pairing Confirm (Cb=81125325:0EAB8811:EAE9EEEE:859F8AC3)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'957	6:48:06 PM.811 039 375	SMP Pairing Random (Na=A0F78DFA:BEEBA44F:DB4F15FB:8A8B7FD9)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'963	6:48:06 PM.842 061 375	SMP Pairing Random (Nb=4C63A979:58FD8AC0:B3C0D6CE:794E1D16)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'970	6:48:06 PM.873 539 375	SMP Pairing Confirm (Ca=49CD6D9A:39DEC335:83BCA421:936B1FD6)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02

Figure 10.33: Packet capture of SC Passkey Entry MISD

- For Both Input, you need define it in feature_smp/app_config.h:

```
#define SMP_TEST_MODE        SMP_TEST_SC_PASSKEY_ENTRY_BOTH_INPUT
```

The Passkey input refers to below definition:

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_MANUALLY
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Connect the UART ports PD2(Tx) and PD3(Rx) of both boards to the PC serial port, configure the baud rate 115200. Press SW4 (start connection) of one of the boards (as Master), after successful connection, you can see the Master lights up green and the Slave lights up red. Fill in the custom Pincode to the UART serial port of Slave and Master respectively within 30s. You can see that the SMP process continues and the white and blue lights are on one after another, which means the Pair is successful and the packet capture is as follows (the highlighted part is waiting for both sides to input Passkey).

P...	Time	Item	Transmitter	Receiver
1	1:04:06 PM.016 044 875	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 7.84 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
350	1:04:09 PM.638 305 250	Connectable ("smp" 55:66:77:00:91:01, Scanner 74:85:51:5E:74:F3 (Resolvable), 18 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
1'121	1:04:13 PM.873 790 875	SMP Pairing Request (Keyboard Only, Bonding, MITM, SC, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'122	1:04:13 PM.874 109 000	SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'133	1:04:13 PM.936 063 000	SMP Pairing Response (Keyboard Only, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'139	1:04:13 PM.967 540 625	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDD...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'148	1:04:13 PM.998 562 875	SMP Pairing Public Key (X=1C6E8B9C:4EF0B9CD:42A17961:EACBDAA0:BE4A4968:71EDC4CB:562F08C...	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'155	1:04:14 PM.029 583 625	Empty LE Packets (x 130, 2 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'500	1:04:16 PM.030 042 125	SMP Pairing Confirm (Ca=6B239FE6:D0B6A0EB:B3C2EA4F:4727C87A)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'501	1:04:16 PM.030 43[illegible]375	Empty LE Packets (x 268, 4.19 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'212	1:04:20 PM.217 316 500	SMP Pairing Confirm (Cb=D60965B8:5FA8A11E:51800CC7:7CD006C2)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'218	1:04:20 PM.248 794 875	SMP Pairing Random (Na=37E406BC:9273AC46:C76B1E00:3DEFEF3A)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'230	1:04:20 PM.311 066 875	SMP Pairing Random (Nb=F93F90ED:C5E432BE:FF513A8C:A19D2855)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'241	1:04:20 PM.373 794 875	SMP Pairing Confirm (Ca=49C6984C:0E031F97:B30FF71C:40AA1692)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'247	1:04:20 PM.404 816 625	SMP Pairing Confirm (Cb=43861976:F7E8BB64:53E461AD:14A5BE61)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'255	1:04:20 PM.436 294 875	SMP Pairing Random (Na=AC5BA5B1:CDE03C9D:F62BF38E:677FDC45)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'261	1:04:20 PM.467 316 875	SMP Pairing Random (Nb=41252631:DA176ED7:0D067E2A:C51E558F)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'267	1:04:20 PM.498 795 125	SMP Pairing Confirm (Ca=7142436E:EE2AACF3:7AD201EB:E539AEDB)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'273	1:04:20 PM.529 816 500	SMP Pairing Confirm (Cb=4F54F2D5:64AE83E3:D328A183:EE581EAB)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'280	1:04:20 PM.561 295 125	SMP Pairing Random (Na=6CF863DE:77393BEB:93F9F100:78BDFDD3)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'286	1:04:20 PM.592 316 875	SMP Pairing Random (Nb=6C01162B:FF02AADC:92CF29B4:476B6597)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'292	1:04:20 PM.623 794 500	SMP Pairing Confirm (Ca=17FA9415:A1610484:E18F7C0E:A27EF048)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'298	1:04:20 PM.654 816 875	SMP Pairing Confirm (Cb=E325C554:E4C8B441:CC17C067:0E48D798)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'304	1:04:20 PM.686 295 000	SMP Pairing Random (Na=36CEE74B:034121CF:441EA389:718317A2)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'312	1:04:20 PM.717 316 875	SMP Pairing Random (Nb=24658D00:FC2FBF9B:1705E61D:796B7E81)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'319	1:04:20 PM.748 794 625	SMP Pairing Confirm (Ca=041B6540:1639D03B:8B3A3B6D:DE1233FB)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'326	1:04:20 PM.779 817 125	SMP Pairing Confirm (Cb=29D64F19:5FB7F5B8:3E190A13:DBDF7FCD)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'333	1:04:20 PM.811 295 250	SMP Pairing Random (Na=D3F2D7D6:C1B455B1:754EF017:FFA2C80C)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'339	1:04:20 PM.842 317 250	SMP Pairing Random (Nb=A7DD2BAA:1E0D448D:E70C1AA7:76B1A70D)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 10.34: Packet capture of SC Passkey Entry Both Input

10.6.10 Secure Connections Numeric Comparison

A demonstration of Master and Slave pairing with Secure Connections Numeric Comparison by enabling SMP. According to the Bluetooth protocol, when Secure Connections is used and both parties are DisplayYesNo or KeyboardDisplay enabled, Numeric Comparison will be used for matching. To use this feature, define in feature_smp/app_config.h:

```
#define SMP_TEST_MODE       SMP_TEST_SC_NUMERIC_COMPARISON
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Press button SW4 (start connection) on one of the development boards (as Master), after successful connection, you can see the Master lights up green and Slave lights up red, at this time, you can see the value of Numeric Comparison Pincode in the Log of both sides, theoretically they should be equal. Press button SW3 on the Master and Slave development boards respectively to send YES, and you can see the white and blue lights of both of them light up one after another, representing the success of the Pair, and the packet capture is as follows (the boxed part is waiting for the Master and Slave to confirm by pressing the button respectively).

P...	Time	Item	Transmitter	Receiver
1	10:18:49 AM.004 797 500	Connectable ("smp" 55:66:77:00:91:01, 21 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
4	10:18:49 AM.009 833 125	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 2.36 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
428	10:18:51 AM.392 156 875	SMP Pairing Request (Keyboard Display, Bonding, MITM, SC, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
429	10:18:51 AM.392 475 500	SMP Security Request (Bonding, MITM, SC)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
439	10:18:51 AM.454 428 625	SMP Pairing Response (Display Yes No, Bonding, MITM, SC, Int=IdKey, Rsp=IdKey)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
445	10:18:51 AM.485 906 875	SMP Pairing Public Key (Debug Key, X=20B003D2:F297BE2C:5E2C83A7:E9F9A5B9:EFF49111:ACF4FDD...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
455	10:18:51 AM.516 928 625	SMP Pairing Public Key (X=3CAAB646:C553B79F:43BE2158:DBEA0548:F9A787C8:19AD56A9:B0F9D507...	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
461	10:18:51 AM.518 855 375	SMP Pairing Confirm (Cb=9BA10FD4:95D7F814:6361D2E9:20AB9444)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
467	10:18:51 AM.548 406 625	SMP Pairing Random (Na=FC1BC417:C4DAA576:D5AA8BEE:150DE51E)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
480	10:18:51 AM.610 678 750	SMP Pairing Random (Nb=C3A14FEE:D3244A53:9ABEA4FA:1E27193D)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
484	10:18:51 AM.641 699 750	Empty LE Packets (x 563, 1 retry, 8.78 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'937	10:19:00 AM.423 412 625	SMP Pairing DH Key Check (E3F33D54:4F349CCE:95CC3006:F411E2C6)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'938	10:19:00 AM.423 810 875	Empty LE Packets (x 142, 2.22 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'306	10:19:02 AM.641 935 625	SMP Pairing DH Key Check (B4BAFC08:DD2FE6D2:2A1FAA1A:AAE013D6)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'312	10:19:02 AM.673 413 875	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0xDF1AA076C1F78C53, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'318	10:19:02 AM.704 435 625	LLCP Encryption Response (SKDs=0xDDED3C7C3E635E5B, IVs=0x90B4C9FD)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'334	10:19:02 AM.798 185 750	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'341	10:19:02 AM.829 664 250	LLCP Start Encryption Response	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'347	10:19:02 AM.860 686 000	LLCP Start Encryption Response	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'357	10:19:02 AM.923 186 000	SMP Identity Information (IRK=A3B086D9:459694CF:7A1C228E:6FD8D393)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'367	10:19:02 AM.985 685 875	SMP Identity Address Information (BDADDR=55:66:77:00:91:02)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'374	10:19:03 AM.017 164 125	SMP Identity Information (IRK=A728C802:7FC864DC:FF9555DC:87B7B5F8)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'382	10:19:03 AM.048 414 125	SMP Identity Address Information (BDADDR=55:66:77:00:91:01)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'383	10:19:03 AM.048 771 375	Empty LE Packets (x 444, 1 retry, 6.94 s)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01

Figure 10.35: Packet capture of SC Numeric Comparison

10.6.11 Legacy OOB

Master and Slave enable the SMP function to demonstrate pairing in the Legacy OOB method. According to the Bluetooth protocol, when both parties support OOB and have each other's OOB data, the Legacy OOB method will be used for matching. To use this feature, define in feature_smp/app_config.h:

```
#define SMP_TEST_MODE        SMP_TEST_LEGACY_OOB
```

Manually input OOB data via UART is chosen here, refer to below definition:

```
#define PASSKEY_ENTRY_METHOD        PASSKEY_ENTRY_MANUALLY
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Connect PD2(Tx) and PD3(Rx) of the two development boards to the PC serial port and configure the baud rate 115200. Press SW4 (start connection) on one of the development boards (as Master), after successful connection, you can see the green light on Master and red light on Slave. At this time, you can see the Requset OOB in the log of both sides. Fill the custom OOB data into the UART serial port of Slave and Master respectively within 30s. You can see the white and blue lights of both of them light up one after another, which means they are successful in Pair.

P...	Time	Item	Transmitter	Receiver
1	4:03:11 PM.000 834 125	Connectable ("smp" 55:66:77:00:91:01, Scanner 74:85:51:5E:74:F3 (Resolvable), 16.6 s)	Master: "smp" 55:66:77:00:91:01	Slave: "Scanning Device"
4	4:03:11 PM.009 979 250	Connectable ("smp" 55:66:77:00:91:02, Initiator "smp" 55:66:77:00:91:01, 5.84 s)	Master: "smp" 55:66:77:00:91:02	Slave: "smp" 55:66:77:00:91:01
139	4:03:11 PM.785 103 375	LLCP Reserved (0x58)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
608	4:03:14 PM.441 873 250	Unknown LE Transfer (Reserved (0x9D))	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'032	4:03:16 PM.875 538 125	SMP Pairing Request (Keyboard Only, OOB, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdKey)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
1'033	4:03:16 PM.875 857 250	SMP Security Request (Bonding, MITM)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'037	4:03:16 PM.937 809 375	SMP Pairing Response (Keyboard Display, OOB, Bonding, MITM, Int=EncKey | IdKey, Rsp=EncKey | IdK...	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
1'045	4:03:16 PM.968 830 750	Empty LE Packets (x 385, 19 retries, 6.06 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'080	4:03:23 PM.031 790 125	SMP Pairing Confirm (Ca=E6F10862:0827386A:9512209D:CF97D08D)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'087	4:03:23 PM.094 290 000	SMP Pairing Confirm (Cb=4C99C047:A392796D:5DA55C8B:E9116B85)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'098	4:03:23 PM.125 311 875	SMP Pairing Random (Nb=00BC5647:B83C0763:890213B3:3B67CC17)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'104	4:03:23 PM.156 790 250	LLCP Encryption Request (Rnd=0x0000000000000000, EDIV=0x0000, SKDm=0x790CA9C40652ED1B, I...	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'110	4:03:23 PM.187 812 000	LLCP Encryption Response (SKDs=0x3290993DC771A965, IVs=0xD2D8DDEF)	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'127	4:03:23 PM.281 562 000	LLCP Start Encryption Request	Slave: "smp" 55:66:77:00:91:02	Master: "smp" 55:66:77:00:91:01
2'131	4:03:23 PM.312 582 750	Empty LE Packets (x 2, 80 us)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02
2'133	4:03:23 PM.313 039 750	Encrypted ACL Link Layer Traffic (x 8, 4.22 s)	Master: "smp" 55:66:77:00:91:01	Slave: "smp" 55:66:77:00:91:02

Figure 10.36: Packet capture of Legacy OOB

You can see that the encrypted content here is not parsed, parsing requires STK, Telink does not provide interface and method to get STK. However, it is possible to get LTK by reading RAM, so that after reconnecting with LTK encryption, LTK can be loaded into the packet capturer.

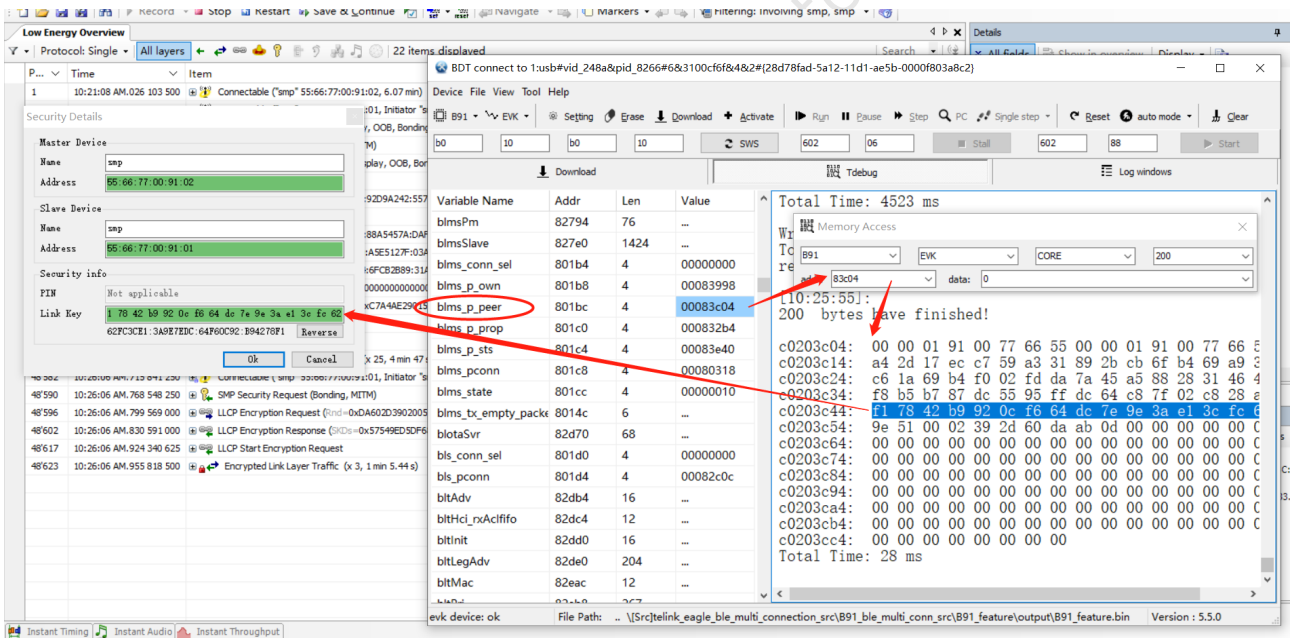


Figure 10.37: Legacy OOB load LTK

Then it can decrypt the encrypted content after reconnection.

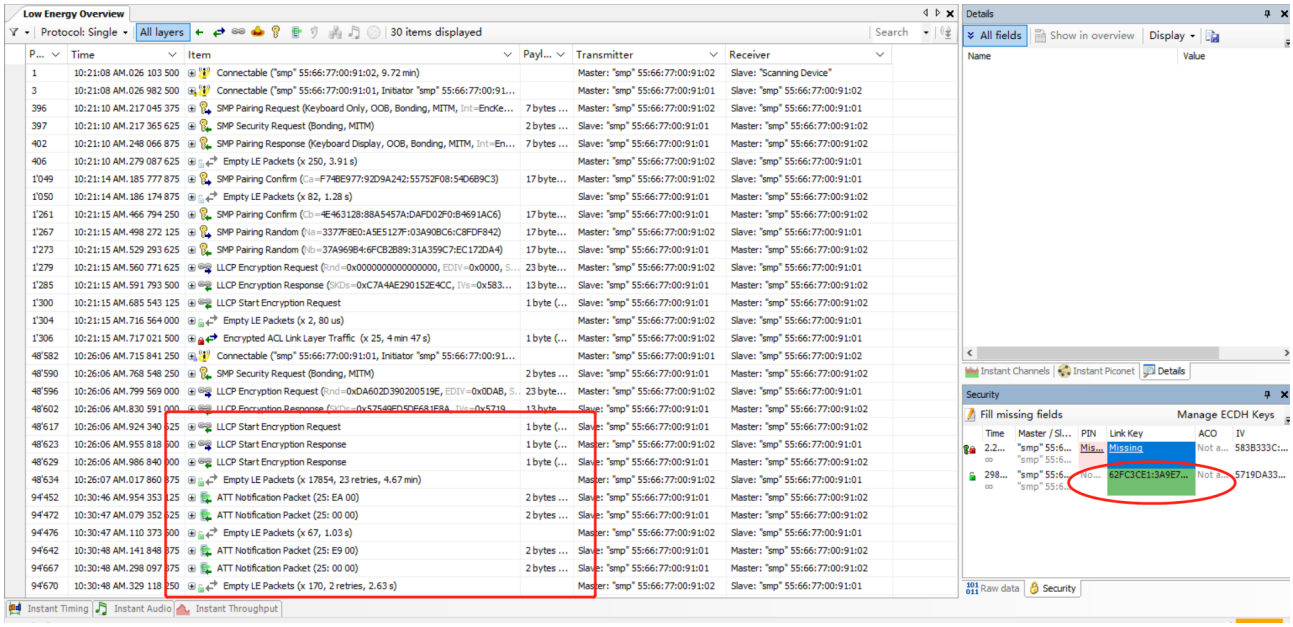


Figure 10.38: Decrypt Legacy OOB LTK

10.6.12 Secure Connections OOB

Master and Slave enable the SMP function to demonstrate pairing as Secure Connections OOB, which is not supported in SDK for now.

10.6.13 Custom Pair

When the SMP function of Slave and Master is disabled, Telink provides a way to automatically connect back to the device without SMP - Custom Pair, which is demonstrated here. To use this feature, define in feature_smp/app_config.h:

```
#define SMP_TEST_MODE       SMP_TEST_CUSTOM_PAIR
```

Compile B91_feature, burn /eagle_ble_sdk/B91_feature/output/B91_feature.bin to the two development boards respectively. Press SW4 (Start Connection) on one of the development boards (as Master), and when the connection is successful, you can see the Master light up green and the Slave light up red, which means both of them are successfully connected. If you re-power the Master or Slave, you can see the lights of both sides light up soon after the lights of the other side go off, which means the reconnection is successful.

10.6.14 Exception handling

10.6.14.1 No response when pressing the button

(1) Hardware reasons: It may be due to the jumper of the key is not connected correctly, please refer to the following figure.

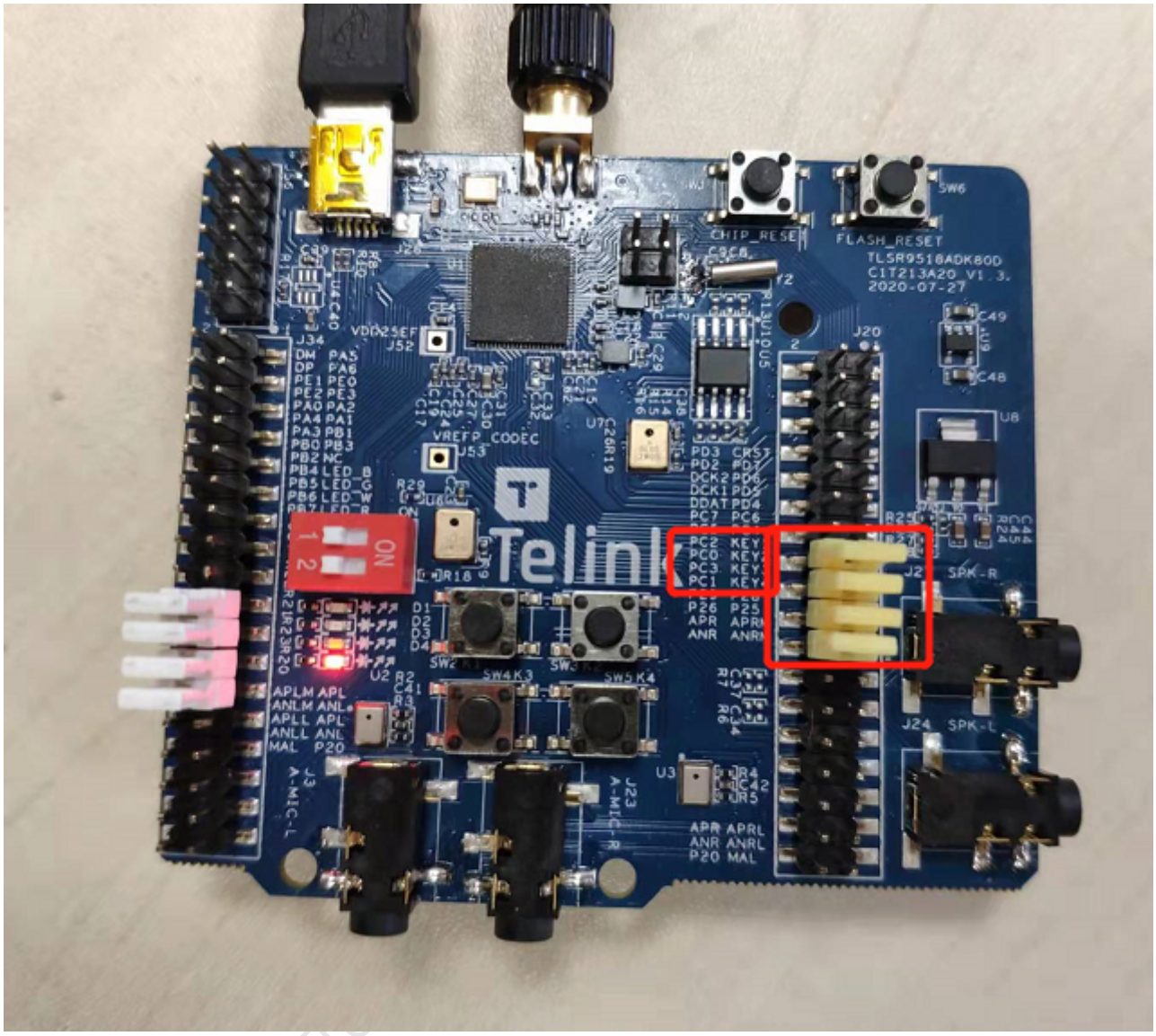


Figure 10.39: Incorrect Key jumper

(2) Environment reason: Probably due to the development board not having reset to make the program run after burning.

10.6.14.2 LED is not on

(1) Hardware reasons: may be due to the LED jumper is not connected correctly, please refer to the following figure.

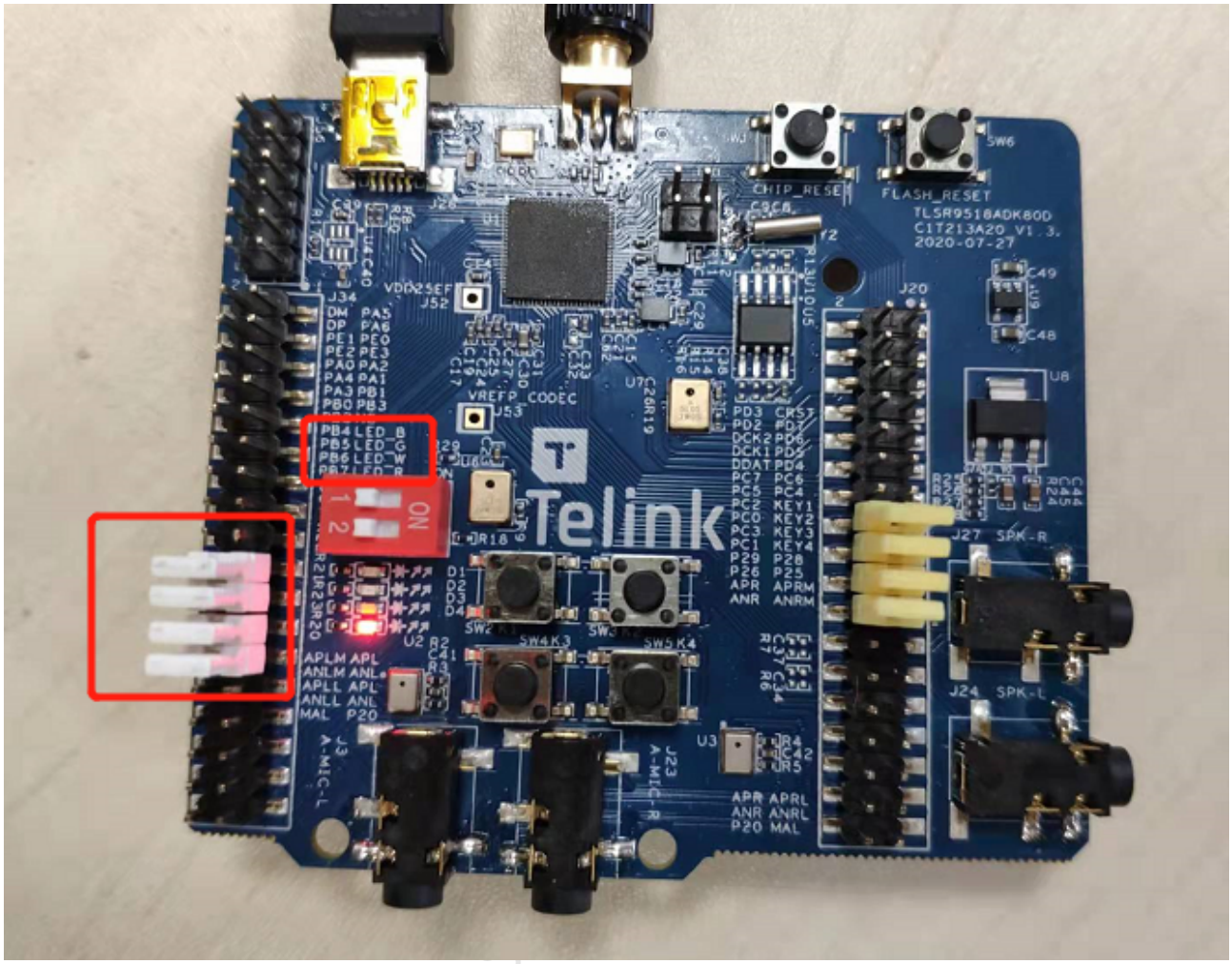


Figure 10.40: Incorrect LED jumper

10.7 feature_ota

- **Function**: function demonstration of BLE OTA
- **Main hardware**: B91 development board x 2

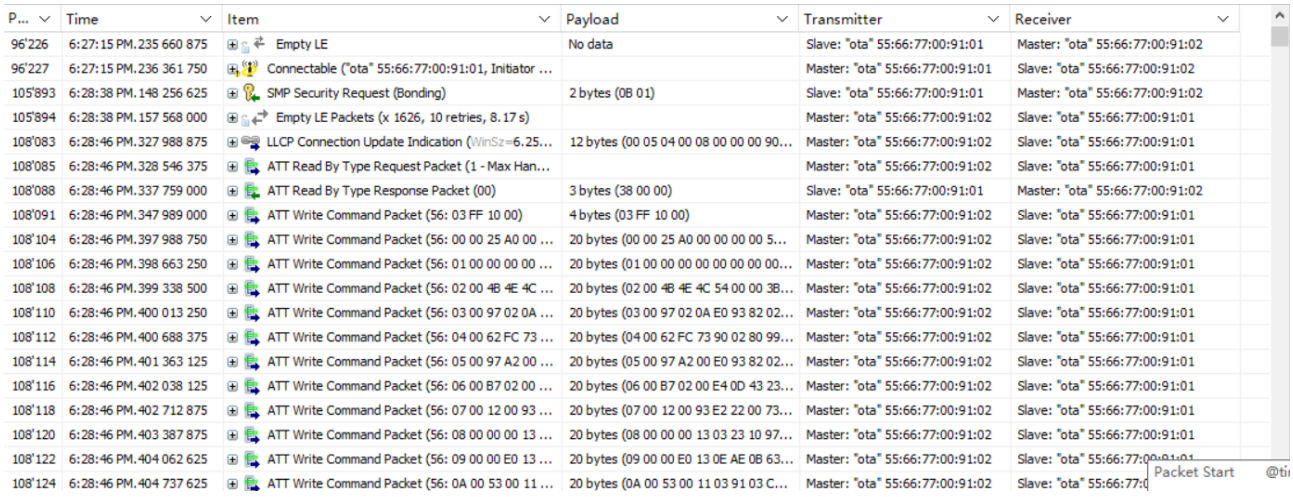
Take Telink B91 BLE Multiple Connection SDK as example, the applicaiton layer code is under eagle_ble_sdk/vendor/B91_feature/feature_ota, you need modify the definition in eagle_ble_sdk/vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                TEST_OTA
```

to activate this part of the code and do the OTA demo.

After compiling the firmware, burn it into two development boards, and burn OTA firmware to Flash 0x80000 on one of the boards (using the same compiled firmware as above). After re-powering both boards, press SW2 or SW3 of Master 5 times to trigger the OTA test mode, you can see the blue light and green light flashing slowly three times, which means the OTA test mode is switched successfully.

Press Master's button SW2 to start OTA, you can see the blue light on both development boards, which means OTA is in progress. The packet capture shows that the Master keeps sending WriteCmd to the Slave.

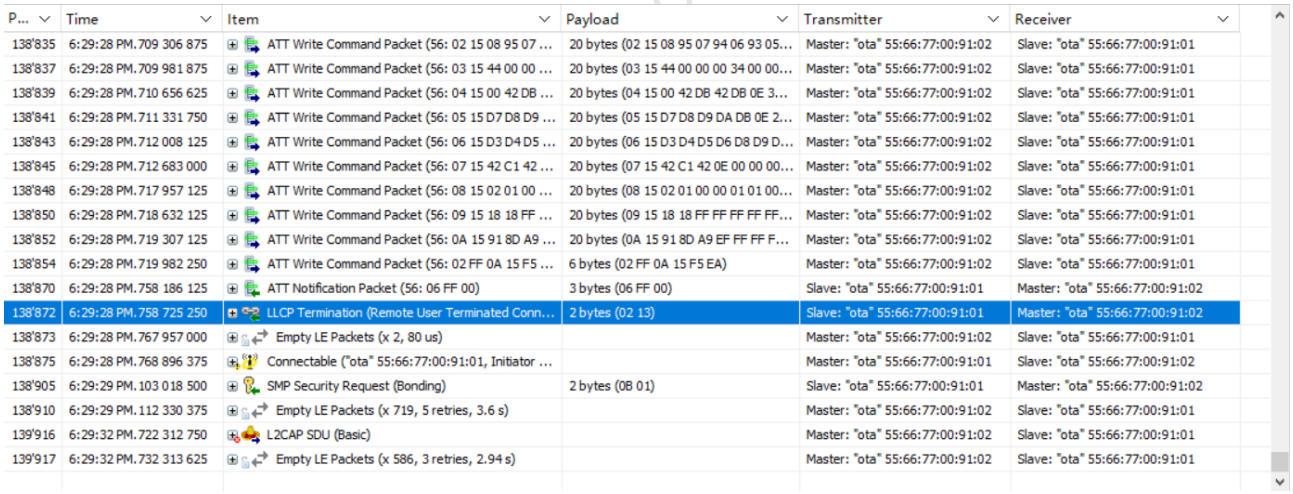


P...	Time	Item	Payload	Transmitter	Receiver
96'226	6:27:15 PM.235 660 875	Empty LE	No data	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
96'227	6:27:15 PM.236 361 750	Connectable ("ota" 55:66:77:00:91:01, Initiator ...		Master: "ota" 55:66:77:00:91:01	Slave: "ota" 55:66:77:00:91:02
105'893	6:28:38 PM.148 256 625	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
105'894	6:28:38 PM.157 568 000	Empty LE Packets (x 1626, 10 retries, 8.17 s)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'083	6:28:46 PM.327 988 875	LLCP Connection Update Indication (WinSz=6.25...	12 bytes (00 05 04 00 08 00 00 00 90...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'085	6:28:46 PM.328 546 375	ATT Read By Type Request Packet (1 - Max Han...		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'088	6:28:46 PM.337 759 000	ATT Read By Type Response Packet (00)	3 bytes (38 00 00)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
108'091	6:28:46 PM.347 989 000	ATT Write Command Packet (56: 03 FF 10 00)	4 bytes (03 FF 10 00)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'104	6:28:46 PM.397 988 750	ATT Write Command Packet (56: 00 00 25 A0 00 ...	20 bytes (00 00 25 A0 00 00 00 00 5...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'106	6:28:46 PM.398 663 250	ATT Write Command Packet (56: 01 00 00 00 00 ...	20 bytes (01 00 00 00 00 00 00 00 00...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'108	6:28:46 PM.399 338 500	ATT Write Command Packet (56: 02 00 4B 4E 4C ...	20 bytes (02 00 4B 4E 4C 54 00 00 3B...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'110	6:28:46 PM.400 013 250	ATT Write Command Packet (56: 03 00 97 02 0A ...	20 bytes (03 00 97 02 0A E0 93 82 02...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'112	6:28:46 PM.400 688 375	ATT Write Command Packet (56: 04 00 62 FC 73 ...	20 bytes (04 00 62 FC 73 90 02 80 99...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'114	6:28:46 PM.401 363 125	ATT Write Command Packet (56: 05 00 97 A2 00 ...	20 bytes (05 00 97 A2 00 E0 93 82 02...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'116	6:28:46 PM.402 038 125	ATT Write Command Packet (56: 06 00 B7 02 00 ...	20 bytes (06 00 B7 02 00 E4 0D 43 23...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'118	6:28:46 PM.402 712 875	ATT Write Command Packet (56: 07 00 12 00 93 ...	20 bytes (07 00 12 00 93 E2 22 00 73...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'120	6:28:46 PM.403 387 875	ATT Write Command Packet (56: 08 00 00 00 13 ...	20 bytes (08 00 00 00 13 03 23 10 97...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'122	6:28:46 PM.404 062 625	ATT Write Command Packet (56: 09 00 00 E0 13 ...	20 bytes (09 00 00 E0 13 0E AE 0B 63...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
108'124	6:28:46 PM.404 737 625	ATT Write Command Packet (56: 0A 00 53 00 11 ...	20 bytes (0A 00 53 00 11 03 91 03 C...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:0

Packet Start @ti

Figure 10.41: Start feature_ota OTA

When the upgrade is complete, the Slave sends a Notify command to the Master, which means that the upgrade was successful, and then sends a Terminate command to disconnect. Since the two are connected by SMP, the Master has the Slave's information and reconnects immediately.



P...	Time	Item	Payload	Transmitter	Receiver
138'835	6:29:28 PM.709 306 875	ATT Write Command Packet (56: 02 15 08 95 07 ...	20 bytes (02 15 08 95 07 94 06 93 05...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'837	6:29:28 PM.709 981 875	ATT Write Command Packet (56: 03 15 44 00 00 ...	20 bytes (03 15 44 00 00 00 34 00 00...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'839	6:29:28 PM.710 656 625	ATT Write Command Packet (56: 04 15 00 42 DB ...	20 bytes (04 15 00 42 DB 42 DB 0E 3...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'841	6:29:28 PM.711 331 750	ATT Write Command Packet (56: 05 15 D7 D8 D9 ...	20 bytes (05 15 D7 D8 D9 DA DB 0E 2...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'843	6:29:28 PM.712 008 125	ATT Write Command Packet (56: 06 15 D3 D4 D5 ...	20 bytes (06 15 D3 D4 D5 D6 D8 D9 D...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'845	6:29:28 PM.712 683 000	ATT Write Command Packet (56: 07 15 42 C1 42 ...	20 bytes (07 15 42 C1 42 0E 00 00 00...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'848	6:29:28 PM.717 957 125	ATT Write Command Packet (56: 08 15 02 01 00 ...	20 bytes (08 15 02 01 00 00 01 01 00...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'850	6:29:28 PM.718 632 125	ATT Write Command Packet (56: 09 15 18 18 FF ...	20 bytes (09 15 18 18 FF FF FF FF FF...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'852	6:29:28 PM.719 307 125	ATT Write Command Packet (56: 0A 15 91 8D A9 ...	20 bytes (0A 15 91 8D A9 EF FF FF F...	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'854	6:29:28 PM.719 982 250	ATT Write Command Packet (56: 02 FF 0A 15 F5 ...	6 bytes (02 FF 0A 15 F5 EA)	Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'870	6:29:28 PM.758 186 125	ATT Notification Packet (56: 06 FF 00)	3 bytes (06 FF 00)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
138'872	6:29:28 PM.758 725 250	LLCP Termination (Remote User Terminated Conn...	2 bytes (02 13)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
138'873	6:29:28 PM.767 957 000	Empty LE Packets (x 2, 80 us)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
138'875	6:29:28 PM.768 896 375	Connectable ("ota" 55:66:77:00:91:01, Initiator ...		Master: "ota" 55:66:77:00:91:01	Slave: "ota" 55:66:77:00:91:02
138'905	6:29:29 PM.103 018 500	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "ota" 55:66:77:00:91:01	Master: "ota" 55:66:77:00:91:02
138'910	6:29:29 PM.112 330 375	Empty LE Packets (x 719, 5 retries, 3.6 s)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
139'916	6:29:32 PM.722 312 750	L2CAP SDU (Basic)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01
139'917	6:29:32 PM.732 313 625	Empty LE Packets (x 586, 3 retries, 2.94 s)		Master: "ota" 55:66:77:00:91:02	Slave: "ota" 55:66:77:00:91:01

Figure 10.42: Complete feature_ota OTA

10.8 feature_whitelist

- **Function:** function demonstration of BLE whitelist
- **Main hardware:** B91 development board x 2

Take Telink B91 BLE Multiple Connection SDK as example, the applicaiton layer code is under eagle_ble_sdk/ vendor/ B91_feature/ feature_whitelist, you need modify the definition in eagle_ble_sdk/ vendor/B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                TEST_WHITELIST
```

to activate this part of the code and do a demonstration of the whitelist function.

Since the Master whitelist to be connected and the Slave whitelist to be connected share the same whitelist list in the SDK, here is a demonstration of connecting a Master whitelist device and connecting a Slave whitelist device separately.

First compile a master-slave device firmware that enables whitelist functionality but does not use a whitelist (called NoWhiteList for ease of reference later), and initialize the code

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
 ↪  OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL);
```

modify it to

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
 ↪  OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY);
```

then modify

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
 ↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_WL);
```

to

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
 ↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
```

Compile, NoWhiteList.bin.

10.8.1 Master set scan Slave whitelist

The demonstration of connecting to a Slave whitelist device is to add the MAC address of the Slave to be connected to the whitelist, and the Telink device as the Master decides whether to connect by filtering the addresses of the scanned advertisings.

Based on the NoWhiteList code, restore the scan parameter configuration statement:

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
 ↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_ANY);
```

to

```
blc_ll_setScanParameter(SCAN_TYPE_PASSIVE, SCAN_INTERVAL_100MS, SCAN_WINDOW_100MS,
↪  OWN_ADDRESS_PUBLIC, SCAN_FP_ALLOW_ADV_WL);
```

Compile, get the firmware for Master, burn it to one of the development boards as Master, burn NoWhiteList.bin to the other board, and change the MAC address to whitelist address 33:88:99:99:99:99.

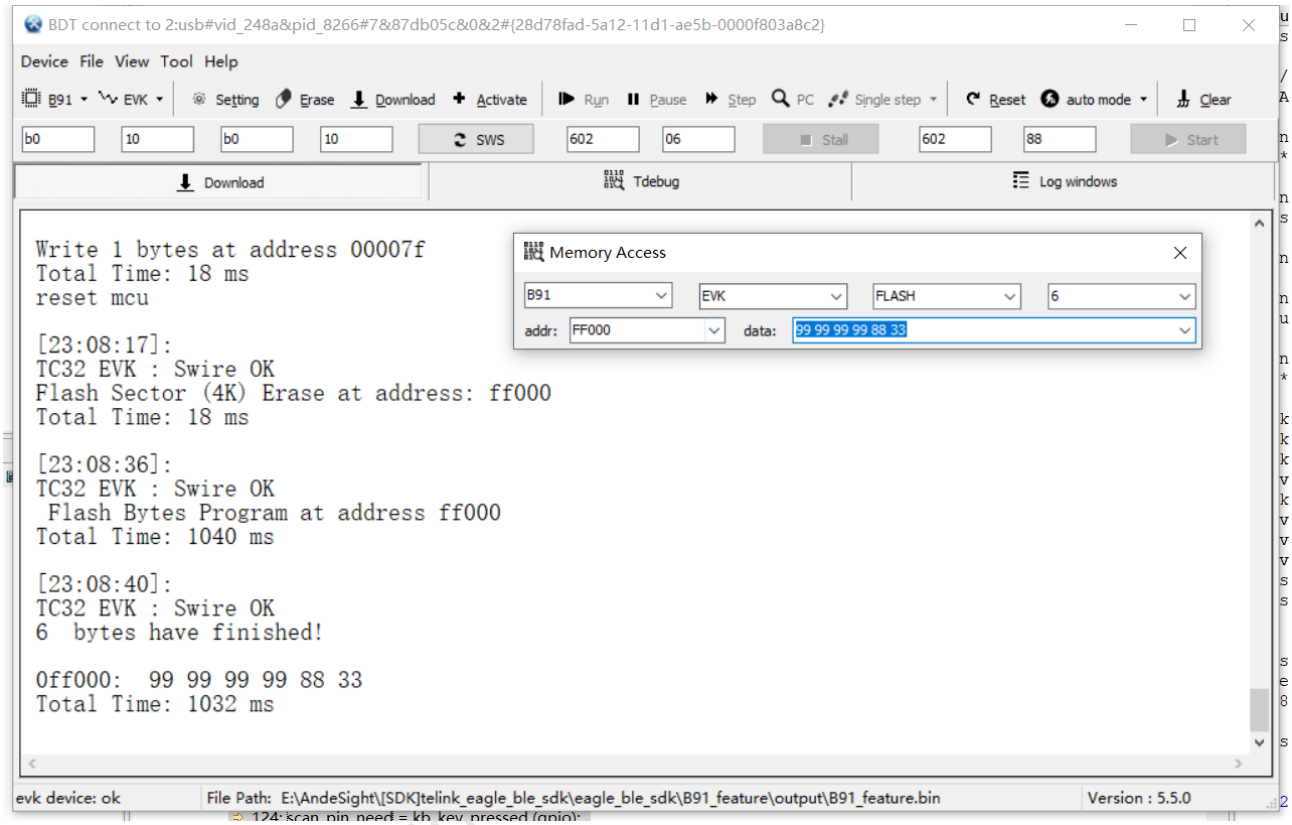


Figure 10.43: Set whitelist device MAC address

After burning is completed, by pressing SW4 of Master to start the connection, you can see that Master and Slave are successfully connected, and the packet capture is as follows.

P...	Time	Item	Payload	Transmitter
1	12:11:12 AM.020 653 375	Connectable ("whitelist" 55:66:77:00:91:02, 8.92 s)		Master: "whitelist" 55:66:77:00:...
256	12:11:14 AM.605 363 125	Connectable ("whitelist" 99:99:99:99:88:33, Initiator "whitelist" 55:66:77:00:91:02, 352 us)		Master: "whitelist" 99:99:99:99:...
260	12:11:14 AM.627 019 875	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "whitelist" 99:99:99:99:88...
269	12:11:14 AM.657 581 375	Empty LE Packets (x 395, 9 retries, 6.28 s)		Master: "whitelist" 55:66:77:00:...

Figure 10.44: Master connection whitelist Slave packet capture

If the MAC address of the Slave is not 33:88:99:99:99:99, press the SW4 button of the Master, there will be no response, and the packets after the connection is established will not be seen in the packet capture.

10.8.2 Slave sets Master connection whitelist

The demonstration of connecting a Master whitelist device is to add the MAC address of the Master to be connected to the whitelist, and the Telink device as Slave decides whether to respond by filtering the scanned or connected packets.

Based on the NoWhiteList code, restore the scan parameter configuration statement:

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
 ↪  OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_ANY_ALLOW_CONN_ANY);
```

to

```
blc_ll_setAdvParam(ADV_INTERVAL_30MS, ADV_INTERVAL_30MS, ADV_TYPE_CONNECTABLE_UNDIRECTED,
 ↪  OWN_ADDRESS_PUBLIC, 0, NULL, BLT_ENABLE_ADV_ALL, ADV_FP_ALLOW_SCAN_WL_ALLOW_CONN_WL);
```

Compile, get the firmware of Slave, burn it to one of the development boards as Slave, burn NoWhiteList.bin to the other board, and change the MAC address to whitelist address 33:88:99:99:99:99. After the burning is completed, by pressing SW4 of Master to start the connection, you can see that Master and Slave are successfully connected, and the packet capture is as follows.

P...	Time	Item	Payload	Transmitter	Receiver
4	12:41:13 AM.026 290 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 99:99:99:99:88:33
591	12:41:15 AM.640 940 375	SMP Security Request (Bonding)	2 bytes (0B 01)	Slave: "whitelist" 55:66:77:00:91:01	Master: "whitelist" 99:99:99:99:88:33
596	12:41:15 AM.650 042 250	Connectable ("whitelist" 55:66:77:00:91:01, 10.5 s)		Master: "whitelist" 55:66:77:00:91:01	Slave: "Scanning Device"
600	12:41:15 AM.671 502 000	Empty LE Packets (x 661, 15 retries, 10.5 s)		Master: "whitelist" 99:99:99:99:88:33	Slave: "whitelist" 55:66:77:00:91:01

Figure 10.45: Slave connects to whitelist Master packet capture

If the MAC address of the Master is not 33:88:99:99:99:99:99, press the SW4 button of Maste and you will see that the Master sends a Connect Request, but the Slave does not respond, so the Master keeps trying to connect.

P...	Time	Item	Payload	Transmitter	Receiver
1	12:43:56 AM.001 288 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
1	12:43:56 AM.001 288 625	Connectable Undirected Adv (55:66:77:00:91:01, Name=".	30 bytes (01 91 00 77 66 ...	Master: "whitelist" 55:66:77:00:91:01	Slave: "Scanning Device"
5'942	12:44:43 AM.648 815 625	Connection Indication Packet (55:66:77:00:91:02 > 55:...	34 bytes (02 91 00 77 66 ...	Slave: "whitelist" 55:66:77:00:91:02	Master: "whitelist" 55:66:77:00:91:0
5'944	12:44:43 AM.679 441 875	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
5'954	12:44:43 AM.763 094 750	Empty LE (2 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'943	12:44:43 AM.669 345 125	Empty LE Unit	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'943	12:44:43 AM.669 345 125	Empty LE Packet	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'950	12:44:43 AM.731 844 750	Empty LE Unit	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'954	12:44:43 AM.763 094 750	Empty LE Unit	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'969	12:44:43 AM.893 817 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
5'980	12:44:43 AM.973 718 875	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
5'992	12:44:44 AM.069 442 625	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
6'003	12:44:44 AM.118 092 250	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
6'018	12:44:44 AM.279 442 750	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
6'029	12:44:44 AM.362 466 875	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
6'041	12:44:44 AM.453 192 250	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02
6'052	12:44:44 AM.536 216 125	Empty LE (3 retries)	No data	Master: "whitelist" 55:66:77:00:91:02	Slave: "whitelist" 55:66:77:00:91:01
6'064	12:44:44 AM.625 693 125	Connectable ("whitelist" 55:66:77:00:91:01, Initiator "whit...		Master: "whitelist" 55:66:77:00:91:01	Slave: "whitelist" 55:66:77:00:91:02

Figure 10.46: Slave connects to non-whitelisted Master to capture packets

10.9 feature_soft_timer

- **Function**: Demonstration of the Soft Timer function. And as a reference for the demonstration of the IO Debug method.
- **Main hardware**: B91 development board, logic analyzer or oscilloscope

Take Telink B91 BLE Multiple Connection SDK as example, the application layer code is under eagle_ble_sdk/ vendor/ B91_feature/ feature_soft_timer, you need modify the definition in eagle_ble_sdk/ vendor/ B91_feature/feature_config.h:

```
#define FEATURE_TEST_MODE                TEST_SOFT_TIMER
```

to activate this part of the code and enable IO Debug in app_config.h:

```
#define DEBUG_GPIO_ENABLE                1
```

to do a functional demonstration of Soft Timer and IO Debug.

Connect PE1 on the development board as PM IO to observe hibernation wake-up. Connect PA3, PB0, PB2, PE0 out as Debug IO 4, 5, 6, 7 for signal display respectively, refer to macro definition DEBUG_GPIO_ENABLE.

After compiling the firmware, burning it into the development board, powering up the logic analyzer to capture the Debug IOs signals, you can see the effect of the 5 IOs captured on the logic analyzer as follows.

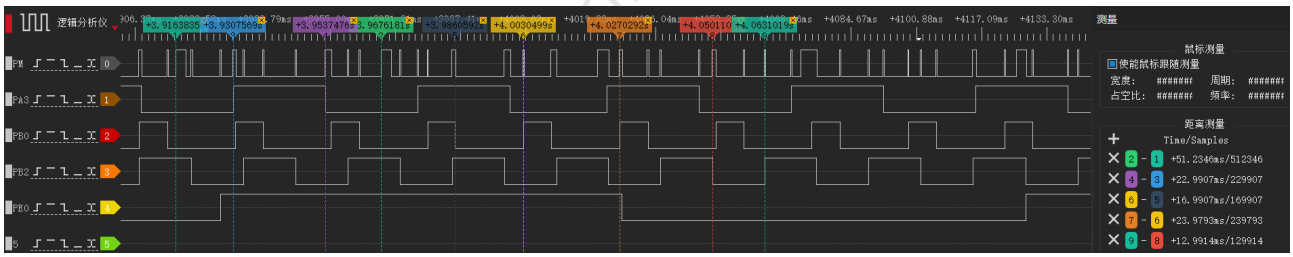


Figure 10.47: SoftTimer Debug IO capture

The graph shows that the advertising interval is 50ms (or so, with a little dynamic adjustment at the bottom), PA3 toggles every 23ms, PB0 toggles at alternating intervals of 7ms and 17ms, PB2 toggles every 13ms, PE0 toggles every 100ms, refer to the code enabled by the macro definition BLT_SOFTWARE_ENABLE. It can be seen that the device is woken up early when the Soft Timer event is triggered, which means that the events set by Soft Timer are not affected by hibernation.

11 Other Modules

11.1 PA

If you need to use RF PA, please refer to drivers/8258/rf_pa.h for 8253/8258, drivers/8278/driver_ext/ext_rf.h for 8273/8278, and drivers/B91/ext_driver/software_pa.h for B91 series chip.

First enable the macro below, by default it is disabled.

```
#ifndef PA_ENABLE
#define PA_ENABLE                   0
#endif
```

During system initialization, call PA initialization.

```
void rf_pa_init(void);
```

Referring to the code implementation, in this initialization, PA_TXEN_PIN and PA_RXEN_PIN are set to GPIO output mode and the initial state is output 0. The GPIOs corresponding to the PA of TX and RX need to be defined by the user:

```
#ifndef PA_TXEN_PIN
#define PA_TXEN_PIN                 GPIO_PB2
#endif

#ifndef PA_RXEN_PIN
#define PA_RXEN_PIN                 GPIO_PB3
#endif
```

Also register void (*rf_pa_callback_t)(int type) as a callback handler function for PA, which actually handles the following 3 PA states: PA off, TX PA on, 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 rf_pa_init above, app_rf_pa_handler is registered to the bottom layer callback, and BLE will automatically call app_rf_pa_handler's processing when it is in various states.

12 Debug Method

This chapter introduces several debugging methods commonly used in the development process.

12.1 GPIO Simulates UART Printing

To facilitate the debugging of B85m and B91 users，Telink BLE Multiple Connection SDK provides an implementation of GPIO simulating UART serial port to output debugging information. This method is only for reference, not an officially recommended method for outputting debugging information. After defining the macro UART_PRINT_DEBUG_ENABLE in app_config.h in the routine as 1, you can directly use the printf interface consistent with the C language syntax rules in the code for serial output. The default configuration of the GPIO for simulating UART is available in each application routine and can be changed by the user as required.

```
////////////////////////////////////// PRINT DEBUG INFO //////////////////////////////////////
#if (UART_PRINT_DEBUG_ENABLE)
    //the baud rate should not bigger than 115200 when MCU clock is 16M)
    //the baud rate should not bigger than 1000000 when MCU clock is 24M)
    #define PRINT_BAUD_RATE                 1000000
    #define DEBUG_INFO_TX_PIN               GPIO_PD7
    #define PULL_WAKEUP_SRC_PD7             PM_PIN_PULLUP_10K
    #define PD7_OUTPUT_ENABLE               1
    #define PD7_DATA_OUT                    1 //must
#endif
```

Figure 12.1: Definition of GPIO for simulating UART

In general, only modify the baud rate and other IO names in the definition of each line (all PD7 in the picture).

Note:

- The baud rate currently supports up to 1 Mbps.
- Since the printing of the GPIO simulating UART serial port will be interrupted by the interrupts, the timing of the simulated UART will be inaccurate, so in the actual use process, there will be occasional garbled printing.
- Since the printing of the GPIO simulating UART serial port will occupy the CPU, it is not recommended to add printing to the interrupts, which will affect the interrupt tasks with high timing requirements.

12.2 BDT Tool Reads the Global Variables Value

The official BDT tool provided by Telink can be used not only for burning firmware, but also for some online debug, where the values of global variables in the code can be read under the Tdebug tab. Users can add global variables in the code according to their needs and read them in Tdebug. For details, please refer to the Debug chapter of the BDT User Guide.

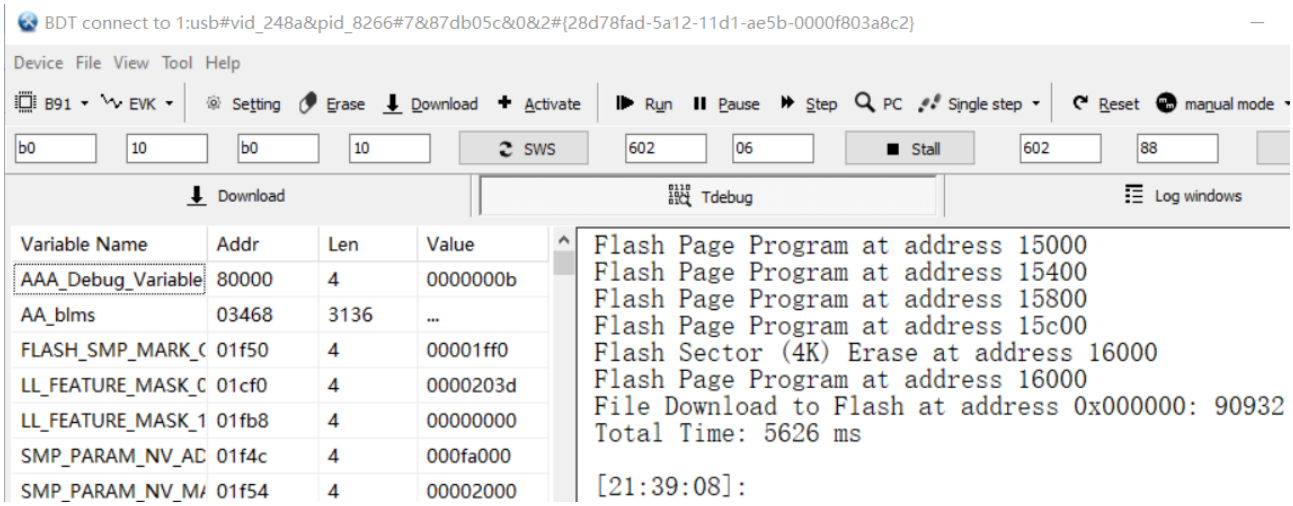


Figure 12.2: BDT tool reads the global variables value

Note:

- When the chip is in sleep state, the read value is all 0.
- This function depends on the generated list file (the default file name generated by B85m is xxx.lst and by B91 is objdump.txt), so the user should also provide the list file as much as possible when providing debug firmware to the official, so that Telink engineers can read the values of some underlying variables through the list file.

12.3 BDT Memory Access Function

The official BDT tool provided by Telink can also read the contents of specified locations in Flash, memory and other spaces, and write data through the Memory Access function. It is necessary to make sure that the SWS is on before reading. For details, please refer to the Debug chapter of the BDT User Guide.

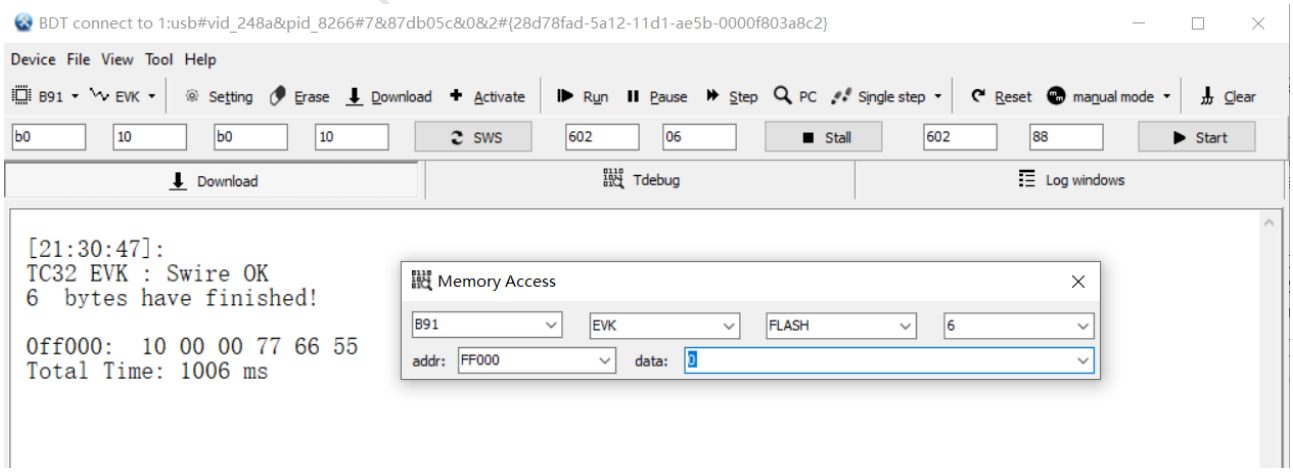


Figure 12.3: BDT memory access function

12.4 BDT Reads PC Pointer

The official BDT tool provided by Telink can be used to read the PC (Program Counter) pointer of B85m, which is very helpful when analyzing the dead problem. For details, please refer to the Debug chapter of the BDT User Guide.

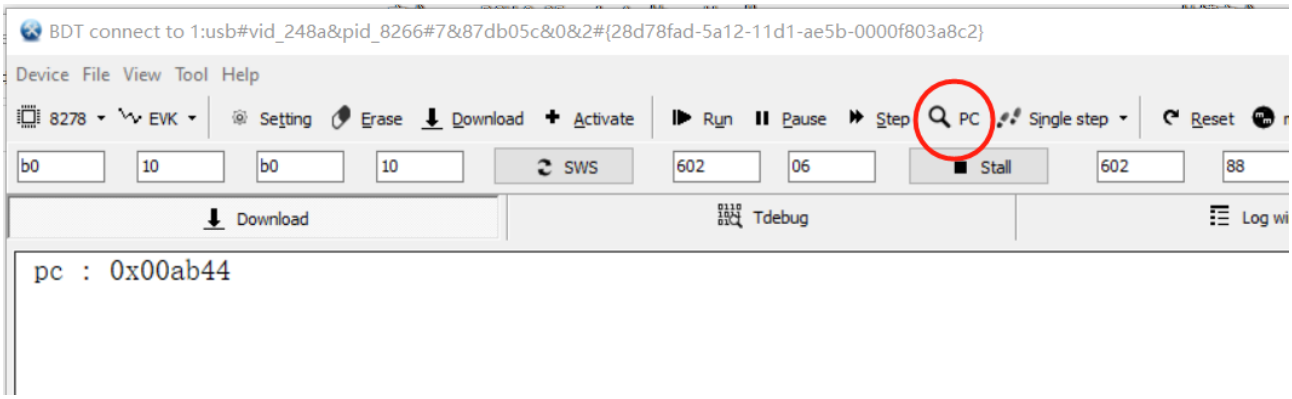


Figure 12.4: BDT reads PC pointer

12.5 Debug IO

In the app_config.h of each routine has an IO-related definition enclosed in the macro DEBUG_GPIO_ENABLE.

```
#if(DEBUG_GPIO_ENABLE)
    #define GPIO_CHN0                       GPIO_PE1
    #define GPIO_CHN1                       GPIO_PE2
    #define GPIO_CHN2                       GPIO_PA0
    #define GPIO_CHN3                       GPIO_PA4
    #define GPIO_CHN4                       GPIO_PA3
    #define GPIO_CHN5                       GPIO_PB0
    #define GPIO_CHN6                       GPIO_PB2
    #define GPIO_CHN7                       GPIO_PE0

    #define GPIO_CHN8                       GPIO_PA2
    #define GPIO_CHN9                       GPIO_PA1
    #define GPIO_CHN10                      GPIO_PB1
    #define GPIO_CHN11                      GPIO_PB3
    #define GPIO_CHN12                      GPIO_PC7
    #define GPIO_CHN13                      GPIO_PC6
    #define GPIO_CHN14                      GPIO_PC5
    #define GPIO_CHN15                      GPIO_PC4


    #define PE1_OUTPUT_ENABLE               1
```

Figure 12.5: Debug IO definition

This function is not enabled by default, it is a unified Debug IO definition for Telink engineers to use internally to grab the waveform of IO for debugging by logic analyzer or oscilloscope. However, users can add their

own Debug IO at the application layer in a similar way, refer to common/default_config.h.

Note:

- In the official release SDK, the Debug IO debugging information in the Stack is included in the library file in a disabled state. So even if the user defines DEBUG_GPIO_ENABLE as 1 at the application layer, it will not enable the Debug IO debugging information in the Stack.
- If Debug IO is enabled, although the Debug IO in the Stack will not work, the Debug IO in the application layer will work, such as rf_irq_handler represented by CHN14 and stimer_irq_handler represented by CHN15 in Telink B91 BLE Multiple Connection SDK.

12.6 USB my_dump_str_data

There are several calls of my_dump_str_data API in the Telink BLE Multiple Connection SDK. This function is an implementation of B91 to output debug information through the USB interface. It is not an officially recommended method for outputting debugging information, and is only for reference. The purpose of this function is to solve the problem of not being able to output through GPIO simulating UART in interrupt. This function is not enabled by default, you need to define DUMP_STR_EN as 1 to enable it.

13 Appendix

13.1 Appendix 1：crc16 algorithm

```
unsigned shortcrc16 (unsigned char *pD, int len)
{
   static unsigned short poly[2]={0, 0xa001};
   unsigned short crc = 0xffff;
   unsigned char ds;
   int i,j;

   for(j=len; j>0; j--)
   {
    unsigned char ds = *pD++;
    for(i=0; i<8; i++)
     {
        crc = (crc >> 1) ^ poly[(crc ^ ds ) & 1];
        ds = ds >> 1;
     }
   }

  return crc;
}
```