Application Note : Telink TLSR826x BLE SDK Developer Handbook

AN-17092700-E4

Ver1.3.0

2019/10/14

Brief:

This document is the guide for TLSR826x BLE SDK 3.2.0 which supports 8261, 8266, 8267 and 8269.

TELINK SEMICONDUCTOR

Published by

Telink Semiconductor

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

Legal Disclaimer

Telink Semiconductor reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Telink Semiconductor disclaims any and allliability for any errors, inaccuracies or incompleteness contained hereinor in any other disclosure relating to any product.

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

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

Information:

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

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

telinkcnsales@telink-semi.com

telinkcnsupport@telink-semi.com

Change Log

Version	Main Changes	Date	Authors
1.0.0	Initial Release	2017/11	Wangsihui, Gaoqiu, Libiao, Chenqiuwei, Cynthia
1.1.0	Updated section 5.1 Audio initialization, 6.1.3 Modify Flash storage architecture, 6.2.4.1 Modify firmware size and OTA FW storage address, 6.3.4.1 Modify firmware size and OTA FW storage address.	2018/12	LX, WSH, Cynthia
1.3.0	3.2.6.1 Add rx overflow description 3.2.8.17 Add BLT_EV_FLAG_LL_REJECT_IND description 3.2.8.18 Add BLT_EV_FLAG_RX_DATA_ABANDOM description 3.2.8.19 Add BLT_EV_FLAG_PHY_UPDATE description 3.2.9.23 Add blc_ll_set_CustomedAdvScanAccessCode( ) description 3.2.10 Add 2M PHY Supported description 3.2.11 Add Data Length Extension description 3. 3.2.1 Add bls_l2cap_setMinimalUpdateReqSendingTime_after_connCreate() description 4.5.3 Add cpu_sleep_wakeup2( ) description	2019/10	Chen Qiuwei

Table of contents

Table of figures

1 SDK Overview

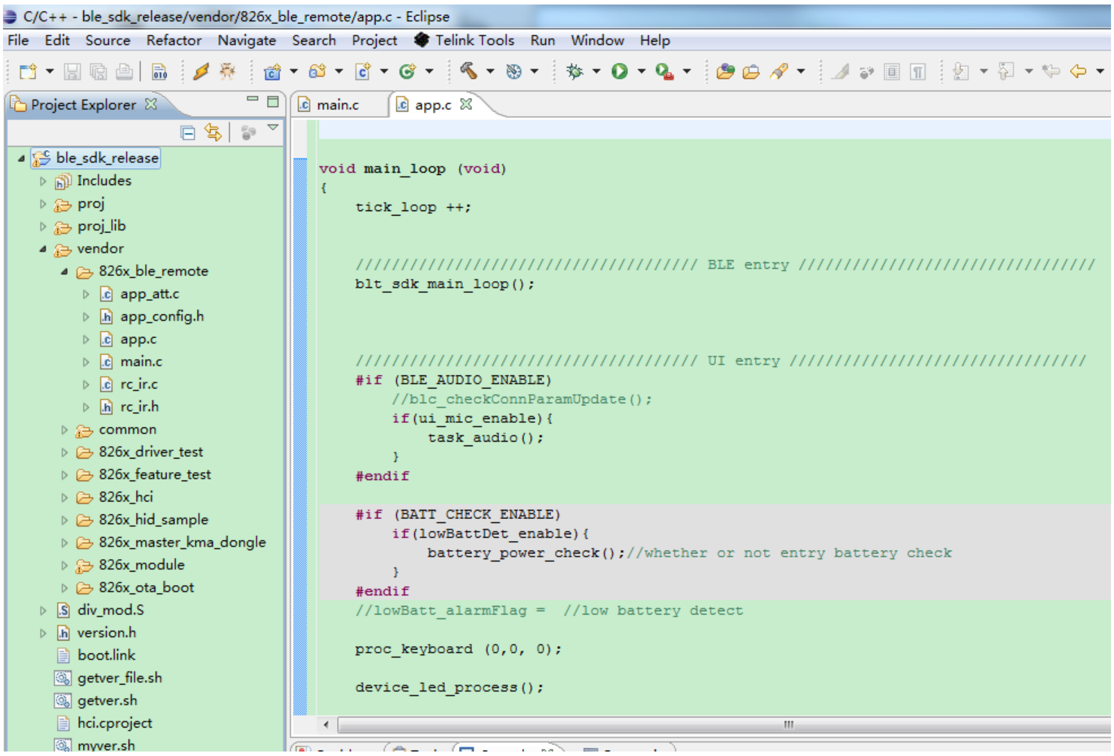
Telink 826x BLE SDK supplies demo code for BLE Slave/Master development, based on which user can develop his own application program.

Currently the SDK applies to four Telink ICs: 8261/8266/8267/8269.

1.1 Software architecture

Software architecture for Telink 826x BLE SDK includes APP layer and BLE protocol stack.

Figure 1-1 shows the file structure after the SDK project is imported in Telink IDE, which mainly contains three top-layer folders including “proj”, “proj_lib” and “vendor”.



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


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



    ////////////////////////////////////// UI entry ////////////////////////////////////
    #if (BLE_AUDIO_ENABLE)
        //blc_checkConnParamUpdate();
        if(ui_mic_enable){
            task_audio();
        }
    #endif

    #if (BATT_CHECK_ENABLE)
        if(lowBattDet_enable){
            battery_power_check();//whether or not entry battery check
        }
    #endif
    //lowBatt_alarmFlag =  //low battery detect

    proc_keyboard (0,0, 0);

    device_led_process();
```

Figure 1-1 SDK file structure

- ✧ proj: This folder contains MCU related peripheral driver, such as flash, I2C, USB, GPIO, UART driver, and etc.
- ✧ proj_lib: This folder contains library files necessary for MCU running, e.g. BLE stack, RF driver, PM driver. Since this folder is supplied in the form of library files (e.g. liblt_8267.a, liblt_8261.a), the source files are not open to users.
- ✧ vendor: This folder contains user APP-layer code, e.g. 826x_ble_remote demo

application. The following four basic files are needed for each new user folder.

1.1.1 main.c

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

```
int main (void) {

    blc_pm_select_internal_32k_crystal(); //select internal 32k rc as 32k counter clock source

    cpu_wakeup_init(CRYSTAL_TYPE);//Basic MCU hardware initialization, negligible to user

    clock_init();          // Clock initialization, user only needs to configure related parameters in app_config.h

    set_tick_per_us(CLOCK_SYS_CLOCK_HZ/1000000);// Clock initialization

    gpio_init();           // GPIO initialization, user only needs to configure related parameters in app_config.h

    rf_drv_init(CRYSTAL_TYPE);// RF initialization, negligible to user

    user_init ();     // BLE initialization, initialization of the whole system, configured by user

    irq_enable();          // Enable global interrupt

    while (1) {

        #if (MODULE_WATCHDOG_ENABLE)
                wd_clear(); //clear watch dog

        #endif

        main_loop ();  // include BLE Rx/Tx processing, power management and user tasks

    }

}
```

1.1.2 app_config.h

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

1.1.3 application file

“app.c”: User file for system initialization and adding user task UI.

"app_att.c" of BLE Slave project: configuration files for services and profiles. Based on Telink Attribute structure, as well as Attributes such as GATT, standard HID, proprietary OTA and MIC, user can add his own services and profiles as needed.

1.1.4 BLE stack entry

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

1) BLE related interrupt processing entry in "irq_handler" function of "main.c" file "irq_blt_sdk_handler".

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

2) BLE logic and data processing function entry in application file mainloop "blt_sdk_main_loop".

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

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

    ////////////////////// UI entry ////////////////////////////
    ……

}
```

1.2 Applied ICs

The four applied ICs can be divided into two categories:

- ✧ 8261/8267/8269: The three ICs share the same IP core, thus their hardware modules are almost the same except in audio, Flash size and SRAM size.
- ✧ 8266: This IC has IP core and hardware modules differenet from 8261/8267/8269.

IC	IP core	Audio	Flash size	SRAM size
8266	IP1	×	512 kB	16 kB

IC	IP core	Audio	Flash size	SRAM size
8261	IP2	×	128 kB	16 kB
8267		√	512 kB	16 kB
8269		√	512 kB	32 kB

Following sections will introduce characteristics of SDK file structure corresponding to IC difference.

1.3 Driver

In SDK, drivers can be divided into two categories: 8266, 8267.

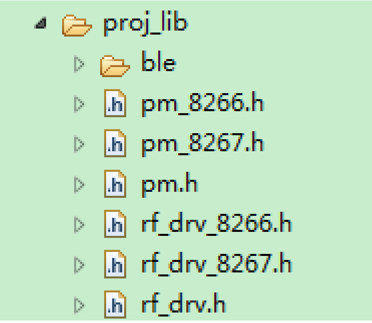
Since 8261/8269 share the IP core of 8267, drivers of 8267 also applies to 8261/8269 application.

E.g. for an 8269 ble remote application based on 8269, to use drivers such as rf/gpio driver, user should find corresponding rf_drv_8267/gpio_8267 files.

In SDK, most drivers are supplied in source code (except for rf and pm driver), and these files are mainly available from the following three locations:

1) proj_lib

As shown below, source code of pm_8266 and pm_8267 are not open to user, while related interfaces are assembled in lib. Only head files are supplied for user reference.

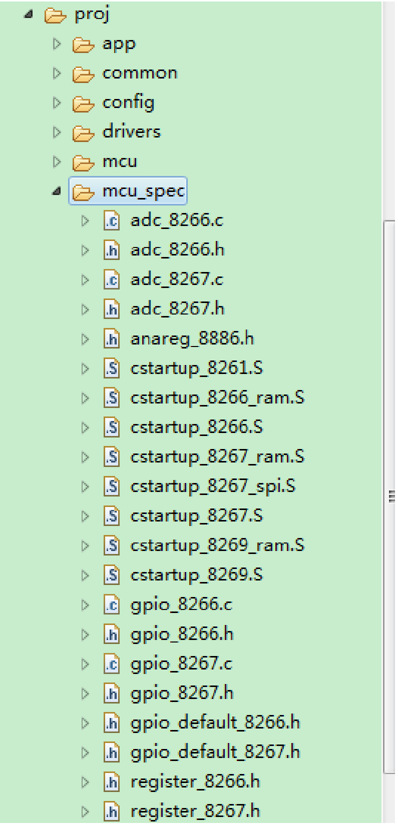


```
proj_lib
    ble
    pm_8266.h
    pm_8267.h
    pm.h
    rf_drv_8266.h
    rf_drv_8267.h
    rf_drv.h
```

Figure1-2 pm and rf driver

2) proj/mcu_spec

For drivers which have great difference in 8267 and 8266, e.g. adc and gpio driver, corresponding source files and head files (adc/gpio/register files as shown below) are placed under the “proj/mcu_spec” folder.

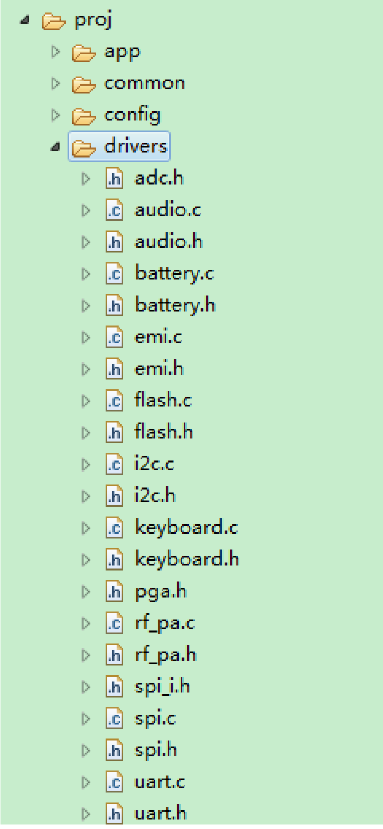


```
proj
  app
  common
  config
  drivers
  mcu
  mcu_spec
    adc_8266.c
    adc_8266.h
    adc_8267.c
    adc_8267.h
    anareg_8886.h
    cstartup_8261.S
    cstartup_8266_ram.S
    cstartup_8266.S
    cstartup_8267_ram.S
    cstartup_8267_spi.S
    cstartup_8267.S
    cstartup_8269_ram.S
    cstartup_8269.S
    gpio_8266.c
    gpio_8266.h
    gpio_8267.c
    gpio_8267.h
    gpio_default_8266.h
    gpio_default_8267.h
    register_8266.h
    register_8267.h
```

Figure1-3 adc/gpio driver and bootloader

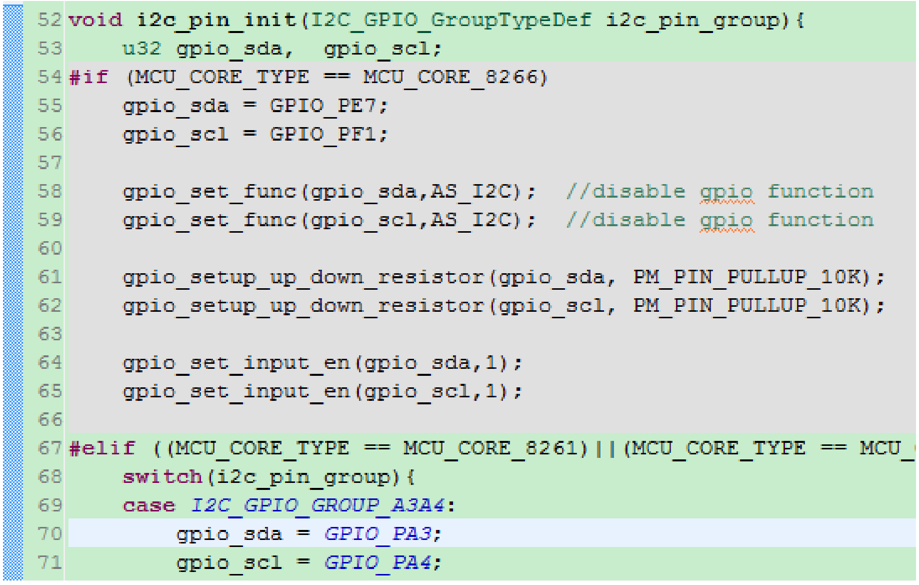
3) proj/drivers

For drivers which have slight difference in 8267 and 8266, e.g. uart/i2c driver, corresponding files are unified under the “proj/driver” folder, while differences are processed by using “MCU_CORE_TYPE” in the source files.



- proj
  - app
  - common
  - config
  - drivers
    - adc.h
    - audio.c
    - audio.h
    - battery.c
    - battery.h
    - emi.c
    - emi.h
    - flash.c
    - flash.h
    - i2c.c
    - i2c.h
    - keyboard.c
    - keyboard.h
    - pga.h
    - rf_pa.c
    - rf_pa.h
    - spi_i.h
    - spi.c
    - spi.h
    - uart.c
    - uart.h

Figure1-4 i2c/spi/uart driver



```
void i2c_pin_init(I2C_GPIO_GroupTypeDef i2c_pin_group){
    u32 gpio_sda,  gpio_scl;
#if (MCU_CORE_TYPE == MCU_CORE_8266)
    gpio_sda = GPIO_PE7;
    gpio_scl = GPIO_PF1;

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

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

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

#elif ((MCU_CORE_TYPE == MCU_CORE_8261)||(MCU_CORE_TYPE == MCU_
    switch(i2c_pin_group){
    case I2C_GPIO_GROUP_A3A4:
        gpio_sda = GPIO_PA3;
        gpio_scl = GPIO_PA4;
```

Figure1-5 Difference in i2c driver

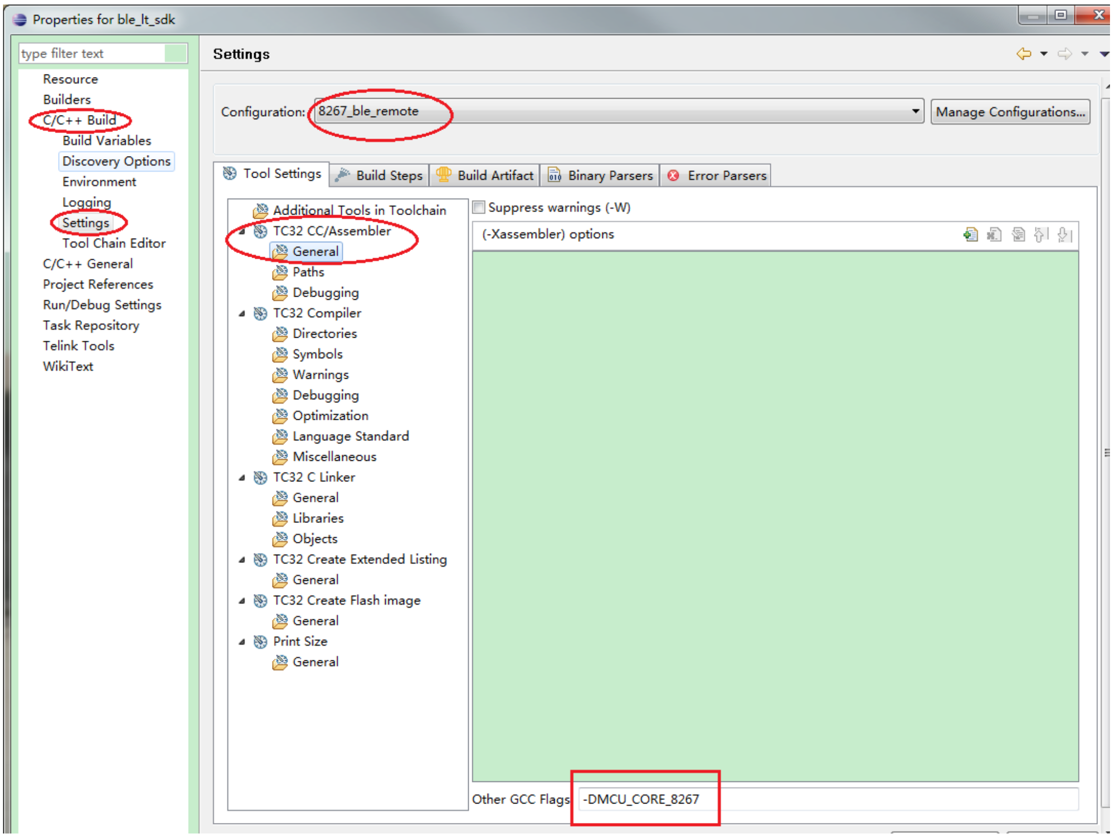
1.4 bootloader

Each of the four ICs has its own bootloader: cstartup_8261.S / cstartup_8266.S / cstartup_8267.S / cstartup_8269.S. Please refer to Figure1-3 for file location in SDK.

Take “cstartup_8267.S” for example: The first sentence “#ifdef MCU_CORE_8267” indicates this bootloader will take effect only when the “MCU_CORE_8267” is defined by user.

Therefore, user can correspondingly define MCU_CORE_8261 / MCU_CORE_8266 / MCU_CORE_8267 / MCU_CORE_8269 to enable compiler to automatically select the correct bootloader.

The “8267_ble_remote” is taken as an example to illustrate how to select 8267 bootloader. As shown below, user can define MCU_CORE_8267 (-DMCU_CORE_8267) in the 8267_ble_remote.



Configuration: 8267_ble_remote

□ Suppress warnings (-W)

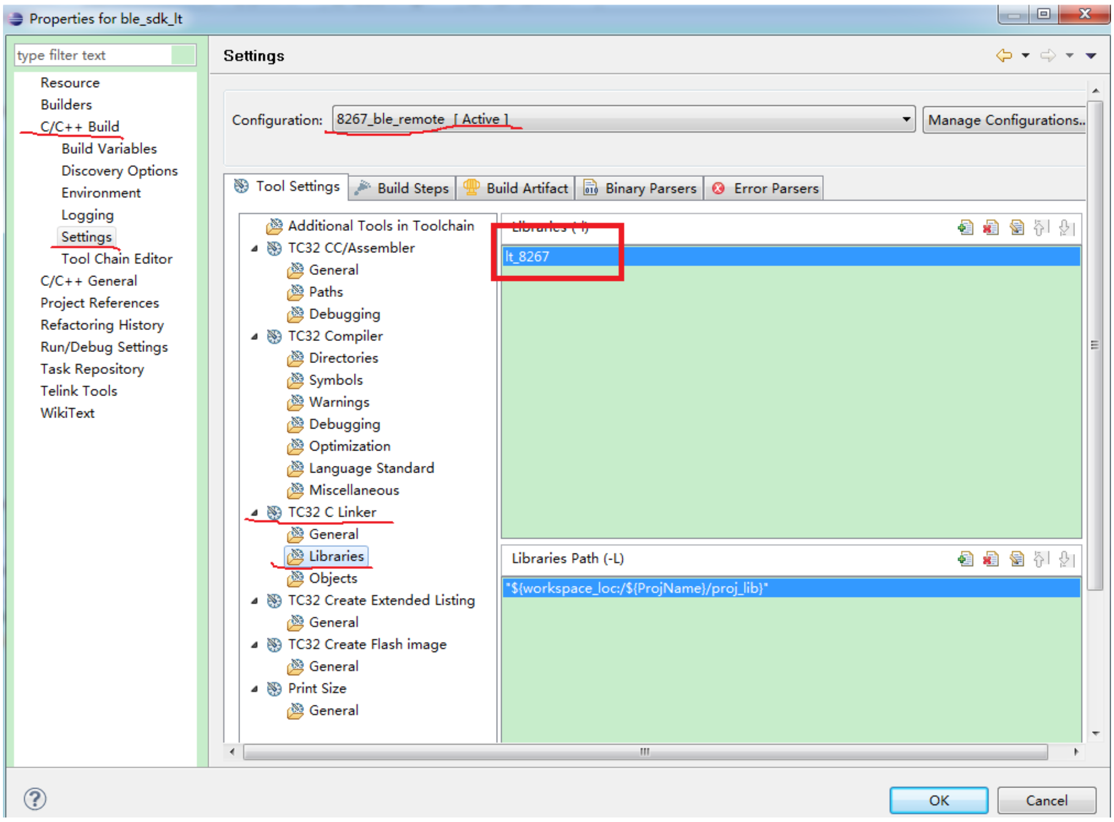
(-Xassembler) options

Other GCC Flags: -DMCU_CORE_8267

Figure1-6 Select bootloader

1.5 library

The figure below shows how to select library corresponding to project.



Configuration: 8267_ble_remote [Active]

Libraries (-l)

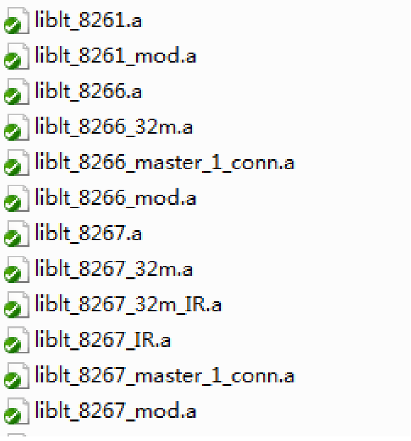
lt_8267

Libraries Path (-L)

"${workspace_loc:/${ProjName}/proj_lib}"

Figure1-7 Select library

Figure1-8 shows libraries currently supplied in SDK 3.2.0. Not all libraries all released in SDK, however, if user has special requirement (e.g. liblt_8267_48m.a) in actual development, he can apply as needed, and will be provided with the the library after evaluation and approval by Telink.



liblt_8261.a
liblt_8261_mod.a
liblt_8266.a
liblt_8266_32m.a
liblt_8266_master_1_conn.a
liblt_8266_mod.a
liblt_8267.a
liblt_8267_32m.a
liblt_8267_32m_IR.a
liblt_8267_IR.a
liblt_8267_master_1_conn.a
liblt_8267_mod.a

Figure1-8 Libraries supplied in BLE SDK

Following sub-sections show features of libraries in Telink BLE SDK.

1.5.1 Category based on IC

According to IC, libraries can be divided into three categories: 8261 relative/8266 relative/8267 relative.

8269 libraries are not supplied independently, and user can directly use 8267 libraries, since:

1) As shown in the IC difference table, the sole difference between 8269 and 8267 is SRAM size: 32kB (8269), 16kB (8267).
2) SRAM configuration of 8267/8269 is implemented in corresponding bootloader (cstartup_8267.S / cstartup_8269.S supplied in source code), user only needs to select his bootloader in setting of project.

1.5.2 Category based on function

Currently SDK supplies libraries corresponding to three basic BLE functions.

1) BLE Slave, Telink 826x acts as Host MCU. Corresponding libraries include:

   liblt_8261/liblt_8261_32m and etc.; liblt_8266/liblt_8266_32m and etc.; liblt_8267/liblt_8267_32m and etc.;

2) BLE Slave, Telink 826x acts as BLE SPP module and communicates with Host MCU via interface such as UART/SPI. Corresponding libraries include:

   liblt_8261_mod/ liblt_8261_mod_32m and etc.; liblt_8266_mod/liblt_8266_mod_32m and etc.; liblt_8267_mod/liblt_8267_mod_32m and etc.;

3) BLE Master, only single connection can be established, Telink 826x acts as Host MCU. Corresponding libraries include: liblt_8266_master_1_conn and liblt_8267_master_1_conn. Note that 8261 libraries are not released independently, thus if 8261 is used to develop BLE single connection master, user can select “liblt_8267_master_1_conn”.

The library category below will be released in following SDK versions:

BLE Master, support multi connection, Telink 826x acts as Host MCU. Corresponding libraries include: liblt_8266_master_n_conn and liblt_8267_master_n_conn.

1.5.3 Category based on system clock

For BLE Master, libraries won't differ according to system clock of application program.

However, for BLE Slave, library name will indicate libray of corresponding system clock, as shown below:

- ✧ By default, library name is not marked with clock, which indicates library corresponding to 16m system clock. For example, "liblt_8266.a" indicates 8266 library corresponding to 16m system clock.
- ✧ Library name marked with 32m/48m (not 16m) indicates library corresponding to 32m/48m system clock. For example, "liblt_8267_32m.a" indicates 8267/8269 library corresponding to 32m system clock.

For BLE master single connection, regardless of 16m, 32m or 48m system clock is used, 8266 will use unified library "liblt_8266_master_1_conn", while 8261/8267/8269 will use unified library "liblt_8267_master_1_conn".

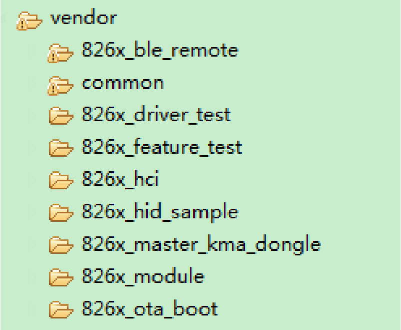
1.5.4 Other special libraries

Considering actual application cases, SDK also provides some special libraries, e.g. "liblt_8267_IR.a" and "liblt_8267_32m_IR.a" for 8267/8269 IR application corresponding to 16m and 32m system clock.

Note: All IR applications need specific library. To develop 8261/8266 IR application, user needs to contact Telink and apply for corresponding library.

1.6 Demo

Telink BLE SDK supplies multiple BLE demos for user. Each demo code corresponding to specific hardware, based on which user can run demo, observe effect and modify demo code for his own application development.



```
vendor
    826x_ble_remote
    common
    826x_driver_test
    826x_feature_test
    826x_hci
    826x_hid_sample
    826x_master_kma_dongle
    826x_module
    826x_ota_boot
```

Figure1-9 Demo code supplied in BLE SDK

1.6.1 BLE Slave demo

BLE Slave demo and differences are listed as in the table below:

Demo	Stack	Application	MCU function
826x hci	BLE controller	No	Controller, communicate with MCU Host via HCI interface
826x module	BLE controller + host	Application in Host MCU	BLE SPP module
826x remote	BLE controller + host	Remote control application	Host MCU
826x hid sample	BLE controller + host	Simple Slave demo with V+/V- volume control only	Host MCU

826x hci is a BLE Slave controller. It supplies USB/UART-based HCI to communicate with MCU Host and forms a complete BLE Slave system.

826x remote/826x module are complete BLE Slave stack.

✧ 826x module only acts as BLE SPP module to communicate with Host MCU via UART interface. Generally application code is written in Host MCU.

✧ 826x remote is a demo of BLE remote controller which supports basic remote control function. It can connect with standard iOS/Android device or Telink 826x master kma dongle to control the peer.

826x hid sample is simplified demo based on 826x remote, which should run on Telink 8266/8267 dongle. The dongle can pair and connect with standard iOS/Android device, and the two buttons on the dongle simulate Vol+/Vol-. This demo can run on 826x dongle, thus user can save demo hardware cost for debugging and development.

826x ota boot is a code section of OTA necessary for all 8261/8266 BLE Slave projects, while 8267/8269 does not need ota boot. Please refer to OTA section for details.

1.6.2 BLE master demo

826x master kma dongle is a demo of BLE Master single connection. It can connect and communicate with 826x hid sample/826x remote/826x module.

Libraries corresponding to 826x remote/826x hid sample supply standard BLE stack, including BLE controller + BLE host. User only needs to add his own application code in APP layer by using APIs of controller and Host, and does not need to process BLE Host.

Libraries corresponding to 826x master kma dongle only provide standard BLE controller function, and does not provide standard Host. 826x master kma dongle demo code gives BLE Host implementation in APP layer for reference, including ATT, simple SDP (service discovery protocol), the most common SMP (security management protocol), and etc.

The most complex function of BLE Master is service discovery of Slave server and recognition of all services, which generally can be implemented in Android/linux system. Limited by Flash size and SRAM size, Telink 826x IC cannot supply complete service discovery. However, SDK supplies all ATT interfaces needed for service discovery. Based on service discovery process of 826x remote by 826x master kma dongle, user can implement traversal of specific services.

1.6.3 Feature demo and driver demo

826x feature test gives demo code for some common features related to BLE. User can implement his own functions based on these demos. All features will be introduced in BLE section.

826x driver test gives sample code for basic drivers, based on which user can implement his own driver functions. The Driver section will introduce various drivers in detail.

2 MCU Basic Modules

2.1 MCU address space

2.1.1 MCU address space allocation

Telink 826x MCU supports maximum addressing space of 16M bytes, including 8M-byte program space from 0 to 0x7fffff and 8M-byte peripheral space (e.g. SRAM, register space) from 0x800000 to 0xffffff.

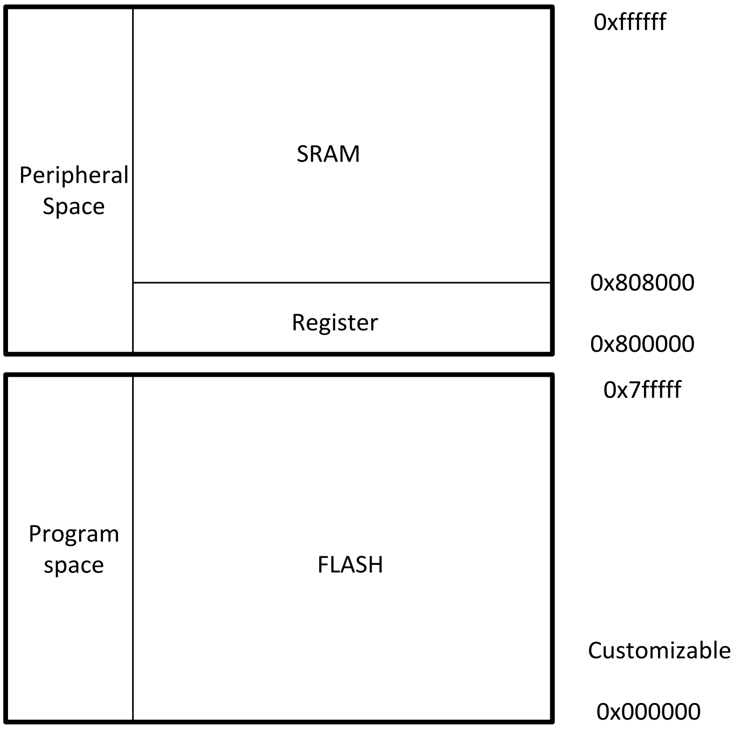


Figure2-1 MCU address space allocation

During physical addressing of 826x MCU, address line BIT (23) serves to differentiate program space / peripheral space:

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

When addressing space is peripheral space (BIT(23) is 1), address line BIT (15) serves to differentiate Register / SRAM.

- ✧ Address line BIT (15) is 0: access Register
- ✧ Address line BIT (15) is 1: access SRAM.

2.1.2 SRAM space ram allocation

2.1.2.1 SRAM and Firmware spcae

For 16kB SRAM, address space range is 0x808000 ~ 0x80C000; while for 32kB SRAM, address space range is 0x808000 ~ 0x810000.

The figure below shows SRAM and Firmware space allocation in 16kB SRAM.

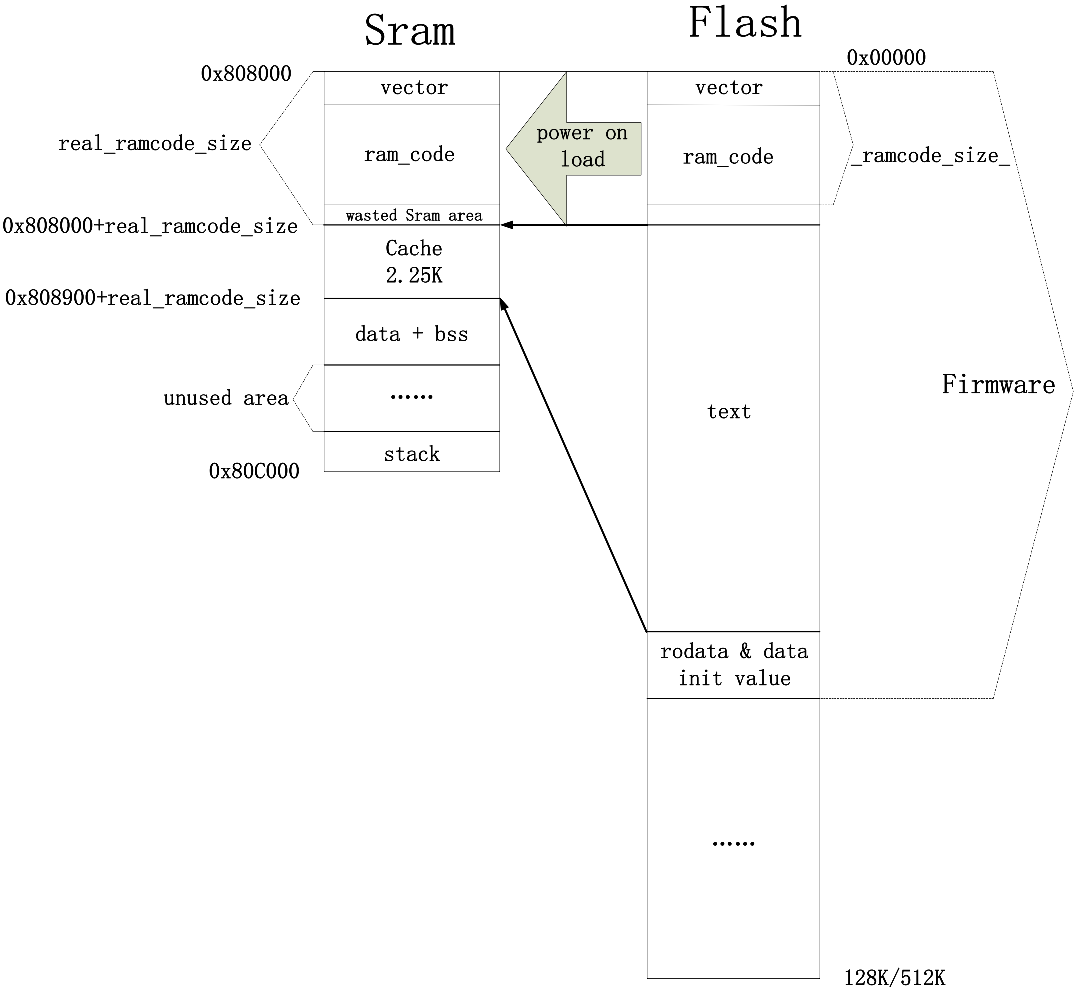


Figure2-2 SRAM and Firmware space

In SDK, files related to SRAM space allocation include “boot.link” and bootloader S file corresponding to IC (e.g. “cstartup_8267.S”).

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

1) vectors, ram_code

Vectors is a code section of Flash Firmware (executable bin file generated by program compiling in SDK), and it corresponds to assembling file “cstartup_826x.S” (i.e. startup code bootloader).

Ramcode is memory resident code in Flash Firmware, and it corresponds to all functions with keyword “_attribute_ram_code_” (e.g. flash erease function).

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

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

- ✧ Some functions (e.g. Flash operation functions) involve timing multiplex with four Flash MSPI pins: If these functions are placed in Flash, it will cause timing conflict and system crash.
- ✧ Whenever functions resident in RAM are invoked, it isn’t needed to re-read them from Flash, thus time will be saved. Therefore, the functions with limited execution time should be memory resident to increase execution efficiency. In SDK, some functions related to BLE timing sequence need frequent execution, in order to decrease execution time and save power consumption, these functions are memory resident.

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

It’s needed to load the vector and ramcode in firmware to RAM when power on. After compiling, the total size of the two parts is “_ramcode_size_”, which is a variable recognizable by compiler. Its calculation is implemented in “boot.link”. As shown below, the compiling result “_ramcode_size_” equal the size of all code including vector and ramcode.

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

2) Cache

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

Memory resident code can be directly read and executed from memory, however, only a small part of firmware is memory resident code, and the majority are still in Flash. According to program locality principle, a part of Flash code can be stored in the Cache. Thus, if the code to be executed is in the Cache, instructions can be directly read and executed from the Cache; otherwise it's needed to read code from Flash to replace the old code in the Cache, then read and execute instructions from the Cache.

As shown in Figure2-2, the "text" in firmware is Flash code not placed in SRAM. According to program locality principle, it's needed to load this part to the Cache so that it can be executed.

Though Cache size is fixed as 2.25K, its starting address in SRAM is configurable. To ensure enough space to store vector and ramcode in Flash, this starting address must exceed "0x808000+_ramcode_size_". As specified by 826x MCU hardware, Cache starting address must be 256-byte aligned, therefore, the "real_ramcode_size" is the 256-byte aligned size of "_ramcode_size_", Cache starting address should be:

0x808000 + real_ramcode_size
= 0x808000 + ((_ramcode_size_+255)/256)* 256
= 0x808000 + _ramcode_size_div_256_* 0x100

Cache starting address is 256-byte aligned "0x808000 + _ramcode_size_div_256_ * 0x100", while the "_ramcode_size_" is not 256-byte aligned generally. The actual size of the code loaded from Flash to RAM when power on is "_ramcode_size_div_256_* 256", which means a part of space in SRAM is wasted area.

For example: Suppose "_ramcode_size_" is 0x780, and the size of code loaded to SRAM is 0x800, then the code of 0x00000 ~ 0x007ff in Flash firmware is memory resident, the 128 bytes of 0x808780 ~0x8087ff in SRAM is wasted area, i.e. non-ramcode is memory redident in SRAM.

If "_ramcode_size_" is 0x701, 255 bytes will be wasted; if "_ramcode_size_" is 0x800, the size of wasted SRAM area is 0. The maximum size of wasted SRAM area is 255 bytes, therefore, during program design, user needs to check list file to view ramcode occupation, and try to avoid large wasted area.

Since Cache size is fixed as 2.25K, Cache ending address should be:

0x808000 + real_ramcode_size + 0x900
= 0x808900 + real_ramcode_size

3) data / bss

"data" in SRAM serves to store initialized global variables of program (i.e. global variables which are non-zero initially). The initial value of the global variables in "data" is "data init value" in firmware, as shown in Figure2-2.

"bss" in SRAM serves to store global variables of program not initialized (i.e. global variables which are zero initially).

Cache is followed by "data", while "data" is followed by "bss". The starting address of "data + bss" is Cache ending address, i.e. "0x808900 + _ramcode_size_div_256_* 0x100".

Following shows the code in "boot.link" which directly defines the starting address of "data".

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

4) stack / unused area

"stack" in SRAM starts from 0x80C000 (default 16kB SRAM) / 0x810000 (32kB SRAM), which is the lowest address. Its SP pointer will descend during push operation, and ascend during pop operation.

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

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

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

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

5) text

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

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

6) rodata/data init value

The remaining data except for "vector", "ram_code" and "text" in firmware are "rodata" and "data initial value".

"rodata" is read-only data in firmware, i.e. variable with keyword "const". E.g. ATT table in Slave:

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

User can view the "my_Attributes" is within the "rodata" by checking corresponding list file.

As introduced above, "data" is initialized global variables in firmware, e.g.

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

The compiler will store the initial value "0x1234" in "data initial value". When the bootloader (cstartup_826x.S) is executed, this initial value will be copied to memory address corresponding to "testValue".

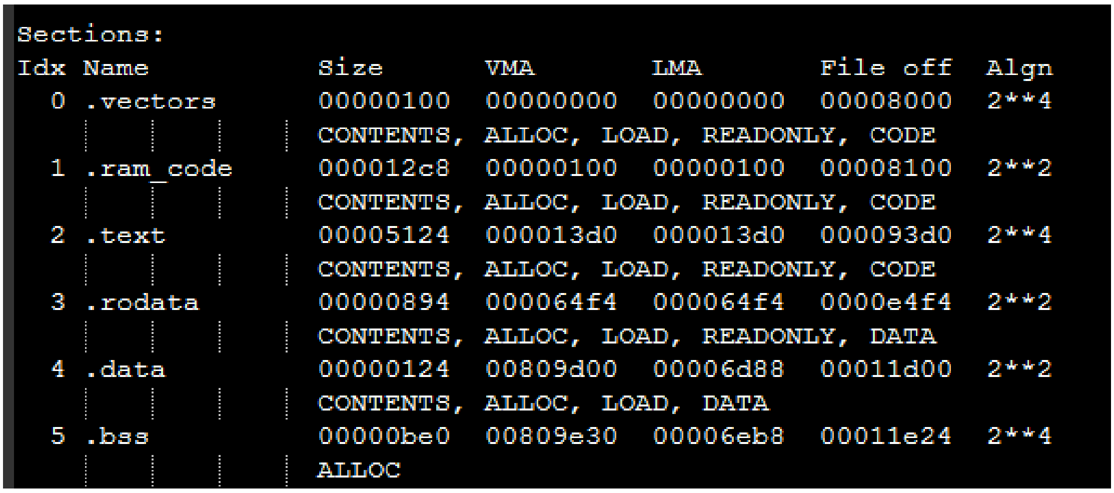
2.1.2.2 List file analysis demo

A simple BLE Slave demo "8267 hid sample" is taken as an example to illustrate SRAM and Flash address space allocation (please refer to Figure2-2 SRAM and Firmware space). Based on this demo, user can analyze SRAM and Flash space allocation of his own program.

Bin file and list file of this demo is available under the directory “SDK” -> “Demo” -> “list file analyze”. Information of SRAM space allocation can be analyzed from the “8267_hid_sample.list” file.

All screeshots herein are available from the files including “boot.link”, “cstartup_8267.S”, “8267_hid_sample.bin” and “8267_hid_sample.list”.

In the list file, each code part of a specific function is called a “section”. The figure below shows section distribution in the list file “8267_hid_sample.list”.



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

Figure2-3 Section distribution in list file

Following lists the sections in the list file.

1) vectors: start from Flash 0, size is 0x100.

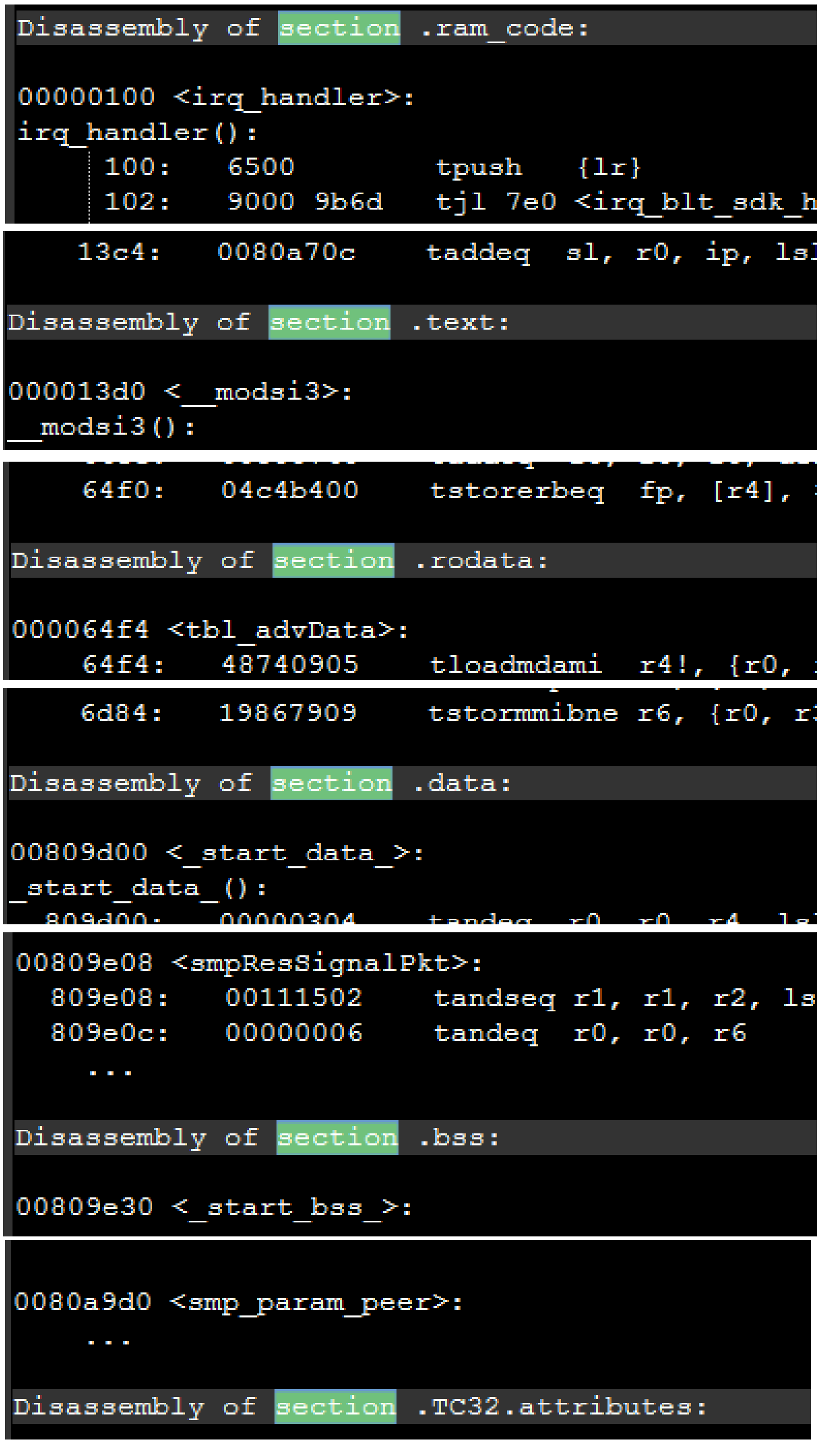
2) ram_code: start from Flash 0x100, size is 0x12c8.

3) text: start from Flash 0x13d0, size is 0x5124.

4) rodata: start from Flash 0x64f4, size is 0x894.

5) data: start from SRAM 0x809d00, size is 0x124.

6) bss: start from SRAM 0x809e30, size is 0xbe0. By calculation, “bss” ending address is 0x809e30 + 0xbe0 = 0x80aa10. The remaining space size following the “bss” is 0x80c000 – 0x80aa10 = 0x15F0 = 5616 bytes. Except for the 256 bytes for stack, the remaining 5360 bytes are unused.



```
Disassembly of section .ram_code:

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

Disassembly of section .text:

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

Disassembly of section .rodata:

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

Disassembly of section .data:

00809d00 <_start_data_>:
_start_data_():
  809d00:     00000304     tandeq  r0, r0, r4, lsl
00809e08 <smpResSignalPkt>:
  809e08:     00111502     tandseq r1, r1, r2, ls
  809e0c:     00000006     tandeq  r0, r0, r6
     ...

Disassembly of section .bss:

00809e30 <_start_bss_>:

0080a9d0 <smp_param_peer>:
     ...

Disassembly of section .TC32.attributes:
```

Figure2-4 Section address in list file

The figure above shows starting/ending address of various sections by searching "section" in the list file.

From Figure2-3 Section distribution in list file and Figure2-4 Section address in list file, the analysis is shown as below:

1) **vector**

"vector" is bootloader corresponding to "startup_826x.S" assembly file. As shown in the list file, this section contains 256 bytes (size) from 0 (starting address) to 0xff (ending address). After power on load to SRAM, the corresponding address in SRAM is 0x808000 ~ 0x8080ff.

2) **ram_code**

"ram_code" section contains 0x12c8 bytes (size) from 0x100 (starting address) to 0x13c8 (ending address). Since "_ramcode_size_" is 0x13c8, 256-byte aligned "real_ramcode_size" is 0x1400, there are actually 56 (0x38) bytes in SRAM are wasted.

3) **Cache**

Cache starting and ending address are:

0x808000 + real_ramcode_size ~ 0x808900 + real_ramcode_size

0x809400 ~ 0x809d00

Cache related information are not shown in the list file.

4) **text**

"text" section contains 0x5124 (size= 0x64f4 – 0x13d0) bytes from 0x13d0 (starting address of "text", i.e. ending address of "ram_code") to 0x64f4 (ending address).

5) **rodata**

"rodata" section starts from 0x64f4 ("text" ending address) and ends till 0x6d88.

As shown in the "8267_hid_sample.bin", the actual bin size is 0x6eac. According to analysis above, the remaining firmware space 0x6d88 ~ 0x6eac is actually "data init value", i.e. initial value of initialized global variables in the firmware.

There is not a specific section in the list file corresponding to the "data init value". User can search the keyword "_dstored_" and find the value "0x6d88" which indicates the starting address of the "data init value".

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

Following is the “_dstored_” definition in the “boot.link”. This will tell the compiler that initial value of initialized global variables in the “data” section are all stored in the “_dstored_” of firmware.

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

6) data

“data” section starts from Cache ending address 0x809d00, and its size is shown as 0x124 in Figure2-3 Section distribution in list file.

The final variable in the “data” section is “smpResSignalPkt”, which is a structure variable in SDK bottom layer. This variable starts from 0x809e08, and its size is 28 = 0x1c. Therefore “data” ending address is 0x809e24, and the size of the “data” section is 0x809e24 - 0x809d00 = 0x124.

7) bss

“data” section is followed by “bss”. Since the first array “_start_bss_” should be 16-byte aligned, the “bss” section starts from 0x9e30, and its size is shown as 0xbe0 in Figure2-3 Section distribution in list file.

The final variable in the “bss” section is “smp_param_peer”, which is a structure variable in SDK bottom layer. This variable starts from 0x80a9d0, and its size is 64 = 0x40. Therefore “bss” ending address is 0x80aa10, and the size of the “bss” section is 0x80aa10 - 0x809e30 = 0xbe0.

By calculation, the remaining SRAM space size is 0x80c000 – 0x80aa10 = 0x15F0 = 5616 bytes. Except for the 256 bytes for stack, the remaining 5360 bytes are unused.

2.1.3 MCU address space access

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

2.1.3.1 Peripheral space access

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

```
u8    x = *(volatile u8*)0x800066;    // read register 0x66
 *(volatile u8*)0x800066 = 0x26;    //write register 0x66
u32 y = *(volatile u32*)0x808000;             //read SRAM 0x8000~0x8003
*(volatile u32*)0x808000 = 0x12345678;    //write SRAM 0x8000~0x8003
```

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

Note: For operation such as write_reg8(0x8000) / read_reg16(0x8000), to ensure the access space is Register/SRAM rather than Flash, the base address “0x800000” is automatically added (address line BIT(23) is 1), as shown below.

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

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

```
u16    x = *(volatile u16*)0x808001;         //0x808001 is not 2-byte aligned
*(volatile u32*)0x808005 = 0x12345678;    //0x808005 is not 4-byte aligned
```

The correct formats should be:

```
u16    x = *(volatile u16*)0x808000;         //0x808000 is 2-byte aligned
*(volatile u32*)0x808004 = 0x12345678;    //0x808004 is 4-byte aligned
```

2.1.3.2 Flash space operation

Read/Write access operation of the Flash space is implemented by using the function "flash_read_page"/"flash_write_page". Codes about flash access and erasing operation are available in "proj/drivers/flash.c" and "flash.h".

1) Flash Read/Write access operation

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

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

Flash read operation example via "flash_read_page":

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

Flash write operation example via "flash_write_page":

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

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

✧ If the "addr" is the starting address of one page, the "len" cannot exceed 256.

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

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

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

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

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

2) Flash erase operation

The function “flash_erase_sector” serves to erase flash.

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

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

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

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

3) Influence to system interrupt caused by flash access/erasing operation

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

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

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

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

4) Read flash via pointer

Firmware of 826x BLE SDK is stored in Flash. When the firmware is running, only former part of the code in Flash (memory resident) is stored and executed in RAM, and the majority will be transferred to the high-speed “Cache” of RAM from Flash when needed. MCU will automatically control internal MSPI hardware module to read Flash.

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

Following shows three examples:：

u16 x = *(volatile u16*)0x10000; //read two bytes from flash 0x10000

u8 data[16];

memcpy(data, 0x20000, 16); //read 16 bytes from flash 0x20000 and copy to data

if(!memcmp(data, 0x30000, 16)){ // read 16 bytes from flash 0x30000 and compare with data

//……

}

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

static inline void blc_app_loadCustomizedParameters(void);

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

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

Example:

u8 result;

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

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

The original data in flash 0x40000 is 0xff; the result of the first reading operation is 0xff; then 0x5A is written into flash 0x40000 by the following writing operation; in theory, the result of the second reading operation should be the new value "0x5A", but the actual result is still the old data buffered in the cache, i.e. "0xff". Therefore, in the case of multiple reading of the same address, if its value will be modified, use the API "flash_read_page" rather than pointer, to ensure the result of reading operation is the new value written into this address rather than the old value in the cache.

The following format is correct:

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

2.1.4 SDK FLASH space allocation

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

Two allocation methods of Flash space are supported depending on flash size: one method is for 512kB flash (8266/8267/8269), while the other method is for 128kB flash (8261).

2.1.4.1 Space allocation for 512kB Flash

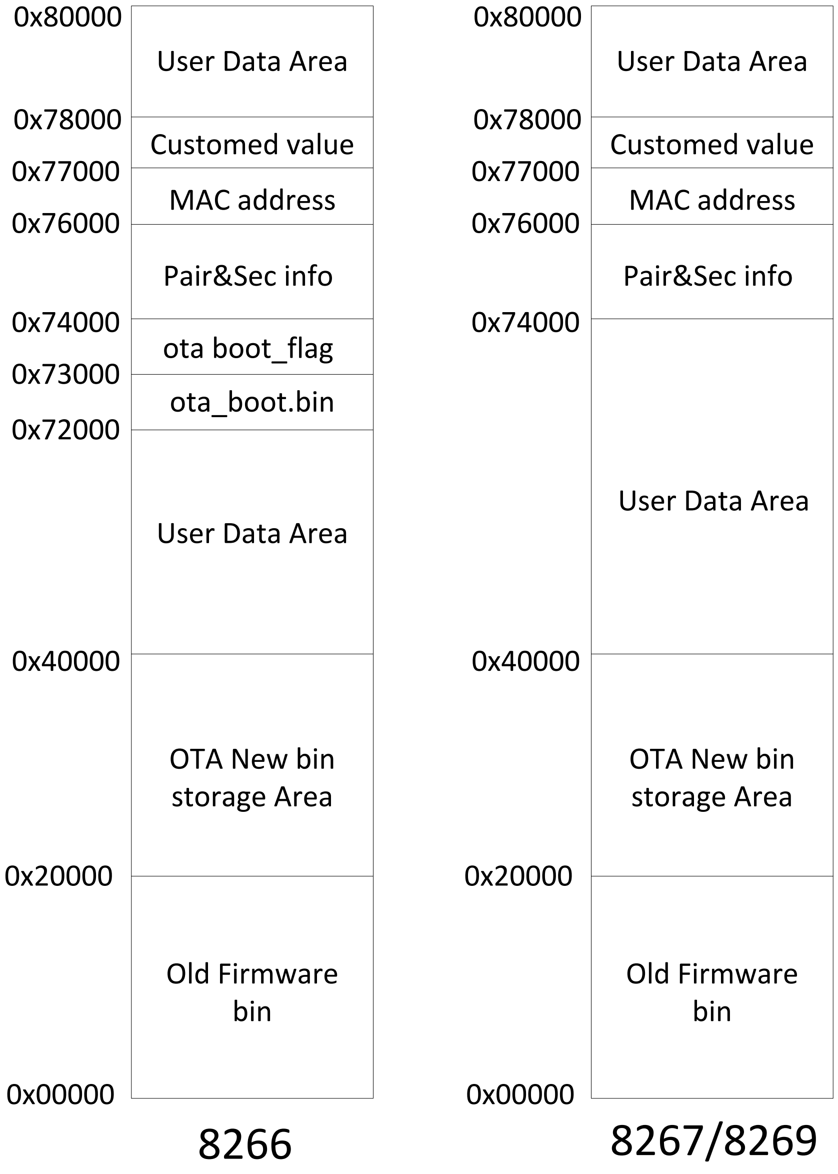


Figure2-5 512kB FLASH address space allocation

The figure above shows the default address allocation for the 512K flash of 8266/8267/8269. Corresponding interfaces are supplied for user to modify flash address allocation.

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

Corresponding to SDK, MAC address of actual product will be downloaded into its flash starting from 0x76000 by Telink jig system. If it's needed to modify this starting address to store MAC address, user should ensure the consistency. The "user_init" function in the SDK will read MAC address from flash area starting from the macro "CFG_ADR_MAC". This macro is modifiable in the "proj_lib/ble/blt_config.h".

```
#ifndef       CFG_ADR_MAC
#define       CFG_ADR_MAC                     0x76000

#endif
```

2. The sector from 0x77000 to 0x77fff serves store customized calibration information for Telink MCU. Only this sector does not follow the rule that puts different information types into different sectors; the 4096 bytes in this sector are divided into 64 units with 64 bytes each, and each unit stores one type of calibration information. Since calibration information are burned to corresponding addresses by jig, they can be stored in the same sector; when firmware is running, these calibration information are read only and they're not allowed to be written or erased.
    1) The first 64-byte unit serves to store frequency offset calibration information. Actually this calibration value is 1 byte stored in 0x77000.
    2) The second 64-byte unit serves to store calibration value of TP value. Actually this calibration information is 2 bytes (TP0, TP1) stored in 0x77040 and 0x77041.
    3) The third 64-byte unit serves to store capacitance calibration value of external 32kHz crystal.
    4) Following units are reserved for other potential calibration values.

Calibration values of actual product will be downloaded into its flash area corresponding to SDK by Telink jig system. If it's needed to modify the starting address to store calibration value, user should ensure the consistency. In the "user_init" function of SDK, the "rf_customized_param_load()" function will read calibration values from flash area starting from the following macros. These macros are modifiable in the "proj_lib/ble/blt_config.h".

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

#ifndef       CUST_TP_INFO_ADDR
```

```
#define         CUST_TP_INFO_ADDR           0x77040
#endif

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

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

   proj_lib/ble/blt_smp_nv.h

   void bls_smp_configParingSecurityInfoStorageAddr (int addr);

4. For 8266, the sector 0x73000 ~ 0x73FFF is occupied by BLE stack system, and it serves to store the ota boot_flag for OTA firmware upgrade. User can follow the instructions in **section 6.2.4.3** to modify the starting address to store the boot_flag.

   For 8267/8269 which supports flash multi-address booting, ota flag is not needed, therefore, the sector 0x73000 ~ 0x73FFF is not occupied by system, and it can be used as user data space.

5. For 8266, the sector 0x72000 ~ 0x72FFF is occupied by BLE stack system, and it serves to store the ota_826x_boot.bin for OTA firmware upgrade. User can follow the instructions in **section 6.2.4.2** to modify the starting address to store the boot bin.

   For 8267/8269 which supports flash multi-address booting, the sector 0x72000 ~ 0x72FFF is not occupied by system, and it can be used as user data space.

6. The 256kB area 0x00000 ~ 0x3FFFF is used as program space by default:

   ✧ The first 128kB area 0x00000 ~ 0x1FFFF is used as storage space for old firmware.

   ✧ The second 128kB area 0x20000 ~ 0x3FFFF is used as storage space for OTA new firmware.

   ✧ If firmware doesn’t need to occupy the whole 128kB space 0x00000 ~ 0x3FFFF, user can use corresponding API to modify the allocation as needed, thus the remaining space can be used as data storage space. Please refer to **section**

6.1.3 and **6.2.4.1** for details.

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

2.1.4.2 Space allocation for 128kB Flash

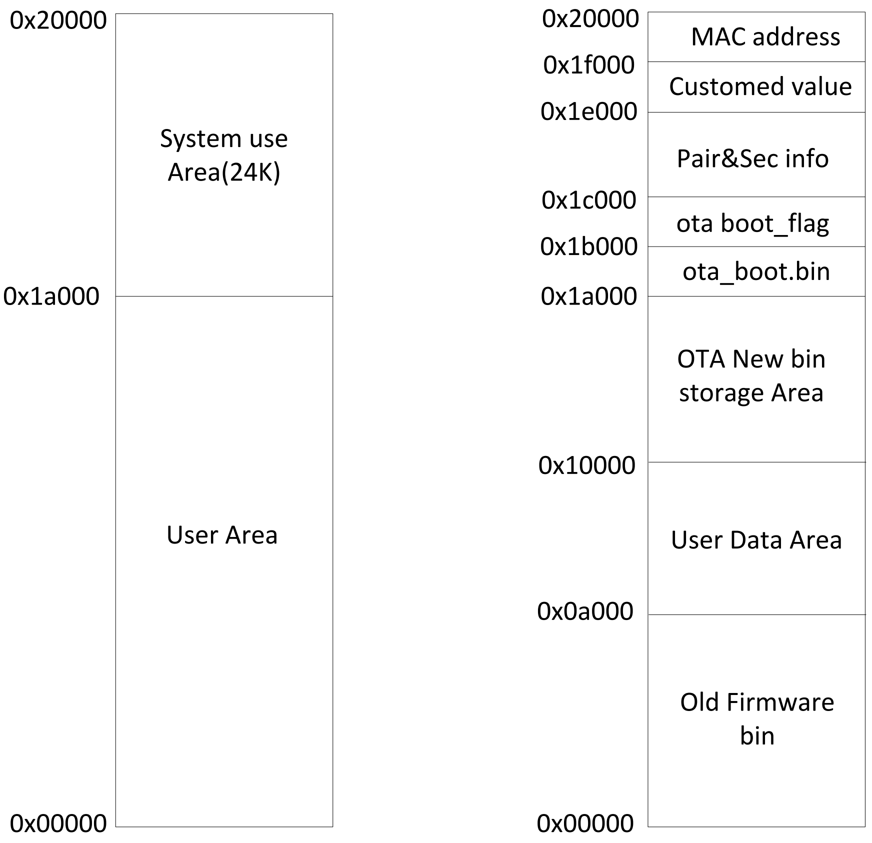


Figure2-6 128kB Flash address space allocation

The figure above shows the default address allocation for the 128kB flash of 8261: The 24kB area 0x1A000 ~ 0x1FFFF is occupied by system, while the 104kB area 0x00000 ~ 0x19FFF is used as storage space for user code and user data. Corresponding interfaces are supplied for user to modify flash address allocation.

1. The sector 0x1F000~0x1FFFF serves to store MAC address. Actually the 6-byte MAC address is stored in area from 0x1F000 (for lower byte of MAC address) to 0x1F005 (for higher byte of MAC address). For example, if “0x11 0x22 0x33 0x44 0x55 0x66” are stored in FLASH 0x1F000~0x1F005, the MAC address is “0x665544332211”.

   Corresponding to SDK, MAC address of actual product will be downloaded into its flash starting from 0x1F000 by Telink jig system. If it’s needed to modify this starting address to store MAC address, user should ensure the consistency. The "user_init" function in the SDK will read MAC address from flash area starting from the macro "CFG_ADR_MAC". This macro is modifiable in the "proj_lib/ble/blt_config.h".

```
#ifndef      CFG_ADR_MAC
#define      CFG_ADR_MAC                    0x1F000

#endif
```

2. The sector 0x1E000~0x1EFFF serves store customized calibration information for Telink MCU. Only this sector does not follow the rule that puts different information types into different sectors; the 4096 bytes in this sector are divided into 64 units with 64 bytes each, and each unit stores one type of calibration information. Since calibration information are burned to corresponding addresses by jig, they can be stored in the same sector; when firmware is running, these calibration information are read only and they're not allowed to be written or erased.
    1) The first 64-byte unit serves to store frequency offset calibration information. Actually this calibration value is 1 byte stored in 0x1E000.
    2) The second 64-byte unit serves to store calibration value of TP value. Actually this calibration information is 2 bytes (TP0, TP1) stored in 0x1E040 and 0x1E041.
    3) The third 64-byte unit serves to store capacitance calibration value of 32kHz RC (reserved in current 8261 BLE SDK).
    4) Following units are reserved for other potential calibration values.

Calibration values of actual product will be downloaded into its flash area corresponding to SDK by Telink jig system. If it's needed to modify the starting address to store calibration value, user should ensure the consistency. In the "user_init" function of SDK, the "rf_customized_param_load()" function will read calibration values from flash area starting from the following macros. These macros are modifiable in the "proj_lib/ble/blt_config.h".

```
#ifndef      CUST_CAP_INFO_ADDR
#define      CUST_CAP_INFO_ADDR          0x1E000
#endif

#ifndef      CUST_TP_INFO_ADDR
#define      CUST_TP_INFO_ADDR           0x1E040
#endif
```

```
#ifndef        CUST_RC32K_CAP_INFO_ADDR
#define        CUST_RC32K_CAP_INFO_ADDR    0x1E080
#endif
```

3. The two sectors 0x1C000 ~ 0x1DFFF are occupied by BLE stack system, and the 8kB area is used to store pairing and security information. User can modify the starting address of the 8K area to store pairing and security information by invoking the function below:

   void bls_smp_configParingSecurityInfoStorageAddr (int addr);

4. The sector 0x1B000 ~ 0x1BFFF is occupied by BLE stack system, and it serves to store the ota boot_flag for OTA firmware upgrade. User can follow the instructions in **section 6.3.4.3** to modify the starting address to store the boot_flag.
5. The sector 0x1A000 ~ 0x1AFFF is occupied by BLE stack system, and it serves to store the ota_826x_boot.bin for OTA firmware upgrade. User can follow the instructions in **section 6.3.4.2** to modify the starting address to store the boot bin.
6. The remaining 104kB space 0x00000 ~ 0x19FFF are configurable area for user code and user data. The default allocation is shown as below:
   - ✧ The 40kB area 0x00000 ~ 0x09FFF is used as storage space for old firmware.
   - ✧ The 40kB area 0x10000 ~ 0x19FFF is used as storage space for OTA new firmware.
   - ✧ The 24kB area 0x0A000 ~ 0x0FFFF is used as storage space for user data.
   - ✧ If the default space allocation does not meet user's requirement, e.g. firmware size exceeds 40kB, or user data need more than 24kB space, corresponding APIs are supplied to modify the allocation as needed. Please refer to **section 6.3.4.1** for details.

2.2 Clock module

System clock is the clock reference for MCU firmware running. The 826x system clock supports multiple sources (PLL, internal OSC, internal RC), but only the most accurate PLL source is used in the 826x BLE SDK. The 192M PLL clock is derived from external 16MHz/12MHz crystal oscillator by automatical process of the internal PLL module. Then lower-frequency system clock can be obtained by configuring related frequency dividing register.

External crystal oscillators including 16M and 12M are supported in 826x BLE SDK. Currently 12M crystal oscillator is used by default, and it's configurable in app_config.h.

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

By setting the value as shown below, the configured crystal type will take effect during RF initialization of main function. “XTAL_12M” indicates 12M crystal type and BLE 1M mode by default.

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

    XTAL_12M   = XTAL_12M_RF_1m_MODE,
    XTAL_16M    = XTAL_16M_RF_1m_MODE,
};
```

```
rf_drv_init(CRYSTAL_TYPE);
```

***Note:** External crystal type (12M/16M) indicates specification for hardware crystal, while system clock (16M) indicates machine cycle for MCU running. No matter which crystal type is configured, a 192M basic clock will be derived from frequency multiplication by internal PLL circuit. According to the system clock frequency configured in app_config, a lower frequency is available from the 192M basic clock by frequency division during clock_init() of main function.

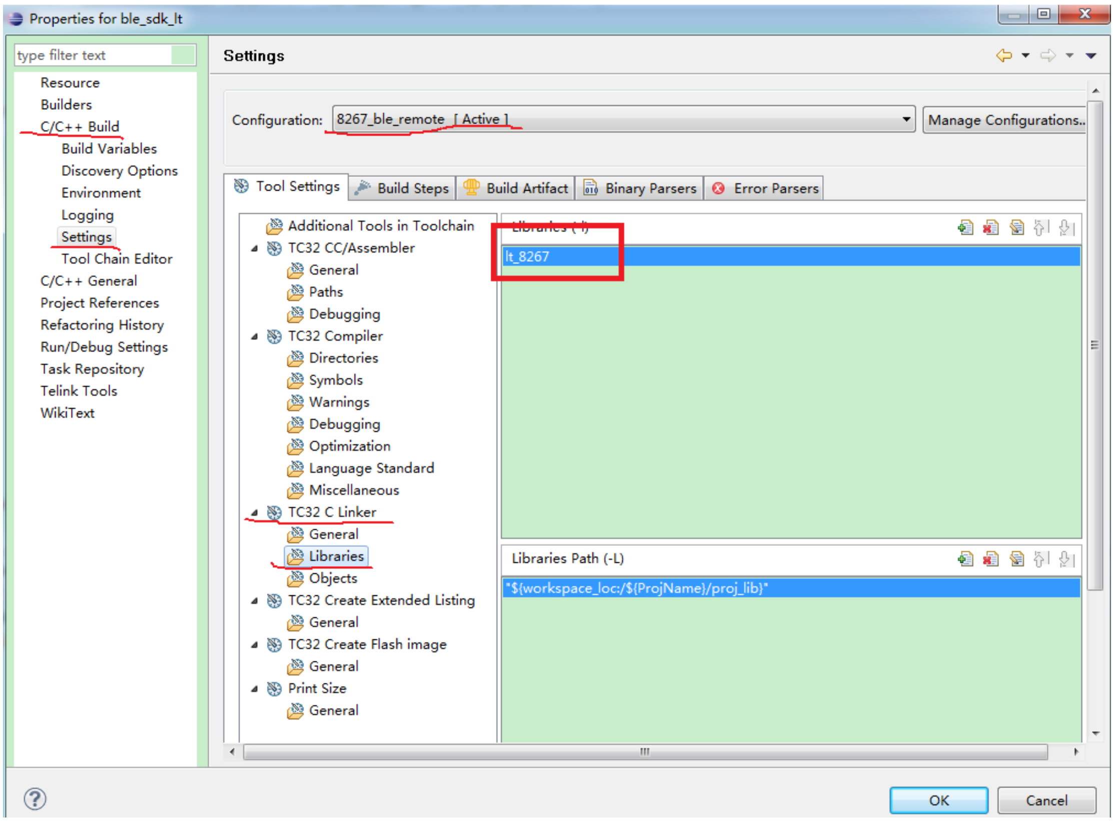
2.2.1 System clock configuration

The “clock_init” function is invoked in the main.c (refer to proj/mcu/clock.c) to configure registers related to clock source and frequency dividing factor. User only needs to configure the following two parameters in the app_config.h.

```
/////////////////// Clock  /////////////////////////////////
#define CLOCK_SYS_TYPE      CLOCK_TYPE_PLL    // Set clock source as PLL
#define CLOCK_SYS_CLOCK_HZ  16000000         //system clock 16M
```

The 16M clock frequency is recommended considering BLE timing sequence and power consumption.

Current 826x BLE SDK supports multiple system clock options: 16M (default), 32M and 48M. To use 32M or 48M options, user needs to configure system clock and select lib library correspondingly. Take 8267 for example: The libraries for 16M/32M/48M system clock are lt_8267 (default), lt_8267_32m and lt_8267_48m, respectively.



Configuration: 8267_ble_remote [Active]

Libraries (-l)
lt_8267

Libraries Path (-L)
"${workspace_loc:/${ProjName}/proj_lib}"

Figure2-7 Modify lib library

2.2.2 system tick usage

The configured 16M system clock starts running after clock initialization (clock_init). The 32-bit system clock counter value (i.e. system clock tick, system tick in brief) will be increased by 1 for each clock cycle (i.e. 1/16us). It takes 268 seconds or so (i.e. (1/16) us * (2^32)) for the system tick to loop from the initial value 0x00000000 to the maximum value 0xffffffff.

Similarly, if the system clock is 32M, the system tick will be increased by 1 for every 1/32us, and it loops with cycle of 134s or so (i.e. (1/32) us * (2^32)).

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

The function “clock_time()” serves to read system tick value.

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

In 826x BLE SDK, the whole BLE timing sequence is based on system tick. It’s highly recommended for user to follow the usage in firmware, i.e. use system tick to implement simple software timer and timeout judgment.

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

1) Start timing: Set an u32 variable, read and record current system clock tick.

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

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

   Suppose timing value is 100ms, for 16M system clock, the following sentence can be used to query the timer:

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

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

   The following sentence shows how to query timer for 32M system clock:

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

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

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

   *Note: For 16M/32M clock, this function only applies to timing within 268s/134s, if exceeds, it’s needed to add timer correspondingly.

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

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

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

2.3 GPIO module

For details about GPIO module, please refer to source code in “proj/mcu_spec/gpio_826x.h”, “gpio_default_826x.h” and “gpio_826x.c”.

To understand register operation, please refer to the two documents including “8266_gpio_lookuptable” and “8267_gpio_lookuptable”, which are available by reading the instruction in “Getting Started with Telink BLE SDK”.

2.3.1 GPIO definition

Telink 826x SDK supports 42 GPIOs divided into six groups, including:

GPIO_PA0 ~ GPIO_PA7, GPIO_PB0 ~ GPIO_PB7, GPIO_PC0 ~ GPIO_PC7,

GPIO_PD0 ~ GPIO_PD7, GPIO_PE0 ~ GPIO_PE7, GPIO_PF0 ~ GPIO_PF1.

Note: Not all of the 42 GPIOs in IC core have corresponding external pins in actual IC packages (e.g. 8261). Please refer to the corresponding pin layout.

Please follow the format above to use GPIO, and refer to “proj/mcu_spec/gpio_826x.h” for details.

There are 7 special GPIOs:

1) MSPI pins: The four GPIOs are dedicated for Flash memory access and correspond to Master SPI bus lines. They are used as MSPI function by default, and it’s not recommended to use them as GPIO function or operate them in firmware. For 8266, MSPI pins are PA2, PA3, PB2 and PB3; for 8261/8267/8269, MSPI pins are PE4~ PE7.

2) SWS (Single Wire Slave): It’s used as SWS function by default for debugging and firmware burning. Generally it is not used in firmware. For 8266, SWS pin is PA0; for 8261/8267/8269, SWS pin is PB0.

3) DM and DP: They are used as USB DM and DP function by default. If USB function is not needed, the two pins can be used as GPIO function. For 8266, DM and DP pins are PB5~ PB6; for 8261/8267/8269, DM and DP pins are PE2~PE3.

2.3.2 GPIO state control

In this section only the basic GPIO states are listed.

All GPIO pins contain the following states:

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

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

Note: "pin" indicates GPIO pin (e.g. GPIO_PA0). "func" can be configured as "AS_GPIO" or other special multiplexed function, as shown below.

```
#define AS_GPIO      0
#define AS_MSPI      1
#define AS_SWIRE  2
#define AS_UART      3
#define AS_PWM       4
#define AS_I2C       5
#define AS_SPI       6
#define AS_I2S       8
#define AS_SDM       9
#define AS_DMIC      10
#define AS_USB       11
#define AS_SWS       12
#define AS_SWM       13

#define AS_ADC       15
```

2) ie: Input enable.

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

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

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

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

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

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

}
```

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

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

4) oe: Output enable.

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

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

5) dataO: Data output.

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

Note: "value": When output is enabled, "1" indicates high-level output, while "0" indicates low-level output.

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

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

Note: "up_down" is configurable as shown below:

- ✧ PM_PIN_PULLUP_1M
- ✧ PM_PIN_PULLUP_10K
- ✧ PM_PIN_PULLDOWN_100K
- ✧ PM_PIN_UP_DOWN_FLOAT

Analog resistor has a feature: In deepsleep, all states of digital modules are invalidated, including input/output state (cannot output level in deepsleep). However, the configured analog resistor can still take effect in deepsleep.

Note: It's not recommended to use 1M pull-up resistor for PC2~PC5 of 8261/8267/8269, since it cannot pull up these pins to stable high level. Therefore, user should try to use 10K pull-up resistor for actual application.

GPIO configuration examples:

1) Configure GPIO_PA4 as high level output.

   gpio_set_func(GPIO_PA4, AS_GPIO) ;   // PA4 is used as general GPIO function by default, so this step to configure "func" can be skipped.

   gpio_set_input_en(GPIO_PA4, 0);

   gpio_set_output_en(GPIO_PA4, 1);

   gpio_write(GPIO_PA4,1)

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

   gpio_set_func(GPIO_PC6, AS_GPIO) ;   // PC6 is used as general GPIO function by default, so this step to configure "func" can be skipped.

   gpio_setup_up_down_resistor(GPIO_PC6, PM_PIN_PULLUP_10K);

   gpio_set_input_en(GPIO_PC6, 1)

   gpio_set_output_en(GPIO_PC6, 0);

   ```
   if(!gpio_read(GPIO_PC6)){    // check if PC6 input is low level
           . . . . . .
   }
   ```

3) Configure USB DM and DP pins of 8261/8267/8269 as general GPIO function.

   gpio_set_func(GPIO_PE2, AS_GPIO) ;

   gpio_set_func(GPIO_PE3, AS_GPIO) ;

2.3.3 GPIO initialization

The "gpio_init" function is invoked in the main.c file to initialize states of all GPIOs. Each IO will be initialized to its default states by the "gpio_init" function, unless related GPIO parameters are pre-configured in the app_config.h.

Default GPIO states are shown as below:

1) func

   Except for the seven special GPIOs (Flash MSPI pins, USB pins, SWS) introduced in **Section 2.3.1**, all other GPIOs are used as general GPIO function by default.

2) ie

   For the seven special GPIOs, the default "ie" state is 1; for other GPIOs, the default "ie" state is 0. User doesn't need to configure "ie" of unused GPIOs as 0; however, to enable input of a general GPIO, corresponding "ie" should be set as 1. Input function needs to be enabled in following cases: scan pin gpio during key scan, core/pad wakeup gpio, irq gpio and etc.

3) oe: all 0 by default.

4) dataO: all 0 by default.

5) Internal pull-up/pull-down resistor: all float by default.

Please refer to “proj/mcu_spec/gpio_826x.h” and “proj/mcu_spec/gpio_default_826x.h” for details.

GPIO default states are indicated by corresponding macros. Take PA7 ie for example:

```
#ifndef PA7_INPUT_ENABLE

#define PA7_INPUT_ENABLE    1

#endif
```

If some macros are pre-configured in the app_config.h, the “gpio_init” function will initialize the corresponding GPIO to the configured value rather than the default value. PA7 is taken as an example to show how to configure GPIO states in app_config.h.

1) Configure func: #define PA7_FUNC AS_GPIO

2) Configure ie: #define PA7_INPUT_ENABLE 1

3) Configure oe: #define PA7_OUTPUT_ENABLE 0

4) Configure dataO: #define PA7_DATA_OUT 0

5) Configure internal pull-up/pull-down resistor:

#define PULL_WAKEUP_SRC_PA7 PM_PIN_UP_DOWN_FLOAT

Conclusion: User can pre-define GPIO initial state in the app_config.h, and initialize corresponding GPIO to the configured value by the gpio_init; or set the GPIO state by the GPIO state control function in the user_init; or combine the two methods to configure the GPIO state. Note that if some state of one GPIO is configured to different values by the app_config.h and user_init, the configuration in the user_init will take effect finally according to firmware timing sequence.

2.3.4 Configure SWS pull-up to avoid MCU error

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

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

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

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

✧ For 8266, SWS is multiplexed with GPIO_PA0. Enable the 1M pull-up resistor for PA0 in the "proj/mcu_spec/gpio_default_8266.h", as shown below.

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

#endif
```

✧ For 8261/8267/8269, SWS is multiplexed with GPIO_PB0. Enable the 1M pull-up resistor for PB0 in the "proj/mcu_spec/gpio_default_8267.h", as shown below.

```
#ifndef PULL_WAKEUP_SRC_PB0
#define PULL_WAKEUP_SRC_PB0      PM_PIN_PULLUP_1M  //sws pullup
#endif
```

3 BLE Module

3.1 BLE SDK software architecture

3.1.1 Standard BLE SDK architecture

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

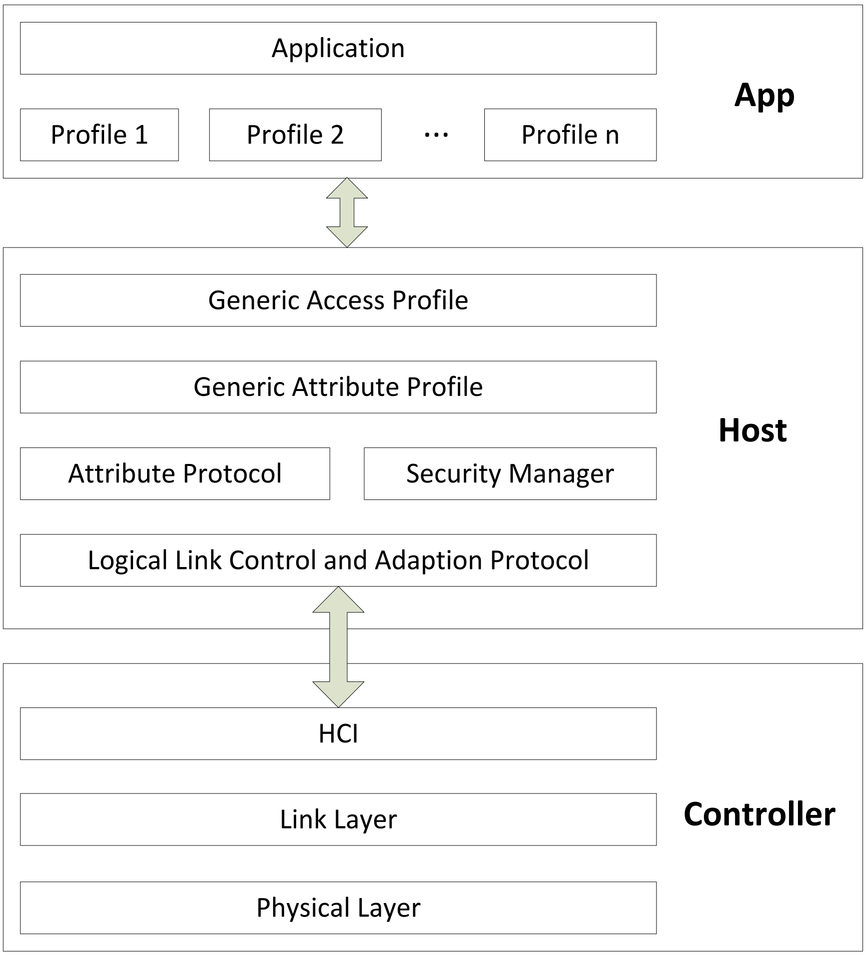


Figure3-1 BLE SDK standard architecture

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

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

and Generic Attribute Profile (GATT).

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

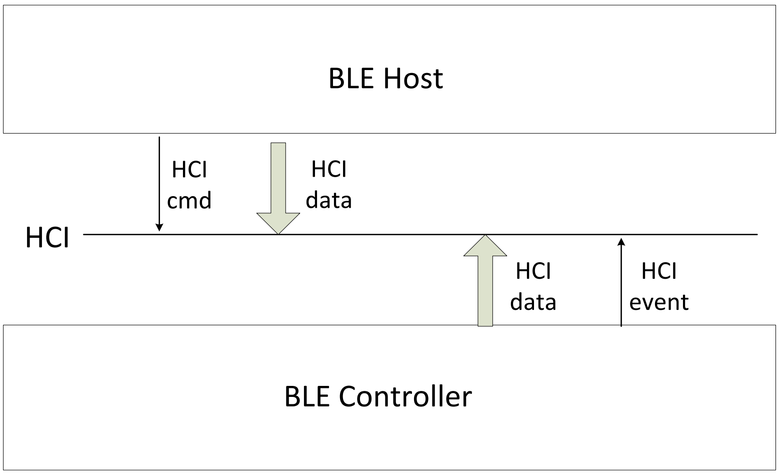


Figure3-2 HCI data transfer between Host and Controller

1) BLE Host will use HCI cmd to operate and set Controller. Controller API corresponding to each HCI cmd will be introduced in **section 0**.
2) Controller will report various HCI events to Host via HCI.
3) Host will send target data to Controller via HCI, while Controller will directly load data to Physical Layer for transfer.
4) When Contoller receives RF data in Physical Layer, it will first check whether the data belong to Link Layer or Host, and then process correspondingly: If the data belong to LL, the data will be processed directly; if the data belong to Host, the data will be sent to Host via HCI.

3.1.2 Telink BLE SDK architecture

3.1.2.1 Telink BLE controller

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

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

In SDK, 826x HCI is a Controller of BLE Slave, to form a standard BLE Slave system, another MCU running BLE Host is needed.

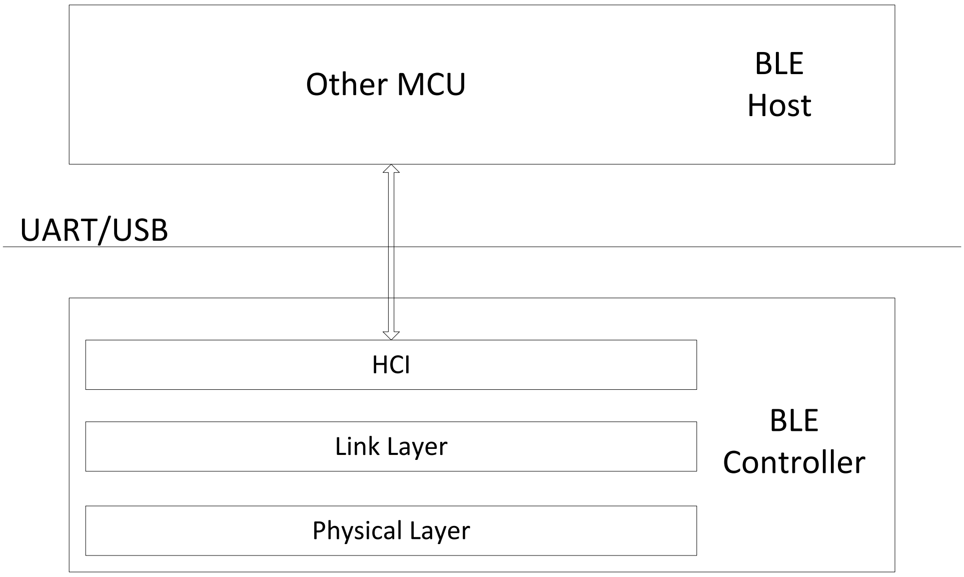


Figure3-3 826x HCI architecture

Link Layer connection state supports Slave and Master of single connection, thus 826x HCI can also be used as BLE Master Controller actually. However, when a BLE Host is running in a complex system (Linux/Android), Master Controller of single connection can only connect with a single device, which is almost meaningless. Therefore, SDK does not include Master role initialization in 826x HCI.

3.1.2.2 Telink BLE Slave

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

When user only needs to use standard BLE Slave, and Telink BLE SDK runs Host (Slave part) + standard Controller, the actual stack architecture will be simplified based on the standard artchitecture, so as to minimize system resource consumption of the whole SDK (including SRAM, running time, power consumption, and etc.). Following shows Telink BLE Slave architecture. In SDK, 826x hid sample, 826x remote and 826x module are all based on this architecture.

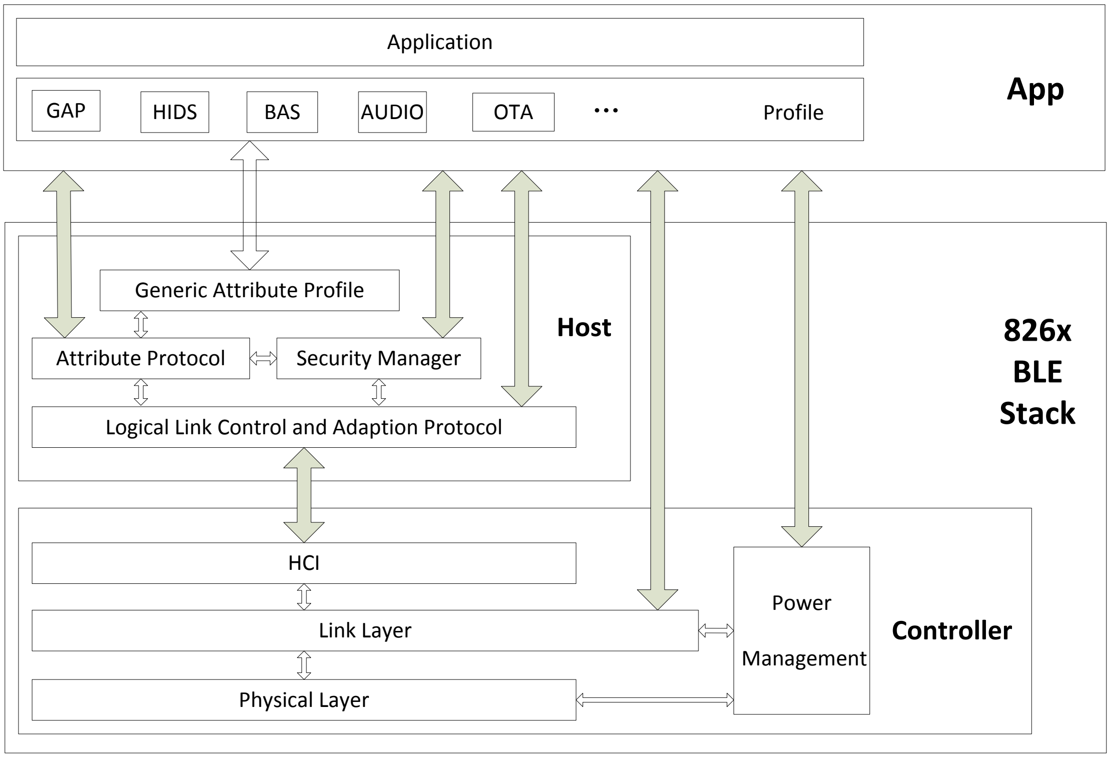


Figure3-4 Telink BLE Slave architecture

In Figure3-4, solid arrows indicate data transfer controllable via user APIs, while hollow arrows indicate data transfer within the protocol stack independent of user.

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

The implementation of Generic Access Profile is deleted from the Host layer, only the service declaration of the GAP profile is retained in the APP layer. Data transfer between the APP layer and the Host is no longer controlled via GAP; the ATT, SMP and L2CAP can directly communicate with the APP layer via corresponding interface.

Generic Attribute Profile (GATT) is implemented in the Host layer based on Attribute Protocol. Various Profiles and Services can be defined in the APP layer based on GATT. Basic Profiles including HIDS, BAS, AUDIO and OTA are provided in 826x BLE SDK demo code.

Physical Layer is totally controlled by Link Layer, and it does not involve the APP layer.

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

3.1.2.3 Telink BLE master

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

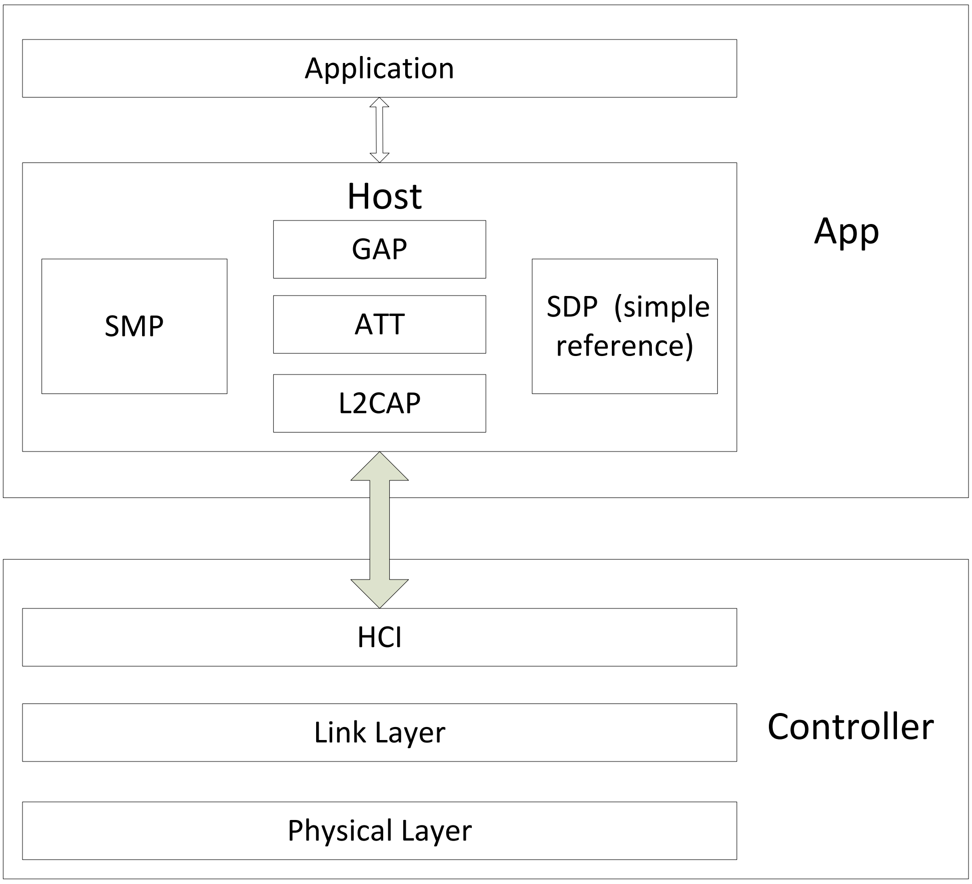


Figure3-5 Telink BLE Master architecture

In SDK, demo code of “826x master kma dongle” project is implemented based on this architecture. Almost all code of Host layer are implemented in APP, and SDK supplies various standard interfaces for user to use these functions.

Standard l2cap and att processing are implemented in APP layer, while the SMP only supplies the basic “just work” method. In the “826x master kma dongle”, SMP is disabled by default, so user needs to enable the corresponding macro to enable SMP. Since SMP implementation is complex, the code is assembled in the library, and the APP layer only needs to invoke related interface. User can search for the corresponding code via the key word “BLE_HOST_SMP_ENABLE”.

```
#define BLE_HOST_SMP_ENABLE                    0
//1 for standard security management,
```

// 0 for telink referenced paring&bonding(no security)

Telink BLE Master does not supply standard implementation for the most complex SDP part, but only gives a simple reference: service discovery of 826x remote. In the "826x master kma dongle", this simple reference SDP is enabled by default.

#define BLE_HOST_SIMPLE_SDP_ENABLE 1 //simple service discovery

In SDK, standard interfaces are supplied for all ATT operations related to service discovery. User can refer to service discovery of 826x remote to implement his own service discovery, or disable "BLE_HOST_SIMPLE_SDP_ENABLE", and use the service ATT handle agreed by Slave to implement data access.

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

3.2 BLE controller

3.2.1 BLE controller introduction

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

Telink BLE SDK fully assembles Physical Layer in the library (corresponding to c file of rf_drv_826x in driver file), while user does not need to learn about it. Power Management will be introduced in detail in **section 4**.

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

3.2.2 Link Layer state machine

Figure3-6 shows Link Layer state machine in BLE spec. Please refer to "Core_v4.2" Page2574 1.1 LINK LAYER STATES.

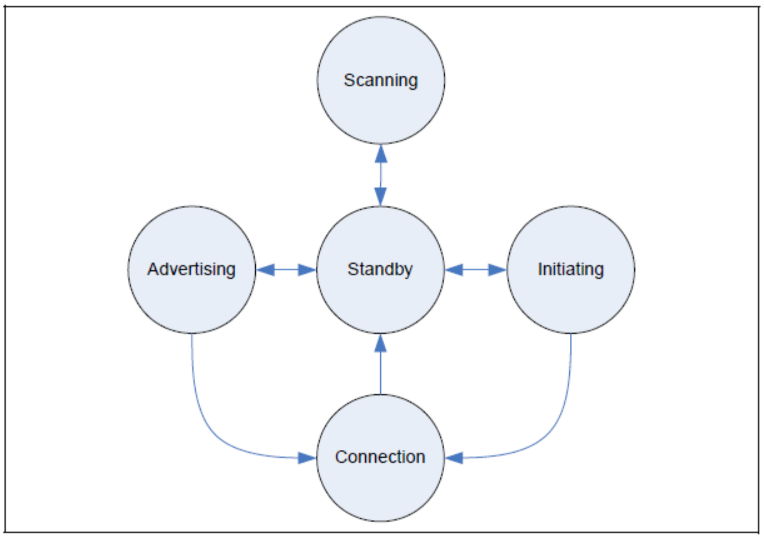


Figure3-6 State diagram of the Link Layer state machine in BLE Spec

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

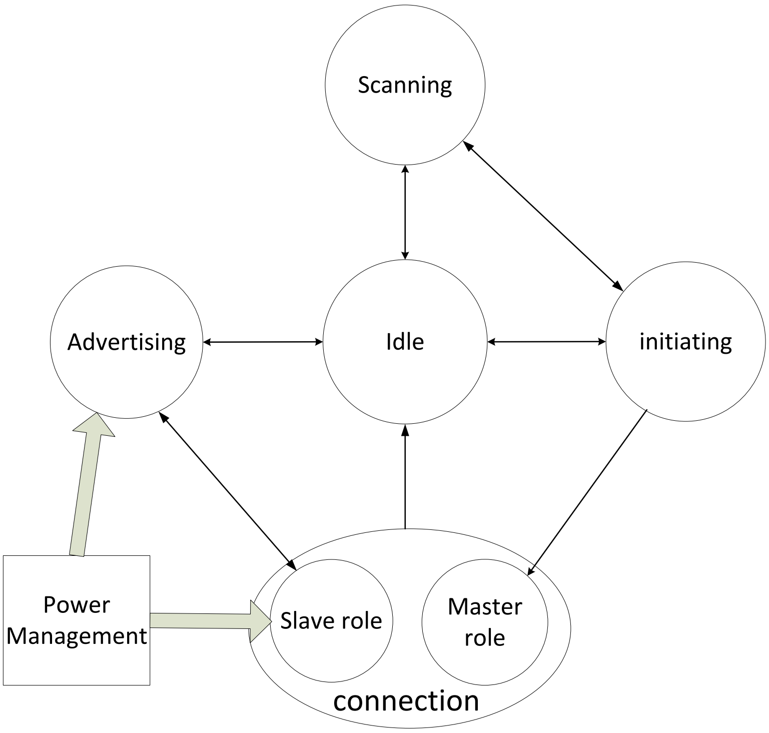


Figure3-7 Telink Link Layer state machine

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

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

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

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

Based on the five states above, corresponding state machine names are defined in "proj_lib/ble/ll/ll.h". "ConnSlaveRole" and "ConnMasterRole" correspond to state name "BLS_LINK_STATE_CONN".

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

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

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

3.2.3 Link Layer state machine combined application

3.2.3.1 Link Layer state machine initialization

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

For example, for BLE Slave application, user needs to add Advertising module and ConnSlaveRole, while the remaining Scanning/Initiating modules are not included so as to save code size and ramcode. The code of unused states won't be compiled.

Following is the API to add the basic Idle module. This API is necessary, since all

BLE applications need initialization.

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

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

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

The actual parameter “public_adr” is the pointer of BLE public mac address.

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

3.2.3.2 Idle + Advtersing

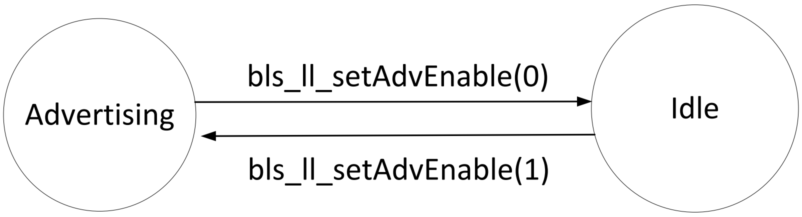


Figure3-8 Idle + Advertising

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

Following is module initialization code of Link Layer state machine.

```
u8  tbl_mac [6] = {……};

blc_ll_initBasicMCU(tbl_mac);

blc_ll_initAdvertising_module(tbl_mac);
```

State switch of Idel and Advertising is implemented via “bls_ll_setAdvEnable”.

3.2.3.3 Idle + Scannning

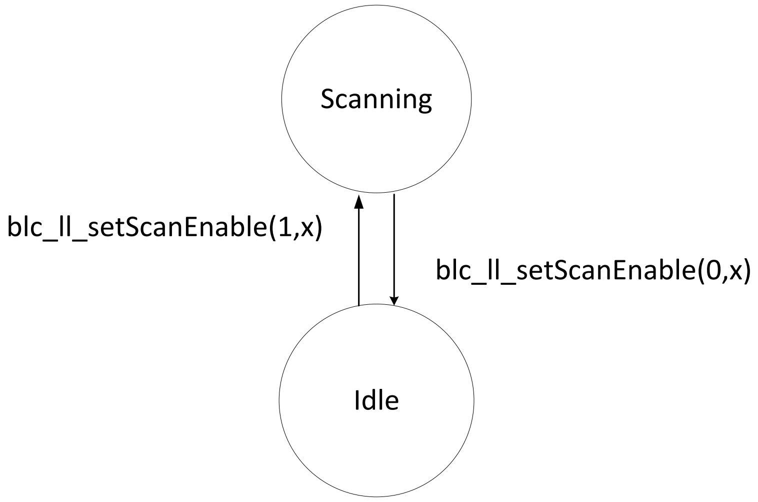


Figure3-9 Idle + Scanning

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

Following is module initialization code of Link Layer state machine.

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

State switch of Idel and Scanning is implemented via “blc_ll_setScanEnable”.

3.2.3.4 Idle + Advtersing + ConnSlaveRole

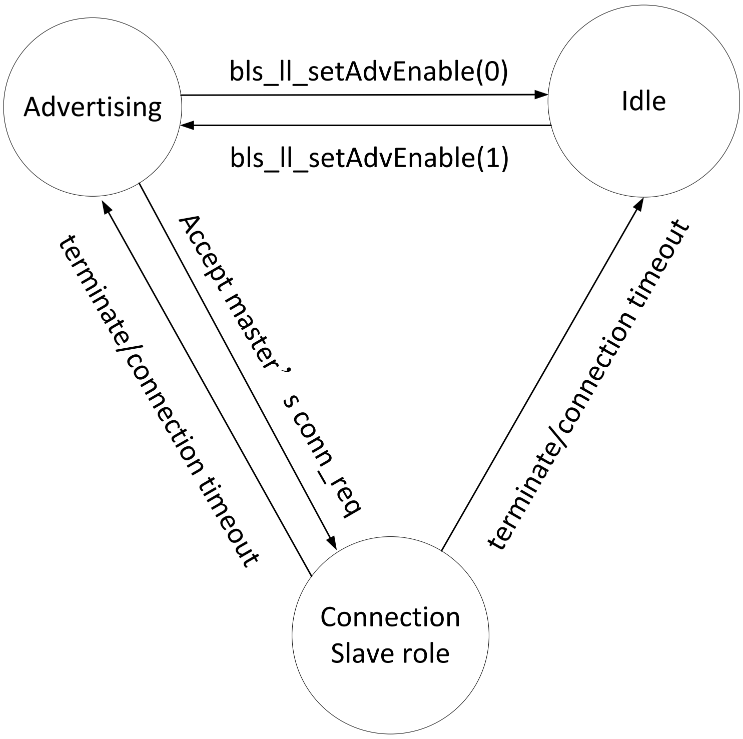

Advertising
Idle
bls_ll_setAdvEnable(0)
bls_ll_setAdvEnable(1)
Accept master' s conn_req
terminate/connection timeout
terminate/connection timeout
Connection
Slave role

Figure3-10 BLE Slave LL state

The figure above shows a Link Layer state machine combination for a basic BLE Slave application. In SDK, 826x hci/826x hid sample/826x remote/826x module are all based on this combination.

Following is module initialization code of Link Layer state machine.

```
u8  tbl_mac [6] = {……};

blc_ll_initBasicMCU(tbl_mac);

blc_ll_initAdvertising_module(tbl_mac);

blc_ll_initSlaveRole_module();
```

State switch in this combination is shown as below:

1) After power on, 826x MCU enters Idle state. In Idle state, adv is enabled, and Link Layer switches to Advertising state; when adv is disabled, it will return to Idle state.

The API “bls_ll_setAdvEnable” serves to enable/disable Adv.

After power on, Link Layer is in Idle state by default. Generally it’s needed to enable Adv in “user_init” so as to enter Advertising state.

2) When Link Layer is in Idle state, Physical Layer won’t take any RF operation

including packet transmission and reception.

3) When Link Layer is in Advertising state, advertising packets are transmitted in adv channels. Master will send conneciton request if it receives adv packet. After Link Layer receives this connection request, it will respond, establish connection and enter ConnSlaveRole.

4) When Link Layer is in ConnSlaveRole, it will return to Idle State or Advertising state in any of the following cases:

   a) Master sends "terminate" command to Slave and requests disconnection. Slave will exit ConnSlaveRole after it receives this command.

   b) By sending "terminate" command to Master, Slave actively terminates the connection and exits ConnSlaveRole.

   c) If Slave fails to receive any packet due to Slave RF Rx abnormity or Master Tx abnormity until BLE connection supervision timeout is triggered, Slave will exit ConnSlaveRole.

When Link Layer exits ConnSlaveRole state, it will switch to Adv/Idle state according to whether Adv is enabled or disabled which depends on the value configured during last invoking of "bls_ll_setAdvEnable" in APP layer. If Adv is enabled, Link Layer returns to Advertising state; if Adv is disabled, Link Layer returns to Idle state.

3.2.3.5 Idle + Scannning + Initiating + ConnMasterRole

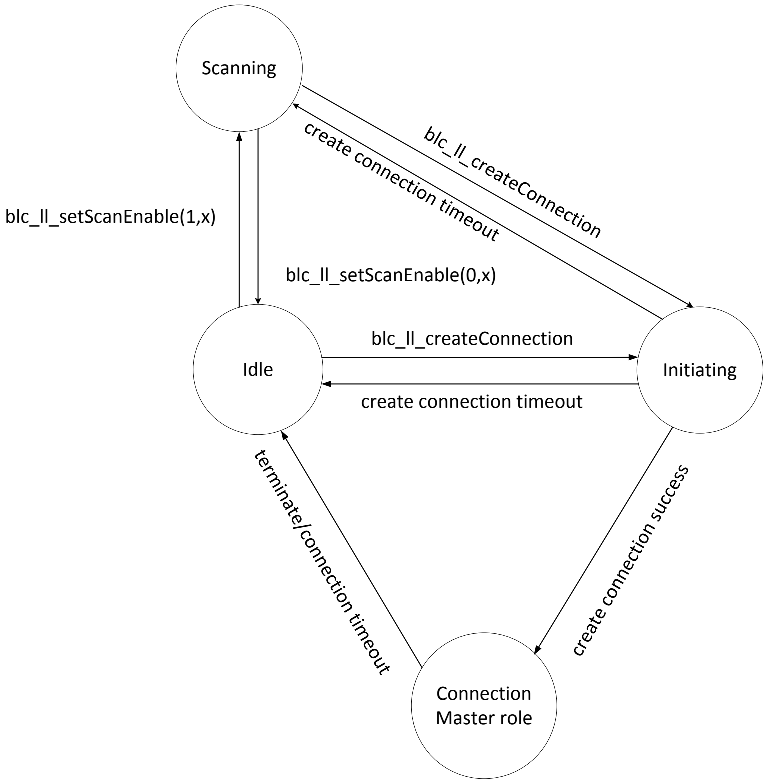

Scanning
blc_ll_setScanEnable(1,x)
blc_ll_setScanEnable(0,x)
blc_ll_createConnection
create connection timeout
Idle
blc_ll_createConnection
create connection timeout
Initiating
terminate/connection timeout
create connection success
Connection Master role

Figure3-11 BLE Master LL state

The figure above shows a Link Layer state machine combination for a basic BLE Master application. In SDK, 826x master kma dongle is based on this combination.

Following is module initialization code of Link Layer state machine.

```
u8  tbl_mac [6] = {……};

blc_ll_initBasicMCU(tbl_mac);

blc_ll_initScanning_module( tbl_mac);

blc_ll_initInitiating_module();

blc_ll_initMasterRoleSingleConn_module();
```

State switch in this combination is shown as below:

1) After power on, 826x MCU enters Idle state. In Idle state, scan is enabled, and Link Layer switches to Scanning State; in Scanning State, when scan is disabled, it will return to Idle state.

   The API “blc_ll_setScanEnable” serves to enable/disable scan.

   After power on, Link Layer is in Idle state by default. Generally it’s needed to enable Scan in “user_init” so as to enter Scanning state.

When Link Layer is in Scanning state, the scanned adv packet will be reported to BLE Host via the event “HCI_SUB_EVT_LE_ADVERTISING_REPORT”.

2) In Idle and Scanning state, Link Layer can be triggered to enter Initiating state via the API “blc_ll_createConnection”.

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

Note that Link Layer can enter Initiating state from Idle state or Scanning state (for example, in the “826x master kma dongle”, LL directly enters Initiating state from Scanning state). After create connection timeout, it will return to previous Idle state or Scanning state.

3) When Link Layer is in ConnMasterRole, it will return to Idle State in any of the following cases:

   a) Slave sends “terminate” command to Master and requests disconnection. Master will exit ConnMasterRole after it receives this command.

   b) By sending “terminate” command to Slave, Master actively terminates the connection and exits ConnMasterRole.

   c) If Master fails to receive any packet due to Master RF Rx abnormity or Slave Tx abnormity until BLE connection supervision timeout is triggered, Master will exit ConnMasterRole.

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

3.2.4 Link Layer timing sequence

In this section, Link Layer timing sequence in various states will be illustrated combining with irq_handler and mainloop of 826x BLE SDK.

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

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

}
```

The “blt_sdk_main_loop” function at BLE entry serves to process data and events related to BLE protocol stack. UI entry is for user application code.

3.2.4.1 Timing sequence in Idle state

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

3.2.4.2 Timing sequence in Advertising state

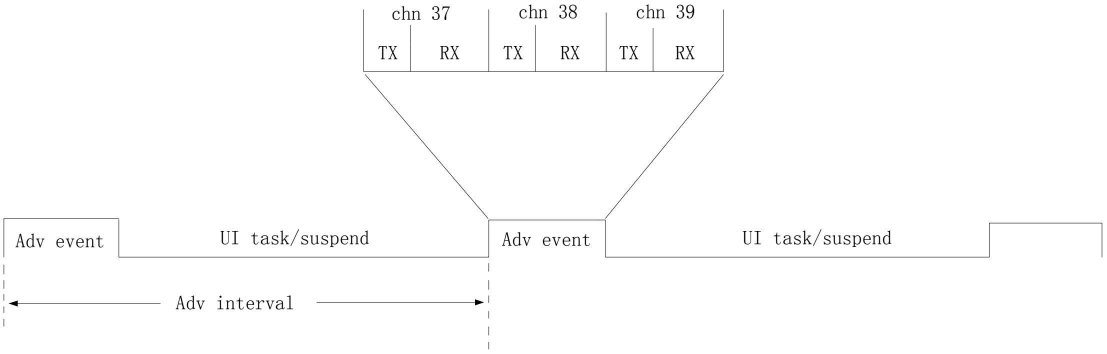


Figure3-12 Timing sequence chart in Advertising State

As shown in Figure3-12, an Adv event is triggered by Link Layer during each adv interval. A typical Adv event with three active adv channels will send an advertising packet in channel 37, 38 and 39, respectively. After an adv packet is sent, Slave enters Rx state, and waits for response from Master: If Slave receives a scan request from Master, it will send a scan response to Master; if Slave receives a connect request from Master, it will establish BLE connection with Master and enter Connection state Slave Role.

Code execution of adv event has some differences in different SDK versions:

1) In BLE SDK 3.0 and 3.1, code of the whole adv event is executed in interrupt irq. MCU will enter interrupt every other adv interval to send and receive packets in

three channels. Adv event will be triggered by system tick irq to enter irq handler.

2) In BLE SDK 3.2, code of adv event is executed in the "blt_sdk_main_loop" function of mainloop instead of interrupt irq. Therefore, adv event won't occupy irq time and lead to failure in real-time irq response.

Code of UI entry in mainloop is executed during UI task/suspend part in Figure3-12. This duration can be used for UI task only, or MCU can enter suspend for the redundant time so as to reduce power consumption.

In Advertising state, the "blt_sdk_main_loop" function does not need to process many tasks, only some callback events related to Adv will be triggered, including BLT_EV_FLAG_ADV_DURATION_TIMEOUT, BLT_EV_FLAG_SCAN_RSP, BLT_EV_FLAG_CONNECT, and etc.

3.2.4.3 Timing sequence in Scanning state

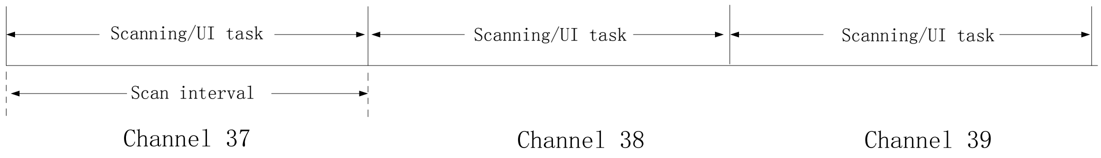


Figure3-13 Timing sequence chart in Scanning state

Scan interval is configured by the API "blc_ll_setScanParameter". During a whole Scan interval, packet reception is implemented in one channel, and Scan window is not designed in SDK. Therefore, SDK won't process the setting of Scan window in "blc_ll_setScanParameter".

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

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

After correct BLE packet is received in Scan interval, the data are first buffered in software RX fifo (corresponding to "my_fifo_t blt_rxfifo" in code), and the "blt_sdk_main_loop" function will check if there are data in software RX fifo. If correct adv data are discovered, the data will be reported to BLE Host via the event "HCI_SUB_EVT_LE_ADVERTISING_REPORT".

3.2.4.4 Timing sequence in Initiating state

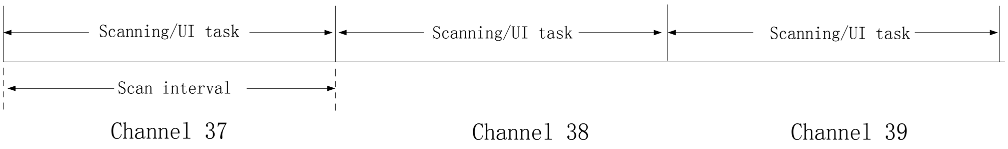


Figure3-14 Timing sequence chart in Initiating state

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

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

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

3.2.4.5 Timing sequence in Conn state Slave role

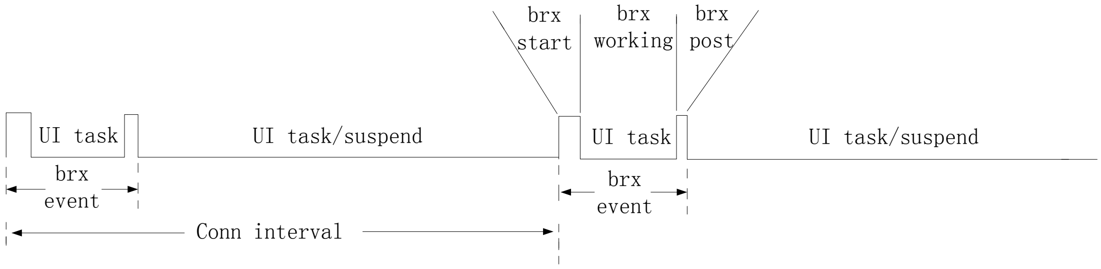


Figure3-15 Timing sequence chart in Conn state Slave role

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

In 826x BLE SDK, each brx process consists of three phases.

1) brx start phase

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

2) brx working phase

After brx start phase ends and MCU exits from irq, hardware in bottom layer enters Rx state first and waits for packet from Master. An ack packet will be sent to respond to each received data packet from Master. During the brx working phase, all packet reception and transmission are implemented automatically without involvement of software.

3) brx post phase

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

During the three phases, brx start and brx post are implemented in interrupt, while brx working phase does not need the involvement of software, and UI task can be executed normally. During the brx working phase, MCU can’t enter suspend since hardware needs to transfer packets.

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

In the ConnSlaveRole, “blt_sdk_main_loop” needs to process the data received during the brx process. During the brx working phase, the data packet received from Master will be copied out during RX interrupt irq handler; these data won’t be processed immediately, but buffered in software RX fifo (corresponding to my_fifo_t blt_rxfifo in code). The “blt_sdk_main_loop” function will check if there are data in software RX fifo, and process the detected data packet correspondingly:

1) Decrypt data packet

2) Analyze data packet

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

3.2.4.6 Timing sequence in Conn state Master role

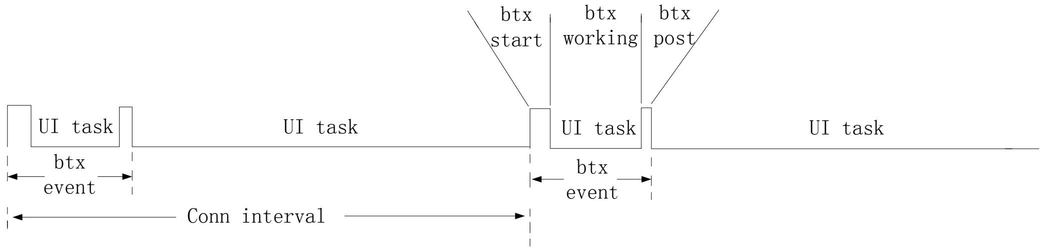


Figure3-16 Timing sequence chart in ConnMasterRole

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

In 826x BLE SDK, each btx process consists of three phases.

1) btx start phase

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

2) btx working phase

After btx start phase ends and MCU exits from irq, hardware in bottom layer enters Tx state first. Master will send packet to Slave and wait for an ack packet from Slave for each transmitted packet. During the btx working phase, all packet reception and transmission are implemented automatically without involvement of software.

3) btx post phase

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

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

In the ConnMasterRole, “blt_sdk_main_loop” needs to process the data received during the btx process. During the btx working phase, the data packet received from Master will be copied out during RX interrupt irq handler; these data won’t be processed immediately, but buffered in software RX fifo. The “blt_sdk_main_loop” function will check if there are data in software RX fifo, and process the detected data packet correspondingly:

1) Decrypt data packet

2) Analyze data packet

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

3.2.4.7 Conn state Slave role timing protection

In ConnSlaveRole state, each interval contains a Brx Event to transfer BLE RF packets. In 826x SDK, since Brx Event is triggered by interrupt, it's needed to enable MCU system interrupt all the time. If user needs to process some time-consuming tasks and must disable system interrupt in Conn state (e.g. erase flash), Brx Event will be stopped, BLE timing sequence will be disturbed, thus connection is terminated.

A timing sequence protection mechanism is supplied in 826x SDK. User should strictly follow this mechanism, so that BLE timing sequence won't be disturbed when Brx Event is stopped. Corresponding APIs are shown as below:

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

void        bls_ll_restoreConnBrxEvent(void);
```

The API "bls_ll_requestConnBrxEventDisable" serves to send a request to disable Brx Event.

1) If the return value is 0, it indicates the request to disable Brx Event is rejected. During Brx working phase in Conn state, the return value must be 0; this request won't be accepted until a whole Brx Event is finished, i.e. it can be accepted only during the remaining UI task/suspend duration.

2) If the return value is not zero, it indicates this request can be accepted, and the returned non-zero value indicates the time (unit: ms) allowed to stop Brx Event.

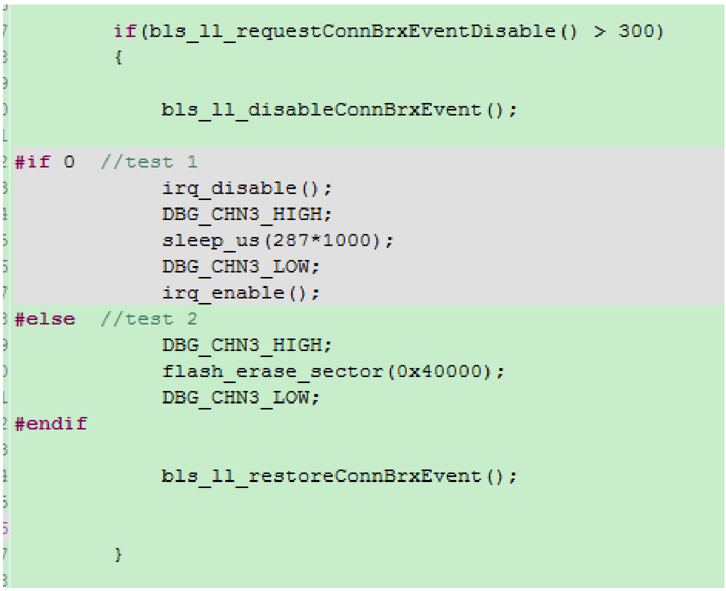
   A. If Link Layer is in Advertising state or Idle state without Brx Event, the return value is "0xffff". In this case, user can disable system interrupt at will.

   B. If Link Layer is in Conn state, and Slave receives "update map" or "update connection parameter" request from Master but does not start updating yet, the retun value should be the difference value of the time to start updating and current time, i.e. it's only allowed to stop Brx Event before the time to start updating, otherwise all following packets won' be received and it will result in disconnection.

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

After the API “bls_ll_requestConnBrxEventDisable” is invoked and the request is accepted, if the time (ms) corresponding to the return value is enough to process user task, the task will be executed. Before the task starts, the API “bls_ll_disableConnBrxEvent” should be invoked to disable Brx Event. After the task is finished, the API “bls_ll_restoreConnBrxEvent” should be invoked to enable Brx Event and restore BLE timing sequence.

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



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

            bls_ll_disableConnBrxEvent();

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

            bls_ll_restoreConnBrxEvent();


        }
```

3.2.5 Link Layer state machine extension

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

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

3.2.5.1 Scanning in Advertising state

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

The API below serves to add Scanning feature:

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

The API below serves to remove Scanning feature:

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

For the two APIs above, the return value of ble_sts_t type should be BLE_SUCCESS.

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

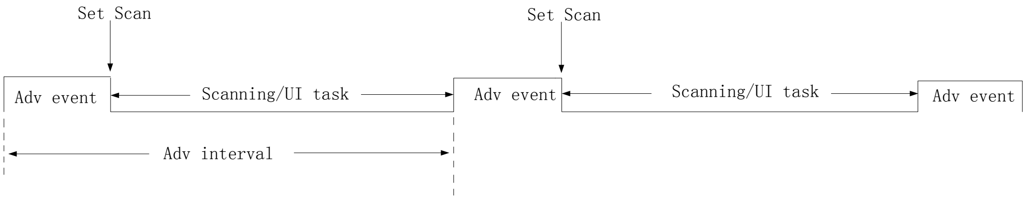


Figure3-17 Timing sequence chart with Scanning in Advertising state

Curretnly Link Layer is still in Advertising state (BLS_LINK_STATE_ADV). During each Adv interval, the remaining time except for Adv event is used for Scanning.

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

For usage of Scanning in Advertising state, please refer to "TEST_SCANNING_IN_ADV_AND_CONN_SLAVE_ROLE" in 826x feature test.

3.2.5.2 Scanning in ConnSlaveRole

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

The API below serves to add Scanning feature:

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

The API below serves to remove Scanning feature:

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

For the two APIs above, the return value of ble_sts_t type should be BLE_SUCCESS.

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

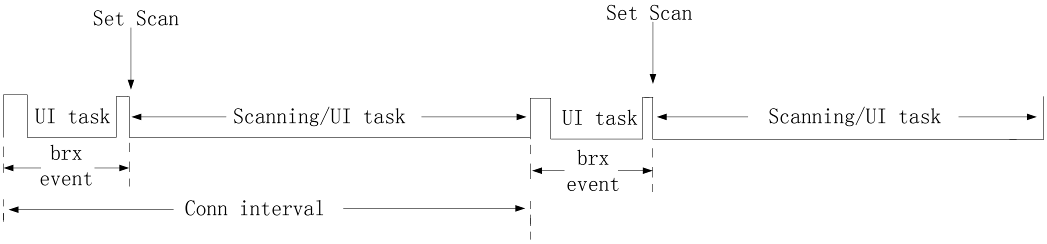


Figure3-18 Timing sequence chart with Scanning in ConnSlaveRole

Curretnly Link Layer is still in ConnSlaveRole (BLS_LINK_STATE_CONN). During each Conn interval, the remaining time except for brx event is used for Scanning.

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

For usage of Scanning in ConnSlaveRole, please refer to "TEST_SCANNING_IN_ADV_AND_CONN_SLAVE_ROLE" in 826x feature test.

3.2.5.3 Advertising in ConnSlaveRole

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

The API below serves to add Advertising feature:

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

The API below serves to remove Advertising feature:

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

For the two APIs above, the return value of ble_sts_t type should be BLE_SUCCESS.

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

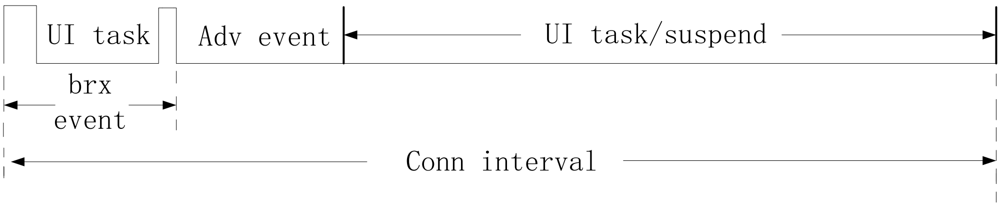


Figure3-19 Timing sequence chart with Advertising in ConnSlaveRole

Curretnly Link Layer is still in ConnSlaveRole (BLS_LINK_STATE_CONN). During each Conn interval, after a brx event is finished, an adv event is executed immediately, and the remaining time is used for UI task or suspend to save power.

For usage of Advertising in ConnSlaveRole, please refer to

"TEST_ADVERTISING_IN_CONN_SLAVE_ROLE" in 826x feature test.

3.2.5.4 Advertising and Scanning in ConnSlaveRole

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

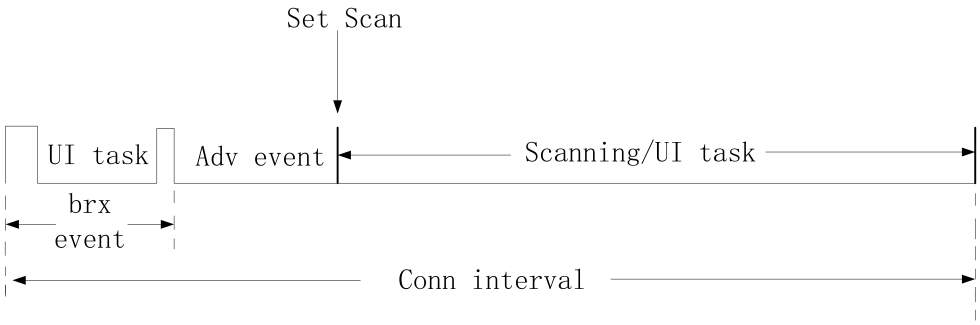


Figure3-20 Timing sequence chart with Advertising and Scanning in ConnSlaveRole

Curretnly Link Layer is still in ConnSlaveRole (BLS_LINK_STATE_CONN). During each Conn interval, after a brx event is finished, an adv event is executed immediately, and the remaining time is used for Scanning.

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

For usage of Advertising and Scanning in ConnSlaveRole , please refer to "TEST_ADVERTISING_SCANNING_IN_CONN_SLAVE_ROLE" in 826x feature test.

3.2.6 Link Layer TX fifo & RX fifo

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

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

BLE TX fifo and BLE RX fifo of Slave role and Master role have some differences in processing, as shown below.

3.2.6.1 Slave role fifo

Both BLE TX fifo and BLE RX fifo in Slave role are defined in APP layer:

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

By default, RX fifo size is 64, and TX fifo size is 40. It's not allowed to modify the two size values unless it's needed to use "data length extension" in core 4.2.

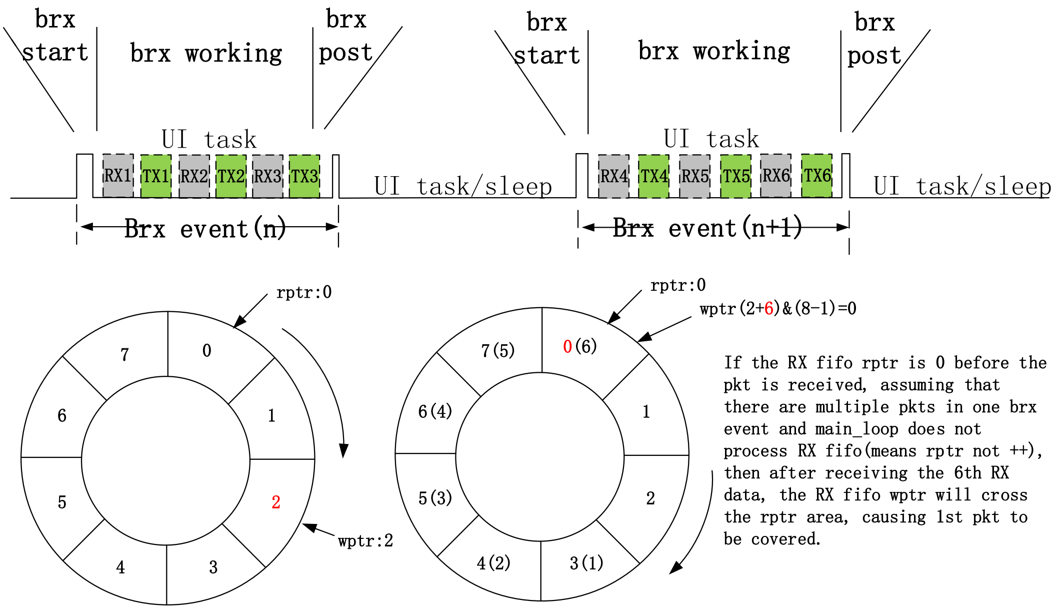
Both TX fifo number and RX fifo number must be configured as a power of 2, i.e. 2, 4, 8, 16, and etc. User can modify as needed.

Default RX fifo number is 8, which is a reasonable value to ensure up to 8 data packets can be buffered in Link Layer bottom layer. If it's set as large value, it will occupy large SRAM area. If it's set as small value, it may bring the risk of data coverage. During brx event, Link Layer is likely to be in more data mode in an interval and continuously reveive multiple packets; if RX fifo number is set as 4, there may be five or six packets in an interval (e.g. in cases such as OTA, play Master audio data), however, due to long decryption time, response to these data by upper layer cannot be processed in real time, then some data may be overflowed. Similarly, if there are more than 8 valid packets in an interval, the default number 8 is not enough.

Below is an example for RX overflow, suppose:

1) RX fifo number is 8;
2) Read/write pointer of RX fifo is 0/2 respectively before brx_event(n) is enabled
3) There are tasks blocked in main_loop during bothbrx_event(n) and brx_event(n+1) stage, and RX fifo is not read in time;
4) Both brx_event stages receive multiple packets.

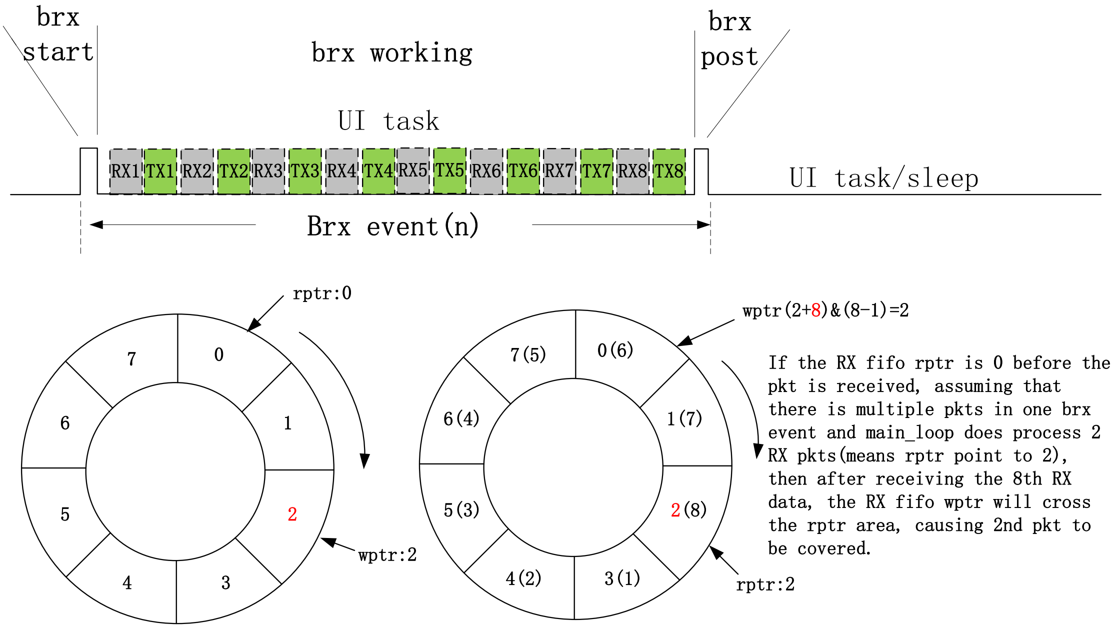
As described in above "Timing Sequence in Conn State Slave Role", during brx_working stage, the received BLE packets will only be copied into RX fifo(RX fifo write pointer++), while reading RX fifo data(RX fifo read pointer++) is executed during main_loop stage, so, the 6th data packet will cover read pointer 0 area. Please be noted, the UI task time interval during brx working stage is the time exclude interrupt time like RX, TX, system timer and ect.



If the RX fifo rptr is 0 before the pkt is received, assuming that there are multiple pkts in one brx event and main_loop does not process RX fifo(means rptr not ++), then after receiving the 6th RX data, the RX fifo wptr will cross the rptr area, causing 1st pkt to be covered.

RX overflow diagram 1

This is an extreme example, because there is a connection interval, and tasks are blocked for a long enough time. Below is a more frequently-seen case: duiring a brx-event, master write multiple data packets into slave, and slave can not process these data in time. As shown below, read pointer shifts 2 bits while write pointer shift 8 bits, this will result in data overflow.



If the RX fifo rptr is 0 before the pkt is received, assuming that there is multiple pkts in one brx event and main_loop does process 2 RX pkts(means rptr point to 2), then after receiving the 8th RX data, the RX fifo wptr will cross the rptr area, causing 2nd pkt to be covered.

RX overflow Diagram 2

Overflow will result in data loss, and will thus cause connection termination of MIC failure for encryption system. (In former version of SDK, brx event Rx IRQ staffs data to RX fifo without overflow checking, so, if main_loop process RX fifo too slow there will be overflow. To avoid this risk, master should avoid sending too many data packets during one connection interval, and UI task processing time should be as short as possible to avoid task blocking).

In new SDK version, we add RX overflow checking: check the difference between write pointer and read pointer of RX fifo and compare it with RX fifo number, if RX fifo is full, then RF will not ACK it, BLE protocol will guarantee re-transmiting the data. SDK also offers RX overflow callback function to acknowledge user, please refer to "Telink defined event" for detail.

If there are more than 8 effective data packets in one interval, the default 8 is not enough.

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

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

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

To send data in APP layer (e.g. invoke "bls_att_pushNotifyData"), user should check current number of TX fifo available for Link Layer.

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

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

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

In audio processing of 826x remote, since a sum of audio data (128-byte) is disassembled into five packets, five TX fifos are needed. Implementation is shown as below (the number of occupied fifos should not exceed 9).

```
if (blc_ll_getTxFifoNumber() < 10)
{
    ……

}
```

To deal with data overflow issue, beside pre-overflow auto-processing mechanism, SDK bottom provides the following API to limit the more data receiving number during 1 interval(users can use this API when they want to limit the data when RX fifo has enough space).

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

The range of more data number for parameter num should not exceed RX fifo number.

Please be noted, the API will enable more data limit function only when be called in APP layer (num > 0).

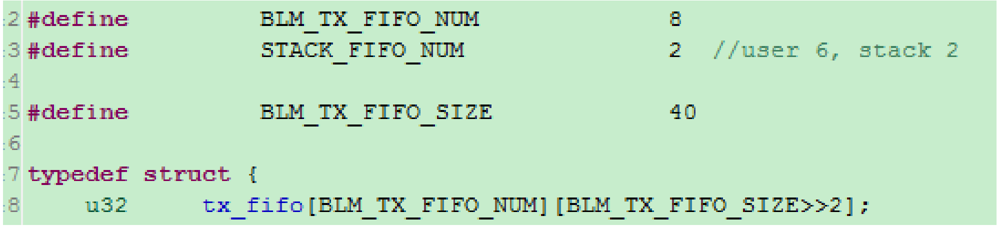
3.2.6.2 Master role fifo

The design of BLE TX fifo and BLE RX fifo in Master role is similar to that of Slave role.

RX fifo is defined in APP layer:

MYFIFO_INIT(blt_rxfifo, 64, 8);

However, user cannot modify TX fifo pre-defined in library. TX fifo number is 8: two fifos are used for stack, while the remaining six fifos are used for APP layer.



```
#define          BLM_TX_FIFO_NUM              8
#define          STACK_FIFO_NUM               2   //user 6, stack 2

#define          BLM_TX_FIFO_SIZE             40

typedef struct {
    u32      tx_fifo[BLM_TX_FIFO_NUM][BLM_TX_FIFO_SIZE>>2];
```

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

```
u8   blm_ll_getTxFifoNumber (u16 connHandle);
```

“connHandle” specifies connection.

Master RX overflow is the same with that in slave. Please refer to RX overflow.

3.2.7 Controller HCI Event

Considering user may need to record and process some key actions of BLE stack bottom layer in APP layer, Telink BLE SDK supplies two types of event: standard HCI event defined by BLE Controller; Telink defined event.

Basically the two sets of event are independent of each other, except for the connect and disconnet event of Link Layer.

User can select one set or use both as needed. In Telink BLE SDK, 826x remote/826x module/826x hid sample use Telink defined event, while 826x hci/826x master kma dongle use Controller HCI event.

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

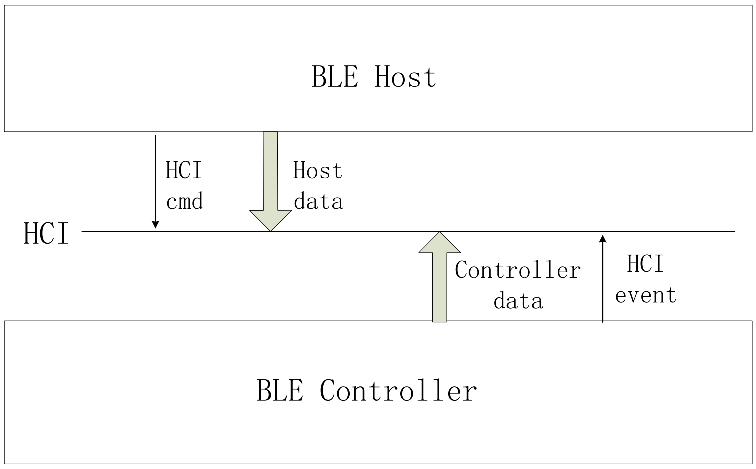


Figure3-21 HCI event

For definition of Controller HCI event, please refer to "Core_v4.2" Page 1152 7.7 "Event". "LE Meta Event" in 7.7.65 indicates HCI LE (low energy) Event, while the others are commom HCI events. As defined in Spec, Telink BLE SDK also divides Controller HCI event into two types: HCI Event and HCI LE event. Since Telink BLE SDK focuses on BLE, it supports most HCI LE events and only a few basic HCI events.

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

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

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

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

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

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

"HCI_FLAG_EVENT_TLK_MODULE" will be introduced in "Telink defined event", while "HCI_FLAG_EVENT_BT_STD" indicates current event is Controller HCI event.

In the callback function prototype, “para” and “n” indicate data and data length of event. The data is consistent with the definition in BLE spec. User can refer to usage in 826x master kma dongle as well as implementation of “app_event_callback” function.

```
    blc_hci_registerControllerEventHandler(app_event_callback);
```

3.2.7.1 HCI event

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

Note: In BLE SDK 3.2.0, “HCI_EVT_DISCONNECTION_COMPLETE” is actually named as “HCI_CMD_DISCONNECTION_COMPLETE”, which is not very reasonable and will be revised in following SDK versions. This document uses “HCI_EVT_DISCONNECTION_COMPLETE” for illustration.

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

1) HCI_EVT_DISCONNECTION_COMPLETE

Please refer to “Core_v4.2” Page 1158 7.7.5 “Disconnection Complete Event”. Total data length of this event is 7, and 1-byte “param len” is 4, as shown below. Please refer to BLE spec for data definition.

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

Figure3-22 Disconnection Complete Event

2) HCI_EVT_ENCRYPTION_CHANGE and HCI_EVT_ENCRYPTION_KEY_REFRESH

The two events (available in 826x master kma dongle) are related to Controller encryption, and the processing is assembled in library.

3) HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE

When Host uses “HCI_CMD_READ_REMOTE_VER_INFO” command to exchange version information between Controller and BLE peer device, and version of peer device is received, this event will be reported to Host.

Please refer to "Core_v4.2" Page 1158 7.7.12 "Read Remote Version Information Complete Event". Total data length of this event is 11, and 1-byte "param len" is 8, as shown below. Please refer to BLE spec for data definition.

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

Figure3-23 Read Remote Version Information Complete Event

4) HCI_EVT_LE_META

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

Except for HCI_EVT_LE_META, other HCI events should use the interface below to enable corresponding mask.

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

Following is definition of mask:

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

If user does not set HCI event mask via this API, by default only the mask corresponding to "HCI_CMD_DISCONNECTION_COMPLETE" is enabled in SDK, i.e. SDK only ensures report of "Controller disconnect event" by default.

3.2.7.2 HCI LE event

When event code in HCI event is "HCI_EVT_LE_META" to indicate HCI LE event, common subevent code are shown as below:

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

1) HCI_SUB_EVT_LE_CONNECTION_COMPLETE

When Controller Link Layer establishes connection with peer device, this event will be reported. Please refer to “Core_v4.2” Page1238 7.7.65.1 “LE Connection Complete Event”. Total data length of this event is 22, and 1-byte “param len” is 19, as shown below. Please refer to BLE spec for data definition.

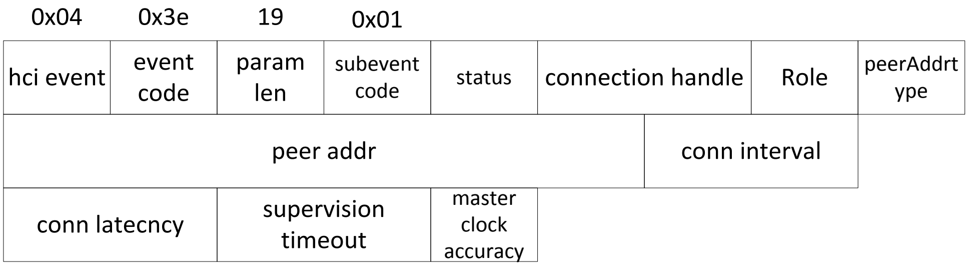


Figure3-24 LE Connection Complete Event

2) HCI_SUB_EVT_LE_ADVERTISING_REPORT

When Controller Link Layer scans right adv packet, it will be reported to Host via “HCI_SUB_EVT_LE_ADVERTISING_REPORT”. Please refer to “Core_v4.2” Page1241 7.7.65.2 “LE Advertising Report Event”. Data length of this event is not fixed and it depends on payload of adv packet, as shown below. Please refer to BLE spec for data definition.

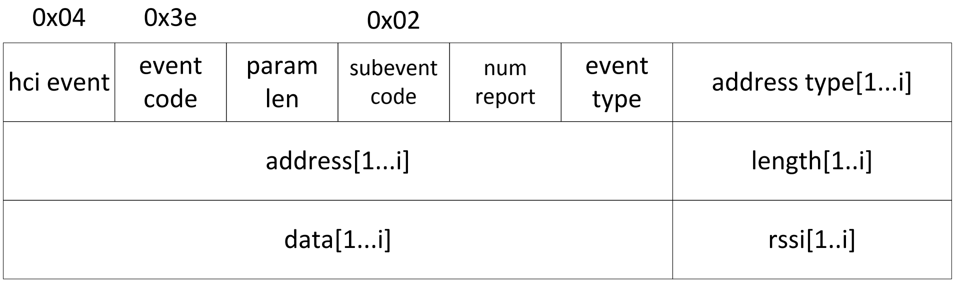


Figure3-25 LE Advertising Report Event

Note: In Telink BLE SDK, each “LE Advertising Report Event” only reports an adv packet, i.e. “i” in Figure3-25 is 1.

3) HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE

When “connection update” in Controller takes effect, “HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE” will be reported to Host. Please refer to “Core_v4.2” Page1243 7.7.65.3 “LE Connection Update Complete Event”. Total data length of this event is 13, and 1-byte “param len” is 10, as shown below. Please refer to BLE spec for data definition.

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

Figure3-26 LE Connection Update Complete Event

4) HCI_SUB_EVT_LE_CONNECTION_ESTABLISH

"HCI_SUB_EVT_LE_CONNECTION_ESTABLISH" is a supplement to "HCI_SUB_EVT_LE_CONNECTION_COMPLETE", so all the parameters except for subevent is the same. In SDK, 826x master kma dongle uses this event.

This Telink private defined event is the sole event which is not standard in BLE spec. This event is only used in 826x master kma dongle.

Following illustrates the reason for Telink to define this event.

When BLE Controller in Initiating state scans adv packet from specific device to be connected, it will send connection request packet to peer device; no matter whether this connection request is received, it will be considered as "Connection complete", "LE Connection Complete Event" will be reported to Host, and Link Layer immediately enters Master role. Since this packet does not support ack/retry mechanism, Slave may miss the connection request, thus it cannot enter Slave role, and won't enter brx mode to transfer packets. In this case, Master Controller will process according to the mechanism below: After it enters Master role, it will check whether any packet is received from Slave during the beginning 6~10 conn intervals (CRC check is negligible). If no packet is received, it's considered that Slave does not receive connection request, suppose "LE Connection Complete Event" has already been reported, it must report a "Disconnection Complete Event" quickly, and indicate disconnect reason is "0x3E (HCI_ERR_CONN_FAILED_TO_ESTABLISH)". If there's packet received from Slave, it can determine Connection Established, thus Master can continue the following flow.

According to the description above, the processing method of BLE Host should be: After it receives "Connection Complete Event" of Controller, it cannot consider connection has already been established, but starts a timer based on conn interval (timing value should be configured as 10 intervals or above to cover the longest time). After the timer is started, it will check whether there is "Disconnection Complete Event" with disconnect reason of 0x3E; if there is no such event, it will be considered as "connection Established".

Considering this processing of BLE Host is very complex and error prone, SDK defines "HCI_SUB_EVT_LE_CONNECTION_ESTABLISH" in the bottom layer. When Host receives this event, it indicates that Controller has determined connection is OK on Slave side and can continue the following flow.

"HCI LE event" needs the interface below to enable mask.

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

Following lists some evtMask definitions. User can view the other events in the "hci_const.h".

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

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

3.2.8 Telink defined event

Besides standard Controller HCI event, SDK also supplies Telink defined event.

In terms of user application, events are from two parts including Host and Controller (equivalent to the whole BLE stack). Most events are from Controller, and will be introduced in this section. The Host part will introduce a few events from Host.

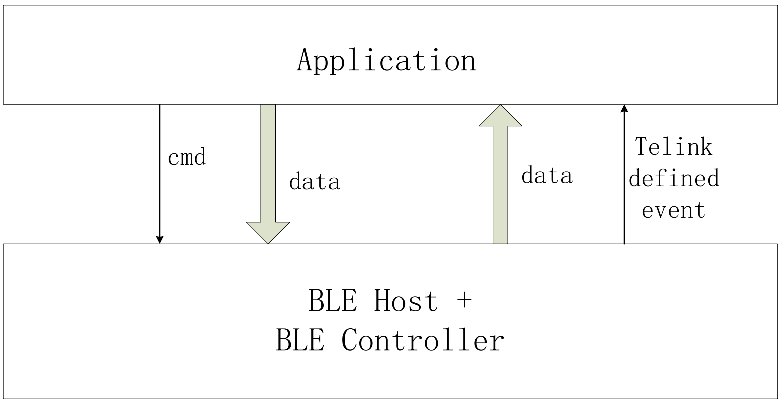


Figure3-27 Architecture of Telink defined event

Up to 20 Telink defined events are supported, which are defined by using macros in "proj_lib/ble/ll/ll.h".

Current new SDK supports the following callback events. “BLT_EV_FLAG_CONNECT/BLT_EV_FLAG_TERMINATE” has the same function as “HCI_SUB_EVT_LE_CONNECTION_COMPLETE” /“HCI_EVT_DISCONNECTION_COMPLETE” in HCI event, but data definition of these events are different.

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

Telink defined event is only used in BLE Slave applications (remote/module), and won’t be triggered in BLE Master. Libraries of BLE Slave can be divided into “Control Core + Bluetooth” type and “SPP” type (826x module, Bluetooth only), which correspond to different event callback processing in SDK by APP layer.

For libraries of “Control Core + Bluetooth” type, e.g. lib_8267/lib_8269, prototype of callback function is shown as below:

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

“e”: event number.

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

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

The API below serves to register callback function:

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

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

Take “BLT_EV_FLAG_CONNECT” as an example to illustrate the usage of register function. When advertising device sends adv packet, Master sends “connection request” to request connection; after this request is received, advertising device processes correspondingly and enters Conn state Slave role. Then stack checks whether the callback function of the event “BLT_EV_FLAG_CONNECT” is registered: if registered, the registered function will be invoked to implement user-defined operations (e.g. recording or related setting when device enters connection).

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

The setting above will invoke the “task_connect” function each time when BLE slave receives connection request and enters connection state.

For libraries of “SPP” type, e.g. lib_8261_mod/lib_8266_mod, function prototype and the interface to register callback function is the similar to HCI event.

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

The difference is actual value of “u32 h”:

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

“e”: event number of Telink defined event.

Similar to mask of HCI event, the API below serves to set the mask to determine whether each event will be responded.

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

The relationship between evtMask and event number is shown as below:

evtMask = BIT(e);

User can refer to processing of Telink defined event in 826x module to help understanding.

In the following sub-sections, all events, event trigger condition and parameters of corresponding callback function for Controller will be introduced in detail. “BLT_EV_FLAG_PAIRING_BEGIN” and “BLT_EV_FLAG_PAIRING_END” event, which do not belong to Controller, will be introduced in Host SMP.

3.2.8.1 BLT_EV_FLAG_ADV

This event is not used in current SDK.

3.2.8.2 BLT_EV_FLAG_ADV_DURATION_TIMEOUT

1) Event trigger condition: If the API “bls_ll_setAdvDuration” is invoked to set advertising duration, a timer will be started in BLE stack bottom layer. When the timer reaches the specified duration, advertising is stopped, and this event is triggered. In the callback function of this event, user can implement operations such as modifying adv event type, re-enabling advertising, re-configuring advertising duration and etc.
2) Pointer “p”: null pointer.
3) Data length “n”: 0.

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

3.2.8.3 BLT_EV_FLAG_SCAN_RSP

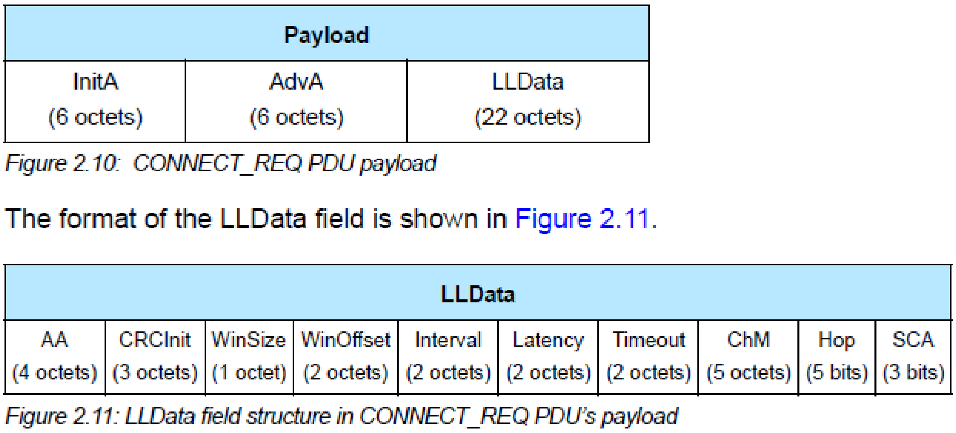
1) Event trigger condition: When Slave is in advertising state, this event will be triggered if Slave responds with scan response to the scan request from Master.
2) Pointer “p”: null pointer.
3) Data length “n”: 0.

3.2.8.4 BLT_EV_FLAG_CONNECT

1) Event trigger condition: When Link Layer is in advertising state, this event will be triggered if it responds to connect reqeust from Master and enters Conn state Slave role.

2) Data length “n”: 34.

3) Pointer “p”: p points to one 34-byte RAM area, corresponding to the “connect request PDU” below.



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

Figure 2.10: CONNECT_REQ PDU payload

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

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

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

Figure3-28 Connect request PDU

Please refer to “rf_packet_connect_t” defined in “ble_common.h”. In the structure below, the connect request PDU is from scanA[6] (corresponding to InitA in Figure3-28) to hop.

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

3.2.8.5 BLT_EV_FLAG_TERMINATE

1) Event trigger condition: This event will be triggered when Link Layer state machine exits Conn state Slave role in any of the three specific cases.

2) Pointer “p”: p points to an u8-type variable “terminate_reason”. This variable

indicates the reason for disconnection of Link Layer.

3) Data length “n”: 1.

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

1) If Slave fails to receive packet from Master for a duration due to RF communication problem (e.g. bad RF or Master is powered off), and “connection supervision timeout” expires, this event will be triggered to terminate connection and return to None Conn state. The terminate reason is HCI_ERR_CONN_TIMEOUT (0x08).
2) If Master sends “terminate” command to actively terminate connection, after Slave responds to the command with an ack, this event will be triggered to terminate connection and return to None Conn state. The terminate reason is the Error Code in the “LL_TERMINATE_IND” control packet received in Slave Link Layer. The Error Code is determined by Master. Common Error Codes include HCI_ERR_REMOTE_USER_TERM_CONN (0x13), HCI_ERR_CONN_TERM_MIC_FAILURE (0x3D), and etc.
3) If Slave invokes the API “bls_ll_terminateConnection(u8 reason)” to actively terminate connection, this event will be triggered. The terminate reason is the actual parameter “reason” of this API.

3.2.8.6 BLT_EV_FLAG_ENCRYPTION_CONN_DONE

1) Event trigger condition: This event will be triggered when encryption of Link Layer is finished (i.e. Link Layer receives “start encryption response” from Master).
2) Pointer “p”: p points to an u8-type variable “smp_flag”, which indicates current encryption of Link Layer is triggered during first pairing or auto re-connection. If “smp_flag” is 0, it indicates first pairing; if “smp_flag” is 1, it indicates auto re-connection.

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

3) Data length “n”: 1.

3.2.8.7 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE

1) Event trigger condition: This event will be triggered when Slave and Master exchange max data length of Link Layer, i.e. one side sends “ll_length_req”, while the other side responds with “ll_length_rsp”. If Slave actively sends “ll_length_req”, this event won’t be triggered until “ll_length_rsp” is received. If Master initiates “ll_length_req”, this event will be triggered immediately after

Slave responds with “ll_length_rsp”.

2) Data length “n”: 12.

3) Pointer “p”: p points to data of a memory area, corresponding to the beginning six u16-type variables in the structure below.

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

    u16     connInitialMaxTxOctets;
    u8      connMaxTxRxOctets_req;

    }ll_data_extension_t;
```

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

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

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

connEffectiveMaxRxOctets = min(supportedMaxRxOctets,connRemoteMaxTxOctets);

connEffectiveMaxTxOctets = min(supportedMaxTxOctets, connRemoteMaxRxOctets);

3.2.8.8 BLT_EV_FLAG_GPIO_EARLY_WAKEUP

1) Event trigger condition: Slave will calculate wakeup time before it enters suspend, so that it can wake up when the wakeup time is due (It’s realized via timer in suspend state). Since user tasks won’t be processed until wakeup from suspend, long suspend time may bring problem for real-time demanding applications. Take keyboard scanning as an example, if user presses keys fast, to avoid key press loss and process debouncing, it’s recommended to set the scan interval as 10~20ms; longer suspend time (e.g. 400ms or 1s, which may be reached when latency is enabled) will lead to key press loss. So it’s needed to judge current suspend time before MCU enters suspend; if it’s too long, the wakeup method of user key press should be enabled, so that MCU can wake up from suspend in advance (i.e. before timer timeout) if any key press is detected. This will be introduced in details in

following PM module section.

The event "BLT_EV_FLAG_GPIO_EARLY_WAKEUP" will be triggered if MCU is woke up from suspend by GPIO in advance before wakeup timer expires.

2) Data length "n": 1.

3) Pointer "p": p points to an u8-type variable "wakeup_status". This variable indicates valid wakeup source status for current suspend. Following types of wakeup status are defined in "proj_lib/pm_826x.h" ("WAKEUP_STATUS_COMP" is not used in SDK).

```
enum {
    WAKEUP_STATUS_COMP   = BIT(0),
    WAKEUP_STATUS_TIMER  = BIT(1),
    WAKEUP_STATUS_CORE   = BIT(2),
    WAKEUP_STATUS_PAD    = BIT(3),
    STATUS_GPIO_ERR_NO_ENTER_PM  = BIT(7),

};
```

For parameter definition above, please refer to the return value "int" of the API in "Power Management":

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

3.2.8.9 BLT_EV_FLAG_CHN_MAP_REQ

1) Event trigger condition: When Slave is in Conn state, if Master needs to update current connection channel list, it will send a "LL_CHANNEL_MAP_REQ" command to Slave; this event will be triggered after Slave receives this request from Master and has not processed the request yet.

2) Data length "n": 5.

3) Pointer "p": p points to the starting address of the following channel list array.

unsigned char type bltc.conn_chn_map[5]

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

Five bytes are used in "conn_chn_map" to indicate current channel list by mapping. Each bit indicates a channel:

conn_chn_map[0] bit0-bit7 indicate channel0~channel7, respectively.

conn_chn_map[1] bit0-bit7 indicate channel8~channel15, respectively.

conn_chn_map[2] bit0-bit7 indicate channel16~channel23, respectively.

conn_chn_map[3] bit0-bit7 indicate channel24~channel31, respectively.

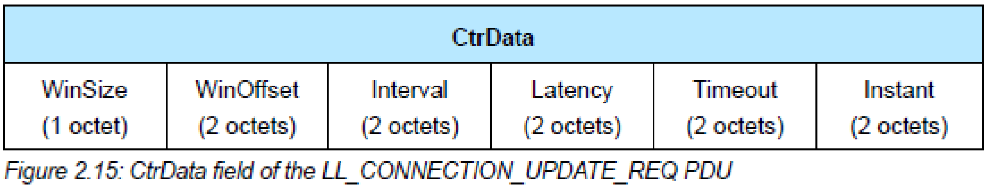
conn_chn_map[4] bit0-bit4 indicate channel32~channel36, respectively.

3.2.8.10 BLT_EV_FLAG_CHN_MAP_UPDATE

1) Event trigger condition: When Slave is in connection state, this event will be triggered if Slave has updated channel map after it receives the "LL_CHANNEL_MAP_REQ" command from Master.
2) Pointer "p": p points to the starting address of the new channel map conn_chn_map[5] after update.
3) Data length "n": 5.

3.2.8.11 BLT_EV_FLAG_CONN_PARA_REQ

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



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

Figure 2.15: CtrData field of the LL_CONNECTION_UPDATE_REQ PDU

Figure3-29 LL_CONNECTION_UPDATE_REQ format in BLE stack

3.2.8.12 BLT_EV_FLAG_CONN_PARA_UPDATE

1) Event trigger condition: When Slave is in connection state, this event will be triggered if Slave has updated connection parameters after it receives the "LL_CONNECTION_UPDATE_REQ" from Master.
2) Data length "n": 6.
3) Pointer "p": p points to the new connection parameters after update, as shown below.

   p[0] | p[1]<<8:   new connection interval in unit of 1.25ms.

p[2] | p[3]<<8:	new connection latency.
p[4] | p[5]<<8:	new connection timeout in unit of 10ms.

3.2.8.13 BLT_EV_FLAG_SUSPEND_ENETR

1) Event trigger condition: When Slave executes the function "blt_sdk_main_loop", this event will be triggered before Slave enters suspend.
2) Pointer "p": Null pointer.
3) Data length "n": 0.

3.2.8.14 BLT_EV_FLAG_SUSPEND_EXIT

1) Event trigger condition: When Slave executes the function "blt_sdk_main_loop", this event will be triggered after Slave is woke up from suspend.
2) Pointer "p": Null pointer.
3) Data length "n": 0.

Note: This callback is executed after SDK bottom layer executes "cpu_sleep_wakeup" and Slave is woke up, and this event will be triggered no matter whether the actual wakeup source is gpio or timer. If the event "BLT_EV_FLAG_GPIO_EARLY_WAKEUP" occurs at the same time, for the sequence to execute the two events, please refer to pseudo code in "Power Management – PM Working Mechanism".

3.2.8.15 BLT_EV_FLAG_READ_P256_KEY

To be added.

3.2.8.16 BLT_EV_FLAG_GENERATE_DHKEY

To be added.

3.2.8.17 BLT_EV_FLAG_LL_REJECT_IND

1) Event trigger condition: when master send LL_ENC_REQ(encryption request) in link layer, and claim using the allocated LTK, and slave can not find the respective LTK, send LL_REJECT_IND(or LL_REJECT_EXT_IND).
2) Pointer "p": point to sending command (LL_REJECT_IND or LL_REJECT_EXT_IND).
3) Data length "n": 0.

Please refer to 《Core_v5.0》(Vol 6/Part B/2.4.2) for more information.

3.2.8.18 BLT_EV_FLAG_RX_DATA_ABANDOM

1) Event trigger condition: when BLE RX fifo overflow(refer to “Link Layer TX fifo & RX fifo”), or in a continuous interval, received packet number > set packet number threshold(when user call API: blc_ll_init_max_md_nums, and the parameter is not 0, SDK bottom layer will check the packet number), trigger BLT_EV_FLAG_RX_DATA_ABANDOM.
2) Pointer ”p”:Null pointer.
3) Data length “n”: 0.

3.2.8.19 BLT_EV_FLAG_PHY_UPDATE

1) Event trigger condition: when PHY(BLE1M/BLE2M) reaches update time, trigger this even.
2) Pointer ”p”: Point to updated PHY, PHY values are the emumeration value, listed as following:

```
BLE_PHY_1M    = BIT(0),
BLE_PHY_2M    = BIT(1),
```

3.2.9 Controller API

3.2.9.1 Controller API brief

In standard BLE stack architecture (see Figure3-1), APP layer cannot directly communicate with Link Layer of Controller, i.e. data of APP layer must be first transferred to Host, and then Host can transfer control command to Link Layer via HCI. All control commands from Host to LL via HCI follow the definition in BLE spec “Core_v4.2”, please refer to Vol2: Core System Package[BR/EDR Controller volume], Part E：Host Controller Interface Functional Specification.

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

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

The APIs below correspond to Host commands in Spec.

In BLE spec, all HCI commands to operate Controller have corresponding “HCI command complete event” or “HCI command status event” as response to Host layer. However, in Telink BLE SDK, the following cases apply:

1) For applications such as 826x_hci, Telink IC only serves as BLE controller, and needs to work together with BLE Host MCU. Each HCI command will generate corresponding “HCI command complete event” or “HCI command status event”.
2) For applications such as 826x master kma dongle, both BLE Host and Controller run on Telink IC, when Host invokes interface to send HCI command to Controller, Controller can receive all data correctly without loss. Therefore, when Controller processes HCI command, it won’t reply with “HCI command complete event” or “HCI command status event”.

Controller API declaration is available in head files under “proj_lib/ble/ll” and “proj_lib/ble/hci”. Corresponding to Link Layer state machine functions, the “ll” directory contains ll.h, ll_adv.h, ll_scan.h, ll_init.h, ll_slave.h and ll_master.h, e.g. APIs related to advertising function should be in ll_adv.h.

3.2.9.2 API return type ble_sts_t

An enum type “ble_sts_t” defined in “proj_lib/ble/ble_common.h” is used as return value type for most APIs in SDK. When API invoking with right parameter setting is accepted by the protocol stack, it will return “0” to indicate BLE_SUCCESS; if any non-zero value is returned, it indicates a unique error type. All possible return values and corresponding error reason will be listed in the subsections below for each API.

The “ble_sts_t” applies to APIs of all layers, including the Link Layer.

3.2.9.3 MAC address initialization

In this document, “BLE MAC address” indicates “public address” by default.

As introduced above, the 6-byte BLE MAC address will be downloaded into specific flash area of the actual product by Telink jig system. User needs to obtain the MAC address from the Bluetooth SIG.

Take 8267 512K Flash for example: Currently during debugging phase, the MAC address is processed by SDK as shown below. When it’s first time to power on Slave, if the MAC address read from flash 0x76000 is null, the MAC address will be set as “0xC7E4E3E2E1xx” (The former five bytes are fixed, and the final one byte is randomly generated); then the six bytes will be written into flash 0x76000~0x76005. After power cycle, the MAC address read from flash 0x76000 already exists and it can be used directly to ensure MAC address consistency of Slave.

Related code sample is shown below for reference, and user can refer to flash access introduction to understand the code. User can also modify the code as needed.

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

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

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

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

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

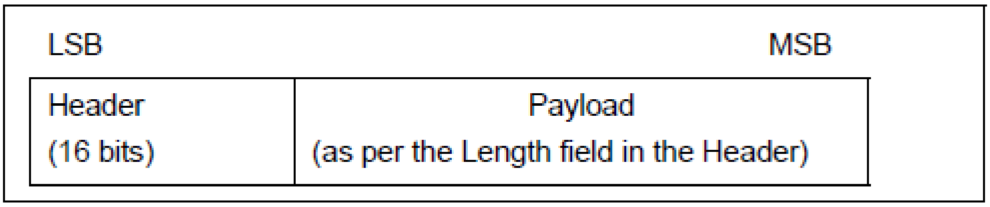
3.2.9.4 Link Layer state machine initialization

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

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

3.2.9.5 bls_ll_setAdvData

Please refer to "Core_v4.2" Page1282 LE Set Advertising Data Command.



LSB	MSB
Header (16 bits)	Payload (as per the Length field in the Header)

Figure3-30 Adv packet format in BLE stack

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

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

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

Note: The "data" pointer points to the starting address of the PDU, while the "len" indicates data length. The table below lists possible results for the return type "ble_sts_t".

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

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

In "826x ble remote" project of 826x BLE SDK, Adv PDU definition is shown as below. Please refer to "Data Type Specifcation" in BLE Spec "CSS v6" (Core Specification Supplement v6.0) for introduction to various fields.

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

};
```

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

3.2.9.6 bls_ll_setScanRspData

Please refer to "Core_v4.2" Page1283 LE Set Scan response Data Command.

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

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

Note: The "data" pointer points to the starting address of the PDU, while the "len" indicates data length. The table below lists possible results for the return type "ble_sts_t".

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

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

In "826x ble remote" project of 826x BLE SDK, definition of Scan response data is shown as below. Please refer to "Data Type Specifcation" in BLE Spec "CSS v6" (Core Specification Supplement v6.0) for introduction to various fields.

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

};
```

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

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

1) If some device only listens for Adv packets, the scanned device name is "thid".

2) If some device sends scan request after Adv packet is received, and reads the scan response, the scanned device name may be "tRemote".

User can configure device name in the two packets (Adv packet & scan response packet) as the same name, so that the scanned device name is consistent. Actually when Master reads device’s Attribute Table after connection is established, the obtained “gap device name” of device will be shown according to the configuration in Attribute Table. Please refer to Attribute Table section for details.

3.2.9.7 bls_ll_setAdvParam

Please refer to “Core_v4.2” Page1277 LE Set Advertising Parameters Command.

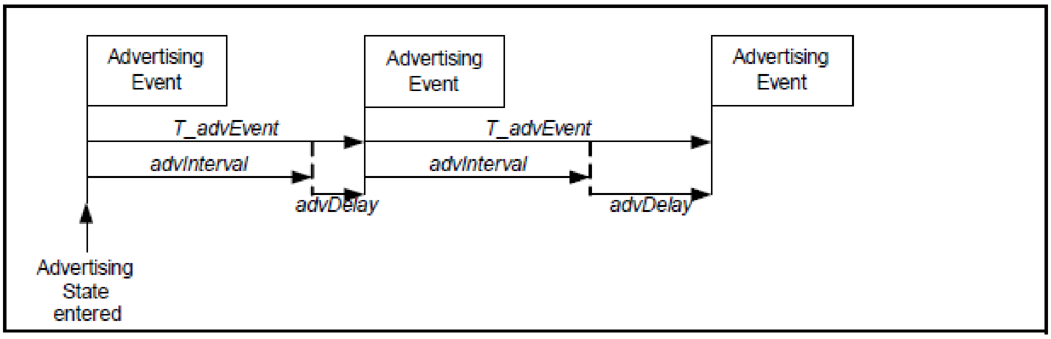


Figure3-31 Advertising Event in BLE stack

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

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

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

1) intervalMin & intervalMax：

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

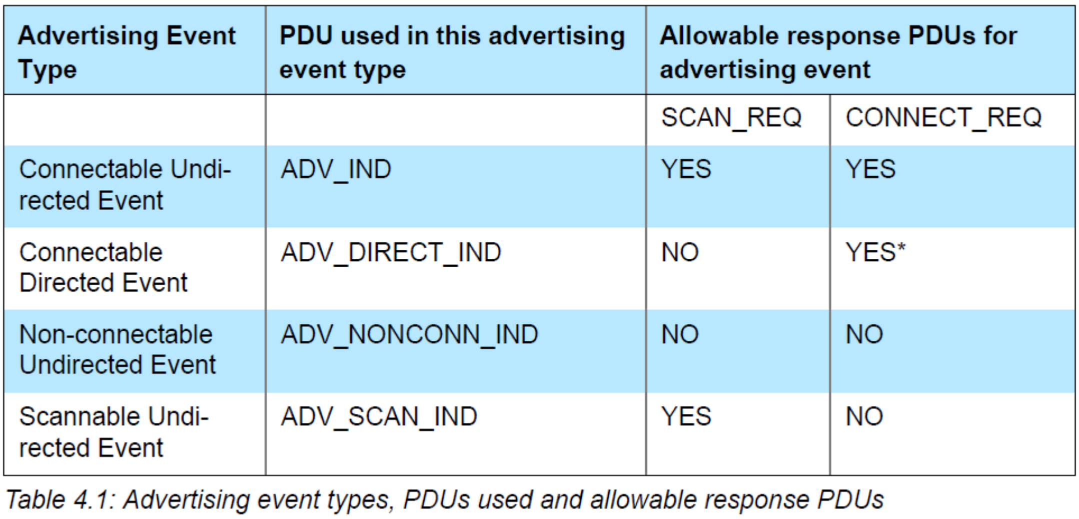
As required by BLE spec, it’s not recommended to set adv interval as fixed value;

in Telink BLE SDK, the eventual adv interval is random variable within the range of intervalMin ~ intervalMax. If intervalMin and intervalMax are set as same value, adv interval will be fixed as the intervalMin.

Adv packet type has limits to the setting of intervalMin and intervalMax. Please refer to “Core_v4.2” Page2609 4.4.2.2 Advertising Interval for details.

2) advType

As specified in “Core_v4.2” Page2607 Advertising State, the following four basic advertising event types are supported.



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

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

Figure3-32 Four adv events in BLE stack

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

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

The “Connectable Directed Event” supports two sub-types including “Low Duty Cycle Directed Advertising” and “High Duty Cycle Directed Advertising”. Therefore, five types of adv events are supported in all, as defined in “proj_lib/ble/ble_common.h”. Please refer to “Core_v4.2” Page2609 ~ Page2615.

```
/* Advertisement Type */
```

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

By default, the most common adv event type is "*ADV_TYPE_CONNECTABLE_UNDIRECTED*".

3) ownAddrType

It serves to specify MAC address type in adv packet.

There are two basic address types: public and random.

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

There are four optional values for "ownAddrType" to specify adv address type. Please refer to "Core_v4.2" Page1279 for address generation method.

```
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 most basic "OWN_ADDRESS_PUBLIC" is introduced herein. Actually the eventual address is the setting from API "blc_ll_initAdvertising_module(u8 *public_adr)" during MAC address initialization.

4) peerAddrType & *peerAddr

When advType is set as directed adv type (*ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY* or *ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY*), the "peerAddrType" and "*peerAddr" serve to specify the type and address of peer device MAC Address.

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

5) adv_channelMap

The "adv_channelMap" serves to set advertising channel. It can be selectable from channel 37, 38, 39 or combination.

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

#define            BLT_ENABLE_ADV_ALL
    (BLT_ENABLE_ADV_37 | BLT_ENABLE_ADV_38 | BLT_ENABLE_ADV_39)
```

6) advFilterPolicy

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

Filtering type options are shown as below. The "ADV_FP_NONE" can be selected if whitelist filter is not needed.

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

The table below lists possible values and reasons for the return value "ble_sts_t".

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	The intervalMin or intervalMax value does not meet the requirement of BLE spec.

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

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

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

ble_sts_t  bls_ll_setAdvFilterPolicy(u8 advFilterPolicy);
```

Please refer to the "bls_ll_setAdvParam" API for the parameters of the three APIs above.

Return value `ble_sts_t`:

1) "bls_ll_setAdvChannelMap" and "bls_ll_setAdvFilterPolicy" will always return "BLE_SUCCESS".
2) "bls_ll_setAdvInterval" will return "BLE_SUCCESS" or "HCI_ERR_INVALID_HCI_CMD_PARAMS".

3.2.9.8 bls_ll_setAdvEnable

Please refer to "Core_v4.2" Page1284 LE Set Advertising Enable Command.

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

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

1) In Idle state, by enabling Advertising, Link Layer will enter Advertising state.
2) In Advertising state, by disabling Advertising, Link Layer will enter Idle state.
3) In other states, Link Layer state won't be influenced by enabling or disabling Advertising.
4) `ble_sts_t` will always return "BLE_SUCCESS".

3.2.9.9 bls_ll_setAdvDuration

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

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

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

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

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

The return value “ble_sts_t” is shown as below.

ble_sts_t	**Value**	**ERR Reason**
BLE_SUCCESS	0	
HCI_ERR_INVALID_HCI_CMD_PARAMS	0x12	Duration Time can’t be configured for “ADV_TYPE_CONNECTABLE_DIRECTED_HIGH _DUTY”.

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

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

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

- If Slave stays in advertising state, when adv time reaches the preset 2s timeout, the “BLT_EV_FLAG_ADV_DURATION_TIMEOUT” will be triggered to execute corresponding callback function.
- If Slave is connected with Master when adv time is less than the 2s timeout

(suppose adv time is 0.5s), the timeout timing is not cleared but cached in bottom layer. When Slave stays in connection state for 1.5s (i.e. the preset 2s timeout moment is reached), since Slave won't check adv event timeout in connection state, the callback of "BLT_EV_FLAG_ADV_DURATION_TIMEOUT" won't be triggered. When Slave stays in connection state for certain duration (e.g. 10s), then terminates connection and returns to adv state, before it sends out the first adv packet, the Stack will regard current time exceeds the preset 2s timeout and trigger the callback of "BLT_EV_FLAG_ADV_DURATION_TIMEOUT". In this case, the callback triggering time largely exceeds the preset timeout moment.

3.2.9.10 blc_ll_setAdvCustomedChannel

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

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

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

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

blc_ll_setAdvCustomedChannel (8, 12, 22);

3.2.9.11 rf_set_power_level_index

826x BLE SDK supplies the API to set output power for BLE RF packet, as shown below.

void   rf_set_power_level_index (int level);

The "level" is selectable from the corresponding enum variable in the "proj_lib/rf_drv_826x.h". Take 8267 for example:

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

    };
```

Suppose it's needed to set the Tx power as the maximum value 8dbm:

```
        rf_set_power_level_index(RF_POWER_8dBm);
```

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

3.2.9.12 blc_ll_setScanParameter

Please refer to "Core_v4.2" Page1286 LE Set Scan Parameters Command.

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

Parameters:

1) scan_type

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

```
        typedef enum {
            SCAN_TYPE_PASSIVE = 0x00,
            SCAN_TYPE_ACTIVE,

          } scan_type_t;
```

2) scan_inetrval/scan window

"scan_interval" serves to set channel switch time in Scanning state (unit: 0.625ms).

"scan_window" is not processed in current Telink BLE SDK. Actual scan window is set as scan_interval.

3) ownAddrType

This parameter serves to specify MAC address type in adv packet.

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

```
    #define BLE_ADDR_RANDOM                 1
```

There are four optional values for "ownAddrType" to specify address type of scan req packet. Please refer to "Core_v4.2" Page1287 for address generation method.

```
    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 most basic "OWN_ADDRESS_PUBLIC" is introduced herein. Actually the eventual address is the setting from API "blc_ll_initScanning_module(u8 *public_adr)" during MAC address initialization.

4) filter_policy

Currently supported scan filter policies include:

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

"SCAN_FP_ALLOW_ADV_ANY" indicates Link Layer won't filter scanned adv packet, but directly report it to BLE Host.

"SCAN_FP_ALLOW_ADV_WL" indicates scanned adv packet must be in whitelist so that it can be reported to BLE Host.

The return value "ble_sts_t" is always "BLE_SUCCESS". Since API won't check rationality of parameters, user should pay attention to this point when setting parameters.

3.2.9.13 blc_ll_setScanEnable

Please refer to "Core_v4.2" Page1289 LE Set Scan Enable Command.

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

"scan_enable": 1 - Enable Scanning; 0 - Disable Scanning.

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

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

The return value "ble_sts_t" is shown as below.

ble_sts_t	**Value**	**ERR Reason**
BLE_SUCCESS	0	
HCI_ERR_CONTROLLER_BUSY	0x3A	Link Layer is in BLS_LINK_STATE_ADV /BLS_LINK_STATE_CONN state.

When "scan_type" is set as "active scan", and Scanning is enabled, for each device, scan_rsp will be read only once and reported to Host. Since after each "enable scanning", Controller will record and store scan_resp of different devices in a scan_rsp list, thus scan_req won't be sent to the device repeatedly.

In order to report scan_rsp of a device for multiple times, user can use "blc_ll_setScanEnable" to repeatedly set "Enable Scanning", since scan_rsp list will be cleared for each "Enable/Disable Scanning".

3.2.9.14 blc_ll_createConnection

Please refer to "Core_v4.2" Page1291 LE Create Connection Command.

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

1) scan_inetrval/scan window

"scan_interval" serves to set Scan channel switch time in Initiating state (unit: 0.625ms).

"scan_window" is not processed in current Telink BLE SDK. Actual scan window is set as scan_interval.

2) initiator_filter_policy

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

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

"INITIATE_FP_ADV_SPECIFY" indicates device address of connection is adr_type/mac;

"INITIATE_FP_ADV_WL" device connection depends on whitelist rather than adr_type/mac.

3) adr_type/ mac

When "initiator_filter_policy" is set as "INITIATE_FP_ADV_SPECIFY", the device with address type of adr_type (BLE_ADDR_PUBLIC or BLE_ADDR_RANDOM) and address of mac[5…0] will be connected.

4) own_adr_type

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

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

Only the most basic "OWN_ADDRESS_PUBLIC" is introduced herein. Actually the eventual address is the setting from API "blc_ll_initBasicMCU(u8 *public_adr)" during MAC address initialization.

5) conn_min/ conn_max/ conn_latency/ timeout

The four parameters specify connection parameters of Master role after connection is established. Since these parameters will be sent to Slave via "connection request", Slave will use the same connection parameters.

"conn_min" and "conn_max" specify the range of conn interval. In Telink BLE SDK, Master role Single Connection directly uses the value of "conn_min". Unit is 0.625ms.

"conn_latency" specifies connection latency, and generally it's set as 0.

"timeout" specifies connection supervision timeout in unit of 10ms.

6) ce_min/ ce_max

"ce_min"/"ce_max" are not processed in current SDK.

The return value is shown as below.

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
HCI_ERR_CONN_REJ_LIMITED_RESOURCES	0x0D	Link Layer is already in Initiating state, and won't receive new "create connection".
HCI_ERR_CONTROLLER_BUSY	0x3A	Link Layer is in Advertising state or Connection state.

3.2.9.15 blc_ll_setCreateConnectionTimeout

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

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

As introduced in Link Layer state machine, when "blc_ll_createConnection" triggers Idle state/Scanning state to enter Initiating state, if if connection fails to be established until "Initiate timeout" is triggered, it will exit Initiating state.

Whenever "blc_ll_createConnection" is invoked, by default current "Initiate timeout" is set as "connection supervision timeout *2" in SDK. User can modify this "Initiate timeout" as needed by invoking "blc_ll_setCreateConnectionTimeout" following "blc_ll_createConnection".

3.2.9.16 blm_ll_updateConnection

Please refer to "Core_v4.2" Page1302 LE Connection Update Command.

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

1) connection handle

   This parameter serves to specify connection to updata connection parameters.

2) conn_min/ conn_max/ conn_latency/ timeout

   The four parameters serve to specify new connection parameters. Currently "Master role single connection" directly use "conn_min" as new interval.

3) ce_min/ce_max

The two parameters are not processed currently.

The return value "ble_sts_t" is always "BLE_SUCCESS". Since API won't check rationality of parameters, user should pay attention to this point when setting parameters.

3.2.9.17 bls_ll_terminateConnection

Please refer to "Core_v4.2" Page2593 LL_TERMINATE_IND.

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

This API is used for BLE Slave device, and it only applies to Connection state Slave role.

In order to actively terminate connection, this API can be invoked by APP Layer to send a "Terminate" to Master in Link Layer. "reason" indicates reason for disconnection and it corresponds to the "ble_sts_t" defined in "ble_common.h". Please refer to "Core_v4.2" Page680 Error Code Descriptions.

If connection is not terminated due to system operation abnormity, generally APP layer specifies the "reason" as:

HCI_ERR_REMOTE_USER_TERM_CONN = 0x13

```
bls_ll_terminateConnection(HCI_ERR_REMOTE_USER_TERM_CONN);
```

In bottom-layer stack of Telink BLE SDK, this API is invoked only in one case to actively terminate connection: When data packets from peer device are decrpted, if an authentication data MIC error is detected, the "bls_ll_terminateConnection(HCI_ERR_CONN_TERM_MIC_FAILURE)" will be invoked to inform the peer device of the decryption error, and connection is terminated.

After Slave invokes this API to actively initiate disconnection, the event "BLT_EV_FLAG_TERMINATE" will be triggered. The terminate reason in the callback function of this event will be consistent with the reason manually configured in this API.

In Connection state Slave role, generally connection will be terminated successfully by invoking this API; however, in some special cases, the API may fail to terminate connection, and the error reason will be indicated by the return value "ble_sts_t". It's recommended to check whether the return value is "BLE_SUCCESS" when this API is invoked by APP layer.

ble_sts_t	**Value**	**ERR Reason**
BLE_SUCCESS	0	

ble_sts_t	**Value**	**ERR Reason**
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer is not in Connection state Slave role.
HCI_ERR_CONTROLLER_BUSY	0x3A	Controller busy (mass data are being transferred), this command cannot be accepted for the moment.

3.2.9.18 blm_ll_disconnect

Please refer to “Core_v4.2” Page2593 LL_TERMINATE_IND.

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

This API is used for BLE Master device and it only applies to Connection Master role.

This API is similar to the function of the API “API bls_ll_terminateConnection” of Slave role, except that a conn handle parameter is added. Since in Telink BLE SDK, Slave role design can only sustain single connection, while Master role supports multi connection, it’s necessary to specify connection handle of disconnect.

The return value is shown as below:

ble_sts_t	**Value**	**ERR Reason**
BLE_SUCCESS	0	
HCI_ERR_UNKNOWN_CONN_ID	0x02	Handle error, cannot find corresponding connection.
HCI_ERR_CONTROLLER_BUSY	0x3A	Controller busy (mass data are being transferred), this command cannot be accepted for the moment.

3.2.9.19 Get Connection Parameters

The following APIs serves to obtain current connection paramters including Connection Interval, Connection Latency and Connection Timeout (only apply to Slave role).

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

```
u16          bls_ll_getConnectionTimeout(void);
```

1) If return value is 0, it indicates current Link Layer state is None Conn state without connection parameters available.
2) The returned non-zero value indicates the corresponding parameter value.
   - Actual conn interval divided by 1.25ms will be returned by the API “bls_ll_getConnectionInterval”. Suppose current conn interval is 10ms, the return value should be 10ms/1.25ms=8.
   - Acutal Latency value will be returned by the API “bls_ll_getConnectionLatency”.
   - Actual conn timeout divided by 10ms will be returned by the API “bls_ll_getConnectionTimeout”. Suppose current conn timeout is 1000ms, the return value would be 1000ms/10ms=100.

3.2.9.20 blc_ll_getCurrentState

The API below serves to obtain current Link Layer state.

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

User can invoke the “bls_ll_getCurrentState()” in APP layer to judge current state, e.g.

if(bls_ll_getCurrentState() == BLS_LINK_STATE_ADV)

if(bls_ll_getCurrentState() == BLS_LINK_STATE_CONN)

3.2.9.21 blc_ll_getLatestAvgRSSI

The API serves to obtain latest average RSSI of connected peer device after Link Layer enters Slave role or Master role.

```
u8          blc_ll_getLatestAvgRSSI(void)
```

The return value is u8-type rssi_raw, and the real RSSI should be: rssi_real = rssi_raw- 110. Suppose the return value is 50, rssi = -60 db.

3.2.9.22 Whitelist & Resolvinglist

As introduced above, “filter_policy” of Advertising/Scanning/Initiating state involves Whitelist, and actual operation may depend on devices in Whitelist. Actually Whitelist contains two parts: Whitelist and Resolvinglist.

User can check whether address type of peer device is RPA (Resolvable Private Address) via "peer_addr_type" and "peer_addr". The API below can be invoked directly.

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

Only non-RPA address can be stored in whitelist. In current SDK, whitelist can store up to four devices.

```
#define    MAX_WHITE_LIST_SIZE                4
```

The API below serves to reset whitelist:

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

The return value is "BLE_SUCCESS".

The API below serves to add a device into whitelist:

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

The return value is shown as below.

ble_sts_t	**Value**	**ERR Reason**
BLE_SUCCESS	0	Add success
HCI_ERR_MEM_CAP_EXCEEDED	0x07	Whitelist is already full, add failure

The API below serves to delete a device from whitelist:

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

The return value is "BLE_SUCCESS".

RPA (Resolvable Private Address) device needs to use Resolvinglist. To save RAM space, "Resolvinglist" can store up to two devices in current SDK.

```
#define    MAX_WHITE_IRK_LIST_SIZE         2
```

The API below serves to reset Resolvinglist.

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

The return value is “BLE_SUCCESS”.

The API below serves to enable/disable device address resolving for Resolvinglist.

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

The API below serves to add device using RPA address into Resolvinglist.

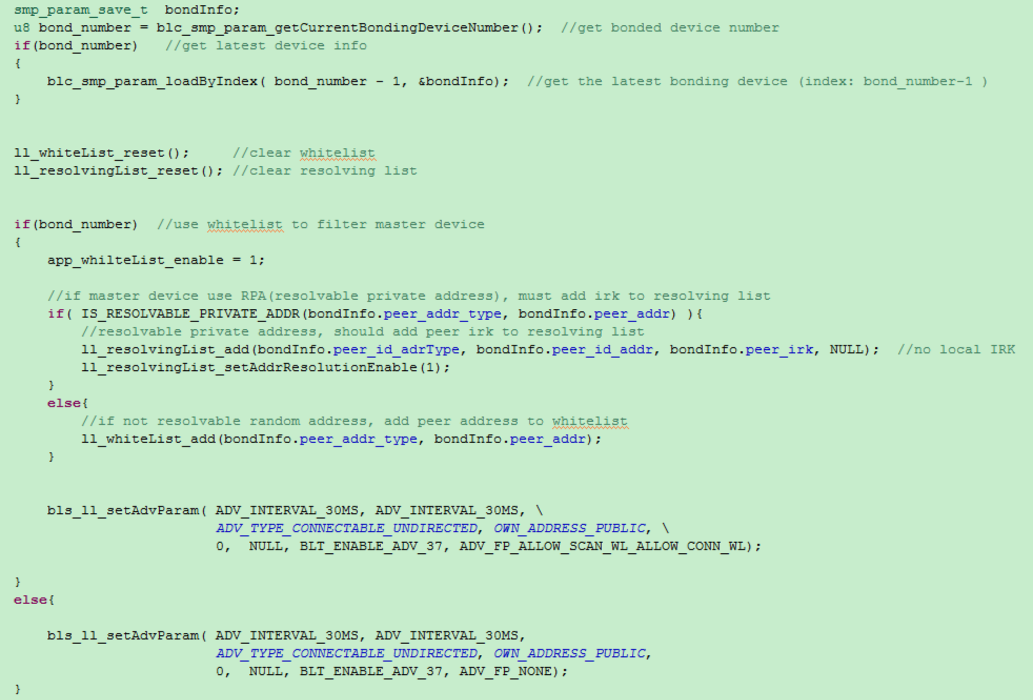
```
ble_sts_t  ll_resolvingList_add(u8 peerIdAddrType, u8 *peerIdAddr,
                                 u8 *peer_irk, u8 *local_irk);
```

peerIdAddrType/ peerIdAddr and peer-irk indicate identity address and irk declared by peer device. These information will be stored into flash during pairing encryption process, and corresponding interfaces to obtain the info are available in SMP part. “local_irk” is not processed in current SDK, and it can be set as “NULL”.

The API below serves to delete a RPA device from Resolvinglist.

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

For usage of address filter based on Whitelist/Resolvinglist, please refer to “TEST_WHITELIST” in 826x feature test.



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


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


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

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


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

}
else{

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

Figure3-33 Whitelist/Resolvinglist address filter

3.2.9.23 blc_ll_set_CustomedAdvScanAccessCode

In some snerio, user need to modify accesscode, SDK now provides API `blc_ll_set_CustomedAdvScanAccessCode()` for this.

Original function:

```
static inline void  blc_ll_set_CustomedAdvScanAccessCode(u32 accss_code);
```

User can call this function when accesscode need to be modified.

3.2.10 2M PHY Supported

2M PHY is a new Link layer feature supported since BLE Core 5.0. Please refer to BLE Spec《Core_v5.0》(Vol 6/Part B/ Link Layer Specification) for detail.

2M PHY use 3 PDU(LL_PHY_REQ/LL_PHY_RSP/LL_PHY_UPDATE_IND) to interact between master and slave contollers, and to set the transmit frenquency of RF transceiver. 2M PHY feature is only avialble when the connection is estabilished. Both master and slave can enable this process, when master enables this process, master will send LL_PHY_REQ PDU, slave will send LL_PHY_RSP PDU to answer, this is master and slave change the first priority PHY they support, then, master will sent LL_PHY_UPDATE_IND, when the instance reaches, master and slave will use new PHY to send/receive data; if slave enable this process, slave will send LL_PHY_REQ PDU, master will send LL_PHY_UPDATE_IND to answer, and when instance reaches, master and slave will use new PHY to send/receive data.

SDK3.3.0 also supports 2M PHY, but it is only applicable for 8269 SoC series. The function is default enabled. SDK provides APIs for users to support 2M PHY function, details as following. Please be noted, SDK supports only symmetrical PHY, i.e., RX PHY and TX PHY should be set as the same.

◆ For Slave, please refer to Demo “8269_feature_test”,

Define macro in vendor/8269_feature_test/app_config.h

```
#define FEATURE_TEST_MODE   TEST_2M_PHY_CONNECTION
```

◆ For master, please refer to Demo “8269_master_kma_dongle”.

All equipments supporting 2M PHY are compatible with Telink Slave equipment, users can choose freely.

If user use Telink SDK, please add API

`blc_ll_init2MPhy_feature ();`

in function `void user_init(void)` in vendor/8269_master_kma_dongle/app.c(disabled by default in SDK).

2M PHY API detail as following:

```
1. void    blc_ll_init2MPhy_feature(void);
```

This API initialize 2M PHY related parameters.

If PHY Update is started by master, user only need to call this initialization function, and BLE stack will do the rest.

2. void 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  );
```

Function description as following:

Parameter	Description
connHandle	**Connection Handle, for slave, it is BLS_CONN_HANDLE, for master, it is BLM_CONN_HANDLE**
all_phys	Set if there are preferred TX and RX PHY. This is emumerated parameter, common value is *PHY_TRX_PREFER.* Refer to le_phy_prefer_mask_t definition for other value.
tx_phys	Preferred TX PHY. This is emumerated parameter, available values are *PHY_PREFER_1M and PHY_PREFER_2M.* Whether this parameter is used depends on whether all_phys defines preferred tx phy.
rx_phys	Preferred RX PHY. This is emumerated parameter, available values are *PHY_PREFER_1M and PHY_PREFER_2M.* Whether this parameter is used depends on whether all_phys defines preferred rx phy.

If PHY update is started by slave, then user need to call this API to set preferred PHY.

3. void bls_app_registerEventCallback(u8 e, blt_event_callback_t p)

If user want to execute some user-defined operation after PHY Update, they can register callback function. Parameter: BLT_EV_FLAG_PHY_UPDATE. Check ”Telink defined event” for detail of this API usage.

3.2.11 Data Length Extension

BLE spec supports data length extension (DLE) for core_4.2 and abover verison.

This BLE SDK supports data length extension on Link Layer, and rf_len supports up to BLE spec max length 251 bytes.

Please refer to 《Core_v5.0》(Vol 6/Part B/2.4.2.21 “LL_LENGTH_REQ and

LL_LENGTH_RSP”) for detail.

Please follow the following steps to use data length extension function.

1) Set suitable TX & RX fifo size

Long packets need bigger TX&RX fifo size, these fifo will take large SRAM space, sol user should set suitable fifo size to avoid wasting SRAM space.

Sending long packet need bigger TX fifo size. TX fifo size should be at least bigger than TX rf_len plus 12, and it must be configured as 4 KB aligned. E.g.:

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

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

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

Recieving long packet need bigger RX fifo size. RX fifo size should be at least bigger than TX rf_len plus 24, and it must be configured as 16 KB aligned. E.g.:

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

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

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

If both TX and RX max supporting length is 200 bytes, the configuration should be:

MYFIFO_INIT(blt_txfifo, 212,   8);

MYFIFO_INIT(blt_rxfifo, 224,   8);

2) data length exchange

Before sending/receiving long packet, data length exchange operation must succeed in BLE connection.

data length exchange operation is the interaction of LL_LENGTH_REQ and LL_LENGTH_RSP on Link Layer. Either slave or master can start this by sending LL_LENGTH_REQ, and the other part will answer LL_LENGTH_RSP. By this interaction of the 2 packets, master and slave will be acknowledged of the max length of TX and RX packet of each other, the smaller value will define the max sending/receiving packet length.

No matter which side start LL_LENGTH_REQ, when data length exchange operation succeeds, SDK will generate BLT_EV_FLAG_DATA_LENGTH_EXCHANGE callback (in case the callback has been registered), users can refer to "Telink defined event” for the definition of this callback function’s parameters.

In this BLT_EV_FLAG_DATA_LENGTH_EXCHANGE callback functions, user can get the final max length of TX packet and RX packet respectively.

When 826x severs as BLE slave equipment, master may or may not actively start LL_LENGTH_REQ. If master does not actively start LL_LENGTH_REQ, then slave need to start LL_LENGTH_REQ. SDK provides the following API to start LL_LENGTH_REQ.

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

in which, configure opcode as LL_LENGTH_REQ, configure maxTxOct as the max supported TX packet length, e.g, when max TX packet length is 200 bytes, the configuration is as following:

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

Slave is unaware of whether master start LL_LENGTH_REQ, here is one way to check: register BLT_EV_FLAG_DATA_LENGTH_EXCHANGE callback, after connention is established, enable a software timer (e.g, 2S) to time, if the callback is not triggered when the timing is over, it means master does not start LL_LENGTH_REQ, in this case, slave need to call API blc_ll_exchangeDataLength to start LL_LENGTH_REQ actively.

3) MTU size exchange

MTU size exchange operation must also succeed to ensure max MTU size is effective, so that to guarantee the opposite equipment can deal with long packet. MTU size should be equal or bigger than max TX&RX packet length. User can set MTU size by calling `blc_att_setRxMtuSize()` when initialazition, otherwise the default value is 23 bytes.

For MTU size exchange realization, please refer to " ATT & GATT" section, or 826x_feature_test demo.

4) Sending/receiving long packet operation

Please refer to “ATT & GATT” first, including Handle Value Notification and Handle Value Indication， Write request and Write Command.

After the 3 operation described above succeed, user can sending/receiving long packet.

To send long packet, call Handle Value Notification and Handle Value Indication related API in ATT layer, as shown below, just input the sending data address and length in formal parameter *p” and "len” respectively.

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

To receive long packet, call callback function "w” of Write request and Write Command, use the data that the formal parameter pointer point to.

3.3 L2CAP

As specified in BLE Spec, L2CAP is mainly used for data transfer between Controller and Host. Most work are finished in stack bottom layer with little involvement of user. User only needs to invoke the following APIs to set correspondingly.

3.3.1 Register L2CAP data processing function

In BLE SDK architecture, Controller transfers data with Host via HCI. Data from HCI to Host will be processed in L2CAP layer first. The API below serves to register this processing function.

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

In BLE Slave applications such as 826x remote/826x module, the function to process data of Controller in L2CAP layer of SDK is shown as below:

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

This function is already implemented in stack, which it will analyze the received data and transfer the data to ATT or SMP.

Initialization:

```
blc_l2cap_register_handler (blc_l2cap_packet_receive);
```

In 826x master kma dongle, APP layer contains BLE Host function, and its processing function is supplied in source code for user reference:

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

Initialization:

```
blc_l2cap_register_handler (app_l2cap_handler);
```

In 826x hci, only Slave controller is implemented. The function “blc_hci_sendACLData2Host” serves to transmit data of controller to BLE Host device via hardware interface such as UART/USB.

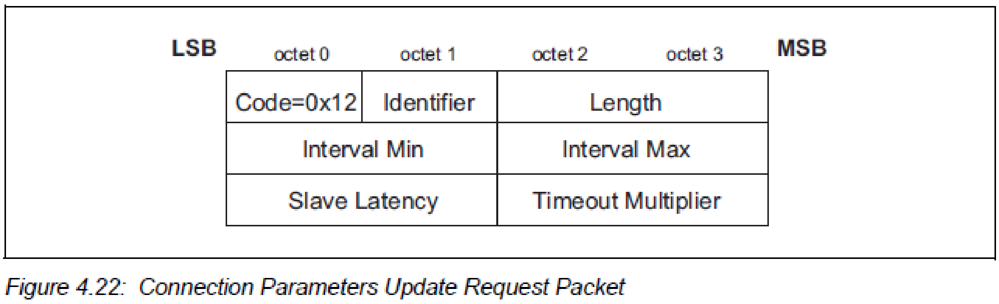
int blc_hci_sendACLData2Host (u16 handle, u8 *p)

Initialization: blc_l2cap_register_handler (blc_hci_sendACLData2Host);

3.3.2 Update connection parameters

3.3.2.1 Slave requests for connection parameter update

In BLE stack, Slave can actively apply for a new set of connection parameters by sending a “CONNECTION PARAMETER UPDATE REQUEST” command to Master in L2CAP layer. The figure below shows the command format. Please refer to “Core_v4.2” Page 1775 CONNECTION PARAMETER UPDATE REQUEST.



LSB	octet 0	octet 1	octet 2	octet 3	MSB
	Code=0x12	Identifier	Length		
	Interval Min		Interval Max		
	Slave Latency		Timeout Multiplier		

Figure 4.22: Connection Parameters Update Request Packet

Figure3-34 Connection Para update Req format in BLE stack

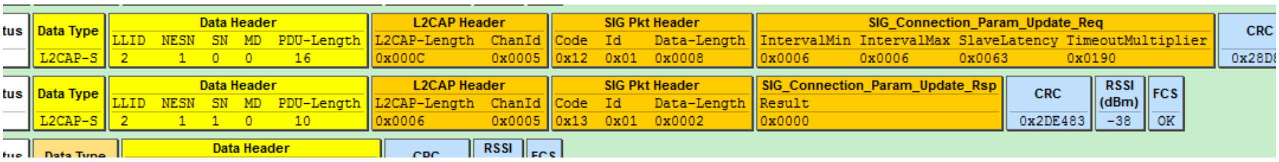
826x BLE SDK supplies an API in L2CAP layer for Slave to send a “CONNECTION PARAMETER UPDATE REQUEST” command to Master and actively apply for a new set of connection parameters.

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

*Note: The four parameters of this API correspond to the parameters in the “data” field of the “CONNECTION PARAMETER UPDATE REQUEST”. The “min_interval” and “max_interval” are the actual interval time divided by 1.25ms (e.g. for 7.5ms connection interval, the value should be 6); the “timeout” is actual supervision timeout divided by 10ms (e.g. for 1s timeout, the value should be 100).

Application example: Slave requests for new connection parameters when connection is established.

```
void task_connect (u8 e, u8 *p)

{

    bls_l2cap_requestConnParamUpdate    (6,    6,    99,    400);
    //interval=7.5ms latency=99 timeout=4s

}
```



```
tus | Data Type | Data Header                          | L2CAP Header          | SIG Pkt Header            | SIG_Connection_Param_Update_Req                                  | CRC
    |           | LLID  NESN  SN  MD  PDU-Length       | L2CAP-Length  ChanId  | Code  Id    Data-Length   | IntervalMin  IntervalMax  SlaveLatency  TimeoutMultiplier        |
    | L2CAP-S   | 2        1   0   0      16           | 0x000C        0x0005  | 0x12  0x01  0x0008        | 0x0006       0x0006       0x0063        0x0190                   | 0x28D
tus | Data Type | Data Header                          | L2CAP Header          | SIG Pkt Header            | SIG_Connection_Param_Update_Rsp | CRC      | RSSI  | FCS
    |           | LLID  NESN  SN  MD  PDU-Length       | L2CAP-Length  ChanId  | Code  Id    Data-Length   | Result                          |          | (dBm) |
    | L2CAP-S   | 2        1   1   0      10           | 0x0006        0x0005  | 0x13  0x01  0x0002        | 0x0000                          | 0x2DE483 | -38   | OK
tus | Data Type | Data Header                          | CRC | RSSI | FCS
```

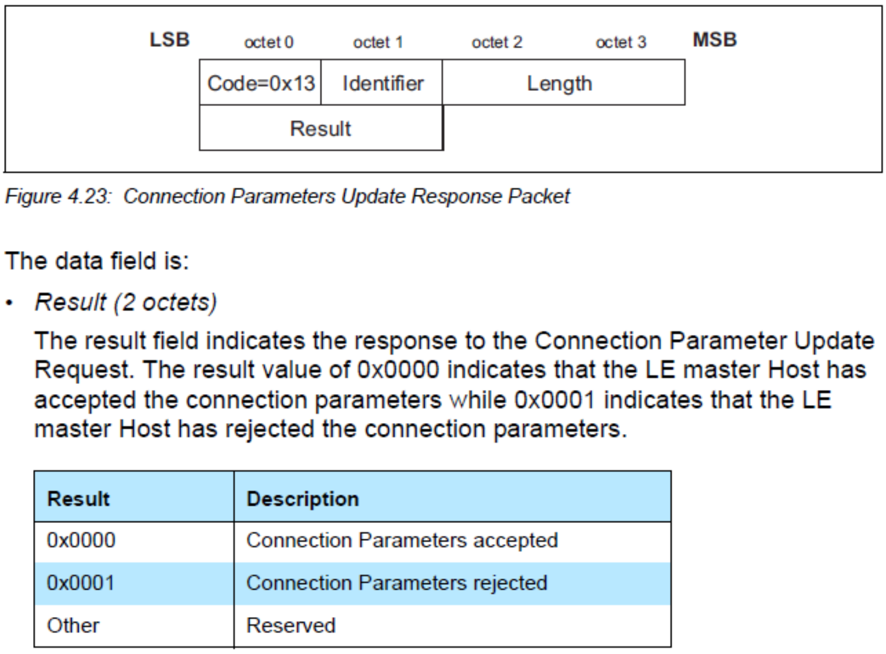
Figure3-35 BLE sniffer packet sample: conn para update request & response

3.3.2.2 Master responds to connection parameter update request

After Master receives the “CONNECTION PARAMETER UPDATE REQUEST” command from Slave, it will respond with a “CONNECTION PARAMETER UPDATE RESPONSE” command. Please refer to “Core_v4.2” Page 1776 CONNECTION PARAMETER UPDATE RESPONSE.

The figure below shows the command format: if “result” is “0x0000”, it indicates the request command is accepted; if “result” is “0x0001”, it indicates the request command is rejected. Whether actual Android/iOS device will accept or reject the connection parameter update request is determined by corresponding BLE Master. User can refer to Master compatibility test.

As shown in Figure3-35, Master accepts the request.



LSB	octet 0	octet 1	octet 2	octet 3	MSB
	Code=0x13	Identifier	Length		
	Result				

Figure 4.23: Connection Parameters Update Response Packet

The data field is:

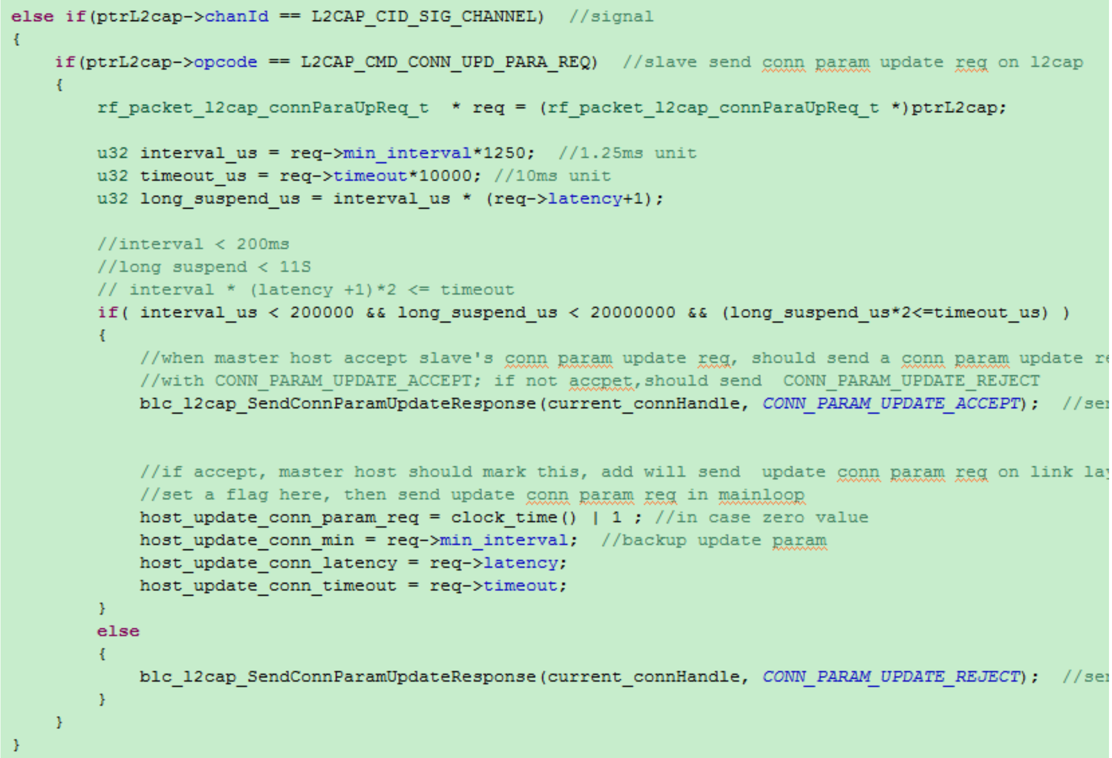
- *Result (2 octets)*

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

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

Figure3-36 conn para update rsp format in BLE stack

Following shows demo code to process connection parameter update request of Slave in Telink 826x master kma dongle.



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

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

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


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

After "L2CAP_CMD_CONN_UPD_PARA_REQ" is received in "L2CAP_CID_SIG_CHANNEL", it will read interval_min (used as eventual interval), supervision timeout and long suspend time (interval * (latency +1)), and check the rationality of these data. If interval < 200ms, long suspend time<20s and supervision timeout >= 2* long suspend time, this request will be accepted; otherwise this request will be rejected. User can modify as needed based on this simple demo design.

No matter whether parameter request of Slave is accepted, the API below can be invoked to respond to this request.

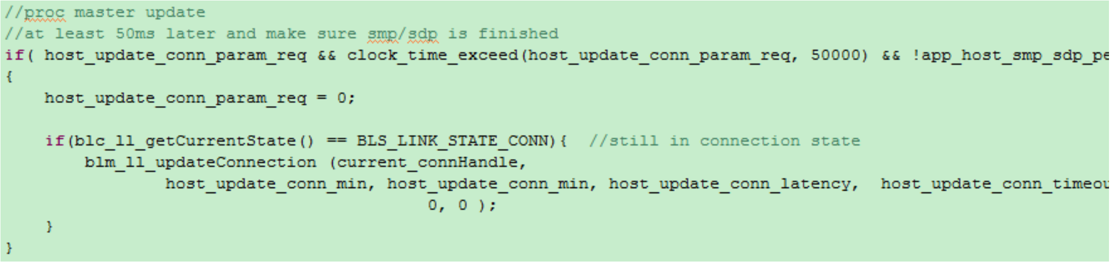
```
void  blc_l2cap_SendConnParamUpdateResponse( u16 connHandle,
                                                          int result);
```

"connHandle" indicates current connection ID.

"result" has two options to indicate "accept" and "reject", respectively.

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

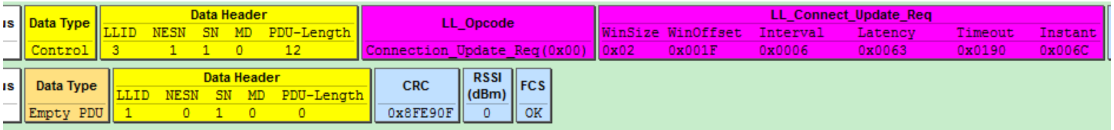
If 826x Master accepts request of Slave, it must send a update cmd to Controller via the API "blm_ll_updateConnection" within certain duration. In demo code, "host_update_conn_param_req" is used as mark, and a 50ms delay is set in mainloop to initiate this update.



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

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

3.3.2.3 Master updates connection parameters in Link Layer

After Master responds with “conn para update rsp” to accept the “conn para update req” from Slave, Master will send a “LL_CONNECTION_UPDATE_REQ” command in Link Layer.



ıs	Data Type	Data Header					LL_Opcode	LL_Connect_Update_Req					
		LLID	NESN	SN	MD	PDU-Length		WinSize	WinOffset	Interval	Latency	Timeout	Instant
	Control	3	1	1	0	12	Connection_Update_Req(0x00)	0x02	0x001F	0x0006	0x0063	0x0190	0x006C

ıs	Data Type	Data Header					CRC	RSSI (dBm)	FCS
		LLID	NESN	SN	MD	PDU-Length			
	Empty PDU	1	0	1	0	0	0x8FE90F	0	OK

Figure3-37 BLE sniffer packet sample: ll conn update req

Slave will mark the final parameter as the instant value of Master after it receives the update request. When the instant value of Slave reaches this value, connection parameters are updated, and the callback of the event “BLT_EV_FLAG_CONN_PARA_UPDATE” is triggered.

The “instant” indicates connection event count value maintained by Master and Slave, and it ranges from 0x0000 to 0xffff. During a connection, Master and Slave should always have consistent “instant” value. When Master sends “conn_req” and establishes connection with Slave, Master switches state from scanning to connection, and clears the “instant” of Master to “0”. When Slave receives the “conn_req”, it switches state from advertising to connection, and clears the instant of Slave to “0”. Each connection packet of Master and Slave is a connection event. For the first connection event after the “conn_req”, the instant value is “1”; for the second connection event, the instant value is 2, and so on.

When Master sends a “LL_CONNECTION_UPDATE_REQ”, the final parameter “instant” indicates during the connection event marked with “instant”, Master will use the values corresponding to the former connection parameters of the “LL_CONNECTION_UPDATE_REQ” packet. After the “LL_CONNECTION_UPDATE_REQ” is received, the new connection parameters will be used during the connection event when the instant of Slave equals the declared instant of Master, thus Slave and Master can finish switch of connection parameters synchronously.

3.4 ATT & GATT

3.4.1 GATT basic unit “Attribute”

GATT defines two roles: Server and Client. In 826x BLE SDK, Slave is Server, and corresponding Android/iOS device is Client. Server needs to supply multiple Services for Client to access.

Each Service of GATT consists of multiple Attributes, and each Attribute contains certain information.

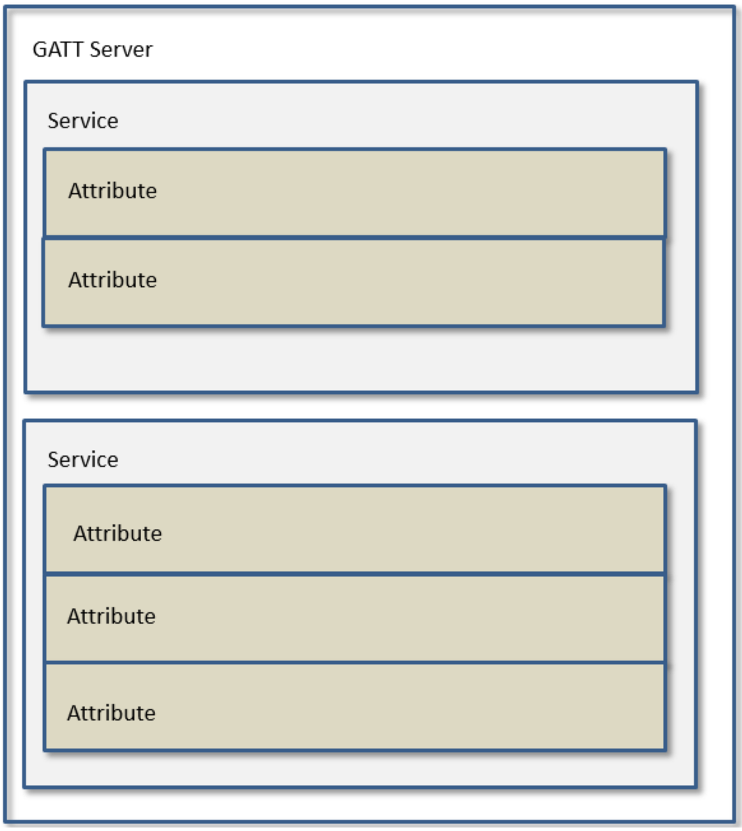


Figure3-38 GATT service containing Attribute group

The basic contents of Attribute are shown as below:

1) Attribute Type: UUID

The UUID is used to identify Attribute type, and its total length is 16 bytes. In BLE standard protocol, the UUID length is defined as two bytes, since Master devices follow the same method to transform 2-byte UUID into 16 bytes.

When standard 2-byte UUID is directly used, Master should know device types indicated by various UUIDs. 826x BLE stack defines some standard UUIDs in “proj_lib/ble_l2cap/hids.h”, “proj_lib/ble_l2cap/gatt_uuid.h” and “proj_lib/ble_l2cap/service.h”.

Telink proprietary profiles (OTA, MIC, SPEAKER, and etc.) are not supported in standard Bluetooth. The 16-byte proprietary device UUIDs are defined in “proj_lib/ble_l2cap/service.h”.

2) Attribute Handle

Slave supports multiple Attributes which compose an Attribute Table. In Attribute Table, each Attribute is identified by an Attribute Handle value. After connection is established, Master will analyze and obtain the Attribute Table of Slave via “Service Discovery” process, then it can identify Attribute data via the Attribute Handle during data transfer.

3) Attribute Value

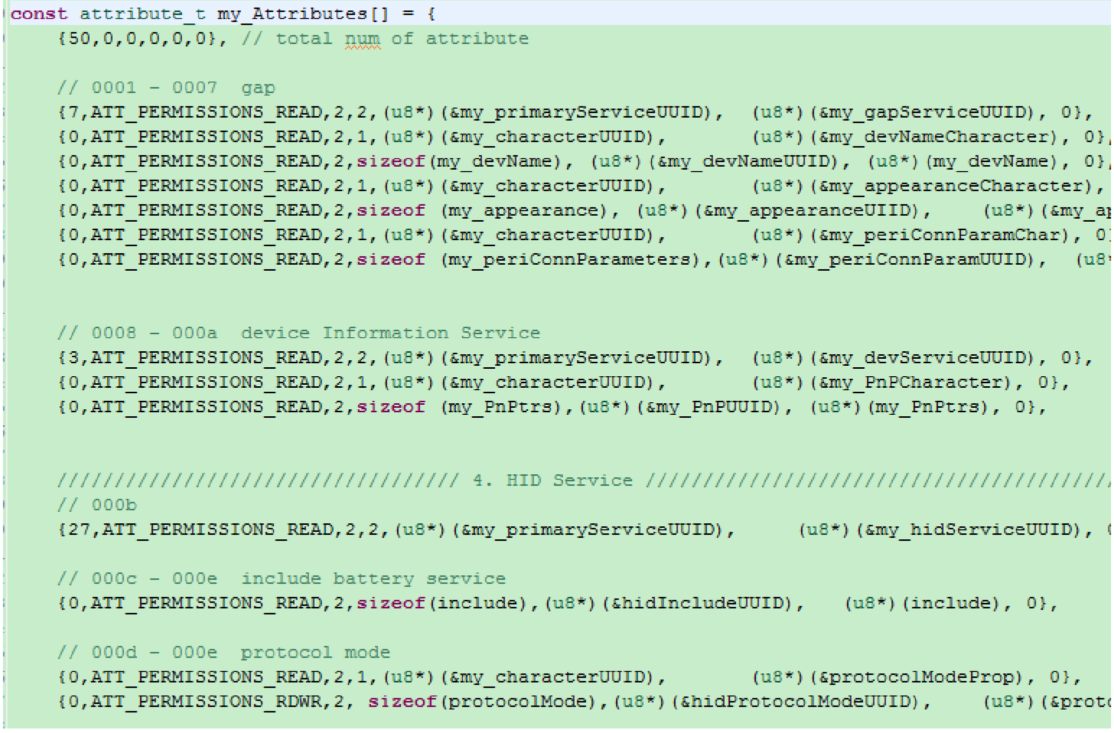
Attribute Value corresponding to each Attribute is used as request, response, notification and indication data. In 826x BLE stack, Attribute Value is indicated by one pointer and the length of the area pointed by the pointer.

3.4.2 Attribute and ATT Table

To implement GATT Service on Slave, an Attribute Table is defined in 826x BLE SDK and it consists of multiple basic Attributes. Attribute definition is shown as below.

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

Attribute Table code is available in “app_att.c”, as shown below:



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

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


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


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

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

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

Figure3-39 Attribute Table in 826x BLE SDK

*Note: The key word "const" is added before Attribute Table definition:

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

By adding the "const", the compiler will store the array data to flash rather than RAM, while all contents of the Attribute Table defined in flash are read only and not modifiable.

3.4.2.1 attNum

The "attNum" supports two functions.

1) The "attNum" can be used to indicate the number of valid Attributes in current Attribute Table, i.e. the maximum Attribute Handle value. This number is only used in the invalid Attribute item 0 of Attribute Table array.

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

"attNum = 50" indicates there are 50 valid Attributes in current Attribute Table.

In BLE, Attribute Handle value starts from 0x0001 with increment step of 1, while the array index starts from 0. When this virtual Attribute is added to the Attribute Table, each Attribute index equals its Attribute Handle value. After the Attribute Table is defined, Attribute Handle value of an Attribute can be obtained by counting its index in current Attribute Table array.

The final index is the number of valid Attributes (i.e. attNum) in current Attribute Table. In current SDK, the attNum is set as 50; if user adds or deletes any Attribute, the attNum needs to be modified correspondingly.

2) The "attNum" can also be used to specify Attributes constituting current Service.

The UUID of the first Attribute for each Service must be "GATT_UUID_PRIMARY_SERVICE(0x2800)"; the first Attribute of a Service sets "attNum" and it indicates following "attNum" Attributes constitute current Service.

As shown in Figure3-39, for the gap service, the Attribute with UUID of "GATT_UUID_PRIMARY_SERVICE" sets the "attNum" as 7, it indicates the seven Attributes from Attribute Handle 1 to Attribute Handle 7 constitute the gap service.

Similarly, for the HID service, the "attNum" of the first Attribute is set as 27, and it indicates the following 27 Attributes constitute the HID service.

Except for Attribute item 0 and the first Attribute of each Service, attNum values of all Attributes must be set as 0.

3.4.2.2 perm

The "perm" is the simplified form of "permission" and it serves to specify access permission of current Attribute by Client.

The "perm" of each Attribute is configurable as one or combination of following values.

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

#define ATT_PERMISSIONS_ENCRYPT_WRITE       0x80
```

3.4.2.3 uuid and uuidLen

As introduced above, UUID supports two types: BLE standard 2-byte UUID, and Telink proprietary 16-byte UUID. The "uuid" and "uuidLen" can be used to describe the two UUID types simultaneously.

The "uuid" is an u8-type pointer, and "uuidLen" specifies current UUID length, i.e. the uuidLen bytes starting from the pointer are current UUID. Since Attribute Table and all UUIDs are stored in flash, the "uuid" is a pointer pointing to flash.

1) BLE standard 2-byte UUID：

E.g. For the Attribute “devNameCharacter” with Attribute Handle of 2, related code is shown as below:

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

“UUID=0x2803” indicates “character” in BLE and the “uuid” points to the address of “my_devNameCharacter” in flash. The “uuidLen” is 2. When Master reads this Attribute, the UUID would be “0x2803”.

2) Telink proprietary 16-byte UUID：

E.g. For the Attribute MIC of audio, related code is shown as below:

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

The “uuid” points to the address of “my_MicData” in flash, and the “uuidLen” is 16. When Master reads this Attribute, the UUID would be “0x000102030405060708090a0b0c0d2b18”.

3.4.2.4 pAttrValue and attrLen

Each Attribute corresponds to an Attribute Value. The “pAttrValue” is an u8-type pointer which points to starting address of Attribute Value in RAM/Flash, while the “attrLen” specifies the data length. When Master reads the Attribute Value of certain Attribute from Slave, the “attrLen” bytes of data starting from the area (RAM/Flash) pointed by the “pAttrValue” will be read by 826x BLE SDK to Master.

Since UUID is read only, the “uuid” is a pointer pointing to flash; while Attribute Value may involve write operation into RAM, so the “pAttrValue” may points to RAM or flash.

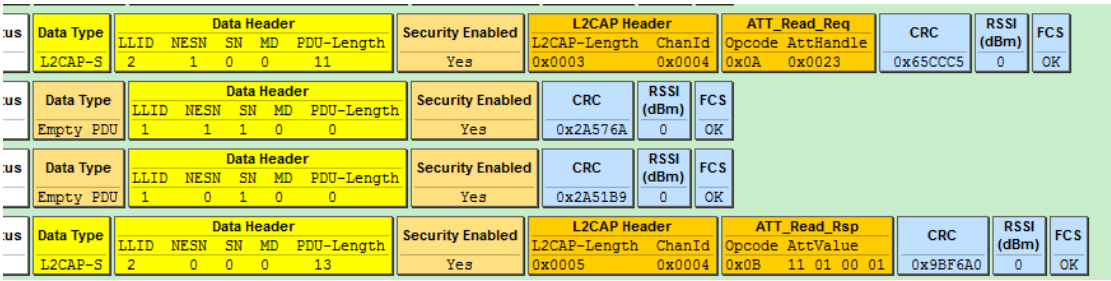
E.g. For the Attribute hid Information with Attribute Handle of 35, related code is as shown below:

```
const u8 hidInformation[] =
{
    U16_LO(0x0111), U16_HI(0x0111),     // bcdHID (USB HID version)，0x11,0x01
```

```
        0x00,                              // bCountryCode
        0x01                              // Flags
    };
    {0,2, sizeof(hidInformation), sizeof(hidInformation),(u8*)(&hidinformationUUID),
                                                       (u8*)(hidInformation), 0},
```

In actual application, the key word “const” can be used to store the read-only 4-byte hid information “0x01 0x00 0x01 0x11” into flash. The “pAttrValue” points to the starting address of hidInformation in flash, while the “attrLen” is the actual length of hidInformation. When Master reads this Attribute, “0x01000111” will be returned to Master correspondingly.

Figure3-40 shows a packet example captured by BLE sniffer when Master reads this Attribute. Master uses the “ATT_Read_Req” command to set the target AttHandle as 0x23 (35), corresponding to the hid information in Attribute Table of SDK.



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

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

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

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

Figure3-40 BLE sniffer packet sample when Master reads hidInformation

E.g. For the Attribute “battery value” with Attribute Handle of 40, related code is as shown below:

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

In actual application, the “my_batVal” indicates current battery level and it will be updated according to ADC sampling result; then Slave will actively notify or Master will actively read to transfer the “my_batVal” to Master. The starting address of the “my_batVal” stored in RAM will be pointed by the “pAttrValue”.

3.4.2.5 Callback function w

The callback function w is write function with prototype as below:

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

User must follow the format above to define callback write function. The callback function w is optional, i.e. for an Attribute, user can select whether to set the callback write function as needed (null pointer 0 indicates not setting callback write function).

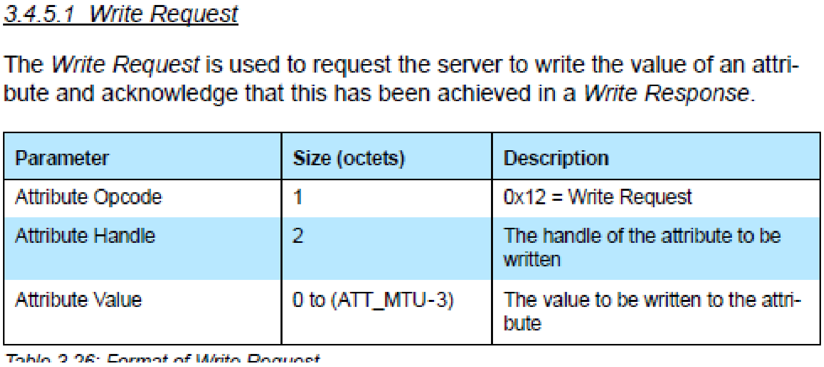
The trigger condition for callback function w is: When Slave receives any Attribute PDU with Attribute Opcode as shown below, Slave will check whether the callback function w is set.

1) opcode = 0x12, Write Request, refer to “Core_v4.2” Page2191.

2) opcode = 0x52, Write Command, refer to “Core_v4.2” Page2193.

After Slave receives a write command above, if the callback function w is not set, Slave will automatically write the area pointed by the “pAttrValue” with the value sent from Master, and the data length equals the “l2capLen” in Master packet format minus 3; if the callback function w is set, Slave will execute user-defined callback function w after it receives the write command, rather than writing data into the area pointed by the “pAttrValue”. Note: Only one of the two write operations is allowed to take effect.

By setting the callback function w, user can process Write Request and Write Command in ATT layer of Master. If the callback function w is not set, user needs to evaluate whether the area pointed by the “pAttrValue” can process the command (e.g. If the “pAttrValue” points to flash, write operation is not allowed; or if the “attrLen” is not long enough for Master write operation, some data will be modified unexpectedly.)



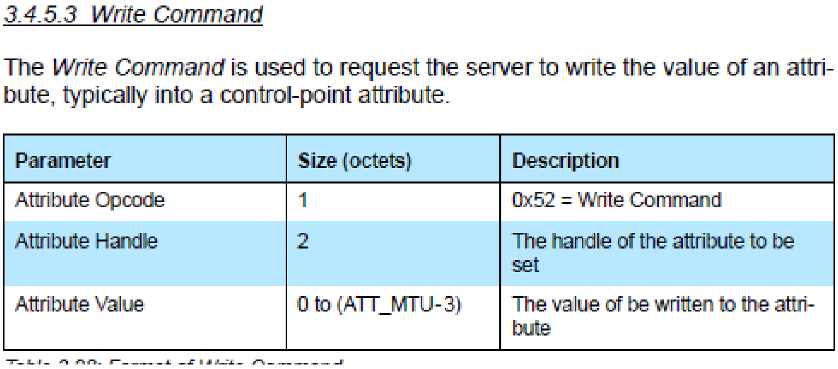
3.4.5.1 *Write Request*

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

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

Table 3.26: Format of Write Request

Figure3-41 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

Table 3.28: Format of Write Command

Figure3-42 Write Command in BLE stack

The void-type pointer “p” of the callback function w points to the value of Master write command. Actually “p” points to a memory area, the value of which is shown as the following structure.

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

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

"p" points to "dma_len", valid length of data is l2cap minus 3, and the first valid data is `pw->dat[0]`.

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

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

    return 1;
}
```

The structure "rf_packet_att_data_t" above is available in "proj_lib/ble/ble_common.h".

3.4.2.6 Callback function r

The callback function r is read function with prototype as below:

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

User must follow the format above to define callback read function. The callback function r is also optional, i.e. for an Attribute, user can select whether to set the callback read function as needed (null pointer 0 indicates not setting callback read function).

The trigger condition for callback function r is: When Slave receives any Attribute PDU with Attribute Opcode as shown below, Slave will check whether the callback function r is set.

1) opcode = 0x0A, Read Request, refer to “Core_v4.2” Page2184.
2) opcode = 0x0C, Read Blob Request, refer to “Core_v4.2” Page2185.

After Slave receives a read command above,

1) If the callback read function is set, Slave will execute this function, and determine whether to respond with “Read Response/Read Blob Response” according to the return value of this function.
   A. If the return value is 1, Slave won’t respond with “Read Response/Read Blob Response” to Master.
   B. If the return value is not 1, Slave will automatically read “attrLen” bytes of data from the area pointed by the “pAttrValue”, and the data will be responded to Master via “Read Response/Read Blob Response”.
2) If the callback read function is not set, Slave will automatically read “attrLen” bytes of data from the area pointed by the “pAttrValue”, and the data will be responded to Master via “Read Response/Read Blob Response”.

Therefore, after a Read Request/Read Blob Request is received from Master, if it’s needed to modify the content of Read Response/Read Blob Response, user can register corresponding callback function r, modify contents in RAM pointed by the “pAttrValue” in this callback function, and the return value must be 0.

3.4.2.7 Attribute Table layout

Figure3-43 shows Service/Attribute layout based on Attribute Table. The “attnum” of the first Attribute indicates the number of valid Attributes in current ATT Table; the remaining Attributes are assigned to different Services, the first Attribute of each Service is the “declaration”, and the following “attnum” Attributes constitute current Service. Actually the first item of each Service is a Primary Service.

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

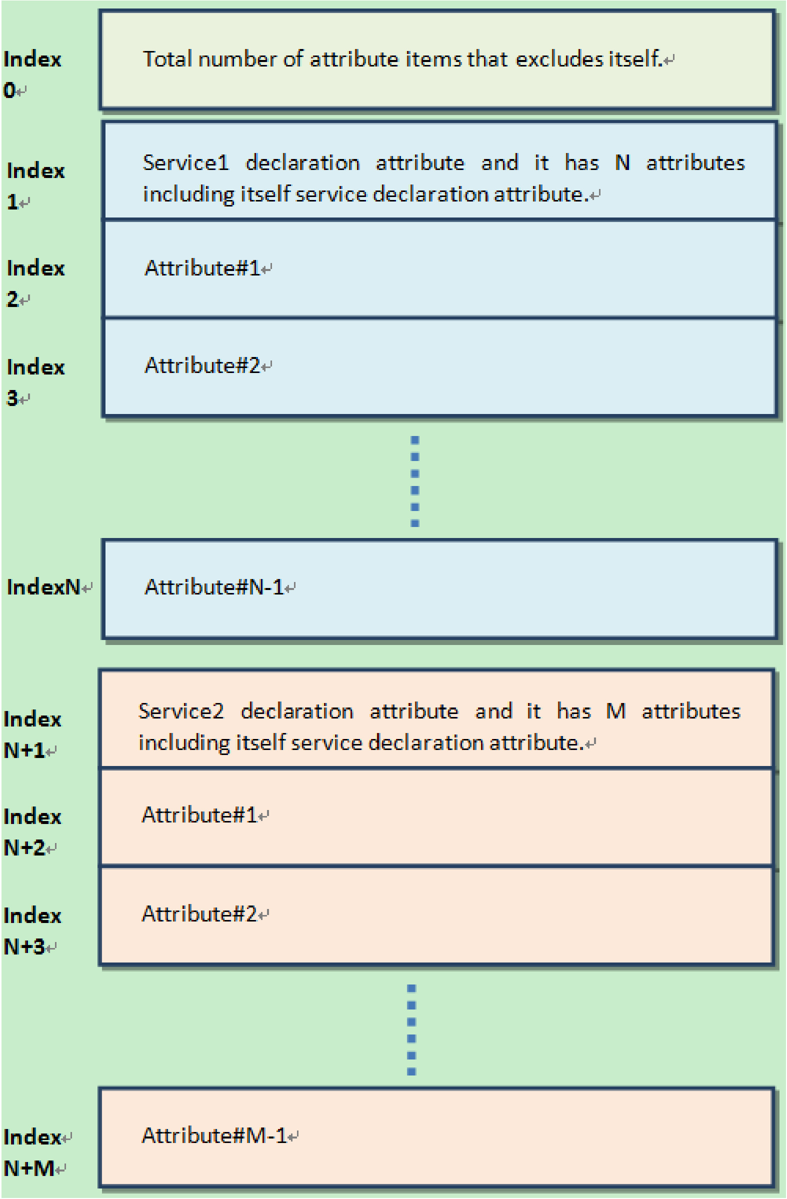

Index 0
Total number of attribute items that excludes itself.
Index 1
Service1 declaration attribute and it has N attributes including itself service declaration attribute.
Index 2
Attribute#1
Index 3
Attribute#2
IndexN
Attribute#N-1
Index N+1
Service2 declaration attribute and it has M attributes including itself service declaration attribute.
Index N+2
Attribute#1
Index N+3
Attribute#2
Index N+M
Attribute#M-1

Figure3-43 Service/Attribute Layout

3.4.2.8 ATT table Initialization

GATT & ATT initialization only needs to transfer the pointer of Attribute Table in APP layer to protocol stack, and the API below is supplied:

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

"p" is the pointer of Attribute Table.

3.4.3 Attribute PDU & GATT API

As required by BLE spec, the following Attribute PDU types are supported in current 826x BLE SDK.

1) Requests: Data request sent from Client to Server.

2) Responses: Data response sent by Server after it receives request from Client.

3) Commands: Command sent from Client to Server.

4) Notifications: Data sent from Server to Client.

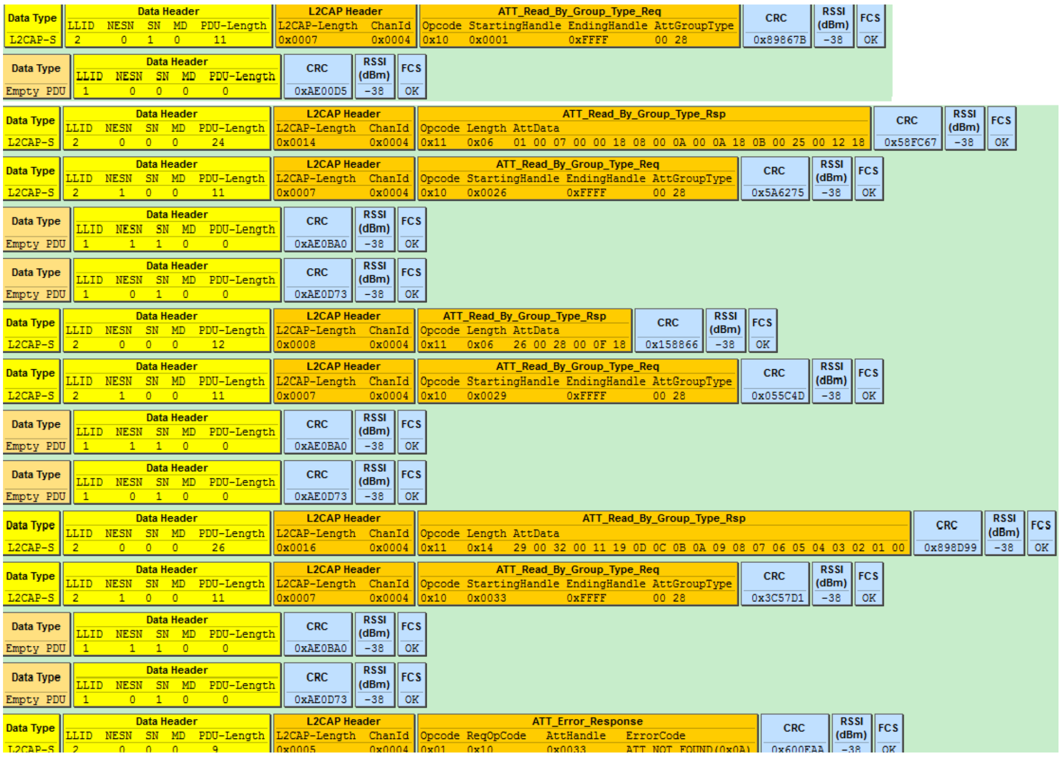
5) Indications: Data sent from Server to Client.

6) Confirmations: Confirmation sent from Client after it receives data from Server.

The subsections below will introduce all ATT PDUs in ATT layer. Please refer to structure of Attribute and Attribute Table to help understanding.

3.4.3.1 Read by Group Type Request, Read by Group Type Response

The “Read by Group Type Request” command sent by Master specifies starting and ending attHandle, as well as attGroupType. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute Group that matches the specified attGroupType. Then Slave will respond to Master with Attribute Group information via the “Read by Group Type Response” command.



```
Data Type | Data Header                    | L2CAP Header           | ATT_Read_By_Group_Type_Req                                 | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode StartingHandle EndingHandle AttGroupType            |          |            |
L2CAP-S   | 2    0    1  0  11             | 0x0007       0x0004    | 0x10   0x0001         0xFFFF       00 28                   | 0x89867B | -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                    | L2CAP Header           | ATT_Read_By_Group_Type_Rsp                                 | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode Length AttData                                      |          |            |
L2CAP-S   | 2    0    0  0  24             | 0x0014       0x0004    | 0x11   0x06   01 00 07 00 00 18 08 00 0A 00 0A 18 0B 00 25 00 12 18 | 0x58FC67 | -38 | OK

Data Type | Data Header                    | L2CAP Header           | ATT_Read_By_Group_Type_Req                                 | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode StartingHandle EndingHandle AttGroupType            |          |            |
L2CAP-S   | 2    1    0  0  11             | 0x0007       0x0004    | 0x10   0x0026         0xFFFF       00 28                   | 0x5A6275 | -38        | OK

Data Type | Data Header                    | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     |          |            |
Empty PDU | 1    1    1  0  0              | 0xAE0BA0 | -38        | OK

Data Type | Data Header                    | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     |          |            |
Empty PDU | 1    0    1  0  0              | 0xAE0D73 | -38        | OK

Data Type | Data Header                    | L2CAP Header           | ATT_Read_By_Group_Type_Rsp    | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode Length AttData         |          |            |
L2CAP-S   | 2    0    0  0  12             | 0x0008       0x0004    | 0x11   0x06   26 00 28 00 0F 18 | 0x158866 | -38      | OK

Data Type | Data Header                    | L2CAP Header           | ATT_Read_By_Group_Type_Req                                 | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode StartingHandle EndingHandle AttGroupType            |          |            |
L2CAP-S   | 2    1    0  0  11             | 0x0007       0x0004    | 0x10   0x0029         0xFFFF       00 28                   | 0x055C4D | -38        | OK

Data Type | Data Header                    | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     |          |            |
Empty PDU | 1    1    1  0  0              | 0xAE0BA0 | -38        | OK

Data Type | Data Header                    | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     |          |            |
Empty PDU | 1    0    1  0  0              | 0xAE0D73 | -38        | OK

Data Type | Data Header                    | L2CAP Header           | ATT_Read_By_Group_Type_Rsp                                 | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode Length AttData                                      |          |            |
L2CAP-S   | 2    0    0  0  26             | 0x0016       0x0004    | 0x11   0x14   29 00 32 00 11 19 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00 | 0x898D99 | -38 | OK

Data Type | Data Header                    | L2CAP Header           | ATT_Read_By_Group_Type_Req                                 | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode StartingHandle EndingHandle AttGroupType            |          |            |
L2CAP-S   | 2    1    0  0  11             | 0x0007       0x0004    | 0x10   0x0033         0xFFFF       00 28                   | 0x3C57D1 | -38        | OK

Data Type | Data Header                    | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     |          |            |
Empty PDU | 1    1    1  0  0              | 0xAE0BA0 | -38        | OK

Data Type | Data Header                    | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     |          |            |
Empty PDU | 1    0    1  0  0              | 0xAE0D73 | -38        | OK

Data Type | Data Header                    | L2CAP Header           | ATT_Error_Response                                         | CRC      | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length     | L2CAP-Length ChanId    | Opcode ReqOpCode AttHandle ErrorCode                       |          |            |
L2CAP-S   | 2    0    0  0  9              | 0x0005       0x0004    | 0x01   0x10      0x0033    ATT_NOT_FOUND(0x0A)             | 0x600FAA | -38        | OK
```

Figure3-44 Read by Group Type Request/Read by Group Type Response

As shown above, Master requests from Slave for Attribute Group information of the “primaryServiceUUID” with UUID of 0x2800.

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

The following groups in Slave Attribute Table meet the requirement according to current demo code.

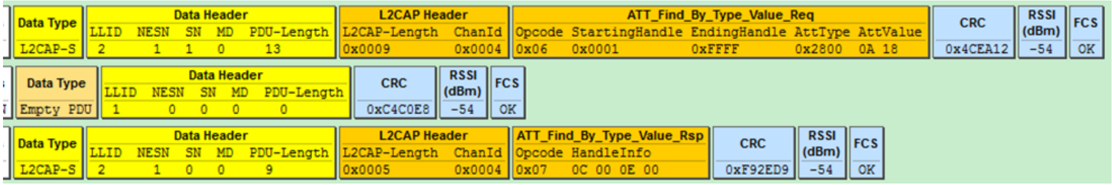
1) Attribute Group with attHandle from 0x0001 to 0x0007, Attribute Value is SERVICE_UUID_GENERIC_ACCESS (0x1800).
2) Attribute Group with attHandle from 0x0008 to 0x000a, Attribute Value is SERVICE_UUID_DEVICE_INFORMATION (0x180A).
3) Attribute Group with attHandle from 0x000B to 0x0025, Attribute Value is SERVICE_UUID_HUMAN_INTERFACE_DEVICE (0x1812).
4) Attribute Group with attHandle from 0x0026 to 0x0028, Attribute Value is SERVICE_UUID_BATTERY (0x180F).
5) Attribute Group with attHandle from 0x0029 to 0x0032, Attribute Value is TELINK_AUDIO_UUID_SERVICE(0x11,0x19,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07, 0x06,0x05,0x04,0x03,0x02,0x01,0x00).

Slave responds to Master with the attHandle and attValue information of the five Groups above via the “Read by Group Type Response” command. The final ATT_Error_Response indicates end of response. When Master receives this packet, it will stop sending “Read by Group Type Request”. Please refer to “Core_v4.2” Page2188 for details about the “Read by Group Type Request” and “Read by Group Type Response” commands.

3.4.3.2 Find by Type Value Request, Find by Type Value Response

The “Find by Type Value Request” command sent by Master specifies starting and ending attHandle, as well as AttributeType and Attribute Value. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute that matches the specified AttributeType and Attribute Value. Then Slave will respond to Master with the Attribute via the “Find by Type Value Response” command.

Please refer to “Core_v4.2” Page2179 for details about the “Find by Type Value Request” and “Find by Type Value Response” commands.

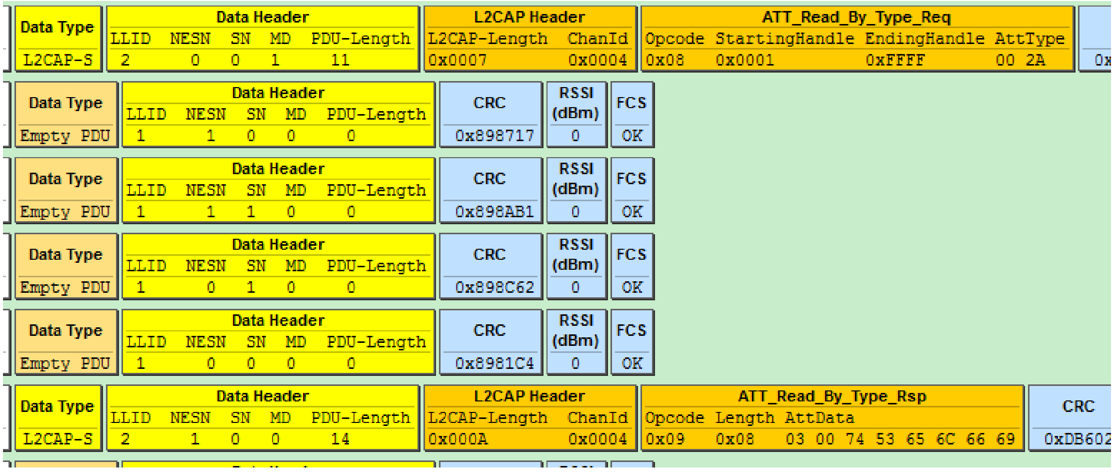


```
Data Type | Data Header                       | L2CAP Header          | ATT_Find_By_Type_Value_Req                                        | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId   | Opcode StartingHandle EndingHandle AttType AttValue               |          | (dBm) |
L2CAP-S   | 2    1    1  0  13                | 0x0009       0x0004   | 0x06   0x0001         0xFFFF       0x2800  0A 18                  | 0x4CEA12 | -54   | OK

Data Type | Data Header                       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        |          | (dBm) |
Empty PDU | 1    0    0  0  0                 | 0xC4C0E8 | -54   | OK

Data Type | Data Header                       | L2CAP Header          | ATT_Find_By_Type_Value_Rsp | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId   | Opcode HandleInfo          |          | (dBm) |
L2CAP-S   | 2    1    0  0  9                 | 0x0005       0x0004   | 0x07   0C 00 0E 00         | 0xF92ED9 | -54   | OK
```

Figure3-45 Find by Type Value Request/Find by Type Value Response

3.4.3.3 Read by Type Request, Read by Type Response

The "Read by Type Request" command sent by Master specifies starting and ending attHandle, as well as AttributeType. After the request is received, Slave will check through current Attribute Table according to the specified starting and ending attHandle, and find the Attribute that matches the specified AttributeType. Then Slave will respond to Master with the Attribute via the "Read by Type Response".



```
Data Type | Data Header                       | L2CAP Header          | ATT_Read_By_Type_Req
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId   | Opcode StartingHandle EndingHandle AttType       |
L2CAP-S   | 2    0    0  1  11                | 0x0007       0x0004   | 0x08   0x0001         0xFFFF       00 2A         | 0x

Data Type | Data Header                       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        |          | (dBm) |
Empty PDU | 1    1    0  0  0                 | 0x898717 | 0     | OK

Data Type | Data Header                       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        |          | (dBm) |
Empty PDU | 1    1    1  0  0                 | 0x898AB1 | 0     | OK

Data Type | Data Header                       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        |          | (dBm) |
Empty PDU | 1    0    1  0  0                 | 0x898C62 | 0     | OK

Data Type | Data Header                       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        |          | (dBm) |
Empty PDU | 1    0    0  0  0                 | 0x8981C4 | 0     | OK

Data Type | Data Header                       | L2CAP Header          | ATT_Read_By_Type_Rsp                         | CRC
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId   | Opcode Length AttData                        |
L2CAP-S   | 2    1    0  0  14                | 0x000A       0x0004   | 0x09   0x08   03 00 74 53 65 6C 66 69        | 0xDB602
```

Figure3-46 Read by Type Request/Read by Type Response

As shown above, Master reads the Attribute with attType of 0x2A00, i.e. the Attribute with Attribute Handle of 00 03 in Slave.

```
const u8  my_devName [] = {'t', 'S', 'e', 'l', 'f', 'i'};

#define GATT_UUID_DEVICE_NAME                    0x2a00

const u16 my_devNameUUID = GATT_UUID_DEVICE_NAME;

{0,2,sizeof (my_devName), sizeof (my_devName),(u8*)(&my_devNameUUID),
                                                        (u8*)(my_devName), 0},
```

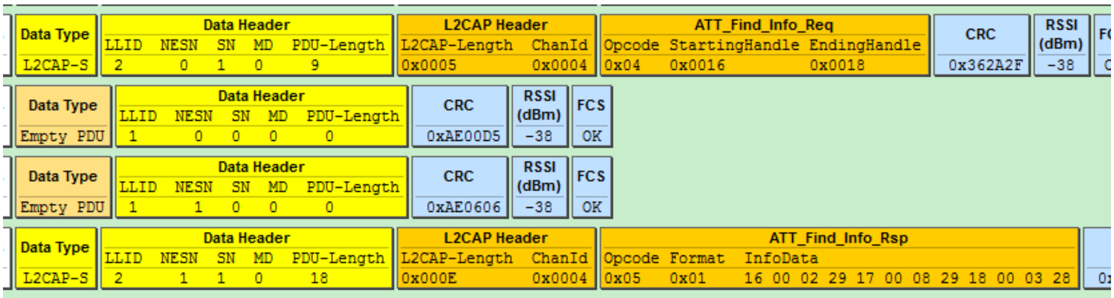
In the "Read by Type response", attData length is 8, the first two bytes are current attHandle "0003", followed by 6-byte Attribute Value.

Please refer to "Core_v4.2" Page2181 for details about the "Read by Type Request" and "Read by Type Response" commands.

3.4.3.4 Find information Request, Find information Response

The “Find information request” command sent by Master specifies starting and ending attHandle. After the request is received, Slave will respond to Master with Attribute UUIDs according to the specified starting and ending attHandle via the “Find information response”.

As shown below, Master requests for information of three Attributes with attHandle of 0x0016~0x0018, and Slave responds with corresponding UUIDs.



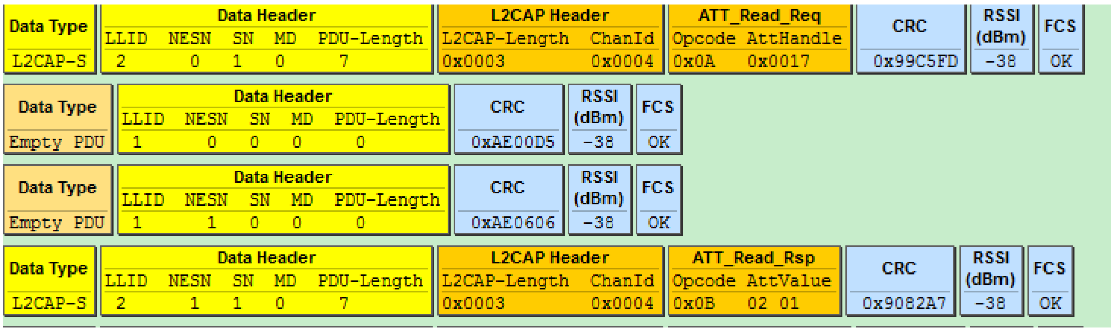
```
Data Type | Data Header                      | L2CAP Header          | ATT_Find_Info_Req                          | CRC      | RSSI  | F
          | LLID NESN SN MD PDU-Length       | L2CAP-Length  ChanId  | Opcode StartingHandle EndingHandle         |          | (dBm) |
L2CAP-S   | 2    0    1  0  9                | 0x0005        0x0004  | 0x04   0x0016         0x0018               | 0x362A2F | -38   | C

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

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

Data Type | Data Header                      | L2CAP Header          | ATT_Find_Info_Rsp                                        |
          | LLID NESN SN MD PDU-Length       | L2CAP-Length  ChanId  | Opcode Format InfoData                                   |
L2CAP-S   | 2    1    1  0  18               | 0x000E        0x0004  | 0x05   0x01   16 00 02 29 17 00 08 29 18 00 03 28          | 0:
```

Figure3-47 Find information request/Find information response

Please refer to “Core_v4.2” Page2177 for details about the “Find information request” and “Find information response” commands.

3.4.3.5 Read Request, Read Response

The “Read Request” command sent by Master specifies certain attHandle. After the request is received, Slave will respond to Master with the Attribute Value of the specified Attribute via the “Read Response” command (If the callback function r is set, this function will be executed), as shown below.



```
Data Type | Data Header                      | L2CAP Header          | ATT_Read_Req       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length       | L2CAP-Length  ChanId  | Opcode AttHandle   |          | (dBm) |
L2CAP-S   | 2    0    1  0  7                | 0x0003        0x0004  | 0x0A   0x0017      | 0x99C5FD | -38   | OK

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

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

Data Type | Data Header                      | L2CAP Header          | ATT_Read_Rsp       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length       | L2CAP-Length  ChanId  | Opcode AttValue    |          | (dBm) |
L2CAP-S   | 2    1    1  0  7                | 0x0003        0x0004  | 0x0B   02 01       | 0x9082A7 | -38   | OK
```

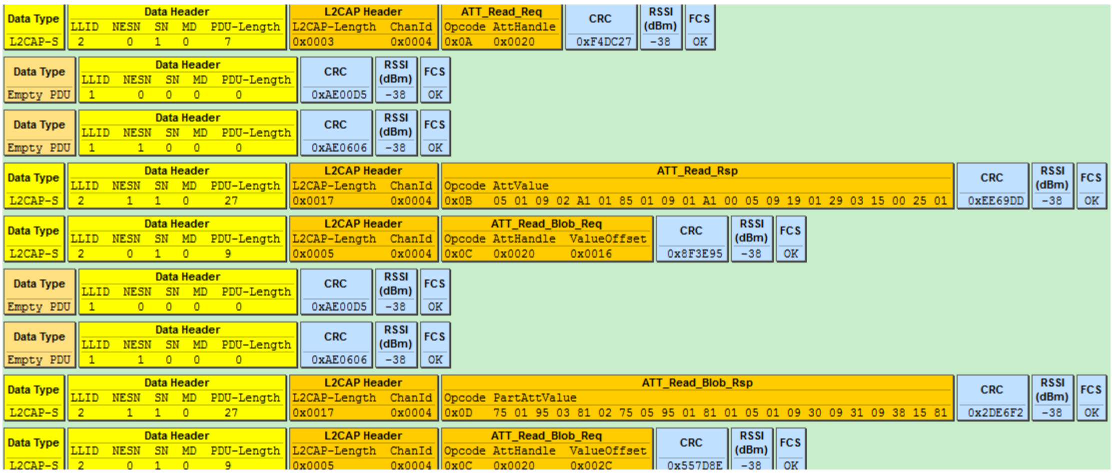
Figure3-48 Read Request/Read Response

Please refer to “Core_v4.2” Page2184 for details about the “Read Request” and “Read Response” commands.

3.4.3.6 Read Blob Request, Read Blob Response

If some Slave Attribute corresponds to Attribute Value with length exceeding MTU_SIZE (It's set as 23 in current SDK), Master needs to read the Attribute Value via the "Read Blob Request" command, so that the Attribute Value can be sent in packets. This command specifies the attHandle and ValueOffset. After the request is received, Slave will find corresponding Attribute, and respond to Master with the Attribute Value via the "Read Blob Response" command according to the specified ValueOffset. (If the callback function r is set, this function will be executed.)

As shown below, when Master needs the HID report map of Slave (report map length largely exceeds 23), first Master sends "Read Request", then Slave responds to Master with part of the report map data via "Read response"; Master sends "Read Blob Request", and then Slave responds to Master with data via "Read Blob Response".



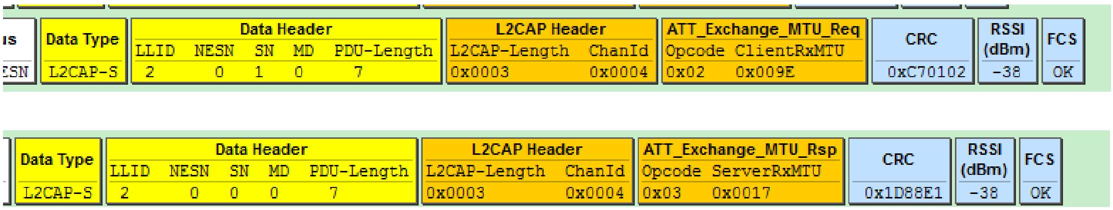
```
Data Type | Data Header                          | L2CAP Header          | ATT_Read_Req       | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length           | L2CAP-Length ChanId   | Opcode AttHandle   |          | (dBm) |
L2CAP-S   | 2    0    1  0  7                    | 0x0003       0x0004   | 0x0A   0x0020      | 0xF4DC27 | -38   | OK

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

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

Data Type | Data Header                | L2CAP Header        | ATT_Read_Rsp                                                                    | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length | L2CAP-Length ChanId | Opcode AttValue                                                                 |          | (dBm) |
L2CAP-S   | 2    1    1  0  27         | 0x0017       0x0004 | 0x0B   05 01 09 02 A1 01 85 01 09 01 A1 00 05 09 19 01 29 03 15 00 25 01       | 0xEE69DD | -38   | OK

Data Type | Data Header                | L2CAP Header        | ATT_Read_Blob_Req              | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length | L2CAP-Length ChanId | Opcode AttHandle ValueOffset   |          | (dBm) |
L2CAP-S   | 2    0    1  0  9          | 0x0005       0x0004 | 0x0C   0x0020    0x0016        | 0x8F3E95 | -38   | OK

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

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

Data Type | Data Header                | L2CAP Header        | ATT_Read_Blob_Rsp                                                               | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length | L2CAP-Length ChanId | Opcode PartAttValue                                                             |          | (dBm) |
L2CAP-S   | 2    1    1  0  27         | 0x0017       0x0004 | 0x0D   75 01 95 03 81 02 75 05 95 01 81 01 05 01 09 30 09 31 09 38 15 81       | 0x2DE6F2 | -38   | OK

Data Type | Data Header                | L2CAP Header        | ATT_Read_Blob_Req              | CRC      | RSSI  | FCS
          | LLID NESN SN MD PDU-Length | L2CAP-Length ChanId | Opcode AttHandle ValueOffset   |          | (dBm) |
L2CAP-S   | 2    0    1  0  9          | 0x0005       0x0004 | 0x0C   0x0020    0x002C        | 0x557D8E | -38   | OK
```

Figure3-49 Read Blob Request/Read Blob Response

Please refer to "Core_v4.2" Page2185 for details about the "Read Blob Request" and "Read Blob Response" commands.

3.4.3.7 Exchange MTU Request, Exchange MTU Response

As shown below, Master and Slave obtain MTU size of each other via the "Exchange MTU Request" and "Exchange MTU Response" commands.



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

Figure3-50 Exchange MTU Request/Exchange MTU Response

Please refer to "Core_v4.2" Page2175 for details about the "Exchange MTU Request" and "Exchange MTU Response" commands.

During data access process of Telink BLE Slave GATT layer, if there's data exceeding a RF packet length, which involves packet assembly and disassembly in GATT layer, Slave and Master need to exchange RX MTU size of each other in advance. Transfer of long packet data in GATT layer can be implemented via MTU size exchange.

1) Callback function of MTU size exchange

Function prototype:

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

The first u16 is current connection handle, and it should be "BLS_CONN_HANDLE" in Slave applications.

The second u16 is ClientRxMTU of Master, based on which Slave can determine the maximum length for data transfer.

The API below serves to register this callback function:

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

2) Processing of long Rx packet data in 826x Slave GATT layer

826x Slave ServerRxMTU is set as 23 by default. Actually maximum ServerRxMTU can reach 241, i.e. 241-byte packet data on Master can be correctly re-assembled on Slave. When it's needed to use packet re-assembly of Master in an application, the API below should be invoked to modify RX size of Slave first.

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

The return value is shown as below:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	
ATT_ERR_INVALID_PARAMETER	0x12	mtu_size exceeds the max value 241.

When Master GATT layer needs to send long packet data to Slave, Master will actively initiate "ATT_Exchange_MTU_req", and Slave will respond with "ATT_Exchange_MTU_rsp". "ServerRxMTU" is the configured value of the API "blc_att_setRxMtuSize". The callback function registered via "blc_att_registerMtuSizeExchangeCb" will be triggered, and the second parameter of the callback is “ClientRxMTU” of Master.

3) Processing of long Tx packet data in 826x Slave GATT layer

When 826x Slave needs to send long packet data in GATT layer, it should obtain Client RxMTU of Master first, and the eventual data length should not exceed ClientRxMTU.

First Slave should invoke the API “blc_att_setRxMtuSize” to set its ServerRxMTU. Suppose it’s set as 158.

```
blc_att_setRxMtuSize (158);
```

Then the API below should be invoked to actively initiate an “ATT_Exchange_MTU_req”.

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

“connHandle” is ID of Slave conection, i.e. “BLS_CONN_HANDLE”, while “mtu_size” is ServerRxMTU.

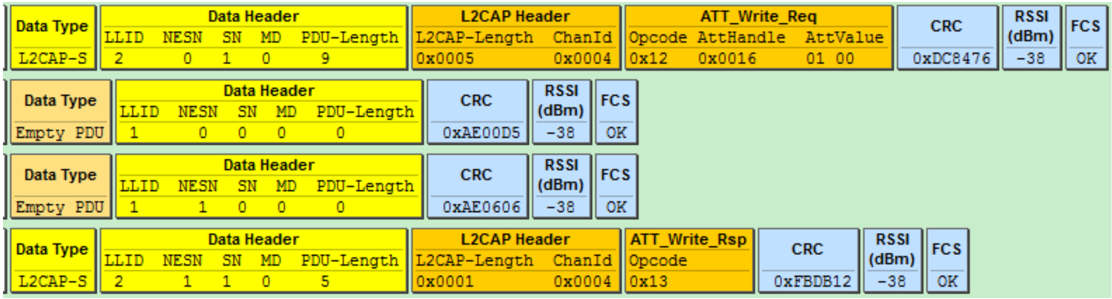
```
blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE,    158);
```

After the “ATT_Exchange_MTU_req” is received, Master will respond with “ATT_Exchange_MTU_rsp”. Then the callback function registered via “blc_att_registerMtuSizeExchangeCb” will be triggered, and the second parameter of the callback function is ClientRxMTU of Master.

3.4.3.8 Write Request, Write Response

The “Write Request” command sent by Master specifies certain attHandle and attaches related data. After the request is received, Slave will find the specified Attribute, determine whether to process the data by using the callback function w or directly write the data into corresponding Attribute Value depending on whether the callback function w is set by user. Finally Slave will respond to Master via “Write Response”.

As shown in below, by sending “Write Request”, Master writes Attribute Value of 0x0001 to the Slave Attribute with the attHandle of 0x0016. Then Slave will execute the write operation and respond to Master via “Write Response”.



Data Type	Data Header					L2CAP Header		ATT_Write_Req			CRC	RSSI (dBm)	FCS
	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

Figure3-51 Write Request/Write Respons

Please refer to "Core_v4.2" Page2191 for details about the "Write Request" and "Write Response" commands.

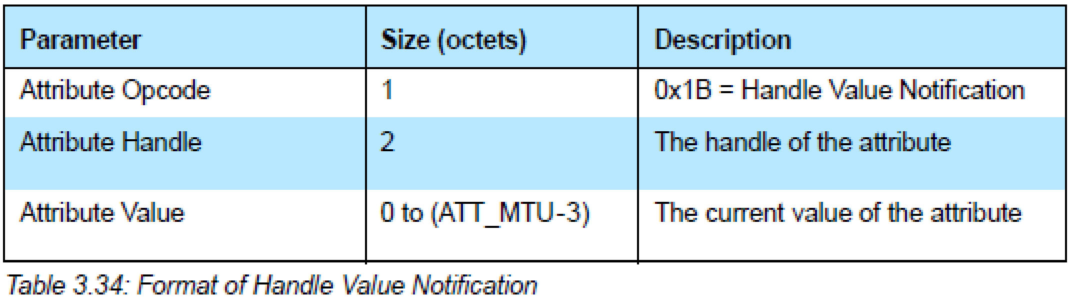
3.4.3.9 Write Command

The "Write Command" sent by Master specifies certain attHandle and attaches related data. After the command is received, Slave will find the specified Attribute, determine whether to process the data by using the callback function w or directly write the data into corresponding Attribute Value depending on whether the callback function w is set by user. Slave won't respond to Master with any information.

Please refer to "Core_v4.2" Page2193 for details about the "Write Command".

3.4.3.10 Handle Value Notification

Please refer to "Core_v4.2" Page2199.



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

Table 3.34: Format of Handle Value Notification

Figure3-52 Handle Value Notification in BLE Spec

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

826x BLE SDK supplies an API for Handle Value Notification of an Attribute. By invoking this API, user can push the notify data into bottom-layer BLE software fifo. Stack will push the data of software fifo into hardware fifo during the latest packet transfer interval, and finally send the data out via RF.

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

"handle" is attHandle of Attribute, "p" is the head pointer of successive memory data to be sent, while "len" specifies byte number of data to be sent. Since this API supports auto packet disassembly, long notify data to be sent can be disassembled into multiple BLE RF packets, large "len" is supported.

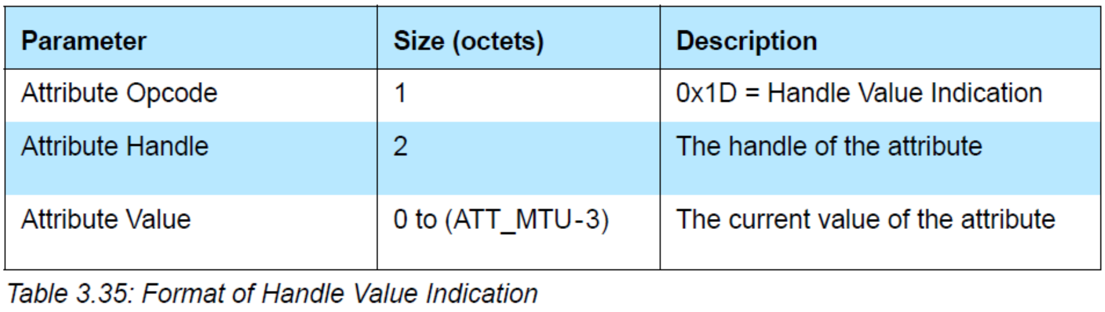
When Link Layer is in Conn state, generally data can be successfully pushed into bottom-layer software fifo by invoking this API. However, some special cases may result in invoking failure, and the return value "ble_sts_t" will indicate the corresponding error reason.

When this API is invoked in APP layer, it's recommended to check whether the return value is "BLE_SUCCESS". If the return value is not "BLE_SUCCESS", a delay is needed to re-push the data. The return value is shown as below:

ble_sts_t	**Value**	**ERR reason**
BLE_SUCCESS	0	
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer is in None Conn state.
SMP_EER_PAIRING_IS_GOING_ON	0x8F	Data cannot be sent during pairing phase.
HCI_ERR_CONTROLLER_TX_FIFO_NOT_ENOUGH	0x3A	Since task with mass data is being executed, software Tx fifo is not enough.

3.4.3.11 Handle Value Indication

Please refer to "Core_v4.2" Page2199.



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

Table 3.35: Format of Handle Value Indication

Figure3-53 Handle Value Indication in BLE spec

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

826x BLE SDK supplies an API for Handle Value Indication of an Attribute. By invoking this API, user can push the indicate data into bottom-layer BLE software fifo. Stack will push the data of software fifo into hardware fifo during the latest packet transfer interval, and finally send the data out via RF.

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

"handle" is attHandle corresponding to Attribute, "p" is the head pointer of successive memory data to be sent, while "len" specifies byte number of data to be sent. Since this API supports auto packet disassembly, long indicate data to be sent can be disassembled into multiple BLE RF packets, large "len" is supported.

As specified in BLE Spec, Slave won't regard data indication as success until Master confirms the data, and the next indicate data won't be sent until the previous data indication is successful.

When Link Layer is in Conn state, generally data will be successfully pushed into bottom-layer software FIFO by invoking this API; however, some special cases may result in invoking failure, and the return value "ble_sts_t" will indicate the corresponding error reason.

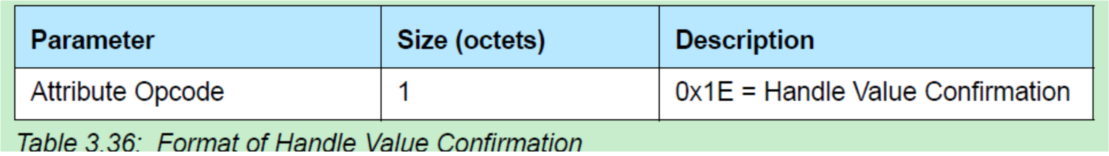
When this API is invoked in APP layer, it's recommended to check whether the return value is "BLE_SUCCESS". If the return value is not "BLE_SUCCESS", a delay is needed to re-push the data. The return value is shown as below:

ble_sts_t	**Value**	**ERR reason**
BLE_SUCCESS	0	
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer is in None Conn state.
SMP_EER_PAIRING_IS_GOING_ON	0x8F	Data cannot be sent during pairing phase.
HCI_ERR_CONTROLLER_TX_FIFO_NOT_ENOUGH	0x3A	Task with mass data is being executed, and software Tx fifo is not enough.
ATT_ERR_PREVIOUS_INDICATE_DATA_HAS_NOT_CONFIRMED	0x6B	The previous indicate data has not been confirmed by Master.

3.4.3.12 Handle Value Confirmation

Please refer to "Core_v4.2" Page2200.

Whenever the API "bls_att_pushIndicateData" is invoked by APP layer to send an indicate data to Master, Master will respond with “Confirmation” to confirm the data, then Slave can continue to send the next indicate data.



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

Table 3.36: Format of Handle Value Confirmation

Figure3-54 Handle Value Confirmation in BLE Spec

As shown above, “Confirmation” is not specific to indicate data of certain handle, and the same “Confirmation” will be responded irrespective of handle.

A callback function is supplied in SDK for the APP layer to check whether the indicate data has already been confirmed by Master. The registered callback function will be executed once when a Handle Value Confirmation is received.

Type definition of the callback function is shown as below:

```
typedef int (*att_handleValueConfirm_callback_t)(void);
```

The API below serves to register the callback function:

```
void   bls_att_registerHandleValueConfirmCb
                          (att_handleValueConfirm_callback_t cb);
```

3.4.4 826x master GATT

In 826x master kma dongle, the following GATT APIs are supplied for simple service discovery or other data access functions.

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

Actual length (byte) of dat: 11.

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

Actual length (byte) of dat: 13 + attr_value length.

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

Actual length (byte) of dat: 11 + uuid length.

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

Actual length (byte) of dat: 9.

```
void   att_req_read_blob (u8 *dat, u16 attHandle, u16 offset);
```

Actual length (byte) of dat: 11.

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

Actual length (byte) of dat: 11 + uuid length.

```
void   att_req_write (u8 *dat, u16 attHandle, u8 *buf, int len);
```

Actual length (byte) of dat: 9 + buf data length.

```
void   att_req_write_cmd (u8 *dat, u16 attHandle, u8 *buf, int len);
```

Actual length (byte) of dat: 9 + buf data length.

For the APIs above, it's needed to pre-define memory space *dat, then invoke corresponding API to assemble data, finally invoke "blm_push_fifo" to send "dat" to Controller for transmission. Note that it's needed to check whether the return value is TRUE. The API "att_req_find_info" is taken as an example for user reference.

```
u8   cmd[12];
att_req_find_info(cmd,   0x0001,   0x0003);
if( blm_push_fifo (BLM_CONN_HANDLE, cmd) ){
    //cmd send OK
}
```

As shown above, after a cmd (e.g. "find info req"/"read req") is received, Slave will respond with corresponding response information (e.g. "find info rsp"/"read rsp") soon. It's only needed to process in "int app_l2cap_handler (u16 conn_handle, u8 *raw_pkt)" according to the frame below:

```
if(ptrL2cap->chanId == L2CAP_CID_ATTR_PROTOCOL)  //att data
{
    if(pAtt->opcode == ATT_OP_EXCHANGE_MTU_RSP){
        //add your code
    }
    if(pAtt->opcode == ATT_OP_FIND_INFO_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_FIND_BY_TYPE_VALUE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BY_TYPE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BLOB_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_READ_BY_GROUP_TYPE_RSP){
        //add your code
    }
    else if(pAtt->opcode == ATT_OP_WRITE_RSP){
        //add your code
    }
}
```

3.5 SMP

3.5.1 SMP parameter configuration

Parameter configuration related to SMP initialization include device bonding and OOB (Out-Of-Band) data verification.

3.5.1.1 Device bonding

When it’s needed to bond peer device information after pairing, the function below should be invoked to enable current device bonding request.

int blc_smp_enableBonding (int en);

*Note: “en” = 0, disable current device bonding;

“en”=1 (default), enable current device bonding.

3.5.1.2 Device OOB data verification

The function below is used for OOB data verification.

void blc_smp_enableOobFlag (int en, u8 *oobData);

*Notes: en: enable (en=1) or disable (en=0, default) OOB data verification.

oobData: OOB data verification value, pointer to a group of 16-byte data.

3.5.2 SMP enable

```
int bls_smp_enableParing (smp_paringTrriger_t encrypt_en);
```

Following introduces the definition of the enum-type "smp_paringTrriger_t" and each parameter.

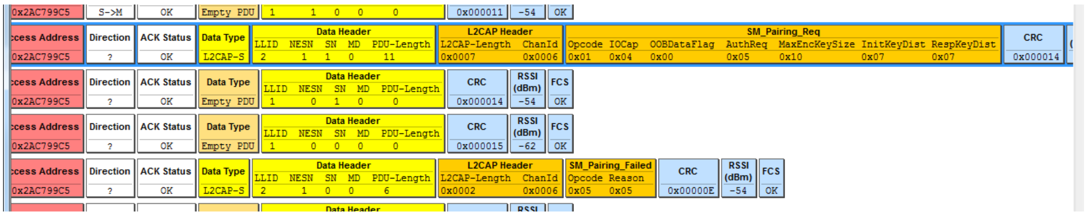
```
typedef enum{
    SMP_PARING_DISABLE_TRRIGER = 0,
    SMP_PARING_CONN_TRRIGER ,
    SMP_PARING_PEER_TRRIGER,
}smp_paringTrriger_t;
```

1) encrypt_en = *SMP_PARING_DISABLE_TRRIGER;*

It indicates pairing encryption is disabled for current device connection. Even if peer device requests for pairing encryption, the device will reject this request.

It applies to the case when current device does not support encrypted pairing.

As shown below, Master sends pairing request, and then Slave responds with "SM_Pairing_Failed".

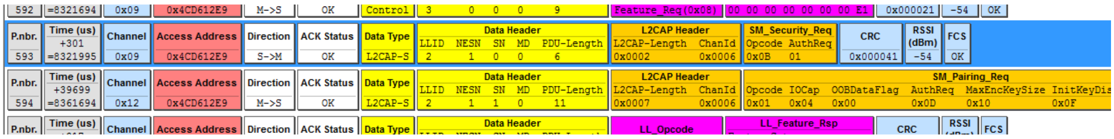


```
0x2AC799C5  S->M  OK  Empty PDU  1  1  0  0  0  0x000011  -54  OK

cess Address  Direction  ACK Status  Data Type  Data Header                        L2CAP Header           SM_Pairing_Req                                                                     CRC
                                                LLID NESN SN MD PDU-Length         L2CAP-Length  ChanId   Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist RespKeyDist
0x2AC799C5    ?          OK          L2CAP-S    2    1    1  0  11                 0x0007        0x0006   0x01   0x04  0x00        0x05    0x10          0x07        0x07        0x000014

cess Address  Direction  ACK Status  Data Type  Data Header                        CRC       RSSI (dBm)  FCS
                                                LLID NESN SN MD PDU-Length
0x2AC799C5    ?          OK          Empty PDU  1    0    1  0  0                  0x000014  -54         OK

cess Address  Direction  ACK Status  Data Type  Data Header                        CRC       RSSI (dBm)  FCS
                                                LLID NESN SN MD PDU-Length
0x2AC799C5    ?          OK          Empty PDU  1    0    0  0  0                  0x000015  -62         OK

cess Address  Direction  ACK Status  Data Type  Data Header                        L2CAP Header           SM_Pairing_Failed  CRC       RSSI (dBm)  FCS
                                                LLID NESN SN MD PDU-Length         L2CAP-Length  ChanId   Opcode Reason
0x2AC799C5    ?          OK          L2CAP-S    2    1    0  0  6                  0x0002        0x0006   0x05   0x05        0x00000E  -54         OK
```

Figure3-55 Pairing Disable

2) encrypt_en = *SMP_PARING_CONN_TRRIGER;*

It indicates current device will actively initiate pairing encryption request once it's connected with peer device. If peer device initiates pairing request first, current device will still send pairing request and also respond to the request from peer device. As shown below, Slave actively sends the "SM_Security_Req".



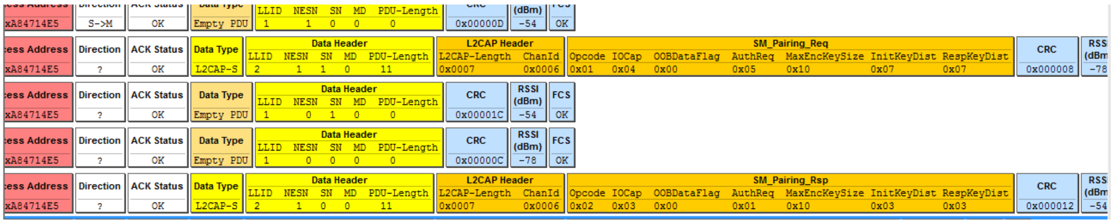
```
592  =8321694  0x09  0x4CD612E9  M->S  OK  Control  3  0  0  0  9  Feature_Req(0x08)  00 00 00 00 00 00 00 E1  0x000021  -54  OK

P.nbr.  Time (us)  Channel  Access Address  Direction  ACK Status  Data Type  Data Header                  L2CAP Header           SM_Security_Req  CRC       RSSI (dBm)  FCS
        +301                                                                  LLID NESN SN MD PDU-Length   L2CAP-Length  ChanId   Opcode AuthReq
593     =8321995   0x09     0x4CD612E9      S->M       OK          L2CAP-S    2    1    0  0  6            0x0002        0x0006   0x0B   01        0x000041  -54         OK

P.nbr.  Time (us)  Channel  Access Address  Direction  ACK Status  Data Type  Data Header                  L2CAP Header           SM_Pairing_Req
        +39699                                                                LLID NESN SN MD PDU-Length   L2CAP-Length  ChanId   Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDis
594     =8361694   0x12     0x4CD612E9      M->S       OK          L2CAP-S    2    1    1  0  11           0x0007        0x0006   0x01   0x04  0x00        0x0D    0x10          0x0F

P.nbr.  Time (us)  Channel  Access Address  Direction  ACK Status  Data Type  Data Header                  LL_Opcode  LL_Feature_Rsp  CRC  RSSI  FCS
```

Figure3-56 Pairing Conn Trigger

3) encrypt_en = *SMP_PARING_PEER_TRRIGER;*

It indicates current device won't actively intiate pairing request, and it will only respond to the pairing request from peer device. If peer device does not send pairing request, current device won't implement encrypted pairing.

As shown below, Slave will respond to the "SM_Pairing_Req" from Master, but won't actively initiate pairing request.



```
xA84714E5 | S->M | OK | Empty PDU | LLID NESN SN MD PDU-Length: 1 1 0 0 0 | CRC 0x00000D | (dBm) -54 | OK

ess Address | Direction | ACK Status | Data Type | Data Header                  | L2CAP Header           | SM_Pairing_Req                                                                        | CRC      | RSS (dBm
            |           |            |           | LLID NESN SN MD PDU-Length   | L2CAP-Length ChanId    | Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist RespKeyDist               |          |
xA84714E5   | ?         | OK         | L2CAP-S   | 2    1    1  0  11           | 0x0007       0x0006    | 0x01   0x04  0x00        0x05    0x10          0x07        0x07                      | 0x000008 | -78

ess Address | Direction | ACK Status | Data Type | Data Header                  | CRC      | RSSI (dBm) | FCS
            |           |            |           | LLID NESN SN MD PDU-Length   |          |            |
xA84714E5   | ?         | OK         | Empty PDU | 1    0    1  0  0            | 0x00001C | -54        | OK

ess Address | Direction | ACK Status | Data Type | Data Header                  | CRC      | RSSI (dBm) | FCS
            |           |            |           | LLID NESN SN MD PDU-Length   |          |            |
xA84714E5   | ?         | OK         | Empty PDU | 1    0    0  0  0            | 0x00000C | -78        | OK

ess Address | Direction | ACK Status | Data Type | Data Header                  | L2CAP Header           | SM_Pairing_Rsp                                                                        | CRC      | RSS (dBm
            |           |            |           | LLID NESN SN MD PDU-Length   | L2CAP-Length ChanId    | Opcode IOCap OOBDataFlag AuthReq MaxEncKeySize InitKeyDist RespKeyDist               |          |
xA84714E5   | ?         | OK         | L2CAP-S   | 2    1    0  0  11           | 0x0007       0x0006    | 0x02   0x03  0x00        0x01    0x10          0x03        0x03                      | 0x000012 | -54
```

Figure3-57 Pairing Peer Trigger

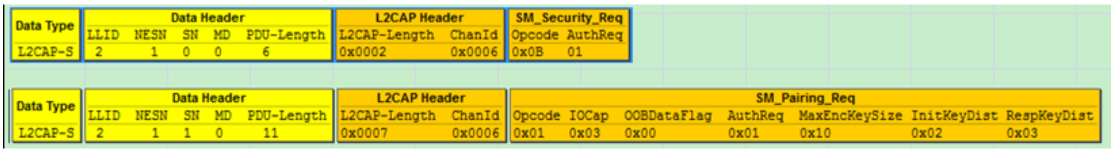
Note: This function can only be invoked before connection. It's recommended to invoke this function during initialization.

3.5.3 SMP event

As introduced in Controller part, except for Telink defined events, there are some SMP events, e.g. "BLT_EV_FLAG_PAIRING_BEGIN", "BLT_EV_FLAG_PAIRING_END".

3.5.3.1 BLT_EV_FLAG_PAIRING_BEGIN

1) Event trigger condition: When Slave just establishes connection with Master and enters connection state, Slave sends "SM_Security_Req" command, and then Master sends "SM_Pairing_Req" to request for pairing. After Slave receives this pairing request, this event will be triggered to indicate pairing starts.



```
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
```

Figure3-58 Pairing_Req sent from Master

2) Pointer "p": Null pointer.

3) Data length "n": 0.

3.5.3.2 BLT_EV_FLAG_PAIRING_END

1) Event trigger condition: This event will be triggered when pairing is finished with success or failure. If Slave or Master fails to follow standard pairing procedure, or communication abnormity occurs (e.g. report error), pairing will fail.

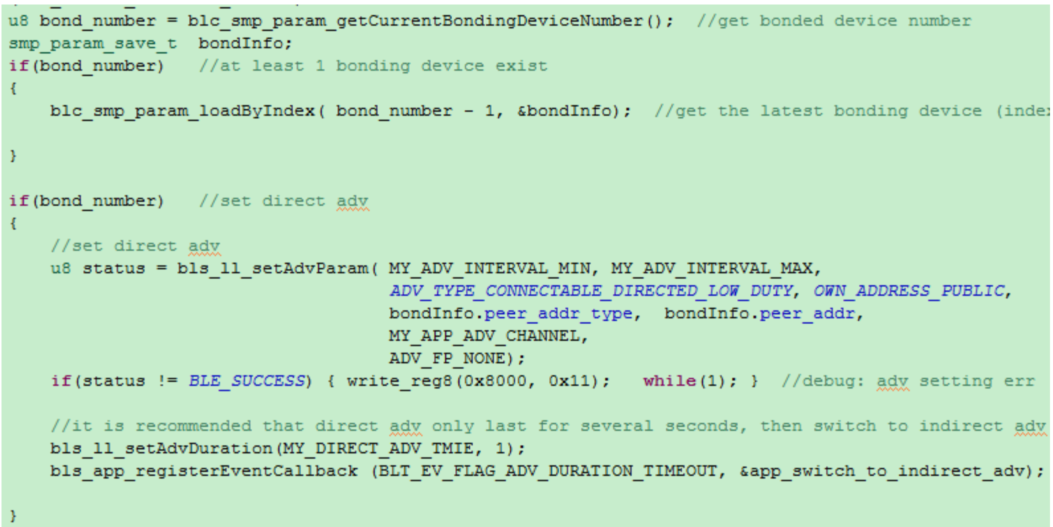
2) Data length "n": 1.

3) Pointer "p": It points to a flag variable, which should be either 0 (pairing success)

or non-zero value (pairing failure).

3.5.4 SMP bonding information

Please refer to the code of “set direct adv” during initialization of 826x remote.



```
u8 bond_number = blc_smp_param_getCurrentBondingDeviceNumber();  //get bonded device number
smp_param_save_t  bondInfo;
if(bond_number)   //at least 1 bonding device exist
{
    blc_smp_param_loadByIndex( bond_number - 1, &bondInfo);  //get the latest bonding device (inde

}

if(bond_number)   //set direct adv
{
    //set direct adv
    u8 status = bls_ll_setAdvParam( MY_ADV_INTERVAL_MIN, MY_ADV_INTERVAL_MAX,
                                    ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY, OWN_ADDRESS_PUBLIC,
                                    bondInfo.peer_addr_type,  bondInfo.peer_addr,
                                    MY_APP_ADV_CHANNEL,
                                    ADV_FP_NONE);
    if(status != BLE_SUCCESS) { write_reg8(0x8000, 0x11);   while(1); }  //debug: adv setting err

    //it is recommended that direct adv only last for several seconds, then switch to indirect adv
    bls_ll_setAdvDuration(MY_DIRECT_ADV_TMIE, 1);
    bls_app_registerEventCallback (BLT_EV_FLAG_ADV_DURATION_TIMEOUT, &app_switch_to_indirect_adv);

}
```

Slave can store pairing information of up to four Master devices at the same time. All of the four devices can be re-connected successfully.

The API below serves to set the maximum device number for current storage, which should not exceed 4 (SMP_BONDING_DEVICE_MAX_NUM). The default value is 4.

```
#define        SMP_BONDING_DEVICE_MAX_NUM                 4

ble_sts_t     blc_smp_param_setBondingDeviceMaxNumber( int device_num);
```

Suppose it’s set as “blc_smp_param_setBondingDeviceMaxNumber (4)”: When pairing information of four paired devices are stored, if the 5th device is paired, the pairing info of the oldest device will be deleted automatically, so that the pairing info of the 5th device can be stored.

Suppose it’s set as “blc_smp_param_setBondingDeviceMaxNumber (2)”: When pairing information of two paired devices are stored, if the 3rd device is paired, the pairing info of the oldest device will be deleted automatically, so that the pairing info of the 3rd device can be stored.

The API below serves to obtain the number of successfully paired Master devices with pairing info stored in Slave flash.

```
        u8           blc_smp_param_getCurrentBondingDeviceNumber(void);
```

If the return value is 3, it indicates three paired devices are stored in flash currently, and all of the three devices can be re-connected successfully.

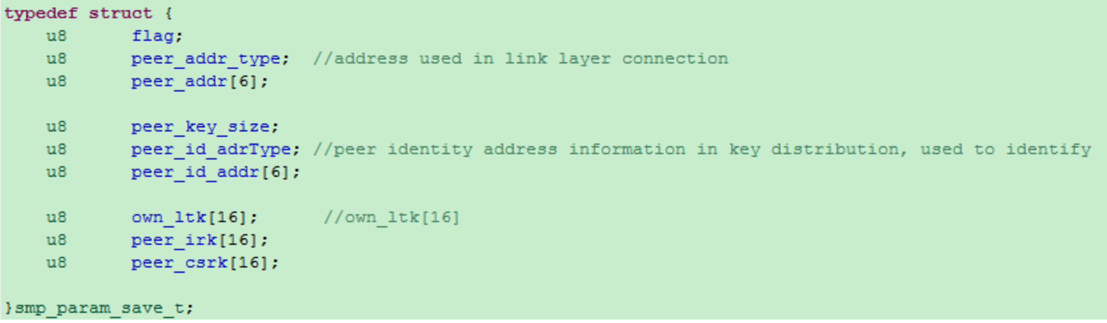
"index" is related to "BondingDeviceNumber": If "BondingDeviceNumber" is 1, there is only one bonding device, and its index is 0. If "BondingDeviceNumber" is 2, there are two bonding devices, and the index of the two devices are 0 and 1, respectively. The index sequence is determined by the latest successful connection rather than the latest pairing: Suppose Slave is successfully paired with MasterA and MasterB, successively, since MasterB is the latest device at this moment, in Slave flash storage, MasterA is index 0, while MasterB is index 1. Then Slave is re-connected with MasterA successfully, since the latest device is MasterA at this moment, MasterB is index 0, while MasterA is index 1.

If "BondingDeviceNumber" is 3, the index of the three devices are 0 (the first connected device), 1, 2 (the latest connected device).

If "BondingDeviceNumber" is 4, the index of the four devices are 0 (the first connected device), 1, 2, 3 (the latest connected device). As introduced above, if Slave is successively paired with MasterA, B, C and D, since MasterD is the latest device at this moment, MasterD is index 3. Then Slave is re-connected with MaserB, since the latest device at this moment, MasterB is index 3.

Please pay attention to the case when more than four Master devices are paired: When Slave is successively paired with MasterA, B, C and D, if it's paired with a new device MasterE, the first paired device MasterA will be deleted automatically. When Slave is successively paired with MasterA, B, C and D, if Slave is re-connected with MasterA (the index sequence is B, C, D, A) and then paired with MasterE, pairing info of MasterB will be deleted.

Master device bonding information are stored in flash with format below:



```
typedef struct {
    u8      flag;
    u8      peer_addr_type;  //address used in link layer connection
    u8      peer_addr[6];

    u8      peer_key_size;
    u8      peer_id_adrType; //peer identity address information in key distribution, used to identify
    u8      peer_id_addr[6];

    u8      own_ltk[16];      //own_ltk[16]
    u8      peer_irk[16];
    u8      peer_csrk[16];

}smp_param_save_t;
```

Bonding info contains 64 bytes:

✧ "peer_addr_type" and "peer_addr" indicate connection address of Master in Link Layer, which will be used during device direct adv.

✧ "peer_id_adrType"/"peer_id_addr" and "peer_irk" indicate identity address and irk declared by Master during "key distribution" phase. Related info won't be added to resolving list, unless "peer_addr_type" and "peer_addr" are PRA (Resolvable Private Addr) and user needs to use adderess filter (Please refer to "TEST_WHITELIST" in 8267 feature test).

✧ Other parameters are negligible to user.

The API below serves to obtain device information from flash via “index”.

```
u32  blc_smp_param_loadByIndex(u8 index,
                               smp_param_save_t* smp_param_load);
```

If the return value is 0, it indicates info obtaining failure; if the return value is non-zero value, it indicates the starting address of the info in flash.

E.g. There are three bonding devices currently, to obtain info of the latest connected device, “index” should be set as “2”:

blc_smp_param_loadByIndex(2， …)

The API below serves to obtain information of bonding device from flash via Master address (connection addr in Link Layer).

```
u32  blc_smp_param_loadByAddr(u8 addr_type,

                    u8* addr, smp_param_save_t* smp_param_load);
```

If the return value is 0, it indicates info obtaining failure; if the return value is non-zero value, it indicates the starting address of the info in flash.

3.6 826x master customized pairing management

In 826x master kma dongle, if SMP is disabled, SDK cannot automatically implement operations such as pairing/un-pairing. Therefore, it’s needed to add pairing management in APP layer.

```
#define BLE_HOST_SMP_ENABLE                    0
```

A set of demo code is supplied in current SDK, based on which user can extend and modify as needed.

3.6.1 Design of Flash storage method

By default, sector of flash data area is 0x78000~0x78FFF, which is modifiable in “app_config.h”.

```
#define FLASH_ADR_PARING   0x78000
```

Starting from flash 0x78000, every 8 bytes constitute an area (8 bytes area). Each area can store one Slave MAC address: 1-byte mark, 1-byte address type, 6-byte MAC address.

```
typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;
```

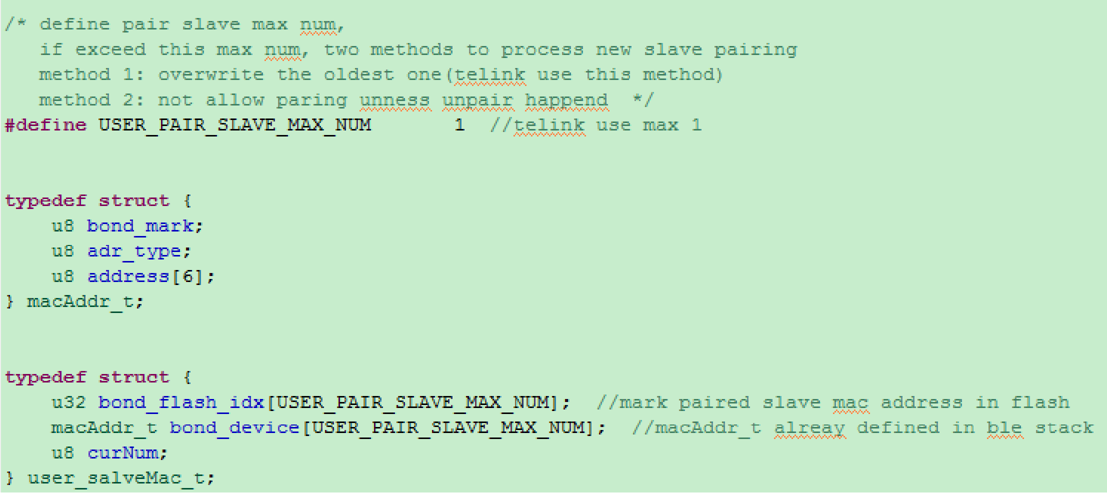
All valid Slave MAC addresses are stored in 8 bytes areas successively: The first valid MAC adderss is stored in 0x78000~0x78007, and the first byte mark of 0x78000 should be set as “0x5A” to indicate current addr is valid; the second valid MAC address should be stored in 0x78008~0x7800f, and the mark of 0x78008 should be set as “0x5A”; the third valid MAC address should be stored in 0x78010~0x78017, and the mark of 0x78010 should be set as “0x5A”……

To un-pair some Slave device, it’s needed to erase its MAC address in Dongle by setting the mark of the corresponding 8 bytes area as “0x00”. For example, to erase the MAC address of the first device as shown above, user should set 0x78000 as “0x00”.

When firmware is being executed, it’s not allowed to invoke the function “flash_erase_sector” to erase flash, since this operation takes 20~200ms to erase a 4K sector of flash and will result in BLE timing error. Therefore, the storage method of 8 bytes areas above is used to store MAC address.

Mark of “0x5A” and “0x00” are used to indicate pairing storage and un-pairing erasing of all Slave MAC addresses. Considering 8 bytes areas may occupy the whole sector 4K flash and thus result in error, a special processing is added during initialization: read valid MAC address information from 8 bytes areas starting from 0x78000, and deliver them to slave mac table in RAM. During this process, it will check whether there are too many 8 bytes areas. If so, the whole sector will be erased, the slave mac table in RAM will be re-writen into 8 bytes areas starting from 0x78000.

3.6.2 Slave Mac table



```
/* define pair slave max num,
   if exceed this max num, two methods to process new slave pairing
   method 1: overwrite the oldest one(telink use this method)
   method 2: not allow paring unness unpair happend  */
#define USER_PAIR_SLAVE_MAX_NUM       1  //telink use max 1


typedef struct {
    u8 bond_mark;
    u8 adr_type;
    u8 address[6];
} macAddr_t;


typedef struct {
    u32 bond_flash_idx[USER_PAIR_SLAVE_MAX_NUM];  //mark paired slave mac address in flash
    macAddr_t bond_device[USER_PAIR_SLAVE_MAX_NUM];  //macAddr_t alreay defined in ble stack
    u8 curNum;
} user_salveMac_t;
```

```
user_salveMac_t user_tbl_slaveMac;
```

The structure above serves to use slave mac table in RAM to maintain all paired devices.

The macro "USER_PAIR_SLAVE_MAX_NUM" serves to set the number of maintainable paired devices. Its default value is 1, which indicates one paired device is maintainable. User can modify this value as needed.

Suppose user needs to maintain three devices, "USER_PAIR_SLAVE_MAX_NUM" should be set as 3. In "user_tbl_slaveMac", "curNum" indicates the number of current valid Slave devices in flash; "bond_flash_idx" array records corresponding offset based on 0x78000 for starting address of each 8 bytes area in flash (during device un-pairing, based on corresponding offset, the mark of 8 bytes area can be found and it should be set as "0x00"); "bond_device" array record MAC addresses.

3.6.3 API

Based on the design of flash storage method and slave mac table, a few APIs are supplied, as shown below.

3.6.3.1 user_tbl_slave_mac_add

```
int user_tbl_slave_mac_add(u8 adr_type, u8 *adr);
```

The API above should be invoked when there's new device paired, and it serves to add a Slave MAC address.

The return value should be either 1 (success) or 0 (failure).

First this function will check whether current number of devices in flash and slave mac table has reached the maximum.

- If not, the MAC address of new device will be added into slave mac table, and stored in an 8 bytes area of flash.
- If the number has reached the maximum, the processing policy can be "pairing is not allowed" or "the oldest device is directly deleted". In Telink demo, the latter policy is adopted, since the maximum number of paired device is set as 1, the "user_tbl_slave_mac_delete_by_index(0)" should be used to delete current paired device, then the new device can be added into slave mac table.

  User can modify the implementation of this function according to his policy.

3.6.3.2 user_tbl_slave_mac_search

```
int user_tbl_slave_mac_search(u8 adr_type, u8 * adr)
```

The API above serves to search device according to device address of adv report, and check whether slave mac table contains this device. That is: It will check whether current device sending adv packet is ever paired with Master; if the device is ever paired, it can be directly connected.

3.6.3.3 user_tbl_slave_mac_delete_by_adr

```
int user_tbl_slave_mac_delete_by_adr(u8 adr_type, u8 *adr)
```

The API above serves to delete a paired device from slave mac table by a specified address.

3.6.3.4 user_tbl_slave_mac_delete_by_index

```
void user_tbl_slave_mac_delete_by_index(int index)
```

The API above serves to delete a paired device from slave mac table by a specified index. “Index” value indicates device pairing sequence. If the maximum pairing number is 1, the index of paired device is always 0; if the maximum pairing number is 2, the index of the first paired device is 0, and the index of the second paired device is 1……

3.6.3.5 user_tbl_slave_mac_delete_all

```
void user_tbl_slave_mac_delete_all(void)
```

The API above serves to delete all paired devices from slave mac table.

3.6.3.6 user_tbl_salve_mac_unpair_proc

```
void user_tbl_salve_mac_unpair_proc(void)
```

The API above serves to process un-pairing command. The processing method in reference code about “delete all paired devices” corresponds to the default maximum pairing number (1).

3.6.4 Connection and pairing

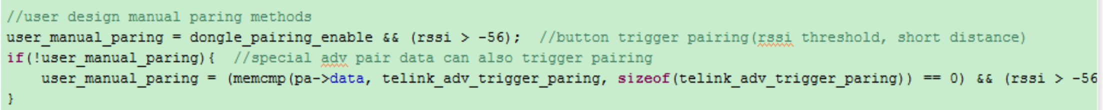
When Master receives adv packet reported by Controller, it will connect with Slave in the following cases:

1) The function “user_tbl_slave_mac_search” is invoked to check whether current device is already paired with Master and un-pairing is not implemented. If yes, connection can be established automatically.

   master_auto_connect = user_tbl_slave_mac_search(pa->adr_type, pa->mac);

   if(master_auto_connect) { create connection }

2) If current adv device is not in slave mac table, auto connection won't be initiated, thus it's needed to check whether the manual pairing condition is met. In SDK, if current adv device is near enough, two manual pairing solutions are supported by default: pairing button on Master dongle is pressed; current adv data is Telink-defined pairing adv packet data. Following is the reference code:



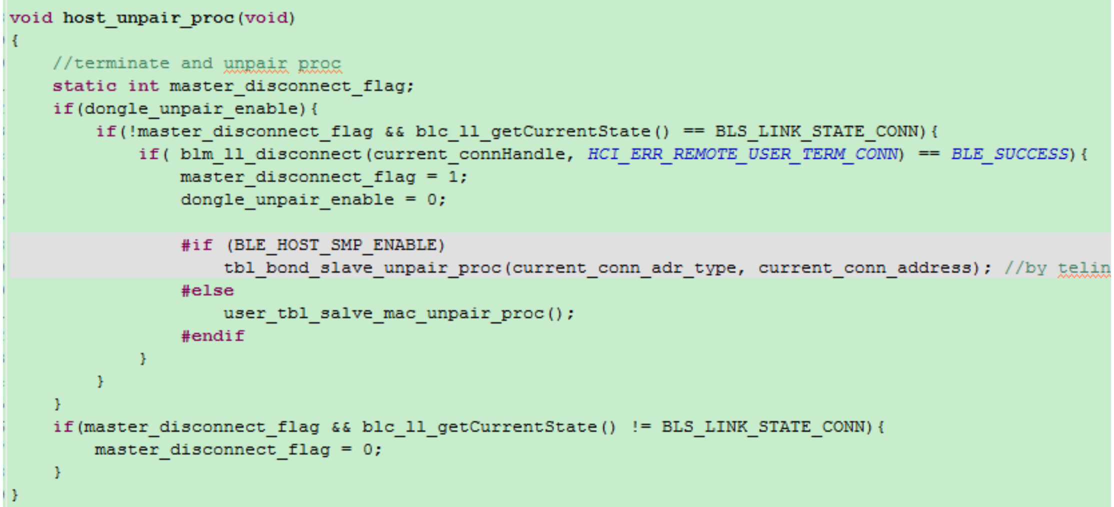
```
//user design manual paring methods
user_manual_paring = dongle_pairing_enable && (rssi > -56);  //button trigger pairing(rssi threshold, short distance)
if(!user_manual_paring){  //special adv pair data can also trigger pairing
    user_manual_paring = (memcmp(pa->data, telink_adv_trigger_paring, sizeof(telink_adv_trigger_paring)) == 0) && (rssi > -56
}
```

if(user_manual_paring) { create connection }

If connection is triggered by manual pairing, after connection is established successfully, i.e. when "HCI LE CONECTION ESTABLISHED EVENT" is reported, current device will be added into slave mac table.

```
// if this connection establish is a new device manual paring, should add this device to slave table
if(user_manual_paring && !master_auto_connect){
        user_tbl_slave_mac_add(pc->peer_adr_type, pc->mac);

}
```

3.6.5 Un-pairing



```
void host_unpair_proc(void)
{
    //terminate and unpair proc
    static int master_disconnect_flag;
    if(dongle_unpair_enable){
        if(!master_disconnect_flag && blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){
            if( blm_ll_disconnect(current_connHandle, HCI_ERR_REMOTE_USER_TERM_CONN) == BLE_SUCCESS){
                master_disconnect_flag = 1;
                dongle_unpair_enable = 0;

                #if (BLE_HOST_SMP_ENABLE)
                    tbl_bond_slave_unpair_proc(current_conn_adr_type, current_conn_address); //by telin
                #else
                    user_tbl_salve_mac_unpair_proc();
                #endif
            }
        }
    }
    if(master_disconnect_flag && blc_ll_getCurrentState() != BLS_LINK_STATE_CONN){
        master_disconnect_flag = 0;
    }
}
```

As shown in the reference code above, when un-pairing condition is triggered, Master will invoke "blm_ll_disconnect" to terminate connection, then invoke "user_tbl_salve_mac_unpair_proc" to process un-pairing. Demo code will directly delete all paired devices, since the maximum number of paired devices is 1 by default, only one device will be deleted. If the configured maximum number exceeds 1, “user_tbl_slave_mac_delete_by_adr” or “” should be invoked to delete specific device.

In Demo code, un-pairing condition will be triggered in the following two cases:

✧ Un-pairing button on Master dongle is pressed;

✧ Un-pairing key value “0xFF” is received in “HID keyboard report service”.

User can modify the trigger condition of un-pairing as needed.

4 Power Management (PM)

4.1 PM driver

Driver files related to PM are available in proj_lib/pm_826x.h, proj_lib/pm_826x.c, proj/mcu_spec/gpio_826x.h and proj/mcu_spec/gpio_826x.c.

4.1.1 Low power modes

826x MCU supports three basic modes.

1) Working mode: In this mode, MCU executes firmware, hardware digital modules work normally, related analog modules and BLE RF transceiver can be enabled depending on firmware. The current in this mode is about 10~30mA.

2) Suspend mode: Low power mode 1. In this mode, firmware stops running, similar to suspend function. Most hardware modules in IC are powered down, while the PM module still works normally. All digital registers, analog registers and memory are non-volatile in this mode, i.e. all data and states are held and won't be lost. The pure IC current in this mode is about 7~8uA. After wakeup from suspend, firmware continues running from the break point.

3) Deepsleep mode: Low power mode 2. In this mode, firmware stops running, the vast majority of hardware modules in IC are powered down, while the PM module still works. Only a few retention analog registers are non-volatile in this mode; other (digital and analog) registers and memory are volatile, i.e. all data won't be held. The retention analog registers (DEEP_ANA_REG in pm_826x.h) can be used to store some necessary information. After wakeup from deepsleep, MCU is rebooted, and it's equivalent to power cycle (power cycle will reset all registers); firmware restarts running and enters initialization. User can store some information in DEEP_ANA_REG before MCU enters deepsleep. Then user can judge whether it's pure power cycle or wakeup from deepsleep, by reading retention analog registers during initialization and checking whether there're pre-configured information. The pure IC current in this mode is about 0.7uA; if internal flash current (~1uA) is added, the total current is about 1.7uA.

As introduced in Link Layer timing sequence, during each Adv Interval / Connection Interval, MCU works with low duty cycle and enters suspend after tasks are processed. Since MCU stays in suspend state at most time and current in suspend is very low, the average current is decreased largely to enable low power.

When MCU does not need to work, it can be configured to enter deepsleep to minimize power, and certain sources can be configured to wake up MCU.

4.1.2 Hardware wakeup sources

Figure4-1 shows wakeup sources available for 826x MCU: In suspend mode, it can be woke up by CORE and timer sources; while in deepsleep mode, it can be woke up by PAD and timer sources. In 826x BLE SDK, the following three types of wakeup sources are selectable.

```
enum {

    PM_WAKEUP_PAD   = BIT(4),

    PM_WAKEUP_CORE  = BIT(5),

    PM_WAKEUP_TIMER = BIT(6),

};
```

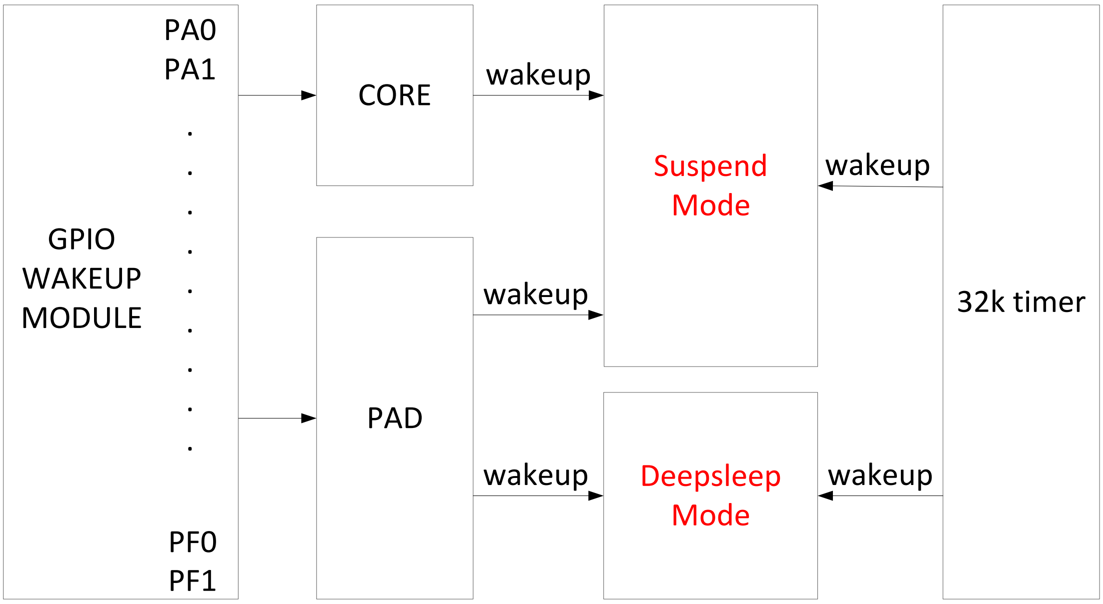


Figure4-1 Hardware wakeup sources for 826x MCU

As shown above, MCU can be woke up from low-power mode (suspend or deepsleep) by hardware wakeup source TIMER, CORE or PAD.

The wakeup source "PM_WAKEUP_TIMER" is derived from hardware 32kHz RC timer. This timer is correctly initialized in SDK, and user only needs to set this wakeup source in "cpu_sleep_wakeup()".

The two wakeup sources including "PM_WAKEUP_CORE" and "PM_WAKEUP_PAD" are derived from GPIO. High/Low level of all GPIOs can be configured to wakeup MCU from suspend/deepsleep via the CORE/PAD module. The CORE module can only wakeup MCU from suspend, while the PAD module can wakeup MCU from both suspend and deepsleep. However, in 826x BLE SDK, GPIO CORE is appointed as wakeup source for suspend, while GPIO PAD is appointed as wakeup source for deepsleep.

The APIs below can be invoked to enable high or low level wakeup of certain GPIO source.

1) Configure GPIO CORE as wakeup source for suspend:

```
void gpio_set_wakeup(u32 pin, u32 level, int en);
```

Note: “pin” indicates GPIO pin; “level” indicates wakeup trigger level, 1-high level wakeup, 0-low level wakeup; “en”: 1-enable, 0-disable.

Examples:

```
gpio_set_wakeup(GPIO_PC2,   1,   1);   // Enable GPIO_PC2 CORE high level wakeup
gpio_set_wakeup(GPIO_PC2,   1,   0);   // Disable GPIO_PC2 CORE wakeup
gpio_set_wakeup(GPIO_PB5,   0,   1);   // Enable GPIO_PB5 CORE low level wakeup
gpio_set_wakeup(GPIO_PB5,   0,   0);   // Disable GPIO_PB5 CORE wakeup
```

2) Configure GPIO PAD as wakeup source for deepsleep:

```
void cpu_set_gpio_wakeup (int pin, int pol, int en);
```

Note: “pin” indicates GPIO pin; “pol” indicates wakeup trigger polarity, 1-high level wakeup, 0-low level wakeup; en: 1-enable, 0-disable.

Examples:

```
cpu_set_gpio_wakeup (GPIO_PC2,   1,   1);   // Enable GPIO_PC2 PAD high level wakeup
cpu_set_gpio_wakeup (GPIO_PC2,   1,   0);   // Disable GPIO_PC2 PAD wakeup
cpu_set_gpio_wakeup (GPIO_PB5,   0,   1);   // Enable GPIO_PB5 PAD low level wakeup
cpu_set_gpio_wakeup (GPIO_PB5,   0,   0);   // Disable GPIO_PB5 PAD wakeup
```

4.1.3 Low power mode entry and wakeup

The API “cpu_sleep_wakeup” in “proj_lb/pm_826x.h” can be invoked to configure MCU to enter low power mode and set wakeup source(s).

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

Notes:

1) “deepsleep”: 0-enter suspend, 1-enter deepsleep.
2) “wakeup_src”: It’s used to configure wakeup source(s) for current suspend/deepsleep, and it’s selectable from PM_WAKEUP_PAD, PM_WAKEUP_CORE and PM_WAKEUP_TIMER correspondingly. Note that

PM_WAKEUP_TIMER and PM_WAKEUP_CORE can be used as wakeup source for suspend, while PM_WAKEUP_TIMER and PM_WAKEUP_PAD can be used as wakeup source for deepsleep. If wakeup_src is set as 0, MCU can't be woke up after it enters low power mode.

3) "wakeup_tick": If the PM_WAKEUP_TIMER is not configured, this parameter is invalid. Only when the PM_WAKEUP_TIMER is configured in the wakeup_src, the wakeup_tick (absolute value) needs to be configured as current system tick plus sleep time tick, and it determines when MCU will be woke up by timer. When system tick value matches the configured wakeup_tick, MCU is woke up from low power mode. If the wakeup_tick is directly configured not considering system tick, wakeup time can't be effectively controlled.

The absolute wakeup_tick value must be within the range of 32-bit system tick, the maximum sleep time configured by this API is limited. In current design, maximum sleep time is set as 32bit max system tick * 3/4. For 16MHz clock, max system tick is 268s, the maximum suspend/deepsleep should be 268s*3/4=201s

The int return value is one or "logic or" result of the five values in the enum below.

```
enum {
    WAKEUP_STATUS_COMP   = BIT(0),
    WAKEUP_STATUS_TIMER  = BIT(1),
    WAKEUP_STATUS_CORE   = BIT(2),
    WAKEUP_STATUS_PAD    = BIT(3),

    STATUS_GPIO_ERR_NO_ENTER_PM  = BIT(7),

};
```

1) WAKEUP_STATUS_COMP is never used in BLE SDK, and it's negligible to user.

2) *WAKEUP_STATUS_TIMER/ WAKEUP_STATUS_CORE/ WAKEUP_STATUS_PAD* correspond to *PM_WAKEUP_TIMER/ PM_WAKEUP_CORE/ PM_WAKEUP_PAD*, which indicate wakeup source to trigger current low power mode.

3) *STATUS_GPIO_ERR_NO_ENTER_PM* is a special state, and indicates a GPIO wakeup error occurs currently. E.g. When a GPIO CORE high level wakeup is configured, when this GPIO is high level, it tries to invoke "cpu_sleep_wakeup" to enter suspend and wakeup source is set as "PM_WAKEUP_CORE". In this case, MCU cannot enter suspend, but will exit "cpu_sleep_wakeup" immediately and return the value "*STATUS_GPIO_ERR_NO_ENTER_PM*".

4) Note: The return value may be (*WAKEUP_STATUS_TIMER | WAKEUP_STATUS_CORE*) and it indicates two wakeup sources take effect simultaneously.

Generally the following method is used to control sleep time:

cpu_sleep_wakeup (0, PM_WAKEUP_TIMER, clock_time() + delta_Tick);

"delta_Tick" is a relative time (e.g. 100*CLOCK_SYS_CLOCK_1MS). The result of "clock_time()" plus "delta_Tick" is absolute time.

cpu_sleep_wakeup usage examples:

1) cpu_sleep_wakeup (0, PM_WAKEUP_CORE, 0);

   MCU enters suspend mode when this function is executed, and it can be woke up by GPIO CORE only.

2) cpu_sleep_wakeup (0, PM_WAKEUP_TIMER, clock_time() + 10**CLOCK_SYS_CLOCK_1MS*);

   MCU enters suspend mode when this function is executed, and it can be woke up by TIMER only; suspend time is 10ms, i.e. wakeup time is function execution moment plus 10ms.

3) cpu_sleep_wakeup (0, PM_WAKEUP_CORE | PM_WAKEUP_TIMER,clock_time() + 50**CLOCK_SYS_CLOCK_1MS*);

   MCU enters suspend mode when this function is executed, and it can be woke up by GPIO CORE and TIMER. Timer wakeup time is set as 50ms relative to function execution moment; if GPIO CORE wakeup is triggered before 50ms expires, MCU will be woke up by GPIO, otherwise MCU will be woke up by Timer.

4) cpu_sleep_wakeup (1, PM_WAKEUP_PAD, 0);

   MCU enters deepsleep mode when this function is executed, and it can be woke up by GPIO PAD.

5) cpu_sleep_wakeup (1, PM_WAKEUP_TIMER, clock_time() + 8* *CLOCK_SYS_CLOCK_1S)*;

   MCU enters deepsleep mode when this function is executed, and it can be woke up by Timer. Deep sleep time is 8s.

6) cpu_sleep_wakeup (1, PM_WAKEUP_PAD | PM_WAKEUP_TIMER,clock_time() + 10**CLOCK_SYS_CLOCK_1S*);

   MCU enters deepsleep mode when this function is executed, and it can be woke up by GPIO PAD and Timer. Timer wakeup time is 10s relative to function execution moment. If GPIO PAD wakeup is triggered before 10s expires, MCU will be woke up by GPIO, otherwise MCU will be woke up by Timer.

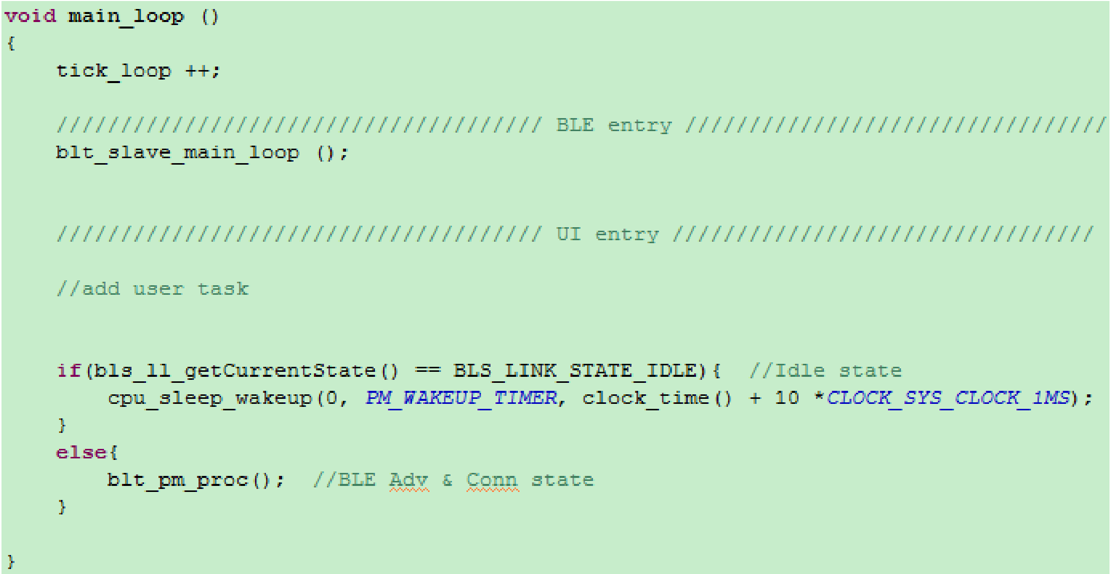
4.2 BLE low power management

In 826x BLE SDK, low power management is implemented via power management of Link Layer.

In current Telink BLE SDK, stack bottom layer only implements low power management for Advertising state and Connection state Slave role, and a set of APIs are supplied for user. As for other states, low power management is not directly supplied, or it's needed to invoke PM driver to implement PM, e.g. PM in Idle state.

4.2.1 PM in Idle state

When Link Layer is in Idle state, the "blt_sdk_main_loop" does not execute any operation, and low power management is not supplied in SDK. User needs to invoke the API "cpu_sleep_wakeup()" to implement low power management, i.e. configure MCU to enter suspend or deepsleep mode, and set wakeup source correspondingly.



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

    ////////////////////////////////////// BLE entry /////////////////////////////////
    blt_slave_main_loop ();


    ////////////////////////////////////// UI entry /////////////////////////////////

    //add user task


    if(bls_ll_getCurrentState() == BLS_LINK_STATE_IDLE){  //Idle state
        cpu_sleep_wakeup(0, PM_WAKEUP_TIMER, clock_time() + 10 *CLOCK_SYS_CLOCK_1MS);
    }
    else{
        blt_pm_proc();  //BLE Adv & Conn state
    }

}
```

Figure4-2 PM in Link Layer Idle state

The figure above shows simple reference code: When Link Layer is in Idle state, there's 10ms suspend during each mainloop.

In Idle state, MCU can also enter deepsleep mode directly.

4.2.2 PM in BLE Adv state & Conn state

When Link Layer is in Advertising state or Conn state Slave role:

1) In Advertising state, during each Adv Interval, the remaining time except for Adv Event can be used to process UI task or enter suspend.

2) In Conn state Slave role, during each Conn interval, the remaining time except for Brx Event (brx start + brx working + brx post) can be used to process UI task or enter suspend.

Actually BLE PM includes the management of the UI task/suspend duration. User can manage this duration, and determine whether to run UI task or enter suspend to save power.

BLE PM does not include the management of deepsleep. User can directly invoke "cpu_sleep_wakeup" in UI layer to enter deepsleep.

BLE PM does not need user to directly invoke the API "cpu_sleep_wakeup" in PM driver layer. In BLE stack part of 826x BLE SDK, according to states and low power modes of Link Layer, a PM mechanism is supplied (code is in "blt_sdk_main_loop"). User only needs to invoke corresponding API to configure and manage low power.

4.3 BLE PM configuration

4.3.1 PM module initialization

Similar to the design of Link Layer state machine, it's needed to enable PM module during initialization by invoking the API below.

```
void        blc_ll_initPowerManagement_module(void);
```

Applications without the need of PM won't invoke this API, thus PM related code and variables won't be compiled to firmware, and resources can be saved.

4.3.2 Set low power mode via "bls_pm_setSuspendMask"

The API below serves to configure PM for Link Layer Advertising state and Conn state in 826x BLE SDK:

```
void        bls_pm_setSuspendMask (u8 mask);

u8          bls_pm_getSuspendMask (void);
```

By using the "bls_pm_setSuspendMask", a bottom-layer variable "SuspendMask" is set to configure low power mode. Actually the variable in code is "bls_pm.suspend_mask", and its default value is "SUSPEND_DISABLE".

The "bls_pm_getSuspendMask" serves to obtain current SuspendMask value, which equals the value configured by previous invoked "bls_pm_setSuspendMask". If the variable is not configured, the value equals the default "SUSPEND_DISABLE".

The SuspendMask is selectable from the values below.

```
////////////////// Power Management ///////////////////////
```

```
#define          SUSPEND_DISABLE          0
#define          SUSPEND_ADV              BIT(0)
#define          SUSPEND_CONN             BIT(1)

#define          MCU_STALL                BIT(6)
```

MCU_STALL is a special mode and it will be introduced later.

Please refer to Link Layer timing sequence (section 3.2.4) and working mechanism of low power management (section 4.3.4) to help understand the configuration of "bls_pm_setSuspendMask".

SuspendMask can be selectable as any one of the values above, or combination value ("or" operation) of Advertising state and Conn state, as shown below:

bls_pm_setSuspendMask(SUSPEND_ADV);

bls_pm_setSuspendMask(SUSPEND_CONN);

bls_pm_setSuspendMask(MCU_STALL);

bls_pm_setSuspendMask(SUSPEND_DISABLE);

bls_pm_setSuspendMask(SUSPEND_ADV | SUSPEND_CONN);

4.3.3 Set low power wakeup source via "bls_pm_setWakeupSource"

MCU can enter suspend or deepsleep by invoking the "bls_pm_setSuspendMask". The API below serves to set corresponding wakeup souce.

```
void        bls_pm_setWakeupSource(u8 source);
```

This API sets a bottom-layer variable "WakeupSource". Actually the variable in code is "bls_pm.wakeup_src".

The WakeupSource is selectable from PM_WAKEUP_PAD, PM_WAKEUP_CORE, PM_WAKEUP_TIMER or corresponding combination ("or" operation).

If MCU enters suspend mode from Advertising state or Conn state Slave role, actual system wakeup source should be:

WakeupSource | PM_WAKEUP_TIMER

The "PM_WAKEUP_TIMER" is necessary and it does not depend on user configuration, which ensures MCU can be woke up to process Adv Event or Brx Event.

The wakeup source configured by "bls_pm_setWakeupSource" only applies to current low power mode; once MCU is woke up from suspend/deepsleep, the WakeupSource will be cleared in bottom layer and become invalid. It's needed to re-configure the wakeup source for each subsequent low power mode.

4.3.4 Working mechanism of low power managment

To help understand the configuration of "SuspendMask" and "WakeupSource", this section will introduce the principle of low power management mechanism.

In SDK, mainloop is a structure of while(1):

```
while(1)
{
    blt_sdk_main_loop();
    //UI    task
}
```

As long as UI task does not take especially long time, the "blt_sdk_main_loop" including the code of BLE low power management mechanism will always be executed.

Corresponding to BLE Link Layer timing sequence (section 3.2.4), two time parameters are defined: "T_advertising" indicates the start time of Link Layer Adv Event in Advertising state; "T_brx" indicates the start time of Link Layer Brx Event in Conn state Slave role.

In blt_sdk_main_loop, the pseudo code corresponding to low power management is shown as below:

```
int  blt_sdk_main_loop (void)
{
    ……
    if(SuspendMask == SUSPEND_DISABLE)   // SUSPEND_DISABLE, not enter low power mode
    {
        return    0;
    }

    if(Link Layer State is in Advertising state or Conn state Slave role)
    {
        if(Link Layer is in Adv Event or Brx Event)  // BLE packet transfer is onging, not enter low power mode
        {
            return    0;
        }
        else
        {
            blt_brx_sleep ();//suspend & wakeup processing function

        }
    }
    return 1;
}

void    blt_brx_sleep (void)
{
    if( (Link Layer state == Adv state)&& (SuspendMask&SUSPEND_ADV) )
    {   // Enter suspend from current Adv state
        Execute callback function of event "BLT_EV_FLAG_SUSPEND_ENTER"
        cpu_sleep_wakeup (0, PM_WAKEUP_TIMER | WakeupSource,
                    T_advertising + advInterval); //suspend
        Execute callback function of event "BLT_EV_FLAG_SUSPEND_EXIT"

        if(current suspend is woke up by GPIO CORE in advance)
        {
            Execute callback function of event
"BLT_EV_FLAG_GPIO_EARLY_WAKEUP"
            Re-enter suspend, until wakeup at
"T_advertising+advInterval"
        }
    }
    else if((Link Layer state == Conn state Slave role)&&
        (SuspendMask&SUSPEND_CONN) )// Enter suspend from current Conn
state
    {
        u32 wakeup_tick;
        if(conn_latency is not 0) //conn_latency != 0
        {
            // refer to section 4.4 for latency_use
            u16 latency_use = bls_calculateLatency();
            wakeup_tick = T_brx + (latency_use+1) * conn_interval;
        }
        else  //conn_latency == 0
        {
            wakeup_tick = T_brx + conn_interval;
        }

        Execute callback function of event "BLT_EV_FLAG_SUSPEND_ENTER"
        cpu_sleep_wakeup(0,PM_WAKEUP_TIMER|WakeupSource, wakeup_tick);
        Execute callback function of event "BLT_EV_FLAG_SUSPEND_EXIT"

        if(current suspend is woke up by GPIO CORE in advance)
        {
            Execute callback function of event
"BLT_EV_FLAG_GPIO_EARLY_WAKEUP"
            BLE timing sequence adjustment related processing
        }
    }

    // clear low power configuration parameters related to user
    WakeupSource= 0;// clear wakeup source configuration
    user_latecny = 0xffff;
}
```

4.4 "latency_use" configuration and calculation

As introduced in working mechanism of low power management (section 4.3.4), if the "suspendMask" is set as "SUSPEND_CONN" in Conn state Slave role, the actual wakeup time should be:

wakeup_tick = T_brx + (latency_use+1) * conn_interval;

"T_brx": Brx Event Rx time during current interval.

If "latency_use" is 0, MCU must be woke up during next interval to listen for packets; if "latency_use" is not 0, MCU can skip "latency_use" intervals to save power.

```
latency_use = bls_calculateLatency();
```

The calculation of "latency_use" involves a "user_latency" with configurable value:

```
void bls_pm_setManualLatency(u16 latency)
{
    user_latency = latency;
}
```

"latency_use" calculation process is shown as below.

First calculate system latency:

1) If connection latency in current connection parameters is 0, system latency would be 0.
2) If connection latency in current connection parameters is not 0:
   A. If current system has unfinished task (e.g. there are data to be sent, or there are data received from Master to be processed), MCU must be woke up during next interval to continue the task, so system latency should be 0.
   B. If current system has no task to process, system latency should equal connection latency except in the case below: If "update map request" or "update connection parameter request" is received from Master, and the actual update moment is earlier than (connection latency+1) intervals, the actual system latency would ensure MCU is woke up during the interval before the actual update moment, so as not to disturb BLE timing sequence.

Acutally the eventual latency_use equals min(system latency, user_latency), i.e. the minimum value of system latency and user_latency.

If the latency manually configured by invoking "bls_pm_setManualLatency" during UI entry is smaller than system latency, it can be used as the eventual latency_use. It only applies to non-zero system latency.

Note that the final sentence of each "blt_sdk_main_loop" will set "user_latency" as "0xffff". Therefore, the user latency configured by invoking "bls_pm_setManualLatency" only applies to the current suspend.

4.5 Other APIs

4.5.1 bls_pm_getSystemWakeupTick

The API below serves to obtain suspend wakeup time (system tick value) calculated by PM module.

u32 bls_pm_getSystemWakeupTick(void);

According to section 4.3.4, this API can be invoked only in the callback function of "BLT_EV_FLAG_SUSPEND_ENTER" event.

When the "blt_brx_sleep" function is executed by PM module, the suspend wakeup time is calculated according to current Link Layer state and the "SuspendMask" set in APP layer. APP layer can read this value only via the callback function of "BLT_EV_FLAG_SUSPEND_ENTER" event.

For example, MCU needs to enter suspend from Conn state, and conn latency is not 0:

```
u16 latency_use = bls_calculateLatency();

wakeup_tick = T_brx + (latency_use+1) * conn_interval;

cpu_sleep_wakeup(0,PM_WAKEUP_TIMER|WakeupSource, wakeup_tick);
```

APP layer can't predict in advance the latency_use calculated by "bls_calculateLatency" and thus does not know the actual wakeup_tick; the wakeup time can be obtained only by invoking "bls_pm_getSystemWakeupTick" in the callback function of "BLT_EV_FLAG_SUSPEND_ENTER" event.

Following is a key scan application example to illustrate the usage of "BLT_EV_FLAG_SUSPEND_ENTER" callback function and "bls_pm_getSystemWakeupTick".

```
bls_app_registerEventCallback(   BLT_EV_FLAG_SUSPEND_ENTER,
                                 &ble_remote_set_sleep_wakeup);
```

```
voidble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( bls_ll_getCurrentState() == BLS_LINK_STATE_CONN &&
    ((u32)(bls_pm_getSystemWakeupTick() - clock_time())) >
                80 * CLOCK_SYS_CLOCK_1MS)
    {
        //gpio CORE wakeup suspend

        bls_pm_setWakeupSource(PM_WAKEUP_CORE);
    }

}
```

The callback function is used to avoid key press loss. Generally, a key press lasts for hundreds of milliseconds~100ms. When "bls_pm_setSuspendMask" configures MCU to enter suspend from both Advertising state and Conn state, if "Conn Latency" is not enabled (0), as long as Adv interval and Conn interval is not especially large (generally set as a value not exceeding 100ms), suspend time won't exceed Adv interval and Conn interval; since it can ensure key scan frequency, key press loss can be avoided. In this case, GPIO wakeup is not configured, so that key press won't wakeup MCU.

However, if latency is enabled, (e.g. conn_interval is 10ms, latency is 99), suspend may last for 1s in Conn state. During this process, there may be key press lost. Check in the "BLT_EV_FLAG_SUSPEND_ENTER" callback, if current state is Conn state, and wakeup time for the following suspend is more than 80ms from current time, GPIO CORE wakeup will be added. If timer wakeup is not triggered yet, and GPIO level changes due to key press, MCU wakeup is triggered in advance, so that key press won't be lost and key scan task can be processed.

4.5.2 bls_pm_enableAdvMcuStall

The API below serves to decrease peak current during advertising.

```
void        bls_pm_enableAdvMcuStall(u8 en);
```

"en": 1-Enable MCU stall; 0-Disable MCU stall.

Note: Timer0 is used in stack bottom layer to implement MCU stall during advertising. If this power optimization is added, APP layer should use Timer1/Timer2 rather than Timer0.

4.5.3 cpu_sleep_wakeup2

The API below serves to set long deep or suspend time, and wake up automatically when setting time is over. The max time interval supports by this API is 71 minutes.

```
int cpu_sleep_wakeup2(int deepsleep,
                      int wakeup_src,
                      unsigned long SleepDurationUs);
```

Please be noted, the unit of parameter SleepDurationUs is us, and it is an abosulte value, i.e., just input sleep duration (differ from relatively value of `cpu_sleep_wakeup`). E.g., if the sleep duration is 10 min,

```
cpu_sleep_wakeup2(DEEPSLEEP_MODE, PM_WAKEUP_TIMER, 10*60*1000*1000);
```

4.6 Notes about GPIO wakeup

4.6.1 Fail to enter suspend/deepsleep when wakeup level is valid

Since 826x CORE/PAD wakeup is triggered by high/low level rather than positive/negative edge, after GPIO CORE or PAD source is configured, e.g. MCU is configured to wake up from suspend by high level of certain GPIO CORE, the GPIO input must be low level when MCU invokes "cpu_wakeup_sleep" to enter suspend. If the GPIO is already high level input currently, the configuration won't take effect, and Slave doesn't enter suspend. This also applies to GPIO PAD wakeup.

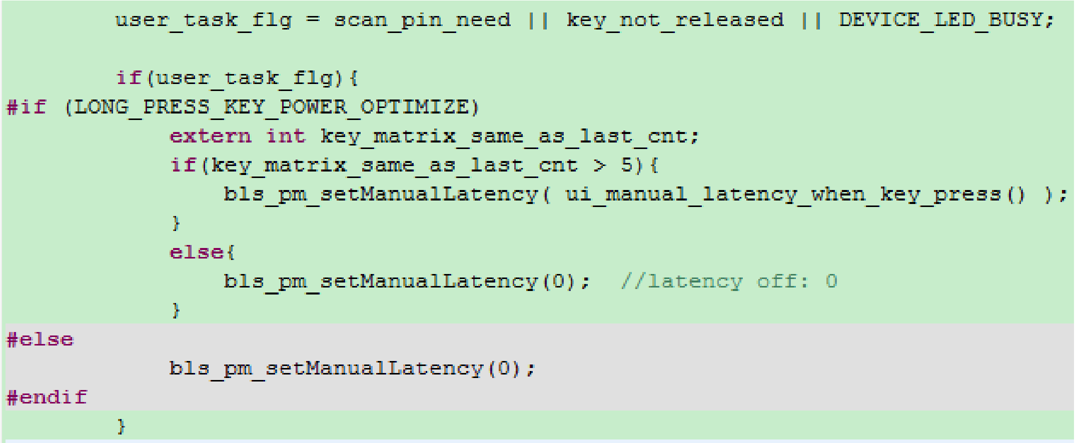
The situation above may lead to unexpected problems. For example, MCU is expected to enter deepsleep and execute firmware after wakeup; however, MCU can't enter deepsleep and continues to execute the code unexpectedly, thus firmware running flow may be messed.

In code of 826x ble remote, a solution is given to solve the problem.

Via configuration in "BLT_EV_FLAG_SUSPEND_ENTER", GPIO CORE wakeup won't be enabled unless suspend time exceeds the specified time (e.g. 80ms).

```
void ble_remote_set_sleep_wakeup (u8 e, u8 *p, int n)
{
    if( bls_ll_getCurrentState() == BLS_LINK_STATE_CONN &&
    ((u32)(bls_pm_getSystemWakeupTick() - clock_time())) >
                80 * CLOCK_SYS_CLOCK_1MS)
    {
        bls_pm_setWakeupSource(PM_WAKEUP_CORE);
    }

}
```

When there's key not released, user can ensure suspend time won't exceed 80ms by manually setting latency as 0 or a small value, thus GPIO CORE high-level wakeup won't be enabled with key held (high level in drive pin). The sample code is shown as below:



```
        user_task_flg = scan_pin_need || key_not_released || DEVICE_LED_BUSY;

        if(user_task_flg){
#if (LONG_PRESS_KEY_POWER_OPTIMIZE)
            extern int key_matrix_same_as_last_cnt;
            if(key_matrix_same_as_last_cnt > 5){
                bls_pm_setManualLatency( ui_manual_latency_when_key_press() );
            }
            else{
                bls_pm_setManualLatency(0);  //latency off: 0
            }
#else
            bls_pm_setManualLatency(0);
#endif
        }
```

MCU will enter deepsleep in following cases:

1) There is no task (including key press task) for successive 60s duration. In this case, the problem MCU can’t enter deepsleep due to high level from drive pin can be avoided.

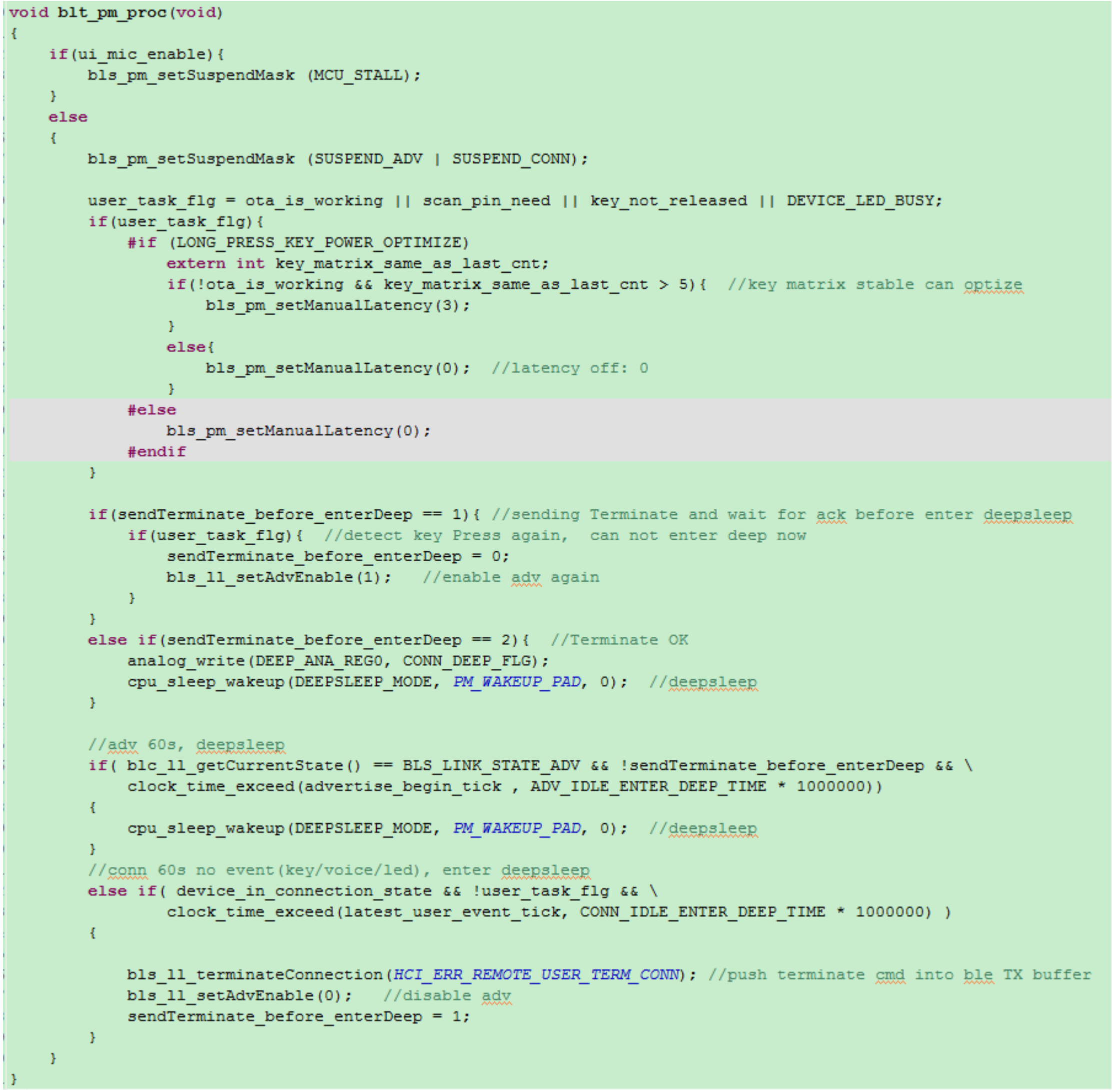
2) Some button is stuck for 60s. In this case, though high level is input in drive pin, by inverting the polarity of the stuck drive pin to low-level wakeup, MCU is allowed to enter deepsleep (refer to section 7.7).

User should pay attention to this problem when using Telink GPIO CORE/PAD wakeup.

4.7 BLE system PM reference

As introduced above, user can flexibly configure low power management in UI entry.

In this section, low power management code sample of 826x remote (audio RC) is given for user reference.



```
void blt_pm_proc(void)
{
    if(ui_mic_enable){
        bls_pm_setSuspendMask (MCU_STALL);
    }
    else
    {
        bls_pm_setSuspendMask (SUSPEND_ADV | SUSPEND_CONN);

        user_task_flg = ota_is_working || scan_pin_need || key_not_released || DEVICE_LED_BUSY;
        if(user_task_flg){
            #if (LONG_PRESS_KEY_POWER_OPTIMIZE)
                extern int key_matrix_same_as_last_cnt;
                if(!ota_is_working && key_matrix_same_as_last_cnt > 5){  //key matrix stable can optize
                    bls_pm_setManualLatency(3);
                }
                else{
                    bls_pm_setManualLatency(0);  //latency off: 0
                }
            #else
                bls_pm_setManualLatency(0);
            #endif
        }

        if(sendTerminate_before_enterDeep == 1){ //sending Terminate and wait for ack before enter deepsleep
            if(user_task_flg){  //detect key Press again,  can not enter deep now
                sendTerminate_before_enterDeep = 0;
                bls_ll_setAdvEnable(1);   //enable adv again
            }
        }
        else if(sendTerminate_before_enterDeep == 2){  //Terminate OK
            analog_write(DEEP_ANA_REG0, CONN_DEEP_FLG);
            cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);  //deepsleep
        }

        //adv 60s, deepsleep
        if( blc_ll_getCurrentState() == BLS_LINK_STATE_ADV && !sendTerminate_before_enterDeep && \
            clock_time_exceed(advertise_begin_tick , ADV_IDLE_ENTER_DEEP_TIME * 1000000))
        {
            cpu_sleep_wakeup(DEEPSLEEP_MODE, PM_WAKEUP_PAD, 0);  //deepsleep
        }
        //conn 60s no event(key/voice/led), enter deepsleep
        else if( device_in_connection_state && !user_task_flg && \
                clock_time_exceed(latest_user_event_tick, CONN_IDLE_ENTER_DEEP_TIME * 1000000) )
        {

            bls_ll_terminateConnection(HCI_ERR_REMOTE_USER_TERM_CONN); //push terminate cmd into ble TX buffer
            bls_ll_setAdvEnable(0);   //disable adv
            sendTerminate_before_enterDeep = 1;
        }
    }
}
```

Figure4-3 Reference code of 8267 remote low power management

The “blt_pm_proc()” function is included in UI entry of main_loop.

Note: If UI entry needs to process multiple tasks, the “blt_pm_proc()” should be close to the “blt_sdk_main_loop”, since its setting depends on processing result of other tasks in UI entry.

Conclusions about low power management are shown as below:

1) If it’s needed to disable suspend for task such as audio (ui_mic_enable) or IR, the “SuspendMask” should be set as “SUSPEND_DISABLE”.

2) In Advertising state, if Slave continuous adv time reaches 60s, it should be

configured to enter deepsleep in current main_loop, and wakeup source should be set as "GPIO PAD" (enable key press wakeup in advance). Software timer is used to check whether adv time exceeds 60s, and the variable "advertise_begin_tick" serves to record the system tick when adv starts.

Slave is configured to enter deepsleep after 60s of no advertising, so as to save power and avoid Slave from advertising when Master fails to respond. Actually user needs to evaluate power consumption and then determine how to process time for Adv state.

3) In Conn state, if Slave has no audio task or LED task, and all keys are released, Slave is configured to enter deepsleep in current main_loop when it exceeds 60s away from the latest valid task, and wakeup source is set as "GPIO PAD" (enable key press wakeup in advance). It will be recorded in the retention register DEEP_ANA_REG0 it's the Conn state from which MCU enters current deepsleep. After wakeup, Slave can configure fast adv packet to establish connection with Master as soon as possible.

   Slave is configured to enter deepsleep after 60s of no valid task, so as to save power. Actually MCU can be configured not to enter deepsleep, as long as its power consumption is very low to maintain connection. User needs to determine the implementation considering actual requirement and power consumption.

   When MCU enters deepsleep from Conn state, first Slave should invoke the "bls_ll_terminateConnection" to send a "TERMINATE" command to Master, and enter deepsleep after this command is acked or the "BLT_EV_FLAG_TERMINATE" is triggered by timeout.

4) User needs to manually set latency as 0, if long time sleep (long suspend duration) is not allowed for task processing, such as key_not_released, DEVICE_LED_BUSY (LONG_PRESS_KEY_POWER_OPTIMIZE is 0).

5) Based on step 4), after latency is disabled manually, MCU will wake up in each conn_interval, thus power consumption is increased; since it's not needed to detect key press and process LED task in every conn_interval, user can manually set latency as other value and further optimize power consumption.

   When the" LONG_PRESS_KEY_POWER_OPTIMIZE" is 1, after key press is stabilized (key_matrix_same_as_last_cnt > 5), user can set latency value manually. If it's configured as "bls_pm_setManualLatency (4)", suspend will last for 5 conn_intervals. When conn_interval is 10 ms, MCU will wake up for every 50 ms (10*(4+1) = 50ms) to process LED task and detect key press. Actually user needs to consider the conn_interval value and task response time, and optimize power consumption without influencing function correspondingly.

4.8 Timer wakeup of APP layer

In Advertising state or Conn state Slave role, once MCU enters suspend, it can be woke up by stack only in specific moment, and user can hardly wake up MCU in advance. To add flexibility of PM, a timer wakeup API in APP layer and corresponding callback function are supplied in SDK. Following is the timer wakeup API in APP layer:

```
void  bls_pm_setAppWakeupLowPower(u32 wakeup_tick, u8 enable);
```

“wakeup_tick” indicates absolute system tick value for timer wakeup.

“enable”: 1-enable this wakeup function; 0-disable this wakeup function.

When timer wakeup in APP layer is triggered, the callback function registerd by “bls_pm_registerAppWakeupLowPowerCb” is executed.

```
typedef   void (*pm_appWakeupLowPower_callback_t)(int);
void      bls_pm_registerAppWakeupLowPowerCb(
                                   pm_appWakeupLowPower_callback_t cb);
```

When “bls_pm_setAppWakeupLowPower” is used to set app wakeup_tick for timer wakeup in APP layer, before SDK bottom enters suspend, it will check whether this app wakeup tick is within current suspend time. If yes, suspend will be triggered to wake up in advance at app wakeup_tick (as shown in Figure4-4). If not, this wakeup_tick is negligible to bottom layer, and wakeup time depends on BLE timing sequence.

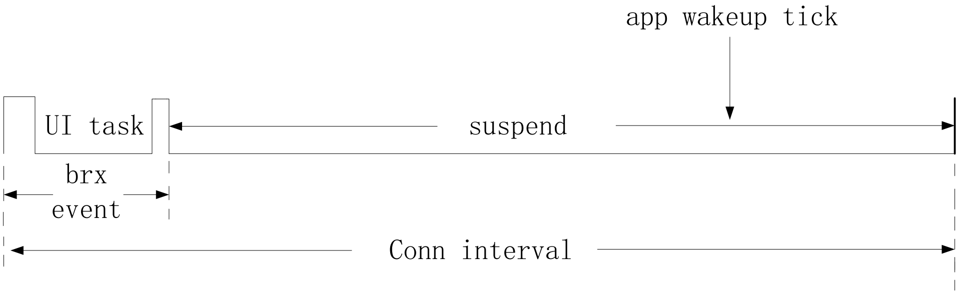


Figure4-4 Trigger app wakup tick in advance

5 Audio Processing

In SDK, only 8267 and 8269 support audio processing function.

5.1 Audio initialization

Figure5-1 shows hardware connection about audio MIC.

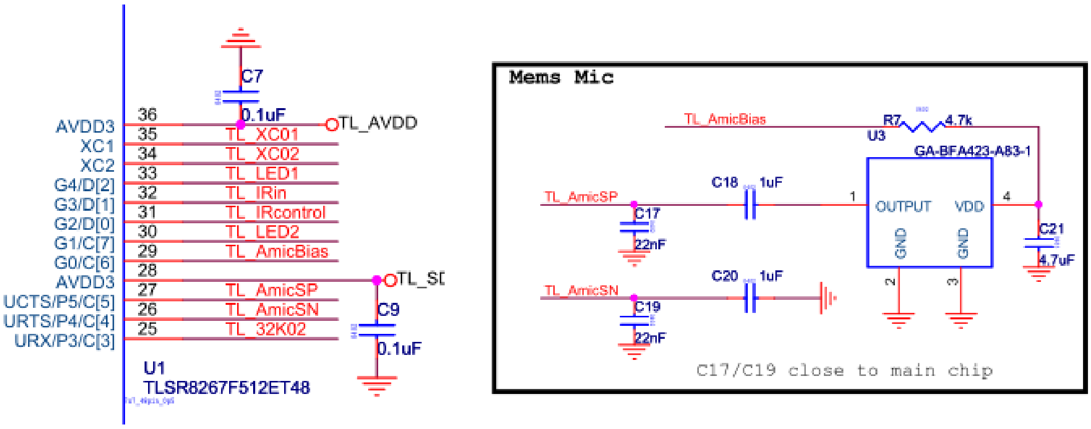


Figure5-1 Audio circuit

Note: The audio circuit shown in Figure5-1 is just supplied for reference. Please refer to actual schematic.

Software Initialization:

config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE); // configure audio buffer

audio_amic_init(DIFF_MODE, 26, 9, R2, CLOCK_SYS_TYPE); // configure MIC module sampling rate as 16K

audio_finetune_sample_rate(2);

audio_amic_input_set(PGA_CH); //audio input set, ignore the input parameter

The API below serves to configure hardware MIC:

```
void audio_amic_init( enum audio_mode_t mode_flag,
                        unsigned short misc_sys_tick,
                        unsigned short left_sys_tick,
                        enum audio_deci_t d_samp,
                        unsigned char fhs_source);
```

“audio_amic_init” is used to configure parameters of ADC module, decimation filter, LPF (low pass filter), HPF (high pass filter) and ALC (volume control). For function of each module, please refer to corresponding Datasheet. Parameters in various cases are configured in SDK, and user can select as needed. Except for a few interface parameters, most parameters are assembled in functions (User does not need to modify this tested part).

Hardware block diagram of MIC configuration is shown as below.

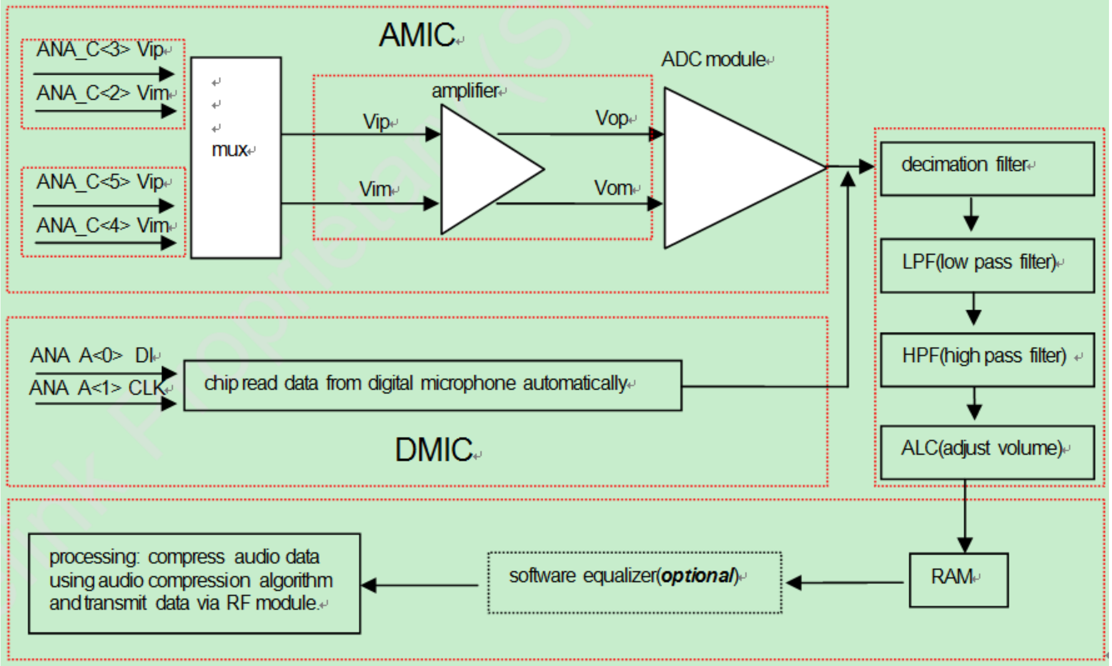


Figure5-2 MIC configuration HW block diagram

1) mode_flag

ADC sampling mode includes differential mode and single-end. Please configure according to descripton of AMIC sensor.

2) misc_sys_tick, left_sys_tick

They are used to configure output rate of ADC module combining the parameter of "audio_finetune_sample_rate(unsigned char **fine_tune**)".

$$\text{adc\_out\_rate} = \frac{\text{system clock}}{\textbf{misc\_sys\_tick}*4+\textbf{left\_sys\_tick}*16+\textbf{fine\_tune}}$$

3) d_samp

It's used to configure ratio of decimation filter, so that adc_out_rate can be down-sampled to the needed rate. In SDK, there are two optional rate including 16K and 32K, which can meet most requirements. If user has other requirements of rate, please refer to the formula above for configuration, or contact Telink for support.

4) fhs_source

It's used to select clock source of FHS (see Datasheet). Generally it's not needed to modify it.

```
void audio_amic_input_set(enum audio_input_t adc_ch);
```

This function serves to configure amplifier and select AMIC input pin. User can directly set amplifier as the supplied value, and select AMIC input pin as ANA_C<3>/ANA_C<2> or ANA_C<5>/ANA_C<4> (default) as needed. To select

ANA_C<3>/ANA_C<2>, it's needed to set the parameter of "setChannel(0)" in pgaInit function as 0.

"audio_amic_input_set()" also configures whether to bypass HPF, LPF and ALC. Current setting is bypass LPF, enable HPF and ALC.

When audio starts, PC6 AMIC BIAS needs to output high level to drive audio. After audio ends, it's needed to disable AMIC BIAS to avoid current leakage during suspend.

```
void      ui_enable_mic (u8 en)

{
    ui_mic_enable = en;
    gpio_set_output_en (GPIO_PC6, en);     //AMIC Bias output

    gpio_write (GPIO_PC6, en);

    // ...

}
```

Audio task should be executed in UI entry of mainloop.

```
#if (BLE_AUDIO_ENABLE)
    if(ui_mic_enable){  //audio
        task_audio();
    }

#endif
```

5.2 Processing of MIC sampled audio data

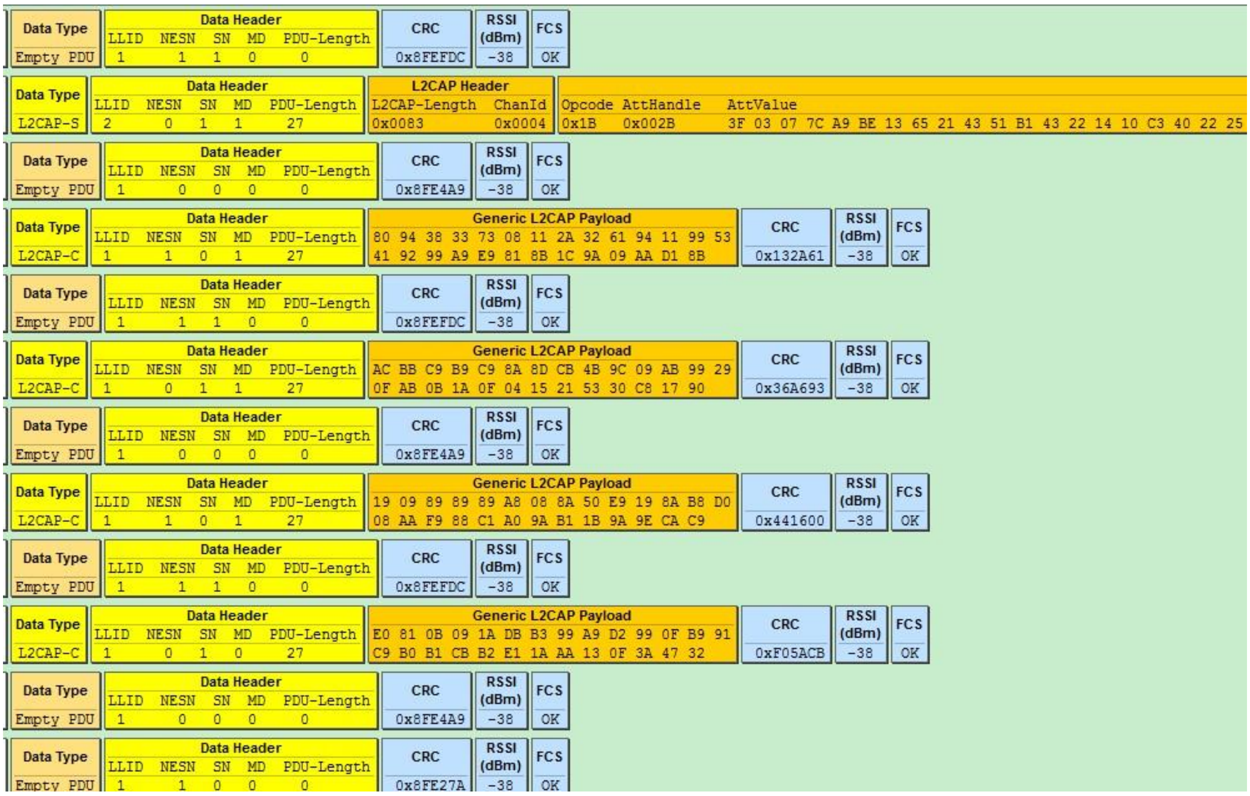
5.2.1 Audio data compression and RF transfer

The raw data sampled by hardware MIC adopt pcm format. The pcm-to-adpcm algorithm can be used to compress the raw data into adpcm format with compression ratio of 25%, thus BLE RF data volume will be decreased largely. Master will decompress the received adpcm-format audio data back to pcm format.

Sampling rate of 8267 hardware MIC is 16K*16bit, so 16K samples of raw data are generated per second, i.e. 16 samples per millisecond (16*16bit=32byte per ms).

For every 15.5ms, 496-byte (15.5*16=248 samples) raw data are generated. Via pcm-to-adpcm transformation with compression ratio of 1/4, the 496-byte data are compressed into 124 bytes. The 128-byte data including 4-byte header and 124-byte compressed data will be disassembled into five packets and sent to Master in L2CAP layer; since the maximum length of each packet is 27 bytes, the first packet must contain 7-byte l2cap information, including: 2-byte l2caplen, 2-byte chanid, 1-byte opcode and 2-byte AttHandle.

Figure5-3 shows the audio data captured by sniffer. The first packet contains 7-byte extra information and 20-byte audio data, followed by four packets with 27-byte audio data each. As a result, total audio data length is 20 + 27*4 = 128 bytes.



```
Data Type  | Data Header                      | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       |          | (dBm) |
Empty PDU  | 1    1    1  0  0                | 0x8FEFDC | -38   | OK

Data Type  | Data Header                      | L2CAP Header          |
           | LLID NESN SN MD PDU-Length       | L2CAP-Length ChanId   | Opcode AttHandle   AttValue
L2CAP-S    | 2    0    1  1  27               | 0x0083       0x0004   | 0x1B   0x002B      3F 03 07 7C A9 BE 13 65 21 43 51 B1 43 22 14 10 C3 40 22 25

Data Type  | Data Header                      | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       |          | (dBm) |
Empty PDU  | 1    0    0  0  0                | 0x8FE4A9 | -38   | OK

Data Type  | Data Header                      | Generic L2CAP Payload                          | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       | 80 94 38 33 73 08 11 2A 32 61 94 11 99 53      |          | (dBm) |
L2CAP-C    | 1    1    0  1  27               | 41 92 99 A9 E9 81 8B 1C 9A 09 AA D1 8B         | 0x132A61 | -38   | OK

Data Type  | Data Header                      | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       |          | (dBm) |
Empty PDU  | 1    1    1  0  0                | 0x8FEFDC | -38   | OK

Data Type  | Data Header                      | Generic L2CAP Payload                          | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       | AC BB C9 B9 C9 8A 8D CB 4B 9C 09 AB 99 29      |          | (dBm) |
L2CAP-C    | 1    0    1  1  27               | 0F AB 0B 1A 0F 04 15 21 53 30 C8 17 90         | 0x36A693 | -38   | OK

Data Type  | Data Header                      | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       |          | (dBm) |
Empty PDU  | 1    0    0  0  0                | 0x8FE4A9 | -38   | OK

Data Type  | Data Header                      | Generic L2CAP Payload                          | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       | 19 09 89 89 89 A8 08 8A 50 E9 19 8A B8 D0      |          | (dBm) |
L2CAP-C    | 1    1    0  1  27               | 08 AA F9 88 C1 A0 9A B1 1B 9A 9E CA C9         | 0x441600 | -38   | OK

Data Type  | Data Header                      | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       |          | (dBm) |
Empty PDU  | 1    1    1  0  0                | 0x8FEFDC | -38   | OK

Data Type  | Data Header                      | Generic L2CAP Payload                          | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       | E0 81 0B 09 1A DB B3 99 A9 D2 99 0F B9 91      |          | (dBm) |
L2CAP-C    | 1    0    1  0  27               | C9 B0 B1 CB B2 E1 1A AA 13 0F 3A 47 32         | 0xF05ACB | -38   | OK

Data Type  | Data Header                      | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       |          | (dBm) |
Empty PDU  | 1    0    0  0  0                | 0x8FE4A9 | -38   | OK

Data Type  | Data Header                      | CRC      | RSSI  | FCS
           | LLID NESN SN MD PDU-Length       |          | (dBm) |
Empty PDU  | 1    1    0  0  0                | 0x8FE27A | -38   | OK
```

Figure5-3 Audio data sample

According to “Exchange MTU size” in ATT & GATT (section 3.4) of BLE Module, since 128-byte long audio data packet are disassembled on Slave side, if peer device is expected to re-assemble these received packets, “Exchange MTU size” should be used to determine maximum ClientRxMTU of peer device. Only when “ClientRxMTU” is 128 or above, can the 128-byte long packet of Slave be correctly processed by peer device.

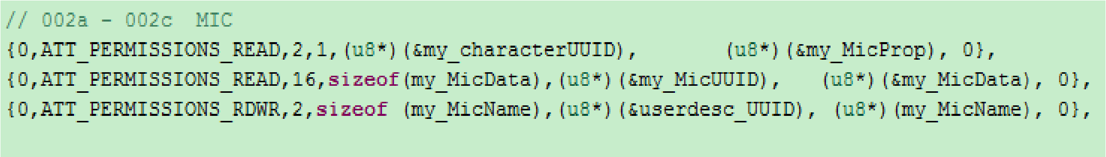
In 826x remote demo, if audio is started, to send 128-byte long packe, “blc_att_requestMtuSizeExchange” will be invoked to Exchange MTU size.

```
void voice_press_proc(void)
{
    key_voice_press = 0;
    ui_enable_mic (1);

   if(ui_mtu_size_exchange_req &&
       blc_ll_getCurrentState() == BLS_LINK_STATE_CONN){

        ui_mtu_size_exchange_req = 0;
```

```
            blc_att_requestMtuSizeExchange(BLS_CONN_HANDLE, 0x009e);
        }

    }
```

Standard method is shown as below: It's also needed to invoke "blc_att_registerMtuSizeExchangeCb" to register the callback function of MTU size Exchange, check in the callback whether "ClientRxMTU" of peer device exceeds or equals 128. Generally ClientRxMTU of Master device is larger than 128, "826x remote" does not check actual ClientRxMTU via callback.

Following is the audio MIC service in Attribute Table:



```
// 002a - 002c  MIC
{0,ATT_PERMISSIONS_READ,2,1,(u8*)(&my_characterUUID),       (u8*)(&my_MicProp), 0},
{0,ATT_PERMISSIONS_READ,16,sizeof(my_MicData),(u8*)(&my_MicUUID),    (u8*)(&my_MicData), 0},
{0,ATT_PERMISSIONS_RDWR,2,sizeof (my_MicName),(u8*)(&userdesc_UUID),  (u8*)(my_MicName), 0},
```

Figure5-4 MIC service in Attribute Table

The Attribute in the middle is used to transfer audio data. Currently its AttHandle value in the Attribute Table is 43 (0x2B) and may be updated in following versions. Data are sent to Master via "Handle Value Notification"; if it's the notification corresponding to the AttHandle of 0x2B, the Attribute Value will be assembled into 128 bytes, and transferred to pre-configured buffer. Then the data are decompressed back to the pcm-format audio data.

Both packet disassembly on Slave and assembly on Master follow BLE stack standard.

5.2.2 Audio data compression processing

Related macros are defined in "app_config.h", as shown below:

```
#define     ADPCM_PACKET_LEN                128

#define     TL_MIC_ADPCM_UNIT_SIZE          248

#define     TL_SDM_BUFFER_SIZE              992

#define     TL_MIC_32K_FIR_16K              1


#if TL_MIC_32K_FIR_16K

    #define     TL_MIC_BUFFER_SIZE              1984

#else

    #define     TL_MIC_BUFFER_SIZE              992

#endif
```

config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE);

Each compression needs to process 248-sample, i.e. 496-byte data. Since hardware MIC continuously samples audio data and transfers the processed pcm-format data into buffer_mic, considering data buffering and preservation, this buffer should be pre-configured so that it can store result of two compressions, i.e. 496 samples.

If 16K sampling rate is directly used, then 496 samples correspond to 992 bytes, i.e. “TL_MIC_BUFFER_SIZE” should be configured as 992.

If MIC adopts 32K sampling rate and transfers the data into buffer with 16K speed after FIR processing, each sample corresponds to four bytes (during compression, the former two bytes are processed as one 16bit raw data, and the latter two bytes are discarded), and the buffer size “TL_MIC_BUFFER_SIZE” should be configured as 1984.

The following example shows the `TL_MIC_BUFFER_SIZE` is configured as 1984 when the macro `TL_MIC_32K_FIR_16K` is enabled (enable FIR).

buffer_mic is defined as below:

s16 buffer_mic[TL_MIC_BUFFER_SIZE>>1]; // Totally 496 samples, 1984 bytes

```
config_mic_buffer ((u32)buffer_mic, TL_MIC_BUFFER_SIZE);
```

Buffer is configured for hardware MIC data output, and the sampled data are transferred into memory starting from buffer_mic address with 16K speed; once the maximum length 1984 is reached, data transfer address returns to the buffer_mic address, the old data will be replaced directly without checking whether it’s read.

It’s needed to maintain a write pointer “reg_audio_wr_ptr” (hardware register value) when transferring data into RAM; the pointer is used to indicate the address in RAM for current newest audio data.

The “buffer_mic_enc” is defined to store the compressed 128-byte data; buffer number is configured as 4 to indicate result of four compressions can be buffered at most.

int buffer_mic_enc[BUFFER_PACKET_SIZE];

Since “BUFFER_PACKET_SIZE” is 128, and “int” occupies four bytes, it’s equivalent to 128*4 signed char.

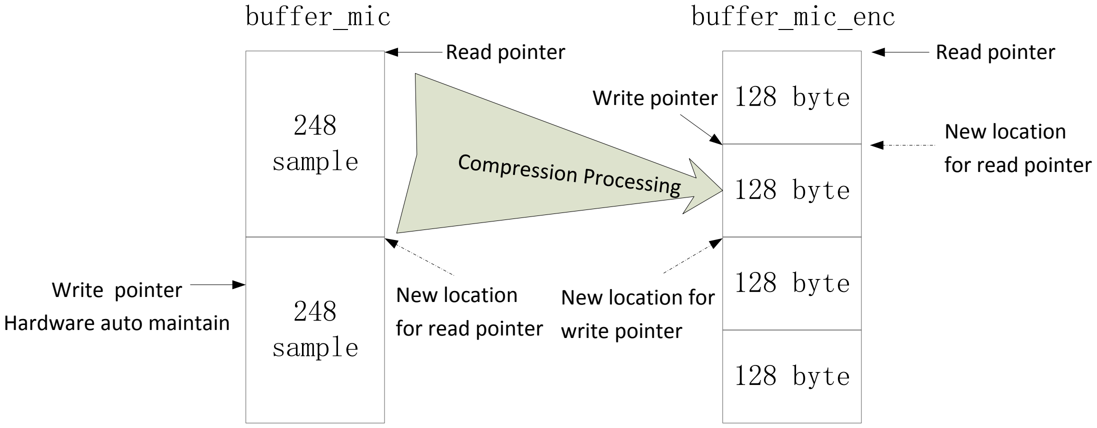


Figure5-5 Compression processing

The figure above shows the compression processing method:

The buffer_mic automatically maintains a write pointer by hardware, and maintains a read pointer by software. Whenever the write pointer is beyond 248 samples away from the read pointer via software detection, the compression processing function is invoked to read 248-sample data starting from the read pointer and compress them into 128 bytes; the read pointer moves to a new location to indicate following data are new and not read.

The buffer_mic is continuously checked whether there're enough 248-sample data; if so, the data are compressed and transferred into the buffer_mic_enc. Since 248-sample data are generated for every 15.5ms, the firmware must check the buffer_mic with maximum frequency of 1/15.5ms. The firmware only executes the task_audio once during each main_loop, so the main_loop duration must be less than 15.5ms to avoid audio data loss. In Conn state, the main_loop duration equals connection interval; so for applications with audio task, connection interval must be less than 15.5ms. It's recommended to configure connection interval as 10ms or 7.5ms; in current SDK, 7.5ms connection interval is used.

The buffer_mic_enc maintains the write pointer and read pointer by software: after the 248-sample data are compressed into 128 bytes, the compressed data are copied into the buffer address starting from the write pointer, and the buffer_mic_enc is checked whether there's overflow; if so, the oldest 128-byte data are discarded and the read pointer switches to the next 128 bytes.

The compressed data are copied into the BLE RF data Tx buffer as below: The buffer_mic_enc is checked if it's non-empty (when writer pointer equals read pointer, it indicates "empty", otherwise it indicates "non-empty); if the buffer is non-empty, the 128-byte data starting from the read pointer are copied into the BLE RF data Tx buffer (if security is enabled, encryption step is also needed), then the read pointer moves to the new location.

Following is the sample code for audio data compression processing:

```
void   proc_mic_encoder (void)

{

    staticu16  buffer_mic_rptr;

    u16 mic_wptr = reg_audio_wr_ptr;

    u16  l  =  ((mic_wptr<<1)  >=  buffer_mic_rptr)  ?  ((mic_wptr<<1)  -
buffer_mic_rptr) : 0xffff;


    if (l >=(TL_MIC_BUFFER_SIZE>>2)) {

        log_task_begin (TR_T_adpcm);

        s16 *ps = buffer_mic + buffer_mic_rptr;

        mic_to_adpcm_split ( ps, TL_MIC_ADPCM_UNIT_SIZE,

                        (s16 *)(buffer_mic_enc + (ADPCM_PACKET_LEN>>2) *

                        (buffer_mic_pkt_wptr & (TL_MIC_PACKET_BUFFER_NUM
- 1))), 1);


       buffer_mic_rptr = buffer_mic_rptr ? 0 : (TL_MIC_BUFFER_SIZE>>2);

        buffer_mic_pkt_wptr++;

        int  pkts  =  (buffer_mic_pkt_wptr  -  buffer_mic_pkt_rptr)  &
(TL_MIC_PACKET_BUFFER_NUM*2-1);

        if (pkts > TL_MIC_PACKET_BUFFER_NUM) {

            buffer_mic_pkt_rptr++;

            log_event (TR_T_adpcm_enc_overflow);

        }

        log_task_end (TR_T_adpcm);

    }

}
```

5.3 Compression and decompression algorithm

The following function is used to invoke the compression algorithm:

void mic_to_adpcm_split (signed short *ps, int len, signed short *pds, int start);

Notes:

- ✧ "ps" points to the starting storage address for data before compression, which corresponds to the read pointer location of the buffer_mic as shown in Figure5-5;
- ✧ "len" is configured as TL_MIC_ADPCM_UNIT_SIZE (248), which indicates 248 samples;
- ✧ "pds" points to the starting storage address for compressed data, which corresponds to the write pointer location of the buffer_mic_enc as shown in Figure5-5.

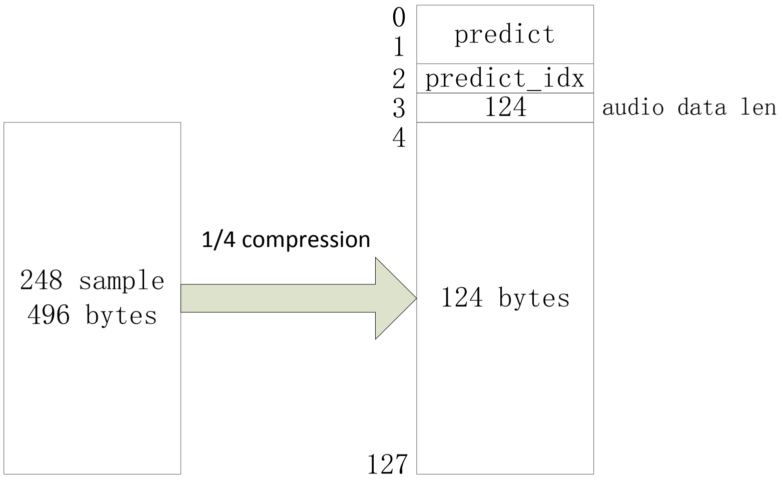


Figure5-6 Data corresponding to compression algorithm

The memory space for compressed data stores 2-byte predict, 1-byte predict_idx, 1-byte length of current valid adpcm-format audio data (i.e. 124), and followed by 124-byte data compressed from the 496-byte raw data with compression ratio of 1/4.

The following function is used to invoke the decompression algorithm:

void adpcm_to_pcm (signed short *ps, signed short *pd, int len);

Notes:

- ✧ "ps" points to the starting storage address for data to be decompressed (i.e. 128-byte adpcm-format data). This address needs user to define a buffer to store 128-byte data copied from BLE RF.
- ✧ "pd" points to the starting storage address for decompressed 496-byte pcm-format audio data. This address needs user to define a buffer to store data to be transferred when playing audio.
- ✧ "len" is 248, same as the "len" during compression.

As shown in Figure5-6, during decompression, the data read from the buffer are two-byte predict, 1-byte predict_idx, 1-byte valid audio data length "124", and the 124-byte adpcm-format data which will be decompressed into 496-byte pcm-format audio data.

6 OTA

Since 8267 and 8269 support 512K flash, as well as flash multi-address booting from multiples of 128KB offsets (e.g. boot from 0, 0x20000 or other address), they have the same flash architecture and OTA procedure.

Since 8261/8266 can only boot from address 0 without flash multi-address booting support, they have the same OTA procedure. Flash size for 8261 and 8266 is 128K and 512K, respectively; they have similar flash architecture.

To implement OTA for 826x Slave, a device is needed to act as BLE OTA Master, which can be the Bluetooth device (supporting OTA in APP) combined with Slave, or simply Telink BLE Master Dongle.

In this section, Telink kma dongle is taken as an example of OTA Master to illustrate how 826x BLE OTA is realized.

6.1 8267/8269 Flash architecture and OTA procedure

6.1.1 8267/8269 FLASH storage architecture

In SDK, by default firmware size should not exceed 128K, i.e. the flash area 0~0x20000 serves to store firmware.

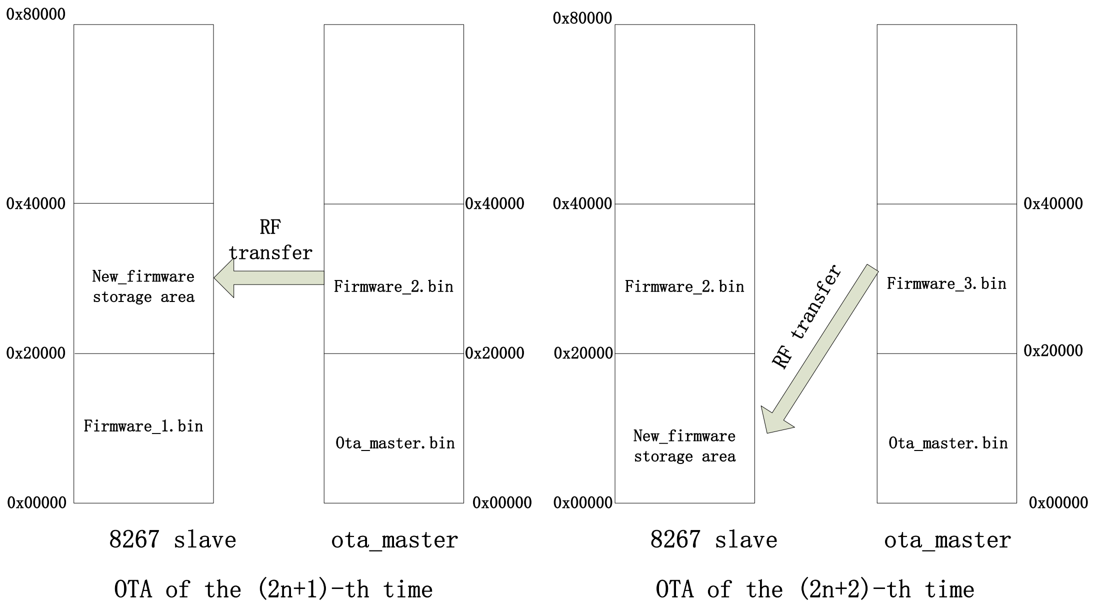


Figure6-1 8267/8269 default Flash storage structure

1) OTA Master burns new firmware2 into the Master flash area starting from 0x20000.

2) OTA for the first time:

A. When power on, Slave starts booting and executing firmware1 from flash 0~0x20000.

B. When firmware1 is running, the area of Slave flash starting from 0x20000 (i.e. flash 0x20000~0x40000) is cleared during initialization and will be used as storage area for new firmware.

C. OTA process starts, Master transfers firmware2 into Slave flash area starting from 0x20000 via RF. Then Slave sets bootloader to boot from the new firmware offset and reboots (similar to power cycle).

3) For subsequent OTA updates, OTA Master first burns new firmware3 into the Master flash area starting from 0x20000.

4) OTA for the second time:

A. When power on, Slave starts booting and executing firmware2 from flash 0x20000~0x40000.

B. When firmware2 is running, the area of Slave flash starting from 0x0 (i.e. flash 0~0x20000) is cleared during initialization and will be used as storage area for new firmware.

C. OTA process starts, Master transfers firmware3 into Slave flash area starting from 0x0 via RF. Then Slave sets bootloader to boot from the new firmware offset and reboots.

5) Subsequent OTA process repeats steps 1)~4): 1)~2) represents OTA of the (2n+1)-th time, while 3)~4) represents OTA of the (2n+2)-th time.

6.1.2 8267/8269 OTA update procedure

Based on the flash storage structure introduced in Section 6.1.1, the OTA update procedure is illustrated as below:

8267/8269 multi-address booting mechanism: OTA only uses two addresses booting (boot from 0 or 0x20000). After MCU is powered on, Slave boots from flash address 0 by default. First flash address 0x8 is read, if its value is 0x4b, the code starting from 0 are transferred to RAM, and the following instruction fetch address equals 0 plus PC pointer value; if the value of flash 0x8 is not 0x4b, MCU directly reads flash address 0x20008, if its value is 0x4b, the code starting from 0x20000 are transferred to RAM, and the following instruction fetch address equals 0x20000 plus PC pointer value. By modifying flag bit value of flash 0x8 and 0x20008, the part of flash code to be executed will be determined.

In 8267/8269 SDK with OTA function support, the OTA upgrade process of the (2n+1)-th or (2n+2)-th time is shown as below:

1) After MCU is powered on, read flash address 0x8 and 0x20008, and compare the value with 0x4b to determine the booting address; then Slave boots from

corresponding address (0 or 0x20000) and starts executing the code. This function is automatically completed by MCU hardware.

2) During firmware initialization, read MCU hardware register to judge the booting address.

    a) If booting address is 0, the ota_program_offset is set as 0x20000, and the area of Slave flash starting from 0x20000 (i.e. 0x20000~0x40000) will be all erased to "0xff", which indicates the new firmware will be transferred into this area by Master during the following OTA process.

    b) If booting address is 0x20000, the ota_program_offset is set as 0x0, and the area of Slave flash starting from 0x0 (i.e. 0~0x20000) will be all erased to 0xff, which indicates the new firmware will be transferred into this area by Master during the following OTA process.

3) Slave MCU executes the firmware after booting; OTA Master is powered on and establishes BLE connection with Slave.

4) Trigger OTA Master to enter OTA mode by UI (e.g. button press, write memory by PC tool, etc.). After entering OTA mode, OTA Master needs to obtain Handle value of Slave OTA Service Data Attribute (The handle value can be pre-appointed by Slave and Master, or obtained via "read_by_type".)

5) After the Atrribute Handle value is obtained, OTA Master may need to obtain version number of current Slave Flash firmware, and compare it with the version number of local stored new firmware. This step is determined by user.

6) To enable OTA upgrade, OTA Master will send an OTA_start command to inform Slave to enter OTA mode.

7) After the OTA_start command is received, Slave enters OTA mode and waits for OTA data to be sent from Master.

8) Master reads the firmware stored in the flash area starting from 0x20000, and continuously sends OTA data to Slave until the whole firmware is sent.

9) After the OTA data are received, Slave stores the data into the area starting from ota_program_offset.

10) After the OTA data are sent, Master will check if all data are correctly received by Slave (invoke related BLE function in bottom layer to judge whether Link Layer data are all correctly acked).

11) After Master confirms all OTA data are correctly received by Slave, it will send an OTA_END command.

12) After Slave receives the OTA_END command, offset address 8 based on the new firmware starting address (i.e. ota_program_offset+8) is written with "0x4b", and offset address 8 based on the old firmware starting address is written with "0x00". This indicates Slave will execute the firmware from the new area after the next

booting.

13) Slave reboots, and the new firmware will take effect.

14) During the whole OTA upgrade process, Slave will continuously check whether there's packet error, packet loss or timeout (A timer is started when OTA starts). Once packet error, packet loss or timeout is detected, Slave will determine the OTA process fails. Then Slave reboots, and executes the old firmware.

The OTA related operations on Slave side described above have been realized in 826x BLE SDK and can be used by user directly. On Master side, extra firmware design is needed and it will be introduced later.

6.1.3 Modify Flash storage architecture

Since 8267/8269 uses booting address of 0x00000 and 0x20000 alternately, the starting address of flash area to store new firmware during an OTA upgrade is fixed, user can only modify firmware size.

By default, maximum firmware size is 128KB, and the flash space 0x00000 ~ 0x40000 can only be used to store firmware. If firmware does not need such a large storage area, e.g. maximum firmware size does not exceed 30KB, only part of the two 128KB space (0x00000 ~ 0x20000, 0x20000 ~ 0x40000) are used.

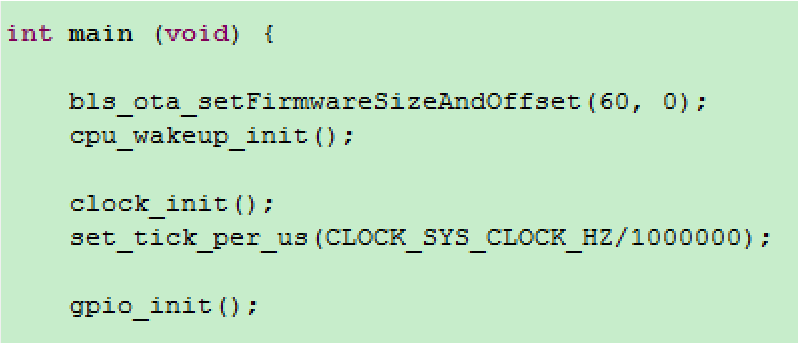
To use the redundant space as data storage area, the API below can be invoked to configure the maximum firmware size as needed.

void bls_ota_setFirmwareSizeAndOffset(int firmware_size_k,
u32 ota_offset);

The API declaration is available in "proj_lib/ble/service/ble_ll_ota.h".

For 8267/8269, the parameter "ota_offset" in this API is invalid.

"firmware_size_k": This parameter indicates the maximum firmware size in KB, and it must be configured as 4KB aligned. For example, suppose the maximum firmware size is 57KB, this parameter should be configured as "60". This API can only be invoked before the cpu_wakeup_init() of main.c to take effect, as shown below.



```
int main (void) {

    bls_ota_setFirmwareSizeAndOffset(60, 0);
    cpu_wakeup_init();

    clock_init();
    set_tick_per_us(CLOCK_SYS_CLOCK_HZ/1000000);

    gpio_init();
```

By configuration above, the two 60K flash areas 0x00000 ~ 0x0F000 and 0x20000 ~ 0x2F000 can be used as firmware storage space, while the two 68K flash areas 0x0F000 ~ 0x20000 and 0x2F000 ~ 0x40000 can be used as user data storage space.

6.2 8266 Flash architecture and OTA procedure

6.2.1 8266 FLASH storage architecture

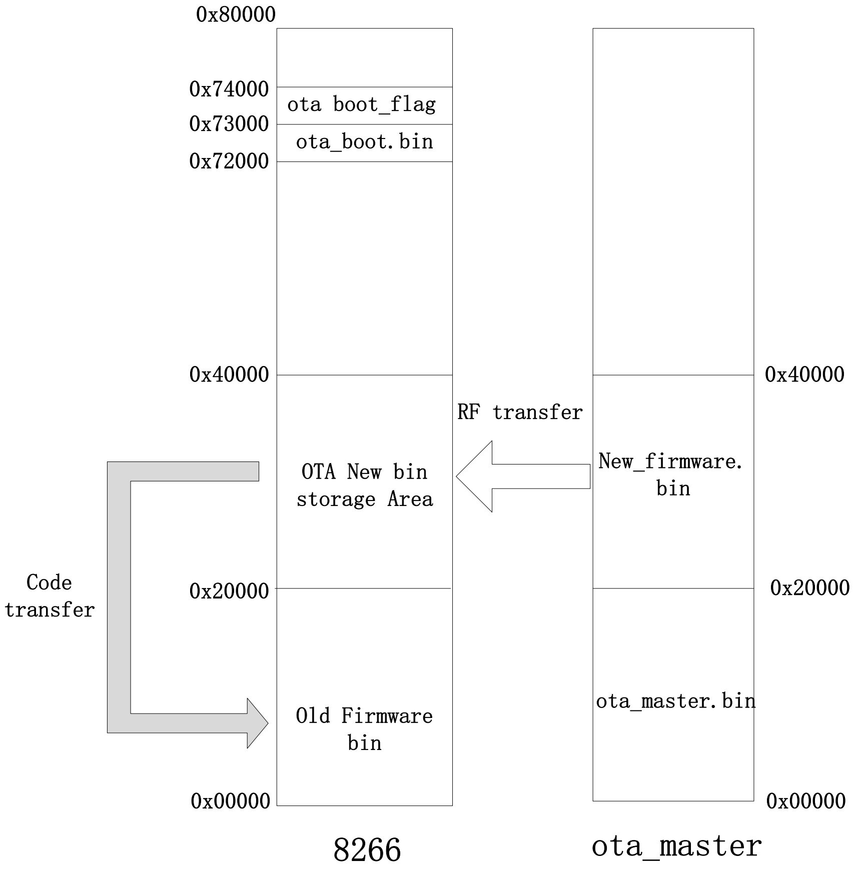


Figure6-2 8266 default Flash storage structure

In SDK, by default firmware size should not exceed 128K, i.e. the area 0~0x20000 in flash serves to store firmware.

On Slave side, current old firmware is stored in the flash starting from 0 (128K area 0~0x20000), ota_boot.bin is stored in the flash starting from 0x72000 (1.5K area 0x72000~0x72600); flash starting from 0x20000 (the 128K area 0x20000~0x40000) is used to store the New_firmware.bin obtained from OTA_Master via RF transfer, 1-byte boot_flag in address 0x73000 is used as check flag during booting.

On OTA_Master side, ota_master.bin is stored in the flash starting from 0 (128K area 0~0x20000), while the new firmware of Slave is stored in the flash starting from 0x20000 (128K area 0x20000~0x40000).

6.2.2 8266 OTA upgrade procedure

Based on the flash storage structure introduced in Section 6.2.1, the OTA update procedure is illustrated as below:

1) Power on Slave normally

   After Slave flash is burned with Old_firmware.bin and ota_boot.bin, the boot_flag in flash 0x73000 is 0xff.

   After power on, Slave MCU transfers beginning instructions of Old_firmware.bin starting from flash 0x00000 into SRAM address starting from 0x808000, and executes starting code corresponding to cstartup_8266.S. The starting code checks boot_flag value; since the value is not 0xa5, it serves as normal Slave function and executes c code of “Old_firmware.bin”.

2) Power on OTA_Master normally

   Power on sequence for OTA_Master and Slave does not matter. After Master and Slave are powered on and booted normally, BLE connection is established to ensure normal communication.

3) Trigger OTA mode

   Trigger OTA_Master to enter OTA mode (by button press or memory writting via PC tool). Send “OTA start” command by Master to make Slave enter OTA mode.

4) Transfer New_firmware.bin from OTA_Master to Slave via RF.

   After OTA_Master and Slave enter OTA mode, OTA_Master sends OTA packets with New_firmware data to Slave via RF. Slave will burn the analyzed data into its flash area starting from 0x20000 (new_firmware storage area 0x20000~0x40000).

5) After all OTA data are sent, OTA_Master sends “OTA End” command, and Slave is rebooted.

   After OTA process is finished successfully, New_firmware.bin is already stored in Slave flash starting from 0x20000. Slave will set the boot_flag value in flash 0x73000 as “0xa5”, then reboot MCU.

6) Slave executes ota_boot.bin.

   After Slave is rebooted, MCU transfers beginning instructions of Old_firmware.bin starting from flash 0x00000 into SRAM starting from 0x808000, and executes the starting code corresponding to cstartup_8266.S in Old_firmware.bin. The starting code checks boot_flag value in flash 0x73000; since the value is “0xa5”, Slave does not execute code of Old_firmware.bin, but transfers ota_boot.bin from 1.5K flash area 0x72000~0x72600 into SRAM starting from 0x808000. After data transfer is finished, reset MCU so that MCU will execute code starting from SRAM 0x808000 (equivalent to executing ota_boot.bin) rather than re-transfer code from flash into SRAM.

7) ota_boot updates code, and Slave is rebooted.

   After the ota_boot.bin is executed, contents of New_firmware.bin will be read page by page starting from flash 0x20000 and written into flash starting from 0x00000. It's equivalent to updating New_firmware.bin to flash starting from 0. Slave will set boot_flag value in flash 0x73000 as "0x00", then reboot MCU.

8) New_firmware.bin is executed normally.

   After Slave MCU is rebooted, it transfers code starting from flash 0 into SRAM starting from 0x808000, and checks the boot_flag. Since the value is not "0xa5", it serves as normal Slave function.

   Similar to Old_firmware.bin, the New_firmware.bin supports OTA function, and user can restart OTA mode to upgrade firmware (the new code should be burned into OTA_Master flash starting from 0x20000 before OTA process).

6.2.3 cstartup_8266.S, reset, reboot, code transfer

6.2.3.1 boot_flag detect and process by cstartup_8266.S

The ota_master.bin is executed on OTA_Master side and its starting file has no special requirement.

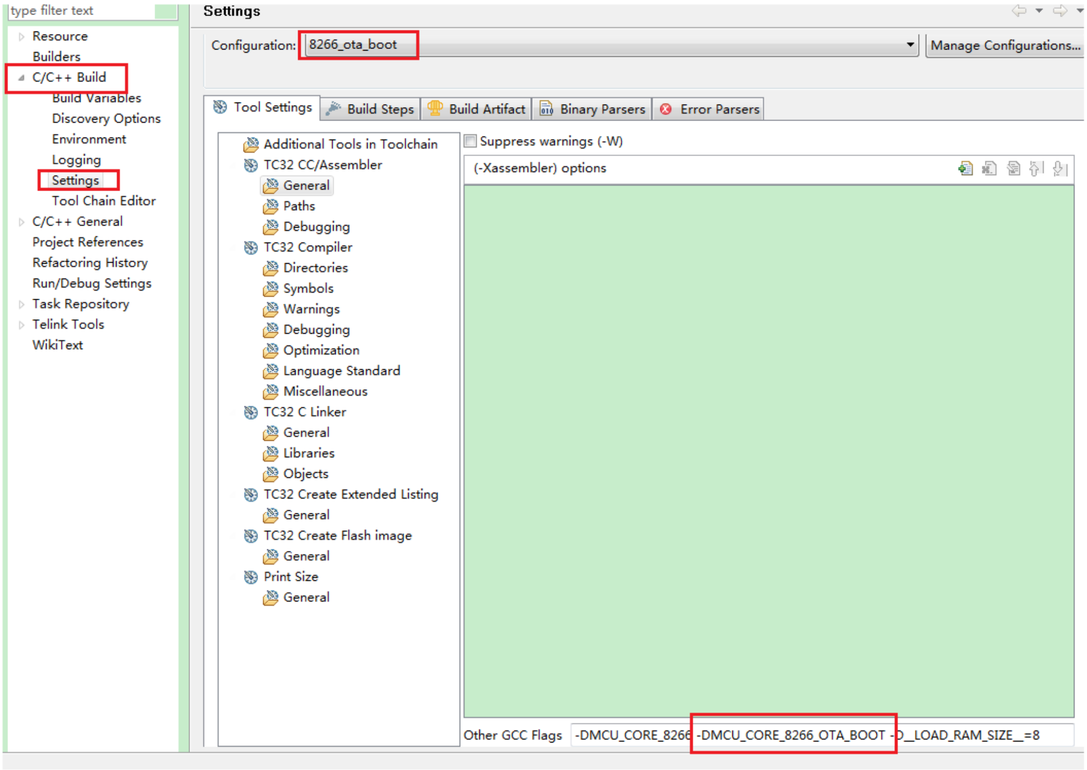
Old_firmware.bin, New_firmware.bin and ota_boot.bin are all executed on Slave side; they must have the same cstartup_8266.S in corresponding project, and consistent locations of iCache and iTag. Since New_firmware.bin is the update firmware to replace Old_firmware.bin, they have the same starting code of course. Old_firmware.bin differs from ota_boot.bin in starting code as shown below:

1) Old_firmware sets iCache and iTag; when ota_boot is executed, MCU is still powered on, and configuration of Old_firmware can be used directly.

2) ota_boot does not check boot_flag value; while Old_firmware will check boot_flag and process accordingly: if it's not 0xa5, the Old_firmware is normally executed; if it's 0xa5, ota_boot.bin from flash area 0x72000~0x72600 will be copied into SRAM starting from 0x808000, and MCU is reset.

According to the two points above, configurations in cstartup_8266.S from SETIC to COPY_DATA should be modified as below.

First define "MCU_CORE_8266_OTA_BOOT" in the bottom layer of ota_boot compile option (It can't be defined in other projects).



Configuration: 8266_ota_boot

Other GCC Flags: -DMCU_CORE_8266 -DMCU_CORE_8266_OTA_BOOT -D_LOAD_RAM_SIZE__=8

Figure6-3 8266_ota_boot project setting

```
SETIC:
    tloadr       r1, DAT0 + 24
    tloadr      r0, DAT0 + 36              @ IC tag start
#ifdefMCU_CORE_8266_OTA_BOOT
    tloadr      r0, DAT0 + 36 @ The three sentences are only for aligning.
    tloadr      r0, DAT0 + 36
    tloadr      r0, DAT0 + 36
#else
    tstorerb  r0, [r1, #0]
    tadd      r0, #1                  @ IC tag end
    tstorerb  r0, [r1, #1]
#endif
    @tmov     r0, #0;
    @tstorerb r0, [r1, #2]
```

```
COPY_CODE_INIT:
    tmov     r3, #115        @ OTA FW ready flag at 0x73000;
      (0x73000 = #115<< 12)
    tshftl      r3, r3, #12     @ 0x73<<12 = 0x73000
    tloadr      r3, [r3, #0]    @ read value of flash 0x73000
into R3
#ifdefMCU_CORE_8266_OTA_BOOT
tcmp      r3, #0 @ when ota boot starts, compare R3 with 0 (only
for aligning)
#else
    tcmp     r3, #165 @ when firmware starts, compare R3 with
0xa5
#endif
    tjne     COPY_DATA_INIT @ if not equal, directly jump to
COPY_DATA_INIT,start executing firmware noramlly; if equeal,
execute the following sentences, transfer the contents of flash
0x72000~0x72600 (i.e. ota_boot.bin) to SRAM 0x8000~0x8600.
    tmov     r2, #114@ OTA boot code at: 0x72000
    tloadr      r3, COPY_CODE_DAT
    tloadr      r0, COPY_CODE_DAT + 4
    tshftl      r2, r2, #12     @ 0x72<<12
COPY_CODE_START:
    tloadm      r2!, {r1}
    tstorem  r3!, {r1}
    tcmp     r3, r0
    tjne     COPY_CODE_START
```

After data transfer, write register “core_602” with “0x88”, reset MCU, restart to execute ota_boot.bin.

```
    tloadr      r3, COPY_CODE_DAT + 8
    tmov     r2, #136     @0x88
    tstorerb r2, [r3, #0]


COPY_CODE_END:
    tj    COPY_CODE_END
   .balign   4
COPY_CODE_DAT:
   .word  (0x808000)
   .word  (0x808600)
   .word  (0x800602)

COPY_DATA_INIT:
   tloadr    r1, DATA_I
   tloadr    r2, DATA_I+4
   tloadr    r3, DATA_I+8
COPY_DATA:
```

6.2.3.2 Firmware size

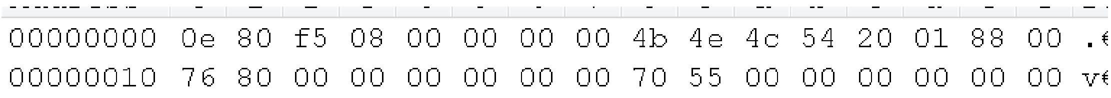
After OTA_Master enters OTA mode, New_firmware.bin is read from flash 0x20000. The size value of the bin file is stored in cstartup_8266.S 0x18~0x1b.

```
        cstartup_8266.S:
          .org 0x18
          .word  (_bin_size_)
         boot.link:
```

PROVIDE(_bin_size_ = _code_size_ + _end_data_ - _start_data_);

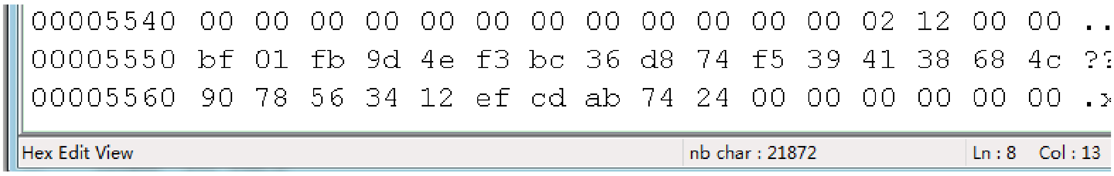
Thus 4-byte data in firmware starting from 0x18 indicates firmware size. Since New_firmware.bin starts from 0x20000, the firmware size is stored in 0x20018. When New_firmware.bin is read by OTA_Master, the firmware size is read first.

As shown in Figure6-4 and Figure6-5, the firmware size is 0x5570, and the bin file actually ranges from flash 0x0000 to 0x556f.



```
00000000 0e 80 f5 08 00 00 00 00 4b 4e 4c 54 20 01 88 00 .
00000010 76 80 00 00 00 00 00 00 70 55 00 00 00 00 00 00 v
```

Figure6-4 firmware size inforamtion



```
00005540  00 00 00 00 00 00 00 00 00 00 00 00 02 12 00 00 ..
00005550  bf 01 fb 9d 4e f3 bc 36 d8 74 f5 39 41 38 68 4c ?:
00005560  90 78 56 34 12 ef cd ab 74 24 00 00 00 00 00 00 .x
```

Figure6-5 firmware ending

6.2.3.3 Reset and reboot

1) Reset MCU: Write “core_0x602” with “0x88”. MCU starts to execute the firmware in SRAM with PC pointer starting from 0.

2) Reboot MCU: MCU enters deepsleep, and it’s woke up by timer after 5ms. MCU starts to load firmware from flash, and executes firmware with PC pointer staring from 0.

3) Slave obtains “New_firmware.bin” via RF in OTA mode, and stores the firmware in flash starting from 0x20000. Then Slave sets “boot_flag” as “0xa5”, and reboots MCU.

```
flash_erase_sector (0x73000);

u32 flag = 0xa5;

flash_write_page (0x73000, 4, &flag);   // set boot_flag 0xa5

start_reboot();


void start_reboot(void)

{

    irq_disable ();

    cpu_sleep_wakeup (1, PM_WAKEUP_TIMER, clock_time() +
                 5*CLOCK_SYS_CLOCK_1MS);

}
```

4) ota_boot transfers New_firmware.bin starting from flash 0x20000 to location starting from 0, clears boot_flag and reboots MCU.

```
    buff[0] = 0;

    flash_write_page (0x73000, 1, buff);

    REG_ADDR8(0x6f) = 0x20;// ”writing core_6f with 0x20” is reboot
equivalent to ”start_reboot()”, need to load flash.

    while(1);
```

No other modifications are needed for starting code since all necessary contents are already added to Cstartup_8266.S of 8266. 8266 ota_boot can use the compiled firmware of 8266_ota_boot branch in SDK.

6.2.4 Modify Flash storage architecture

In SDK, by default maximum firmware size should not exceed 128K, the 128K flash area 0x00000~0x20000 serves to store firmware, 0x20000~0x40000 serves to store OTA new firmware, the sector starting from 0x72000 serves to store 8266_ota_boot.bin, and 0x73000 serves to store 1-byte ota boot_flag. All the addresses and firmware size above are modifiable via corresponding API.

6.2.4.1 Modify firmware size and OTA FW storage address

The API below serves to modify maximum firmware size and the starting address to store OTA new firmware.

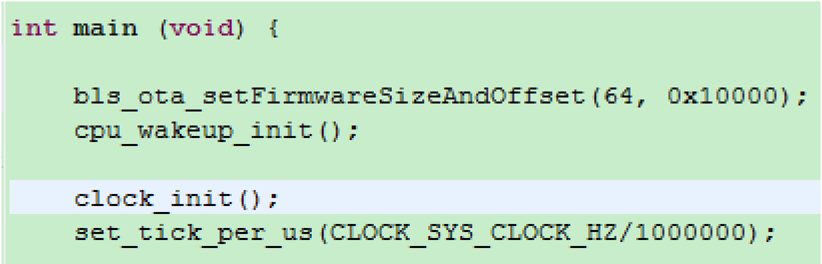
```
void bls_ota_setFirmwareSizeAndOffset(int firmware_size_k,
                                       u32 ota_offset);
```

The API declaration is available in proj_lib/ble/service/ble_ll_ota.h.

“ota_offset”: This parameter indicates the starting address to store OTA new firmware.

“firmware_size_k”: This parameter indicates the maximum firmware size in KB, and it must be configured as 4KB aligned. For example, suppose the maximum firmware size is 57KB, this parameter should be configured as “60”. This API can only be invoked before the cpu_wakeup_init() of main.c to take effect.

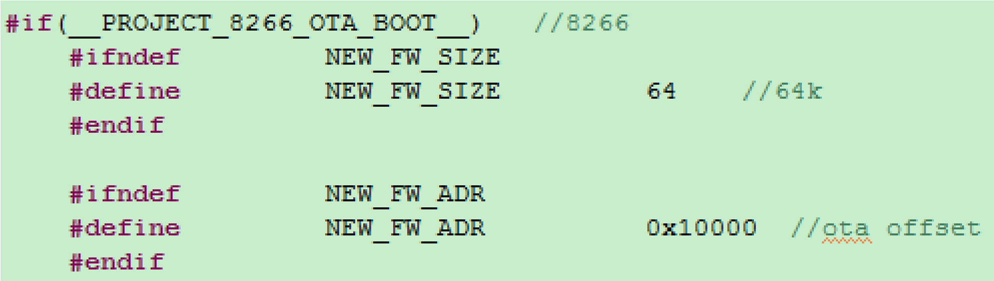
Suppose maximum firmware size is 64K, the 64K flash space 0x00000~0x10000 is used to store firmware, and the 64K space 0x10000 ~ 0x20000 is used to store OTA new firmware, the configuration is shown as below:



```
int main (void) {

    bls_ota_setFirmwareSizeAndOffset(64, 0x10000);
    cpu_wakeup_init();

    clock_init();
    set_tick_per_us(CLOCK_SYS_CLOCK_HZ/1000000);
```

By configuration above, the redundant 128K flash space 0x20000 ~ 0x40000 can be used to store user data.

Note: The value of “NEW_FW_SIZE” and “NEW_FW_ADR” in vendor/826x_ota_boot/main.c of SDK must be modified correspondingly, as shown below.

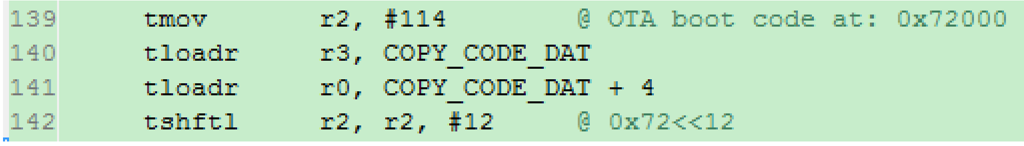


```
#if(__PROJECT_8266_OTA_BOOT__)   //8266
    #ifndef         NEW_FW_SIZE
    #define         NEW_FW_SIZE         64     //64k
    #endif

    #ifndef         NEW_FW_ADR
    #define         NEW_FW_ADR          0x10000  //ota offset
    #endif
```

6.2.4.2 Modify storage address of OTA boot bin

In SDK, 8266_ota_boot.bin is stored in the flash sector starting from 0x72000 by default, and its maximum size is 1.5K. The starting address to store 8266_ota_boot.bin is modifiable in cstartup_8266.S.

Suppose the sector starting from 0x40000 serves to store 8266_ota_boot.bin, replace “114” with “64” (64=0x40, 0x40<<12=0x40000) in the code below (proj/mcu_spec/cstartup_8266.S).



```
    tmov        r2, #114         @ OTA boot code at: 0x72000
    tloadr      r3, COPY_CODE_DAT
    tloadr      r0, COPY_CODE_DAT + 4
    tshftl      r2, r2, #12      @ 0x72<<12
```

6.2.4.3 Modify storage addrss of OTA boot flag

In SDK, 1-byte ota boot flag is stored in flash address 0x73000 by default. The address to store ota boot flag is modifiable.

Suppose the address 0x41000 serves to store ota boot flag, follow the modifications below:

1) Invoke the API below during initialization:

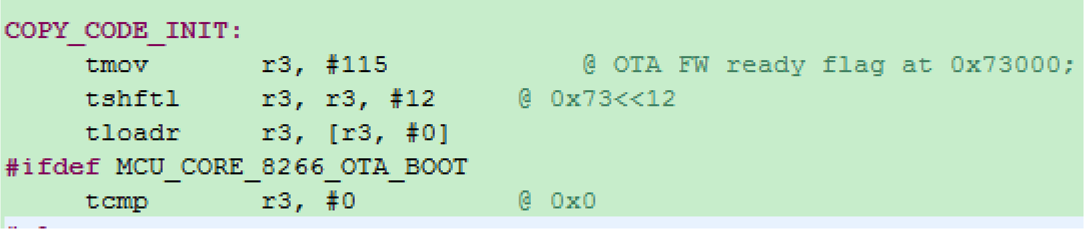
    void bls_ota_setBootFlagAddr(u32 bootFlag_addr);

   The API declaration is available in proj_lib/ble/service/ble_ll_ota.h.

    bls_ota_setBootFlagAddr(0x41000);

2) Modify proj/mcu_spec/cstartup_8266.S:

   Replace “115” with “65” (65=0x41, 0x41<<12=0x41000) in the code below.



```
COPY_CODE_INIT:
        tmov          r3, #115              @ OTA FW ready flag at 0x73000;
        tshftl        r3, r3, #12       @ 0x73<<12
        tloadr        r3, [r3, #0]
#ifdef MCU_CORE_8266_OTA_BOOT
        tcmp          r3, #0            @ 0x0
```

3) Modify "vendor/826x_ota_boot/main.c" correspondingly:

Modify the value of OTA_FLG_ADR as 0x41000.

```
#ifndef          OTA_FLG_ADR
#define          OTA_FLG_ADR          0x41000

#endif
```

6.3 8261 Flash architecture and OTA procedure

6.3.1 8261 FLASH storage architecture

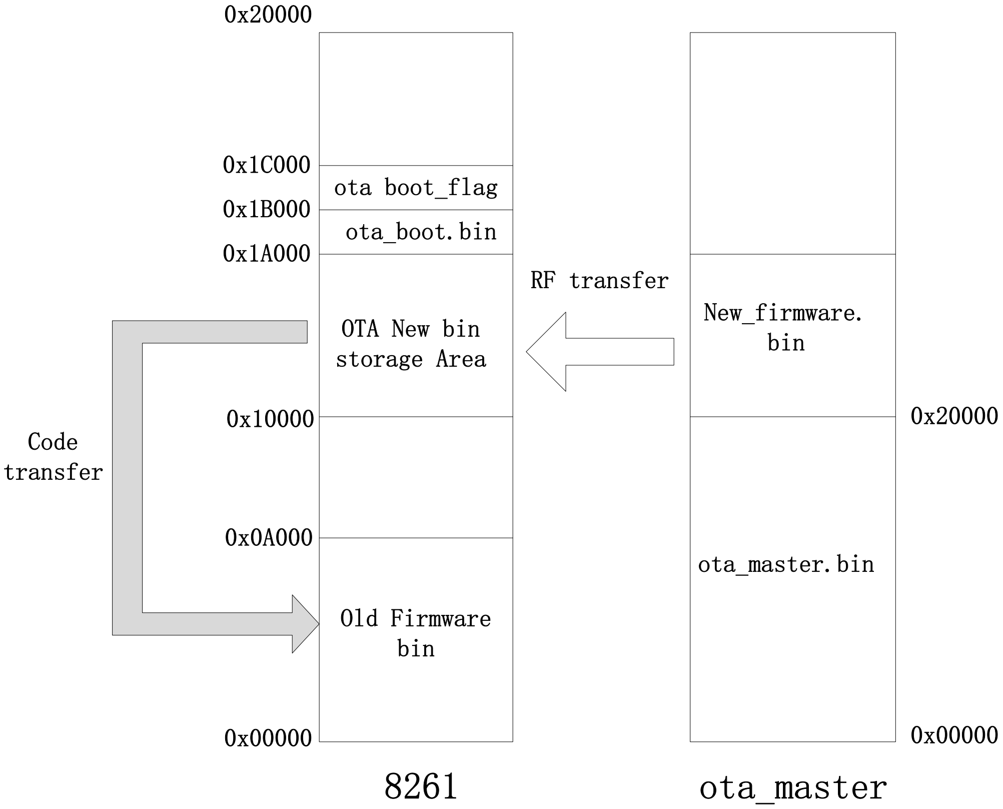


Figure6-6 8261 default Flash storage structure

In SDK, 104K flash space 0x00000 ~ 0x1A000 of 8261 is allocated as "40K+24K +40K" by default.

Figure6-6 shows the default Flash storage structure in 8261 BLE SDK. To realize OTA based on this structure, firmware size should not exceed 40K.

On Slave side, current old firmware is stored in the flash starting from 0 (40K area 0~0x0A000), ota_boot.bin is stored in the flash starting from 0x1A000 (1.5K area 0x1A000~0x1A600); flash starting from 0x10000 (40K area 0x10000~0x1A000) is used to store the New_firmware.bin obtained from OTA_Master via RF transfer, 1-byte boot_flag in address 0x1B000 is used as check flag during booting.

On OTA_Master side, ota_master.bin is stored in the flash starting from 0 (128K area 0~0x20000), while the new firmware of Slave is stored in the flash starting from 0x20000.

6.3.2 8261 OTA update procedure

Based on the flash storage structure introduced in Section 6.3.1, the OTA update procedure is illustrated as below:

1) Power on Slave normally

   After Slave flash is burned with Old_firmware.bin and ota_boot.bin, the boot_flag in flash 0x1B000 is 0xff.

   After power on, Slave MCU transfers beginning instructions of Old_firmware.bin starting from flash 0x00000 into SRAM address starting from 0x808000, and executes starting code corresponding to cstartup_8261.S. The starting code checks boot_flag value; since the value is not 0xa5, it serves as normal Slave function and executes c code of “Old_firmware.bin”.

2) Power on OTA_Master normally

   Power on sequence for OTA_Master and Slave does not matter. After OTA_Master and Slave are powered on and booted normally, BLE connection is established to ensure normal communication.

3) Trigger OTA mode

   Trigger OTA_Master to enter OTA mode (by button press or memory writting via PC tool). Send “OTA start” command by Master to make Slave enter OTA mode.

4) Transfer New_firmware.bin from OTA_Master to Slave via RF.

   After OTA_Master and Slave enter OTA mode, OTA_Master sends OTA packets with New_firmware data to Slave via RF. Slave will burn the analyzed data into its flash starting from 0x10000 (new_firmware storage area 0x10000~0x1A000).

5) After all OTA data are sent, OTA_Master sends “OTA End” command, and Slave is rebooted.

   After OTA process is finished successfully, New_firmware.bin is already stored in Slave flash starting from 0x10000. Slave will set the boot_flag value in flash

0x1B000 as “0xa5”, then reboot MCU.

6) Slave executes ota_boot.bin.

   After Slave is rebooted, MCU transfers beginning instructions of Old_firmware.bin starting from flash 0x00000 into SRAM starting from 0x808000, and executes the starting code corresponding to cstartup_8261.S in Old_firmware.bin. The starting code checks boot_flag value in flash 0x1B000; since the value is “0xa5”, Slave does not execute code of Old_firmware.bin, but transfers ota_boot.bin from 1.5K flash area 0x1A000~0x1A600 to SRAM starting from 0x808000 (0x808000~0x808600). After data transfer is finished, reset MCU so that MCU will execute code starting from SRAM 0x808000 (equivalent to executing ota_boot.bin) rather than re-transfer code from flash to SRAM.

7) ota_boot updates code, and Slave is rebooted.

   After the ota_boot.bin is executed, contents of New_firmware.bin will be read page by page starting from flash 0x10000 and written into flash starting from 0x00000. It’s equivalent to updating New_firmware.bin to flash starting from 0. Slave will set boot_flag value in flash 0x1B000 as “0x00”, then reboot MCU.

8) New_firmware.bin is executed normally.

   After Slave MCU is rebooted, it transfers code starting from flash 0 to SRAM starting from 0x808000, and checks the boot_flag. Since the value is not “0xa5”, it serves as normal Slave function.

   Similar to previous Old_firmware.bin, the New_firmware.bin supports OTA function, and user can restart OTA mode to upgrade firmware (the new code should be burned into OTA_Master flash starting from 0x20000 before OTA process).

The OTA related operations on Slave side decribed above have been realized in 8261 BLE SDK and can be used by user directly. On Master side, extra firmware design is needed and it will be introduced later.

6.3.3 cstartup_8261.S, reset, reboot , code transfer

6.3.3.1 boot_flag detect and process by cstartup_8261.S

The ota_master.bin is executed on OTA_Master side and its starting file has no special requirement.

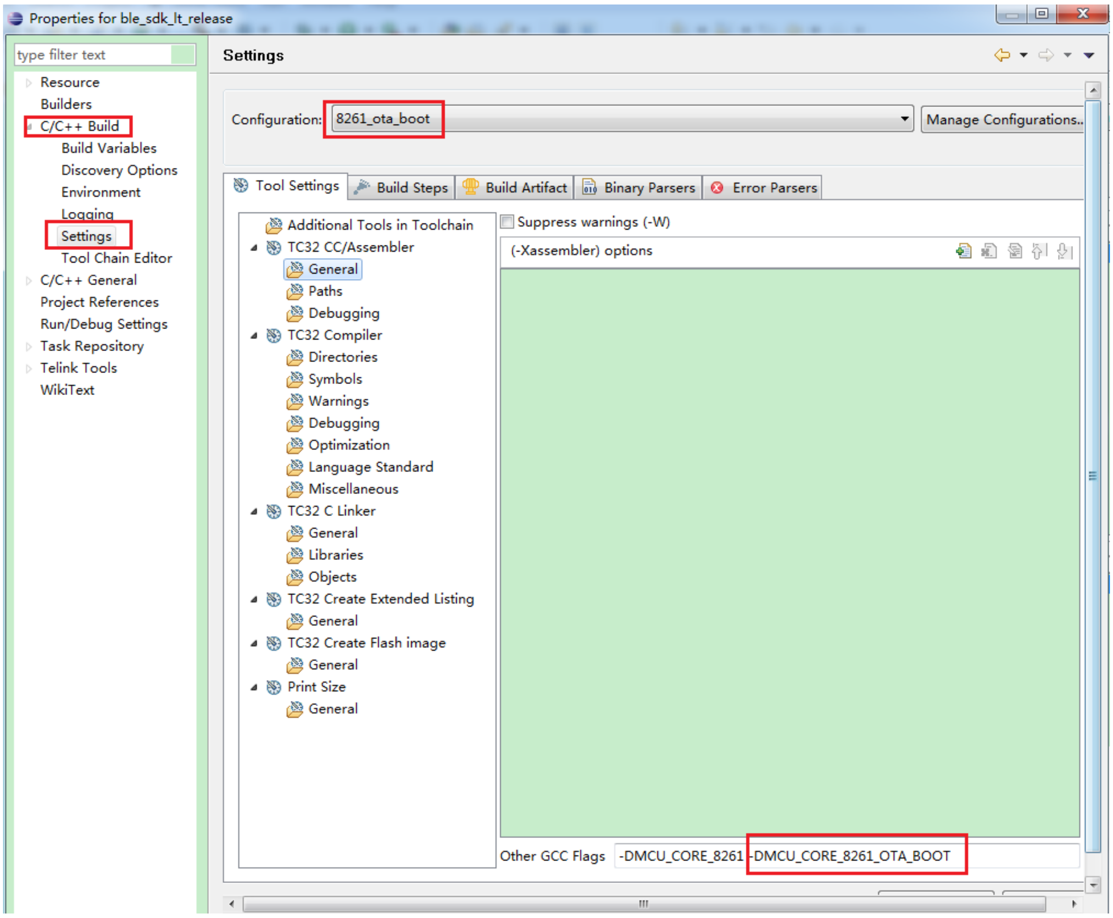
Old_firmware.bin, New_firmware.bin and ota_boot.bin are all executed on Slave side; they must have the same cstartup_8261.S in corresponding project, and consistent locations of iCache and iTag. Since New_firmware.bin is the update firmware to replace Old_firmware.bin, they have the same starting code of course. Old_firmware.bin differs from ota_boot.bin in starting code as shown below:

1) Old_firmware sets iCache and iTag; when ota_boot is executed, MCU is still powered on, and configuration of Old_firmware can be used directly.

2) ota_boot does not check boot_flag value; while Old_firmware will check boot_flag and process accordingly: if it’s not 0xa5, the Old_firmware is normally executed; if it’s 0xa5, ota_boot.bin in flash 1.5K area starting from 0x1A000 will be copied into SRAM starting from 0x808000, and MCU is reset.

According to the two points above, configurations in cstartup_8261.S from SETIC to COPY_DATA should be modified as below.

First define “MCU_CORE_8261_OTA_BOOT” in the bottom layer of ota_boot compile option (It can’t be defined in other projects).



Configuration: 8261_ota_boot

□ Suppress warnings (-W)

Other GCC Flags -DMCU_CORE_8261 -DMCU_CORE_8261_OTA_BOOT

Figure6-7 8261_ota_boot project setting

```
SETIC:
    tloadr        r1, DAT0 + 24
    tloadr      r0, DAT0 + 36                @ IC tag start
#ifdefMCU_CORE_8261_OTA_BOOT
```

```
    tloadr      r0, DAT0 + 36  @ The three sentences are only for
aligning.
    tloadr      r0, DAT0 + 36
    tloadr      r0, DAT0 + 36
#else
    tstorerb  r0, [r1, #0]
    tadd      r0, #1                   @ IC tag end
    tstorerb  r0, [r1, #1]
#endif
COPY_CODE_INIT:
     tmov     r3, #27          @ OTA FW ready flag at 0x1B000;
     tshftl      r3, r3, #12     @ 0x1B<<12 = 0x1B000
     tloadr      r3, [r3, #0]    @  read  value  of  flash  0x1B000
into R3
#ifdefMCU_CORE_8261_OTA_BOOT
tcmp      r3, #0 @ when ota boot starts, compare R3 with 0 (only
for aligning)
#else
     tcmp     r3,  #165 @  when  firmware  starts,  compare  R3  with
0xa5
#endif
     tjne     COPY_DATA_INIT  @  if  not  equal,  directly  jump  to
COPY_DATA_INIT,start  executing  firmware  normally;  if  equeal,
execute the following sentences, transfer the contents of flash
0x1A000~0x1A600 (i.e. ota_boot.bin) to SRAM 0x8000~0x8600.
     tmov     r2, #26 @ OTA boot code at: 0x1A000
     tloadr      r3, COPY_CODE_DAT
     tloadr      r0, COPY_CODE_DAT + 4
     tshftl      r2, r2, #12     @ 0x1A<<12 = 0x1A000
COPY_CODE_START:
     tloadm      r2!, {r1}
     tstorem  r3!, {r1}
     tcmp     r3, r0
    tjne     COPY_CODE_START
```

After data transfer, write register "core_602" with "0x88", reset MCU, restart to execute ota_boot.bin.

```
    tloadr      r3, COPY_CODE_DAT + 8
    tmov     r2, #136     @0x88
    tstorerb r2, [r3, #0]

COPY_CODE_END:
    tj    COPY_CODE_END
   .balign   4
COPY_CODE_DAT:
   .word  (0x808000)
   .word  (0x808600)
   .word  (0x800602)

COPY_DATA_INIT:
   tloadr    r1, DATA_I
   tloadr    r2, DATA_I+4
   tloadr    r3, DATA_I+8
COPY_DATA:
   ......
```

6.3.3.2 Firmware size, reset and reboot

Please refer to section 6.2.3.2 and 6.2.3.3 for details about firmware size, reset and reboot.

6.3.4 Modify Flash storage architecture

In SDK, by default maximum firmware size should not exceed 40K, the 40K flash area 0x00000~0x0A000 serves to store firmware, 0x10000~0x1A000 serves to store OTA new firmware, the sector starting from 0x1A000 serves to store 8261_ota_boot.bin, and 0x1B000 serves to store 1-byte ota boot_flag. The remaining 24K space 0x0A000~0x10000 is used as user data storage area.

All the addresses and firmware size above are modifiable via corresponding API.

6.3.4.1 Modify firmware size and OTA FW storage address

The API below serves to modify maximum firmware size and the starting address to store OTA new firmware.

void bls_ota_setFirmwareSizeAndOffset(int firmware_size_k,

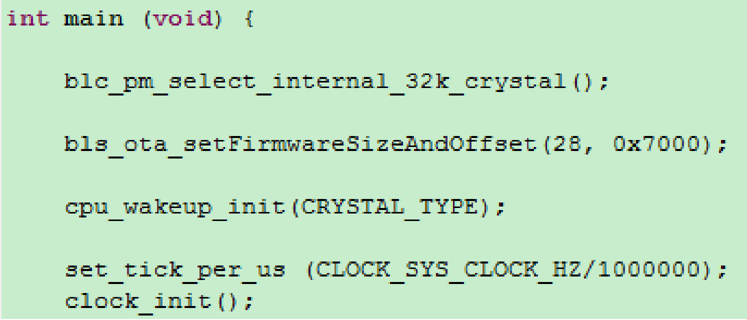
u32 ota_offset);

The API declaration is available in proj_lib/ble/service/ble_ll_ota.h.

“ota_offset”: This parameter indicates the starting address to store OTA new firmware.

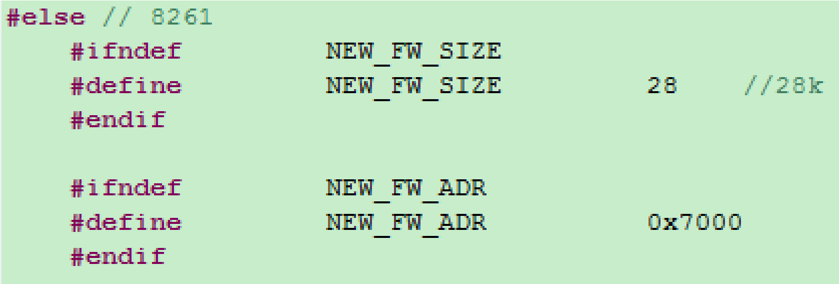
“firmware_size_k”: This parameter indicates the maximum firmware size in KB, and it must be configured as 4KB aligned. For example, suppose the maximum firmware size is 25KB, this parameter should be configured as “28”. This API can only be invoked before the cpu_wakeup_init() of main.c to take effect.

Suppose maximum firmware size is 25K, the 28K flash space 0x00000~0x07000 is used to store firmware, and the 28K space 0x07000 ~ 0x0E000 is used to store OTA new firmware, the remaining 48K space 0x0E000 ~ 0x1A000 is used to store user data, the configuration is shown as below:



```
int main (void) {

    blc_pm_select_internal_32k_crystal();

    bls_ota_setFirmwareSizeAndOffset(28, 0x7000);

    cpu_wakeup_init(CRYSTAL_TYPE);

    set_tick_per_us (CLOCK_SYS_CLOCK_HZ/1000000);
    clock_init();
```

Note that the value of “NEW_FW_SIZE” and “NEW_FW_ADR” in vendor/826x_ota_boot/main.c of SDK must be modified correspondingly, as shown below.

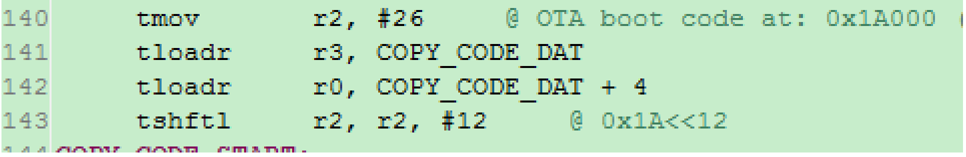


```
#else // 8261
    #ifndef         NEW_FW_SIZE
    #define         NEW_FW_SIZE         28    //28k
    #endif

    #ifndef         NEW_FW_ADR
    #define         NEW_FW_ADR          0x7000
    #endif
```

6.3.4.2 Modify storage address of OTA boot bin

In SDK, 8261_ota_boot.bin is stored in the flash sector starting from 0x1A000 by default, and its maximum size is 1.5K. The starting address to store 8261_ota_boot.bin is modifiable in cstartup_8261.S.

Suppose the sector starting from 0x18000 serves to store 8261_ota_boot.bin, replace “26” with “24” (24=0x18, 0x18<<12=0x18000) in the code below (proj/mcu_spec/cstartup_8261.S).



```
        tmov        r2, #26      @ OTA boot code at: 0x1A000
        tloadr      r3, COPY_CODE_DAT
        tloadr      r0, COPY_CODE_DAT + 4
        tshftl      r2, r2, #12      @ 0x1A<<12
```

6.3.4.3 Modify storage addrss of OTA boot flag

In SDK, 1-byte ota boot flag is stored in flash address 0x1B000 by default. The address to store ota boot flag is modifiable.

Suppose the address 0x19000 serves to store ota boot flag, follow the modifications below:

1) Invoke the API below during initialization:

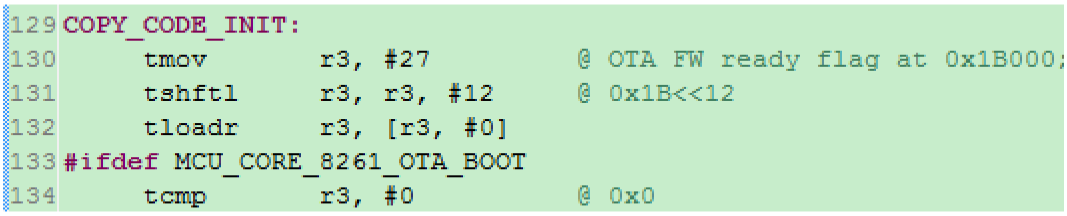
void bls_ota_setBootFlagAddr(u32 bootFlag_addr);

The API declaration is available in proj_lib/ble/service/ble_ll_ota.h.

bls_ota_setBootFlagAddr(0x19000);

2) Modify proj/mcu_spec/cstartup_8261.S:

Replace “27” with “25” (25=0x19, 0x19<<12=0x19000) in the code below.



```
COPY_CODE_INIT:
        tmov        r3, #27           @ OTA FW ready flag at 0x1B000;
        tshftl      r3, r3, #12       @ 0x1B<<12
        tloadr      r3, [r3, #0]
#ifdef MCU_CORE_8261_OTA_BOOT
        tcmp        r3, #0            @ 0x0
```

3) Modify vendor/826x_ota_boot/main.c:

Modify the value of OTA_FLG_ADR as 0x19000.

```
#ifndef         OTA_FLG_ADR
#define         OTA_FLG_ADR         0x19000

#endif
```

6.4 RF data proceesing for OTA mode

6.4.1 OTA processing in Attribute Table on Slave side

First, it’s needed to add ota reference in app_att.c which contains the Attribute Table:

```
#include "../../proj_lib/ble/service/ble_ll_ota.h"。
```

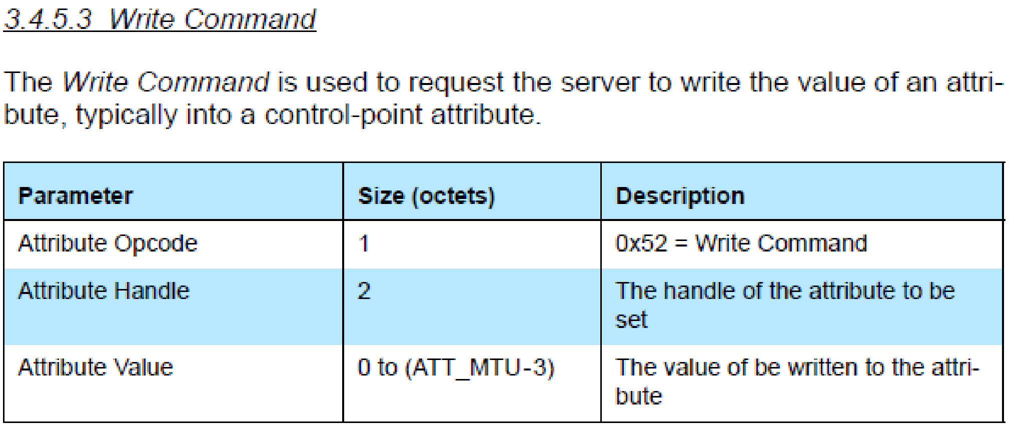
Second, add OTA related contents in the Attribute Table. The “att_readwrite_callback_t r” and “att_readwrite_callback_t w” of the OTA data Attribute should be set as otaRead and otaWrite, respectively; the attribute should be set as Read and Write_without_Rsp (Master sends data via Write Command, and does not need Slave to respond with ack to enable faster speed).

```
static u8 my_OtaProp= CHAR_PROP_READ | CHAR_PROP_WRITE_WITHOUT_RSP;


{0,2,1,1,(u8*)(&my_characterUUID), (u8*)(&my_OtaProp), 0},
{0,2,1,1,(u8*)(&my_OtaUUID),      (&my_OtaData), &otaWrite, &otaRead},
{0,2,sizeof (my_OtaName), sizeof (my_OtaName),(u8*)(&userdesc_UUID),
                                   (u8*)(my_OtaName), 0},
```

When Master sends OTA data to Slave, it actually writes data to the second Attribute as shown above, so Master needs to know the Attribute Handle of this Attribute in the Attribute Table. To use the Attribute Handle value pre-appointed by Master and Slave, user needs to count the Attribute Handle value, and then define it on Master side.

6.4.2 OTA data packet format

Master sends command and data to Slave via “Write Command” in L2CAP layer.



3.4.5.3 Write Command

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

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

Figure6-8 Write Command format in BLE stack

The Attribute Handle value is the handle_value of OTA data on Slave side.

The Attribute Value length is set as 20, and its format is shown as below.

OTA_cmd

0 1	19
OTA_CMD	invalid data

OTA_data

0 1	2 17	18 19
adr_index	firmware data:adr_index*16 - adr_index*16+15	CRC

Figure6-9 Format of OTA command and data

When the first two bytes are 0xff00 ~0xff10, it indicates it's an OTA command, and the command type is determined by the two bytes:

1) 0xff00: OTA_FW_VERSION, request to obtain current Slave firmware version number. This command is reserved and optional. To use this command, corresponding callback function is available on Slave side for user to transfer firmware version number.
2) 0xff01: OTA_Start command. To start OTA upgrade process, Master needs to send this command to Slave.
3) 0xff02: OTA_end command. When Master confirms all OTA data are correctly received by Slave, it will send this command, which can be followed by four valid bytes to double check Slave has received all data from Master.
4) 0xff03 ~ 0xff0f: to be added.

When the first two bytes are 0~0x1000, it indicates it's an OTA data. Each OTA data packet transfers 16-byte firmware data, and the adr_index is the actual firmware address divided by 16. "adr_index=0" indicates OTA data are values of firmware addresses 0x0~0xf; "adr_index=1" indicates OTA data are values of firmware addresses 0x10~0x1f. The last two bytes are the first CRC value calculated by CRC_16 operation to the former 18 bytes. After Slave receives the OTA data, it will also carry out CRC calculation, the data will be regarded as valid only when the result matches the CRC (19^{th}~20^{th} byte) of the data.

6.4.3 RF transfer processing on Master side

Since BLE link-layer RF data will be automatically responded with ack to avoid packet loss, during OTA data transfer Master won't check if every OTA data is responded with ack, that is, after sending an OTA data via write command, Master won't check if there's ack response from Slave by software, and directly push the following data into TX buffer as long as the number of data to be sent in TX buffer does not reach the threshold.

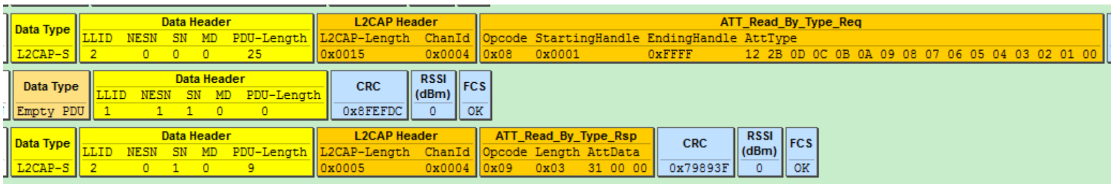
The OTA Master processes RF transfer by software as below:

1) Check if there's any action to trigger entering OTA mode. If so, Master enters OTA mode.

2) To send OTA commands and data to Slave, Master needs to know the Attribute Handle value of current OTA data Atrribute on Slave side. User can decide to directly use the pre-appointed value or obtain the Handle value via "Read By Type Request". UUID of OTA data in Telink BLE SDK is always 16-byte value as shown below:

```
#define TELINK_SPP_DATA_OTA
    {0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,
0x03,0x02,0x01,0x00}     //!< TELINK_SPP data for ota
```

In "Read By Type Request" from Master, the "Type" is set as the 16-byte UUID. The Attribute Handle for the OTA UUID is available from "Read By Type Rsp" responded by Slave. In the figure below, the Attribute Handle value is shown as "0x0031".



```
Data Type | Data Header                      | L2CAP Header              | ATT_Read_By_Type_Req
          | LLID  NESN  SN  MD  PDU-Length   | L2CAP-Length  ChanId      | Opcode StartingHandle EndingHandle AttType
L2CAP-S   | 2       0    0   0   25          | 0x0015        0x0004      | 0x08   0x0001         0xFFFF      12 2B 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

Data Type | Data Header                      | CRC       | RSSI (dBm) | FCS
          | LLID  NESN  SN  MD  PDU-Length   |           |            |
Empty PDU | 1       1    1   0   0           | 0x8FEFDC  | 0          | OK

Data Type | Data Header                      | L2CAP Header              | ATT_Read_By_Type_Rsp      | CRC       | RSSI (dBm) | FCS
          | LLID  NESN  SN  MD  PDU-Length   | L2CAP-Length  ChanId      | Opcode Length AttData     |           |            |
L2CAP-S   | 2       0    1   0   9           | 0x0005        0x0004      | 0x09   0x03   31 00 00    | 0x79893F  | 0          | OK
```

Figure6-10 Master obtains OTA Attribute Handle via "Read By Type Request"

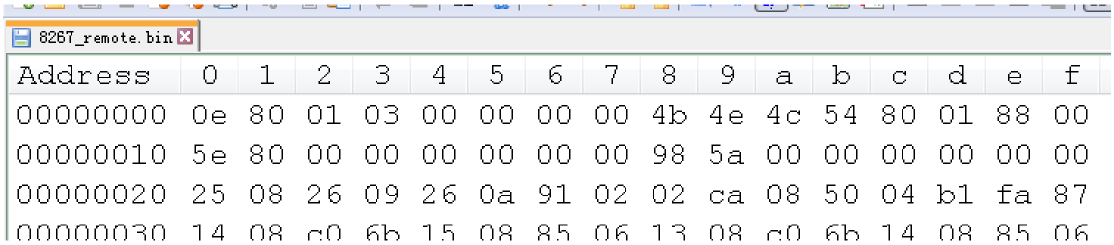
3) (optional) Obtain current Slave firmware version number. User can check if it's the newest version and decide whether to start OTA upgrade correspondingly. In 826x BLE SDK, user needs to determine the method to obtain FW version number.

An OTA version command is reserved, however, the transfer of version number is not realized in current 826x BLE SDK. An "OTA version cmd" can be sent to Slave in the form of "write cmd"; Slave only supplies a callback function after it receives the request, and user needs to decide in the callback function how to transfer Slave firmware version number to Master (e.g. manually send a NOTIFY/INDICATE data).

4) Start a timer when OTA starts, and continuously check if the count value exceeds the timeout duration (e.g. 15s, only for reference). If so, it's regarded as OTA failure due to timeout. Since Slave will check CRC after the OTA data are received, once there's CRC error or any other error (e.g. flash burning error), OTA fails, and firmware is directly rebooted; the link layer can't respond to Master with ack, and Master fails to send data until timeout.

5) Read four bytes of Master flash 0x20018~0x2001b to determine firmware size which is realized by compiler. Suppose firmware size is 20k (0x5000), the value of firmware 0x18~0x1b is 0x00005000, so the firmware size can be read from

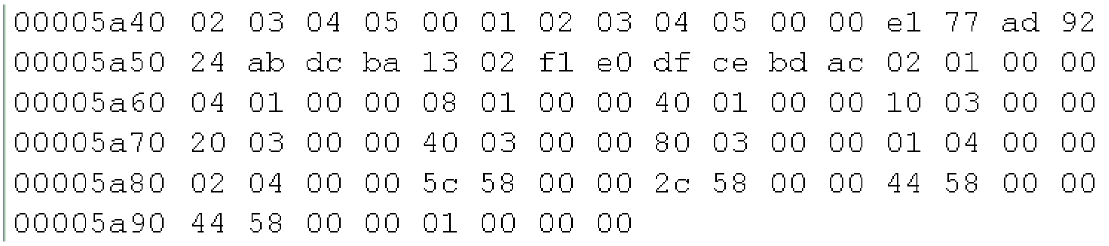
20018~0x2001b. As shown below, 0x18~0x1b of "8267_remote.bin" is "0x00005a98", so the firmware size is 0x5a98, i.e. 23192 bytes from 0 to 0x5a97.



8267_remote.bin

```
Address   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000  0e 80 01 03 00 00 00 00 4b 4e 4c 54 80 01 88 00
00000010  5e 80 00 00 00 00 00 00 98 5a 00 00 00 00 00 00
00000020  25 08 26 09 26 0a 91 02 02 ca 08 50 04 b1 fa 87
00000030  14 08 c0 6b 15 08 85 06 13 08 c0 6b 14 08 85 06
```

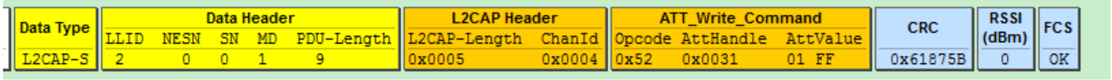
Figure6-11 firmware sample: starting part



```
00005a40  02 03 04 05 00 01 02 03 04 05 00 00 e1 77 ad 92
00005a50  24 ab dc ba 13 02 f1 e0 df ce bd ac 02 01 00 00
00005a60  04 01 00 00 08 01 00 00 40 01 00 00 10 03 00 00
00005a70  20 03 00 00 40 03 00 00 80 03 00 00 01 04 00 00
00005a80  02 04 00 00 5c 58 00 00 2c 58 00 00 44 58 00 00
00005a90  44 58 00 00 01 00 00 00
```

Figure6-12 firmware sample: ending part

6) Master sends an OTA start command "0xff01" to Slave, so as to inform it to enter OTA mode and wait for OTA data from Master, as shown below.



Data Type	Data Header					L2CAP Header		ATT_Write_Command			CRC	RSSI (dBm)	FCS
	LLID	NESN	SN	MD	PDU-Length	L2CAP-Length	ChanId	Opcode	AttHandle	AttValue			
L2CAP-S	2	0	0	1	9	0x0005	0x0004	0x52	0x0031	01 FF	0x61875B	0	OK

Figure6-13 master sends "OTA start"

7) Read 16-byte firmware each time starting from Master flash 0x20000, assemble them into OTA data packet, set corresponding adr_index, calculate CRC value, and push the packet into TX FIFO, until all data of the firmware are sent to Slave. OTA data format is used in data transfer : 20-byte valid data contains 2-byte adr_index, 16-byte firmware data and 2-byte CRC value to the former 18 bytes.

   Note: If firmware data for the final transfer are less than 16 bytes, the remaining bytes should be complemented with "0xff" and need to be considered for CRC calculation.

The 8267_remote.bin as shown in Figure6-11 and Figure6-12 is taken as an example to illustrate how to assemble OTA data.

Data for first transfer: "adr_index" is "0x00 00", 16-byte data are values of addresses 0x0000 ~ 0x000f. Suppose CRC calculation result for the former 18 bytes is "0xXYZW", the 20-byte data should be:

0x00 0x00 0x0e 0x80 ... (12 bytes not listed)... 0x88 0x00 0xZW 0xXY

Data for second transfer:

0x01 0x00 0x5e 0x80 ... (12 bytes not listed)... 0x00 0x00 0xJK 0xHI

Data for third transfer:

0x02 0x00 0x25 0x08 ... (12 bytes not listed)... 0xfa 0x87 0xNO 0xLM

....

Data for penultimate transfer:

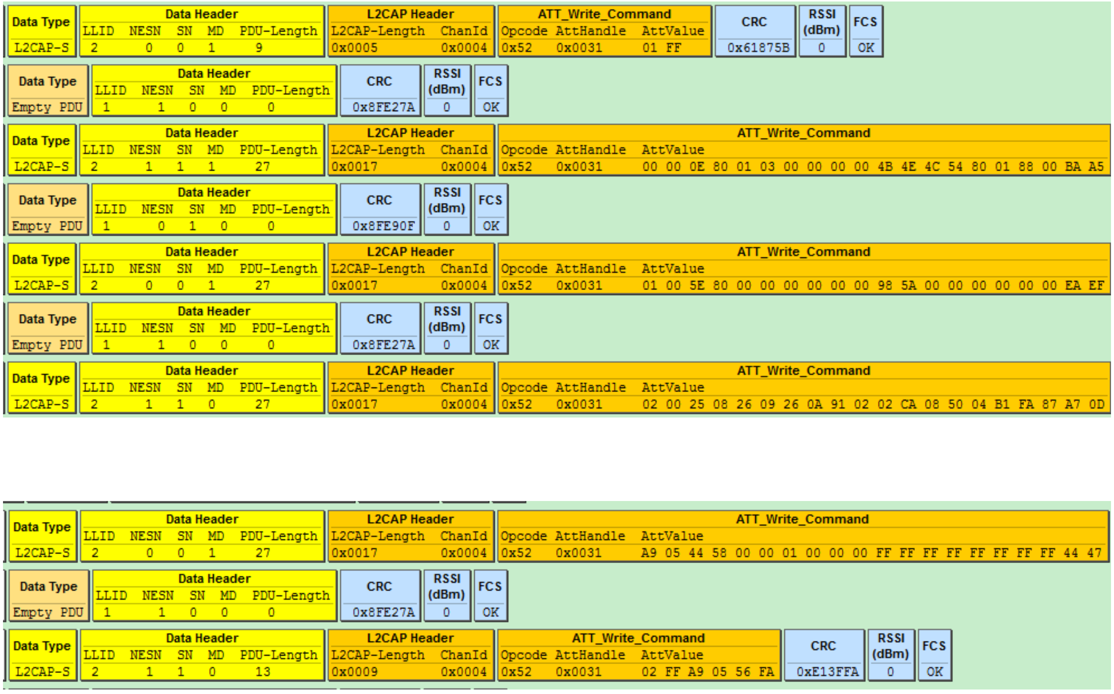
0xa8 0x05 0x02 0x04 ... (12 bytes not listed)... 0x00 0x00 0xST 0xPQ

Data for final transfer:

0xa9 0x05 0x44 0x58 0x00 0x00 0x01 0x00 0x00 0x00

0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xWX 0xUV

Since the firmware data for final transfer are only 8 bytes, eight "0xff" are added to complement 16 bytes. CRC calculation result for the former 18 bytes (0xa9 ~ 0xff) is "0xUVWX".



```
Data Type | Data Header                       | L2CAP Header             | ATT_Write_Command                  | CRC        | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId      | Opcode AttHandle AttValue          |            | (dBm) |
L2CAP-S   | 2    0    0  1  9                 | 0x0005       0x0004      | 0x52   0x0031    01 FF             | 0x61875B   | 0     | OK

Data Type | Data Header                       | CRC        | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length        |            |            |
Empty PDU | 1    1    0  0  0                 | 0x8FE27A   | 0          | OK

Data Type | Data Header                       | L2CAP Header             | ATT_Write_Command
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId      | Opcode AttHandle AttValue
L2CAP-S   | 2    1    1  1  27                | 0x0017       0x0004      | 0x52   0x0031    00 00 0E 80 01 03 00 00 00 00 4B 4E 4C 54 80 01 88 00 BA A5

Data Type | Data Header                       | CRC        | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length        |            |            |
Empty PDU | 1    0    1  0  0                 | 0x8FE90F   | 0          | OK

Data Type | Data Header                       | L2CAP Header             | ATT_Write_Command
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId      | Opcode AttHandle AttValue
L2CAP-S   | 2    0    0  1  27                | 0x0017       0x0004      | 0x52   0x0031    01 00 5E 80 00 00 00 00 00 00 98 5A 00 00 00 00 00 00 EA EF

Data Type | Data Header                       | CRC        | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length        |            |            |
Empty PDU | 1    1    0  0  0                 | 0x8FE27A   | 0          | OK

Data Type | Data Header                       | L2CAP Header             | ATT_Write_Command
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId      | Opcode AttHandle AttValue
L2CAP-S   | 2    1    1  0  27                | 0x0017       0x0004      | 0x52   0x0031    02 00 25 08 26 09 26 0A 91 02 02 CA 08 50 04 B1 FA 87 A7 0D


Data Type | Data Header                       | L2CAP Header             | ATT_Write_Command
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId      | Opcode AttHandle AttValue
L2CAP-S   | 2    0    0  1  27                | 0x0017       0x0004      | 0x52   0x0031    A9 05 44 58 00 00 01 00 00 00 FF FF FF FF FF FF FF FF 44 47

Data Type | Data Header                       | CRC        | RSSI (dBm) | FCS
          | LLID NESN SN MD PDU-Length        |            |            |
Empty PDU | 1    1    0  0  0                 | 0x8FE27A   | 0          | OK

Data Type | Data Header                       | L2CAP Header             | ATT_Write_Command                  | CRC        | RSSI  | FCS
          | LLID NESN SN MD PDU-Length        | L2CAP-Length ChanId      | Opcode AttHandle AttValue          |            | (dBm) |
L2CAP-S   | 2    1    1  0  13                | 0x0009       0x0004      | 0x52   0x0031    02 FF A9 05 56 FA | 0xE13FFA   | 0     | OK
```

Figure6-14 Master OTA data

8) After firmware data are sent, Master checks if BLE link-layer data are all sent out (Only when link-layer data is acked by Slave, it's considered the data is sent successfully). If all data are sent, Master will send an ota_end command to inform Slave.

"OTA end" packet is set as 6 valid bytes: first two bytes are "0xff02", followed by maximum adr_index value of new firmware (the two bytes are used to double check if there're OTA data lost on Slave side), the final two bytes are inverted value of the maximum adr_index (equivalent to simple check). CRC check is not needed for "OTA end".

The maximal adr_index and inverted value of "8267_remote.bin" are "0x05a9" and "0xfa56", respectively. Figure6-14 shows the final OTA end packet.

9) Check if link-layer TX FIFO on Master side is empty: If it's empty, it indicates all data and commands in above steps are sent successfully, i.e. OTA process on Master succeeds.

Please refer to Appendix for CRC_16 calculation function.

6.4.4 RF receive processing on Slave side

As introduced above, Slave can directly invoke the otaWrite and otaRead in OTA Attribute. After Slave receives write command from Master, it will be parsed and processed automatically in BLE stack by invoking the otaWrite function. In the otaWrite function, the 20-byte packet data will be parsed: first judge whether it's OTA CMD or OTA data, then process correspondingly (respond to OTA cmd; check CRC to OTA data and burn data into specific addresses of flash).

The OTA related operations on Slave side are shown as below:：

1) OTA_FIRMWARE_VERSION command is received (first two bytes are 0xff00): Master requests to obtain Slave firmware version number. In 826x BLE SDK, after Slave receives this command, it will only check whether related callback function is registered and determine whether to trigger the callback function correspondingly.

   The interface in ble_ll_ota.h to register this callback function is shown as below:

```
typedef void (*ota_versionCb_t)(void);
void bls_ota_registerVersionReqCb(ota_versionCb_t cb);
```

2) OTA start command is received (first two bytes are 0xff01): Slave enters OTA mode. If the "bls_ota_registerStartCmdCb" function is used to register the callback function of OTA start, then the callback function is executed to modify some parameter states after entering OTA mode (e.g. disable PM to stabilize OTA data transfer). Slave starts and maintains a slave_adr_index to record the adr_index of the latest correct OTA data. The initial value of slave_adr_index is -1, and it's used to judge whether there's packet loss in the whole OTA process; if so, OTA fails, Slave MCU exits OTA and reboots, since Master can't receive any ack packet from Slave, it will discover OTA failure by software after timeout.

   The following interface is used to register the callback function of OTA start:

```
typedefvoid (*ota_startCb_t)(void);
void bls_ota_registerStartCmdCb(ota_startCb_t cb);
```

User needs to register this callback function to carry out operations when OTA starts, for example, configure LED blinking to indicate OTA process. After Slave receives "OTA start", it enters OTA and starts a timer (The timeout duration is set as 30s by default in current SDK). If OTA process is not finished within the duration, it's regarded as OTA failure due to timeout. User can evaluate firmware size (larger size takes more time) and BLE data bandwidth on Master (narrow bandwidth will influence OTA speed), and modify this timeout duration accordingly via the interface as shown below.

```
void bls_ota_setTimeout(u32 timeout_us);// unit: us
```

3) Valid OTA data are received (first two bytes are 0~0x1000): Whenever Slave receives one 20-byte OTA data packet, it will first check if the adr_index equals slave_adr_index plus 1. If not equal, it indicates packet loss and OTA failure; if equal, the slave_adr_index value is updated. Then carry out CRC_16 check to the former 18 bytes; if not match, OTA fails; if match, the 16-byte valid data are written into corresponding addresses of flash (ota_program_offset+adr_index*16 ~ ota_program_offset+adr_index*16 + 15). During flash writing process, if there's any error, OTA also fails.

4) "OTA end" command is received (first two bytes are 0xff02): Check whether adr_max in OTA end packet and the inverted check value are correct. If yes, the adr_max can be used to double check whether maximum index value of data received by Slave from Master equals the adr_max in this packet. If equal, OTA succeeds; if not equal, OTA fails due to packet loss.

   After successful OTA, Slave will set the booting flag of the old firmware address in flash as 0, set the booting flag of the new firmware address in flash as 0x4b, then MCU reboots.

5) Slave supplies OTA state callback function:

   After Slave starts OTA, MCU will finally reboot regardless of OTA result. If OTA succeeds, Slave will set flag before rebooting so that MCU executes the new firmware; if OTA fails, the incorrect new firmware will be erased before rebooting, so that MCU still executes the old firmare. Before rebooting, user can judge whether the OTA state callback function is registered and determine whether to trigger it correspondingly.

```
typedef void (*ota_resIndicateCb_t)(int result);
```

```
enum{
    OTA_SUCCESS = 0,     //success
    OTA_PACKET_LOSS,     //lost one or more OTA PDU
    OTA_DATA_CRC_ERR,     //data CRC err
    OTA_WRITE_FLASH_ERR,  //write OTA data to flash ERR
    OTA_DATA_UNCOMPLETE,  //lost last one or more OTA PDU
    OTA_TIMEOUT,      //
};
    void bls_ota_registerResultIndicateCb
                         (ota_resIndicateCb_t cb);
```

The "enum" lists the 6 options for parameter "result": the first value indicates OTA success; the other five values indicate reasons for OTA failure. The "result" is mainly used for debugging: When OTA fails, user can read the "result", stop MCU by using "while(1)", and find the reason for current OTA failure.

LED indication can be added to indicate OTA success, as shown below:

```
void LED_show_ota_result(int result)
{
    irq_disable();
    WATCHDOG_DISABLE;

    gpio_set_output_en(GPIO_LED, 1);

    if(result == OTA_SUCCESS){  //OTA success
        gpio_write(GPIO_LED, 1);
        sleep_us(2000000);  //led on for 2s
        gpio_write(GPIO_LED, 0);
    }
    else{  //OTA fail

    }

    gpio_set_output_en(GPIO_LED, 0);
}


bls_ota_registerResultIndicateCb (LED_show_ota_result);
```

The otaWrite function on Slave is assembled in lib, while other related interfaces are available in proj_lib/ble/service/ble_ll_ota.h of SDK.

7 Key Scan

Keyscan architecture based on row/column scan is used to detect and process key state update (press/release). User can directly use the sample code, or realize the function by developing his own code.

7.1 Key matrix

Take Telink 8267 Demo board as an example: It's a 4*6 matrix and supports up to 24 buttons. Four drive pins (Row0~Row3) serve to output drive level, while six scan pins (CoL0~CoL5) serve to scan for button press in current column.

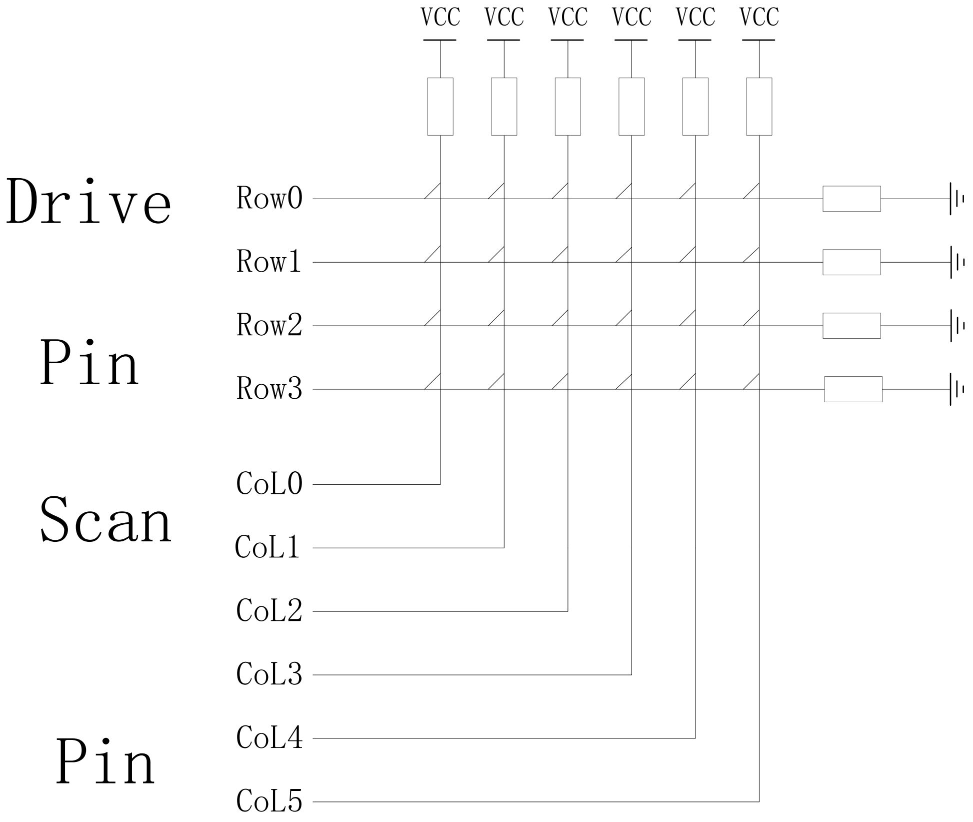


Figure7-1 Row/Column key matrix

Keyscan related configurations in app_config.h are shown as below:

On Telink demo board, Row0~Row3 pins are PB1, PB2, PB3 and PB6, while CoL0~CoL5 pins are PD4, PD5, PD6, PD7, PE0 and PE1.

Define drive pin array and scan pin array:

```
#define  KB_DRIVE_PINS  {GPIO_PB1, GPIO_PB2, GPIO_PB3, GPIO_PB6}
#define  KB_SCAN_PINS   {GPIO_PD4, GPIO_PD5, GPIO_PD6, GPIO_PD7,
                          GPIO_PE0, GPIO_PE1}
```

Keyscan adopts analog pull-up/pull-down resistor in 8267 IC: drive pins use 100K pull-down resistor, and scan pins use 10K pull-up resistor. When no button is pressed, scan pins act as input GPIOs and read high level due to 10K pull-up resistor. When key scan starts, drive pins output low level; if low level is detected on a scan pin, it indicates there's button pressed in current column (Note: Drive pins are not in float state, if output is not enabled, scan pins still detect high level due to voltage division of 100K and 10K resistor.)

Define valid voltage level detected on scan pins when drive pins output low level in Row/Column scan:

```
#define    KB_LINE_HIGH_VALID             0
```

Define pull-up resistor for scan pins and pull-down resistor for drive pins:

```
#define    MATRIX_ROW_PULL                PM_PIN_PULLDOWN_100K

#define    MATRIX_COL_PULL                PM_PIN_PULLUP_10K


#define    PULL_WAKEUP_SRC_PB1      MATRIX_ROW_PULL
#define    PULL_WAKEUP_SRC_PB2      MATRIX_ROW_PULL
#define    PULL_WAKEUP_SRC_PB3      MATRIX_ROW_PULL
#define    PULL_WAKEUP_SRC_PB6      MATRIX_ROW_PULL


#define    PULL_WAKEUP_SRC_PD4      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PD5      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PD6      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PD7      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PE0      MATRIX_COL_PULL
#define    PULL_WAKEUP_SRC_PE1      MATRIX_COL_PULL
```

Since "ie" of general GPIOs is set as 0 by default in gpio_init, to read level on scan pins, corresponding "ie" should be enabled.

```
#define PD4_INPUT_ENABLE    1
#define PD5_INPUT_ENABLE    1
#define PD6_INPUT_ENABLE    1
#define PD7_INPUT_ENABLE    1
#define PE0_INPUT_ENABLE    1
#define PE1_INPUT_ENABLE    1
```

When MCU enters suspend or deepsleep, it’s needed to configure CORE/PAD GPIO wakeup. Set drive pins as high level wakeup; when there’s button pressed, drive pin reads high level, which is 10/11 VCC (i.e. VCC * 100K/(100K+10K)). To read level state of drive pins, corresponding “ie” should be enabled.

```
#define PB1_INPUT_ENABLE    1
#define PB2_INPUT_ENABLE    1
#define PB3_INPUT_ENABLE    1
#define PB6_INPUT_ENABLE    1
```

7.2 Keyscan, keymap and keycode

7.2.1 Keyscan

After configuration as shown in section 7.1, the function below is invoked in mainloop to implement keyscan.

```
u32 kb_scan_key (int numlock_status, int read_key)
```

- numlock_status: Generally set as 0 when invoked in mainloop. Set as “KB_NUMLOCK_STATUS_POWERON” only for fast keyscan after wakeup from deepsleep (refer to section 7.5, corresponding to DEEPBACK_FAST_KEYSCAN_ENABLE).
- read_key: Buffer processing for key values, generally not used and set as 1 (if it’s set as 0, key values will be pushed into buffer and not reported to upper layer).
- The return value is used to inform user whether matrix keyboard update is detected by current scan: if yes, return 1; otherwise return 0.

The kb_scan_key function is invoked in mainloop. As introduced in section 3.2.4, each main loop is an adv_interval or conn_interval. In advertising state (suppose adv_interval is 30ms), key scan is processed once for each 30ms; in connection state (suppose conn_interval is 10ms), key scan is processed once for each 10ms.

In theory, when button states in matrix are different during two adjacent key scans, it’s considered as an update. In actual code, a debounce filtering processing is enabled: It will be considered as a valid update, only when button states stay the same during two adjacent key scans, but different with the latest stored matrix keyboard state. “1” will be returned by the function to indicate valid update, matrix keyboard state will be indicated by the structure “kb_event”, and current button state will be updated to the newest matrix keyboard state. Corresponding code in keyboard.c is shown as below:

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
```

The newest button state means press or release state set of all (24) buttons in the matrix. When power on, initial matrix keyboard state shows all buttons are “released” by default, and debounce filtering processing is enabled; as long as valid update occurs to the button state, “1” will be returned, otherwise “0” will be returned. For example: press a button, a valid update is returned; release a button, a valid update is returned; press another button with a button held, a valid update is returned; press the third button with two buttons held, a valid update is returned; release a button of the two pressed buttons, a valid update is returned……

7.2.2 Keymap &kb_event

If a valid button state update is detected by invoking the “kb_scan_key”, user can obtain current button state via a global structure variable “kb_event”.

```
#define KB_RETURN_KEY_MAX   6

typedef struct{
    u8 cnt;
    u8 ctrl_key;
    u8 keycode[KB_RETURN_KEY_MAX];
}kb_data_t;
kb_data_t  kb_event;
```

The “kb_event” consists of 8 bytes:

“cnt” serves to indicate valid count number of pressed buttons currently;

“ctrl_key” is not used generally, and it will be used only for standard USB HID keyboard (user is not allowed to set keycode in keymap as 0xe0~0xe7).

keycode[6] indicates keycode of up to six pressed buttons can be stored (if more than six buttons are pressed actually, only the former six can be reflected).

Keycode definition of 24 buttons in app_config.h is shown as below:

```
#define        KB_MAP_NORMAL {\
          {CR_VOL_MUTE, VK_3,     VK_1,      VK_MEDIA, }, \
          {VK_2,        VK_5,     VK_M,      VK_4,     }, \
          {CR_RIGHT,    VK_NONE,  CR_SEL,       CR_LEFT,   }, \
          {CR_REWIND,      VK_NONE,  CR_DN,     CR_HOME,   }, \
          {CR_VOL_UP,      VK_NONE,  VK_MMODE,  CR_VOL_DN, }, \
          {VK_WEB,      VK_NONE,  CR_UP,     CR_POWER, }, }
```

The keymap follows the format of 4*6 matrix structure. The keycode of pressed button can be configured accordingly, for example, the keycode of the button between Row0 and CoL0 is “CR_VOL_MUTE”.

In the "kb_scan_key" function, the "kb_event.cnt" will be cleared before each scan, while the array "kb_event.keycode[]" won't be cleared automatically. Whenever "1" is returned to indicate valid update, the "kb_event.cnt" will be used to check current valid count number of pressed buttons.

1) If current kb_event.cnt = 0, previous valid matrix state "kb_event.cnt" must be uncertain non-zero value; the update must be button release, but the released button number is uncertain. Data in kb_event.keycode[] (if available) is invalid.

2) If kb_event.cnt = 1, the previous kb_event.cnt indicates button state update. If previous kb_event.cnt is 0, it indicates the update is one button is pressed; if previous kb_event.cnt is 2, it indicates the update is one of the two pressed buttons is released; if previous kb_event.cnt is 3, it indicates the update is two of the three pressed buttons are released……

   kb_event.keycode[0] indicates the key value of currently pressed button. The subsequent keycodes are negligible.

3) If kb_event.cnt = 2, the previous kb_event.cnt indicates button state update. If previous kb_event.cnt is 0, it indicates the update is two buttons are pressed at the same time; if previous kb_event.cnt is 1, it indicates the update is another button is pressed with one button held; if previous kb_event.cnt is 3, it indicates the update is one of the three pressed buttons is released……

   kb_event.keycode[0] and kb_event.keycode[1] indicate key values of the two pressed buttons currently. The subsequent keycodes are negligible.

User can manually clear the "kb_event.keycode" before each key scan, so that it can be used to check whether valid update occurs, as shown in the example below.

In the sample code, when `kb_event.keycode[0]`is not zero, it's considered a button is pressed, but the code won't check further whether two buttons are pressed at the same time or one of the two pressed buttons is released.

```
kb_event.keycode[0] = 0;// manually clear keycode[0]
int det_key = kb_scan_key (0, 1);

if (det_key)
{
    key_not_released = 1;

    u8 key = kb_event.keycode[0];
    if (key)           //key press
    {
        key_buf[2] = key;
```

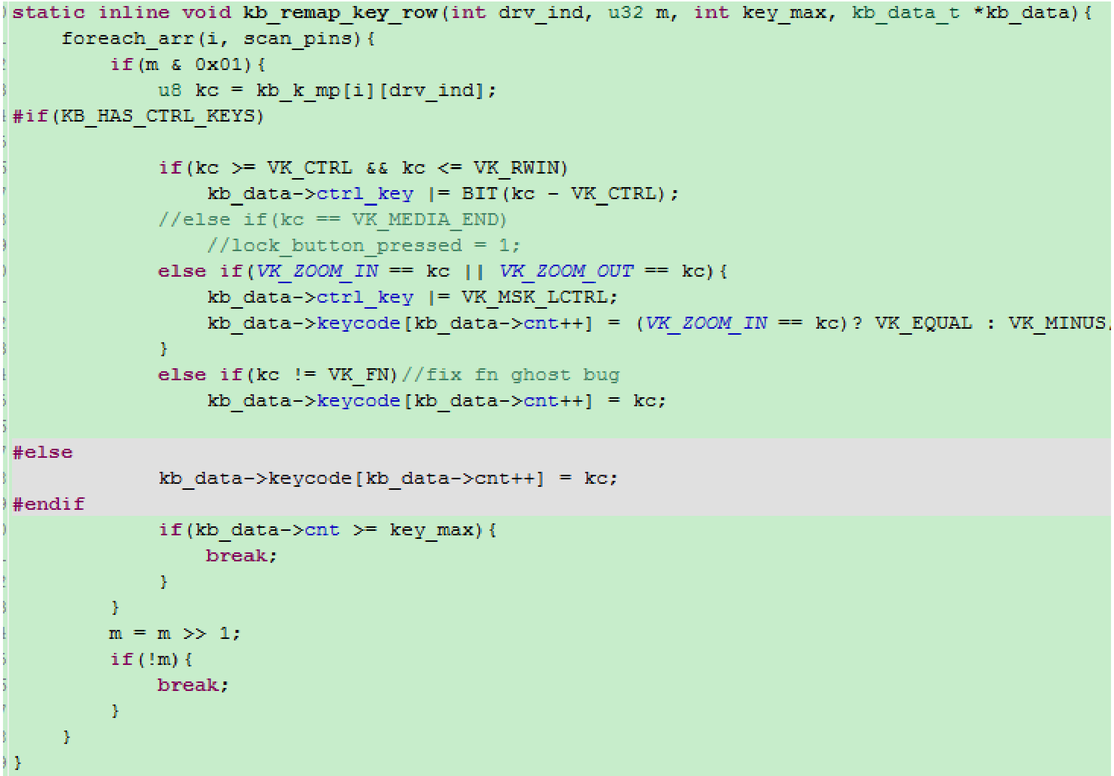
```
            //send key press
            blt_push_notify_data (HID_HANDLE_KEYBOARD_REPORT, key_buf,
                                                              8);
        }
        else
        {
            key_not_released = 0;
            key_buf[2] = 0;
            //send key release
            blt_push_notify_data (HID_HANDLE_KEYBOARD_REPORT, key_buf,
                                                              8);
        }
    }
```

7.3 Keycode

The section above introduces keymap definition in app_config.h and keycode filling in KB_MAP_NORMAL. To realize standard USB HID keyboard, some special keycodes need to be processed, so user should pay attention to details for keycode definition.

The “kb_remap_key_row” function in keyboard.c serves to process keycode.

```
void kb_remap_key_row(int drv_ind, u32 m, int key_max, kb_data_t
                                                        *kb_data)
```



```
static inline void kb_remap_key_row(int drv_ind, u32 m, int key_max, kb_data_t *kb_data){
    foreach_arr(i, scan_pins){
        if(m & 0x01){
            u8 kc = kb_k_mp[i][drv_ind];
#if(KB_HAS_CTRL_KEYS)

            if(kc >= VK_CTRL && kc <= VK_RWIN)
                kb_data->ctrl_key |= BIT(kc - VK_CTRL);
            //else if(kc == VK_MEDIA_END)
                //lock_button_pressed = 1;
            else if(VK_ZOOM_IN == kc || VK_ZOOM_OUT == kc){
                kb_data->ctrl_key |= VK_MSK_LCTRL;
                kb_data->keycode[kb_data->cnt++] = (VK_ZOOM_IN == kc)? VK_EQUAL : VK_MINUS
            }
            else if(kc != VK_FN)//fix fn ghost bug
                kb_data->keycode[kb_data->cnt++] = kc;

#else
            kb_data->keycode[kb_data->cnt++] = kc;
#endif
            if(kb_data->cnt >= key_max){
                break;
            }
        }
        m = m >> 1;
        if(!m){
            break;
        }
    }
}
```

Figure7-2 keycode processing function

CTRL KEY will be obtained by kb_event.ctrl_key, and its keycode ranges from 0xe0 to 0xe7 which cannot be used by user.

In proj/drivers/usbkeycode.h:

```
#define    VK_CTRL         0xe0
#define    VK_SHIFT        0xe1
#define    VK_ALT          0xe2
#define    VK_WIN          0xe3
#define    VK_RCTRL        0xe4
#define    VK_RSHIFT       0xe5
#define    VK_RALT         0xe6
#define    VK_RWIN         0xe7
```

For the following key values, after they are transferred by Slave to Telink Master Dongle, special processing will be realized by PC, and it depends on report descriptor configuration of BLE HID in app_att.c.

```
enum{
VK_EXT_START =              0xa0,

VK_SYS_START =              VK_EXT_START,  //0xa0
VK_SLEEP =                  VK_SYS_START,  //0xa0,    sleep
VK_POWER,                           //0xa1,    power
VK_WAKEUP,                          //0xa2, wake-up
VK_SYS_END,                            //0xa3
    VK_SYS_CNT =          (VK_SYS_END - VK_SYS_START),//0xa3-0xa0=0x03

VK_MEDIA_START =      VK_SYS_END,        //0xa3
VK_W_SRCH =           VK_MEDIA_START,    //0xa3
VK_WEB,                                  //0xa4
VK_W_BACK,
VK_W_FORWRD,
VK_W_STOP,
VK_W_REFRESH,
VK_W_FAV,                           //0xa9
VK_MEDIA,
VK_MAIL,
VK_CAL,
VK_MY_COMP,
VK_NEXT_TRK,
VK_PREV_TRK,
VK_STOP,       //b0
VK_PLAY_PAUSE,
VK_W_MUTE,
VK_VOL_UP,
VK_VOL_DN,

VK_MEDIA_END,
VK_EXT_END =      VK_MEDIA_END,
    VK_MEDIA_CNT =    (VK_MEDIA_END - VK_MEDIA_START),//0xb5-0xa3=0x12

    VK_ZOOM_IN =          (VK_MEDIA_END + 1),//0xb6

    VK_ZOOM_OUT ,                          //0xb7

}
```

7.4 Keyscan flow

7.4.1 Basic keyscan flow

When kb_scan_key is invoked, a basic keyscan flow is shown as below:

1) Initial full scan through the whole matrix.

   All drive pins output drive level (0). Meanwhile read all scan pins, check for valid level, and record the column on which valid level is read. (The scan_pin_need is used to mark valid column number.)

```
scan_pin_need = kb_key_pressed (gpio);
```

If row-by-row scan is directly adopted without initial full scan through the whole matrix, each time all rows (four rows in current demo firmware) should be scanned at least, even if no button is pressed. To save scan time, initial full scan through the whole matrix can be added, thus it will directly exit keyscan if no button press is detected on any column.

In the kb_key_pressed function, all rows output low level, and stabilized level of scan pins will be read after 20us delay. A release_cnt is set as 6; if a detection shows all pressed buttons in the matrix are released, it won't consider no button is pressed and stop row-by-row scan immediately, but buffers for six frames. If six successive detections show buttons are all released, it will stop row-by-row scan. Thus key debounce processing is realized.

2) Scan row by row according to full scan result through the whole matrix.

If button press is detected by full scan, row-by-row scan is started: Drive pins (ROW0~ROW3) output valid drive level row by row; read level on columns, and find the pressed button. Following is related code:

```
        u32 pressed_matrix[ARRAY_SIZE(drive_pins)] = {0};

        kb_scan_row (0, gpio);
        for (int i=0; i<=ARRAY_SIZE(drive_pins); i++) {
            u32 r = kb_scan_row (i < ARRAY_SIZE(drive_pins) ? i : 0,
gpio);
            if (i) {
                pressed_matrix[i - 1] = r;
            }

        }
```

The following methods are used to optimize code execution time for row-by-row scan.

- When a row outputs drive level, it's not needed to read level of all columns (CoL0~CoL5). Since the scan_pin_need marks valid column number, user can read the marked columns only.
- After a row outputs drive level, a 20us or so delay is needed to read stabilized level of scan pins, and a buffer processing is used to utilize the waiting duration.

The array variable "u32 pressed_matrix[4]" (up to 32 columns are supported) is used to store final matrix keyboard state: pressed_matrix[0] bit0~bit5 mark button state on CoL0~CoL5 crossed with Row0……pressed_matrix[3] bit0~bit5 mark button state on CoL0~CoL5 crossed with Row3.

3) Debounce filtering for pressed_matrix[].

```
unsigned int key_debounce_filter( u32 mtrx_cur[], u32 filt_en );
```

```
u32 key_changed = key_debounce_filter( pressed_matrix, \
            (numlock_status & KB_NUMLOCK_STATUS_POWERON) ? 0 : 1);
```

During fast keyscan after wakeup from deepsleep, "numlock_status" equals "KB_NUMLOCK_STATUS_POWERON", the "filt_en" is set as 0 to skip filtering and fast obtain key values. In other cases, the "filt_en" is set as 1 to enable filtering. Only when pressed_matrix[] stays the same during two adjacent key scans, but different from the latest valid pressed_matrix[], the "key_changed" is set as 1 to indicate valid update in matrix keyboard.

4) Buffer processing for pressed_matrix[].

Push pressed_matrix[] into buffer. When the "read_key" in "**kb_scan_key** (**int** numlock_status, **int** read_key)" is set as 1, the data in the buffer will be read out immediately. When the "read_key" is set as 0, the buffer stores the data without notification to the upper layer; the buffered data won't be read until the read_key is 1.

In current SDK, the "read_key" is fixed as 1, i.e. the buffer does not take effect actually.

5) According to pressed_matrix[], look up the KB_MAP_NORMAL table and return key values.

Corresponding functions are "kb_remap_key_code" and "kb_remap_key_row".

7.4.2 Keyscan flow timing optimization

As introduced above, even if no button is pressed, each mainloop takes about 100us to execute initial full scan through the whole matrix at least.

GPIO IRQ status bit inquiry can be used to optimize the time for full scan with no button pressed.

As shown in PM section, in "user_init" all drive GPIO pins are configured as high-level CORE wakeup for suspend.

```
u32 pin[] = KB_DRIVE_PINS;
```

```
for (int i=0; i<(sizeof (pin)/sizeof(*pin)); i++)
{
    gpio_set_wakeup(pin[i],1,1);    //drive pin core(gpio) high
                                        wakeup suspend
}
```

The “gpio_set_wakeup(pin[i],1,1)” sets wakeup polarity of drive pins as high level and enables wakeup.

Since GPIO interrupt enabling and polarity adopts the same configuration registers as wakeup, the “gpio_set_wakeup(pin[i],1,1)” will also enable GPIO interrupt and set interrupt polarity as high level.

High level on GPIO will set GPIO IRQ service flag bit (core_648 BIT(18)); this flag bit can be used to check whether any button is pressed (when a button is pressed, 10/11 VCC high level will be read on corresponding drive pin).

```
#define reg_irq_mask         REG_ADDR32(0x640)
#define reg_irq_src          REG_ADDR32(0x648)

FLD_IRQ_GPIO_EN =        BIT(18),
```

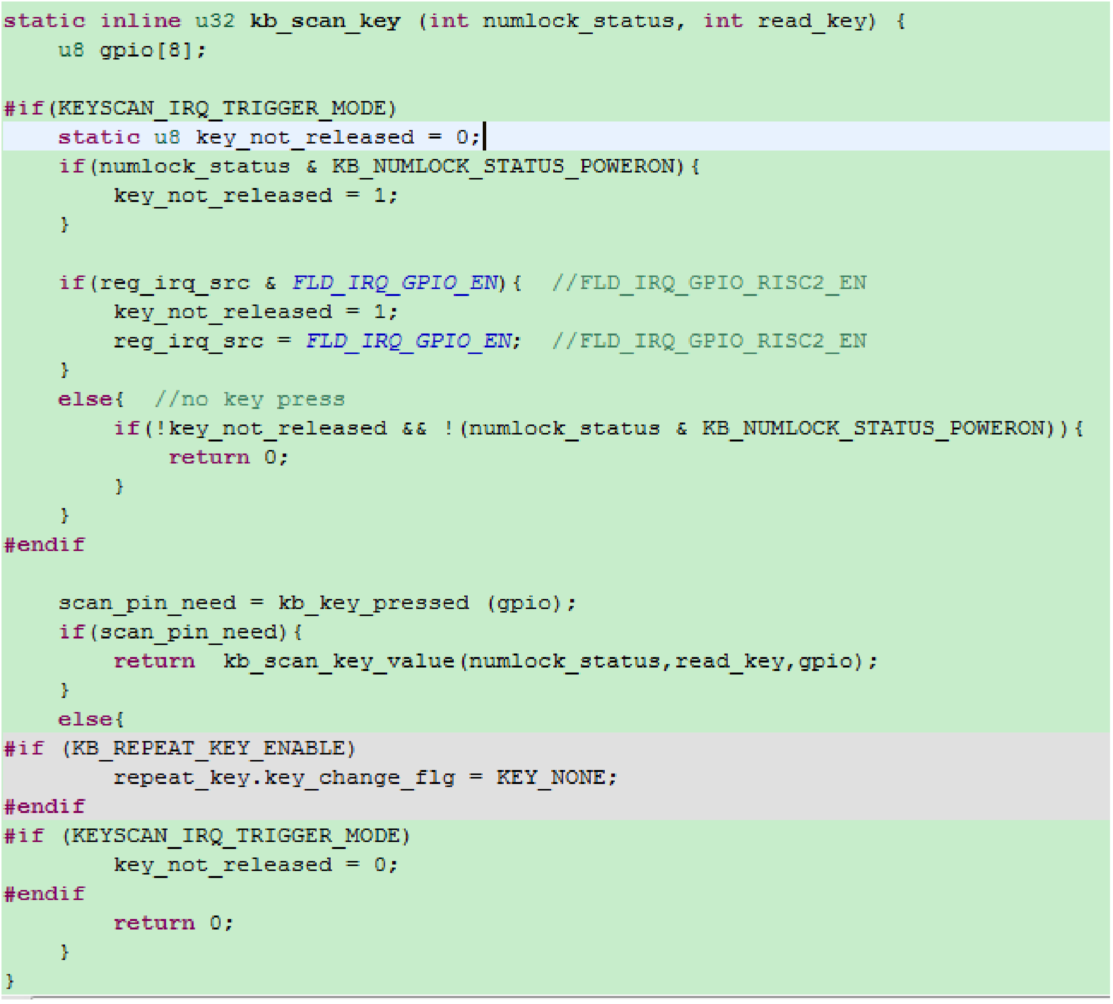
As long as GPIO interrupt mask bit (core_640 BIT(18)) is not enabled, the configuration will only set the IRQ flag bit, but won’t trigger interrupt.

The “KEYSCAN_IRQ_TRIGGER_MODE” definition in app_config.h serves to enable time optimization for the keyscan flow.

```
#define KEYSCAN_IRQ_TRIGGER_MODE       1
```

Initialization:

```
    gpio_core_irq_enable_all(1);

    reg_irq_src = FLD_IRQ_GPIO_EN;
```



```
static inline u32 kb_scan_key (int numlock_status, int read_key) {
    u8 gpio[8];

#if(KEYSCAN_IRQ_TRIGGER_MODE)
    static u8 key_not_released = 0;
    if(numlock_status & KB_NUMLOCK_STATUS_POWERON){
        key_not_released = 1;
    }

    if(reg_irq_src & FLD_IRQ_GPIO_EN){   //FLD_IRQ_GPIO_RISC2_EN
        key_not_released = 1;
        reg_irq_src = FLD_IRQ_GPIO_EN;   //FLD_IRQ_GPIO_RISC2_EN
    }
    else{  //no key press
        if(!key_not_released && !(numlock_status & KB_NUMLOCK_STATUS_POWERON)){
            return 0;
        }
    }
#endif

    scan_pin_need = kb_key_pressed (gpio);
    if(scan_pin_need){
        return  kb_scan_key_value(numlock_status,read_key,gpio);
    }
    else{
#if (KB_REPEAT_KEY_ENABLE)
        repeat_key.key_change_flg = KEY_NONE;
#endif
#if (KEYSCAN_IRQ_TRIGGER_MODE)
        key_not_released = 0;
#endif
        return 0;
    }
}
```

Figure7-3 Keyscan time optimization

As shown above, it will first check whether IRQ flag bit is set after previous keyscan is finished. If yes, it indicates there’s button press action during this duration; since manual button press lasts for 200ms at least, the pressed button is not released yet, and the subsequent basic keyscan flow (including full scan and row-by-row scan) will be executed.

After the pressed button is released, the debounce function in kb_key_pressed takes effect. Only when six successive detections all show button release state, the keyscan flow will be stopped.

7.5 Deepsleep wakeup fast keyscan

After Slave enters deepsleep during connection state, it can be woke up by button press action. After wakeup, firmware is rebooted; in mainloop following user_init, Slave will first send adv packets, establishes connection, and then sends the key value to BLE Master.

Though 826x BLE SDK adopts some processing to speed up the deepback (resumption after wakeup from deepsleep), the duration may still reach several hundreds of milliseconds (e.g. 300ms). To avoid action loss of the wakeup pin, fast keyscan and data buffer are added. Fast keyscan is designed to avoid potential button action loss caused by re-initialization time after MCU reboots and debounce filter processing time during keyscan in mainloop. Data buffer is designed considering valid button data detected in adv state and pushed into BLE TX FIFO will be cleared after entering connection state.

The macro "DEEPBACK_FAST_KEYSCAN_ENABLE" in app_config.h is used to control fast keyscan and data buffer.

```
#define DEEPBACK_FAST_KEYSCAN_ENABLE   1

void deep_wakeup_proc(void)
{
    #if(DEEPBACK_FAST_KEYSCAN_ENABLE)
    if(analog_read(DEEP_ANA_REG0) == CONN_DEEP_FLG){
        if(kb_scan_key (KB_NUMLOCK_STATUS_POWERON,1) && kb_event.cnt){
            deepback_key_state = DEEPBACK_KEY_CACHE;
            key_not_released = 1;
            memcpy(&kb_event_cache,&kb_event,sizeof(kb_event));
        }
    }
    #endif
}
```

During initialization key scan is processed before user_init. After it's detected by reading retention analog register that MCU enters deep wakeup from connection state, the "kb_scan_key" is invoked to directly obtain the whole matrix button state without enabling the debounce filtering. If key scan process shows a button is pressed (button state update is returned, and kb_event.cnt in non-zero value), the "kb_event" variable will be copied to the cache variable "kb_event_cache".

The "deepback_pre_proc" and "deepback_post_proc" processing are added in keyscan during mainloop.

```
void proc_keyboard (u8 e, u8 *p)
{
    kb_event.keycode[0] = 0;
    int det_key = kb_scan_key (0, 1);

#if(DEEPBACK_FAST_KEYSCAN_ENABLE)
    if(deepback_key_state != DEEPBACK_KEY_IDLE){
        deepback_pre_proc(&det_key);
    }
#endif

    if (det_key){
        key_change_proc();
    }

#if(DEEPBACK_FAST_KEYSCAN_ENABLE)
    if(deepback_key_state != DEEPBACK_KEY_IDLE){
        deepback_post_proc();
    }
#endif
}
```

The “deepback_pre_proc” realizes buffer processing of fast keyscan value, as shown below: After connection is established between Slave and Master, if no button state update is detected in a kb_key_scan, the buffered kb_event_cache value will be used as the current newest button state update.

For button release processing, it’s needed to check current matrix keyboard state: If there’s button pressed, since actual button release generates a release action, it’s not needed to add manual release; if current button is released, it’s needed to mark that a manual release event should be added, otherwise button may fail to be released since buffered button press event stays valid.

The “deepback_pre_proc” specifies whether manual release is needed. The “deepback_post_proc” will determine whether to push a button release event into BLE TX FIFO accordingly.

7.6 Repeat Key processing

When a button is pressed and held, it’s needed to enable repeat key function to repeatedly send the key value with a specific interval.

The “repeat key” function is masked by default. By configuring related macros in app_config.h, this function can be controlled correspondingly.

```
//repeat key
#define KB_REPEAT_KEY_ENABLE          0
#defineKB_REPEAT_KEY_INTERVAL_MS      200
#define KB_REPEAT_KEY_NUM             1

#define KB_MAP_REPEAT                 {VK_1, }
```

1) KB_REPEAT_KEY_ENABLE

   This macro serves to enable or mask the repeat key function. To use this function, first set “KB_REPEAT_KEY_ENABLE” as 1.

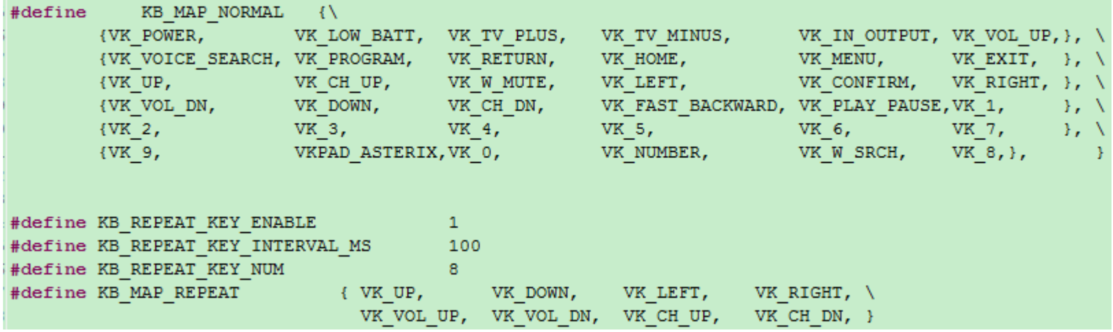
2) KB_REPEAT_KEY_INTERVAL_MS

   This macro serves to set the repeat interval time. For example, if it’s set as 200ms, it indicates when a button is held, kb_key_scan will return an update with the interval of 200ms. Current button state will be available in kb_event.

3) KB_REPEAT_KEY_NUM 和 KB_MAP_REPEAT

   The two macros serve to define current repeat key values: KB_REPEAT_KEY_NUM specifies the number of keycodes, while the KB_MAP_REPEAT defines a map to specify all repeat keycodes. Note that the keycodes in the KB_MAP_REPEAT must be the values in the KB_MAP_NORMAL.

Following example shows a 6*6 matrix keyboard: by configuring the four macros, eight buttons including UP, DOWN, LEFT, RIGHT, V+, V-, CHN+ and CHN- are set as repeat keys with repeat interval of 100ms, while other buttons are set as non-repeat keys.



```
#define     KB_MAP_NORMAL   {\
        {VK_POWER,        VK_LOW_BATT,  VK_TV_PLUS,   VK_TV_MINUS,      VK_IN_OUTPUT, VK_VOL_UP,}, \
        {VK_VOICE_SEARCH, VK_PROGRAM,   VK_RETURN,    VK_HOME,          VK_MENU,      VK_EXIT,  }, \
        {VK_UP,           VK_CH_UP,     VK_W_MUTE,    VK_LEFT,          VK_CONFIRM,   VK_RIGHT, }, \
        {VK_VOL_DN,       VK_DOWN,      VK_CH_DN,     VK_FAST_BACKWARD, VK_PLAY_PAUSE,VK_1,     }, \
        {VK_2,            VK_3,         VK_4,         VK_5,             VK_6,         VK_7,     }, \
        {VK_9,            VKPAD_ASTERIX,VK_0,         VK_NUMBER,        VK_W_SRCH,    VK_8,},     }


#define KB_REPEAT_KEY_ENABLE           1
#define KB_REPEAT_KEY_INTERVAL_MS      100
#define KB_REPEAT_KEY_NUM              8
#define KB_MAP_REPEAT           { VK_UP,      VK_DOWN,    VK_LEFT,    VK_RIGHT, \
                                  VK_VOL_UP,  VK_VOL_DN,  VK_CH_UP,   VK_CH_DN, }
```

User can search for the four macros in the project to locate the code about repeat key.

7.7 Stuck Key processing

Stuck key processing is used to save power when one or multiple buttons of a remote control/keyboard is/are pressed and held for a long time unexpectedly, for example a RC is pressed by a cup or ashtray. If keyscan detects some button is pressed and held, without the stuck key processing, MCU won’t enter deepsleep or other low power state since it always considers the button is not released.

Following are two related macros in the app_config.h:

```
//stuck key
#define STUCK_KEY_PROCESS_ENABLE       0

#define STUCK_KEY_ENTERDEEP_TIME       60//in s
```

By default the stuck key processing function is masked. User can set the "STUCK_KEY_PROCESS_ENABLE" as 1 to enable this function. The "STUCK_KEY_ENTERDEEP_TIME" serves to set the stuck key time: if it's set as 60s, it indicates when button state stays fixed for more than 60s with some button held, it's considered as stuck key, and MCU will enter deepsleep.

User can search for the macro "STUCK_KEY_PROCESS_ENABLE" to locate related code in keyboard.c, as shown below:

```
#if (STUCK_KEY_PROCESS_ENABLE)
    u8 stuckKeyPress[ARRAY_SIZE(drive_pins)];

#endif
```

An u8-type array stuckKeyPress[4] is defined to record row(s) with stuck key in current key matrix. The array value is obtained in the function "key_debounce_filter".

Upper-layer processing is shown as below:

```
kb_event.keycode[0] = 0;
int det_key = kb_scan_key (0, 1);

if (det_key){
    #if (STUCK_KEY_PROCESS_ENABLE)
    if(kb_event.cnt){  //key press
        stuckKey_keyPressTime = clock_time();
    }
    #endif

    .......
}
```

For each button state update, when button press is detected (i.e. kb_event.cnt is non-zero value), the "stuckKey_keyPressTime" is used to record the time for the latest button press state.

Processing in the blt_pm_proc is shown as below:

```
#if (STUCK_KEY_PROCESS_ENABLE)
   if(key_not_released && clock_time_exceed(stuckKey_keyPressTime,
STUCK_KEY_ENTERDEEP_TIME*1000000)){
        u32 pin[] = KB_DRIVE_PINS;
        for (int i=0; i<(sizeof (pin)/sizeof(*pin)); i++)
        {
            extern u8 stuckKeyPress[];
            if(stuckKeyPress[i]){
                cpu_set_gpio_wakeup (pin[i],0,1);   //reverse stuck
                                                           key  pad  wakeup
level
            }
        }
        cpu_sleep_wakeup(1, PM_WAKEUP_PAD, 0);  //deepsleep
    }
#endif
```

Check whether the latest pressed button is held for more than 60s: if yes, it's considered as stuck key, all row numbers with stuck key will be obtained via the bottom-layer "stuckKeyPress[]"; then modify corresponding PAD wakeup polarity as low level from high level, so that MCU can enter deepsleep and wake up by button release normally (when button is pressed, corresponding drive pin reads high level of 10/11 VCC; after release, the drive pin turns to low level).

7.8 Power optimization for long key press

Power optimization can be enabled for long pressed keys, by enabling the macro "LONG_PRESS_KEY_POWER_OPTIMIZE". Please refer to the PM section for details.

8 LED Management

8.1 LED task related invoking functions

Source code about LED management is available in vendor/common/blt_led.c of 826x BLE SDK for user reference. User can directly include the "vendor/common/blt_led.h" into his C file.

User needs to invoke the following three functions:

```
    void device_led_init(u32 gpio,u8 polarity);
    int device_led_setup(led_cfg_t led_cfg);
    static inline void device_led_process(void);
```

During initialization, the "device_led_init(u32 gpio,u8 polarity)" is used to set current GPIO and polarity corresponding to LED. If "polarity" is set as 1, it indicates LED will be turned on when GPIO outputs high level; if "polarity" is set as 0, it indicates LED will be turned on when GPIO outputs low level.

The "device_led_process" function is added in UI Entry of mainloop. It's used to check whether LED task is not finished (DEVICE_LED_BUSY). If yes, MCU will carry out corresponding LED task operation.

8.2 LED task configuration and management

8.2.1 Led event definition

The following structure serves to define a LED event.

```
    typedef struct{
        unsigned short onTime_ms;
        unsigned short offTime_ms;
        unsigned char repeatCount;  //0xff special for long
                                on(offTime_ms=0)/long  off(onTime_ms=0)
        unsigned char priority;     //0x00 < 0x01 < 0x02 < 0x04 < 0x08
<                                    0x10 < 0x20 < 0x40 < 0x80

    } led_cfg_t;
```

The unsigned short int type "onTime_ms" and "offTime_ms" specify light on and off time (unit: ms) for current LED event, respectively. The two variables can reach the maximum value of 65535.

The unsigned char type "repeatCount" specifies blinking times (i.e. repeat times for light on and off action specified by the "onTime_ms" and "offTime_ms"). The variable can reach the maximum value of 255.

The “priority” specifies the priority level for current LED event.

To define a LED event when the LED light stays on/off, set the “repeatCount” as 255(0xff), set “onTime_ms”/“offTime_ms” as 0 or non-zero correspondingly.

LED event examples:

1) Blink for 3s with the frequency of 1Hz: turn on for 500ms, turn off for 500ms, and repeat for 3 times.

```
led_cfg_t led_event1 = {500,      500 ,      3,          0x00,   };
```

2) Blink for 50s with the frequency of 4Hz: turn on for 125ms, turn off for 125ms, and repeat for 200 times.

```
led_cfg_t     led_event2    =    {125,     125 ,     200,          0x00,   };
```

3) Always on: onTime_ms is non-zero, offTime_ms is zero, and repeatCount is 0xff.

```
led_cfg_t     led_event3    = {100,     0 ,          0xff,       0x00,    };
```

4) Always off: onTime_ms is zero, offTime_ms is non-zero, and repeatCount is 0xff.

```
led_cfg_t     led_event4    =    {0,         100,          0xff,       0x00,    };
```

5) Turn on for 3s, and then turn off: onTime_ms is 1000, offTime_ms is 0, and repeatCount is 0x3.

```
led_cfg_t     led_event5    =    {1000,     0,       3,        0x00,    };
```

The “device_led_setup” can be invoked to deliver a led_event to LED task management.

```
device_led_setup(led_event1);
```

8.2.2 Led event priority

User can define multiple LED events in SDK, however, only a LED event is allowed to be executed at the same time. No task list is set for the simple LED management: When LED is idle, LED will accept any LED event delivered by invoking the “device_led_setup”. When LED is busy with a LED event (old LED event), if another event (new LED event) comes, MCU will compare priority level of the two LED events; if the new LED event has higher priority level, the old LED event will be discarded and MCU starts to execute the new LED event; if the new LED event has the same or lower priority level, MCU continues executing the old LED event, while the new LED event will be completely discarded, rather than buffered.

By defining LED events with different priority levels, user can realize corresponding LED indicating effect.

Since inquiry scheme is used for LED management, MCU should not enter long suspend (e.g. 10ms * 50 = 500ms) with latency enabled and LED task ongoing (DEVICE_LED_BUSY); otherwise LED event with small onTime_ms value (e.g. 250ms) won't be responded in time, thus LED blinking effect will be influenced.

```
#define    DEVICE_LED_BUSY   (device_led.repeatCount)
```

The corresponding processing is needed to add in blt_pm_proc, as shown below:

```
user_task_flg = scan_pin_need || key_not_released || DEVICE_LED_BUSY;

if(user_task_flg){
    bls_pm_setManualLatency(0);    // manually disable latency

}
```

User can refer to the code in current 826x ble remote project for LED related processing.

9 blt software timer

Telink BLE SDK supplies source code of blt software timer demo for user reference on timer task. User can directly use this timer or modify as needed.

Source code are available in "vendor/common/blt_soft_timer.c" and "blt_soft_timer.h". To use this timer, the macro below should be set as 1.

```
#define     BLT_SOFTWARE_TIMER_ENABLE       0   //enable or disable
```

Since blt software timer is inquiry timer based on system tick, it cannot reach the accuracy of hardware timer, and it should be continuously inquired during mainloop. The blt soft timer applies to the use case with timing value more than 5ms and without high requirement for time error.

Its key feature is: This timer will be inquired during mainloop, and it ensures MCU can wake up in time from suspend and execute timer task. This design is implemented based on "Timer wakeup of APP layer" (section 4.8).

Current design can run up to four timers, and maximum timer number is modifiable via the macro below:

```
#define     MAX_TIMER_NUM     4   //timer max number
```

9.1 Timer initialization

The API below is used for blt software timer initialization:

```
void   blt_soft_timer_init(void);
```

Timer initialization only registers "blt_soft_timer_process" as callback function of APP layer wakeup in advance.

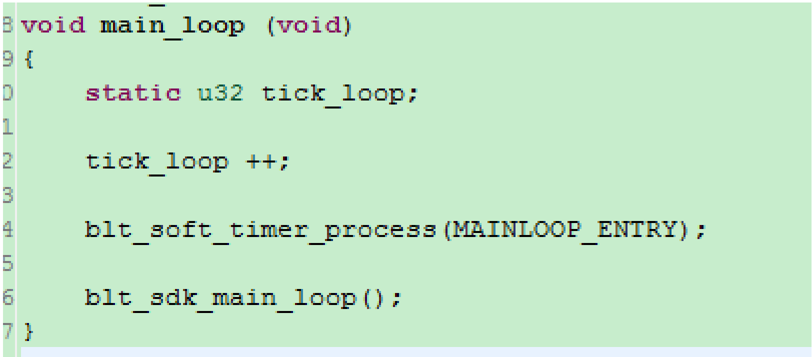
```
void   blt_soft_timer_init(void)
{
   bls_pm_registerAppWakeupLowPowerCb(blt_soft_timer_process);
}
```

9.2 Timer inquiry processing

The function "blt_soft_timer_process" serves to implement inquiry processing of blt software timer.

```
void   blt_soft_timer_process(int type);
```

On one hand, mainloop should always invoke this function in the location as shown in the figure below. On the other hand, this function must be registered as callback function of APP layer wakeup in advance. Whenever MCU is woke up from suspend in advace by timer, this function will be quickly executed to process timer task.



```
void main_loop (void)
{
    static u32 tick_loop;

    tick_loop ++;

    blt_soft_timer_process(MAINLOOP_ENTRY);

    blt_sdk_main_loop();
}
```

The parameter “type” of the “blt_soft_timer_process” indicates two cases to enter this function: If “type” is 0, it indicates entering this function via inquiry in mainloop; if “type” is 1, it indicates entering this function when MCU is woke up in advance by timer.

```
#define    MAINLOOP_ENTRY                    0
#define    CALLBACK_ENTRY                    1
```

The implementation of “blt_soft_timer_process” is rather complex, and its basic principle is shown as below:

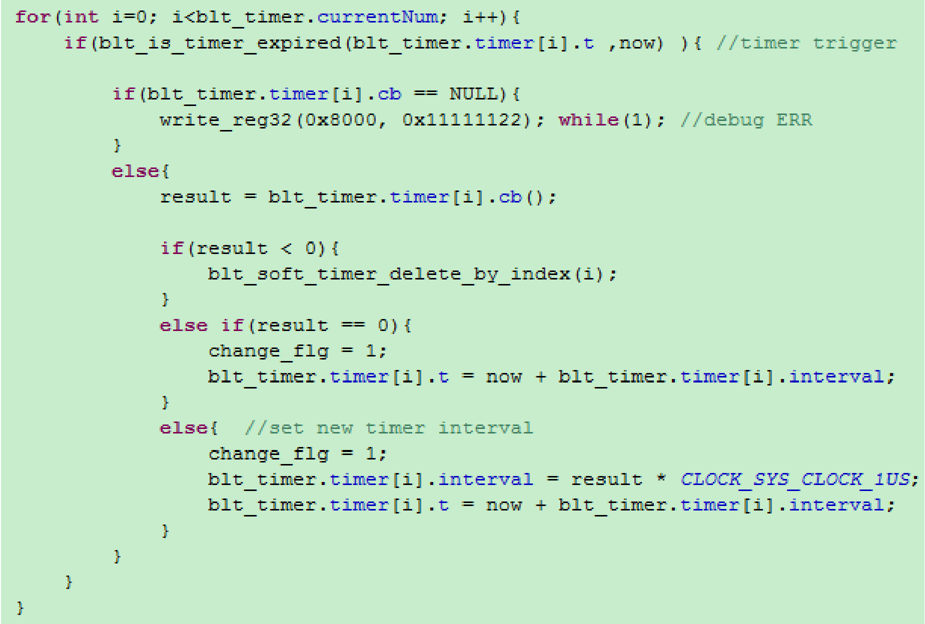
1) First check whether there is still user-defined timer in current timer table.If not, directly exit the function and disable timing wakeup of APP layer; if there’s timer task, continue the flow.

```
if(!blt_timer.currentNum){
    bls_pm_setAppWakeupLowPower(0, 0);  //disable
    return;

}
```

2) Check whether the nearest timer task is reached: if the task is not reached, exit the function; otherwise continue the flow. Since the design will ensure all timers are time-ordered, herein it’s only needed to check the nearest timer.

```
if( !blt_is_timer_expired(blt_timer.timer[0].t, now) ){
    return;

}
```

3) Inquire all current timer tasks, and execute corresponding task as long as timer value is reached.



```
for(int i=0; i<blt_timer.currentNum; i++){
    if(blt_is_timer_expired(blt_timer.timer[i].t ,now) ){ //timer trigger

        if(blt_timer.timer[i].cb == NULL){
            write_reg32(0x8000, 0x11111122); while(1); //debug ERR
        }
        else{
            result = blt_timer.timer[i].cb();

            if(result < 0){
                blt_soft_timer_delete_by_index(i);
            }
            else if(result == 0){
                change_flg = 1;
                blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
            }
            else{  //set new timer interval
                change_flg = 1;
                blt_timer.timer[i].interval = result * CLOCK_SYS_CLOCK_1US;
                blt_timer.timer[i].t = now + blt_timer.timer[i].interval;
            }
        }
    }
}
```

The code above shows processing of timer task function: If the return value of this function is less than 0, this timer task will be deleted and won't be responded; if the return value is 0, the previous timing value will be retained; if the return value is more than 0, this return value will be used as the new timing cycle (unit: us).

4) In step 3), if tasks in timer task table change, the previous time sequence may be disturbed, and re-ordering is needed.

```
if(change_flg){
        blt_soft_timer_sort();

}
```

5) If the nearest timer task will be responded within 3s (it can be modified as a value larger than 3s as needed) from now, the response time will be set as wakeup time of APP layer in advance; otherwise APP layer wakeup in advance will be disabled.

```
if( (u32)(blt_timer.timer[0].t - now) < 3000 * CLOCK_SYS_CLOCK_1MS){
        bls_pm_setAppWakeupLowPower(blt_timer.timer[0].t, 1);
}
else{
        bls_pm_setAppWakeupLowPower(0, 0);  //disable

}
```

9.3 Add timer task

The API below serves to add timer task.

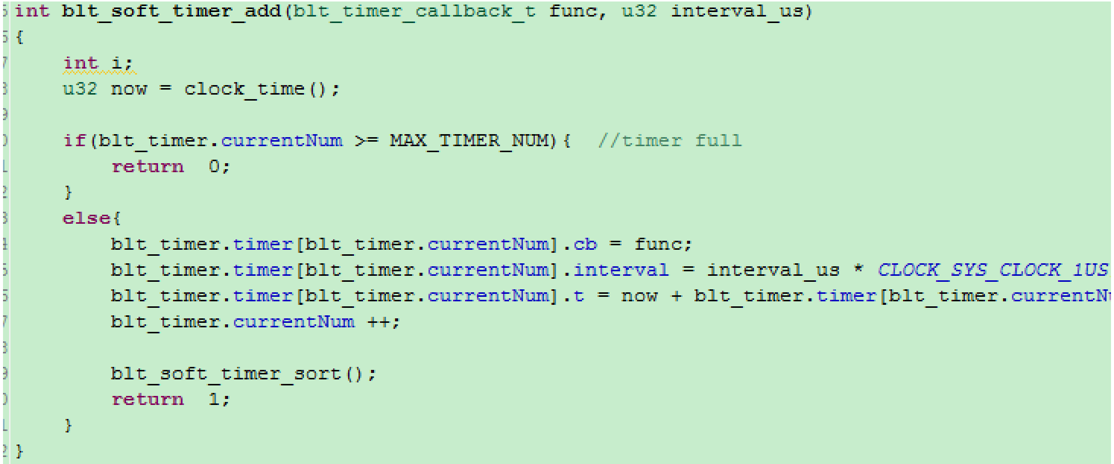
```
typedef int (*blt_timer_callback_t)(void);

int   blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us);
```

“func”: timer task function.

“interval_us”: timing value (unit: us).

The int-type return value correspons to three processing methods:

1) If the return value is less than 0, this executed task will be automatically deleted.
2) If the return value is 0, the old interval_us will be used as timing cycle.
3) If the return value is more than 0, this return value will be used as the new timing cycle (unit: us).



```
int blt_soft_timer_add(blt_timer_callback_t func, u32 interval_us)
{
    int i;
    u32 now = clock_time();

    if(blt_timer.currentNum >= MAX_TIMER_NUM){   //timer full
        return  0;
    }
    else{
        blt_timer.timer[blt_timer.currentNum].cb = func;
        blt_timer.timer[blt_timer.currentNum].interval = interval_us * CLOCK_SYS_CLOCK_1US
        blt_timer.timer[blt_timer.currentNum].t = now + blt_timer.timer[blt_timer.currentN
        blt_timer.currentNum ++;

        blt_soft_timer_sort();
        return  1;
    }
}
```

As shown in the implementation code, if timer number exceeds the maximum value, the adding operation will fail. Whenever a new timer task is added, re-ordering must be implemented to ensure timer tasks are time-ordered, while the index corresponding to the nearest timer task should be 0.

9.4 Delete timer task

As introduced above, timer task will be automatically deleted when the return value is less than 0. Except for this case, the API below can be invoked to specify the timer task to be deleted.

```
int   blt_soft_timer_delete(blt_timer_callback_t func);
```

9.5 Demo

For Demo code of blt soft timer, please refer to "TEST_USER_BLT_SOFT_TIMER" in 826x feature.

```
int gpio_test0(void)
{
    DBG_CHN0_TOGGLE;        //gpio 0 toggle to see the effect
    return 0;
}


int gpio_test1(void)
{
    DBG_CHN1_TOGGLE;        //gpio 1 toggle to see the effect

    static u8 flg = 0;
    flg = !flg;
    if(flg){
        return 7000;
    }
    else{
        return 17000;
    }
}

int gpio_test2(void)
{
    DBG_CHN2_TOGGLE;        //gpio 2 toggle to see the effect
    //timer last for 5 second
    if(clock_time_exceed(0, 5000000)){
        //return  -1;
        blt_soft_timer_delete(&gpio_test2);
    }

    return 0;
}

int gpio_test3(void)
{
    //gpio 3 toggle to see the effect
    DBG_CHN3_TOGGLE;
    return 0;
```

```
        }
```

Initialization:

```
        blt_soft_timer_init();
        blt_soft_timer_add(&gpio_test0, 23000);
        blt_soft_timer_add(&gpio_test1, 7000);
        blt_soft_timer_add(&gpio_test2, 13000);
        blt_soft_timer_add(&gpio_test3, 27000);
```

Four tasks are defined with differenet features:

1) Toggle gpio_test0 once for every 23ms.
2) gpio_test1 uses 7ms/17ms toggle timer.
3) Delete gpio_test2 after 5s, which can be implemented by invoking "blt_soft_timer_delete(&gpio_test2)" or "return -1".
4) Toggle gpio_test3 once for every 27ms.

10 IR

10.1 PWM Driver

Pleaser refer to PWM secion in Telink 8266/8267 IC Datasheet to help understanding PWM driver.

Since PWM related hardware configuration basically implemented via operating registers is very simple, BLE SDK does not use specific driver file, but define operation interfaces in “register_8266.h/register_8267.h” which are implemented by using “static inline function”, so as to improve efficiency and save code size.

10.1.1 PWM id and pin

8266/8267 supports up to 12-channel PWM: PWM0 ~ PWM5 and PWM0_N ~ PWM5_N. Six-channel PWM is defined in driver:

```
typedef enum {
    PWM0_ID = 0,
    PWM1_ID,
    PWM2_ID,
    PWM3_ID,
    PWM4_ID,
    PWM5_ID,
}pwm_id;
```

Only six-channel PWM0~PWM5 are configured in software, while the other six-channel PWM0_N~PWM5_N is inverted output of PWM0~PWM5 waveform. For example: PWM0_N is inverted output of PWM0 waveform. When PWM0 is high level, PWM0_N is low level; When PWM0 is low level, PWM0_N is high level. Therefore, as long as PWM0~PWM5 are configured, PWM0_N~PWM5_N are configured.

For 8266, IC pins corresponding to 12-channel PWM are shown as below:

Pin	PWM	Pin	PWM
PC0	PWM0	PB7	PWM0_N
PC3	PWM1	PC1/PC2	PWM1_N
PC4	PWM2	PC5	PWM2_N
PA1/PD2	PWM3	PA4	PWM3_N
PA5/PD3	PWM4	PA6	PWM4_N
PB0	PWM5	PB1	PWM5_N

For 8267, IC pins corresponding to 12-channel PWM are shown as below:

Pin	PWM	Pin	PWM
PA0/PC0/PD5/PE0	PWM0	PA2	PWM0_N
PA3/PC1/PD6/PE1	PWM1	PA4	PWM1_N
PB0/PC2/PD7	PWM2	PA5/PB1	PWM2_N
PB2/PC3	PWM3	PB3	PWM3_N
PB4/PC4	PWM4	PB5	PWM4_N
PB6/PC5	PWM5	PB7	PWM5_N

The “void gpio_set_func(u32 pin, u32 func)” serves to set specific pin as PWM function. E.g. To use PA0 of 8267 as PWM0:

gpio_set_func(GPIO_PA0,   AS_PWM)

10.1.2 PWM clock

The “pwm_set_clk(int system_clock_hz, int pwm_clk)” serves to set PWM clock.

✧ “system_clock_hz”: current system clock CLOCK_SYS_CLOCK_HZ.

✧ “pwm_clk”: clock to be configured.

Note that “system_clock_hz” must be an integral multiple of “pwm_clk” so as to get the wanted clock via frequency division.

To increase accuracy of PWM time, it’s recommended to set “pwm_clk” as “system_clock_hz”, i.e.

pwm_set_clk(CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_HZ);

Both “system_clock_hz” and “pwm_clk” are tick value corresponding to system clock, i.e. system clock increased value per second.

10.1.3 PWM cycle and duty

After PWM clock is configured, it’s needed to set maximum cycle and duty cycle for each PWM waveform.

“static inline pwm_set_cycle(pwm_id id, u16 cycle_tick)” serves to set maximum cycle of specific PWM. “cycle_tick” unit is PWM clock tick value.

“static inline pwm_set_cmp(pwm_id id, u16 cmp_tick)” serves to set high level duration during PWM cycle. “cmp_tick” unit is PWM clock tick value.

Thus duty cycle equals cmp_tick/cycle_tick.

For PWM0~PWM5, by default hardware will set PWM output high level followed by low level during a frame cycle. To obtain PWM waveform with low level followed by high level, following two methods applys:

1. Use corresponding PWM0_N~PWM5_N (inverted output of PWM0 ~ PWM5).
2. Use “void pwm_revert(pwm_id id)” to invert PWM0~PWM5 waveform.

Suppose current PWM clock is 16MHz, to set PWM cycle and duty cycle for PWM0 as 1ms and 40%:

```
pwm_set_cycle(PWM0_ID , 16000);      //16*1000
pwm_set_cmp (PWM0_ID , 6400);        //16*400
```

“static inline pwm_set_cycle_and_duty(pwm_id id, u16 cycle_tick, u16 cmp_tick)” combines the two interfaces above, which can be used to improve configuration efficiency.

10.1.4 PWM revert

“static inline void pwm_revert(pwm_id id)” serves to invert PWM0~PWM5 waveform.

“static inline pwm_n_revert(pwm_id id)” serves to invert PWM0_N~PWM5_N waveform.

10.1.5 PWM mode

PWM supports up to three modes: PWM0~PWM5 support normal mode, while only PWM0~PWM1 support count mode and IR mode.

```
typedef enum{
    PWM_NORMAL_MODE = 0x00,
    PWM_COUNT_MODE  = 0x01,
    PWM_IR_MODE     = 0x03,
  }pwm_mode;
```

1. normol mode

   Normal PWM mode, PWM timing sequence with configured pwm_set_clk/ pwm_set_cycle_and_duty is called a frame. After a PWM is enabled via “pwm start”, it will continuously output frames until this PWM is disabled via “pwm stop”.

2. count mode

After a PWM frame is configure via "pwm_set_clk" /"pwm_set_cycle_and_duty", "pwm_set_pulse_num" is used to specify the frame number, i.e. the number of pulses consisiting output waveform. Suppose pulse number is n, after a PWM is enabled via "pwm start", it will continuously output n frames, and it will be stopped automatically without the need to use "pwm stop".

In this mode, after PWM is stopped automatically, it's needed to use "pwm start" to restart the PWM timing sequence. The new configuration of PWM cycle and duty cycle during PWM timing sequence will take effect in the next frame immediately. Suppose the initial PWM frame is set as 1ms cycle and 1/2 duty cycle, and frame number is set as 10; during the 6th frames, cycle and duty cycle are modified as 2ms and 1/3, then the new setting will take effect in the remaining four frames.

3. IR mode

Similar to count mode, pwm_set_clk/pwm cycle and dut/pwm_set_pulse_num are used to set timing sequence of a group of PWM frames (ir task, marked with ir_taskn or irn). After a PWM is enabled via "pwm start", hardware will continuously running this ir_task until this PWM is disabled via "pwm stop".

Following shows the differences between IR mode and count mode:

1) count mode will be stopped automatically, while IR mode won't.
2) In count mode, new configuration of cycle and duty cycle will take effect immediately in the next frame; while in IR mode, new setting won't take effect until the current ir_task is finished, i.e. it will take effect in the next new ir_task.

Following shows an example of IR mode timing sequence.

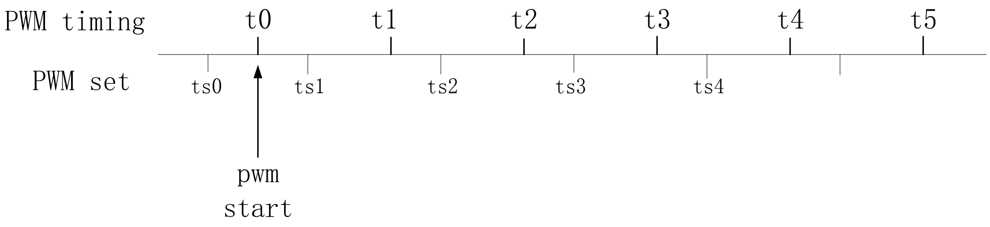


Figure10-1 PWM timing and PWM set

"ts0" indicates the moment to set pwm frame/pulse number for the first time.

Suppose ir_task0 is set herein.

t0 indicates the moment to start PWM IR mode via “pwm start”.

tn (t1, t2, t3……) indicates the moment when an IR task is finished and the next IR task is started.

tsn (ts1, ts2, ts3, ts4……) indicates the moment when user can set timing sequence of new IR task.

After “pwm start”, hardware runs ir_task0 from t0 to t1. Following shows several setting and corresponding running state:

1) After ir_task0 is set at ts0, PWM setting is not modified, i.e. no new IR task is set. Hardware will continuously run ir_task0 (t0~t1, t1~t2, t2~t3, t3~t4, t4~t5) until PWM is disabled via “pwm stop”.
2) Suppose ir_task0 is set at ts0, and ir_task1 is set at ts1. Hardware will run ir_task0 during t0~t1, and then continuously run ir_task1 (t1~t2, t2~t3, t3~t4, t4~t5) until PWM is disabled via “pwm stop”.
3) Suppose ir_task0 is set at ts0, ir_task1 is set at ts1, and ir_task2 is set at ts2. Hardware wil run ir_task0 during t0~t1, run ir_task1 during t1~t2, and then continuously run ir_task2 (t2~t3, t3~t4, t4~t5) until PWM is disabled via “pwm stop”.

10.1.6 PWM start and stop

The two interfaces below serve to enable (start)/disable (stop) certain PWM.

static inline void pwm_start(pwm_id id)

static inline void pwm_stop(pwm_id id)

10.1.7 PWM pulse number

“static inline void pwm_set_pulse_num(pwm_id id, u16 pulse_num)” serves to specify PWM frame number in PWM0~PWM1 count mode and IR mode.

This interface does not apply to PWM2~PWM5 which support normal mode only.

10.1.8 PWM phase

“static inline pwm_set_phase(pwm_id id, u16 phase)” serves to set delay time before PWM is started. Generally it can be set as 0 (no delay).

10.1.9 PWM interrupt

PWM supports eight types of interrupt: IRQ_PWMn_PNUM (n=0,1), IRQ_PWMn_FRAME (n=0,1,2,3,4,5).

```
typedef enum{
    FLD_IRQ_PWM0_PNUM =      BIT(0),
    FLD_IRQ_PWM1_PNUM =      BIT(1),
    FLD_IRQ_PWM0_FRAME =     BIT(2),
    FLD_IRQ_PWM1_FRAME =     BIT(3),
    FLD_IRQ_PWM2_FRAME =     BIT(4),
    FLD_IRQ_PWM3_FRAME =     BIT(5),
    FLD_IRQ_PWM4_FRAME =     BIT(6),
    FLD_IRQ_PWM5_FRAME =     BIT(7),
}PWM_IRQ;
```

Whenever a frame configured by pwm_set_clk/ pwm_set_cycle_and_duty is finished, PWMn will generate a frame-done IRQ (Interrupt Request) signal "FLD_IRQ_PWMn_FRAME". This applies to PWM0 ~ PWM5 normal mode/count mode/IR mode.

Whenever a frame group (it's called IR task in IR mode) consisting "pwm_set_pulse_num" frames is finished, PWMn will generate a Pnum IRQ signal "FLD_IRQ_PWMn_PNUM". This only applies to PWM0~PWM1 count mode/IR mode.

Suppose PWM mode is count mode, to set cycle as 1ms, duty cycle as 1/2, pulse number as 50:

pwm_set_clk(16000000, 16000000);

pwm_set_cycle_and_duty(PWM0_ID , 16000, 8000);

pwm_set_pulse_num(IR_PWM_ID, 50);

PWM0 will generate a frame done IRQ "FLD_IRQ_PWM0_FRAME" for every 1ms. After 50 frams are finished, PWM0 will generate a Pnum IRQ "FLD_IRQ_PWM0_PNUM". At this moment, totally "FLD_IRQ_PWM0_FRAME" have been generated for 50 times.

To enable PWM interrupt processing during irq_hander (interrupt entry in SDK), it's also needed to enable the mask of corresponding PWM interrupt.

Take IRQ_PWM0_PNUM as an example.

1. Enable mask of FLD_IRQ_PWM0_PNUM:

   reg_pwm_irq_mask |= FLD_IRQ_PWM0_PNUM;

   Generally it's recommended to clear the previous status before mask is enabled, so that it won't be triggered to enter irq_handler by previous status.

reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;

2. Enable mask of PWM interrupt in MCU system interrupt.

   reg_irq_mask |= FLD_IRQ_SW_PWM_EN;

3. Ensure MCU global interrupt is enabled, i.e. irq_enable ().

4. The settings above ensure this interrupt can trigger MCU to enter irq_handler(). Following is suggested processing in irq_handler().

```
u8 pwm_sta = reg_pwm_irq_sta;
if(pwm_sta & FLD_IRQ_PWM0_PNUM){
    func();
    reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;

}
```

10.2 IR implementation method

IR transmission needs to switch PWM output at specific time, to avoid IR error, the switch time has high requirement of accuracy.

As introduced in BLE Link Layer timing sequence (section 3.2.4), Link Layer uses system interrupt to process brx event (In newest SDK, processing of adv event is placed in mainloop, and it does not occupy system interrup time). When IR is going to switch PWM output soon, if brx event related interrupt comes first and occupies MCU time, the time to swtich PWM output may be delayed, thus to result in IR error.

To avoid the problem above, IR implementation uses PWM IR mode. As introduced in IR mode, BLE SDK will divides an IR data into multiple ir tasks: ir_task0, ir_task1, ir_task2……ir_task(n-1), ir_taskn.

First set ir_task0, after pwm_start, PWM outputs ir_task0, then set ir_task1 immediately; when ir_task0 is finished, “FLD_IRQ_PWMn_PNUM” is generated, then set ir_task2 in irq_handler(), at this moment PWM outputs ir_task1; when ir_task1 is finished, “FLD_IRQ_PWMn_PNUM” is generated, then set ir_task3 in irq_handler(), at this moment PWM outputs ir_task2……

The next IR task should be set when the previous IR task is output. As long as maximum time of MCU system interrupt does not exceed the shortest IR task time, it can ensure IR timing sequence won’t be delayed. In BLE SDK, maximum interrupt processing time in brx event is generally shorter than IR task.

10.3 IR Demo details

10.3.1 NEC IR

NEC IR protocol is shown as below:

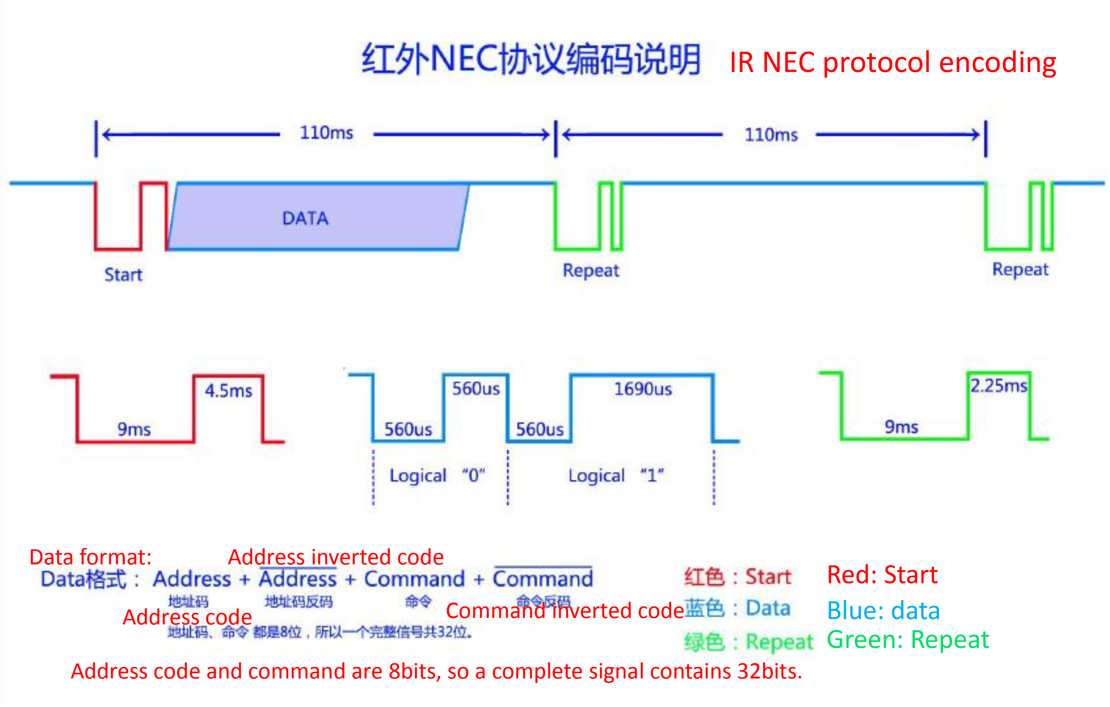


Figure10-2 NEC IR protocol

To implement NEC IR by using PWM IR mode, a complete timing sequence is divided into multiple IR tasks. Actually each IR task only contains three parameters: Cycle, duty cycle and pulse number, which can be configured via the two APIs below.

static inline pwm_set_cycle_and_duty(pwm_id id, u16 cycle_tick, u16 cmp_tick)

static inline void pwm_set_pulse_num(pwm_id id, u16 pulse_num)

Telink BLE SDK does not need user to divide each IR task, but supplies a set of unified processing mechanism with some configuration interfaces for user to automatically implement IR.

User only needs to understand this IR management mechanism, and configure his IR accordingly. Note that some interfaces are not modifiable and can only be invoked by user, and some functions may need user to modify as needed.

The core of this mechanism is a structure, as shown below.

```
typedef struct{
    ir_send_ctrl_data_t  data[IR_GROUP_MAX];
    u8 group_index;
    u8 group_cnt;
    u8 is_sending;
    u8 repeat_timer_enable;

    u8 ir_send_irq_idx;
    u8 ir_send_start_high;
    u8 last_cmd;
    u8 rsvd;

    u16 carrier_cycle;
    u16 carrier_high;
    u32 sending_start_time;
    u32 repeat_time;
    }ir_send_ctrl_t;
```

10.3.2 Set carrier

NEC IR carrier frequency is 38kHz, while duty cycle is 1/3.

User needs to invoke the interface below to set carrier cycle and duty cycle. This interface can only be invoked, and user cannot modify its internal implementation.

void ir_config_carrier(u16 cycle_tick, u16 high_tick)

When system clock is CLOCK_SYS_CLOCK_HZ,

cycle_tick = CLOCK_SYS_CLOCK_HZ/38000

Since duty cycle is 1/3, high_tick should be set as (cycle_tick * 1/3).

As shown in the code, carrier cycle and duty cycle configured by user are stored in the “carrier_cycle” and “carrier_high” variable.

10.3.3 Set logic1 and logic0 time

The interface below serves to set PWM duration of IR data bit 1 and bit 0. This interface can only be invoked, and user cannot modify its internal implementation.

void ir_config_byte_timing(u32 tick_logic_1_carr, u32 tick_logic_1_none,
u32 tick_logic_0_carr, u32 tick_logic_0_none)

“tick_logic_1_carr” and “tick_logic_1_none” indicate carrier and low level duration of logic level 1.

“tick_logic_0_carr” and “tick_logic_0_none” indicate carrier and low level duration of logic level 0.

According to NEC IR protocol, the setting of the four values is shown as below:

```
ir_config_byte_timing(  560 * CLOCK_SYS_CLOCK_1US,
                        1690 * CLOCK_SYS_CLOCK_1US,
                        560 * CLOCK_SYS_CLOCK_1US,

                        560 * CLOCK_SYS_CLOCK_1US);
```

Each IR task only has three parameters: cycle, duty cycle and pulse number. The structure below is used to describe an IR task.

```
typedef struct{
    u32 cycle;
    u16 hich;
    u16 cnt;

}ir_ctrl_t;
```

Create two IR task array data for logic 1 and logic 0 with carrier and without carrier (i.e. low level), as shown below:

```
ir_ctrl_t  ir_bit_1_controll[2];

ir_ctrl_t  ir_bit_0_controll[2];
```

Actually the four parameters configured by “ir_config_byte_timing(…)” are stored in the two arrays above.

When IR sends a byte (e.g. 0x55, 8b’ 01010101), each bit 0 and bit 1 will be configured according to the three values of IR task pre-calculated by ir_bit_1_controll and ir_bit_0_controll. User only needs to write the data to be sent (0x55), and SDK will automatically disassemble this data into eight corresponding IR tasks.

As shown in the code of “ir_config_byte_timing(…)”, the timing sequence of carrier is determined by user-configured carrier parameters (cycle, high) and pulse number which is tick_logic_1_carr duration divided by carrier cycle.

```
ir_bit_1_controll[0].cycle = ir_send_ctrl.carrier_cycle;
ir_bit_1_controll[0].hich = ir_send_ctrl.carrier_high;
ir_bit_1_controll[0].cnt=( tick_logic_1_carr)/
                                ir_send_ctrl.carrier_cycle;
```

PWM timing sequence without carrier should also be transformed to an IR task: cycle is duration, duty cycle is 0, and pulse number is 1. Since large cycle may result in data overflow, user can correspondingly decrease cycle and increase pulse number, e.g. cycle is duration *1/2, duty cycle is 0, pulse number is 2.

```
ir_bit_1_controll[1].cycle = tick_logic_1_none;
ir_bit_1_controll[1].hich = 0;
ir_bit_1_controll[1].cnt = 1;
```

10.3.4 Configure a complete NEC IR

According to NEC IR protocol, to sent a cmd (e.g. 7), it's needed to send start (9ms carrier + 4.5ms low level), followed by "address+ ~address+ cmd + ~cmd". In demo code, address is set as 0x88.

When the final bit of the final "~cmd" is sent, whether it's bit 0 or bit 1, a duration of low level (without carrier) is needed; if "~cmd" is not followed by any data, there may bring a problem on Rx side: Since no boundary of carrier is used for differentiation, user cannot know whether the low level duration of the final bit is 560us or 190us, thus cannot identify whether the data is 0 or 1. To solve this problem, a customized carrier with duration of 563us is used as end.

Another problem is: In PWM IR mode, the timing sequence of the next IR task should be configured in the previous IR task. As shown in the final three IR tasks of NEC IR, "ir_task n-2" and "ir_task n-1" correspond to carrier and low level duration of the final bit, while "ir_task n" corresponds to the customized 563us end carrier duration.

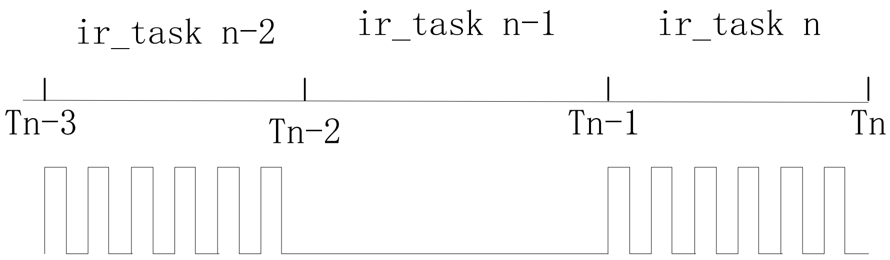


Figure10-3 IR ending

Actually in software, "ir_task n-1" is set at "Tn-3" (FLD_IRQ_PWM0_PNUM interrupt after "ir_task n-3" is finished), "ir_task n" is set at "Tn-2" (FLD_IRQ_PWM0_PNUM interrupt after "ir_task n-2" is finished), while at "Tn-1" (FLD_IRQ_PWM0_PNUM interrupt after "ir_task n-1" is finished), no IR task is set. At "Tn", if FLD_IRQ_PWM0_PNUM interrupt after "ir_task n" is finished can be responded immediately, "pwm stop" can be used to disable PWM; if this interrupt is delayed by BLE interrupt and cannot be responded immediately, MCU will start a new "ir_task n", and the carrier duration may be a unexpected value other than the customized duration 563us (e.g. 600us). If special format (e.g. IR low level followed by carrier) results in the final IR task with carrier, it's not needed to add a customized carrier as differentiation, this case may cause error.

To solve the problem above, we make an appointment: As long as an IR starts, no matter whether the final IR task is customized, an IR task of low level (without carrier) should be added as the eventual end, and the duration is configurable. E.g. add an IR task of 500us low level as end, i.e. "ir_task n+1" in Figure10-4, "ir_task n+1" can be set at "Tn-1", at "Tn", even if FLD_IRQ_PWM0_PNUM interrupt is not responded in time, PWM will send a low level singal and timing sequence won't be influenced. User only needs to use "pwm stop" to disable PWM in FLD_IRQ_PWM0_PNUM interrupt.

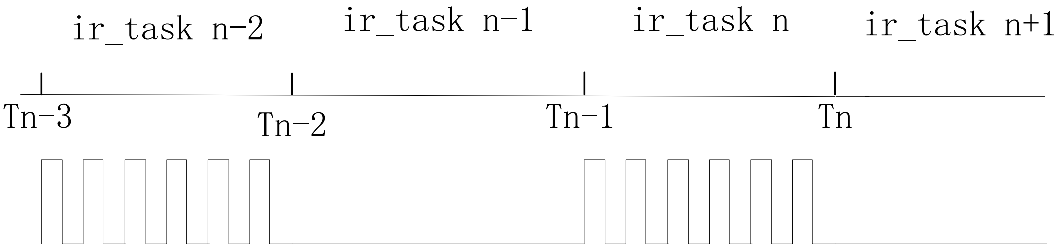


Figure10-4 Add low level ir task as IR end

According to the description above, an NEC IR data mainly contains three parts:

1. start signal, 9ms carrier + 4.5ms low level (without carrier)
2. valid data: address+ ~address+ cmd + ~cmd
3. stop signal, customized 563us carrier + 560us low level as end

10.3.5 Add timing sequence signal

In NEC IR, start and stop signal are timing sequence signal, which can be added via the interface below. This interface can only be invoked, and user cannot modify its internal implementation.

void ir_send_add_series_item(u32 *time_series, u8 series_cnt, ir_ctrl_t *ir_control, u8 start_high)

"time_series" and "series_cnt" are description of timing sequence. User should define the two parameters as const variable (store in flash) or local variable, so as to save RAM space.

NEC IR start signal is:

```
const u32 ir_lead_times[] = {   9000 * CLOCK_SYS_CLOCK_1US,
                                4500 * CLOCK_SYS_CLOCK_1US};
```

"ir_control" is ir_ctrl_t structure used to store signals including cycle/duty/pulse number after corresponding timing sequence is transformed to IR tasks, and user should define it as global variable:

ir_ctrl_t nec_start[ARRAY_SIZE(ir_lead_times)];

"start_high" indicates the order of carrier and low level during IR transmission. 1-carrier first, followed by low level; 0-low level first, followed by carrier.

Configure NEC IR start signal as IR task:

ir_send_add_series_item(ir_lead_times, ARRAY_SIZE(ir_lead_times), &nec_start, 1);

The configuration of NEC IR stop signal is the same as start signal.

10.3.6 Add data

The interface below serves to add data in unit of byte. This interface can only be invoked, and user cannot modify its internal implementation.

void ir_send_add_byte_item(u8 code, u8 start_high)

“code”: data.

“start_high” indicates the order of carrier and low level during IR transmission. 1-carrier first, followed by low level; 0-low level first, followed by carrier.

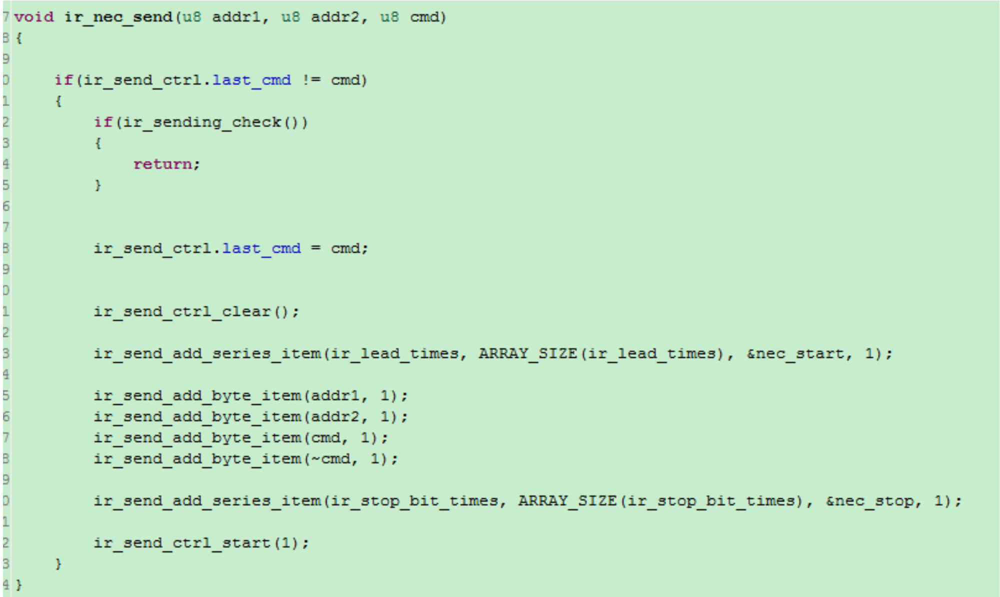
NEC IR data is: address+ ~address+ cmd + ~cmd

If address is 0x88, and cmd is 7:

```
ir_send_add_byte_item(0x88, 1);
ir_send_add_byte_item(~0x88, 1);
ir_send_add_byte_item(0x07, 1);

ir_send_add_byte_item(~0x07, 1);
```

10.3.7 NEC IR send

The figure below shows the implementation reference of “ir_nec_send”.



```
void ir_nec_send(u8 addr1, u8 addr2, u8 cmd)
{

    if(ir_send_ctrl.last_cmd != cmd)
    {
        if(ir_sending_check())
        {
            return;
        }


        ir_send_ctrl.last_cmd = cmd;


        ir_send_ctrl_clear();

        ir_send_add_series_item(ir_lead_times, ARRAY_SIZE(ir_lead_times), &nec_start, 1);

        ir_send_add_byte_item(addr1, 1);
        ir_send_add_byte_item(addr2, 1);
        ir_send_add_byte_item(cmd, 1);
        ir_send_add_byte_item(~cmd, 1);

        ir_send_add_series_item(ir_stop_bit_times, ARRAY_SIZE(ir_stop_bit_times), &nec_stop, 1);

        ir_send_ctrl_start(1);
    }
}
```

Figure10-5 ir_nec_send

User needs to invoke interface to configure and modify as needed, so as to realize his own IR send function.

“int ir_sending_check(void)” checks whether PWM still sends previous IR. If yes, new IR is not processed. This interface can only be invoked, and user cannot modify its internal implementation.

“void ir_send_ctrl_clear(void)” must be invoked before a new IR is configured. This interface can only be invoked, and user cannot modify its internal implementation.

After timing sequence and data configuration are finished (see 10.3.5 and 10.3.6), the interface below is invoked to start an IR transmission. This interface can only be invoked, and user cannot modify its internal implementation.

void ir_send_ctrl_start(int need_repeat)

“need_start” indicates whether repeat signal is needed. 1-need repeat signal.

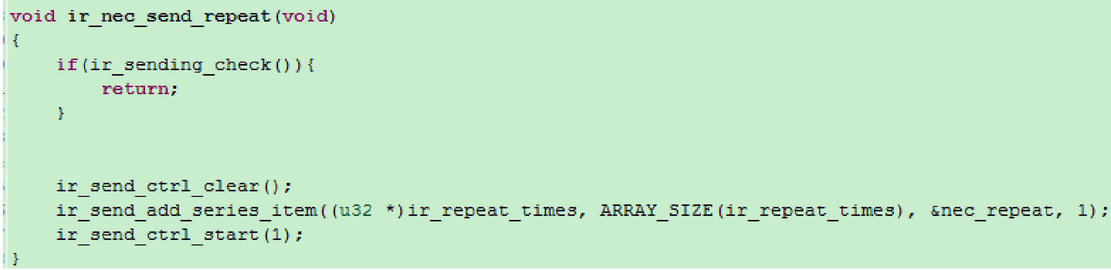
10.3.8 NEC IR repeat

Repeat signal will be enabled by “ir_send_ctrl_start (1)”. The configuration and transmission of repeat signal is similar to IR data signal.

According to NEC protocal, repeat signal is “9ms carrier + 2.25ms low level (without carrier) + 560us carrier”, with 500us end signal, the eventual repeat signal is defined as below:

```
const u32 ir_repeat_times[]   = {   9000 * CLOCK_SYS_CLOCK_1US,
                                    2250 * CLOCK_SYS_CLOCK_1US,
                                     560 * CLOCK_SYS_CLOCK_1US,

                                     500 * CLOCK_SYS_CLOCK_1US};
```

The analysis of the ir_nec_send_repeat is similar to ir_nec_send function. User needs to invoke interface to configure and modify as needed, so as to realize his own IR send repeat function.



```
void ir_nec_send_repeat(void)
{
    if(ir_sending_check()){
        return;
    }


    ir_send_ctrl_clear();
    ir_send_add_series_item((u32 *)ir_repeat_times, ARRAY_SIZE(ir_repeat_times), &nec_repeat, 1);
    ir_send_ctrl_start(1);
}
```

Figure10-6 ir_nec_send_repeat

NEC IR repeat signal will be sent with interval of 110ms. In SDK, hardware Timer2 is used to implement 108ms timer. Whenever the timeout expires, one repeat signal transmission is added. When button is released to finish IR, Timer2 should be disabled.

10.3.9 Interrupt processing

It’s not needed to modify Timer2 and FLD_IRQ_PWMn_PNUM interrupt processing in “irq_handler” of demo code.

For FLD_IRQ_PWMn_PNUM processing, user needs to note whether PWM0 or PWM1 is used, which is defined by the following macros in code:

```
#define PWM0_IR_MODE                   0
#define PWM1_IR_MODE                   1




#define IR_PWM_SELECT                       PWM0_IR_MODE
```

“void ir_irq_send(void)” serves to get ir_task0, ir_task1……ir_taskn successively according to IR signal and data configuration. PWM will be automatically disabled after all IR tasks are finished. This function is not modifiable.

“void ir_repeat_handle(void)” serves to restart an IR repeat signal with 108ms interval.

10.3.10 APP layer checks IR busy status

User can check the two variables below, so as to determine whether current IR is busy with transmission of data or repeat signal:

ir_send_ctrl.is_sending

ir_send_ctrl.repeat_timer_enable

Following shows the demo code in PM management to check IR busy status. When IR is busy with transmission of data or repeat signal, MCU cannot enter suspend.

```
if( ir_send_ctrl.is_sending ||ir_send_ctrl.repeat_timer_enable)
{
    bls_pm_setSuspendMask(SUSPEND_DISABLE);

}
```

11 Drivers in BLE SDK

11.1 External capacitor for 12/16 MHz crystal

By default, SDK uses internal capacitor of 826x MCU (i.e. cap corresponding to ana_81<4:0>) as matching capacitor of 12MHz/16MHz crystal oscillator, which is measurable and adjustable in Telink jig system to reach optimal frequency point value of final application product.

If it's needed to use external soldered capacitor as matching capacitor of 12MHz/16MHz crystal oscillator instead, the API below should be invoked at the beginning of main function and before "cpu_wakeup_init" function.

```
static inline void blc_app_setExternalCrystalCapEnable(u8  en)
{
    blt_miscParam.ext_cap_en = en;

}
```

As long as this API is invoked before "cpu_wakeup_init", SDK will automatically implement all operations (e.g. disable internal matching capacitor and stop reading frequency offset calibration value).

11.2 External 32kHz crystal oscillator

By default SDK uses internal 32kHz crystal, i.e. 32kHz RC. The maximum error of this capacitor is 500ppm, so its accuracy will be influenced for application with long suspend time. Currently 32kHz RC supports up to 1.5s connection by default. Once connection time exceeds this duration, inaccurate packet Rx time will be caused by BLE timing error; this case usually needs packet Rx/Tx retry, thus to increase power consumption and result in disconnection.

To ensure time accuracy for long suspend applications, external 32kHz crystal (i.e. 32kHz pad) should be used instead. Currently SDK supports this mode.

Either of the two APIs below should be invoked at the beginning of main function and before "cpu_wakeup_init" function, so as to select 32kHz RC (default) or 32kHz pad.

```
void blc_pm_select_internal_32k_crystal(void);

void blc_pm_select_external_32k_crystal(void);
```

Note: Currently this function is not supported by 8261, completely supported by 8266, and only supported by 8267/8269 above A2/A1 respectively (A2 and A1 indicate hardware version). User must check with Telink to avoid failure to use 32kHz pad caused by usage of incorrect IC.

11.3 PA

To use RF PA, please refer to “proj/drivers/rf_pa.c” and “rf_pa.h”.

First enable the macro below, which is disabled by default.

```
#ifndef PA_ENABLE
#define PA_ENABLE                       0

#endif
```

Invoke PA initialization during system initialization.

```
void rf_pa_init(void);
```

In this initialization, “PA_TXEN_PIN” and “PA_RXEN_PIN” are set as GPIO output mode, and initial status is “output 0”. User needs to define GPIOs corresponding to TX and RX PA.

```
#ifndef PA_TXEN_PIN
#define PA_TXEN_PIN                     GPIO_PB2
#endif

#ifndef PA_RXEN_PIN
#define PA_RXEN_PIN                     GPIO_PB3

#endif
```

And “void app_rf_pa_handler(int type)” is registered as callback function of PA. Acutally this function processes the three PA status below: disable PA, enable TX PA, and enable RX PA.

```
#define PA_TYPE_OFF                     0
#define PA_TYPE_TX_ON                   1
#define PA_TYPE_RX_ON                   2
```

User only needs to invoke the “rf_pa_init” above; “app_rf_pa_handler” is registered as the bottom-layer callback, so that it will be automatically invoked to process correspondingly in various BLE states.

11.4 PWM

For illustration of PWM driver, please refer to section 10 IR.

11.5 UART

11.5.1 UART GPIO

Telink 8261/8266/8267/8269 embeds an UART module: 8261/8267/8269 supports three groups of GPIOs for UART, while 8266 supports one group of GPIOs for UART, as shown in the table below.

UART GPIO mapping

IC type	UART ID	GPIO Pin	UART Pin
8261/8267/8269	1	PA6	Tx
		PA7	Rx
	2	PB2	Tx
		PB3	Rx
	3	PC2	Tx
		PC3	Rx
		PC4	RTS
		PC5	CTS
8266	1	PC6	Tx
		PC7	Rx
		PD0	RTS
		PD1	CTS

To use the internal UART module, first it’s needed to configure GPIO pins for UART. Telink GPIO pins support multiplexed functions, and default function of most pins is GPIO function. SDK supplies GPIO configuration function for user to use other multiplexed functions. The function prototype is shown as below:

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

Parameter	Description
pin	GPIO pin to be set
func	Function of the specific GPIO pin to be set

E.g.

```
gpio_set_func(GPIO_PA6,AS_UART);
gpio_set_func(GPIO_PA7,AS_UART);
```

To simplify the usage of UART, macros are defined in “uart.h” of SDK for GPIO configuration of all serial pins. For example:

```
#define    UART_GPIO_CFG_PA6_PA7()
#define    UART_GPIO_CFG_PB2_PB3()
#define    UART_GPIO_CFG_PC2_PC3()
#define    UART_GPIO_CFG_PC6_PC7()
```

User only needs to invoke corresponding macro to implement UART GPIO configuration.

11.5.2 UART configuration

11.5.2.1 UART common configuration

UART module of Telink 826x MCU is basically the same. Common UART configuration parameters include baudrate, data bit, parity check bit, stop bit, and etc. UART initialization function can be invoked to configure the common parameters above. The function prototype is shown as below:

```
unsigned char uart_Init( unsigned short uartCLKdiv,
                         unsigned char bwpc,
                         UART_ParityTypeDef Parity,
                         UART_StopBitTypeDef StopBit)
```

Parameter	Description
uartCLKdiv	The two parameters serve to determine Baudrate (See IC Datasheet). Parameter values corresponding to common Baudrates will be listed below.
bwpc	
Parity	Set parity check bit (enum-type value).
StopBit	Set stop bit (enum-type value).

E.g.

```
uart_Init（9,13,PARITY_NONE,STOP_BIT_ONE）;
```

Enum type definition for parity check bit:

```
typedef enum {
    PARITY_NONE = 0,
    PARITY_EVEN,
    PARITY_ODD,
} UART_ParityTypeDef;
```

Enum type definition for stop bit:

```
typedef enum{
    STOP_BIT_ONE = 0,
    STOP_BIT_ONE_DOT_FIVE = BIT(12),
    STOP_BIT_TWO = BIT(13),
} UART_StopBitTypeDef;
```

The table below shows “clk_div” and “bwpc” parameter values corresponding to common baudrates.

Common Baudrates

Baudrate	clk_div		bwpc	
	sys_clk 16MHz	sys_clk 32MHz	sys_clk 16MHz	sys_clk 32MHz
4800	302	605	10	10
9600	118	302	13	10
19200	118	118	6	13
38400	25	118	15	6
57600	30	36	8	14
115200	9	30	13	8

User can use Telink tool “TScript” to obtain “clk_div” and “bwpc” parameter values corresponding to other baudrates. Please refer to section 11.5.5 for the usage of “TScript”.

11.5.2.2 UART proprietary configuration

Except for UART common confirmation as introduced in section 11.5.2.1, some UART proprietary configurations are needed. Telink UART module supports two working modes: DMA mode, and non-DMA mode (Normal mode). Via proprietary configuration, UART can work in either of the two modes.

1) Proprietary configuration for UART DMA mode

When UART works in DMA mode, it’s needed to configure DMA channel of UART via DMA initialization funcation. The function prototype is shown as below:

```
void uart_DmaModeInit(   unsigned char dmaTxIrqEn,
                          unsigned char dmaRxIrqEn)
```

Parameter	Description
dmaTxIrqEn	UART Tx interrupt enable (1)/disable (0)
dmaRxIrqEn	UART Rx interrupt enable (1)/disable (0)

E.g.

```
uart_DmaModeInit(1,0);
```

In UART DMA mode, user also needs to configure a Rx buffer via "uart_RecBufInit()" function, so that DMA can store received UART data in this buffer. The function prototype is shown as below:

```
void uart_RecBuffInit(   unsigned char *recAddr,
                          unsigned short recBuffLen)
```

Parameter	Description
recAddr	Pointer pointing to Rx buffer
recBuffLen	Rx buffer length

E.g.

```
unsigned char rxBuf[100];
uart_RecBuffInit(rxBuf,100);
```

2) Proprietary configuration for UART Normal mode

In Normal (Non-DMA) mode, user needs to invoke "uart_Init()" function to initialize UART, and invoke "uart_notDmaModeInit()" function to configure UART. The function prototype of "uart_notDmaModeInit()" is shown as below:

```
void uart_notDmaModeInit(   unsigned char rx_level,
                             unsigned char tx_level,
                             unsigned char rx_irq_en,
                             unsigned char tx_irq_en)
```

Parameter	Description
rx_level	Set the number of received data to generate UART Rx IRQ, i.e. UART Rx interrupt will be generated after "rx_level" data are received (if Rx interrupt is enabled). Maximum value is 8.
tx_level	Set the number of transmitted data to generate UART Tx IRQ, i.e. UART Tx interrupt will be generated after "tx_level" data are transmitted (if Tx interrupt is enabled). Maximum value is 8.
rx_irq_en	UART Rx interrupt enable (1)/disable (0).
tx_irq_en	UART Tx interrupt enable (1)/disable (0).

Note: To implement UART data transmission, it's recommended to adopt inquiry of "Tx done flag" used in current UART driver, rather than interrupt method. If user does need to adopt interrupt method, please contact Telink for support.

E.g.

```
uart_notDmaModeInit(1,0,1,0);
```

SDK also supplies some macro functions in "uart.h" for user to implement UART configurations (only supply UART macro functions of DMA mode). Current supported macro sunctions are shown as below:

```
#define CLK32M_UART9600  //sys_clk = 32MHz,Baudrate = 9600
#define CLK32M_UART115200//sys_clk=32MHz,Baudrate = 115200
#define CLK16M_UART115200//sys_clk=16MHz,Baudrate = 115200
#define CLK16M_UART9600  //sys_clk = 16MHz,Baudrate = 9600
```

11.5.3 UART Data Rx/Tx in DMA mode

UART adopts interrupt method to implement data reception.

Note: As introduced above, UART configurations of DMA mode only enable DMA interrupt related to UART. To enable CPU to detect UART interrupt, global interrupt must be enabled by invoking "irq_enable()" function.

11.5.3.1 UART data Rx in DMA mode

In DMA mode, UART data reception does not need intervention of CPU, while DMA module will automatically store the received UART data into the Rx buffer speicified during UART initialization.

When data reception is finished, a Rx interrupt will be generated to inform MCU the finish of data reception. Then user can read data from Rx buffer. The “uart_IRQSourceGet()” function can be invoked to get IRQ source. The function prototype is shown as below:

```
enum UARTIRQSOURCE uart_IRQSourceGet(void)
```

This function will return an enum-type value which specifies interrupt type. Its definition is shown as below:

```
enum UARTIRQSOURCE
{
    UARTNONEIRQ = 0,
    UARTRXIRQ = BIT(0),
    UARTTXIRQ = BIT(1),
};
```

After UART interrupt is generated, user does not need to clear IRQ flag, i.e. it will be automcatically cleared by hardware.

11.5.3.2 UART data Tx in DMA mode

User can invoke “uart_Send ()” function to send data. The function prototype is shown as below:

```
unsigned char uart_Send(unsigned char* addr)
```

Parameter	Description
addr	Pointing to user data buffer
Return value	0: DMA busy; 1: Tx success

E.g.

```
unsigned char txBuf[] = {0x02,0x00,0x00,0x00,0xAA,0xBB};
uart_Send(txBuf);
```

Notes:

1) For UART data Tx in DMA mode, the Tx buffer stores 4 bytes (lower byte first) to indicate the length of data to be transmitted, which are followed by actual data. Following shows the format to send data:

0	1	2	3	4	5	6	…
dataLen	0x00	0x00	0x00	data0	data1	data2	…

2) In DMA mode, one UART transmission can send up to 507-byte (512-5) data, therefore, long data with length exceeding 507 bytes should be disassembled into several transmissions.

11.5.4 UART Data Rx/Tx in Non-DMA mode

UART adopts interrupt method to implement data reception.

Note: As introduced above, UART configurations of Non-DMA mode only enable interrupt related to UART. To enable CPU to detect UART interrupt, global interrupt must be enabled by invoking "irq_enable()" function.

11.5.4.1 UART data Rx in Non-DMA mode

In Non-DMA mode, UART data reception is implemented in interrupt. When serial port receives data, a Rx interrupt will be generated. User can get Rx IRQ by invoking "GET_UART_NOT_DMA_IRQ()" function which is given in the form of macro definition (Return value: 1 - data are received; 0 - no data is received). Then the received data can be obtained by invoking "uart_notDmaModeRevData()" function, the prototype of which is shown as below:

```
unsigned char uart_notDmaModeRevData(void)
```

Parameter	Description
Return value	Return one-byte received data. Multiple-byte data can be obtained via multiple invoking.

Note: The generation of Rx interrupt depends on the setting of "rx_level" in "uart_notDmaModeInit ()". E.g. If rx_level = 1, UART Rx interrupt will be generated after one byte data is received; if rx_level = 2, UART Rx interrupt will be generated after two-byte data is received……

11.5.4.2 UART data Tx in Non-DMA mode

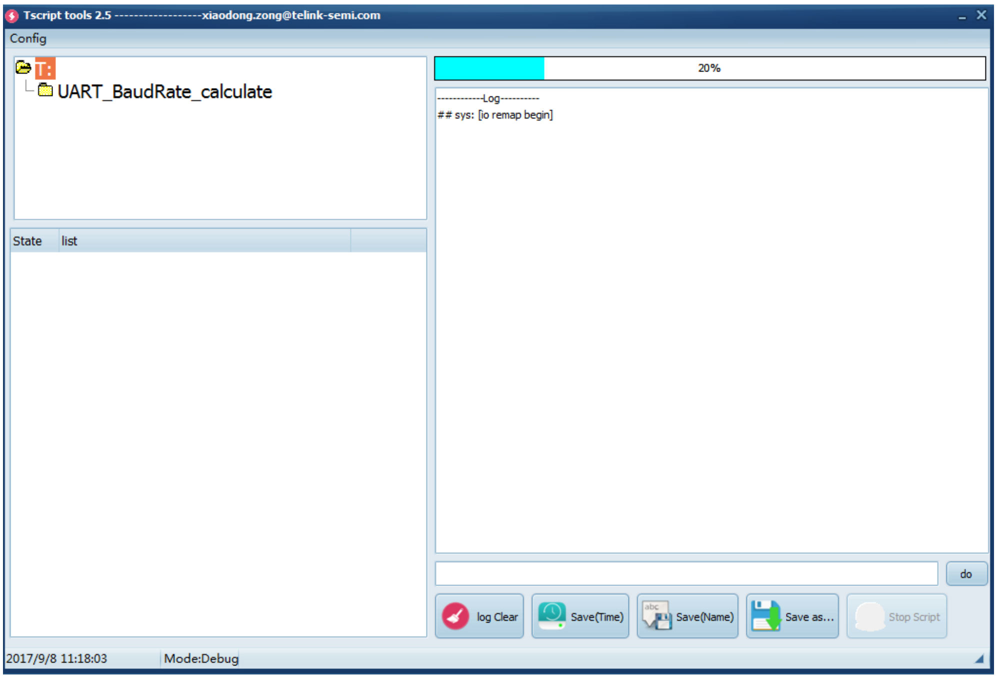
User can send data by invoking "uart_notDmaModeSendByte()”, the prototypeof which is shown as below:

unsigned char uart_notDmaModeSendByte(unsigned char uartData)

Parameter	Description
uartData	Data to be sent
Return value	Not used

11.5.5 UART baudrate calculation tool

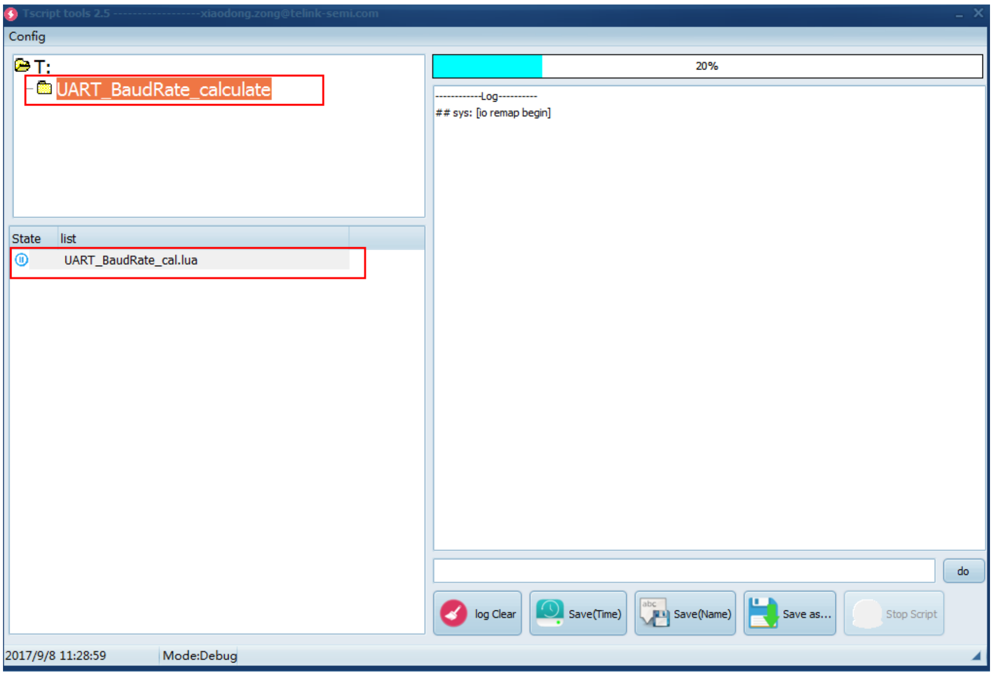
1) Open software tool “TScript”, the interface of which is shown as below.



```
------------Log----------
## sys: [io remap begin]
```

Figure11-1 Tscript initial interface

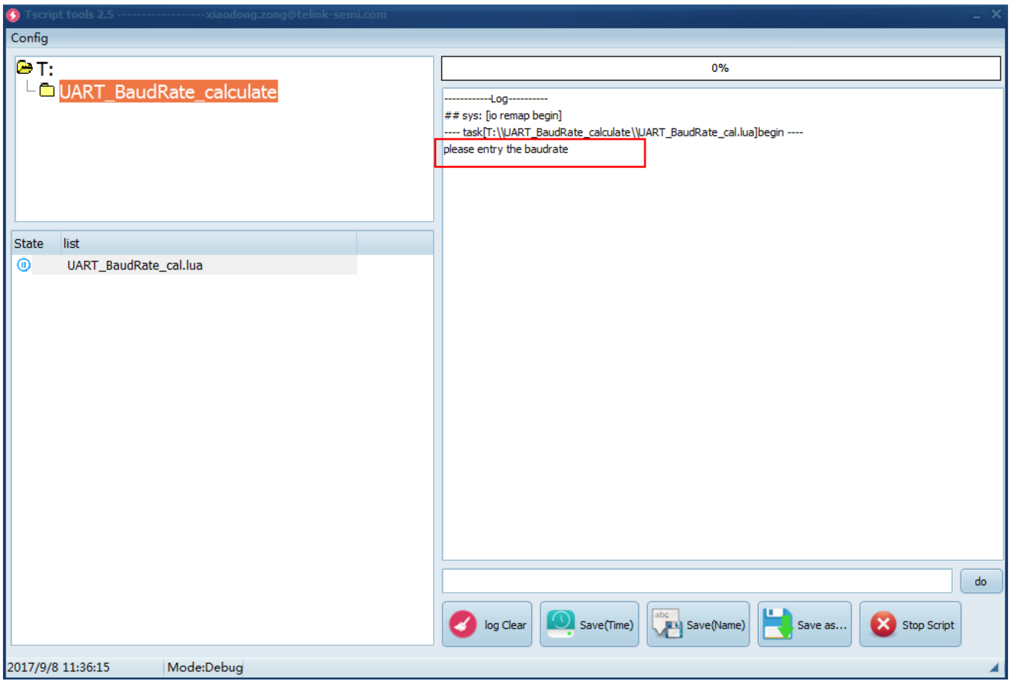
2) Click “UART_Baudrate_calculate” icon at the top-left corner, the window at the lower left corner will show “UART_BaudRate_cal.lua”, as shown below.



```
-----------Log----------
## sys: [io remap begin]
```

Figure11-2 UART_BaudRate_cal.lua

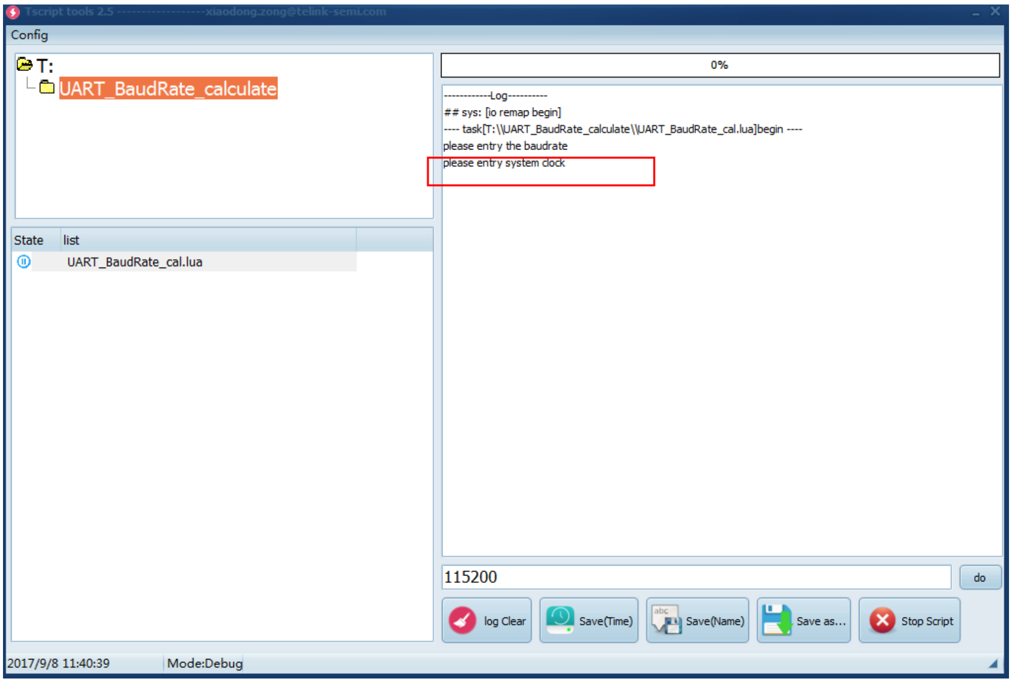
3) Dobule click “UART_BaudRate_cal.lua”, the right log window will show “please entry the baudrate”, as shown below.



```
-----------Log----------
## sys: [io remap begin]
---- task[T:\\UART_BaudRate_calculate\\UART_BaudRate_cal.lua]begin ----
please entry the baudrate
```

Figure11-3 Input baudrate

4) Input the wanted baudrate in the text box, and then click "Enter". The log window will show "please entry system clock", as shown below.

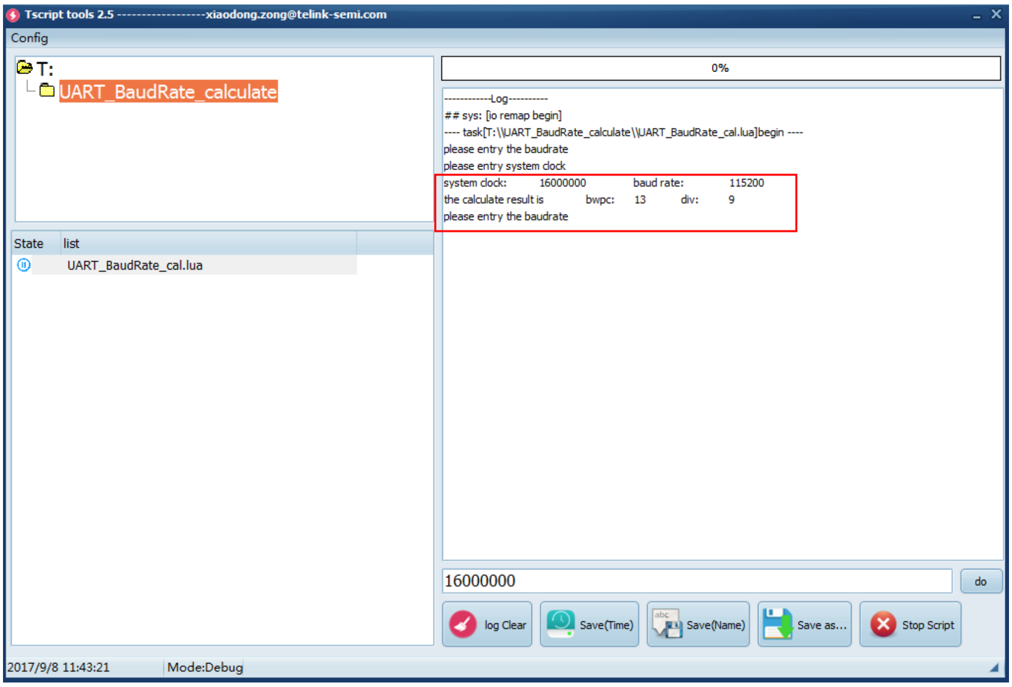


```
------------Log----------
## sys: [io remap begin]
---- task[T:\\UART_BaudRate_calculate\\UART_BaudRate_cal.lua]begin ----
please entry the baudrate
please entry system clock
```

115200

Figure11-4 Input system clock

5) Input system clock, and then click "Enter". The log window will show values of "clk_div" and "bwpc" corresponding to the specified baudrate, as shown below.



```
------------Log----------
## sys: [io remap begin]
---- task[T:\\UART_BaudRate_calculate\\UART_BaudRate_cal.lua]begin ----
please entry the baudrate
please entry system clock
system clock:        16000000          baud rate:          115200
the calculate result is          bwpc:     13          div:        9
please entry the baudrate
```

16000000

Figure11-5 Get "clk_div" and "bwpc" result

11.6 ADC

11.6.1 ADC Clock

Working clock of ADC module is derived from FHS (High speed clock) via frequency division. The maximum working frequency of Telink ADC is 4MHz (please refer to IC Datasheet for configuration guide). To use ADC, ADC clock must be enabled. Since ADC clock has already been configured in ADC driver, user can directly use the configuration.

11.6.2 ADC configuration

Telink ADC supports seven resolution options, two working modes (single-end mode and differential mode), 12 single-end input channels and multip groups of differential input channels, 2 or 3 reference voltage options (8266: 1.3V, 3.3V; 8267: 3.3V, 1.428V, 1.224V), as well as detection of battery voltage and temperature (based on internal temperature sensor). User can invoke "adc_Init()" function to initialize ADC.

11.6.2.1 8261/8267/8269 ADC initializaiton

The prototype of 8261/8267/8269 ADC initialization function is shown as below:

```
void adc_Init(enum ADCCLOCK adc_clk,
              enum ADCINPUTCH chn,
              enum ADCINPUTMODE mode,
              enum ADCRFV ref_vol,
              enum ADCRESOLUTION resolution,
              enum ADCST sample_cycle)
```

Parameter	Description
adc_clk	Set ADC clock. It's an enum-type value, and can only be configured as *ADC_CLK_4M*.
chn	Set ADC channel. It's an enum-type value.
mode	Set ADC working mode. It's an enum-type value.
ref_vol	Set ADC reference voltage. It's an enum-type value.
resolution	Set ADC resolution. It's an enum-type value.
sample_cycle	Set ADC sampling time. It's an enum-type value.

E.g.

```
adc_Init(ADC_CLK_4M, B6, SINGLEEND, RV_AVDD, RES14, S_3);
```

Enum type definition for ADC clock:

```
enum ADCCLOCK {
        ADC_CLK_4M = 4,
        ADC_CLK_5M = 5,
};
```

Enum type definition for ADC input channel:

```
enum ADCINPUTCH{
    NOINPUT,
    C0,
    C1,
    C6,
    C7,
    B0,
    B1,
    B2,
    B3,
    B4,
    B5,
    B6,
    B7,
    PGAVOM,
    PGAVOP,
    TEMSENSORN,
    TEMSENSORP,
    AVSS,
    OTVDD,//1/3 voltage division detection
};
```

Enum type definition for ADC working mode:

```
enum ADCINPUTMODE{
    SINGLEEND,
    INVERTB_1,
    INVERTB_3,
    PGAVOPM,
};
```

Enum type definition for ADC reference voltage:

```
enum ADCRFV{
    RV_1P428,
    RV_AVDD,
    RV_1P224,
};
```

Enum type definition for ADC resolution:

```
enum ADCRESOLUTION{
    RES7,
    RES9,
    RES10,
    RES11,
    RES12,
    RES13,
    RES14,
};
```

Enum type definition for ADC sampling time:

```
enum ADCST{
    S_3,
    S_6,
    S_9,
        S_12,
        S_18,
        S_24,
        S_48,
        S_144,
    };
```

11.6.2.2 8266 ADC initializaiton

The prototype of 8266 ADC initialization function is shown as below:

```
    void adc_Init(ADC_CLK_t adc_clock,
                  ADC_INPUTCHN_t chn,
                  ADC_INPUTMODE_t mode,
                  ADC_REFVOL_t ref_vol,
                  ADC_RESOLUTION_t resolution,
                  ADC_SAMPCYC_t sample_cycle)
```

Parameter	Description
adc_clock	Set ADC clock. It's an enum-type value, and can only be configured as *ADC_CLK_4M*
chn	Set ADC channel. It's an enum-type value.
mode	Set ADC working mode. It's an enum-type value.
ref_vol	Set ADC reference voltage. It's an enum-type value.
resolution	Set ADC resolution. It's an enum-type value.
sample_cycle	Set ADC sampling time. It's an enum-type value.

E.g.

```
    adc_Init(ADC_CLK_4M, ADC_CHN_D2,SINGLEEND,ADC_REF_VOL_AVDD,
             ADC_SAMPLING_RES_14BIT, ADC_SAMPLING_CYCLE_6);
```

Enum type definition for ADC clock:

```
    typedef enum{
            ADC_CLK_4M = 4,
            ADC_CLK_5M = 5,
    } ADC_CLK_t;
```

Enum type definition for ADC input channel:

```
typedef enum{
    ADC_CHN_D0          = 0x01,
    ADC_CHN_D1          = 0x02,
    ADC_CHN_D2          = 0x03,
    ADC_CHN_D3          = 0x04,
    ADC_CHN_D4          = 0x05,
    ADC_CHN_D5          = 0x06,
    ADC_CHN_C2          = 0x07,
    ADC_CHN_C3          = 0x08,
    ADC_CHN_C4          = 0x09,
    ADC_CHN_C5          = 0x0a,
    ADC_CHN_C6          = 0x0b,
    ADC_CHN_C7          = 0x0c,
    ADC_CHN_PGA_R       = 0x0d,
    ADC_CHN_PGA_L       = 0x0e,
    ADC_CHN_TEMP_POS    = 0x0f,
    ADC_CHN_TEMP_NEG    = 0x10,
    ADC_CHN_VBUS        = 0x11,
    ADC_CHN_GND         = 0x12,
} ADC_INPUTCHN_t;
```

Enum type definition for ADC working mode:

```
typedef enum{
        SINGLEEND,
        INVERTD_5,
        INVERTC_3,
        CHN_PGA_L,
} ADC_INPUTMODE_t;
```

Enum type definition for ADC reference voltage:

```
typedef enum{
   ADC_REF_VOL_1V3  =0x00, //!< ADC Reference:1.3v
   ADC_REF_VOL_AVDD = 0x01, //!< ADC Reference:AVDD
} ADC_REFVOL_t;
```

Enum type definition for ADC resolution:

```
typedef enum{
      ADC_SAMPLING_RES_7BIT  =0,
      ADC_SAMPLING_RES_9BIT  =1,
      ADC_SAMPLING_RES_10BIT = 2,
      ADC_SAMPLING_RES_11BIT = 3,
      ADC_SAMPLING_RES_12BIT = 4,
      ADC_SAMPLING_RES_13BIT = 5,
      ADC_SAMPLING_RES_14BIT = 7,
} ADC_RESOLUTION_t;
```

Enum type definition for ADC sampling time:

```
typedef enum{
      ADC_SAMPLING_CYCLE_3    = 0,
      ADC_SAMPLING_CYCLE_6    = 1,
      ADC_SAMPLING_CYCLE_9    = 2,
      ADC_SAMPLING_CYCLE_12  =3,
      ADC_SAMPLING_CYCLE_18  =4,
      ADC_SAMPLING_CYCLE_24  =5,
      ADC_SAMPLING_CYCLE_48  =6,
      ADC_SAMPLING_CYCLE_144 = 7,
  } ADC_SAMPCYC_t;
```

11.6.3 Obtain ADC convertion value

ADC driver supplies the function of obtaining ADC convertion value for user to invoke. The prototype is shown as below:

```
unsigned short adc_SampleValueGet(void)
```

Parameter	Description
Return value	ADC conversion value

Actual voltage value should be calculated according to ADC conversion value obtained via “adc_SampleValueGet” function.

11.6.3.1.1 Calculate actual voltage value for 8261/8267/8269

In theory, the relationship between ADC conversion value and voltage value is linear, shown as the dotted red line in the figure below. However, there will be an offset for 8261/8267/8269 actually, shown as the solid black line in the figure below. At zero voltage, conversion value has a 128 or so offset relative to “0”; at full voltage, conversion value also has a 128 or so offset relative to “2^resolution -1”.

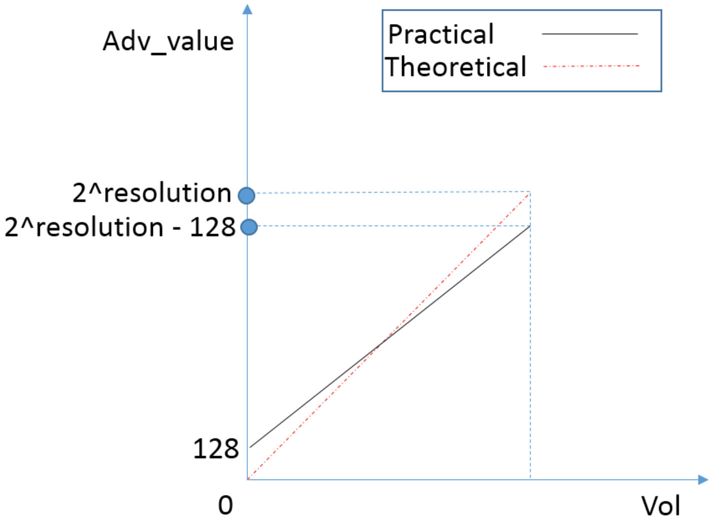


Figure11-6 ADC conversion curve

Telink supplies a calculation method to eliminate this offset.

Vol= V_{ref} * (adc_value - 128) / ($2^{resolution}$ - 256);

Parameter	Description
Vol	Actual voltage value
V_{ref}	ADC reference voltage
adc_value	ADC conversion value
resolution	ADC resolution

Note: Actually the offset value at zero voltage and full voltage is a slightly changed range, therefore, 128 is only a typical average value obtained from test of multiple ICs. Generally test voltage of actual application is not close to zero voltage and full voltage, the typical value can be used directly without problem; however, when test voltage is close to zero voltage, if 128 is directly used, it may cause data overflow. Actual application should not measure zero voltage (the actual error is large due to resolution problem).

11.6.3.1.2 Calculate actual value for 8266

Following shows the calculation method of actual voltage for 8266.

$$Vol = V_{ref} * (adc\_value) / (2^{resolution});$$

The parameters are the same as 8261/8267/8269.

Note: 8266 ADC supports two reference voltage options: AVDD (voltage of IC AVDD pin) or internal 1.3V (1.3V is used in firmaware instead of 1.428V given in Datasheet).

11.7 Low battery voltage detect

“Low battery check” function uses ADC function to check whether current voltage is lower than the preset normal threshold.

For applications with lithium battery or coin-cell battery, “Low battery check” function should be added. If 826x works in low level below normal threshold, chip working may become unstablized, which will bring unexpected risk, such as Flash write/erase error. If product upgrades its firmware via OTA at low voltage, the OTA process may fail, new firmware cannot be executed, which will cause product failure.

Once APP layer detects low level, the whole MCU must be cut off to stop all operations.

11.7.1 "Low battery check" implementation

11.7.1.1 "Low battery check" for 8266

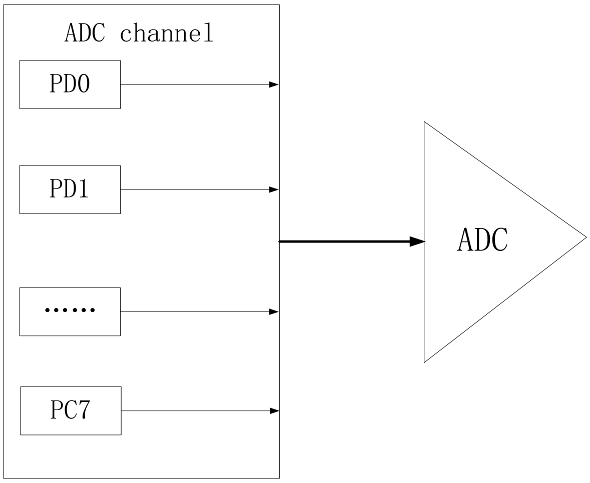


Figure11-7 8266 ADC channel

"Low battery check" for 8266 can only be implemented via ADC channel. Hardware chart for 8266 ADC channel is shown as above. "adc_Init" is used to initialize ADC configuration.

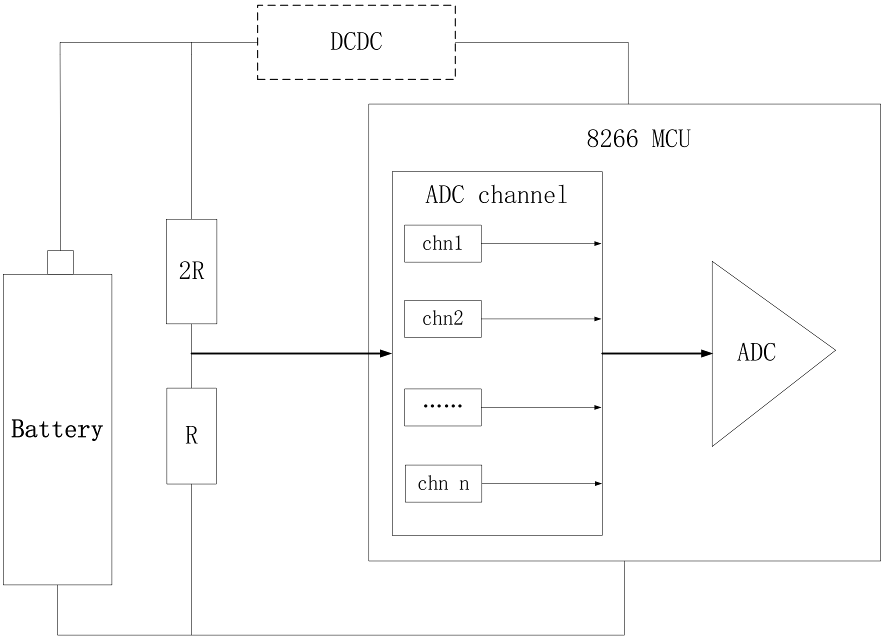


Figure11-8 Hardware chart for 8266 low battery check

Since the voltage to be checked is battery voltage, it cannot be used as reference voltage source, i.e. reference voltage can only be set as 1.3V voltage. In this case, measurable voltage cannot exceed 1.3V. To meet VCC test range, an external voltage division network should be used in hardware circuit. As shown in the figure above, 1/3 voltage division network is used to extend measurable voltage to 3.9V. User can measure voltage larger than 3.9V by using voltage division network such as 1/4, 1/5 and etc. An ADC channel should be selected to input the voltage after 1/3 voltage division of VCC.

If DCDC is used to convert battery voltage and supply power for MCU, the voltage to be measured should be the voltage before DCDC conversion, as shown in the figure above.

11.7.1.2 “Low battery check” for 8261/8267/8269

For 8261/8267/8269, ADC hardware module embeds a 1/3 voltage division network, which can be used for low battery check as well as ADC function extension. This internal voltage division network has two voltage sources: PB7, VCC (actual voltage at AVCC pin).

ADC input channel is selectable as GPIO pin with ADC function or the voltage obtained from 1/3 voltage division network.

Note: PB7 can be used as GPIO pin with ADC function and directly set as ADC input, or use PB7 as voltage source of 1/3 voltage division network and use the voltage from 1/3 voltage division network as ADC intput.

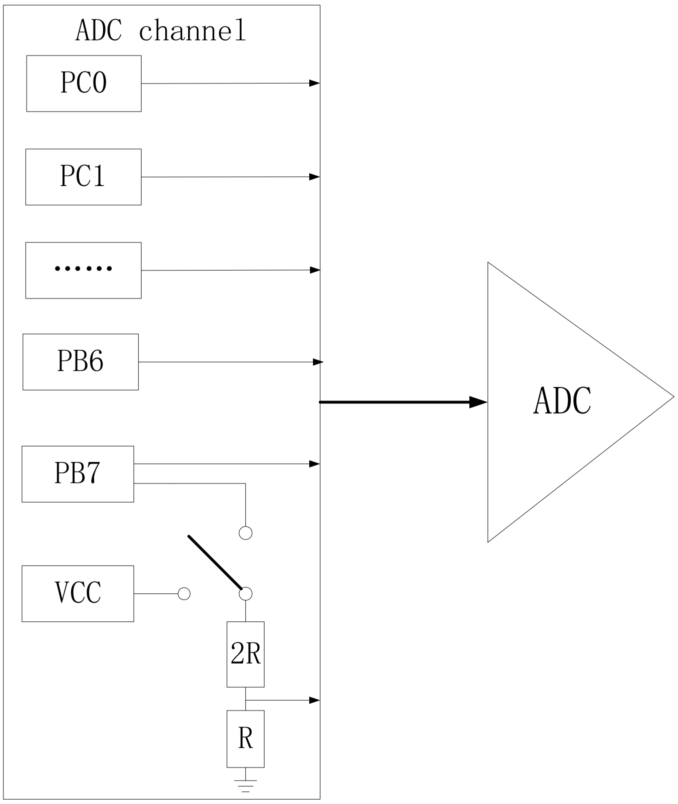


Figure11-9 8261/8267/8269 ADC channel

Since 8261/8267/8269 adds hardware 1/3 voltage division network for ADC input channel, SDK supplies battery voltage check API function “adc_BatteryCheckInit” to replace “adc_Init”, which adds the configuration to ADC channel of 1/3 voltage division network based on “adc_Init”. The protorype is shown as below:

```
void adc_BatteryCheckInit(enum ADCCLOCK adc_clk,
                          unsigned char div_en,
                          enum ONETHIRD_INPUTCHN oneThirdChn,
                          enum ADCINPUTCH notOneThirdChn,
                          enum ADCINPUTMODE mode,
                          enum ADCRFV ref_vol,
                          enum ADCRESOLUTION resolution,
                          enum ADCST sample_cycle)
```

Parameter	Description
adc_clk	Set ADC clock. It’s an enum-type value, which can be configured as *ADC_CLK_4M* or *ADC_CLK_5M*.
div_en	Set whether to select internal 1/3 voltage division network channel. 1: use; 0: not use.
oneThirdChn	If div_en = 1, this parameter is used to set voltage source for internal voltage division network as *Battery_Chn_VCC* or *Battery_Chn_B7*.
notOneThirdChn	If div_en = 0, internal voltage division network is not used, this parameter is used to set ADC input channel. “adc_BatteryCheckInit” acts the same as “adc_Init”.
mode	Set ADC working mode. It’s an enum-type value.
ref_vol	Set ADC reference voltage. It’s an enum-type value.
Resolution	Set ADC resolution. It’s an enum-type value.
sample_cycle	Set ADC sampling time. It’s an enum-type value.

The common function “adc_BatteryCheckInit” can be used for battery voltage detection in two cases:

1) Directly use internal 1/3 voltage division network as ADC input channel.
2) Use GPIO pin with ADC function as ADC input channel, and use external 1/3 voltage division network. In this case, user needs to implement external 1/3 voltage division network.

In “adc_BatteryCheckInit” function, the effect of “oneThirdChn” and “notOneThirdChn” depends on “div_en”.

✧ If div_en = 1, internal 1/3 voltage division network is used as ADC input channel to detect battery voltage. In this case, “notOneThirdChn” is invalid, while “oneThirdChn” is valid, which serves to set Battery channel (VCC) as voltage source for the network.

✧ If div_en = 0, GPIO pin with ADC function is used as ADC input channel, and external 1/3 voltage division network is needed to implement battery voltage detection. In this case, “oneThirdChn” is invalid, while “notOneThirdChn” is valid, which serves to set ADC input channel.

According to ADC hardware circuit feature for 8261/8267/8269, implementation methods of low battery check are shown as below:

1. Adopt the method of 8266, i.e. use external voltage division network, use GPIO pin with ADC function (PC0,PC1…PB6,PB7) as ADC input channel. DCDC is optional. If DCDC is used, ADC should measure the voltage before DCDC conversion.

   This method can use either “adc_Init” or “adc_BatteryCheckInit” for initialization.

   Suppose external 1/3 external voltage division network is used, GPIO PC0 is used as ADC input channel, initialization should be:

```
 adc_Init(ADC_CLK_4M, C0, SINGLEEND, RV_1P428, RES14, S_3);
```

   or

```
adc_BatteryCheckInit(ADC_CLK_4M, 0, 0, C0,
                         SINGLEEND, RV_1P428, RES14, S_3);
```

2. Use 1/3 internal voltage division network as ADC input channel, and select VCC as voltage source of the network. Since VCC is actual voltage of IC AVCC pin, hardware DCDC should not be used.

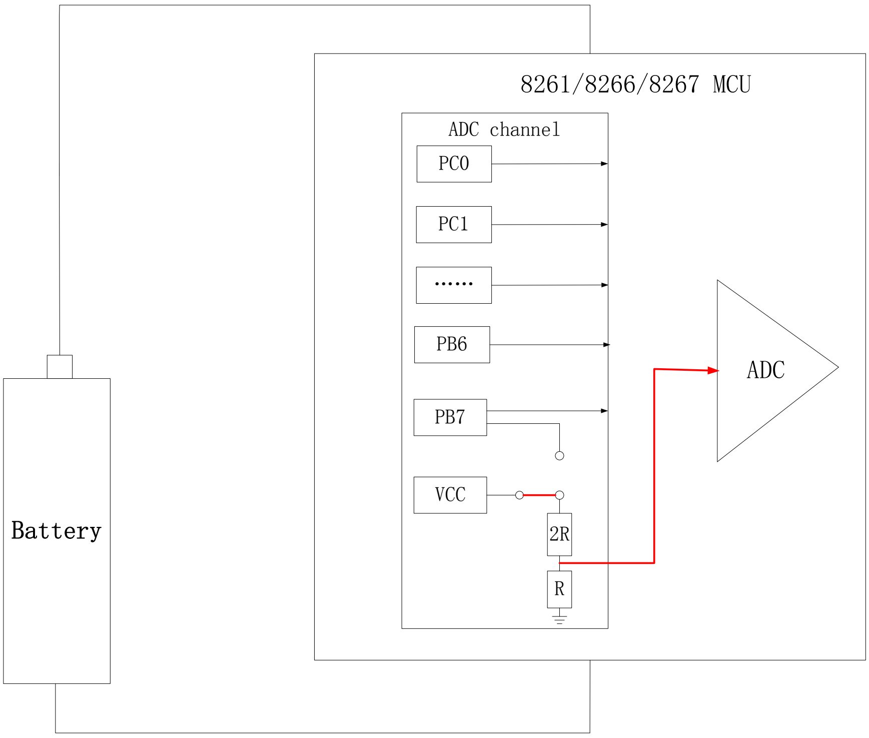


Figure11-10 1/3 internal voltage division network, VCC channel

Only "adc_BatteryCheckInit" can be used for initialization.

```
        adc_BatteryCheckInit(ADC_CLK_4M,1,Battery_Chn_VCC, 0,
                                   SINGLEEND, RV_1P428, RES14, S_3);
```

3. Use 1/3 internal voltage division network as ADC input channel, and select PB7 as voltage source of the network. DCDC is optional. If DCDC is used, ADC should measure the voltage before DCDC conversion (i.e. PB7 should be connected before DCDC conversion).

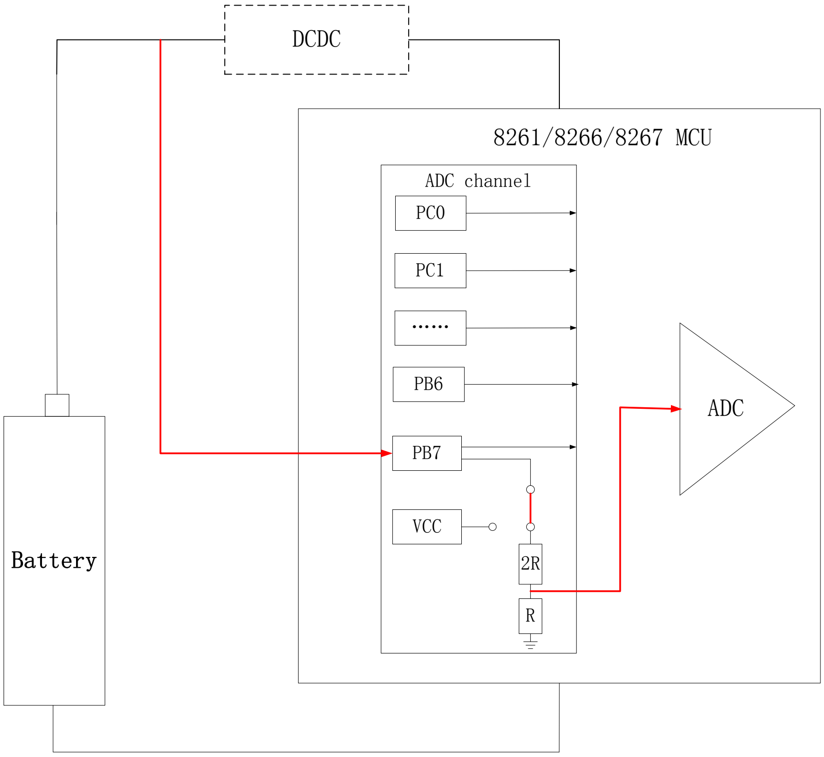


Figure11-11 1/3 internal voltage division network, PB7 channel

Only "adc_BatteryCheckInit" can be used for initialization.

```
    adc_BatteryCheckInit(ADC_CLK_4M, 1, Battery_Chn_B7, 0,
                         SINGLEEND, RV_1P428, RES14, S_3);
```

11.7.2 Demo for "Low battery check"

The project "826x remote"/"826x module" supplies demo for "Low battery check". User can use this method or optimize as needed to implement his low battery check solution.

Take 826x remote demo as an example.

Enable the macro of low battery check in app_config.h.

```
#define BATT_CHECK_ENABLE       1   //enable or disable battery
```

Implement low battery check initialization according to current MCU type:

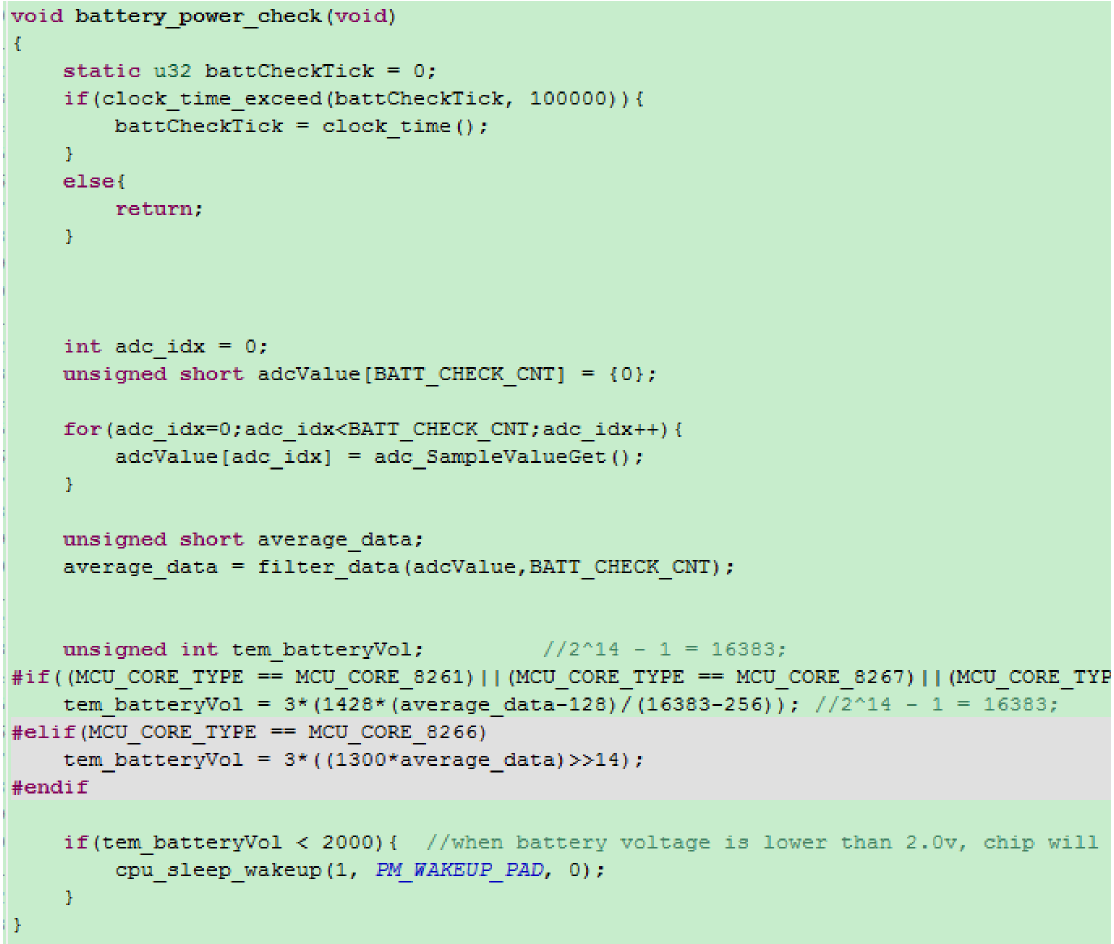
✧ 8266 remote uses external 1/3 voltage division network, connects the voltage after 1/3 voltage division to PC4 pin (ADC input), and uses "adc_init" for initialization.

- 8267 remote does not use DCDC, uses internal 1/3 voltage division network as ADC input, uses VCC as the voltage source of the network, and uses “adc_BatteryCheckInit” for initialization.

```
#if((MCU_CORE_TYPE == MCU_CORE_8261)||(MCU_CORE_TYPE == MCU_CORE_8267)
  || (MCU_CORE_TYPE == MCU_CORE_8269))
        adc_BatteryCheckInit(ADC_CLK_4M,   1,   Battery_Chn_VCC,   0,
                             SINGLEEND, RV_1P428, RES14, S_3);
#elif(MCU_CORE_TYPE == MCU_CORE_8266)
        adc_Init(ADC_CLK_4M, ADC_CHN_C4, SINGLEEND, ADC_REF_VOL_1V3,
                     ADC_SAMPLING_RES_14BIT, ADC_SAMPLING_CYCLE_6);

#endif
```

During battery check in mainloop, once battery voltage is lower than the threshold (2.0V), MCU will be cut off, and invoke “cpu_sleep_wakeup” to directly enter deepsleep. GPIO PAD wakeup is enabled.

Demo code is shown as below.



```
void battery_power_check(void)
{
    static u32 battCheckTick = 0;
    if(clock_time_exceed(battCheckTick, 100000)){
        battCheckTick = clock_time();
    }
    else{
        return;
    }



    int adc_idx = 0;
    unsigned short adcValue[BATT_CHECK_CNT] = {0};

    for(adc_idx=0;adc_idx<BATT_CHECK_CNT;adc_idx++){
        adcValue[adc_idx] = adc_SampleValueGet();
    }

    unsigned short average_data;
    average_data = filter_data(adcValue,BATT_CHECK_CNT);


    unsigned int tem_batteryVol;         //2^14 - 1 = 16383;
#if((MCU_CORE_TYPE == MCU_CORE_8261)||(MCU_CORE_TYPE == MCU_CORE_8267)||(MCU_CORE_TYP
    tem_batteryVol = 3*(1428*(average_data-128)/(16383-256)); //2^14 - 1 = 16383;
#elif(MCU_CORE_TYPE == MCU_CORE_8266)
    tem_batteryVol = 3*((1300*average_data)>>14);
#endif

    if(tem_batteryVol < 2000){  //when battery voltage is lower than 2.0v, chip will
        cpu_sleep_wakeup(1, PM_WAKEUP_PAD, 0);
    }
}
```

Code contains average and filter processing in software, which ensures the accuracy of measured voltage. User can directly use the method in the SDK, or use other better software algorithm.

11.8 IIC

11.8.1 IIC Pin

Telink MCU embeds an IIC module. SDA and SCK line of the IIC module can be mapped to multiple groups (8261/8267/8269: 3 groups; 8266: 1 group) of GPIOs, as shown in the table below.

IIC pin mapping table

IC type	IIC No.	SDA	SCL
8261/8267/8269	1	PA3	PA4
	2	PB6	PB7
	3	PC0	PC1
8266	1	PE7	PF1

Since GPIOs of Telink MCU support multiplexed functions, to use IIC function, it's needed to configure corresponding GPIO pins by invoking "i2c_pin_init ()" function. The function prototype is shown as below:

```
void i2c_pin_init(I2C_GPIO_GroupTypeDef i2c_pin_group)
```

Parameter	Descripton
i2c_pin_group	GPIO pin group to be configured. It's an enum-type value.

E.g.

```
i2c_pin_init(I2C_GPIO_GROUP_C0C1);
```

Enum type definition for "I2C_GPIO_GroupTypeDef" is shown as below:

```
typedef enum {
#if ((MCU_CORE_TYPE == MCU_CORE_8261)||(MCU_CORE_TYPE == MCU_CORE_8267)
      ||(MCU_CORE_TYPE == MCU_CORE_8269))
    I2C_GPIO_GROUP_A3A4,
    I2C_GPIO_GROUP_B6B7,
    I2C_GPIO_GROUP_C0C1,
#elif (MCU_CORE_TYPE == MCU_CORE_8266)
```

```
    I2C_GPIO_GROUP_E7F1,
#endif

    }I2C_GPIO_GroupTypeDef;
```

11.8.2 IIC configuration

IIC module of Telink MCU supports Master mode and Slave mode.

11.8.2.1 IIC Master initialization

Considering different user requirements, driver supplies two functions to initialize IIC Master, including “i2c_master_init_div ()” and “i2c_master_init_khz ()”.

Protorype of “i2c_master_init_div ()” is shown as below:

```
    void i2c_master_init_div( unsigned char slave_id,
                              unsigned char div_clock)
```

Parameter	Descripton
slave_id	Set address of Slave device to be accessed.
div_clock	Set IIC frequency: $f_{iic} = f_{sys\_clk}/(4 * div\_clk)$

E.g.

```
            //When sys_clk=16MHz, f_iic=200KHz
            i2c_master_init_div(0xA0, 0x14);
```

Protorype of “i2c_master_init_khz ()” is shown as below:

```
        void i2c_master_init_khz(  unsigned char slave_id,
                                   unsigned int i2c_speed)
```

Parameter	Descripton
slave_id	Set address of Slave device to be accessed.
i2c_speed	Directly set IIC frequency.

E.g.

```
    //When sys_clk=16MHz, f_iic=200KHz

    i2c_master_init_khz(0xA0, 200);
```

Note: Maximum working frequency for Telink IIC module should be $f_{max\_iic}= f_{sys}/10$. It’s not recommended to use this maximum frequency.

11.8.2.2 IIC Slave initialization

When Telink IIC module is used as Slave, two working modes are supported: DMA mode, Mapping mode. Please refer to IC Datasheet for the detailed introduction.

SDK supplies Slave initialization function to initialize II2 Slave, the prototype of which is shown as below:

```
void i2c_slave_init(   unsigned char device_id,
                        enum I2C_SLAVE_MODE i2c_mode,
                        unsigned char* pbuf)
```

Parameter	Description
device_id	Set device address
i2c_mode	Set Slave Mode. It’s an enum-type value.
pbuf	Pointing to Slave device buffer. This parameter is only valid in Slave Mapping mode, and it should be set as “NULL” in Slave DMA mode.

E.g.

```
    i2c_slave_init(0xA0, I2C_SLAVE_DMA,NULL);
```

Enum type definition for “I2C_SLAVE_MODE” is shown as below:

```
enum I2C_SLAVE_MODE{
    I2C_SLAVE_DMA = 0,
    I2C_SLAVE_MAP,
};
```

When Telink MCU is used as Slave, IIC Master can use Telink MCU or others.

- If Slave works in DMA mode, IIC Master will access register and SRAM space of Slave, so it’s needed to specify the target address in register or SRAM.
- If Slave works in Mapping mode, IIC Master will directly access the space pointed by “pbuf” in “i2c_slave_init ()”, so it’s not needed to specify storage address information.

11.8.3 IIC data transfer

When Telink MCU works in IIC Master mode, it can access Slave device with IIC interface from Telink or other manufacturers.

When Telink MCU works in IIC Slave mode, SPI Master can access the register and SRAM space of Slave in DMA mode or access memory space (buffer) specified by user in Mapping mode.

11.8.3.1 IIC Master write transfer

When Telink MCU works in IIC Master mode, it can access Slave address of 8-bit or 16-bit length. SDK supplies IIC write function, the prototype of which is shown as below:

```
void i2c_write_dma(   unsigned short addr,
                      unsigned char addr_len,
                      unsigned char* pbuf,
                      int len)
```

Parameter	Description
addr	Set Slave address to be accessed.
addr_len	Set Slave address length. 1: 8bit addr; 2: 16bit addr
pbuf	Pointing to buffer which stores data to be written.
len	Length of data to be written.

E.g.

```
unsigned char dataBuf[] = {0x00,0x11,0x22,0x33,0x44};

i2c_write_dma(0x0000,2,dataBuf,sizeof(dataBuf));
```

When Telink MCU works in IIC Slave Mapping mode, user needs to invoke “i2c_write_mapping ()” in IIC Master to write data into Slave, and it’s not needed to specify memory address. The function prototype is shown as below.

```
void i2c_write_mapping(unsigned char* pbuf, int len)
```

Parameter	Description
pbuf	Pointing to buffer which stores data to be written.
len	Date length

E.g.

```
unsigned char writeBuf[]={0x00,0x01,0x02,0x03};

i2c_write_mapping(writeBuf, sizeof(writeBuf));
```

11.8.3.2 IIC Master read transfer

User can invoke “i2c_read_dma ()” to read data of Slave device. The function prototype is shown as below.

```
void i2c_read_dma(     unsigned short addr,
                      unsigned char addr_len,
                      unsigned char* pbuf,
                      int len)
```

Parameter	Description
addr	Set Slave address to be accessed.
addr_len	Set Slave address length 1: 8bit addr; 2: 16bit addr
pbuf	Pointing to buffer which stores the read data.
len	Length of data to be read.

E.g.

```
unsigned char dataBuf[] = {0x00,0x11,0x22,0x33,0x44};
i2c_read_dma(0x00ff,2,dataBuf,sizeof(dataBuf));
```

When Telink MCU works in IIC Slave Mapping mode, user needs to invoke “i2c_read_mapping ()” in IIC Master to read data from Slave, and it’s not needed to specify memory address. The function prototype is shown as below.

```
void i2c_read_mapping(unsigned char* pbuf, int len)
```

Parameter	Description
pbuf	Pointing to data read buffer
len	Data length

E.g.

```
unsigned char readBuf[]={0x00,0x01,0x02,0x03};
i2c_read_mapping(readBuf, sizeof(readBuf));
```

11.8.3.3 IIC Slave data transfer

When IIC works in Slave mode and it’s correctly initialized, data transmission and reception are completely processed by hardware. If Telink MCU works in Slave Mapping mode, when IIC Master needs to access this Slave, it can directly read/write data without specifying memory address.

11.8.4 IIC interrupt

When Telink MCU works in IIC Slave mode, if IIC Master writes data into Slave or reads data from Slave, Slave will check start and stop signal of IIC Master, and then generate interrupt (if enabled). IIC interrupt can be enabled by invoking “I2C_IRQ_EN()” macro function. Note that global interrupt should aslo be enabled by invoking “irq_enable()”.

After IIC interrupt is enabled, user can invoke “I2C_SlaveIrqGet ()” in interrupt handler to obtain IRQ flag bit. The function prototype is shown as below:

```
I2C_I2CIrqSrcTypeDef I2C_SlaveIrqGet(void)
```

Parameter	Description
Return value	Return IRQ flag. It’s an enum value.

Enum type definition for “I2C_I2CIrqSrcTypeDef” is shown as below:

```
typedef enum {
    I2C_IRQ_NONE = 0,
    I2C_IRQ_HOST_WRITE_ONLY,
    I2C_IRQ_HOST_READ_ONLY,
}I2C_I2CIrqSrcTypeDef;
```

11.9 SPI

11.9.1 SPI Pin

Telink MCU embeds a SPI module. This SPI module supports Master/Slave mode, and its MISO, MOSI, CS and CK can be mapped to different GPIOs (8266: 1 group of GPIOs; 8261/8267/8269: two groups of GPIOs), as shown below.

SPI GPIO mapping table

IC type	SPI No.	GPIO Pins	SPI Ctrl Lines
8261/8267/8269	1	PA2	DO
		PA3	DI
		PA4	CK
		PA5	CS
	2	PB4	CS
		PB5	DO
		PB6	DI
		PB7	CK
8266	1	PE6	CS
		PE7	DI
		PF0	DO
		PF1	CK

Since GPIOs of Telink MCU support multiplexed functions, to use SPI function, it's needed to configure corresponding GPIO pins by invoking SPI pin initialization function.

For 8266, the prototype of SPI pin initialization function is shown as below:

```
void spi_master_pin_init (unsigned int cs_pin)
```

Parameter	Description
cs_pin	Set SPI chip select pin

E.g.

```
spi_master_pin_init(GPIO_PC0);
```

For 8261/9267/8269, the prototype of SPI pin initialization function is shown as below:

```
void spi_master_pin_init(    enum spi_pin_t data_clk_pin,
                             unsigned int cs_pin)
```

Parameter	Description
data_clk_pin	Set SPI data and clock pins. It's an enum value.
cs_pin	Set SPI chip select pin.

E.g.

```
    spi_master_pin_init(SPI_PIN_GROUPA,GPIO_PC0);
```

Enum type definition for “spi_pin_t” is shown as below:

```
    enum spi_pin_t{
        SPI_PIN_GROUPA,
        SPI_PIN_GROUPB,
    };
```

Note: If Telink MCU is used as SPI Master, its chip select pin can select CS pin of SPI module, or set other GPIO as SPI CS pin. User only needs to specifiy “cs_pin” in SPI pin initialization function above. If Telink MCU is used as SPI Slave, its chip select pin can only select CS pin of SPI module. Therefore, driver assembles Slave pin initialization in “slave_init ()” (see section 11.9.2), and does not supply independent Slave pin initialization function.

11.9.2 SPI configuration

Telink SPI module supports Master and Slave mode.

Four standard working modes are supported (see IC Datasheet).

11.9.2.1 SPI Master initialization

When Telink MCU works as SPI Master, user needs to invoke “spi_master_init ()” function to configure SPI module. The function prototype is shown as below.

```
void spi_master_init(    unsigned char div_clk,
                         enum spi_mode_t spi_mode)
```

Parameter	Description
div_clk	Set SPI working clock frequency: $f_{spi} = f_{sys}/$ (2 *(div_clk + 1))
spi_mode	Set SPI working mode. It’s an enum value.

E.g.

```
    spi_master_init(0x0f, SPI_MODE0);
```

spi_mode_t definition is shown as below:

```
enum spi_mode_t{
        SPI_MODE0 = 0,
        SPI_MODE1 = 2,
        SPI_MODE2 = 1,
        SPI_MODE3 = 3,
};
```

Note: $f_{spi}< f_{sys}/ 5$;

11.9.2.2 SPI Slave initialization

When Telink MCU works as SPI Slave, user needs to invoke “spi_slave_init()” function to configure SPI module.

For 8266, the function prototype is shown as below.

```
void spi_slave_init(enum spi_mode_t spi_mode)
```

Parameter	Description
spi_mode	Set SPI working mode. It’s an enum value.

E.g.

```
spi_slave_init(SPI_MODE0);
```

For 8261/9267/8269, the function prototype is shown as below.

```
void spi_slave_init(    enum spi_pin_t spi_grp,
                        enum spi_mode_t spi_mode)
```

Parameter	Description
spi_grp	SPI GPIO pins. It’s an enum value.
spi_mode	SPI mode. It’s an enum value.

例：

```
spi_slave_init(SPI_PIN_GROUPA,SPI_MODE0);
```

Enum type definition for “spi_pin_t” definition:

```
enum spi_pin_t{
    SPI_PIN_GROUPA,
    SPI_PIN_GROUPB,
};
```

Enum type definition for “spi_mode_t”:

```
enum spi_mode_t{
        SPI_MODE0 = 0,
        SPI_MODE1 = 2,
        SPI_MODE2 = 1,
        SPI_MODE3 = 3,
};
```

11.9.3 SPI Data transfer

When Telink MCU is used as SPI Master, it can access Slave device with SPI interface from Telink or other manufacturers.

When Telink MCU is used as SPI Slave, SPI Master can access the register and SRAM space of Slave.

11.9.3.1 SPI Master write transfer

When SPI works in Master mode, after initialization is finished, “spi_write ()” function can be invoked to write data. The function prototype is shown as below.

```
void spi_write(  unsigned char* addr_cmd,
                 unsigned char addr_cmd_len,
                 unsigned char* pbuf,
                 int buf_len,
                 unsigned int cs_pin)
```

Parameter	Description
addr_cmd	Pointing to Register address to be written or buffer of write command
addr_cmd_len	Length of address and command
pbuf	Pointing to buffer which stores data to be written
buf_len	Length of data to be written
cs_pin	Chip select pin

E.g.

```
 #define SLAVE_REG_ADD_H 0x80
 #define SLAVE_REG_ADD_L 0x00
 #define SPI_WRITE_CMD   0x00//telink SPI write cmd=0x00
unsigned char slaveRegAddr_WriteCMD[]= {SLAVE_REG_ADD_H,
                                        SLAVE_REG_ADD_L,
                                         SPI_WRITE_CMD};
unsigned char spi_write_buff[]= {0x00,0x11,0x22};
spi_write(slaveRegAddr_WriteCMD, 3,
            spi_write_buff,  sizeof (spi_write_buff),
            GPIO_PC0
            );
```

11.9.3.2 SPI Master read transfer

User can invoke “spi_read()” function to read data. The function prototype is shown as below.

```
void spi_read(  unsigned char* addr_cmd,
                 unsigned char addr_cmd_len,
                 unsigned char* pbuf,
                 int buf_len,
                 unsigned int cs_pin)
```

Parameter	Description
addr_cmd	Pointing to Register address to be read or buffer of read command
addr_cmd_len	Length of address and command
pbuf	Pointing to buffer which stores the read data
buf_len	Length of data to be read
cs_pin	Chip select pin

E.g.

```
 #define SLAVE_REG_ADD_H 0x80
 #define SLAVE_REG_ADD_L 0x00
 #define SPI_READ_CMD    0x80//telink SPI read cmd=0x80
```

```
unsigned char slaveRegAddr_ReadCMD[]= {SLAVE_REG_ADD_H,
                                       SLAVE_REG_ADD_L,
                                       SPI_READ_CMD};
unsigned char spi_read_buff[10];
spi_read(slaveRegAddr_ReadCMD, 3,
            spi_read_buff,  sizeof (spi_read_buff),
            GPIO_PC0
            );
```

11.9.3.3 SPI Slave data transfer

When SPI works in Slave mode and it's correctly initialized, data transmission and reception are completely processed by hardware.

11.9.4 SPI interrupt

When Telink MCU works in SPI Slave mode, if SPI Master writes data into Slave or reads data from Slave, Slave will generate interrupt (if enabled). SPI interrupt can be enabled by invoking "SPI_IRQ_EN()" macro function. Note that global interrupt should aslo be enabled by invoking "irq_enable()".

After SPI interrupt is enabled, user can invoke "SPI_IRQ_GET()" macro function in interrupt handler to obtain IRQ flag bit. The IRQ flag can be cleared by invoking "SPI_IRQ_CLK()" macro function.

11.10 EMI

11.10.1 EMI Test

This section will take 8267 as an example to illustrate EMI test. 8261/8269 share the same hardware registers as 8267, and they actually invoke EMI test interface function of 8267; 8266 also invokes the same interface function.

During EMI Test, it's needed to invoke interafces related to rfdrv, e.g. rf_drv_init(), rf_drv_1m(), rf_drv_2m(), and etc. All of these interfaces are assembled in library. API declaration is viewable in "rf_drv_826x.h".

EMI Test supports four test mode: Carrire mode (send carrier only), CD mode (send Carrirer with data), RX mode, TX mode. TX mode supports three sub-modes with different packet types.

```
Struct test_list_sate_list[] = {
        {0x01,EmiCarrierOnly},
        {0x02,EmiCarrierData},
        {0x03,EmiRx},
        {0x04,EmiTxPrbs9},
        {0x05,EmiTx55},
        {0x06,EmiTxff},

    };
```

11.10.1.1 Emi initialization setting

1) Before EMI test, first it's needed to invoke "rf_drv_init()" function to initialize RF.

   void rf_drv_init (int xtal_type);

   "xtal_type" serves to select external crystal, XTAL_12M/ XTAL_16M.

2) After RF configuration, it's needed to invoke "Rf_Emi_Init()" function to record some data before test.

   int Rf_EmiInit(void);

3) If it's needed to enable usbprint function, the configurations below should be implemented.

   WriteAnalogReg (0x88, 0x0f);

   WriteAnalogReg (0x05, 0x60);

   write_reg8(0x80013c,0x10);

4) Set EMI initial status:

```
write_reg8(RUN_STATUE_ADDR,run);                  //run,0
write_reg8(TEST_COMMAND_ADDR,cmd_now);            //cmd,1
write_reg8(POWER_ADDR,power_level);               //power,0
write_reg8(CHANNEL_ADDR,chn);                     //chn,2
write_reg8(RF_MODE_ADDR,mode);                    //mode,1
write_reg8(TX_PACKET_MODE_ADDR,tx_mode);          //tx_mode,0
```

5) Invoke "PhyTest_PRBS9()" function to implement data initialization setting in Tx buffer.

   void PhyTest_PRBS9 (unsigned char *p, int n);

"PhyTest_PRBS9()" will write prbs9 data into buffer with initial address "p" and length "n".

6) Before EMI test, finally it's needed to invoke "irq_disable()" function to disable all interrupts.

11.10.1.2 Power level and Channel

During EMI test, user can configure “rf power level” and “rf channel”, which will determine energy and channel for packet transmission.

Power level	Select Tx power, enum data type
RF_POWER_8dBm	0, 7dBm Tx power (actual value)
RF_POWER_4dBm	1, 5dBm Tx power (actual value)
RF_POWER_0dBm	2, -0.6dBm Tx power (actual value)
RF_POWER_m4dBm	3, -4.3dBm Tx power (actual value)
RF_POWER_m10dBm	4, -9.5dBm Tx power (actual value)
RF_POWER_m14dBm	5, -13.6dBm Tx power (actual value)
RF_POWER_m20dBm	6, -18.8dBm Tx power (actual value)
RF_POWER_m24dBm	8, -23.3dBm Tx power (actual value)
RF_POWER_m28dBm	9, -27.5dBm Tx power (actual value)
RF_POWER_m30dBm	10, -30dBm Tx power
RF_POWER_m37dBm	11, -37dBm Tx power
RF_POWER_OFF	16, disable PA

Note: Power level will be configured as actual Tx power value. For example, if power level is set as “RF_POWER_8dBm”, actual Tx power should be 7dBm rather than 8dBm.

Power setting can be implemented by invoking the function below.

```
void	rf_set_power_level_index (int level);
```

“level”: Power level.

RF Channel: Set frequency as (2400+chn) MHz. (0≤chn≤100)

For example, to set channel as 2405MHz, “chn” should be set as 5. The function below can be invoked.

```
void	SetRxMode (signed char chn, unsigned short set);
```

“chn”: RF channel.

“set”: set as 0.

11.10.1.3 Emi Carrier Only

For Carrier mode, user only needs to directly invoke “EmiCarrierOnly()”.

```
void  EmiCarrierOnly(int power_level, signed char rf_chn);
```

“power_level”: Power level. (See section 11.10.1.2).

“rf_chn”: RF channel. (See section 11.10.1.2).

In Carrier mode, “EmiCarrierOnly()” will invoke “Rf_EmiCarrierRecovery()” function to restore some registers of EMI to default setting.

```
int        Rf_EmiCarrierRecovery(void);
```

11.10.1.4 Emi Carrier Data

In CD mode, data in carrier are updated via “Rf_EmiDataUpdate()” function to ensure the data are random numbers. User only needs to invoke “EmiCarrierData()” function to enter CD mode.

```
void     EmiCarrierData(int power_level,signed char rf_chn);
```

“power_level”: Power level. (See section 11.10.1.2).

“rf_chn”: RF channel. (See section 11.10.1.2).

In CD mode, “EmiCarrierData()” will invoke “Rf_EmiCarrierDataTest()” function to implement the setting of CD mode (e.g. power level, chn, and etc), and invoke “Rf_EmiDataUpdate()” functioin to update data in carrier.

```
void     Rf_EmiCarrierDataTest(int power_level,signed char rf_chn)；
void     Rf_EmiDataUpdate(void)；
```

11.10.1.5 Emi TX

TX mode supports three sub-modes with different packet types, including “PRBS9 packet payload”, “00001111 packet payload” and “10101010 packet payload”. User can directly invoke “EmiTXff()”/“EmiTx55()”/“EmiTxPrbs9()” to enter corresponding TX sub mode.

```
void     EmiTxPrbs9(int power_level,   signed char rf_chn);
void     EmiTx55(int power_level,   signed char rf_chn);
void      EmiTxff(int power_level,   signed char rf_chn);
```

“power_level”: Power level. (See section 11.10.1.2).

“rf_chn”: RF channel. (See section 11.10.1.2).

"EmiTXff()"/"EmiTx55()"/"EmiTxPrbs9()" will invoke "Rf_EmiTxInit()" function to implement EMI TX initialization setting, and invoke "Rf_EmiSingleTx()" function to start packet transmission.

void Rf_EmiTxInit(int power_level, signed char rf_chn);

void Rf_EmiSingleTx(unsigned char *addr, int power_level);

"addr": Starting adderss of Tx buffer.

11.10.1.6 EMI RX

Rx mode adopts inquiry method to receive data. In Rx mode, "EmiRx()" function is used to implement initialization setting of status register and flag bit. Flag bit can be used to check whether there are new data received, and "EmiRxProc()" function serves to process the received data. User only needs to directly invoke "EmiRx()" and "EmiRxProc()" to enter Rx mode.

void EmiRx(intpower_level, signed char rf_chn);

void EmiRxProc(void);

"power_level": Power level. (See section 11.10.1.2).

"rf_chn": RF channel. (See section 11.10.1.2).

"EmiRx()" will invoke "Rf_EmiRxTest()" function to set starting address of Rx buffer, size, chn and etc.

```
void Rf_EmiRxTest (    unsigned char *addr,
                       signed char rf_chn,
                       int buffer_size,
                       unsigned char    pingpong_en)
```

"addr": Pointer of Rx buffer in RAM (Generally it's starting addres of an array).

"buffer_size": buffer length (integral multiple of 16, mainly used for pingpong buffer).

"rf_chn": set Tx RF channel.

"pingpong_en": Enable (1)/Disable (0) pingpong buffer.

11.10.1.7 Set configuration parameters

Run：

0	Default	1	Start test

Cmd：

1	Carrier	2	CD	3	RX
4	TX(PRBS9)	5	TX(0x55)	6	TX(0x0f)

For **Power** and **channel**, please refer to section 11.10.1.2.

Mode:

0	Ble_2M	1	Ble_1M

The default setting of these parameter are (mode=1; power=0; channel=2; cmd=1), i.e. send carrier in ble_1M mode with 2402MHz frequency and 7dBm Tx power.

Parameters can be configured via flash or RAM address correspondingly. To modify customized parameters in flash, it’s needed to erase this area before writing new value. If RAM address method is used, the configured parameter will restore to its default value after power down.

Note: Flash adderss is modifiable. Please refer to Flash space allocation.

Take 3f000 sector as an example to illustrate how to set configuration parameters via flash.

```
#define EMI_TEST_TX_MODE           0x3f005
#define EMI_TEST_RUN               0x3f006
#define EMI_TEST_CMD               0x3f007
#define EMI_TEST_POWER_LEVEL       0x3f008
#define EMI_TEST_CHANNEL           0x3f009

#define EMI_TEST_MODE              0x3f00a
```

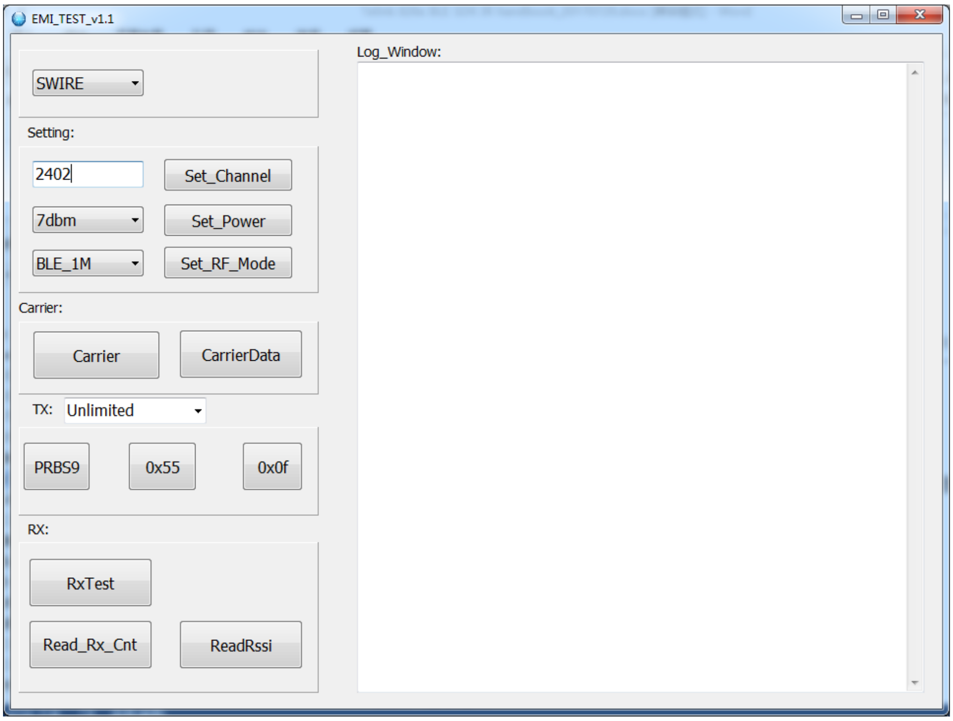
Test status can be modified via flash address space 0x3f007~0x3f00a. User can write parameter into corresponding flash address, and power cycle DUT to get the wanted status.

Test status can also be modified via RAM address 0x8007~0x800a. User can write parameter into corresponding RAM address, and write RAM "0x8006” with "1” to get the wanted status.

Ram_address	Flash_address	Function
0x8004		Rssi
0x8005	0x3f005	Set number of Tx packets as 1000 or unlimited for cmd(4/5/6). 1: send 1000 packets and then stop 0: continuously send packets
0x8006		Run
0x8007	0x3f007	Cmd
0x8008	0x3f008	Power
0x8009	0x3f009	Channel
0x800a	0x3f00a	Mode
0x800c-0x800f		RX_packet_num (In RX mode, the address space stores the number of received packets, 4 bytes).
	0x1E000/0x77000	0x81_Cap_value(0xbf<cap< 0xe0）
	0x1E040/0x77040	Tp0 (1M: 0x13<Tp0<0x27; 2M: 0x36<Tp0<0x4a)
	0x1E080/0x77041	Tp1 (1M: 0x0f<Tp0<0x23; 2M: 0x2f<Tp1<0x43)

11.10.2 EMI Test Tool

“EMI Test Tool” can be used to implement EMI test. The tool interface is shown as below.



EMI_TEST_v1.1

SWIRE

Setting:

2402

7dbm

BLE_1M

TX: Unlimited

Log_Window:

Figure11-12 EMI test tool

Step 1: User can select hardware connection method as needed. When “Swire” is selected, if system clock is 16MHz or below, it’s needed to implement “SWB SPEED” (click “SWB SP”) on Wtcdb tool to ensure normal communication.

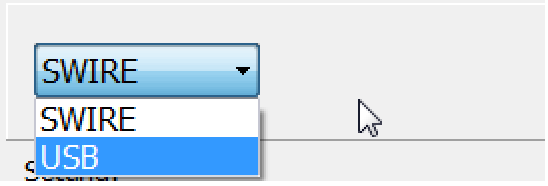
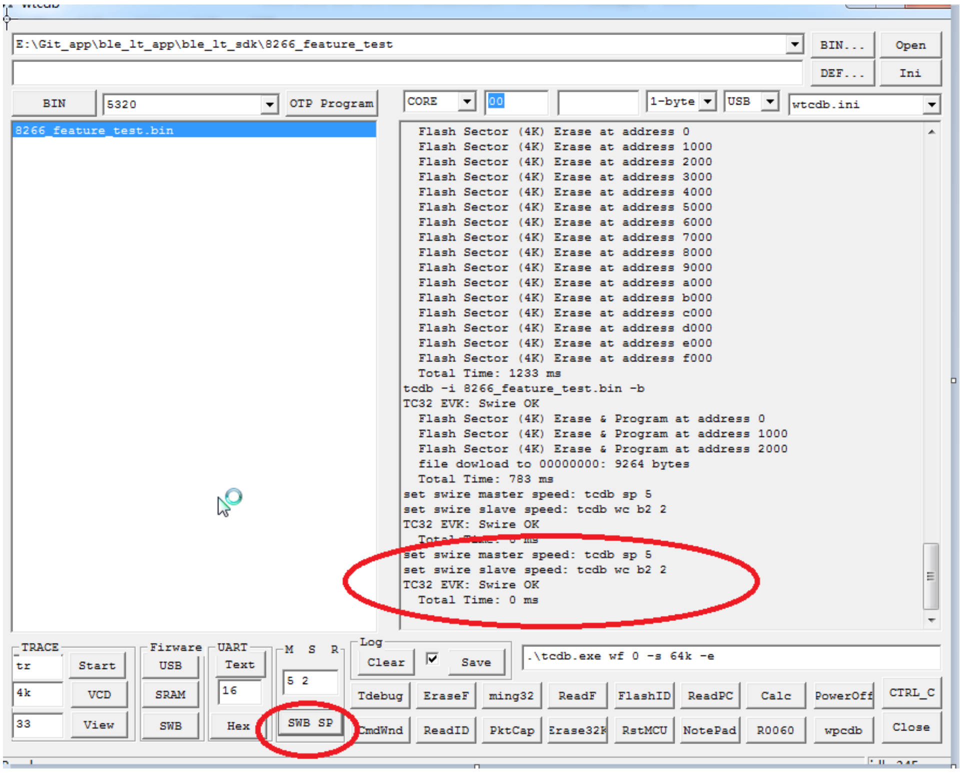


Figure11-13 Select data bus



E:\Git_app\ble_lt_app\ble_lt_sdk\8266_feature_test

5320 | CORE | 00 | 1-byte | USB | wtcdb.ini

8266_feature_test.bin

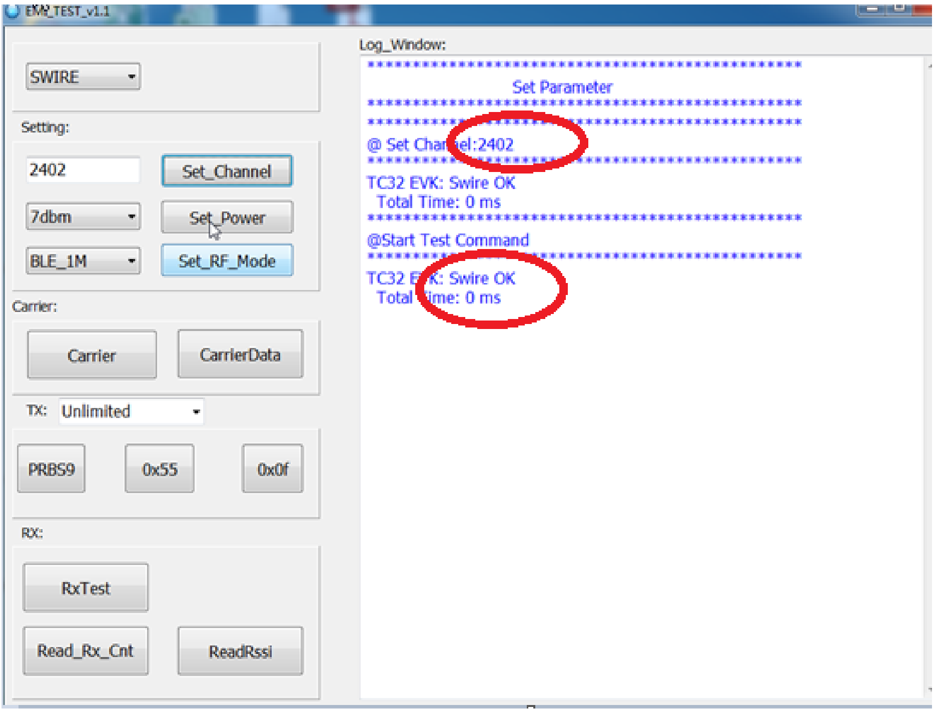
```
  Flash Sector (4K) Erase at address 0
  Flash Sector (4K) Erase at address 1000
  Flash Sector (4K) Erase at address 2000
  Flash Sector (4K) Erase at address 3000
  Flash Sector (4K) Erase at address 4000
  Flash Sector (4K) Erase at address 5000
  Flash Sector (4K) Erase at address 6000
  Flash Sector (4K) Erase at address 7000
  Flash Sector (4K) Erase at address 8000
  Flash Sector (4K) Erase at address 9000
  Flash Sector (4K) Erase at address a000
  Flash Sector (4K) Erase at address b000
  Flash Sector (4K) Erase at address c000
  Flash Sector (4K) Erase at address d000
  Flash Sector (4K) Erase at address e000
  Flash Sector (4K) Erase at address f000
  Total Time: 1233 ms
tcdb -i 8266_feature_test.bin -b
TC32 EVK: Swire OK
  Flash Sector (4K) Erase & Program at address 0
  Flash Sector (4K) Erase & Program at address 1000
  Flash Sector (4K) Erase & Program at address 2000
  file dowload to 00000000: 9264 bytes
  Total Time: 783 ms
set swire master speed: tcdb sp 5
set swire slave speed: tcdb wc b2 2
TC32 EVK: Swire OK
  Total Time: 0 ms
set swire master speed: tcdb sp 5
set swire slave speed: tcdb wc b2 2
TC32 EVK: Swire OK
  Total Time: 0 ms
```

tr | 4k | 33 | 16 | 5 2 | ☑

```
.\tcdb.exe wf 0 -s 64k -e
```

Figure11-14 Swire synchronization operation

Step 2: Set “chn”, i.e. input frequency (e.g. 2402) in the corresponding box and click “Set_Channel”. The log window will show “Swire OK” to indicate normal communication, as shown below.



```
*************************************************
                 Set Parameter
*************************************************
*************************************************
@ Set Channel:2402
*************************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
*************************************************
@Start Test Command
*************************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
```

Figure11-15 Set channel

Step 3: Select power level and BLE mode via the corresponding drop-down box, and click “Set_Power”/”Set_RF_Mode”.

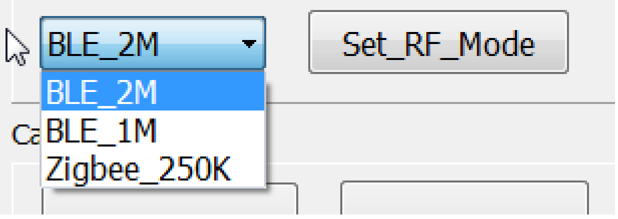
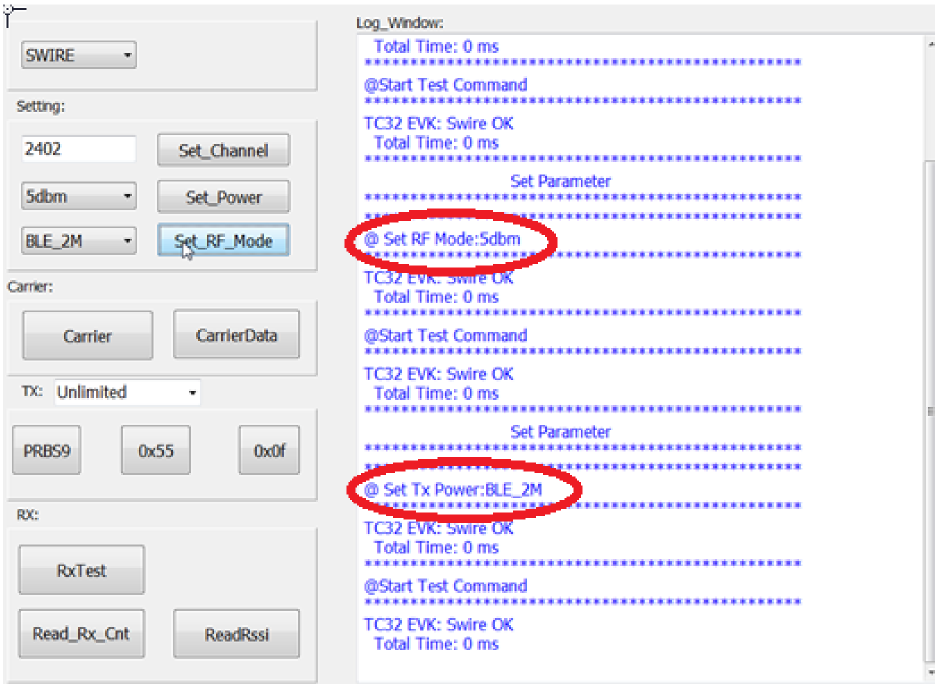


Figure11-16 Select RF mode



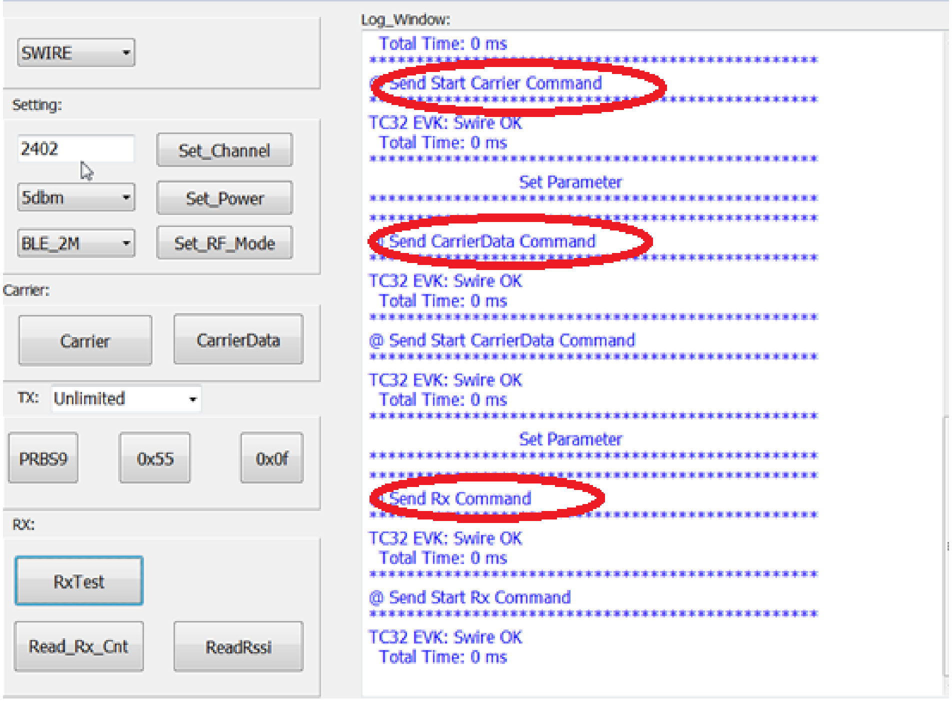
Setting: 2402, 5dbm, BLE_2M

TX: Unlimited

```
  Total Time: 0 ms
**********************************************
@Start Test Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
                 Set Parameter
**********************************************
**********************************************
@ Set RF Mode:5dbm
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@Start Test Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
                 Set Parameter
**********************************************
**********************************************
@ Set Tx Power:BLE_2M
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@Start Test Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
```

Figure11-17 Interface after RF mode setting

Step 4: Click “Carrier”/“CarrierData”/“RXTest”/“PRBS9”/“0x55”/“0x0f” to enter corresponding test mode.



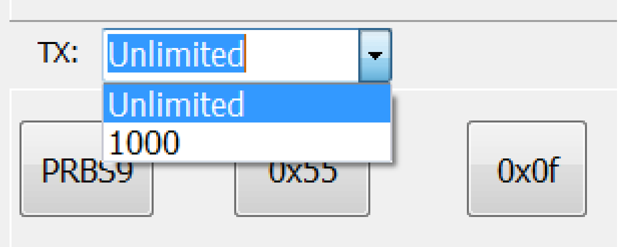
Setting: 2402, 5dbm, BLE_2M

TX: Unlimited

```
  Total Time: 0 ms
**********************************************
@ Send Start Carrier Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
                 Set Parameter
**********************************************
**********************************************
@ Send CarrierData Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@ Send Start CarrierData Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
                 Set Parameter
**********************************************
**********************************************
@ Send Rx Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@ Send Start Rx Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
```

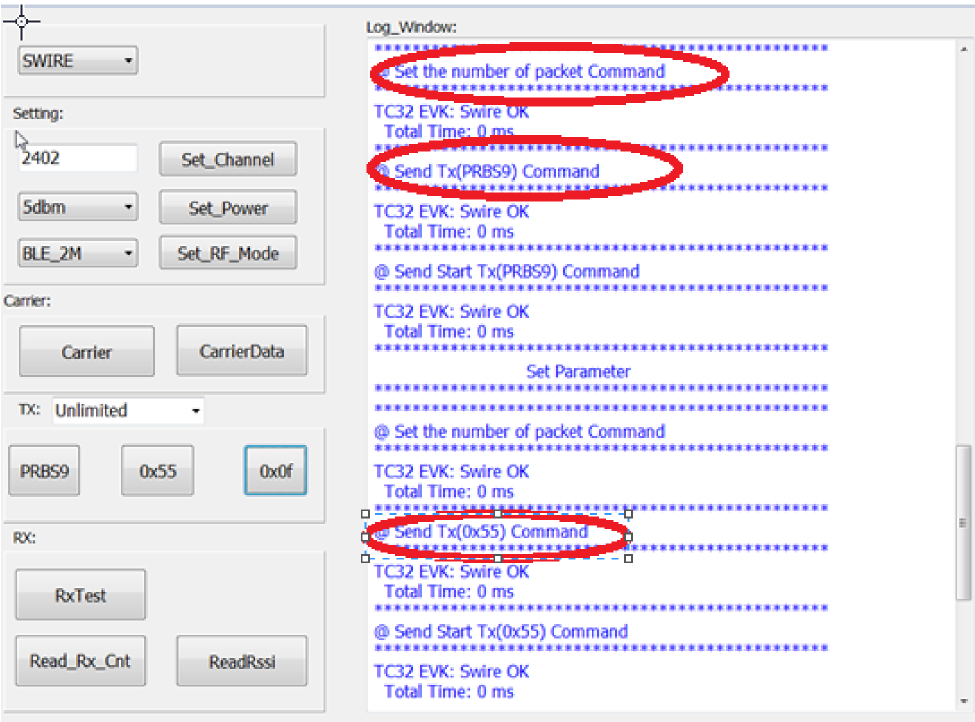
Figure11-18 Select test mode

Step 5: In TX mode, user can select to send 1000 packets or unlimited packets.



TX: Unlimited

Figure11-19 Set TX packet number



Setting: 2402, 5dbm, BLE_2M

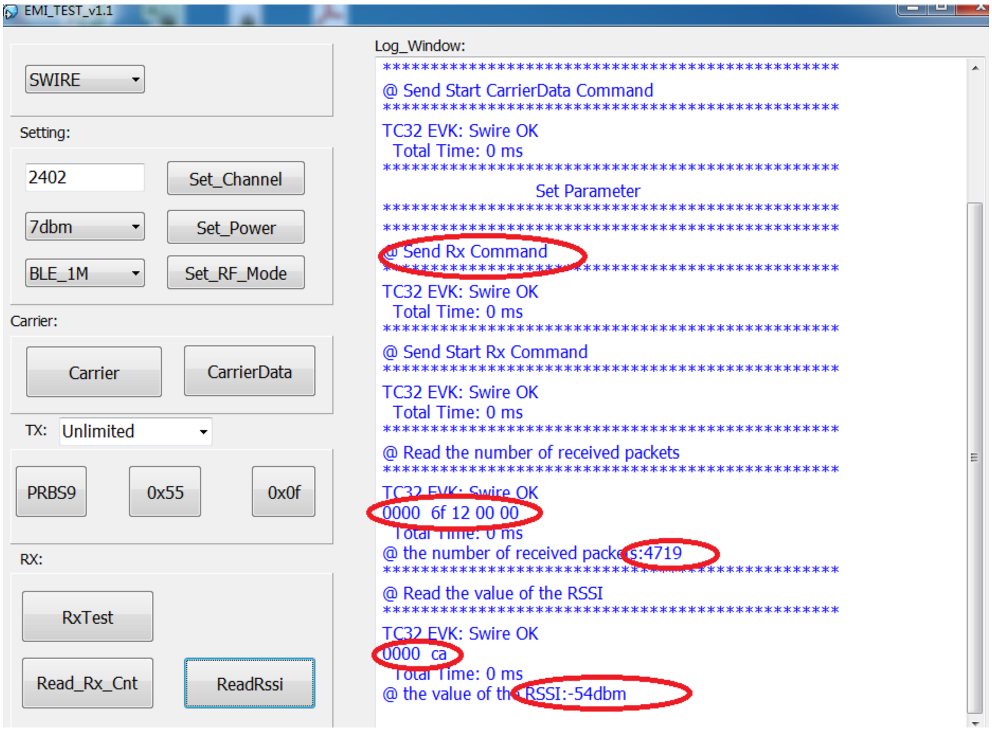
TX: Unlimited

Log_Window:

```
**********************************************
@ Set the number of packet Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@ Send Tx(PRBS9) Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@ Send Start Tx(PRBS9) Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
                Set Parameter
**********************************************
**********************************************
@ Set the number of packet Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@ Send Tx(0x55) Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
**********************************************
@ Send Start Tx(0x55) Command
**********************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
```

Figure11-20 TX mode interface

Step 6: In RX mode, number of received packets can be read by clicking “Read_Rx_Cnt”, while current RSSI can be obtained by clicking “ReadRssi”, as shown below.



```
************************************************
@ Send Start CarrierData Command
************************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
************************************************
                  Set Parameter
************************************************
************************************************
@ Send Rx Command
************************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
************************************************
@ Send Start Rx Command
************************************************
TC32 EVK: Swire OK
  Total Time: 0 ms
************************************************
@ Read the number of received packets
************************************************
TC32 EVK: Swire OK
0000  6f 12 00 00
  Total Time: 0 ms
@ the number of received packets:4719
************************************************
@ Read the value of the RSSI
************************************************
TC32 EVK: Swire OK
0000  ca
  Total Time: 0 ms
@ the value of the RSSI:-54dbm
```

Figure11-21 Read RX packet number and RSSI

11.11 PHY test

To be added.

12 Appendix

Appendix 1: crc16 algorithm

```
unsigned short crc16 (unsigned char *pD, int len)
{
    static unsigned short poly[2]={0, 0xa001};
    unsigned short crc = 0xffff;
    unsigned char ds;
    int i,j;

    for(j=len; j>0; j--)
    {
        unsigned char ds = *pD++;
        for(i=0; i<8; i++)
        {
            crc = (crc >> 1) ^ poly[(crc ^ ds ) & 1];
            ds = ds >> 1;
        }
    }

    return crc;
}
```