Application Note

Telink TLSR8232 BLE SDK Developer Handbook

AN-19112700-E1

Version 1.0.0

2019-11-27

Brief:

This document is the developer guide for TLSR8232 BLE SDK 1.3.0.

TELINK SEMICONDUCTOR

Published by

Telink Semiconductor

Bldg 3, 1500 Zuchongzhi Rd,

Zhangjiang Hi-Tech Park, Shanghai, China

Legal Disclaimer

Information:

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

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

telinkcnsales@telink-semi.com

telinkcnsupport@telink-semi.com

Revision History

Version 1.0.0 (2019-11-27)

This is the Initial release.

Contents

Contents of Figures

1. SDK Overview

Telink 5316 BLE SDK provides demo code for BLE slave development, based on which users can develop their own application programs.

Currently 5316 BLE SDK applies to ICs TLSR8232F512 and TLSR8232F128 (5316 and 8232 refer to the same IC, 5316 is the name for Telink internal use, while 8232 is the name for external use).

1.1 Software Architecture

Software architecture for Telink 5316 BLE SDK includes APP layer and BLE protocol stack.

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

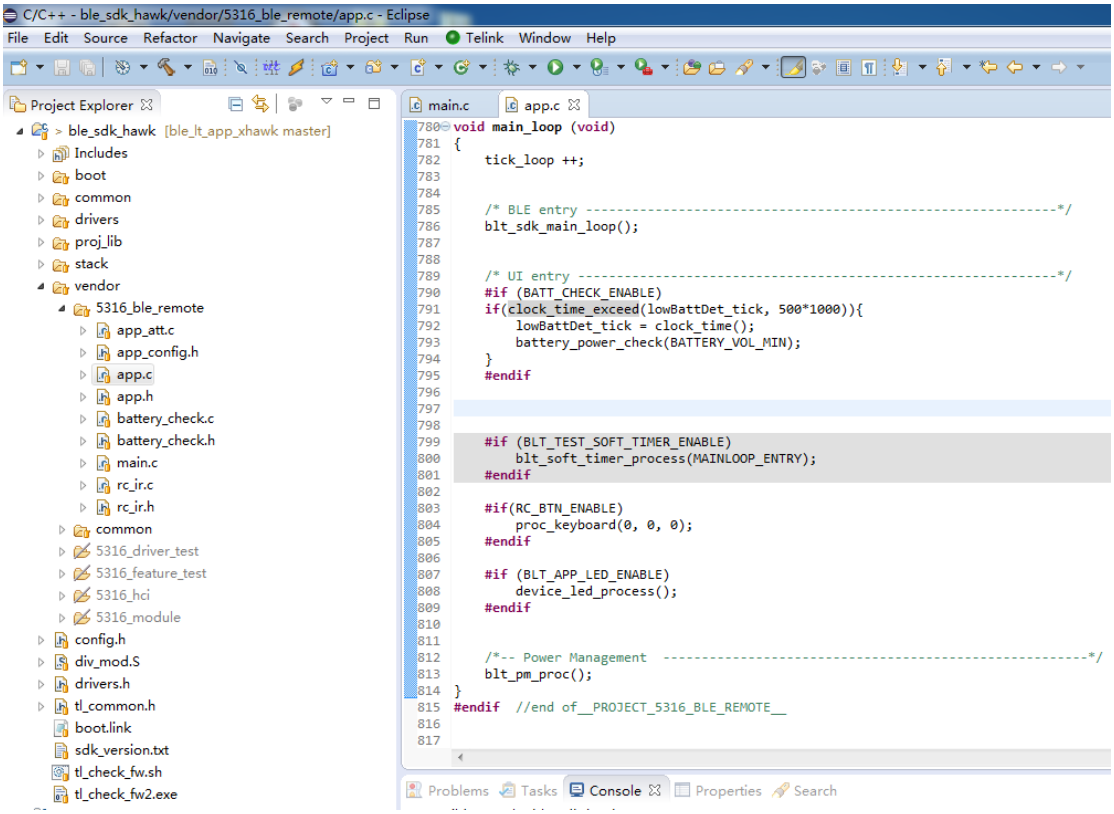


Figure 1-1 SDK File Structure

- ✧ boot: This folder contains software bootloader.
- ✧ common: This folder contains generic APIs, such as printf and shift operation.
- ✧ drivers: This folder contains all the peripheral drivers of 5316, such as ADC, I2C, SPI, and UART.
- ✧ proj_lib: This folder contains library files necessary for MCU running, including BLE stack, RF driver, PM driver, etc. This folder is provided in the form of library files, the source files, like liblt_5316.a, are not open to users.

✧ vendor: This folder contains user APP-layer code, e.g. 5316_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 the entry function “main” of the program, 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(); // basic MCU hardware initialization, negligible to users


    clock_init(SYS_CLK_16M_Crystal); // clock initialization, users only needs to fill related parameters accordingly.


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


    rf_drv_init(RF_MODE_BLE_1M); // RF initialization, negligible to users, only support BLE 1M


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


    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, and proprietary OTA, users can add their 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

TLSR8232F512/ TLSR8232F128: The two ICs share the same IP core, thus their hardware modules are almost the same except Flash size as shown below.

IC	Flash size	SRAM size
TLSR8232F512	512kB	16kB
TLSR8232F128	128kB	16kB

1.3 library

SDK 1.2.0 provides two libraries, liblt_5316_512K.a and liblt_5316_128K.a. Providing two libraries is to save some Flash space for 128k Flash IC, which is at the expense of removing BLE4.2 DLE features, etc.

It is verified that removing some features of BLE saves little space while loses some functions of BLE and brings inconvenience to users. Therefore, SDK 1.3.0 reduced its library to liblt_5316.a only.

If the user uses SDK 1.2.0, he should adjust the library according to Flash size. Figure 1-2 shows the adjustment method.

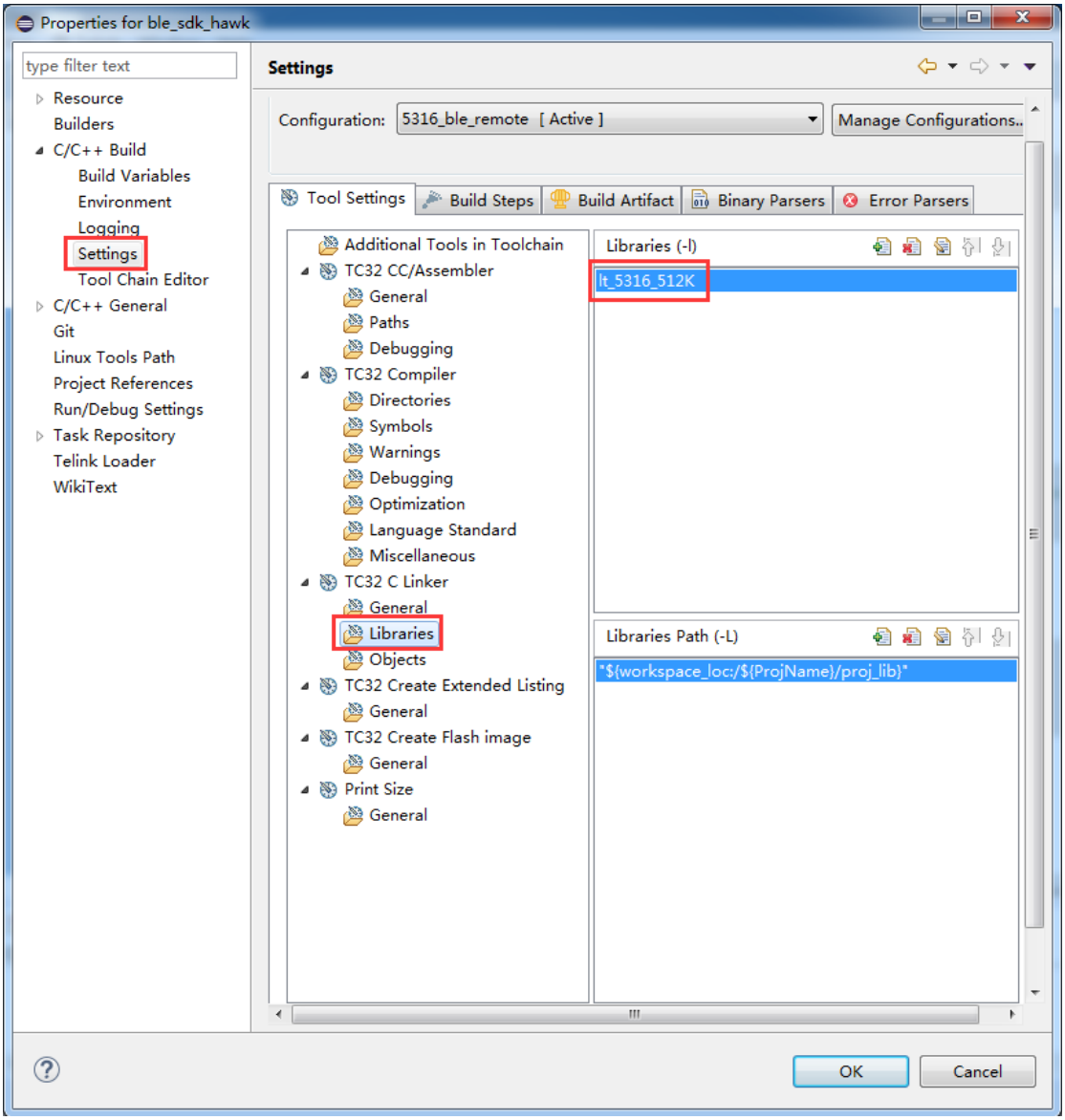


Figure 1-2 Select library

1.4 Demo

5316 BLE SDK provides multiple BLE demos for users. Each demo code has its specific hardware. Through running the demo, a user can observe effects directly and modify demo code for his own application development.

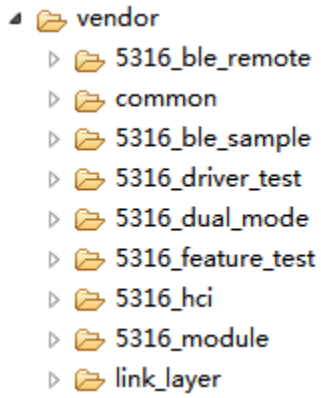


Figure 1-3 Demos in BLE SDK

1.4.1 BLE Slave Demo

BLE Slave demos and their differences are listed in the table below:

Demo	Stack	Application	MCU function
5316 hci	BLE controller	No	Controller, communicate with peer MCU hosts via HCI interface
5316 module	BLE controller + host	Application in Host MCU	BLE SPP module
5316 remote	BLE controller + host	Remote control application	Host MCU
5316 sample	BLE controller + host	No	Host MCU
5316 dual mode	BLE + 2.4G	Dual mode	Host MCU

5316 hci is a BLE Slave controller. Through its UART-based HCI, 5316 hci can communicate with other MCU Host, which therefor forms a complete BLE Slave system.

5316 remote/5316 module/5316 sample are all complete BLE Slave stacks. 5316 module only acts as BLE SPP module to communicate with Host MCU via UART interface. Usually applications are written in the (peer) MCU with BLE host. 5316 remote is a demo of BLE remote controller which supports basic functions of remote. It can connect with standard iOS/Android device or Telink 826x master kma dongle to control the peer. 5316 sample has the same functions with 5316 remote but different hardware. 5316 sample is used for TLSR8232 development board. It can save the hardware cost for users for that users can use Telink BLE without purchasing Telink demo RCU.

5316 dual mode supports both BLE and 2.4G. BLE and 2.4G work in a switch mode, which means 2.4 cannot work when 5316 dual mode works in BLE and vice versa, but you can switch the two modes in operation.

1.4.2 Other Demos

5316 feature test provides demo code for some common features related to BLE. Users can implement their own functions based on these demos. All features will be introduced in BLE section.

5316 driver test provides sample code for basic drivers, based on which users can implement their own driver functions.

2. MCU Basic Modules

2.1 MCU Address Space

2.1.1 MCU Address Space Allocation

Telink 5316 MCU supports maximum addressing space of 16M bytes, including 8M-byte program space from 0 to 0x7FFFFF (Please see the datasheet for Flash sizes) and 8M-byte peripheral space from 0x800000 to 0xffffff. Among the 8M-byte peripheral space, 0x800000 to 0x808000 are for register space, 0x808000 to 0xFFFFFF for SRAM space (Please see the datasheet for SRAM sizes).

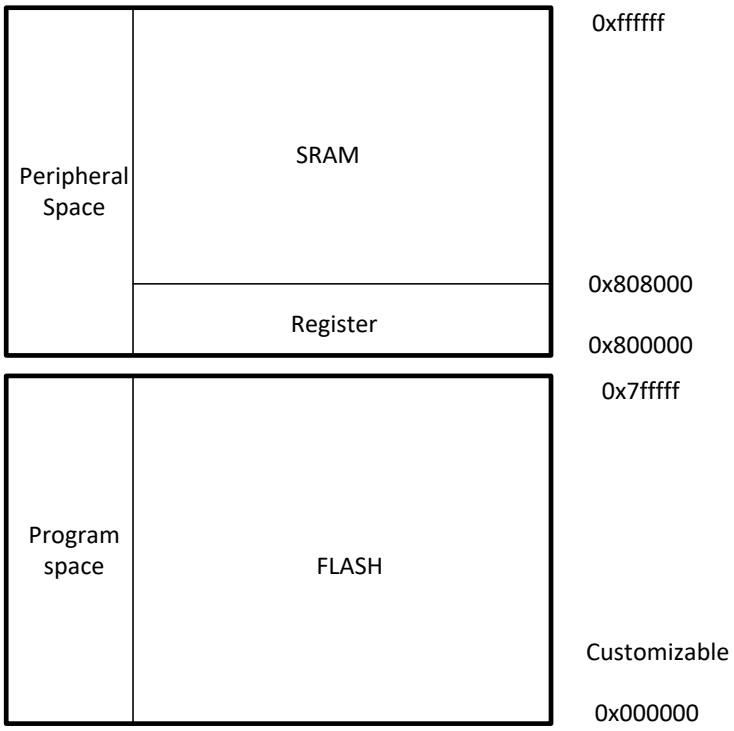


Figure 2-1 MCU Address Space Allocation

2.1.2 SRAM Space Allocation

2.1.2.1 SRAM and Firmware Space

This section provides a further description of SRAM space allocation in MCU address space.

For 16kB SRAM, the address space range is 0x808000 to 0x80C000.

The figure below shows SRAM and Firmware space allocation.

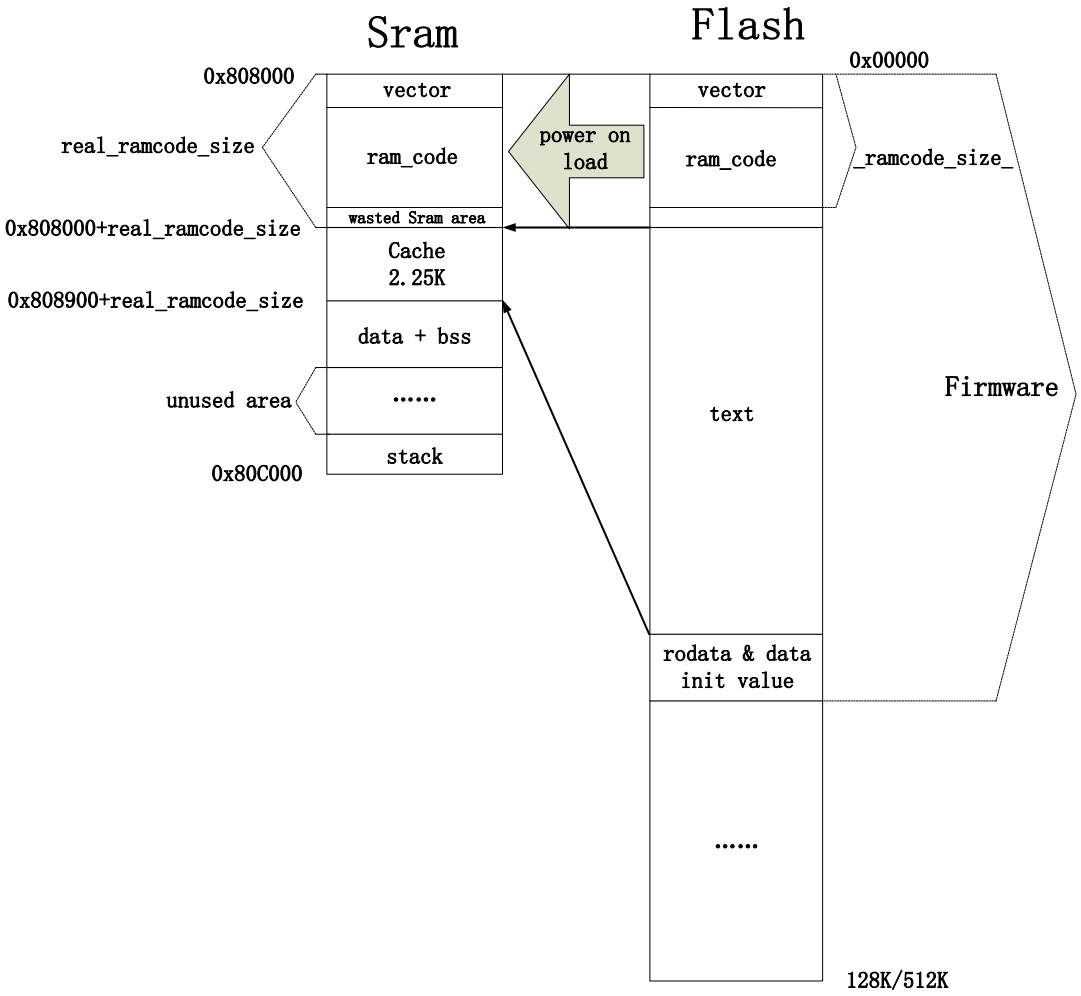


Figure 2-2 SRAM and Firmware Space

In SDK, files related to SRAM space allocation include “boot.link” and “cstartup_5316.s”.

Firmware in Flash includes vector, ramcode, text, rodata, and data initial value. SRAM includes vector, ramcode, cache, data, bss, stack and unused area. vector/ramcode in SRAM is a copy of vector/ramcode in Flash.

1) vectors, ram_code

vectors is a code section of Flash Firmware (the executable bin file generated by program compiling in SDK), and it corresponds to the assembling file “cstartup_5316.s”, i.e. the startup code “bootloader”.

ramcode is memory resident code in Flash Firmware, and it corresponds to all functions with keyword “_attribute_ram_code_” in SDK, such as function Flash erase.

```
_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 must be memory resident. If these functions are placed in Flash, it would cause timing conflict and system crash.

✧ Whenever functions resident in RAM are called, 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. Users can set a function as memory resident by adding the keyword "_attribute_ram_code_" as function flash_erase_sector above. After compiling, users can find this function in ramcode section of list files.

The vector and ramcode in firmware should be loaded to RAM when MCU powers on. After compiling, the total size of the two parts is "_ramcode_size_", which is a variable recognizable by compiler. Its calculation is implemented in "boot.link". As shown below, the compiling result "_ramcode_size_" equals the code size of 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 cache. Thus, if the code to be executed is in cache, instructions can be directly read and executed from cache; otherwise the code must be read from Flash to replace the old code in cache, and then MCU reads and executes instructions from cache.

As shown in Figure 2-2, the “text” in firmware is the Flash code not placed in SRAM. According to program locality principle, this part needs to be loaded to cache for execution.

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 5316 MCU hardware, cache starting address must be 256-byte aligned, therefore, the “real_ramcode_size” is the 256-byte aligned size of “_ramcode_size_”. The starting address of cache should be:

0x808000 + real_ramcode_size

= 0x808000 + ((_ramcode_size_+255)/256)* 256

= 0x808000 + _ramcode_size_div_256_* 0x100

The starting address of cache is 256-byte aligned “0x808000 + _ramcode_size_div_256_* 0x100”, while generally “_ramcode_size_” is not 256-byte aligned. 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 SRAM space is wasted.

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 as non-ramcode is resident in SRAM.

If “_ramcode_size_” is 0x701, 255 bytes will be wasted; if “_ramcode_size_” is 0x800, no byte will be wasted. The maximum size of wasted SRAM area is 255 bytes, therefore, in program design users need to check list files to view ramcode occupation and try to avoid serious waste.

Since cache size is fixed as 2.25K (0x900), the ending address of cache should be:

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

3) data / bss

“data” in SRAM serves to store initialized global variables of a program, i.e. global variables which are non-zero initially. “bss” in SRAM serves to store uninitialized global variables of a program, i.e. global variables which are zero initially). “data” and “bss” are introduced here as one section as they are connected – “data” is followed by “bss”.

cache is followed by “data” and “bss”, The starting address of “data + bss” is the ending address of cache, i.e. “0x808900 + _ramcode_size_div_256_* 0x100”. The code in “boot.link” shown below directly defines the starting address of “data”.

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

.data :
```

The initial value of the initialized global variables in “data” is “data init value” in Firmware shown in Figure 2-2.

4) stack / unused area

“stack” in SRAM starts from the highest address 0x80C000 (default 16kB SRAM) or 0x810000 (32kB SRAM) and to low address. Its SP pointer will descend during push operation, and ascend during pop operation.

By default, the size of stack used by SDK library does not exceed 256 bytes. However, since the size of used stack depends on the deepest stack address, the stack's final size is related to users' upper-layer program design. Any case which causes deep stack, e.g. a complex recursive function is called, or a large local array variable is used in a function, will increase the final size of the stack.

When large area of SRAM is used, users need to know the size of the stack used by program. This cannot be obtained by analyzing list files. Users can only run actual product application with all of the code which may use deep stack being executed, then reset MCU and read SRAM space to determine the size of used stack.

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

"bss" ending address can be obtained via the list file, thus the maximum size for stack is determined. Users need to analyze whether this space is enough for stack usage. Please refer to section 2.1.2.2 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 of Firmware, which largely exceeds SRAM size generally. Therefore, the code needs to be executed after it is loaded into cache by cache buffer function.

6) rodata/data init value

In Firmware except "vector", "ram_code" and "text", there are "rodata" and "data initial value".

"rodata" is read-only data in firmware, i.e. variables with keyword "const", such as ATT table in Slave:

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

Users can see the "my_Attributes" is within the "rodata" by checking the 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

"5316_ble_remote" is taken as an example to illustrate SRAM and Flash address space allocation (please refer to Figure 2-2). Based on the understanding of this demo, users can analyze SRAM and Flash space allocation of their own programs.

The bin file and list file of this demo are available under the directory "SDK" -> "Demo" -> "list file analyze". Information of SRAM space allocation can be analyzed from the "5316_ble_remote.lst" file.

All screenshots herein are from files “boot.link”, “cstartup_5316.s”, “5316_ble_remote.bin” and “5316_ble_remote.lst”.

In the list file, each code of a specific function is called a “section”. The figure below shows section distribution in the list file “5316_ble_remote.lst”.

```
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     0000168c  00000100  00000100  00008100  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text         00006f30  00001790  00001790  00009790  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       00000a64  000086c0  000086c0  000106c0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00000194  0080a100  00009124  00012100  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000c98  0080a2a0  000092c4  00012294  2**4
                  ALLOC
```

Figure 2-3 Section Distribution in list File

Below lists the sections in the list file. Detailed introductions are followed.

1) vectors: start from Flash 0, size is 0x100.
2) ram_code: start from Flash 0x100, size is 0x168c.
3) text: start from Flash 0x1790, size is 0x6f30.
4) rodata: start from Flash 0x86c0, size is 0xa64.
5) data: start from SRAM 0x80a100, size is 0x194.
6) bss: start from SRAM 0x80a2a0, size is 0xc98. By calculation “bss” ending address is 0x80a2a0 + 0xc98 = 0x80af38. The remaining space size following the “bss” is 0x80c000 – 0x80af38 = 0x10c8 = 4296 bytes, from which minus 256 bytes for stack, the remaining 4040 bytes are unused.

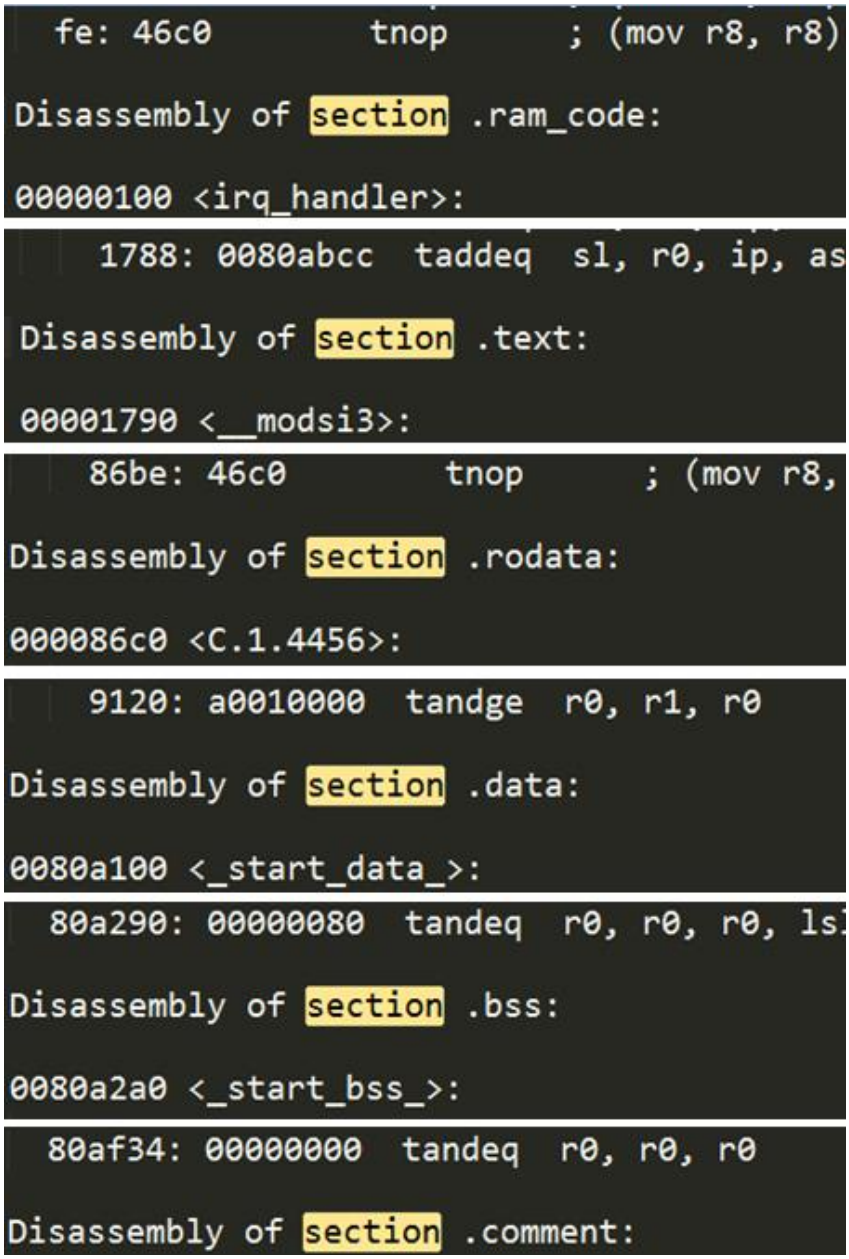


Figure 2-4 Section Address in list File

Figure 2-4 shows the starting/ending addresses of various sections by searching "section" in the list file. From this figure and Figure 2-3 Section Distribution in list File, the analysis is shown as below:

1) vector

"vector" is the bootloader corresponding to the assembly file "startup_5316.s". As shown in the list file, the size of this section is 256 bytes with starting address in Flash being 0, ending address 0xff. After power on it is loaded to SRAM and the corresponding address in SRAM is 0x808000 ~ 0x8080ff.

2) ram_code

"ram_code" section contains 0x168c bytes with starting address being 0x100, ending address 0x178c. Since "_ramcode_size_" is 0x178c, 256-byte aligned "real_ramcode_size" is 0x1800, actually 116 bytes (0x38) in SRAM are wasted.

3) cache

The starting address and ending address of cache are:

0x808000 + real_ramcode_size ~ 0x808900 + real_ramcode_size

0x809800 ~ 0x80a100

cache related information is not shown in the list file.

4) text

“text” section contains 0x6f30 bytes (size = 0x86c0 – 0x1790) with starting address being 0x1790 (ending address of “ram_code”), ending address 0x86c0, the same as what shown in Figure 2-3 above.

5) rodata

The starting address of “rodata” is the ending address of “text” 0x86c0 and the ending address is 0x9124.

As shown in “5316_ble_remote.bin”, the actual bin size is 0x92c4. According to the analysis above, the remaining firmware space 0x9124 ~ 0x92c4 is actually “data init value”, i.e. initial values of initialized global variables in Firmware. “data init value” is not a specific section in the list file. Users can search the keyword “_dstored_” and find the value “0x9124” which is the starting address of “data init value”.

00009124 g *ABS* 00000000 _dstored_

The “_dstored_” definition in the “boot.link is shown below. It tells the compiler that initial values of initialized global variables in the “data” section are all stored in “_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

The starting address of “data” is the ending address of cache 0x80a100. The size of “data” shown in Figure 2-3 is 0x194.

The final variable in “data” section is “ota_firmware_size_k”, a int variable. Its address is 0x80a290 and its size is 4-byte. Therefore the ending address of “data” is 0x80a294, and the size of “data” is 0x80a294 - 0x80a100 = 0x194, as shown in Figure 2-3.

7) bss

“data” is followed by “bss”. Since the first array “_start_bss_” should be 16-byte aligned, the “bss” section starts from 0x80a2a0, and its size is 0xc98, as shown in Figure 2-3.

The final variable in “bss” is “blt_ota_start_tick”, a int variable. Its address is 0x80af34, and its size is 4-byte. Therefore the ending address of “bss” is 0x80af38, and the size of the “bss” is 0x80af38 - 0x80a2a0 = 0xc98, as shown in Figure 2-3.

By calculation the remaining SRAM space size is 0x80c000 – 0x80af38 = 0x10c8 = 4296 bytes, from which minus 256 bytes for stack, the remaining 4040 bytes are unused.

2.1.3 MCU Address Space Access

MCU address space 0x000000 ~ 0xffffff can be accessed in the program in the following two cases.

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 the program, 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 see “drivers/5316/bsp.h” for details.

Please note that for operations such as write_reg8 (0x8000), read_reg16 (0x8000) , whose definitions are shown as below, the base address “0x800000” is automatically added (address line BIT(23) is 1) to ensure the access space is Register/SRAM rather than Flash.

```
#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 one thing of 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. The 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

Reading or writing Flash space is implemented via functions “flash_read_page” and “flash_write_page”. Code of Flash erasing is available in “drivers/5316/flash.c” and “flash.h”.

1) Flash read/write

Functions “flash_read_page” and “flash_write_page” serve to read or write 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 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 0x11000 in Flash into a data array
```

Flash write 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
```

“flash_write_page” accesses pages in Flash. The maximum “len” for operations of “flash_write_page” is 256 bytes, the size of one page. This function is not allowed to write Flash space across two or more pages.

- ✧ If the “addr” is the starting address of one page, the “len” cannot exceed 256 bytes. flash_write_page (0x12000, 256, data) is correct. flash_write_page (0x12000, 257, data) is incorrect as the final byte does not belong to the page where 0x12000 is, and the write will fail.
- ✧ 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, flash_write_page (0x120f0, 20, data) is incorrect as the first 16 bytes are in page of 0x12000 while the last 4 bytes are in page of 0x12100.

“flash_read_page” can read data more than 256 bytes once. It’s allowed to read Flash area across pages.

2) Flash erase operation

Use function “flash_erase_sector” 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 every time the function erases a complete sector.

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

3) Influence of flash access/erasing operations to system interrupt

System interrupt must be disabled via “irq_disable()” when flash_read_page, flash_write_page or flash_erase_sector is executed, and then restored via “irq_restore()” after operations are 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 section 3.2.4.4 Conn State Slave Role Timing Protection should be adopted.

4) Read Flash via pointer

Firmware of 5316 BLE SDK is stored in Flash. When the firmware is running, only former part of the code in Flash is stored and executed as memory resident code 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.

The 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 SDK:

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

Flash can be read by using function “flash_read_page” or pointer, but it can be written via function “flash_write_page” only. Pointer does not support Flash writing operation.

Please note that when Flash is read by 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 if the data is not covered by other data and new request of accessing the data is received. If a user’s code is shown as below:

```
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 is 0xff; then 0x5A is written into Flash 0x40000 by the following writing; in theory, the result of the second reading 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 is the new value written into this address rather than the old value in the cache.

```
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 the basic unit to store information as Flash erases information based on a sector. (Erase function “flash_erase_sector”). In theory, information of the same type should be stored in a sector, and information of different types should be stored in different sectors to avoid unexpected erasing. It’s recommended to follow this rule to store customized information in Flash.

Two allocation methods of Flash space are supported according to actual Flash sizes: one is for 512kB Flash (TLSR8232F512), the other is for 128kB Flash (TLSR8232F128).

2.1.4.1 Space Allocation of 512kB Flash

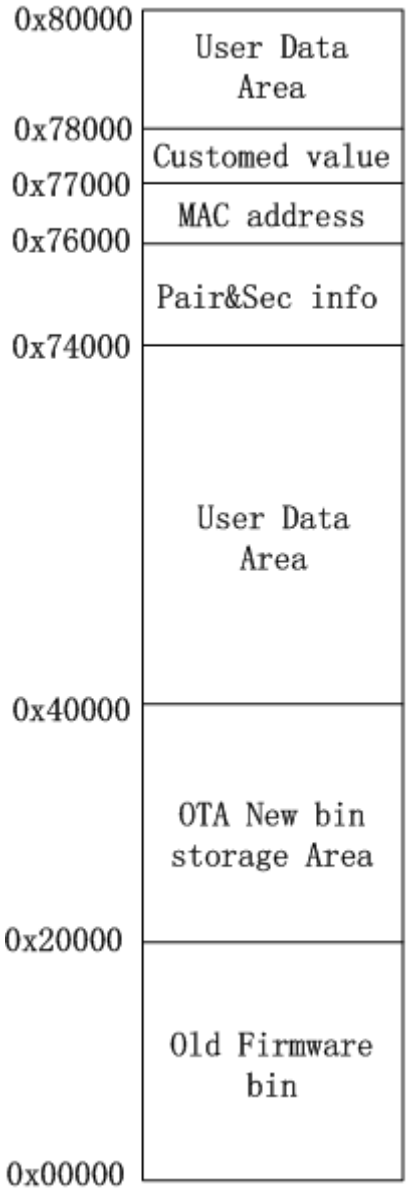


Figure 2-5 512kB FLASH Space Allocation

The figure above shows the default FLASH space allocation for TLSR8232F512 IC. Corresponding interfaces are provided to users for modifying Flash space allocation. The following introduces the default address space allocation and corresponding interfaces.

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 to 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 users want to modify this starting

address to store MAC address, please 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 can be modified in the “drivers/5316/flash.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 storing information of different 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 is burned to corresponding addresses by jig, it can be stored in the same sector; when firmware is running, the calibration information is read only and not allowed to be written or erased.
   1) The first 64-byte unit serves to store frequency offset calibration information. Actually this calibration value has only 1 byte and is stored in 0x77000.
   2) The second 64-byte unit serves to store calibration value of TP value which has 2 bytes (TP0, TP1) and is stored in 0x77040 and 0x77041 correspondingly.
   3) The third 64-byte unit serves to store capacitance calibration value of external 32kHz crystal.

Corresponding to SDK, actual calibration values will be burned into the addresses above by Telink jig system. If users want to modify the address, please ensure the burning address of Telink jig system is also modified correspondingly . In “user_init” function of SDK, “blc_app_loadCustomizedParameters()” function will read calibration values from these addresses starting from the following macros. These macros can be modified in the “drivers/5316/flash.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
```

3. The two sectors 0x74000 ~ 0x75FFF are occupied by BLE stack system, and the 8kB area is used to store pairing and security information. Users can modify the starting address of this 8kB area by calling the function below:
   stack/ble/ble_smp.h
   void bls_smp_configParingSecurityInfoStorageAddr (int addr);

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

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

✧ The second 128kB area 0x20000 ~ 0x3FFFF is used as storage space for OTA new firmware, which means the maximum space for Firmware is 128kB.

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

5. The remaining Flash space is all used as user data area (storage space for user data).

2.1.4.2 Space Allocation of 128kB Flash

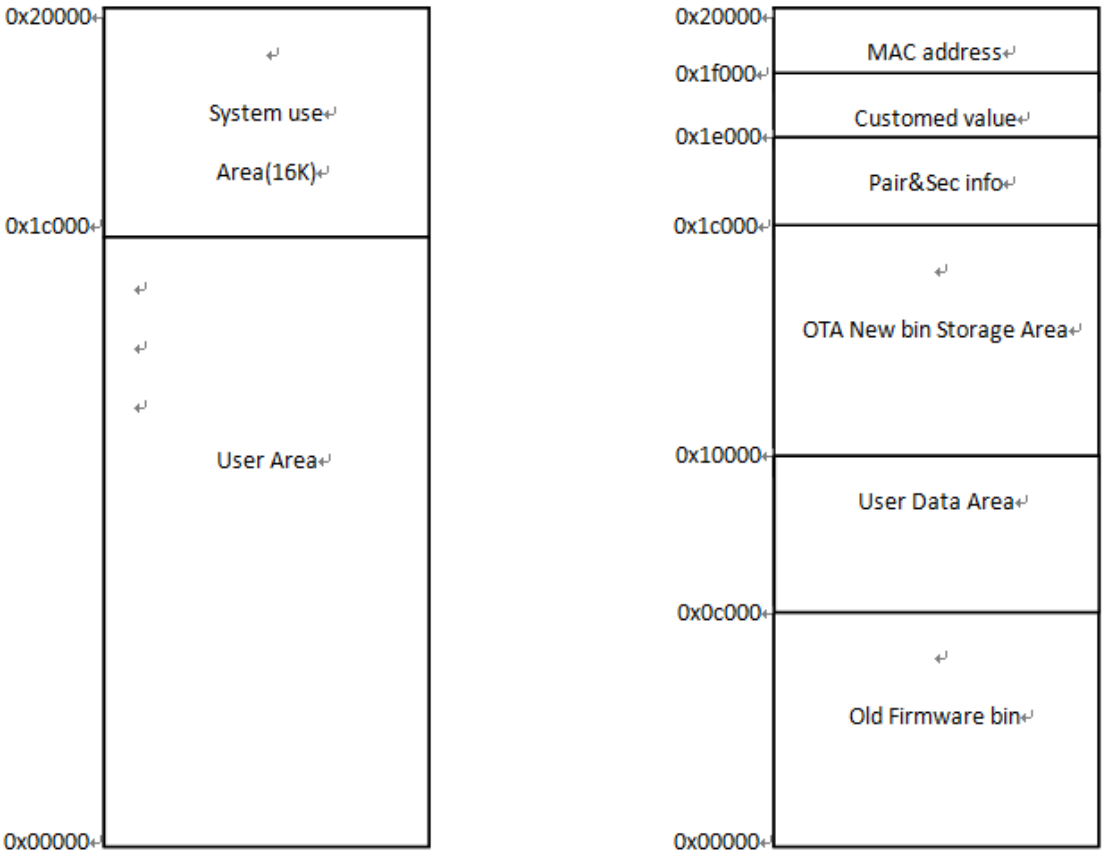


Figure 2-6 128kB Flash Space Allocation

The figure above shows the default space allocation for the 128kB Flash of TLSR8232F128. Users can choose different allocation methods as needed. Please see OTA section for allocation methods.

The space allocation shown above is the default allocation method of 128K chip. Actually all the space allocations provide users corresponding interfaces for modification, by which users can modify Flash space allocation as needed. The following introduces the default space allocation and corresponding interfaces.

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 products will be burned into its Flash starting from 0x1F000 by Telink jig system. If users want to modify this starting

address to store MAC address, please ensure the burning addresses of Telink jig system are also modified correspondingly. The “user_init” function in the SDK will read MAC address from Flash area starting from the macro “CFG_ADR_MAC”. This macro can be modified in “drivers/5316/flash.h”.

```
#ifndef         CFG_ADR_MAC
#define         CFG_ADR_MAC                     0x1F000
#endif
```

2. The sector 0x1E000~0x1EFFF serves to store customized calibration information for Telink MCU. Only this sector does not follow the rule that storing information of different 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 is burned to corresponding addresses by jig, it can be stored in the same sector; when firmware is running, these calibration information is read only and not allowed to be written or erased.
   1) The first 64-byte unit serves to store frequency offset calibration information. Actually this calibration value has only 1 byte and is stored in 0x1E000.
   2) The second 64-byte unit serves to store calibration value of TP value. Actually this calibration value has only 2 bytes (TP0, TP1) and stored in 0x1E040 and 0x1E041 correspondingly.

   Corresponding to SDK, actual calibration values will be burned into the addresses above by Telink jig system. If users want to modify the address, please ensure the burning address of Telink jig system is also modified correspondingly. In “user_init” function of SDK, “blc_app_loadCustomizedParameters()” function will read calibration values from the addresses starting from the following macros. These macros can be modified in the “drivers/5316/flash.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
```

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

   void bls_smp_configParingSecurityInfoStorageAddr (int addr);

4. The remaining 112kB space 0x00000 ~ 0x1bFFF are configurable area for user code and user data. The default allocation is as below. The 48kB area 0x00000 ~0x0BFFF is used as storage space for old firmware. The 48kB area 0x10000 ~0x1BFFF is

used as storage space for OTA new firmware. The 16kB area 0xC000 ~ 0x10000 is used as storage space for user data.

If the default space allocation does not meet users' requirements, e.g. firmware size exceeds 48kB, or user data need more than 16kB space, corresponding APIs are provided to modify allocation as needed. Please refer to section 6.1.3 for details.

2.2 Clock Module

2.2.1 System Clock & System Timer

System clock is the clock reference for MCU firmware running.

System timer, a read-only timer, is formerly used as time reference for BLE timing. For some reasons, currently we use system timer with timer0 to provide time reference for BLE timing. Therefore, users cannot use timer0 for other functions.

For last generation IC of Telink (826x), the clock for system timer is the system clock, while for TLSR8232, as the figure shown below, though system timer has multiple sources SDK divides the external 24MHz crystal oscillator by 2/3 and obtains 16M clock which will not change with the change of system clock.

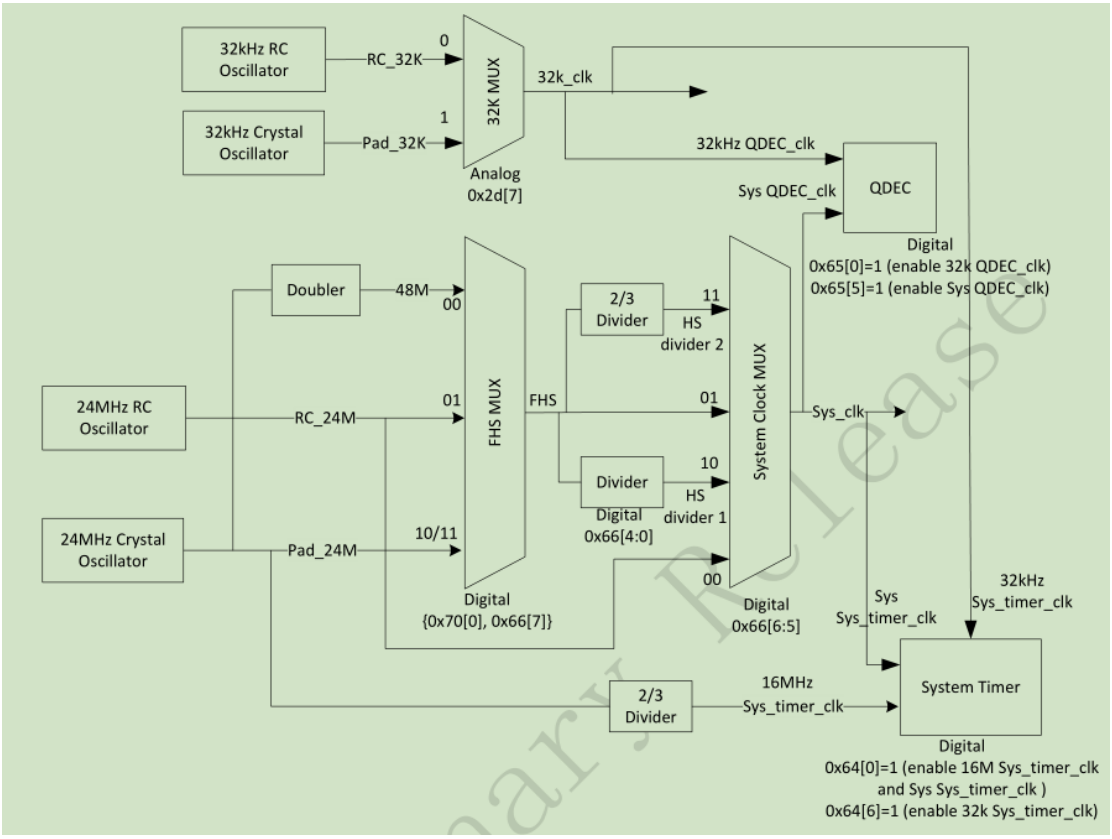


Figure 2-7 System Clock & System Timer

From the figure above, system clock can obtain 16MHz/32MHz/48MHz and other clocks in the way that doubling the external 24MHz crystal oscillator to 48M and then dividing frequency, we call these clocks crystal clocks(such as 16M crystal system clock, 32M crystal system clock); system clock can also obtain 16MHz/32MHz/48MHz and other clocks by processing internal 24MHz RC Oscillator, we call these clocks RC clocks (BLE SDK does not support RC clock.)

In BLE SDK, we recommend using crystal clock.

In initialization, call API below to configure system clock and choose the corresponding clock from the definition of enumeration variable SYS_CLK_TYPEDEF:

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

As 5316 System Timer is different from system clock, users need to know whether the clock of each MCU hardware module is derived from system clock or System Timer. Taking a system clock of 32MHz crystal for example, the system clock is 32MHz while System Timer is 16MHz.

The definitions of system clock and S, mS, uS in “app_config.h” are as follows:

```
#define CLOCK_SYS_CLOCK_HZ      16000000

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

All the clock sources are the hardware modules of system clock. When setting the clock for a module, only CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_1S can be used. In other words, if a user see that the above definitions are used for clock setting, it means the clock source of this module is system clock.

If the setting of PWM cycle and duty ratio in PWM driver is as follows, it means the clock source of PWM is system clock.

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

As System Timer is fixed 16MHz, for it SDK code uses the values below for S, mS and uS.

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

The following APIs in SDK is related to System Timer, therefore, time of these API operations is showed in a similar way as “*CLOCK_16M_SYS_TIMER_CLK_xxx*”.

```
void sleep_us(unsigned long us);
unsigned int clock_time(void);
int clock_time_exceed(unsigned int ref, unsigned int span_us);
#define WaitUs          sleep_us
#define WaitMs(t)       sleep_us((t)*1000)
```

2.2.2 System Timer Usage

After the Main function “cpu_wakeup_init” is initialized System Timer starts running, and users can read the counter value of System Timer (“System Timer tick” for short).

The 32-bit System Timer tick will increase 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.

The System Timer tick won’t stop counting during MCU running process.

The System Timer tick value can be obtained by function “clock_time()”, for instance, recording current system tick:

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

Function “clock_time()” actually reads the value counted by System Timer.

5316 BLE SDK uses System Timer tick massively to time and judge timeout. It’s highly recommended to use the System Timer tick to implement simple timing and timeout judgment.

The software timer based on query mechanism cannot ensure high real-time and accuracy. Generally it applies to applications which have not very harsh error requirement. The usage of the software timer is shown as below:

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

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

2) Continuously query if the difference between current System Timer tick and start_tick exceeds the timing value at somewhere of the firmware. If yes, the timer is triggered to execute corresponding operation, and clear timer or start a new timing cycle as needed. Suppose the timing value is 100ms, for 16MHz 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 to slove the extreme case that System Timer tick counts from 0xffffffff to 0.

   In SDK, a unified calling function to solve the u32 switching problem caused by different system clocks. No matter how many system clocks there are, the function below can be used for query:

   ```
   if( clock_time_exceed(start_tick,100 * 1000)) // unit of the second parameter is us, it’s unnecessary to worry about “16” and “32”.
   ```

   Please note that for 16M clock this function only applies to timing within 268s, if exceeds, a counter must be added to the software.

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

   ```
   u32    a_trig_tick;

   int    a_trig_flg = 0;

   while(1)

   {

       if(A){
   ```

```
            a_trig_tick = clock_time();

            a_trig_flg = 1;
        }

        if(a_trig_flg &&clock_time_exceed(a_trig_tick,2 *1000 * 1000)){

            a_trig_flg = 0;

            B();
        }
    }
```

2.3 GPIO Module

For details about GPIO module, please refer to source code in "drivers/5316/gpio_default.h", "gpio.c" and "gpio.h".

Please refer to document *Hawk_gpio_lookuptable* for understanding register operations in the code.

2.3.1 GPIO Definition

5316 IC has 23 GPIOs in three groups:

GPIO_PA0 - GPIO_PA7, GPIO_PB0 - GPIO_PB7, GPIO_PC1 - GPIO_PC7

Please note: there are 23 GPIOs in IC core, but in actual IC packages not all the GPIOs are packaged, such as the 24-pin package has part of the GPIOs. Therefore, users should refer to the actual IC package when using GPIOs.

Please follow the format above to use GPIO, see "drivers/5316/gpio.c" for details.

There is a special GPIO with SWS (Single Wire Slave) function. Its SWS function for debugging and firmware burning is enabled when power on. Generally it is not used in firmware. The SWS pin of 5316 is PC7.

2.3.2 GPIO State Control

In this section only the basic GPIO states are listed.

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

   void gpio_set_func(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func);

   "func" can be configured as "AS_GPIO" or other special functions.
2. ie: Input enable

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

   Value: 1-enable, 0-disable.
3. datai: Data input. When input is enabled for some GPIO pin, the datai value indicates its current input level.

```
static inline unsigned char gpio_read(GPIO_PinTypeDef pin);
```

Note: If GPIO input is low level, 0 is returned; if GPIO input is high level, non-zero value (**may not be 1**) is returned.
In firmware, it's recommended to invert the read values rather than using the format such as "if(gpio_read(GPIO_PA0) == 1)". Inverted values will be either 1 or 0.

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

4. oe: Output enable

```
static inline void gpio_set_output_en(GPIO_PinTypeDef pin, unsigned int value)
```

Value: 1-enable, 0-disable

5. dataO: Data output. Value: When output is enabled, "1" indicates high-level output, while "0" indicates low-level output.

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

6. Configurations for internal analog pull-up/pull-down resistor: x1 pull-up, x10 pull-down, x100 pull- up. The resistance range for x1 is 8Kohm~60Kohm; x10 80Kohm~600Kohm; x100 500Kohm~2Mohm.

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

There are four configurations for up_down.
PM_PIN_PULLUP_1M
PM_PIN_PULLUP_10K
PM_PIN_PULLDOWN_100K
PM_PIN_UP_DOWN_FLOAT
Note: PM_PIN_PULLUP_1M represents x100 pull-up; PM_PIN_PULLUP_10K represents x1 pull-up; PM_PIN_PULLDOWN_100K represents x10 pull-down.

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

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 PA0 pin as PWM function

```
gpio_set_func(GPIO_PA0, AS_PWM) ;
```

2.3.3 GPIO Initialization

The "gpio_init" function is called in main.c file to initialize states of all GPIOs. Each IO will be initialized to its default state by "gpio_init" function, unless related GPIO parameters are pre-configured in the app_config.h. The default states of 23 GPIOs are as below.

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

Please see "drivers/5316/gpio.c" and "drivers/5316/gpio_default.h" for details.

If one or multiple GPIOs are configured in app_config.h, the "gpio_init" will use the values specified in app_config.h instead of default values. The reason is that GPIO default states are all defined by macros.

Macro example:

```
#ifndef PA7_INPUT_ENABLE
#define PA7_INPUT_ENABLE    0
#endif
```

If these macros are pre-defined in app_config.h, they will not use the default values above. PA7 is taken as an example to illustrate GPIO state configuration 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/ down resistor:
   #define PULL_WAKEUP_SRC_PA7 PM_PIN_UP_DOWN_FLOAT

Conclusions for GPIO initialization:

1) Users can pre-define GPIO initial state in app_config.h, and initialize corresponding GPIO to the configured value by gpio_init;
2) Users can set the GPIO states by GPIO state control functions (gpio_set_input_en, etc.) in user_init;
3) Users can combine the two methods to configure the GPIO states.
   Please note that if the state of one GPIO is configured to different values in app_config.h and user_init, the configuration in user_init will take effect finally according to firmware timing sequence.

2.3.4 Configure SWS Pull-up to Avoid MCU Errors

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. Set ie to 1 to enable input so as to receive commands from EVK to operate MCU.
3. Both “oe” and “dataO” are set to 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.

This problem can be solved by enabling internal 1M pull-up resistor for SWS to change its float state.

For 5316, SWS is multiplexed with GPIO_PC7. Enable the 1M pull-up resistor for PC7 in the “drivers/5316/gpio_default.h”.

```
#ifndef PULL_WAKEUP_SRC_PC7

#define PULL_WAKEUP_SRC_PC7     PM_PIN_PULLUP_1M // SWS pullup

#endif
```

3. BLE Module

3.1 BLE SDK Software Architecture

3.1.1 Standard BLE SDK Architecture

Figure 3-1 shows a standard BLE SDK software architecture compliant with BLE Spec.

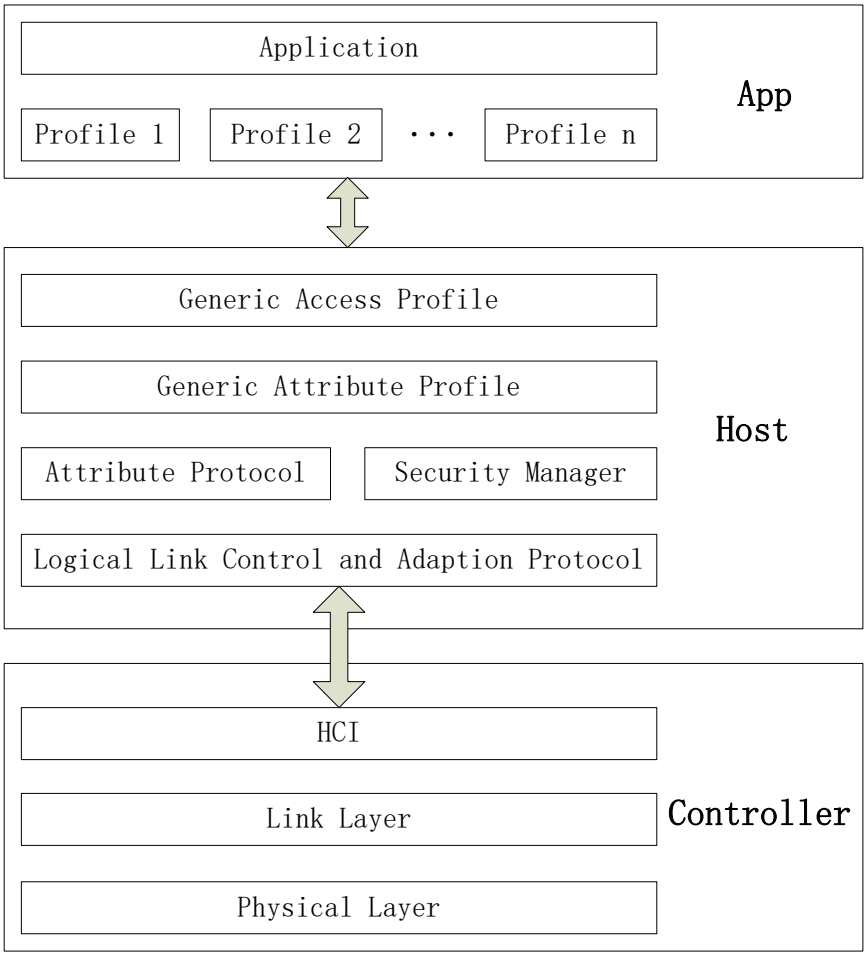


Figure 3-1 BLE SDK Standard Architecture

As shown above, BLE protocol stack includes two parts 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 code and Profiles of various Services. User controls and accesses Host via “GAP”.

Host transfers data with Controller via “HCI”. See below.

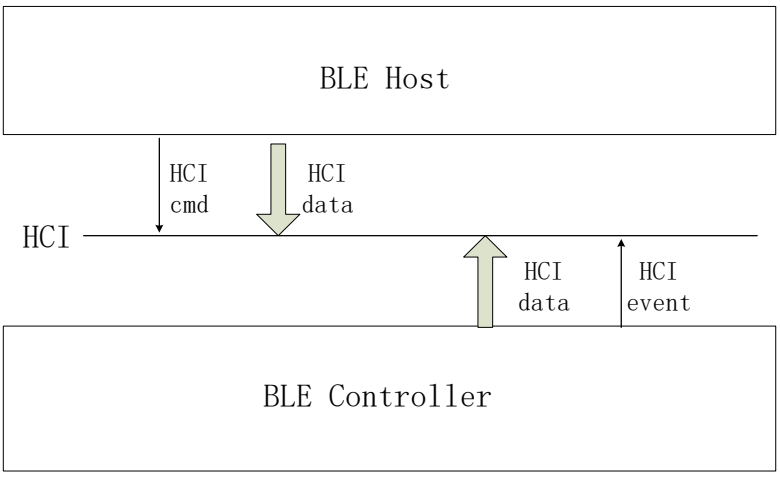


Figure 3-2 HCI Data Transfer Between Host and Controller

1) BLE Host will use HCI cmd to operate and set Controller. Controller API corresponding to each HCI cmd will be introduced in section 3.2.8.
2) Controller will report various HCI events to Host via HCI.
3) Host will send target data to Controller via HCI, while Controller will directly load data to Physical Layer for transfer.
4) When Controller receives RF data in Physical Layer, it will first check whether the data belong to Link Layer or Host, and then process correspondingly: If the data belong to LL, the data will be processed directly; if the data belong to Host, the data will be sent to Host via HCI.

3.1.2 Telink BLE SDK Architecture

3.1.2.1 Telink BLE Controller

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

Telink BLE SDK contains five standard states of Link Layer (standby, advertising, scanning, initiating, and connection), and supports slave role only in connection state.

In SDK, 5316 hci is a Controller of BLE Slave, to form a standard BLE Slave system, another MCU running BLE Host is needed.

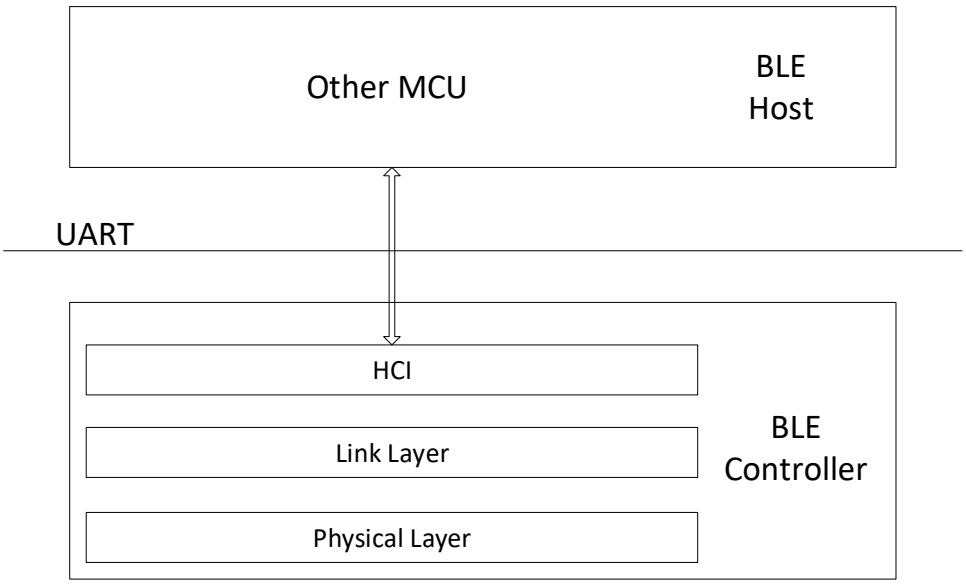


Figure 3-3 5316 hci Architecture

3.1.2.2 5316 BLE Slave

5316 BLE SDK in BLE Host fully supports stack of Slave.

When users only need to use standard BLE Slave, and 5316 BLE SDK runs Host (Slave part) + standard Controller, the actual stack architecture will be simplified based on the standard architecture, so as to minimize system resource consumption of the whole SDK (including SRAM, running time, power consumption, and etc.). Following shows Telink BLE Slave architecture. In SDK, 5316 ble remote and 5316 module are both based on this architecture.

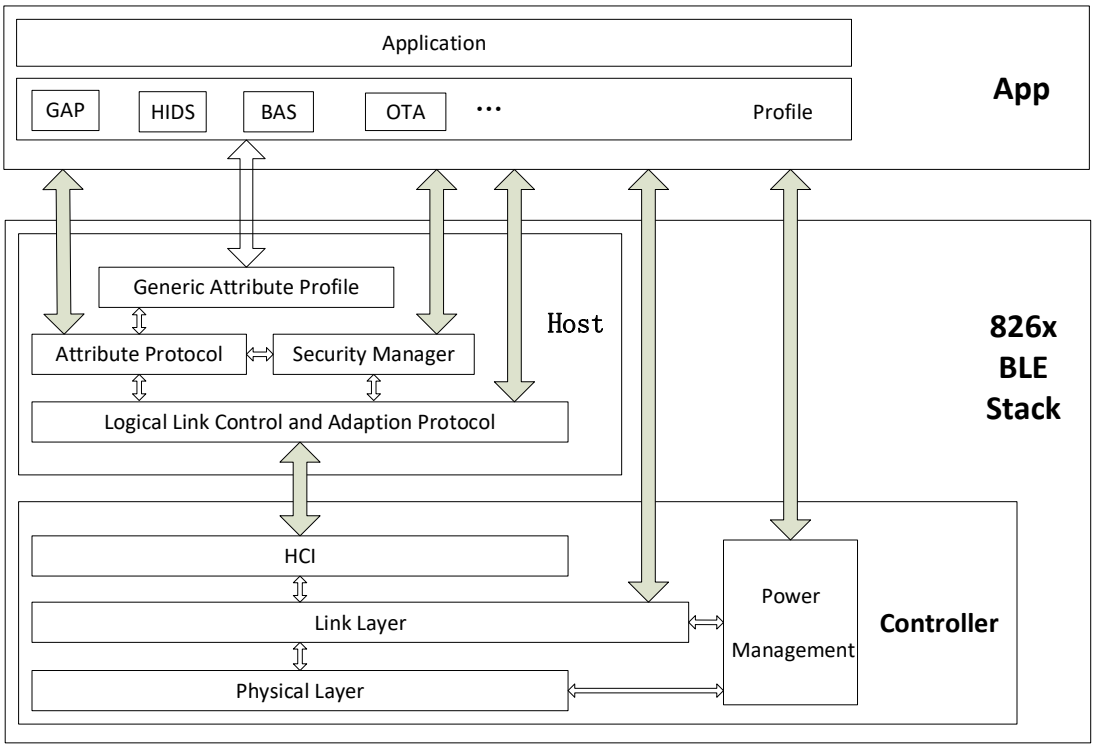


Figure 3-4 Telink BLE Slave Architecture

Figure 3-4, solid arrows indicate data transfer controllable via user APIs, while hollow arrows indicate data transfer within the protocol stack which users are unable to involve in.

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 interfaces.

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, and OTA are provided in 5316 BLE SDK demo code.

Based on this architecture the following provides a basic introduction of 5316 BLE protocol stack and user APIs of each layer.

Physical Layer is totally controlled by Link Layer, and it does not involve any other layers.

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. Users only need to register HCI data callback processing function in the L2CAP layer.

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 (the rf_drv.c file in corresponding 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

The figure below shows Link Layer state machine in BLE Spec. Please refer to *Core_v5.0* (Vol 6/Part B/1.1 “LINK LAYER STATES”) for more information.

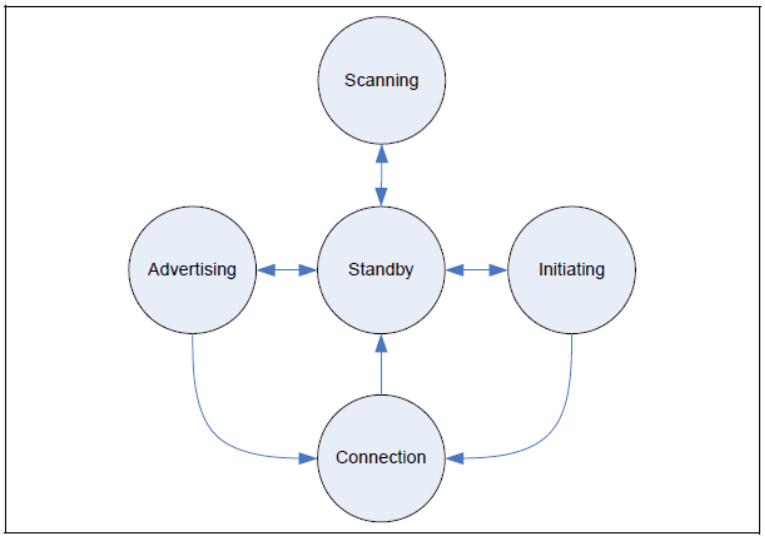


Figure 3-5 State Diagram of Link Layer State Machine in BLE Spec

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

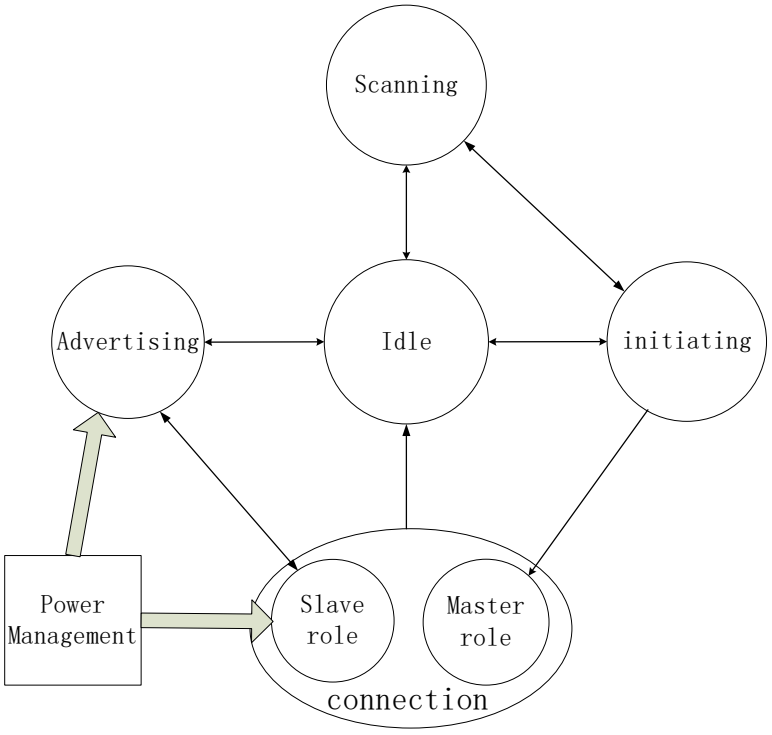


Figure 3-6 Telink Link Layer State Machine

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 only, no Master Role.

Slave Role is single connection by default.

In this document, Slave Role will be called as “Conn state Slave role” or “ConnSlaveRole/Connection Slave Role”, or “ConnSlaveRole” for short.

"Power Management" in Figure 3-6 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 "stack/ble/ll/ll.h". The state name "ConnSlaveRole" is "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, users 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, users need to add Advertising module and ConnSlaveRole, while the remaining Scanning/Initiating modules are not included so as to save code size and ramcode. The code of unused states won't be compiled.

The API adding to the basic Idle module is as follows. This API is necessary, since all BLE applications need initialization.

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

Initialization APIs of modules corresponding to the other states (Advertising, Initiating, Slave Role) are as follows.

```
void        blc_ll_initAdvertising_module(u8 *public_adr);

void        blc_ll_initSlaveRole_module(void);
```

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

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

3.2.3.2 Idle + Advertising

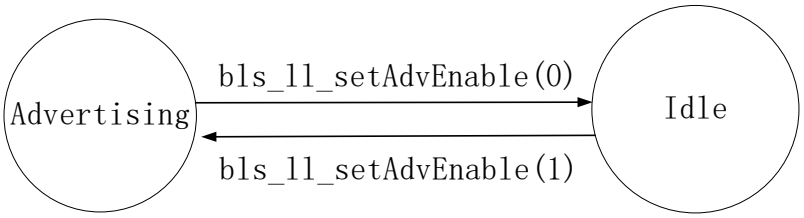


Figure 3-7 Idle + Advertising

As shown above, only Idle module 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.

The module initialization code of Link Layer state machine is:

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

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

3.2.3.3 Idle + Advertising + ConnSlaveRole

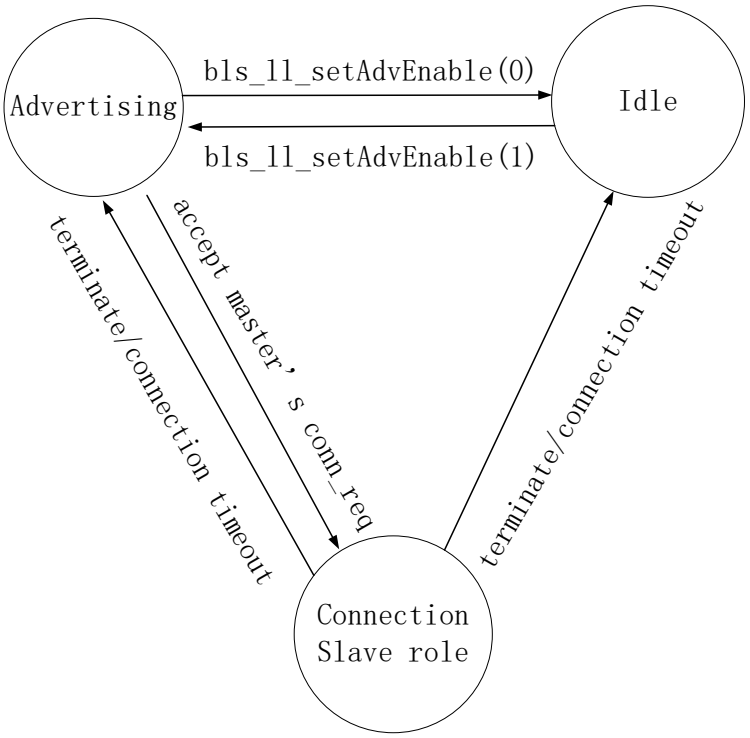


Figure 3-8 BLE Slave LL State

The figure above shows a Link Layer state machine combination for a basic BLE Slave application. In SDK, 5316 hci/5316 remote/5316 module/5316 sample/5316 dual mode are all based on this combination.

The module initialization code of Link Layer state machine is:

```
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, 5316 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 Adv needs to be enabled 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 connection request if it receives adv packet. After Link Layer receives this connection request, it will respond, establish connection and enter ConnSlaveRole.
4) When Link Layer is in ConnSlaveRole, it will return to Idle State or Advertising state in any of the following cases:
   a) Master sends “terminate” command to Slave and requests disconnection. Slave will exit ConnSlaveRole after it receives this command.
   b) By sending “terminate” command to Master, Slave actively terminates the connection and exits ConnSlaveRole.
   c) If Slave fails to receive any packet due to Slave RF Rx 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 calling 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.

3.2.4 Link Layer Timing Sequence

In this section, Link Layer timing sequence in various states will be illustrated combining with irq_handler and mainloop of 5316 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 /////////////////////////////
    ……
}
```

Function “blt_sdk_main_loop” 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, function “blt_sdk_main_loop” doesn’t work 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

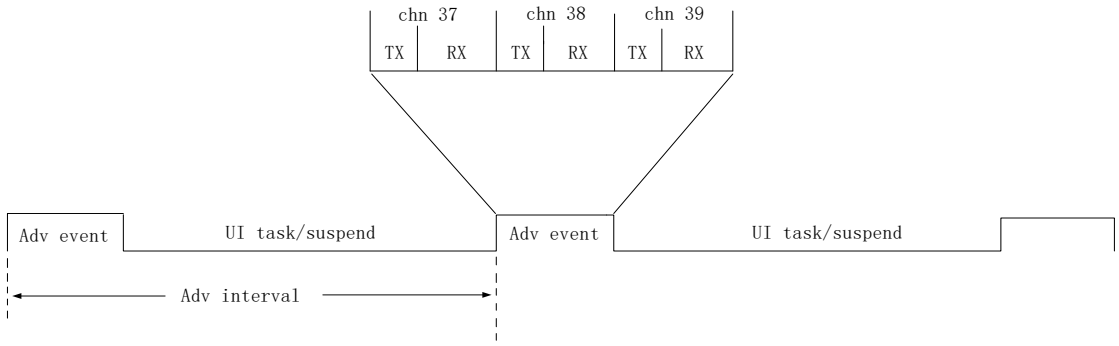


Figure 3-9 Timing Sequence in Advertising State

As shown in Figure 3-9, 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.

It should be noted that the code of adv event is executed in function “blt_sdk_main_loop” of mainloop. Therefore, adv event will not 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 Figure 3-9. 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, function “blt_sdk_main_loop” does not have much tasks to process, only needs to trigger some callback events related to Adv, including BLT_EV_FLAG_ADV_DURATION_TIMEOUT, BLT_EV_FLAG_CONNECT, etc.

3.2.4.3 Timing Sequence in Conn state Slave Role

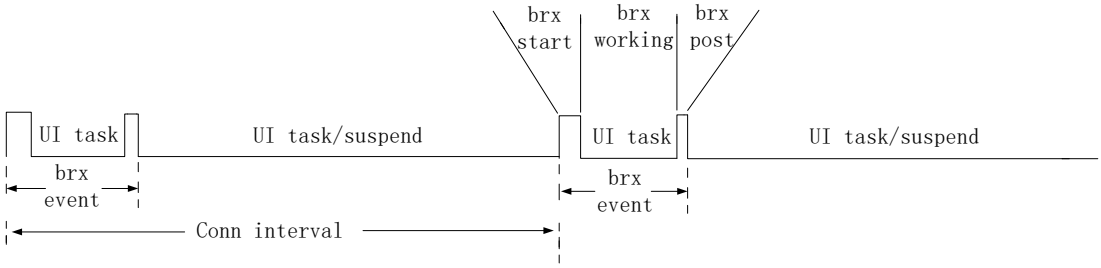


Figure 3-10 Timing Sequence in Conn state Slave Role

As shown in Figure 3-10, 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 5316 BLE SDK, each brx process has three phases.

1) brx start phase
   When Master needs to send packet, an interrupt is triggered by Timer0 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. Timer0 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 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). Function “blt_sdk_main_loop” will check if there are data in software RX fifo, and process the detected data packets correspondingly. The processing includes:

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.

3.2.4.4 Conn State Slave Role Timing Protection

In ConnSlaveRole state, each interval contains a Brx Event to transfer BLE RF packets. In 5316 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 provided in 5316 SDK. Users 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);
```

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 return 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 called 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 called to disable Brx Event. After the task is finished, the API “bls_ll_restoreConnBrxEvent” should be called 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 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.

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. Users 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 receive multiple packets; if RX FIFO number is set as 4, there may be five or six packets in an interval (e.g. OTA,), 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.

Take RX overflow for example, if:

1) The number RX FIFOs is 8;
2) The read pointer and write pointer is 0 and 2 respectively before brx_event(n) is enabled;
3) main_loop is blocked by task in brx_event(n) and brx_event(n+1) and does not process RX FIFOs in time;
4) Multiple packets in two brx_event.

As per the description in section 3.2.4.3, BLE packets received during brx_working phase will only be copied into RX FIFO (RX FIFO write pointer ++), and the operations to fetch and process data from RX FIFO are implemented in main_loop (RX FIFO read pointer ++). As shown below, the sixth packet will cover the area of rptr (read pointer) 0. It should be noted that during brx working phase, UI task occupies time slots other than interrupt handler time for RX, TX, Timer0, etc.

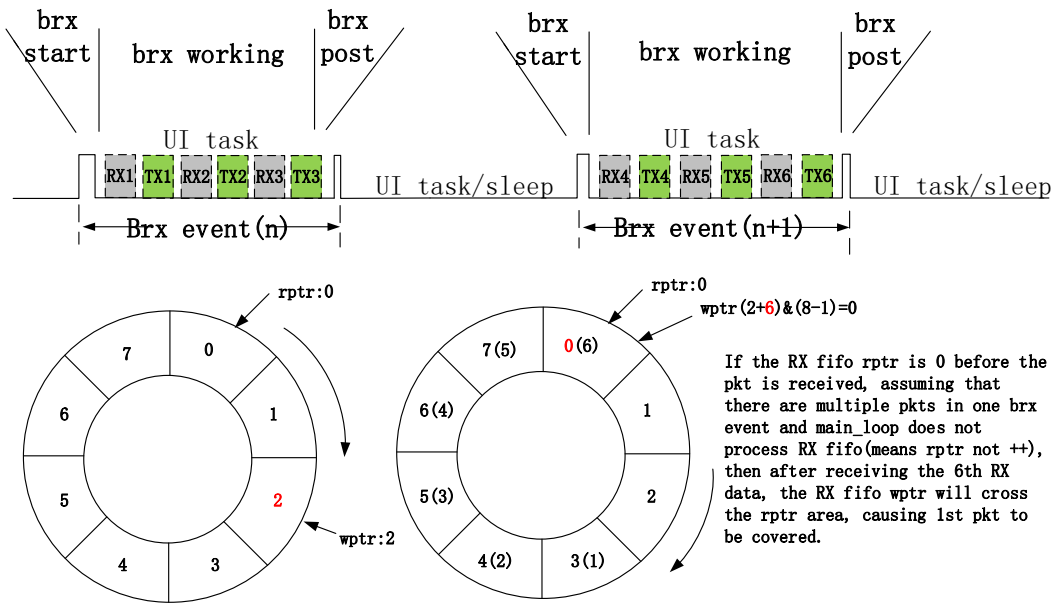


Figure 3-11 RX Overflow Case 1

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

During one brx_event, since master writes multiple packets into slave, like 7 or 8 packets, slave fails to process these packets in time. As shown below, the rptr (read pointer) is increased by two, but the wptr (write pointer) is also increased by eight, which thus causes data overflow.

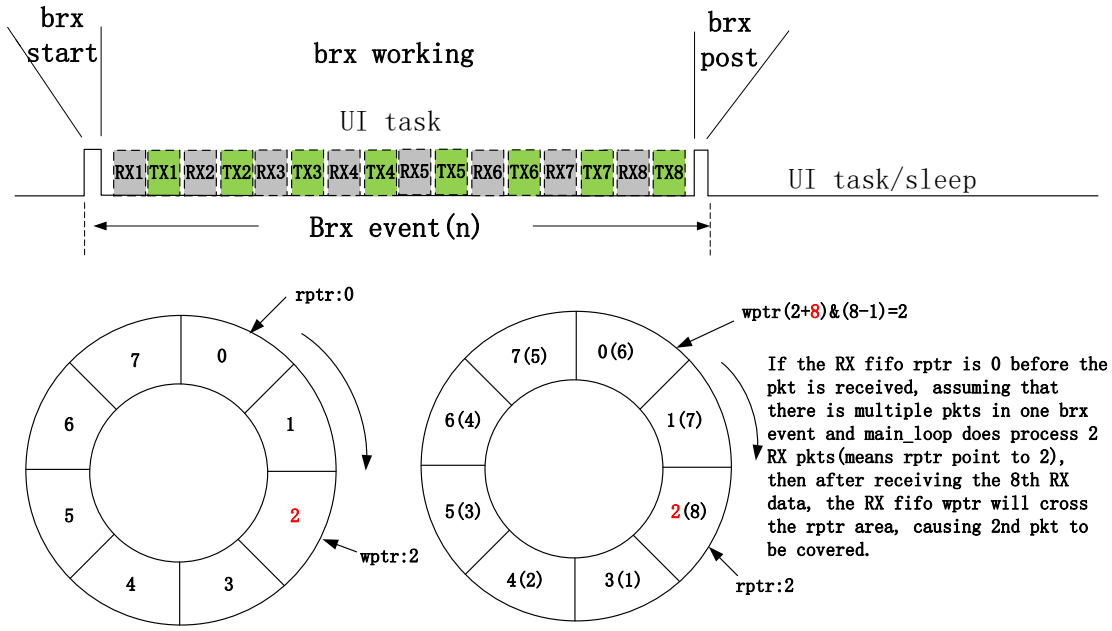


Figure 3-12 RX Overflow Case 2

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

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

Considering this, Rx overflow check is added in the new SDK: During the brx/btx event Rx IRQ, check whether the difference of current RX FIFO wptr and rptr exceeds Rx FIFO number. If the check result shows current Rx FIFO is fully occupied, the RF won't send ACK to the peer and BLE protocol ensures data will be re-transmitted.

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

Default TX FIFO number is 16, which is enough to process common audio remote control function with large data volume. User can modify this number 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. call "bls_att_pushNotifyData"), users should check current number of TX FIFOs available for Link Layer.

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

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

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

3.2.6 Controller HCI Event

Considering users may need to record and process some key actions of BLE stack bottom layer in APP layer, 5316 BLE SDK provides two types of events: standard HCI event defined by BLE Controller; Telink defined event.

HCI event is designed as BLE Spec standard, while Telink defined event only applies to BLE slave (5316 ble remote/5316 module/5316 sample/5316 dual mode, etc.), which means for BLE slaves HCI event and Telink defined event both works.

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

Users can select one set or use both at the same time as needed. In 5316 BLE SDK, 5316 ble remote, 5316 module, etc. use Telink defined event, while 5316 hci uses Controller HCI event.

As the “Host + Controller” architecture shown below, Controller HCI event reports all events of Controller to Host via HCI.

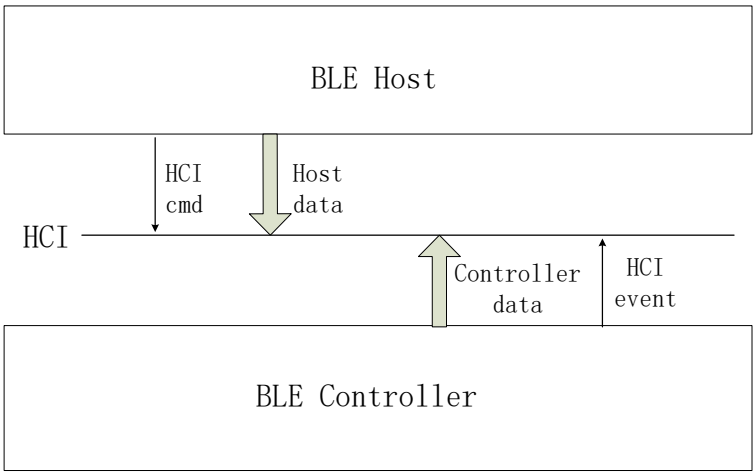


Figure 3-13 HCI Event

For the definition of Controller HCI event, please refer to *Core_v5.0* (Vol 2/Part E/7.7 “Events”). “LE Meta Event” in 7.7.65 refers to HCI LE (low energy) Event, while others are common HCI events. As defined in the Spec, 5316 BLE SDK also divides Controller HCI event into two types: HCI Event and HCI LE event. Since 5316 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, users 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 is used frequently in bottom-layer stack, and users only need 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 the 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. Users can refer to usage in 5316 module as well as the implementation of “controller_event_handler” function.

```
blc_hci_registerControllerEventHandler(controller_event_handler);
```

3.2.6.1 HCI Event

T5316 BLE SDK supports a few HCI events. The following lists some events users would like to know.

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

1) HCI_EVT_DISCONNECTION_COMPLETE
   Please refer to *Core_v5.0* (Vol 2/Part E/7.7.5 “Disconnection Complete Event”).
   Total data length of this event is 7, and 1-byte “param len” is 4, as shown below.
   Please refer to BLE Spec for data definition.

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

Figure 3-14 Disconnection Complete Event

2) HCI_EVT_ENCRYPTION_CHANGE and CI_EVT_ENCRYPTION_KEY_REFRESH
   Please refer to *Core_v5.0* (Vol 2/Part E/7.7.8 & 7.7.39).
   The two events are related to Controller encryption, and the processing is assembled in library.
3) HCI_EVT_READ_REMOTE_VER_INFO_COMPLETE
   Please refer to *Core_v5.0* (Vol 2/Part E/7.7.12).
   When Host uses “HCI_CMD_READ_REMOTE_VER_INFO” command to exchange version information between Controller and BLE peer device, and version of peer device is received, this event will be reported to Host.
   Total data length of this event is 11, and 1-byte “param len” is 8, as shown below.
   Please refer to BLE Spec for data definition.

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

Figure 3-15 Read Remote Version Information Complete Event

4) HCI_EVT_LE_META
It indicates the current event is HCI LE event, and event types can be checked according to sub event code.

Except for HCI_EVT_LE_META, other HCI events should use the API below to enable corresponding event masks.

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

The definitions of event masks are as follows:

```
#define HCI_EVT_MASK_DISCONNECTION_COMPLETE          0x0000000010

#define HCI_EVT_MASK_ENCRYPTION_CHANGE               0x0000000080

#define HCI_EVT_MASK_READ_REMOTE_VERSION_INFORMATION_COMPLETE
                                                      0x0000000800
```

If users do not set HCI event masks via this API, only the mask corresponding to "HCI_CMD_DISCONNECTION_COMPLETE" will be enabled in SDK by default, which means SDK only ensures the report of Controller disconnect event by default.

3.2.6.2 HCI LE Event

When the event code in HCI event is "HCI_EVT_LE_META", it indicates it's a 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
```

1) HCI_SUB_EVT_LE_CONNECTION_COMPLETE
Please refer to *Core_v5.0* (Vol 2/Part E/7.7.65.1 "LE Connection Complete Event"). When Controller Link Layer establishes connection with peer device, this event will be reported.
Total data length of this event is 22, and 1-byte "param len" is 19, as shown below. Please refer to BLE Spec for data definition.

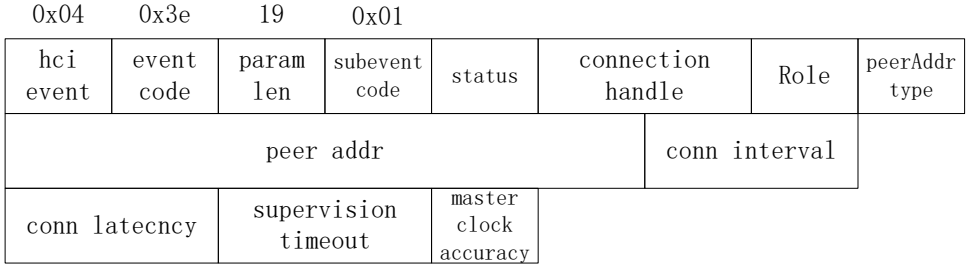


Figure 3-16 LE Connection Complete Event

2) HCI_SUB_EVT_LE_ADVERTISING_REPORT
Please refer to *Core_v5.0* (Vol 2/Part E/7.7.65.2 "LE Advertising Report Event").

When Controller Link Layer scans the right adv packet, the packet will be reported to Host via “HCI_SUB_EVT_LE_ADVERTISING_REPORT”.

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

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

Figure 3-17 LE Advertising Report Event

Note: In Telink BLE SDK, each “LE Advertising Report Event” only reports an adv packet, i.e. “i” in Figure 3-16 is 1.

3) HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE

Please refer to *Core_v5.0* (Vol 2/Part E/7.7.65.3 “LE Connection Update Complete Event”).

When “connection update” in Controller takes effect, “HCI_SUB_EVT_LE_CONNECTION_UPDATE_COMPLETE” will be reported to Host.

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

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

Figure 3-18 LE Connection Update Complete Event

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

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

The following lists some evtMask definitions. Users can view other events in “hci_const.h”.

```
#define HCI_LE_EVT_MASK_CONNECTION_COMPLETE             0x00000001
#define HCI_LE_EVT_MASK_ADVERTISING_REPORT              0x00000002
#define HCI_LE_EVT_MASK_CONNECTION_UPDATE_COMPLETE      0x00000004
```

If users do not set HCI LE event masks via this API, masks of all HCI LE events in SDK are disabled by default.

3.2.7 Telink Defined Event

Besides the standard Controller HCI event, SDK also provides Telink defined event, the architecture is shown below.

In terms of user application, events are from 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.

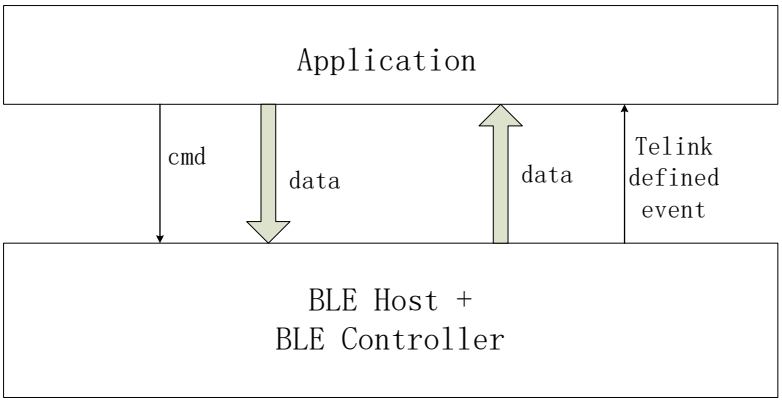


Figure 3-19 Architecture of Telink Defined Event

Up to 20 Telink defined events are supported, which are defined by using macros in “stack/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” and “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_RX_DATA_ABANDOM        16
#define          BLT_EV_FLAG_SMP_PINCODE_PROCESS    17
#define          BLT_EV_FLAG_SMP_KEY_MISSING        18
#define          BLT_EV_FLAG_PHY_UPDATE             19
```

Telink defined event is only used in BLE Slave applications (remote/module). There are two methods to call back Telink defined events in BLE slave applications.

1) The first: Registering the callback function of each event independently, which we call “independent registration.” The prototype of callback function is:

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

“e”: event number.
“p”: It’s the pointer of the data transmitted from the bottom layer when callback function is executed, and it varies with the callback functions.
“n”: length of valid data pointed by pointer.
The API for registering callback functions is:

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

2) The second: Callback functions of all the events share one entry. Whether the event responds depends on if the corresponding event mask is enabled. This we call “shared event entry”.
Registering event callback via “shared event entry” use the same API of HCI event:

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

void blc_hci_registerControllerEventHandler (
                                    hci_event_handler_t  handler);
```

Even the same register callback function of HCI event is used, the implementation of the function is slightly different.
HCI event callback function:
h = HCI_FLAG_EVENT_BT_STD | hci_event_code;
Telink defined event “shared event entry”:
H = HCI_FLAG_EVENT_TLK_MODULE | e;
“e” is the event number of Telink defined event.
“shared event entry” of Telink defined event is similar to the mask of HCI event.
Whether each event repsonds depends on the mask set by the API below:
ble_sts_t bls_hci_mod_setEventMask_cmd(u32 evtMask);
The relationship between evtMask and event number is:
evtMask = BIT(e);

The two methods to implement Telink defined event is mutually exclusive. It is recommended to use the first one “independent registration” as most SDKs use this one. Only 5316 module uses the second one “shared event entry”.

For the usage of Telink defined event, please see the demo code of project “5316_ble_remote” for method 1 “independent registration”; please see the demo code of project “5316_module” for method 2 “shared event entry”.

The implementation of the two methods are shown below by taking connect event callback and terminate event callback for example.

1) Method 1 “independent registration”

```
void task_connect (u8 e, u8 *p, int n)
{
    //add connect callback code here.
}
void task_terminate (u8 e, u8 *p, int n)
{
    //add terminate callback code here.
}
bls_app_registerEventCallback (BLT_EV_FLAG_CONNECT,
                                                &task_connect);
bls_app_registerEventCallback(BLT_EV_FLAG_TERMINATE,
                                                &ble_remote_terminate);
```

2) Method 2 “shared event entry”

```
int controller_event_handler(u32 h, u8 *para, int n)
{

    if((h&HCI_FLAG_EVENT_TLK_MODULE)!= 0)         //module event
    {
        switch(event)
        {
            case BLT_EV_FLAG_CONNECT:
            {
                bls_l2cap_requestConnParamUpdate (8, 12, 99, 400);
                spp_send_data(HCI_FLAG_EVENT_TLK_MODULE, pEvt);
            }
            break;
            case BLT_EV_FLAG_TERMINATE:
```

```
            {
                spp_send_data(HCI_FLAG_EVENT_TLK_MODULE, pEvt);
            }
            break;
        }
    }
}


    blc_hci_registerControllerEventHandler(controller_event_handler);
    bls_hci_mod_setEventMask_cmd(0xfffff);
```

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

3.2.7.1 BLT_EV_FLAG_ADV

This event is not used in current SDK.

3.2.7.2 BLT_EV_FLAG_ADV_DURATION_TIMEOUT

Event trigger condition: If the API “bls_ll_setAdvDuration” is called 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, users can implement operations such as modifying adv event type, re-enabling advertising, re-configuring advertising duration and etc.

Pointer “p”: null pointer.

Data length “n”: 0.

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

3.2.7.3 BLT_EV_FLAG_SCAN_RSP

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.

Pointer “p”: null pointer.

Data length “n”: 0.

3.2.7.4 BLT_EV_FLAG_CONNECT

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

Data length “n”: 34.

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

Figure 3-20 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 Figure 3-20) 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;
```

3.2.7.5 BLT_EV_FLAG_TERMINATE

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.

Pointer “p”: p points to an u8-type variable “terminate_reason”. This variable indicates the reason for disconnection of Link Layer.

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 Code includes HCI_ERR_REMOTE_USER_TERM_CONN (0x13), HCI_ERR_CONN_TERM_MIC_FAILURE (0x3D), and etc.
3) If Slave calls 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.

3.2.7.6 BLT_EV_FLAG_ENCRYPTION_CONN_DONE

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).

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
```

Data length “n”: 1.

3.2.7.7 BLT_EV_FLAG_DATA_LENGTH_EXCHANGE

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”.

Data length “n”: 12.

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);

3.2.7.8 BLT_EV_FLAG_GPIO_EARLY_WAKEUP

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.

Data length “n”: 1.

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 “drivers/5316/pm.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),
        STATUS_ENTER_SUSPEND  = BIT(30),
    } PM_WakeupStatusTypeDef;
```

For definitions of the parameters 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);
```

3.2.7.9 BLT_EV_FLAG_CHN_MAP_REQ

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.

Data length “n”: 5.

Pointer “p”: p points to the starting address of the following channel list array.

```
unsigned char type/ bltc.conn_chn_map[5]
```

Please note that when the callback function is executed, p points to the old channel map which is not updated.

Five bytes are used in “conn_chn_map” to represent current channel list by mapping. Each bit represents a channel:

conn_chn_map[0] bit0-bit7 represent channel0-channel7 respectively.

conn_chn_map[1] bit0-bit7 represent channel8-channel15 respectively.

conn_chn_map[2] bit0-bit7 represent channel16-channel23 respectively.

conn_chn_map[3] bit0-bit7 represent channel24-channel31 respectively.

conn_chn_map[4] bit0-bit4 represent channel32-channel36 respectively.

3.2.7.10 BLT_EV_FLAG_CHN_MAP_UPDATE

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.

Pointer “p”: p points to the starting address of the new channel map conn_chn_map[5] after update.

Data length “n”: 5.

3.2.7.11 BLT_EV_FLAG_CONN_PARA_REQ

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.

Data length “n”: 11.

Pointer “p”: p points to the 11-byte PDU of the LL_CONNECTION_UPDATE_REQ, as shown below.

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

Figure 3-21 LL_CONNECTION_UPDATE_REQ Format in BLE Stack

3.2.7.12 BLT_EV_FLAG_CONN_PARA_UPDATE

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.

Data length "n": 6.

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

3.2.7.13 BLT_EV_FLAG_SUSPEND_ENETR

Event trigger condition: When Slave executes function "blt_sdk_main_loop", this event will be triggered before Slave enters suspend.

Pointer "p": Null pointer.

Data length "n": 0.

3.2.7.14 BLT_EV_FLAG_SUSPEND_EXIT

Event trigger condition: When Slave executes function "blt_sdk_main_loop", this event will be triggered after Slave is woken up from suspend.

Pointer "p": Null pointer.

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".

3.2.7.15 BLT_EV_FLAG_PHY_UPDATE

Event trigger condition: when Salve receives LL_PHY_UPDATE_IND PDU sent by master and PHY is updated successfully, this event will be triggered.

Pointer "p": p points to the updated Tx and Rx PHY. PHY could be the enum values below:

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

Note: TLSR8232 could only use symmetric PHY, that is, Rx and Tx use the same PHY. Please see section 2M PHY Supported for usage.

3.2.8 Controller API

3.2.8.1 Controller API Brief

In standard BLE stack architecture (see Figure 3-1), APP layer cannot directly communicate with Link Layer of Controller, i.e. data of APP layer must be first transferred to Host, and then Host can transfer control command to Link Layer via HCI. All control commands from Host to LL via HCI follow the definition in BLE Spec *Core_v5.0*, please refer to *Core_v5.0* (Vol 2/Part E/ Host Controller Interface Functional Specification).

5316 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

Figure 3-4 is used in 5316 BLE SDK, APP layer can directly operate Link Layer, it still uses the standard APIs of HCI part. Corresponding Host commands in Spec are provided in the API introduction below. Users can refer to the detailed descriptions 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, there are different situations:

1) For applications such as 5316_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) The processing is different for master. As 5316 does not support master we do not provide detailed information here.

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

3.2.8.2 API Return Type ble_sts_t

An enum type “ble_sts_t” defined in “stack/ble/ble_common.h” is used as return value type for most APIs in SDK. When API calling with right parameter setting is accepted by the protocol stack, it will return BLE_SUCCESS, value being“0”; 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 APIs of Link Layer.

3.2.8.3 MAC Address Initialization

In this document, “BLE MAC address” refers to “public address” and “random static address” by default.

In 5316 BLE SDK, by calling the interface below “public address” and “random static address” will be obtained.

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

Fill “flash_addr” with the address for storing MAC address in Flash. From the content above, the address of 5316 512K Flash should be 0x76000. If “random static address” is not needed, fill “mac_random_static” with “NULL”.

The Link Layer initialization API can be called to load the obtained BLE public MAC address into BLE protocol stack.

```
blc_ll_initBasicMCU(mac_public);
```

In order to use advertising state in Link Layer state machine, MAC address needs to be loaded.

```
blc_ll_initAdvertising_module(mac_public);
```

As introduced above, the 6-byte BLE MAC address will be downloaded into specific Flash area of the actual products by Telink jig system. Users need to obtain the MAC address from Bluetooth SIG.

3.2.8.4 Link Layer State Machine Initialization

The APIs below are used to configure initialization of each module when BLE state machine is established. Please see the previous section for introduction of Link Layer state machine.

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

3.2.8.5 bls_ll_setAdvData

Please refer to *Core_v5.0* (Vol 2/Part E/ 7.8.7 “LE Set Advertising Data Command”) for details.

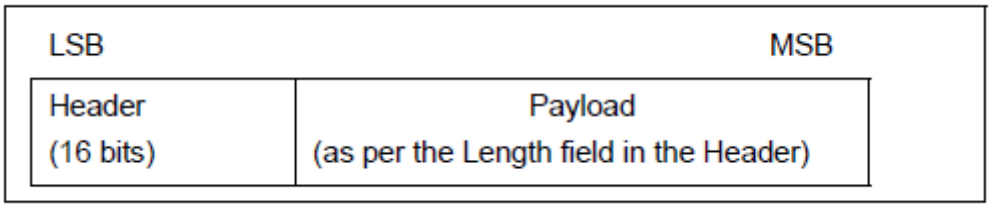


Figure 3-22 Adv Packet Format in BLE Stack

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

The API below is used to set AdvData data in Payload:

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

The “data” pointer points to the starting address of AdvData, 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 called in initialization to set adv data, or called in mainloop to modify adv data when the firmware is running.

In “5316 ble remote” project of 5316 BLE SDK, the definition of AdvData is shown as below. Please refer to “Data Type Specification” in BLE Spec *CSS v6* (Core Specification Supplement v6.0) for meanings of fields.

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

The adv device name is set as " Ghid " in the adv data shown above.

3.2.8.6 bls_ll_setScanRspData

Please refer to *Core_v5.0* (Vol 2/Part E/ 7.8.8 “LE Set Scan response Data Command”).

Similar to the setting of Adv packet Payload, an API is used to set Scan response Payload:

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

The “data” pointer points to the starting address of scanRspData in Payload, 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 called in initialization to set Scan response data, or called in mainloop to modify scan response data when the firmware is running.

In project “5316 ble remote” of 5316 BLE SDK, the definition of Scan response data is shown as below. The name of scan device is "GRemote". Please refer to “Data Type Specification” in BLE Spec *CSS v6* (Core Specification Supplement v6.0) for meanings of fields.

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

Since device names configured in advertising data and scan response data are different, the device names scanned by a mobile phone or IOS system may be different:

1) If some devices only listen for Adv packets, the scanned device name is " Ghid ".
2) If some devices send scan request after Adv packet is received, and read the scan response, the scanned device name may be " GRemote ".

Users can configure device name in the two packets (Adv packet & scan response packet) as the same one, 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.

3.2.8.7 bls_ll_setAdvParam

Please refer to *Core_v5.0* (Vol 2/Part E/ 7.8.5 "LE Set Advertising Parameters Command").

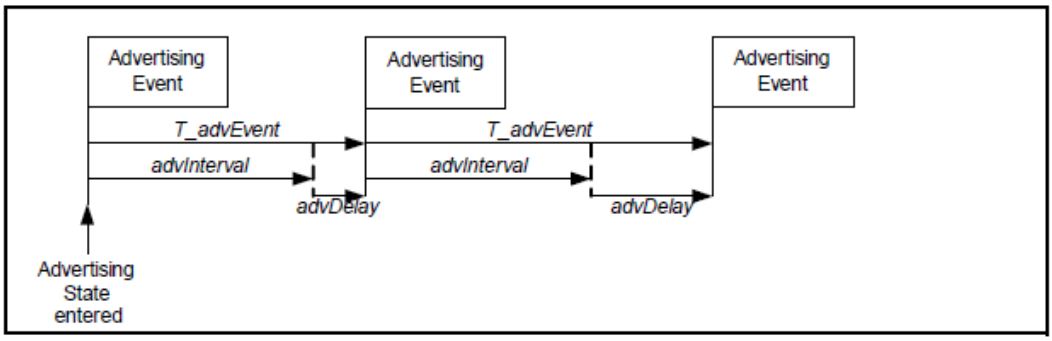


Figure 3-23 Advertising Event in BLE Stack

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, channel 38, and channel 39) respectively. The API below is used 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 and 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_v5.0* (Vol 6/Part B/ 4.4.2.2 "Advertising Events") for details.

2) advType

As specified in BLE Spec, the following four basic advertising event types are supported.

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

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

Figure 3-24 Four Adv Events in BLE Stack

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

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

Among the four advertising events "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 "stack/ble/ble_common.h":

```
/* Advertisement Type */
typedef enum{
    ADV_TYPE_CONNECTABLE_UNDIRECTED          = 0x00,  // ADV_IND
    ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY  = 0x01,
                              //ADV_INDIRECT_IND (high duty cycle)
    ADV_TYPE_SCANNABLE_UNDIRECTED          = 0x02 //ADV_SCAN_IND
```

```
    ADV_TYPE_NONCONNECTABLE_UNDIRECTED     = 0x03,
//ADV_NONCONN_IND

   ADV_TYPE_CONNECTABLE_DIRECTED_LOW_DUTY = 0x04,
                                //ADV_INDIRECT_IND (low duty cycle)

}advertising_type;
```

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

3) ownAddrType

It is used to specify MAC address type in adv packets.

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 4” to specify adv address type.

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

Only two parameters are introduced herein.

OWN_ADDRESS_PUBLIC means using public MAC address for advertising. The actual address is the set by API “API blc_ll_initAdvertising_module(u8 *public_adr)” in MAC address initialization.

OWN_ADDRESS_RANDOM means using random static MAC address for advertising. The address is set by the API below.

```
blc_ll_setRandomAddr(mac_random_static);
```

4) peerAddrType and *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” is used to set advertising channel. It can choose any one or more of channel 37, 38, 39. The value “adv_channelMap” can be the following three or any combination of the three.

```
#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 value of intervalMin or intervalMax 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 API “bls_ll_setAdvParam”. 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.

Considering users may often modify some common parameters, three independent APIs are provided, so that users can directly call one API to modify corresponding parameter(s), rather than calling 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);
```

The parameters of the three APIs are the same as “bls_ll_setAdvParam”.

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”.

3.2.8.8 bls_ll_setAdvEnable

Please refer to *Core_v5.0* (Vol 2/Part E/ 7.8.9 “LE Set Advertising Enable Command”).

```
ble_sts_t  bls_ll_setAdvEnable(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, neither enabling or disabling Advertising does not affect Link Layer state.
4) `ble_sts_t` will always return “BLE_SUCCESS”.

3.2.8.9 bls_ll_setAdvDuration

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

After “bls_ll_setAdvParam” sets all adv parameters successfully, and the “bls_ll_setAdvEnable(1)” is called to start advertising, the API “bls_ll_setAdvDuration” can be called 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”(unit: us): The “duration_us” is valid only when the “duration_en” is set to 1.

The firmware starts timing from the set timing. When this duration expires, “AdvEnable” becomes invalid, and advertising will stop. None Conn state will switch to Idle State. The Link Layer event “BLT_EV_FLAG_ADV_DURATION_TIMEOUT” will be triggered.

As specified in BLE Spec, the duration time of adv type “ADV_TYPE_CONNECTABLE_DIRECTED_HIGH_DUTY” should equal or less than 1.28s. The duration time in telink SDK is fixed as 1.28s, i.e. advertising will stop after the 1.28s duration. Therefore, for this adv type, calling “bls_ll_setAdvDuration” will be invalid.

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 users want 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 called 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.

3.2.8.10 blc_ll_setAdvCustomedChannel

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

```
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 called:

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

3.2.8.11 rf_set_power_level_index

5316 BLE SDK provides the API to set output power for BLE RF packet:

```
void    rf_set_power_level_index (int level);
```

For the setting of "level" values, the enum variable `RF_TxPowerTypeDef` in "drivers/5316/rf_drv.h" can be referred to.

The Tx power configured by this API is valid for both adv packets and conn packets, and it can be set anywhere in the firmware. The actual Tx power is subject to the latest setting.

3.2.8.12 bls_ll_terminateConnection

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

This API applies to BLE Slave device.

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 the specified reason for disconnection and it corresponds to the "ble_sts_t" defined in "ble_common.h". Please refer to *Core_v5.0* (Vol 2/Part D/2 "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 decrypted, if an authentication data MIC error is detected, the "bls_ll_terminateConnection(HCI_ERR_CONN_TERM_MIC_FAILURE)" will be called 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	
HCI_ERR_CONN_NOT_ESTABLISH	0x3E	Link Layer 处于非 Connection state Slave role 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.

3.2.8.13 Get Connection Parameters

The following APIs are used to obtain current connection parameters 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.

- ✧ API “bls_ll_getConnectionInterval” returns the current conn interval, unit being 1.25ms. Suppose current conn interval is 10ms, the return value should be 10ms/1.25ms=8.
- ✧ API “bls_ll_getConnectionLatency” returns the actual Latency value.
- ✧ API “bls_ll_getConnectionTimeout” returns the current conn timeout, unit being 10ms. Suppose current conn timeout is 1000ms, the return value would be 1000ms/10ms=100.

3.2.8.14 blc_ll_getCurrentState

The API below is used to obtain current Link Layer state.

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

The return values of this function are as below (5316 SDK does not support BLS_LINK_STATE_SCAN and BLS_LINK_STATE_INIT)

```
#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)
```

3.2.8.15 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.

3.2.8.16 Whitelist & Resolvinglist

As introduced above, “filter_policy” of Advertising involves Whitelist, and actual operations depend on devices in Whitelist. Actually Whitelist contains two parts: Whitelist and Resolvinglist.

Users 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 called 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 is for whitelist reset:

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

The return value is “BLE_SUCCESS”.

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

Add a device to whitelist and the return values are shown below:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	Add success
HCI_ERR_MEM_CAP_EXCEEDED	0x07	Whitelist is full, add failure

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

Delete a device from whitelist and 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 only in current SDK.

```
#define    MAX_WHITE_IRK_LIST_SIZE        2
```

The API below is for Resolvinglist reset:

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

Reset Resolvinglist, the return value is “BLE_SUCCESS”.

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

The API above is for address resolution. Resolvinglist must be enabled for address resolution. It can be disabled if there is no need for address resolution.

```
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”.

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

This API is used to delete a RPA device from Resolvinglist.

For usage of address filter based on Whitelist/Resolvinglist, please refer to “feature_whitelist demo” in 5316 feature test. (In order to enable this demo set “FEATURE_TEST_MODE” to TEST_WHITELIST in “appconfig.h”)

3.2.9 2M PHY Supported

2M PHY is new Link layer feature available from BLE Core 5.0. Please refer to BLE Spec *Core_v5.0* (Vol 6/Part B/ Link Layer Specification) for details.

2M PHY interacts between the master controller and slave controller via three PDUs (LL_PHY_REQ/LL_PHY_RSP/LL_PHY_UPDATE_IND) so as to set the transmission rate of RF receiver finally. 2M PHY is only available after the connection is established, it can not be used in disconnection. Both master and slave can start this process. If it is started by master, master will send “LL_PHY_REQ PDU” and slave will send “LL_PHY_RSP PDU” to respond master. During this process master and slave actually exchange the PHYs they each support and prefer, then master will send “LL_PHY_UPDATE_IND” and after “instance” reaches, master and slave will use their new PHYs to transmit and receive data. If it is started by slave, slave will send LL_PHY_REQ PDU, master will directly send “LL_PHY_UPDATE_IND to respond slave” and after “instance” reaches, master and slave will use their new PHYs to transmit and receive data after.

5316 BLE SDK supports 2M PHY. This function is enabled by default. APIs are provided by 5316 BLE SDK for using 2M PHY. The following provides detailed descriptions of each API. Please note that 5316 SDK only supports symmetric PHY which means the settings of Rx PHY and Tx PHY must be the same.

To use 2M PHY, users must initialize 2M PHY feature. The initialization function must be called to initialize 2M PHY. The prototype of the function is:

```
void blc_ll_init2MPhy_feature(void)
```

This function initializes all the parameters for 2M PHY operation.

If it is master to start PHY Update, users only need to call the initialization function above and BLE stack will do other work. If it is slave to start PHY Update, users need to call the PHY set function to set PHY. The prototype of the PHY set function is:

```
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)
```

Descriptions of function parameters are as below:

Parameter	Description

connHandle	Connect handle. Use BLS_CONN_HANDLE by default.
all_phys	Set if there are preferred Tx PHY and Rx PHY. An enum variable. *PHY_TRX_PREFER* is the common value. See the definitions of le_phy_prefer_mask_t for other values.
tx_phys	Preferred Tx PHY. An enum. Two optional values: *PHY_PREFER_1M* and *PHY_PREFER_2M*. Whether this parameter will be used depends on if there is preferred Tx PHY in all_phys.
rx_phys	Preferred Rx PHY. An enum. Two optional values: *PHY_PREFER_1M* and *PHY_PREFER_2M*. Whether this parameter will be used depends on if there is preferred Rx PHY in all_phys.

If users want to do some custom operations after PHY Update is finished, users can register callback function. The function below is used for the registration of the callback function:

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

Please see section 3.2.7 “Telink Defined Event” for the usage of this function.

3.2.10 Data Length Extension

Data Length Extension(DLE for short) is available from BLE core 4.2. For BLE versions before BLE 4.2, a BLE packet can transmit 27Byte (ATT_MTU = 23) at most, and long packets only can be transmitted in fragmented packets, thus the transmission efficiency is low and it cannot meet actual demands. With DLE data of 251B can be transmitted at once, which greatly improves the transmission efficiency and is very favorable for applications of big data transmission, such as Audio and OTA.

DLE is a feature of link layer, please refer to *Core_v5.0* (Vol 6/Part B/ Link Layer Specification) for details.

DLEs use LL_LENGTH_REQ and LL_LENGTH_RSP to exchange the largest byte numbers of transmitting and receiving they each support and finally the smallest byte numbers among them are used as the the numbers of bytes for Tx and Rx. DLE is only available in connection and unavailable in disconnection. DLE can be started by master or slave.

5316 BLE SDK supports DLE and provides interfaces for users. If DLE is started by master, users do not need to do any configuration, BLE stack will process automatically; if it is started by slave, users need to call the function below after the connection is established.

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

After DLE is finished, BLT_EV_FLAG_DATA_LENGTH_EXCHANGE will be reported. Users can register callback to complete custom operations.

The appropriate sizes of Tx FIFO and Rx FIFO must be configured when using DLE. A large size takes too much RAM space, while a small size cannot meet demands.

Users should balance on it. There are requirements for configurations of Tx FIFO and Rx FIFO. Misconfigurations will lead to unexpected errors. The following provides the configurations and examples.

If users want to set the maximum transmission data length to 251 Bytes, Tx FIFO should be set as 251 + 12 = 263 Bytes, considering 4-byte alignment, Tx FIFO should be set to 264 Bytes finally.

If users want to set the maximum receiving data length to 251 Bytes, Rx FIFO should be set as 251 + 28 = 279 Bytes，considering 16-byte alignment, Tx FIFO should be set to 288 Bytes finally.

Conclusion:

Tx FIFO size = actual maximum Payload size of transmission + 12 (4-byte aligned)

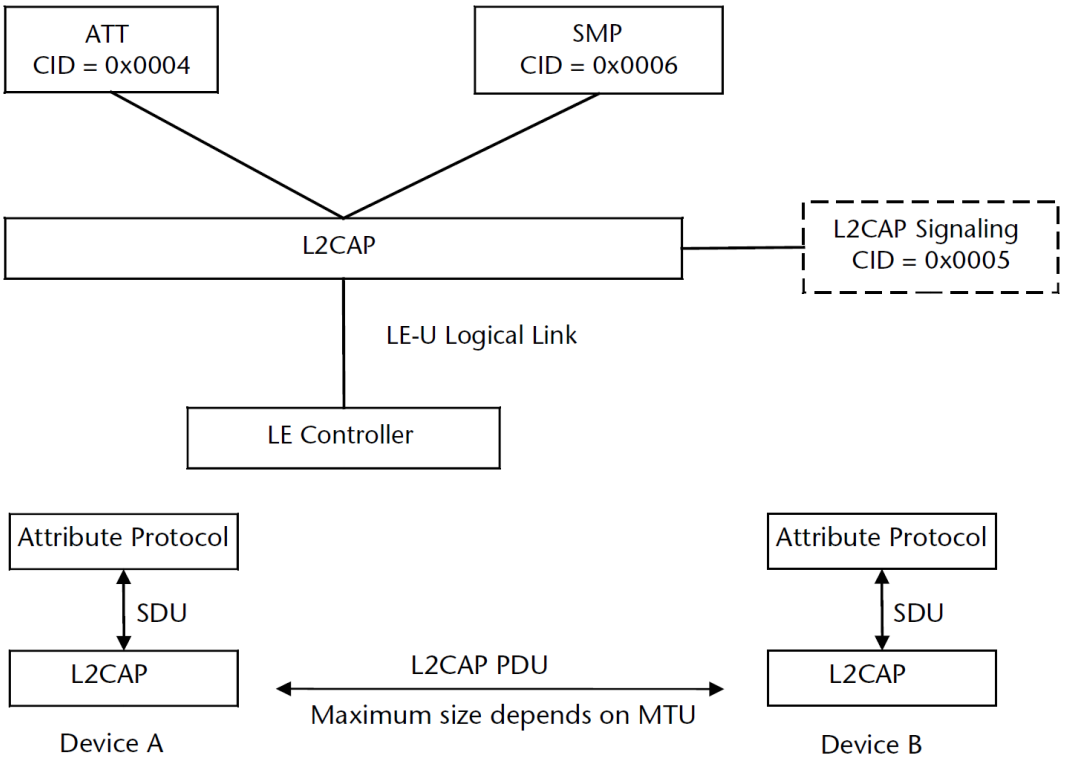
Rx FIFO size = actual maximum payload size of receiving + 28 (16-byte aligned)

3.3 L2CAP

Logical Link Control and Adaptation Protocol, L2CAP for short, connects the upper APP layer and the lower Controller layer. By acting as an adaptor between the Host and the Controller, the L2CAP makes data processing details of the Controller become negligible to the upper-layer application operations.

The L2CAP layer of BLE is a simplified version of classical Bluetooth. In basic mode, it does not implement segmentation and re-assembly, has no involvement of flow control and re-transmission, and only uses fixed channels for communication.

The figure below shows simple L2CAP structure: Data of the APP layer are sent in packets to the BLE Controller. The BLE Controller assembles the received data into different CID data and report them to the Host layer.



As specified in BLE Spec, L2CAP is mainly used for data transfer between Controller and Host. Most work is finished in stack bottom layer with little involvement of the user. Users only need to call 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 5316 remote/5316 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 5316 hci, only Slave controller is implemented. Function “blc_hci_sendACLData2Host” serves to transmit data of controller to BLE Host device via hardware interface such as UART.

```
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 command “CONNECTION PARAMETER UPDATE REQUEST” to Master in L2CAP layer. The figure below shows the command format. Please refer to *Core_v5.0* (Vol 3/Part A/ 4.20 “CONNECTION PARAMETER UPDATE REQUEST”).

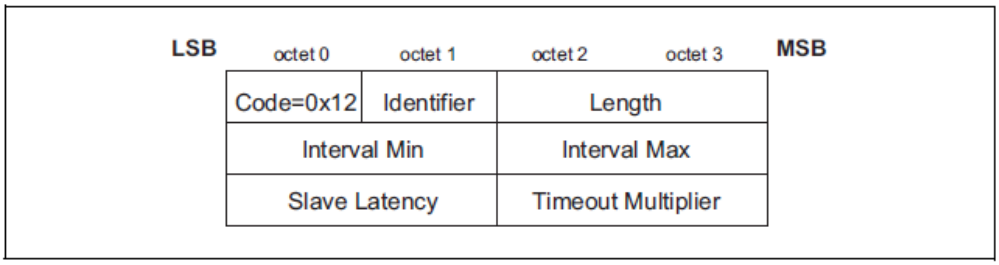


Figure 4.22: Connection Parameters Update Request Packet

Figure 3-25 Connection Para Update Req Format in BLE Stack

5316 BLE SDK provides an API in L2CAP layer for Slave to send command “CONNECTION PARAMETER UPDATE REQUEST” 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);
```

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 in a unit of 1.25ms (e.g. for 9ms connection interval, the value should be 10ms); the “timeout” is in a unit of 10ms (e.g. for 100s timeout, the value should be 1000ms).

Application example: Slave requests for new connection parameters when connection is established.

```
void task_connect (u8 e, u8 *p)
{
    //interval=7.5ms latency=99 timeout=4s
    bls_l2cap_requestConnParamUpdate (6, 6, 99, 400);
}
```

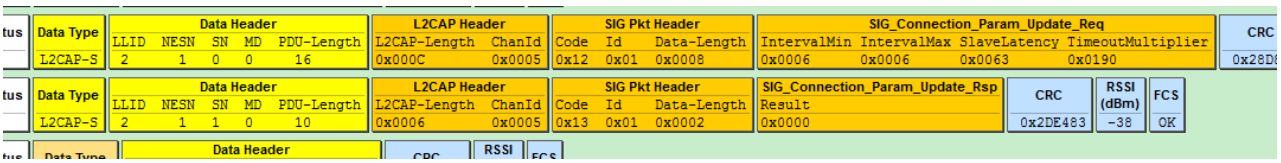


Figure 3-26 BLE Sniffer Packet Sample: conn para Update Request & Response

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 command “CONNECTION PARAMETER UPDATE RESPONSE”. Please refer to *Core_v5.0* (Vol 3/Part A/ 4.20 “CONNECTION PARAMETER UPDATE RESPONSE”).

The figure below shows the command format: if “result” is “0x0000”, it indicates the request command is accepted; if “result” is “0x0001”, it indicates the request command is rejected. Whether actual Android/iOS device will accept or reject the connection parameter update request is determined by corresponding BLE Master. User can refer to Master compatibility test.

As shown in Figure 3-26, Master accepts the request.

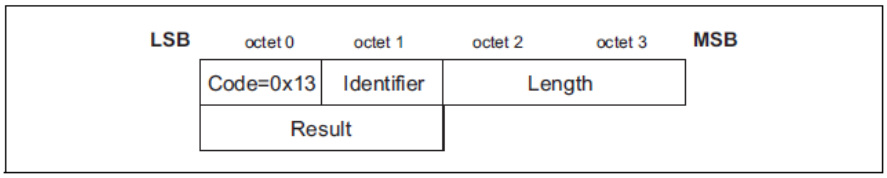


Figure 4.23: Connection Parameters Update Response Packet

The data field is:

- *Result (2 octets)*

  The result field indicates the response to the Connection Parameter Update Request. The result value of 0x0000 indicates that the LE master Host has accepted the connection parameters while 0x0001 indicates that the LE master Host has rejected the connection parameters.

Result	Description
0x0000	Connection Parameters accepted
0x0001	Connection Parameters rejected
Other	Reserved

Figure 3-27 conn para update rsp Format in BLE Stack

The 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 called to respond to this request.

```
void   blc_l2cap_SendConnParamUpdateResponse( u16 connHandle,
                                              int result);
```

“connHandle” indicates current connection ID. “result” has two options, “accept” and “reject”.

```
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 an 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.

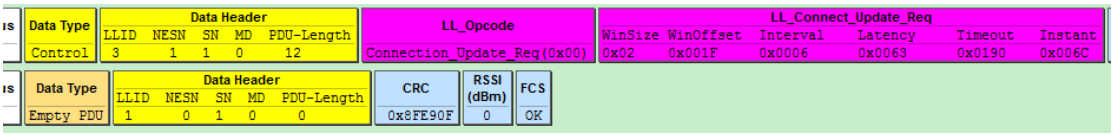


Figure 3-28 BLE Sniffer Packet Sample: ll conn update req

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.

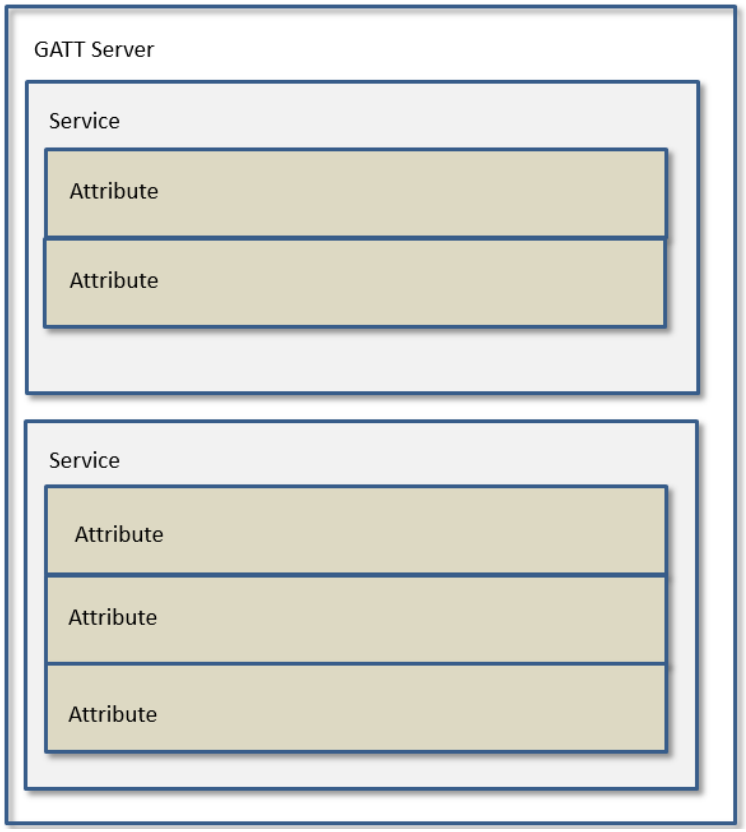


Figure 3-29 GATT Service Containing Attribute Group

The basic contents and properties of an 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. 5316 BLE stack defines some standard UUIDs in “stack/ble/service/hids.h” and “stack/ble/uuid.h”.
   Telink proprietary profiles (OTA, etc.) are not supported in standard Bluetooth. The 16-byte proprietary device UUIDs are defined in “stack/ble/uuid.h”.
2) Attribute Handle
   Slave supports multiple Attributes which compose an Attribute Table. In Attribute Table, each Attribute is identified by an Attribute Handle value. After connection is established, Master will analyze and obtain the Attribute Table of Slave via “Service Discovery” process, then it can identify Attribute data via the Attribute Handle during data transfer.
3) Attribute Value
   Attribute Value corresponding to each Attribute is used as data of request, response, notification indication and confirm. In 5316 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 5316 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[] = {
    {ATT_END_H - 1, 0,0,0,0,0}, // total num of attribute

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

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


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


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

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

Figure 3-30 5316 BLE SDK Attribute Table

Please note that the key word “const” is added to 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.

   {49,0,0,0,0,0},//ATT_END_H – 1 = 49 in “5316_ble_remote”

   “attNum = 49” indicates there are 49 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 49; 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 Figure 3-30, 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 short for “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, 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:
   For example, for 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, 1, 2, sizeof(my_devNameCharacter),(u8*)(&my_characterUUID),
    (u8*)(&my_devNameCharacter), 0,0},
```

   “UUID=0x2803” indicates “character” in BLE and the “uuid” points to the address of “my_characterUUID” in Flash. The “uuidLen” is 2. When Master reads this Attribute, the UUID would be “0x2803”.
2) Telink proprietary 16-byte UUID :
   For example, for Attribute of OTA, related code is shown as below:

```
#define TELINK_SPP_DATA_OTA

{0x12,0x2B,0x0d,0x0c,0x0b,0x0a,0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x02,
0x01,0x00}

const u8 my_OtaUUID[16]        = TELINK_SPP_DATA_OTA;
{0,ATT_PERMISSIONS_RDWR,16,sizeof(my_OtaData),(u8*)(&my_OtaUUID),
    (&my_OtaData), &otaWrite, &otaRead},
```

   The “uuid” points to the address of “my_OtaUUID” in Flash, and the “uuidLen” is 16. When Master reads this Attribute, the UUID would be “0x000102030405060708090a0b0c0d2b12”.

3.4.2.4 pAttrValue, 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 5316 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.

For Attribute hid Information with Attribute Handle being 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), 0,0},

In practical 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.

Figure 3-31 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.

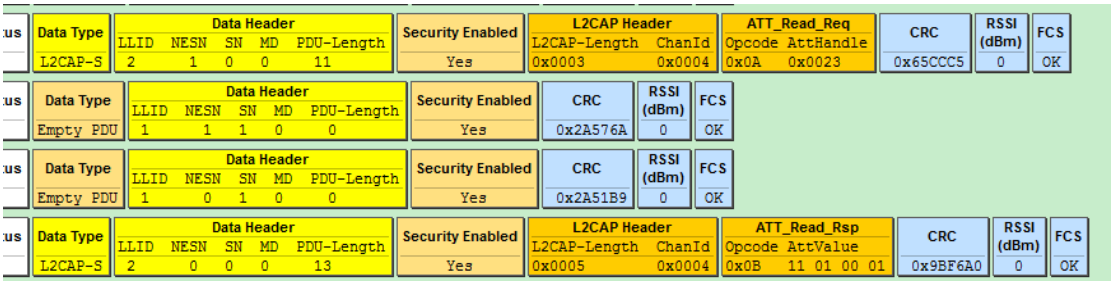


Figure 3-31 BLE Sniffer Packet Sample When Master Reads hidInformation

For Attribute “battery value” with Attribute Handle being 40, related code is as shown below:

u8 my_batVal[1] = {99};

{0,1,2,1,(u8*)(&my_batCharUUID), (u8*)(my_batVal), 0},

In practical 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 a write function, its prototype is:

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, see *Core_v5.0*.

2) opcode = 0x52, Write Command, see *Core_v5.0*.

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 “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

Figure 3-32 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

Figure 3-33 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 “stack/ble/ble_common.h”.

3.4.2.6 Callback Function r

The callback function r is a read function, its prototype is:

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

Users 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, see *Core_v5.0*.
2) opcode = 0x0C, Read Blob Request, *see Core_v5.0*.

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

Figure 3-34 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
```

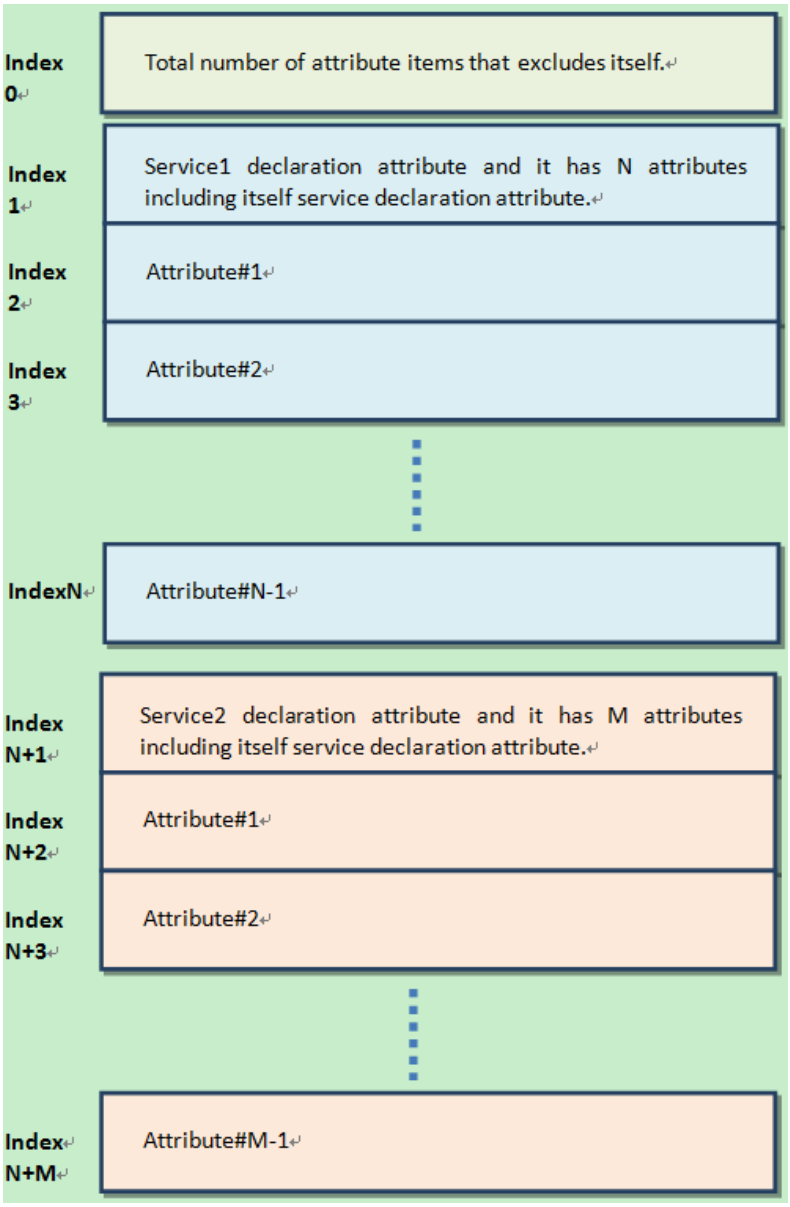


Figure 3-34 Service/Attribute Layout

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 is:

```
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, 5316 BLE SDK currently supports the following Attribute PDU types.

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

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.4.9 and 3.4.4.10) for details of Read by Group Type Request and 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.

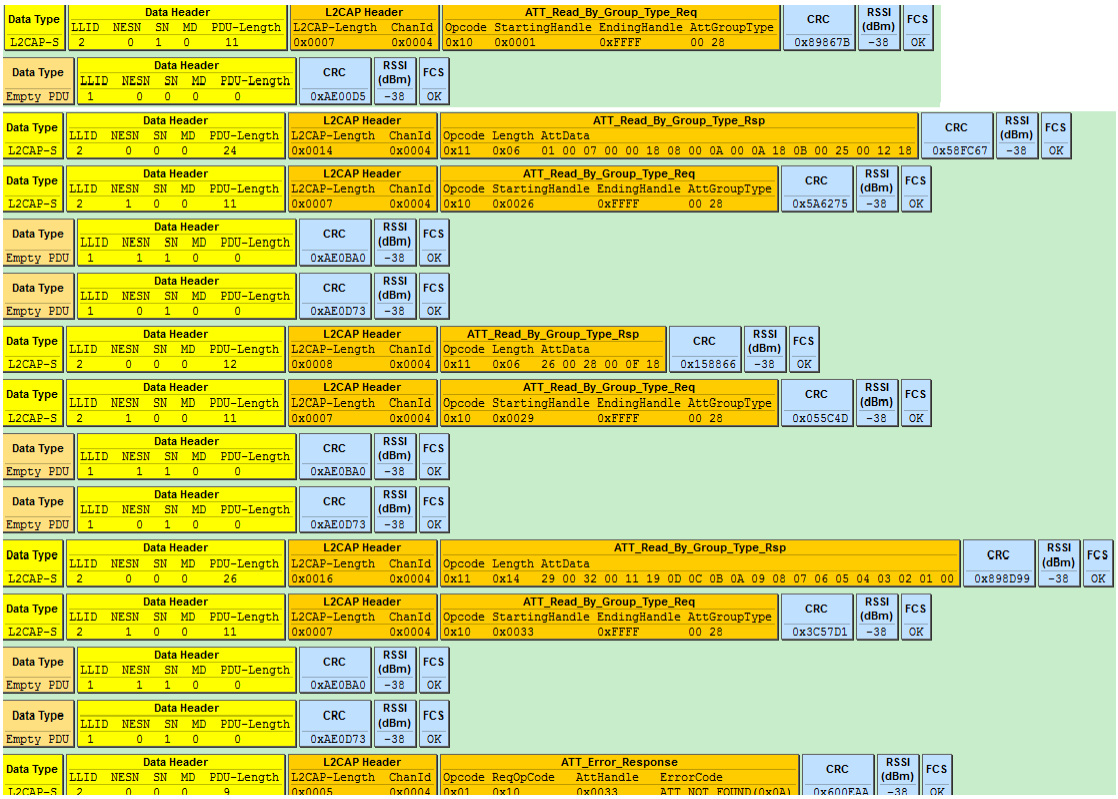


Figure 3-35 Read by Group Type Request/Read by Group Type Response

As shown above, Master requests from Slave for Attribute Group information of the “primaryServiceUUID” with UUID being 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_v5.0* for details of “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

Please see *Core_v5.0* (Vol 3/Part F/3.4.3.3 and 3.4.3.4) for details of “Find by Type Value Request” and “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.

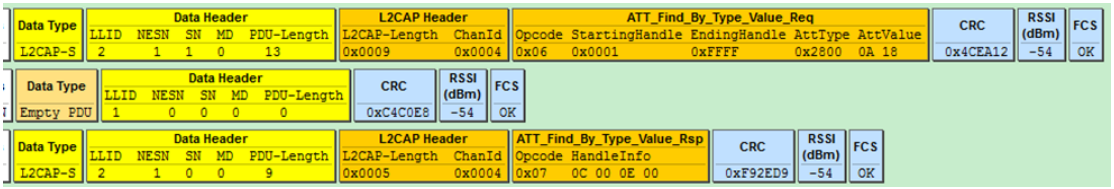


Figure 3-36 Find by Type Value Request/Find by Type Value Response

3.4.3.3 Read by Type Request, Read by Type Response

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.4.1 and 3.4.4.2) for details about “Read by Type Request” and “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”.

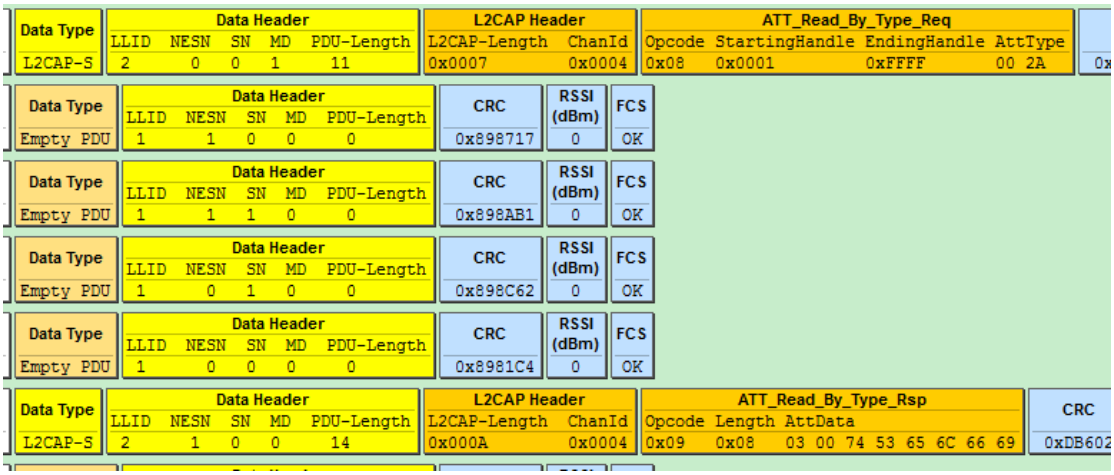


Figure 3-37 Read by Type Request/Read by Type Response

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.

3.4.3.4 Find Information Request, Find Information Response

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.3.1 and 3.4.3.2) for details about “Find information request” and “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.

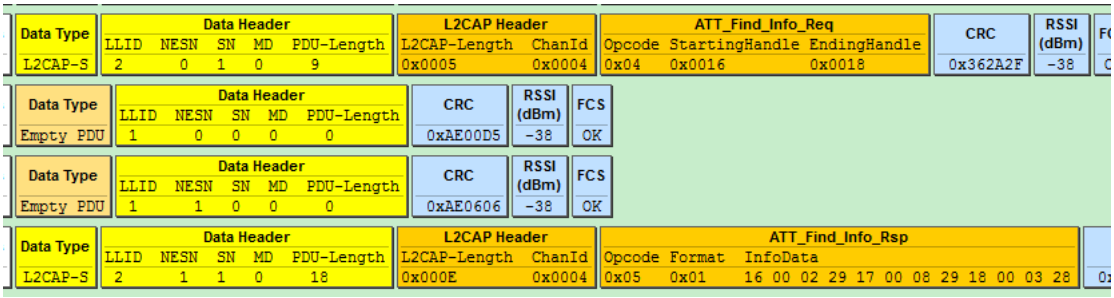


Figure 3-38 Find Information Request/Find Information Response

3.4.3.5 Read Request, Read Response

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.4.3 and 3.4.4.4) for details about “Read Request” and “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.

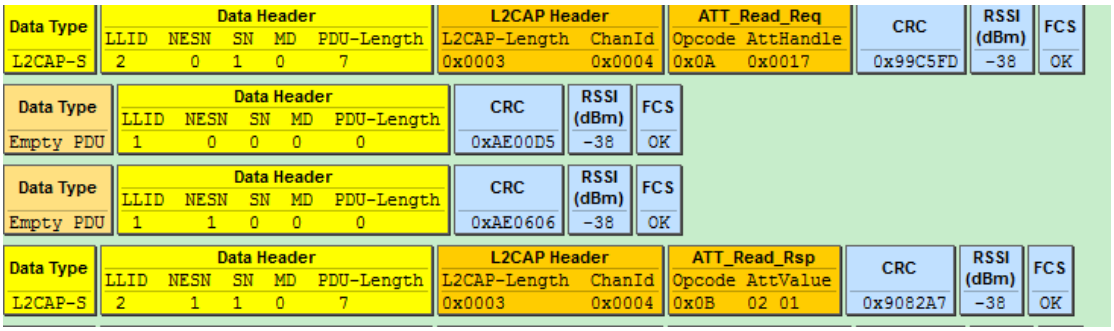


Figure 3-39 Read Request/Read Response

3.4.3.6 Read Blob Request, Read Blob Response

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.4.5 and 3.4.4.6) for details about “Read Blob Request” and “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”.

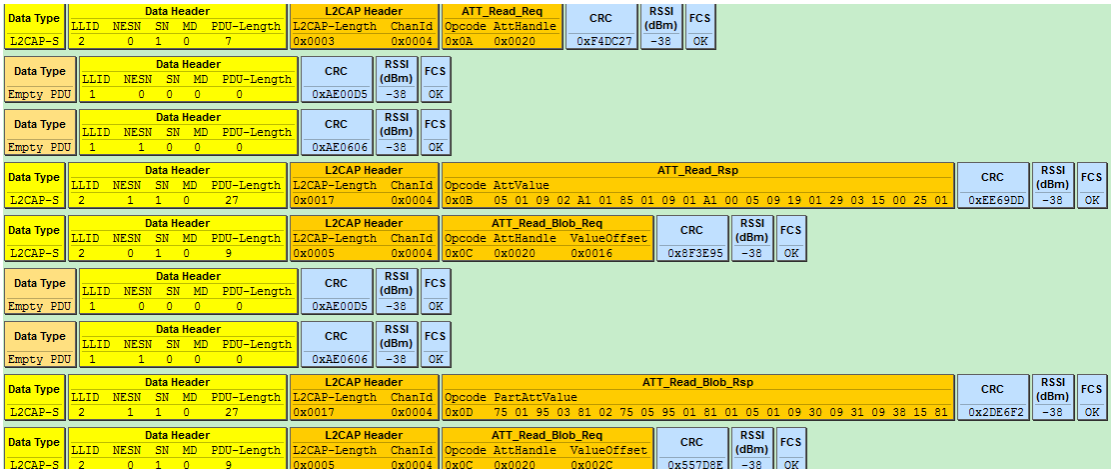


Figure 3-40 Read Blob Request/Read Blob Response

3.4.3.7 Exchange MTU Request, Exchange MTU Response

Please refer to Core_v5.0 (Vol 3/Part F/3.4.2.1 and 3.4.2.2) for details about “Exchange MTU Request” and “Exchange MTU Response”.

As shown below, Master and Slave obtain MTU size of the other via the “Exchange MTU Request” and “Exchange MTU Response” commands.

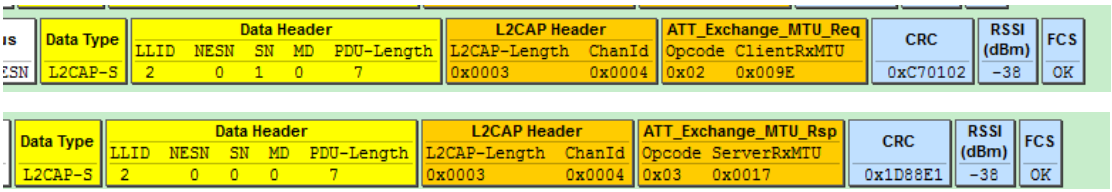


Figure 3-41 Exchange MTU Request/Exchange MTU Response

During data access process of Telink BLE Slave GATT layer, if there’s data exceeding a RF packet length, which involves packet assembly and disassembly in GATT layer, Slave and Master need to exchange RX MTU size of each other in advance. Transfer of long packet data in GATT layer can be implemented via MTU size exchange.

1) Callback function of MTU size exchange
   Function prototype:

```
typedef void (*attRxMtuSizeExchangeCommpleteCb)(
                                    u16 connHandle,
                                    u16 remoteMtuSize,
                                    u16 effectMtuSize);
```

   The first u16 is current connection handle, and it should be “BLS_CONN_HANDLE” in Slave applications. The second u16 is ClientRxMTU of Master. The third is the Rx MTU size finally used.
   The API is for registerring this callback function:

```
void blc_att_registerMtuSizeExchangeCb(
                                attRxMtuSizeExchangeCommpleteCb cb);
```

2) Processing of long Rx packet data in 5316 Slave GATT layer
   5316 Slave Server Rx MTU is set as 23 by default. Actually maximum Server Rx MTU can reach 247, i.e. 247-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 called to modify RX size of Slave first.

```
ble_sts_t   blc_att_setRxMtuSize(u16 mtu_size);
```

   The return values are shown as below:

ble_sts_t	Value	ERR Reason
BLE_SUCCESS	0	

ATT_ERR_INVALID_PARAMETER	0x85	mtu_size exceeds the max value 247.

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 5316 Slave GATT layer

When 5316 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 call the API “blc_att_setRxMtuSize” to set its ServerRxMTU. Suppose it is set to 158.

```
blc_att_setRxMtuSize (158);
```

Then the API below should be called 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

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.5.1 and 3.4.5.2) for details about “Write Request” and “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 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”.

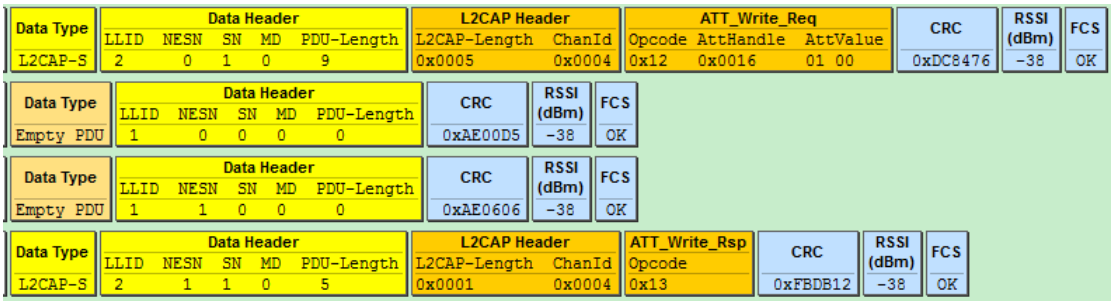


Figure 3-42 Write Request/Write Response

3.4.3.9 Write Command

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.5.3) for details about “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.

3.4.3.10 Handle Value Notification

Please refer to *Core_v5.0* (Vol 3/Part F/3.4.7.1) for details about “Handle Value Notification”.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1B = Handle Value Notification
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.34: Format of Handle Value Notification

Figure 3-43 Handle Value Notification in BLE Spec

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

5316 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 called 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 values are 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	Tasks with mass data are being executed, software Tx FIFO is not enough.

3.4.3.11 Handle Value Indication

Please refer to Core_v5.0 (Vol 3/Part F/3.4.7.2) for details about “Handle Value Indication”.

Parameter	Size (octets)	Description
Attribute Opcode	1	0x1D = Handle Value Indication
Attribute Handle	2	The handle of the attribute
Attribute Value	0 to (ATT_MTU-3)	The current value of the attribute

Table 3.35: Format of Handle Value Indication

Figure 3-44 Handle Value Indication in BLE Spec

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

5316 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 values are 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	Tasks with mass data are being executed, 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_v5.0* (Vol 3/Part F/3.4.7.3) for details about "Handle Value Confirmation".

Whenever the API "bls_att_pushIndicateData" is called 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

Figure 3-45 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.

Definition of type of the callback function is:

```
typede fint (*att_handleValueConfirm_callback_t)(void);
```

The API below serves to register the callback function:

```
void   bls_att_registerHandleValueConfirmCb
                          (att_handleValueConfirm_callback_t cb);
```

3.5 SMP

Security Manager (SM) in BLE is mainly used to provide various encryption keys for LE device to ensure data security. Encrypted link can protect the original contents of data in the air from being intercepted, decoded or read by any attacker. Please refer to *Core_v5.0* (Vol 3/Part H/ Security Manager Specification) for details of SMP.

3.5.1 SMP Parameter Configuration

Parameter configuration related to SMP initialization includes device bonding, OOB (Out-Of-Band) data verification and Secure Connection (SC).

3.5.1.1 Device Bonding

When it's needed to bond peer device information after pairing, the function below should be called to enable current device bonding request.

int blc_smp_enableBonding (int en);

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);

en: enable or disable OOB data verification;

en = 0, disable (default) OOB data verification;

en = 1, enable OOB data verification;

oobData: OOB data verification value, pointer to a group of 16-byte data.

3.5.1.3 Secure Connection Pairing (SC)

Secure Connection Pairing method is available from BLE 4.2. To distinguish it from former pairings, the old ones are called Legacy Pairing. Compared with Legacy Pairing, SC Pairing method is more secure.

5316 BLE SDK supports both Legacy Pairing and Secure Connection Pairing, SDK uses Legacy Pairing by default. If users want to use SC, call the function below:

```
void blc_smp_enableScFlag(int en);
```

en = 1, enable SC;

en = 0 (default), disable SC.

Please note that this function should be called in initialization.

3.5.2 Enable SMP

```
int bls_smp_enableParing (smp_paringTrriger_t encrypt_en);
```

The following introduces the definition of the enum type “smp_paringTrriger_t” and meanings of 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”.

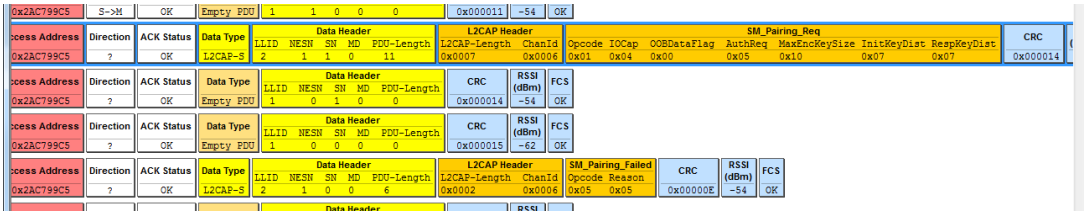


Figure 3-46 Pairing Disable

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”:

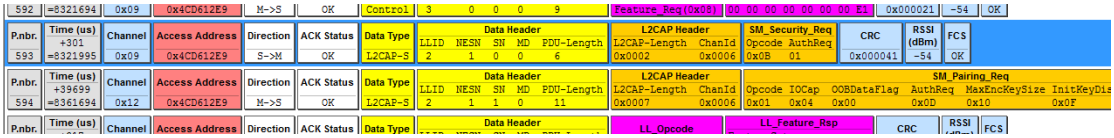


Figure 3-47 Pairing Conn Trigger

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.

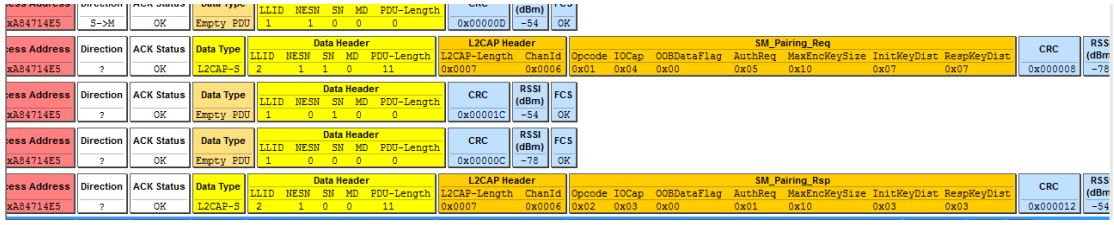


Figure 3-48 Pairing Peer Trigger

Note: This function can only be called before connection. It's recommended to call this function in 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

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.

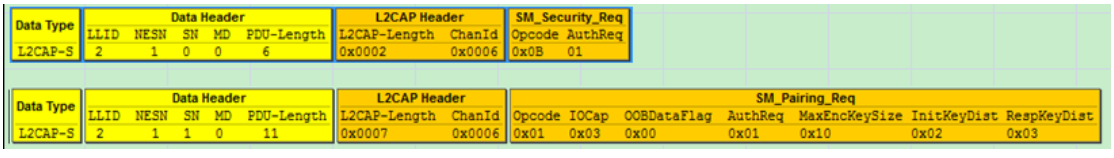


Figure 3-49 Pairing_Req Sent From Master

Pointer "p": Null pointer.

Data length "n": 0.

3.5.3.2 BLT_EV_FLAG_PAIRING_END

Event trigger condition: This event will be triggered when pairing is finished in success or failure. If Slave or Master fails to follow standard pairing procedure, or communication abnormity occurs (e.g. report error), pairing will fail.

Data length “n”: 1.

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

SMP bonding information herein is discussed relative to Slave device. Please refer to the code of “direct adv” setting in initialization in “5316 remote” project.

```
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 is used 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 (see “TEST_WHITELIST” in 5316 feature test).
- ✧ Other parameters are negligible to users.

The API below obtains 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. For example, if there are three bonding devices currently, to obtain info of the latest connected device, “index” should be set to “2”.

```
blc_smp_param_loadByIndex(2,  …)
```

The API below obtains 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.

4. Power Management (PM)

Power Management hereinafter is referred to as “PM”.

4.1 PM Driver

Driver files related to PM are available in drivers/5316/pm_5316.c, drivers/5316/pm.h, drivers/5316/pm_5316_32krc.c, drivers/5316/pm_5316_32kpad.c.

4.1.1 Low Power Modes

TLSR8232 supports three basic modes.

1) Working mode: In this mode, program runs normally, 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 8~30mA.
2) Suspend mode: Low power mode 1. In this mode, program execution pauses, most hardware modules in IC are powered off, and 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 8uA. After wakeup from suspend, program continues running from the break point.
3) Deepsleep mode: Low power mode 2. In this mode, program 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.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

Figure 4-1 shows wakeup sources available for TLSR8232: In suspend mode, it can be woken up by PAD, CORE and timer sources; while in deepsleep mode, it can be woken

up by PAD and timer sources. In 5316 BLE SDK, the following three types of wakeup sources are available.

```
enum {
    PM_WAKEUP_PAD   = BIT(4),
    PM_WAKEUP_CORE  = BIT(5),
    PM_WAKEUP_TIMER = BIT(6),
};
```

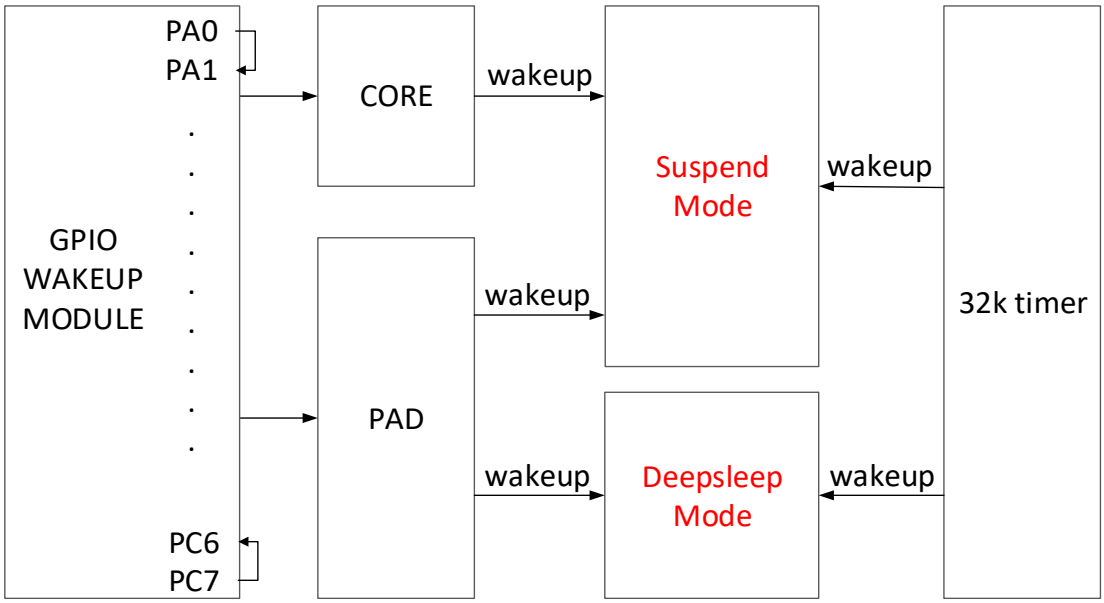


Figure 4-1 Hardware Wakeup Sources for 5316 MCU

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, 5316 BLE SDK, GPIO CORE works as wakeup source for suspend, while GPIO PAD works as wakeup source for deepsleep.

As all the GPIO levels can wake up corresponding low power modes by CORE and PAD, if users want to specify a level of certain GPIOs as the wakeup source, use the API below：

1) Configure GPIO CORE as wakeup source for suspend:

```
void gpio_set_wakeup(u32 pin, u32 level, int en);
```

“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);

"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" can be called to configure MCU to enter low power mode and set wakeup source(s).

```
int cpu_sleep_wakeup (SleepMode_TypeDef, SleepWakeupSrc_TypeDef,
                      unsigned int wakeup_tick);
```

1) Parameter "deepsleep": 0-enter suspend, 1-enter deepsleep.
2) Parameter "wakeup_src": It's used to configure wakeup source(s) for current suspend/deepsleep, and PM_WAKEUP_PAD, PM_WAKEUP_CORE and PM_WAKEUP_TIMER can be selected. 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) Parameter "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 timer tick plus sleep time tick, and it determines when MCU will be woke up by 32k timer. When system timer tick value matches the configured wakeup_tick, MCU is woken up from low power mode. If the wakeup_tick is directly configured without considering system timer tick, wakeup time can't be effectively controlled.

The absolute wakeup_tick value must be within the range 32 bits can represent, the maximum sleep time configured by this API is limited. In current design, maximum sleep time is set as 3/4 of the maximum scope 32 bits can represent. For 16MHz clock, 32 bits can represent about 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 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),
    STATUS_ENTER_SUSPEND  = BIT(30),
};
```

1) WAKEUP_STATUS_COMP is never used in BLE SDK, users do not need to know it.
2) WAKEUP_STATUS_TIMER/ WAKEUP_STATUS_CORE/ WAKEUP_STATUS_PAD correspond to PM_WAKEUP_TIMER/ PM_WAKEUP_CORE/ PM_WAKEUP_PAD, which represents the current low power mode is woken up by the wakeup sources.
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) *STATUS_ENTER_SUSPEND* represents the status enters suspend successfully.
5) The return value may be (*WAKEUP_STATUS_TIMER | WAKEUP_STATUS_CORE*) and it means 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.

Examples for cpu_sleep_wakeup usage:

1) cpu_sleep_wakeup (0, PM_WAKEUP_CORE, 0);
   MCU enters suspend mode when this function is executed, and it can be woken 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 woken 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 woken 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 woken 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 woken 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 woken 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 woken up by Timer.

4.2 BLE Low Power Management

In 5316 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. In this state SDK does not provide any low power management and users need to configure it themselves. Users may call "cpu_sleep_wakeup()" to implement low power management to configure MCU to enter suspend or deepsleep mode, and set wakeup sources 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
    }

}
```

Figure 4-2 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"). Users only need to call 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, PM module needs to be enabled in initialization by calling the API below. For applications with no need of PM this API does

not need to be called, thus PM related code and variables won't be compiled to the firmeware, and resources can be saved.

```
void        blc_ll_initPowerManagement_module(void);
```

4.3.2 Set Low Power Modes via "bls_pm_setSuspendMask"

The API used to configure PM for Link Layer Advertising state and Conn state in 5316 BLE SDK is:

```
void        bls_pm_setSuspendMask (u8 mask);
u8          bls_pm_getSuspendMask (void);
```

By using "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".

"bls_pm_getSuspendMask" is used 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".

Values for SuspendMask include:

```
/////////////////// 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 bls_pm_setWakeupSource

Users can enable or disable PM in different states by calling "bls_pm_setSuspendMask". The API below is used to set corresponding wakeup sources.

```
void        bls_pm_setWakeupSource(u8 source);
```

This API sets a bottom-layer variable “WakeupSource”. The actual variable in code is “bls_pm.wakeup_src”.

The WakeupSource includes PM_WAKEUP_PAD, PM_WAKEUP_CORE, PM_WAKEUP_TIMER, and their “or” combinations.

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 woken up from suspend or deepsleep, the WakeupSource will be cleared in bottom layer and become invalid. The wakeup source for suspend or deepsleep needs to be re-configured.

4.3.4 Working Mechanism of Low Power Managment

To better understand the configurations of “SuspendMask” and “WakeupSource”, this section introduces the principle of low power management mechanism.

In SDK, mainloop is a structure of while(1):

```
while(1)
{
    blt_sdk_main_loop();
    //UI   task
}
```

“blt_sdk_main_loop” is being executed in while(1) all the time. As the code of BLE low power management mechanism is in “blt_sdk_main_loop”, it is also being executed all the time.

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 of 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 orConn state Slave role)
    {
        if(Link Layer in Adv Event or Brx Event)  // BLE packet transfer is ongoing, not enter low power mode
        {
            return  0;
        }
        else
        {
            blt_brx_sleep (); //suspend & wakeup processing function

        }
    }
    return 1;
}
```

1) When “bltPm.suspend_mask” is set to “SUSPEND_DISALE”, “blt_brx_sleep” will not be executed. When users use “bls_pm_setSuspendMask (SUSPEND_DISABLE)”, the logic of BLE low power management mechanism will become totally invalid, MCU will not enter low power mode and loop of while(1) is being executed nonstop.
2) If Adv Event of Advertising State or Brx Event of Conn state Slave role is being executed, “blt_brx_sleep” will not be executed as RF task is in operation, SDK only enter sleep mode after Adv Event or Brx Event is completed.

“blt_brx_sleep” will be executed only if it is not under the two conditions above.

```
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 woken 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 woken 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 “user_latency” which users can configure. The API and source code are as below:

```
void bls_pm_setManualLatency(u16 latency)
{
    user_latency = latency;
}
```

The following shows how to calculate “latency_use”:

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.

Actually 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 calling “bls_pm_setManualLatency” only applies to the current suspend.

4.5 Other APIs

This section provides descriptions of other APIs except the APIs introduced above.

4.5.1 bls_pm_getSystemWakeupTick

The API below is used 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);
```

```
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)

    {

        //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 is used to decrease peak current during advertising.

```
void        bls_pm_enableAdvMcuStall(u8 en);
```

en: 1 - Enable MCU stall; 0 - Disable MCU stall.

Please note that Timer1 is used in stack bottom layer to implement MCU stall during advertising. If this power optimization is added, APP layer should not use Timer1.

4.6 Notes about GPIO Wakeup

4.6.1 Fail to Enter Suspend/Deepsleep When Wakeup Level is Valid

Since TLSR8232 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 5316 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.

```
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 is key not released, users 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 note this problem when using Telink GPIO CORE/PAD wakeup.

4.7 BLE System PM Reference

Based on the understanding of the principle of BLE SDK low power management, users can flexibly configure low power management. See the reference code of low power management of SDK demo "5316 ble remote" for reference.

Function "blt_pm_proc()" is included in UI entry of main_loop. Please note that 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 of low power management are:

1) If suspend needs to be disabled for task such as 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 LED task, etc., 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 users 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. Below 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. It's prototype and API are as follows:

```
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 Figure 4-3). If not, this wakeup_tick is negligible to bottom layer, and wakeup time depends on BLE timing sequence.

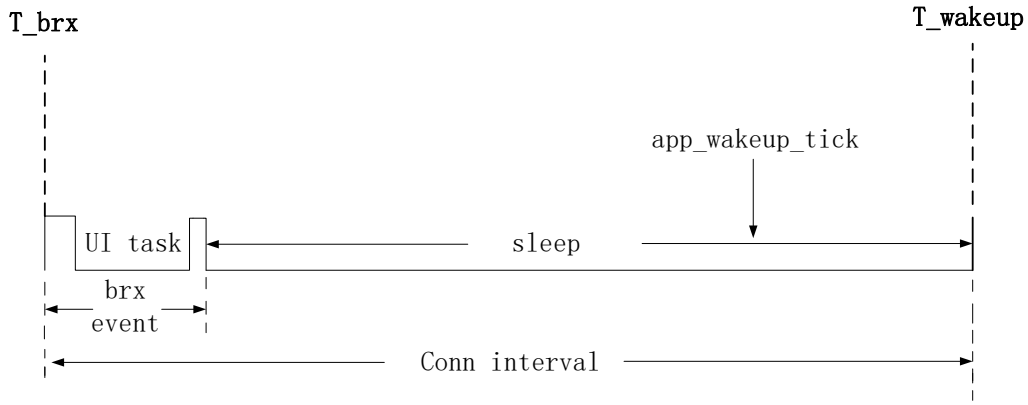


Figure 4-3 Trigger APP Wakup Tick in Advance

5. Low Battery Detect

Telink BLE SDK and related documents may refer to this subject with different names such as “battery power detect/check”, “battery power detect/check”, “low battery detect/check”, “low power detect/check”, “battery detect/check”. In SDK code, there are file and function names such as “battery_check”, “battery_detect”, “battery_power_check”. In this document, we use “low battery detect”.

5.1 Significance of Low Battery Detect

Since battery voltage would gradually drop with time, for battery-powered products, problems would occur once battery voltage drops below a certain level.

1) For TLSR8232, operation voltage range is 1.9V~3.6V. Below 1.8V, normal operation is not guaranteed.
2) At low battery voltage, unstable voltage may result in error for Flash “write” or “erase” operation, so that unexpected disruption of program FW or user data would lead to product function failure. Based on our MP experience, the threshold is set at 2.0V.

It is necessary to declare a “secure voltage”. Once battery drops below this voltage level, MCU should shutdown to stop working, or enter deepsleep as implemented in the SDK.

Before MCU shutdown, certain UI behavior, e.g. quick LED blinking in the “5316_ble_remote”), can be used as low battery alarm, so as to remind user that it’s time to recharge or replace the low battery.

“Secure voltage”, or “alarm voltage”, is set at 2.0V in current SDK. For certain extreme condition of power instability caused by HW design, secure voltage can be set at higher value, e.g. 2.1 or 2.2V.

For battery-powered products supported by Telink BLE SDK, Low Battery Detect must be implemented to ensure product stability across its lifetime.

5.2 Implementation of Low Battery Detect

For Low Battery Detect, ADC should be used to measure supply voltage. It’s recommended to read TLSR8232 datasheet or ADC driver related document and get familiar with ADC module.

In the SDK, Low Battery Detect is implemented in the “battery_check.h” and “battery_check.c”.

Please ensure the macro “BATT_CHECK_ENABLE” is enabled in the "app_config.h”. DO NOT change the macro since it’s enabled by default.

```
#define BATT_CHECK_ENABLE              1   //must enable
```

5.2.1 Cautions of Low Battery Detect

Low Battery Detect is a basic ADC task to sample and measure supply voltage. Please pay attention to the following items.

5.2.1.1 MUST Use GPIO Input Channel

For Telink 826x family, ADC module supports “VCC/VBAT” input channel to sample supply voltage.

For TLSR8232, though this design is reserved (corresponding to “VBAT” in the enum type “ADC_InputPchTypeDef”), based on certain reasons, it’s not allowed to use the “VBAT” channel. Therefore, user must adopt GPIO input channel of ADC instead, which includes PA6, PA7, and PB0~PB7.

```
/*ADC analog positive input channel selection enum*/

typedef enum {
    NOINPUTP,
    A6P,
    A7P,
    B0P,
    B1P,
    B2P,
    B3P,
    B4P,
    B5P,
    B6P,
    B7P,
    PGA0P,
    PGA1P,
    TEMSENSORP,
    RSSI_P,
    VBAT,
}ADC_InputPchTypeDef;
```

There are two ways to implement ADC sampling of supply voltage via GPIO input channel.

1) Directly connect power supply to GPIO input channel of ADC
   At ADC initialization, by setting specific GPIO to high impedance, voltage at the GPIO equals supply voltage, so ADC can directly sample supply voltage.
2) Use GPIO high-level output to sample supply voltage without connection between supply and GPIO input channel.
   As per the design of TLSR8232 internal circuit, voltage of GPIO high-level output always equals supply voltage and thus can be sampled by ADC.

In the project “5316_ble_remote” of the SDK, the second method is employed with GPIO input channel selected as PA7.

To use PA7 as GPIO input channel, PA7 should act as general GPIO, and at initialization user should follow its default configurations (ie, oe, output) without the need of special modification.

```
#define GPIO_VBAT_DETECT            GPIO_PA7
#define PA7_FUNC                    AS_GPIO
```

PA7 should output high level at ADC sampling.

```
gpio_set_input_en(GPIO_VBAT_DETECT, 0); //disable input function
gpio_set_output_en(GPIO_VBAT_DETECT, 1);//enable output function
gpio_write(GPIO_VBAT_DETECT, 1); //output 1
```

Generally, after ADC sampling, PA7 output can be disabled. However, for the "5316_ble_remote", PA7 on the corresponding HW is floating (NC), high-level output won't bring any current leakage, PA7 output is not disabled actually.

5.2.1.2 MUST Use ADC Differential Mode

Though in theory TLSR8232 ADC input supports both Single Ended Mode and Differential Mode, in the SDK as well as actual applications, only Differential Mode is allowed, and Single Ended Mode is forbidden.

Differential mode supports positive and negative input channel, thus voltage to be measured equals the voltage difference of positive end and negative end.

If only one GPIO input channel is available for ADC, this GPIO should be set as positive input channel, while GND should be set as negative input channel. By this setting, voltage difference equals voltage of positive end.

The code for Low Battery Detect in the SDK is as below. API "adc_set_all_differential_p_n_ain" selects PA7 as positive input channel, and GND as negative input channel.

For "adc_set_all_differential_p_n_ain(ADC_MISC_CHN, gpio_no, GND)", please see SDK for code.

5.2.1.3 MUST Use DFIFO for ADC Sampling Valu

For Telink 826x family, ADC result is readable via related register. For TLSR8232, DFIFO mode is used instead to get ADC result. Please refer to the following function in driver.

```
unsigned int adc_set_sample_and_get_result(void);
```

5.2.2 Dedicated Low Battery Detect Demo

In the project "5316_ble_remote" of the "app_config.h" file, set the macro "BATT_CHECK_ENABLE" to 1, ADC is dedicated for Low Battery Detect. Users can also refer to the project "5316_ble_remote" for Low Battery Detect demo.

5.2.2.1 Initialization of Low Battery Detect

Refer to function “adc_vbat_init”.

ADC initialization must always follow the flow: power off SAR ADC; configure ADC parameters; power on SAR ADC. TLSR8232 sets SAR ADC as power down by default, so “adc_vbat_init” does not set power down of “adc_power_on(0)”. TLSR8232 enables ADC power Every time before reading ADC data and disables ADC power after reading is completed.

It’s not recommended to modify the initialization setting of ADC, users can use the default setting. However, users can select other GPIO input channel via the macro “BATTERY_CHECK_PIN”.

If on HW design, supply is directly connected to GPIO input channel, high level output needs to be removed from the “BATTERY_CHECK_PIN”.

Below is code of “adc_vbat_init” in “battery_power_check”.

```
if(!adc_hw_initialized){

    adc_hw_initialized = 1;

    adc_init();

    adc_vbat_init(BATTERY_CHECK_PIN);

}
```

Herein a variable “adc_hw_initialized” is used to call one initialization. When the variable is set to 0, one initialization is called, and then the variable should be set to 1 to disable further initialization.

By using the “adc_hw_initialized” ADC tasks can switch between Low Battery Detect and other ADC tasks (“ADC other task”).

Due to dynamic ADC task switch, the “adc_vbat_init” may be executed multiple times, so it must be implemented in the main_loop rather than user_init().

On first calling of “battery_power_check”, “adc_vbat_init” is executed and will not be executed repeatedly.

To switch to “ADC other task”, for proper initialization in new task, the “adc_hw_initialized” will be set to 0.

After completion of “ADC other task”, the “battery_power_check” will be executed again. Since current adc_hw_initialized is 0, the “adc_vbat_init” must be executed again, so as to guarantee re-initialization on every switch back.

5.2.2.2 Low Battery Detect Processing

In main_loop, the function “battery_power_check” is called to process Low Battery Detect. Related code is shown as below:

```
if(clock_time_exceed(lowBattDet_tick, 500*1000)){

    lowBattDet_tick = clock_time();

    battery_power_check(BATTERY_VOL_MIN);

}
```

The “lowBattDet_tick” can be used to set frequency of battery detect. In the demo, the period is set as 500ms by default, and it can be modified as needed.

Implementation of the “battery_power_check” involves details of battery detect initialization, DFIFO setup, data acquisition and processing, as well as low voltage alarming.

Complicated ADC usage and special HW limits may bring user difficulty in understanding. It is highly recommended to follow the demo code as much as possible. Except a few settings which is illustrated as modifiable in this document, DO NOT make any change.

DFIFO mode is used to acquire ADC result by sampling 8 times (default), removing the maximum and minimum and calculating the average of 6 values. As shown in the “adc_vbat_init”, period for each sampling is 10.4us, so it takes 83us or so to get the result.

In demo code, the macro “ADC_SAMPLE_NUM” can be changed to 4, so as to reduce total ADC time to 41us. Sampling 8 times is recommended to get more accurate result. If users modify the marco definition, the calculation of ADC should be modified accordingly. The source code is:

```
u32 adcValueAvg = (adc_sample[2] + adc_sample[3] + adc_sample[4] +
adc_sample[5]) >> 2;
```

5.2.2.3 Low Battery Voltage Alarm

The parameter “minVol_mV” of the “battery_power_check” specifies secure or alarm voltage in unit of mV. As explained earlier, default in the SDK is 2000 mV. In low battery detect of main_loop, once supply voltage drops below 2000mV, MCU enters low voltage range.

The following shows demo code to process low battery alarm. MCU must be shut down once it enters low voltage range.

In “5316_ble_remote”, MCU can be shut down by entering deepsleep, and wake up by key press.

Except mandatory MCU shutdown operation, users can modify other alarm behaviors.

In the code below, alarm indication is set as fast blinking for three times, reminding users to recharge or replace battery.

```
if(vol < minVol_mV){
        #if (1 && BLT_APP_LED_ENABLE)  //led indicate
            gpio_set_output_en(GPIO_LED, 1);  //output enable
            for(int k=0;k<3;k++){
                gpio_write(GPIO_LED, 1);
                sleep_us(200000);
                gpio_write(GPIO_LED, 0);
                sleep_us(200000);
            }
            gpio_set_output_en(GPIO_LED, 0);
        #endif
```

```
        analog_write(DEEP_ANA_REG2, BATTERY_VOL_LOW);
        cpu_sleep_wakeup(PM_SLeepMode_Deep, PM_WAKEUP_PAD, 0);
    }
```

For “5316_ble_remote”, after shutdown at low battery, MCU enters deepsleep which supports key press wakeup.

At the moment of wakeup by key press, SDK will perform one fast battery detect in user initialization (function user_init) rather than in main_loop, so as to avoid errors as shown below.

Given that LED blinking has been issued for low battery, MCU entered deepsleep, but wakes up from deepsleep by key press. If processed in main_loop, battery detect needs to wait 500ms at least to be executed ; during this duration Slave could have sent many advertising packets, and may even connect with master, which will lead to a bug that device resumes working after low battery alarm.

For this reason, battery detect is performed at user initialization instead. So battery detect is added in user initialization:

```
        if(analog_read(DEEP_ANA_REG2) == BATTERY_VOL_LOW){
            battery_power_check(BATTERY_VOL_MIN + 200);//2.2V
        }
        else{
            battery_power_check(BATTERY_VOL_MIN);//2.0 V
        }
```

Please note that the initialization of GPIO wakeup must be placed before the code above or it may lead to GPIO wakeup failure.

Value of the analog register “DEEP_ANA_REG2” can tell whether it’s wakeup from low battery shutdown. Battery detect after this wakeup will raise 2000mV alarm voltage to 2200mV recover voltage, based on the reason below:

Tolerance in battery detect makes it difficult to ensure consistent result on every measurement. Given 20mV error, for first time, 1990mV might trigger MCU to shutdown; however, after wakeup, at user initialization it could be measured at 2005mV, so 2000mV alarm voltage would lead to the bug above.

Considering this, at battery detect after wakeup from shutdown, alarm voltage is raised by a value slightly higher than maximum tolerance of low battery detect.

Since the example 2200mV recover voltage only occurs after voltage lower than 2000mV is detected to shutdown MCU, user does not need to worry that applications would give low voltage alarm when actual voltage is 2.0~2.2V. End user should recharge or replace battery at low battery alarm to ensure normal product performance.

6. OTA

TLSR8232 supports Flash multi-address booting, addresses including 0, 0x10000, 0x20000, and 0x40000.

To implement OTA for TLSR8232 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 TLSR8232 BLE OTA is realized.

6.1 Flash Architecture and OTA Procedure

6.1.1 Flash Storage Architecture

When boot from 0x20000, in SDK the default firmware size should not exceed 128K, i.e. the Flash area 0~0x20000 is for storing firmware.

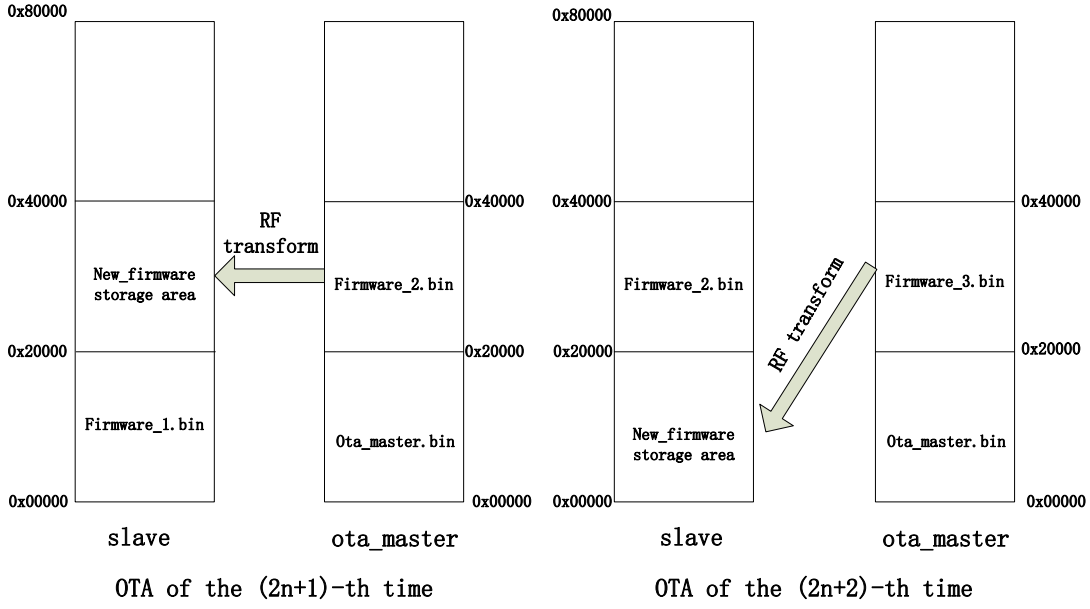


Figure 6-1 5316F512K Flash Storage Structure

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 firmware2 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 OTA Update Procedure

Based on the Flash storage structure introduced in section 6.1.1, the OTA update procedure is illustrated as below:

8232 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 TLSR8232 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 5316 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 Firmware Size and Boot Address

The API “bls_ota_setFirmwareSizeAndOffset” supports two functions: modifying maximum firmware size and booting address in the OTA design. Herein booting address means the adderss except 0 to store New_firmware, so it should be 0x10000 or 0x20000 or 0x40000.

In SDK, by default, the default maximum firmware size is 128kB, and the booting address is 0x20000.

```
void bls_ota_setFirmwareSizeAndOffset(int firmware_size_k, u32 boot_addr);
```

“firmware_size_k” must be 4kB aligned, i.e. it must be integral multiples of 4kB. For example, for 97kB size, the “firmware_size_k” must be set as 100kB.

In the main function, since “cpu_sleep_wakeup” contains settings related to “firmware_size_k” and “boot_addr”, this API must be called before “cpu_wakeup_init” takes effect.

If maximum firmware_size exceeds 128kB, booting address needs to be changed to 0x40000. For example, maximum firmware_size may reach 200kB; corresponding setting should be:

bls_ota_setFirmwareSizeAndOffset (200, 0x40000);

By using this API, not only booting address can be modified, but also Flash area usage can be optimized.

By default, maximum firmware size is 128kB, and the Flash space 0x00000 ~ 0x40000 can be used to store firmware only. If firmware does not need such a large area, e.g. FW size does not exceed 60kB, only part of the two 128kB space (0x00000 ~ 0x20000, 0x20000 ~ 0x40000) are used.

To use the redundant space as data storage area, the setting below can be followed.

bls_ota_setFirmwareSizeAndOffset (60, 0x20000);

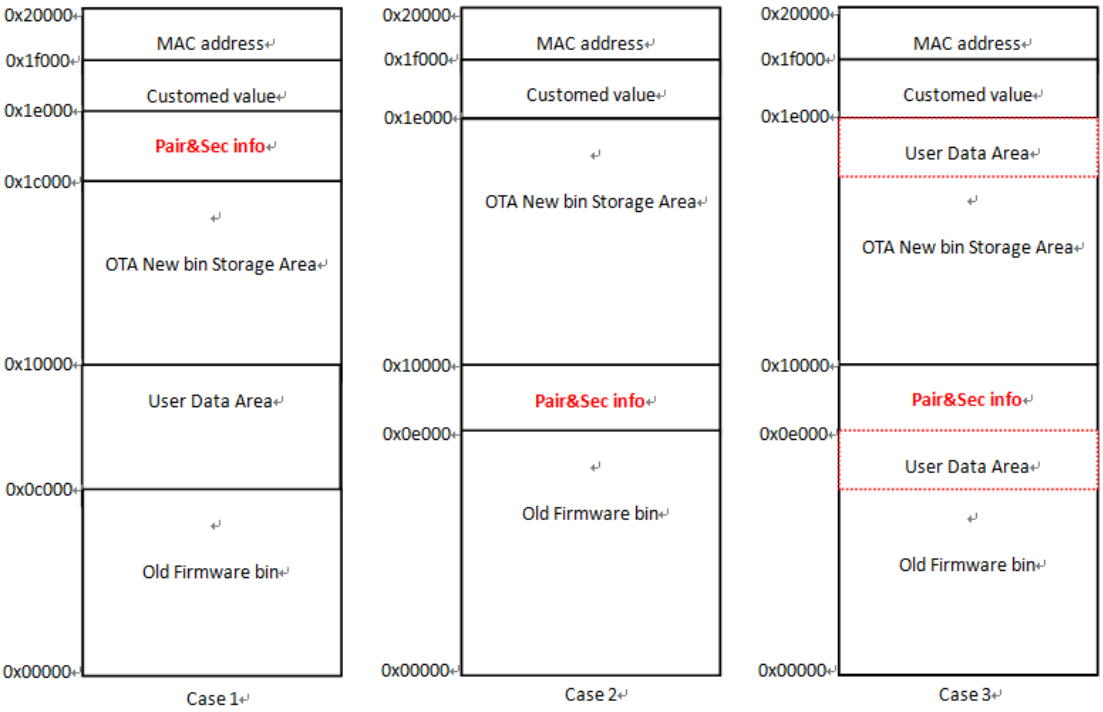
By the configuration above, the two 60kB Flash areas 0x00000 ~ 0x0F000 and 0x20000 ~ 0x2F000 can be used as firmware storage space, while the two 68kB Flash areas 0x0F000 ~ 0x20000 and 0x2F000 ~ 0x40000 can be used as user data storage space.

The situation in TLSR8232F128 OTA is essentially the same as TLSR8232F512, just change offset 0x20000 to 0x10000.

The following should be noted:

1) For TLSR8232F128, the default firmware size of SDK is 48kB.
2) For TLSR8232F128, by modifying the SDK configuration the maximum firmware size can be up to 56kB but there will be no extra Flash space for storing user data.

See figure below:



Case 1 is the default distribution, the maximum firmware size is 48kB with 16kB left for User Data Area.

If 52K < firmware size <=56K, users can use Case 2 which has no extra Flash space for user data.

If 48K < firmware size <=52K, users can use Case 3 which has extra Flash area for user data.

For Case 2 and Case 3, the API below should be called to set actual firmware size (firmware size should be 4kB aligned), and ota_offset should be set to 0x10000.

```
void bls_ota_setFirmwareSizeAndOffset(int firmware_size_k, u32 ota_offset);
```

6.2 RF Data Proceesing in OTA Mode

6.2.1 OTA Processing in Attribute Table on Slave Side

First, OTA reference needs to be added to app_att.c which includes the Attribute Table:

```
#include <stack/ble/ble.h> //includes the reference of 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.

6.2.2 OTA Data Packet Format

Master sends command and data to Slave via “Write Command” in L2CAP layer.

3.4.5.3 Write Command

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

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

Figure 6-2 Write Command Format in BLE Stack

The Attribute Handle value is the handle_value of OTA data on Slave side. The Attribute Value length is set as 20, and the format is shown as below.

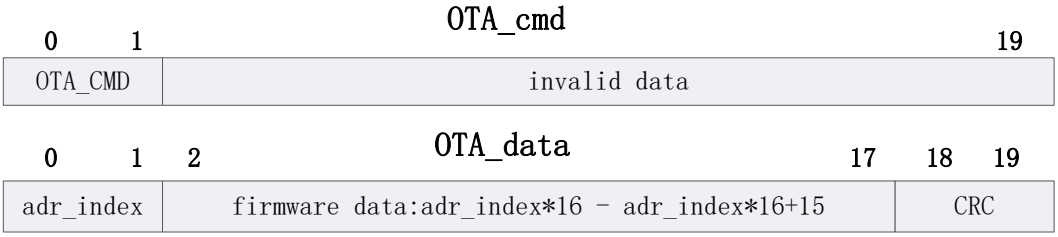


Figure 6-3 Format of OTA Command and Data

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: reserved for future use.

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 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 (19th~20th byte) of the data.

6.2.3 RF Transfer Processing on Master Side

Since BLE link-layer RF data will be automatically responded with ack to avoid packet loss, during OTA data transfer Master won't check 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 Attribute on Slave side. Users 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".

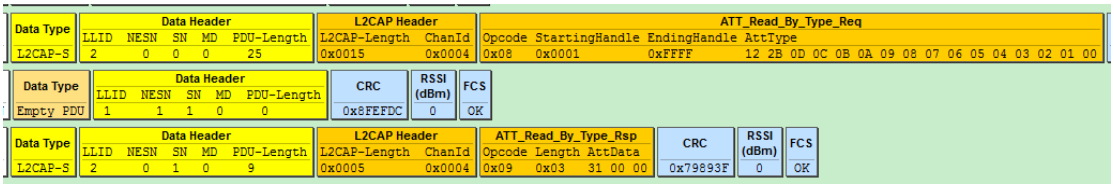
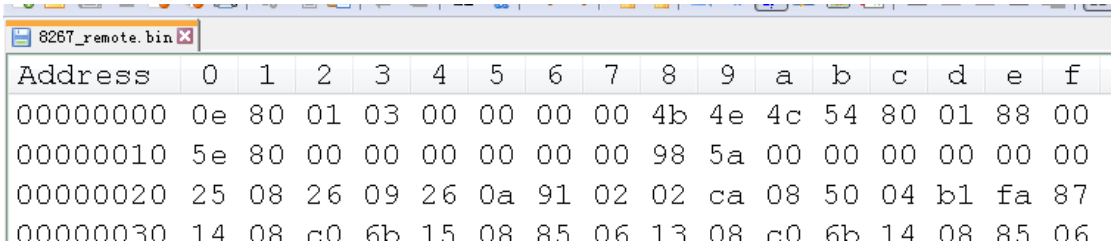


Figure 6-4 Master Obtains OTA Attribute Handle via "Read By Type Request"

3) (optional) Obtain current Slave firmware version number. User can check if it's the newest version and decide whether to start OTA upgrade correspondingly.
   In 5316 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 5316 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 "5316_remote.bin" is "0x00005a98", so the firmware size is 0x5a98, i.e. 23192 bytes from 0 to 0x5a97.


8267_remote.bin

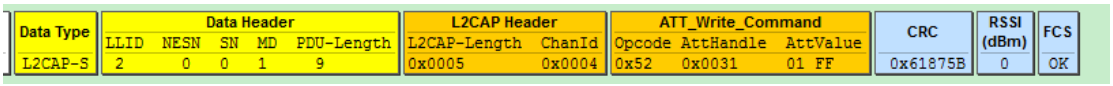
Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000	0e	80	01	03	00	00	00	00	4b	4e	4c	54	80	01	88	00
00000010	5e	80	00	00	00	00	00	00	98	5a	00	00	00	00	00	00
00000020	25	08	26	09	26	0a	91	02	02	ca	08	50	04	b1	fa	87
00000030	14	08	c0	6b	15	08	85	06	13	08	c0	6b	14	08	85	06

Figure 6-5 Firmware: 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								

Figure 6-6 Firmware: 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

Figure 6-7 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 5316_remote.bin as shown in Figure 6-5 and Figure 6-6 is taken as an example to illustrate how to assemble OTA data.

   Data for first transfer: "adr_index" is "0x00 00", 16-byte data are values of addresses 0x0000 ~ 0x000f. Suppose CRC calculation result for the former 18 bytes is "0xXYZW", the 20-byte data should be:

   0x00 0x00 0x0e 0x80 ... (12 bytes not listed)... 0x88 0x00 0xZW 0xXY

Data for second transfer:

0x01 0x00 0x5e 0x80 ... (12 bytes not listed)... 0x00 0x00 0xJK 0xHI

Data for third transfer:

0x02 0x00 0x25 0x08 ... (12 bytes not listed)... 0xfa 0x87 0xNO 0xLM

.....

Data for penultimate transfer:

0xa8 0x05 0x02 0x04 ... (12 bytes not listed)... 0x00 0x00 0xST 0xPQ

Data for final transfer:

0xa9 0x05 0x44 0x58 0x00 0x00 0x01 0x00 0x00 0x00

0xff 0xff 0xff 0xff 0xff 0xff 0xff 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”.

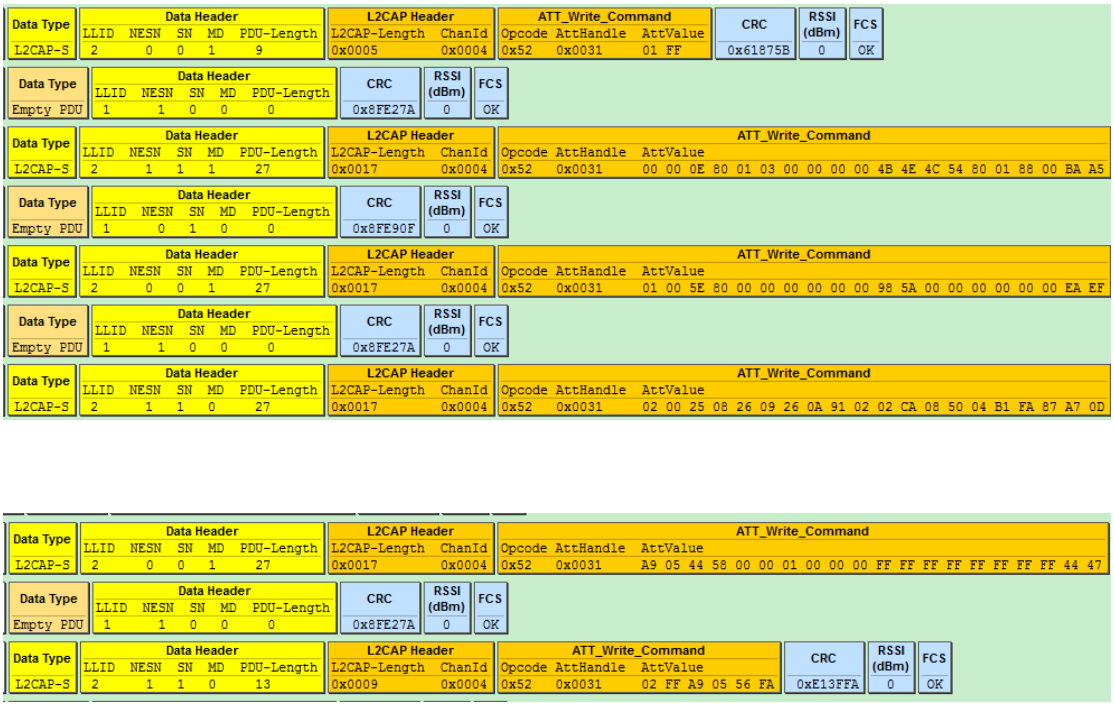


Figure 6-8 Master OTA Data

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 “5316_remote.bin” are “0x05a9” and “0xfa56”, respectively. Figure 6-8 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.

6.2.4 RF Receive Processing on Slave Side

As introduced above, Slave can directly invoke the otaWrite and otaRead in OTA Attribute. After Slave receives write command from Master, it will be parsed and processed automatically in BLE stack by invoking the otaWrite function.

In the otaWrite function, the 20-byte packet data will be parsed: first judge whether it’s OTA CMD or OTA data, then process correspondingly (respond to OTA cmd; check CRC to OTA data and burn data into specific addresses of Flash).

The OTA related operations on Slave side are shown as below:

1) OTA_FIRMWARE_VERSION command is received (first two bytes are 0xff00): Master requests to obtain Slave firmware version number.
In 5316 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:

```
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 15s by default in current SDK). If OTA process is not finished within the duration, it’s regarded as OTA failure due to timeout. Users 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 provides 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 in Slave is assembled in lib, other related interfaces are available in stack/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). Users can directly use the sample code, or realize the function by developing his own code.

7.1 Key Matrix

Take Telink 5316 BLE remote demo board as an example: It's a 5*6 matrix and supports up to 30 buttons. Five drive pins (Row0~Row4) are used to output drive level, while six scan pins (CoL0~CoL5) serve to scan for button press in current column.

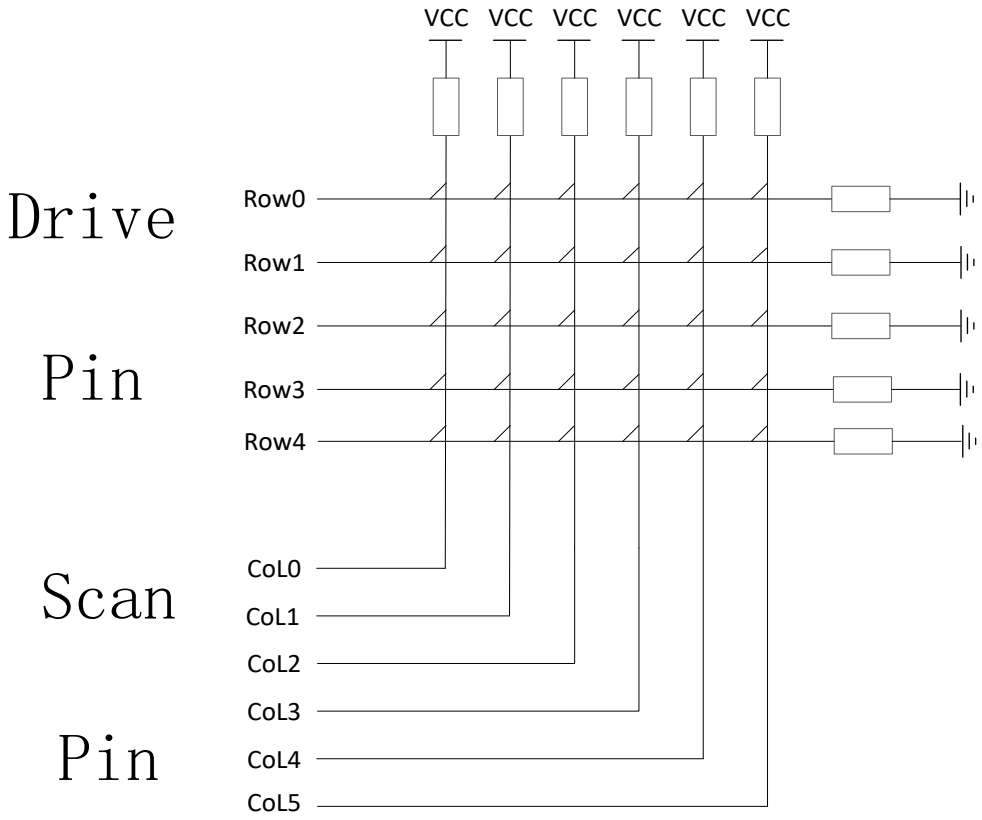


Figure 7-1 Row/Column Key Matrix

Keyscan related configurations in app_config.h are shown as below:

On Telink demo board, Row0~Row4 pins are PA5, PA4, PA3, PA2, and PA1, while CoL0~CoL5 pins are PC6, PC5, PC4, PC3, PC2, and PC1.

Define drive pin array and scan pin array:

```
#define  KB_DRIVE_PINS  {GPIO_PA5, GPIO_PA4, GPIO_PA3, GPIO_PA2, GPIO_PA1}

#define  KB_SCAN_PINS   {GPIO_PC6, GPIO_PC5, GPIO_PC4, GPIO_PC3, GPIO_PC2,
GPIO_PC1}
```

Keyscan adopts analog pull-up/pull-down resistor in TLSR8232 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_PA5    MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PA4    MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PA3    MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PA2    MATRIX_ROW_PULL
#define PULL_WAKEUP_SRC_PA1    MATRIX_ROW_PULL

#define PULL_WAKEUP_SRC_PC6    MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PC5    MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PC4    MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PC3    MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PC2    MATRIX_COL_PULL
#define PULL_WAKEUP_SRC_PC1    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 PC6_INPUT_ENABLE       1
#define PC5_INPUT_ENABLE       1
#define PC4_INPUT_ENABLE       1
#define PC3_INPUT_ENABLE       1
#define PC2_INPUT_ENABLE       1
#define PC1_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 PA5_INPUT_ENABLE       1
```

```
#define PA4_INPUT_ENABLE      1
#define PA3_INPUT_ENABLE      1
#define PA2_INPUT_ENABLE      1
#define PA1_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 (30) 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” is used 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 30 buttons in app_config.h is shown as below:

```
#define     KB_MAP_NORMAL   {\
    {VK_NONE,          VK_UP,      VK_ENTER,    VK_DOWN,     VK_NONE},    \
    {KEY_MODE_SWITCH, VK_LEFT,    CR_MENU,     CR_VOL_MUTE, VK_RIGHT},   \
    {VK_POWER,         CR_HOME,    VK_7,        VK_2,        CR_BACK},    \
    {VK_NONE,         CR_VOL_DN,   VK_NONE,     VK_5,        CR_VOL_UP}, \
    {VK_NONE,         VK_1,         VK_0,       VK_8,        VK_3},       \
    {VK_NONE,         VK_4,        VK_NONE,     VK_9,        VK_6}, }
```

The keymap follows the format of 5*6 matrix structure. The keycode of pressed button can be configured accordingly, for example, the keycode of the button between Row0 and CoL1 is “VK_UP”.

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;
        }
    }
}
```

Figure 7-2 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 users.

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 called, 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.)
   If row-by-row scan is directly adopted without initial full scan through the whole matrix, each time all rows (five 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.

   ```
   scan_pin_need = kb_key_pressed (gpio);
   ```

   In the function kb_key_pressed, all rows output low level, and stabilized level of scan pins will be read after 20us delay. A release_cnt is set as 6; if a detection shows all pressed buttons in the matrix are released, it won't consider no button is pressed and stop row-by-row scan immediately, but buffers for six frames. If six successive detections show buttons are all released, it will stop row-by-row scan. Thus key debounce processing is realized.
2) Scan row by row according to full scan result through the whole matrix.
   If button press is detected by full scan, row-by-row scan is started: Drive pins (ROW0~ROW4) output valid drive level row by row; read level on columns, and find the pressed button. Following is related code:

```
    u32 pressed_matrix[ARRAY_SIZE(drive_pins)] = {0};


    kb_scan_row (0, gpio);
    for (int i=0; i<=ARRAY_SIZE(drive_pins); i++) {
        u32 r = kb_scan_row (i < ARRAY_SIZE(drive_pins) ? i : 0,
gpio);
        if (i) {
            pressed_matrix[i - 1] = r;
        }
    }
```

The following methods are used to optimize code execution time for row-by-row scan.

- ✧ When a row outputs drive level, it's not needed to read level of all columns (CoL0~CoL5). Since the scan_pin_need marks valid column number, user can read the marked columns only.
- ✧ After a row outputs drive level, a 20us or so delay is needed to read stabilized level of scan pins, and a buffer processing is used to utilize the waiting duration.
  The array variable "u32 pressed_matrix[5]" (up to 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;
    }
}
```

Figure 7-3 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 5316 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
}
```

In 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 and 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, }
```

Users 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.

Two related macros in app_config.h are:

```
//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.

Users 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[5] is defined to record row(s) with stuck key in current key matrix. The array value is obtained in the function "key_debounce_filter".

Upper-layer processing is shown as below:

```
kb_event.keycode[0] = 0;
```

```
int det_key = kb_scan_key (0, 1);


if (det_key){
    #if (STUCK_KEY_PROCESS_ENABLE)
    if(kb_event.cnt){  //key press
        stuckKey_keyPressTime = clock_time();
    }
    #endif

    .......
}
```

For each button state update, when button press is detected (i.e. kb_event.cnt is non-zero value), the “stuckKey_keyPressTime” is used to record the time for the latest button press state.

Processing in 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 Functions

Source code about LED management is available in vendor/common/blt_led.c of 5316 BLE SDK for user reference. Users can directly include the “vendor/common/blt_led.h” into their C files.

The following three functions need to be called:

```
void device_led_init(u32 gpio,u8 polarity);

int device_led_setup(led_cfg_t led_cfg);

static inline void device_led_process(void);
```

In 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 is used 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

Users 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 needs to be added 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
```

```
}
```

Users can refer to the code in current 5316 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. Users can directly use this timer or modify it as needed.

Source code is 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

Call the API below to initialize blt software timer:

```
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” is used 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 woken up from suspend in advance 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 reaches: if the task reaches, exit the function; otherwise continue the flow. Since the design will ensure all timers are time-ordered, only the nearest timer needs to be checked.

```
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 reaches.

```
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 is used to add timer task.

```
typedef int (*blt_timer_callback_t)(void);


int    blt_soft_timer_add(blt_timer_callback_t func, u32
interval_us);
```

“func”: timer task function;“interval_us”: timing value (unit: us). The int-type return value corresponds to three processing methods:

1) If the return value is less than 0, this 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 called 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 5316 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 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 every 27ms.

10. IR

10.1 PWM Driver

Please refer to PWM section in TLSR8232 datasheet to better understand PWM driver.

PWM related hardware configurations are very simple and are basically implemented by operating registers. APIs are all defined in “pwm.h” (c files not needed) and are implemented by using “static inline function”, which improves efficiency and saves code size.

10.1.1 PWM id and Pin

TLSR8232 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.

IC pins of the 12-channel PWM are shown as below:

PWMx	Pin		PWMx_n	Pin
PWM0	PA0/PB3		PWM0_N	PB6/PC2
PWM1	PB1/PB7		PWM1_N	PA2/PB4
PWM2	PA4/PB2		PWM2_N	PC1
PWM3	PB0/PC7		PWM3_N	PA1
PWM4	PA3/PB5/PC6		PWM4_N	PA6
PWM5	PA5/PA7		PWM5_N	PC3

Use “void gpio_set_func(GPIO_PinTypeDef pin, GPIO_FuncTypeDef func)” to set PWM function of pins.

Pin: actual output pins of PWM waveform. func must select from AS_PWM0 ~ AS_PWM5_N in the definition of GPIO_FuncTypeDef according to actual PWM functions of GPIOs in the table above.

```
typedef enum{
    ……
    AS_PWM0    = 20,
    AS_PWM1    = 21,
    AS_PWM2    = 22,
    AS_PWM3    = 23,
    AS_PWM4    = 24,
    AS_PWM5    = 25,
    AS_PWM0_N  = 26,
    AS_PWM1_N  = 27,
    AS_PWM2_N  = 28,
    AS_PWM3_N  = 29,
    AS_PWM4_N  = 30,
    AS_PWM5_N  = 31,
}GPIO_FuncTypeDef;
```

For example, use PA0 as PWM0:

```
gpio_set_func(GPIO_PA0,    AS_PWM0)
```

10.1.2 PWM Clock

The “pwm_set_clk(int system_clock_hz, int pwm_clk)” is used to set PWM clock.

- ✧ “system_clock_hz”: current system clock CLOCK_SYS_CLOCK_HZ. (The marco is defined in app_config.h.)
- ✧ “pwm_clk”: PWM clock to be configured. “system_clock_hz” must be an integral multiple of “pwm_clk” so as to get the right PWM clock via frequency division.

To increase accuracy of PWM waveform, PWM clock must be as large as possible but smaller than system clock. It’s recommended to set “pwm_clk” as “CLOCK_SYS_CLOCK_HZ”:

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, CLOCK_SYS_CLOCK_HZ);
```

Suppose the current system clock CLOCK_SYS_CLOCK_HZ is 16000000, the PWM clock set above equals to the system clock, as 16M.

If PWM clock needs to be 8M, set as below. No matter how the system clock changes (CLOCK_SYS_CLOCK_HZ is 16000000, 32000000 or 48000000), PWM clock will be 8M.

```
pwm_set_clk(CLOCK_SYS_CLOCK_HZ, 8000000);
```

10.1.3 PWM Cycle and Duty

PWM waveform consists of PWM Signal Frames. For a PWM Signal Frame, “cycle” and “cmp” need to be configured via related APIs.

```
void pwm_set_cycle(pwm_id id, unsigned short cycle_tick)
```

This API is used to set PWM cycle, the unit is the number of PWM clocks.

```
void pwm_set_cmp(pwm_id id, unsigned short cmp_tick)
```

This API is used to set PWM cmp, the unit is the number of PWM clocks.

The API below combines the two APIs into one, which improves the configuration efficiency.

```
void pwm_set_cycle_and_duty(pwm_id id, unsigned short cycle_tick,
                            unsigned short cmp_tick)
```

PWM duty of a PWM signal frame is calculated as below:

PWM duty = PWM cmp/PWM cycle

The figure below shows the result of pwm_set_cycle_and_duty(PWM0_ID, 5, 2). The cycle of a Signal Frame is five PWM clocks, 2 PWM clocks in high level, PWM duty is 40%.

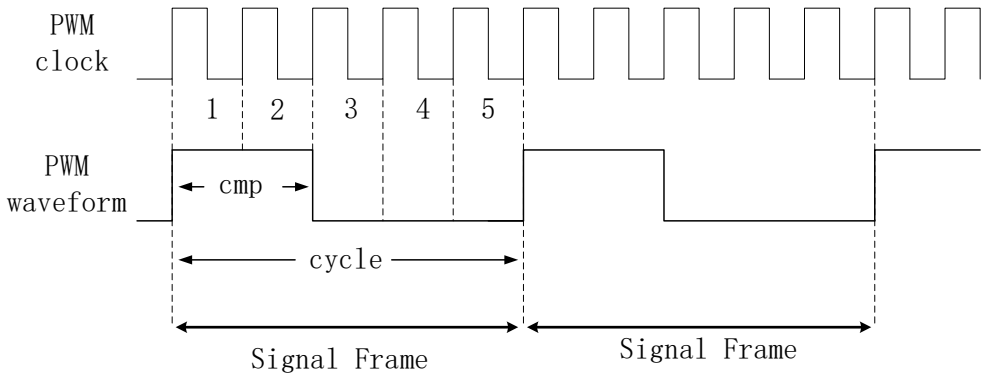


Figure 10-1 PWM Cycle & Duty

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, use the following two methods:

1) Use corresponding PWM0_N ~ PWM5_N (inverted output of PWM0 ~ PWM5).
2) Use API “`static inline void pwm_revert(pwm_id id)`” to invert PWM0 ~ PWM5 waveform.

Suppose current PWM clock is 16MHz, to set PWM cycle to 1ms and duty cycle for PWM0 to 50%:

pwm_set_cycle(PWM0_ID , 16000)

pwm_set_cmp (PWM0_ID , 8000)

or

pwm_set_cycle_and_duty(PWM0_ID, 16000, 8000);

10.1.4 PWM Revert

“static inline void pwm_revert(pwm_id id)” is used to invert PWM0 ~ PWM5 waveform.

“static inline pwm_n_revert(pwm_id id)” is used to invert PWM0_N ~ PWM5_N waveform.

10.1.5 PWM Start and Stop

Use the two interfaces below to enable (start)/disable (stop) certain PWM.

```
void pwm_start(pwm_id id) ;

void pwm_stop(pwm_id id) ;
```

10.1.6 PWM Mode

PWM supports five modes: Normal mode(also Continuous mode), Counting mode, IR mode, IR FIFO mode and IR DMA FIFO mode.

```
typedef enum{
    PWM_NORMAL_MODE   = 0x00,
    PWM_COUNT_MODE    = 0x01,
    PWM_IR_MODE       = 0x03,
    PWM_IR_FIFO_MODE  = 0x07,
    PWM_IR_DMA_FIFO_MODE  = 0x0F,
}pwm_mode;
```

The API to set PWM mode is:

```
void pwm_set_mode(pwm_id id, pwm_mode mode)
```

PWM0 supports all five modes, Normal mode, Counting mode, IR mode, IR FIFO mode and IR DMA FIFO mod, while PWM1 ~ PWM5 only support normal mode. In other words, PWM0 supports other four special modes besides normal mode.

Please section 8.5 in TLSR8232 datasheet for details of PWM modes.

10.1.7 PWM Pulse Number

`void pwm_set_pulse_num(pwm_id id, unsigned short pulse_num)` is used to set the number of Signal Frames of specified PWM waveforms. Here “pulse” means Signal Frames.

For normal mode (Continuous mode), there is no limit to the number of Signal Frames, therefore this API is meaningless to normal mode. This API is only useful to other four special modes.

10.1.8 PWM Phase

void pwm_set_phase(pwm_id id, **unsigned short** phase) is used to set delay time before PWM is started. "phase" is the delay time, the unit is the number of PWM clocks. Generally it can be set as 0 (no delay).

10.1.9 PWM Interrupt

Some basic concepts about Telink MCU interrupt are introduced here.

Interrupt "status" is state flag bit generated by hardware interrupt request of certain IRQ source, and it does not depend on software setting. No matter whether "mask" is enabled, interrupt request will always set corresponding IRQ "status" to 1. Generally, "status" can be cleared to 0 by writing it with "1".

Interrupt response: When CPU receives an interrupt request (IRQ) from certain IRQ source, it will determine whether to respond to the IRQ. If yes, firmware pointer PC will jump to interrupt handling part "irq_handler".

To enable interrupt response, please make sure all "mask" bits corresponding to current IRQ are enabled. One IRQ may correspond to multiple "mask" bits which are the relation of logic "And". IRQ request won't trigger interrupt response unless all of its related "mask" bits are enabled.

PWM driver in the "register.h" only involves the following IRQ sources.

```
#define reg_pwm_irq_mask        REG_ADDR8(0x7b0)

#define reg_pwm_irq_sta         REG_ADDR8(0x7b1)

enum{

    FLD_IRQ_PWM0_PNUM =                 BIT(0),

    FLD_IRQ_PWM0_IR_DMA_FIFO_DONE  =    BIT(1),

    FLD_IRQ_PWM0_FRAME =            BIT(2),

    FLD_IRQ_PWM1_FRAME =            BIT(3),

    FLD_IRQ_PWM2_FRAME =            BIT(4),

    FLD_IRQ_PWM3_FRAME =            BIT(5),

    FLD_IRQ_PWM4_FRAME   =              BIT(6),

    FLD_IRQ_PWM5_FRAME =            BIT(7),

};
```

The eight IRQ sources listed in the enum correspond to core_7b0 BIT<0:7> ("mask") / core_7b1 BIT<0:7> ("status").

In the figure below, PWM0 works in IR mode, duty cycle of Signal Frame is 50%, pulse number (i.e. Signal Frame number) for each IR task is 3. This figure will help to illustrate the three types for PWM IRQ "status".

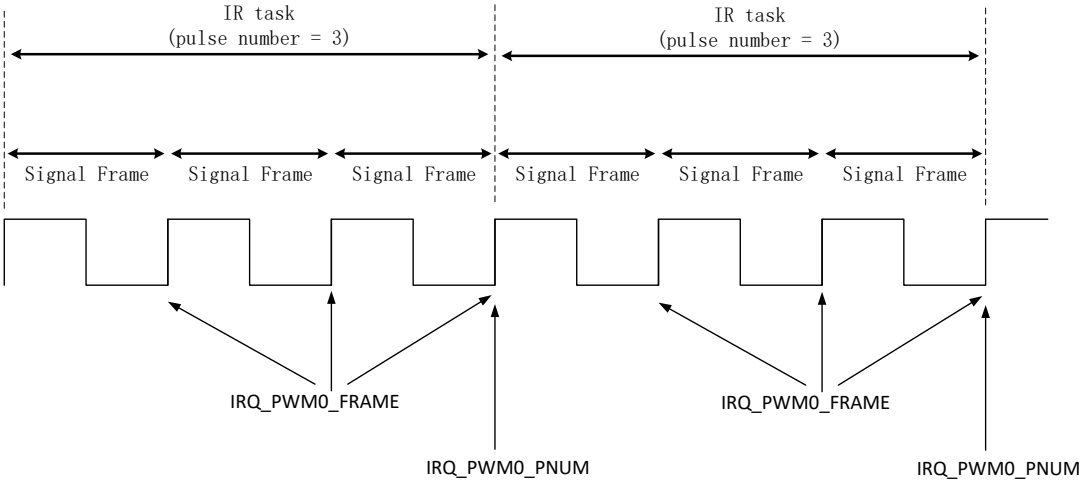


Figure 10-2 PWM interrupt

1) IRQ_PWMn_FRAME(n=0,1,2,3,4,5) for PWM0~PWM5: After each signal frame, PWM#n (n=0~5) will generate a frame-done IRQ (Interrupt Request) signal "IRQ_PWMn_FRAME".
   As shown in the figure above, six frame-done IRQ signal are generated at the end of each PWM0 Signal Frame.
2) IRQ_PWM0_PNUM: In Counting mode and IR mode, PWM0 will generate a Pnum IRQ signal "IRQ_PWM0_PNUM" after completing a group of Signal Frames (pulse number is determined by the API pwm_set_pulse_num).
   As shown in the figure above, PWM0 will generate a Pnum IRQ signal at the end of a pulse group containing three Signal Frames.
3) IRQ_PWM0_IR_DMA_FIFO_DONE
   In IR DMA FIFO mode, PWM0 will generate an IR waveform send done IRQ signal "IRQ_PWM0_IR_DMA_FIFO_DONE", after all PWM waveforms configured in DMA are sent.

As described above, IRQ request won't trigger interrupt response unless all of its related "mask" bits are enabled. Taking "FLD_IRQ_PWM0_PNUM" for an example, three "mask" bits need to be enabled.

1) Enable "mask" of FLD_IRQ_PWM0_PNUM, i.e. core_7b0:
   reg_pwm_irq_mask |= FLD_IRQ_PWM0_PNUM;
   Generally, to avoid false triggering of interrupt response, previous "status" needs to cleared before enabling "mask".
   reg_pwm_irq_sta = FLD_IRQ_PWM0_PNUM;
2) Enable PWM "mask" in MCU system interrupt, i.e. core_640 BIT<14>.

```
#define reg_irq_mask            REG_ADDR32(0x640)

enum{

    ......

    FLD_IRQ_SW_PWM_EN =  BIT(14),  //irq_software | irq_pwm

    ......
```

```
}
```

The method to enable this “mask”:

```
Reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
```

3) Enable MCU global interrupt “mask”, i.e. irq_enable().

10.1.10 API for IR DMA FIFO Mode

This section introduces the APIs dedicated for IR DMA FIFO mode. Please refer to PWM demo code in the SDK.

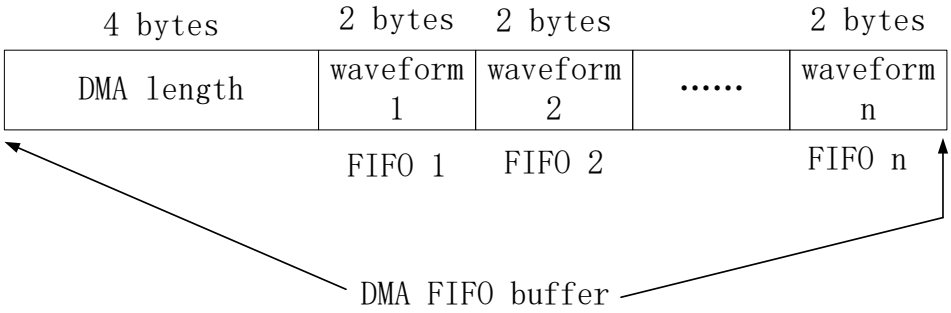


Figure 10-3 DMA FIFO Buffer for IR DMA FIFO Mode

DMA FIFO buffer is a data block defined in SRAM, and the “DMA length” of the first 4 bytes indicates the number of bytes occupied by FIFO. As shown above, DMA length = n*2.

There are n FIFOs, and each FIFO has two bytes to indicate one PWM waveform. For TLSR8232, “n” can be up to 256.

After DMA data buffer takes effect, PWM HW module will send out waveform 1 ~ waveform n successively.

After all waveforms are sent, PWM is stopped automatically and IRQ_PWM0_IR_DMA_FIFO_DONE is triggered.

10.1.10.1 Configuration of DMA FIFO

Each DMA FIFO uses 2 bytes (16 bits) to configure one PWM waveform. When the API below is called, 2-byte DMA FIFO data will be returned.

```
Unsigned short pwm_config_dma_fifo_waveform(int carrier_en,
    Pwm0Pulse_SelectDef pulse,  unsigned short pulse_num);
```

By configuring the three parameters “carrier_en”, “pulse” and “pulse_num”, the PWM output waveform contains “pulse_num” PWM pulses (Signal Frames).

As shown in the configuration format in the 8232 datasheet, BIT(15) specifies Signal Frame format of PWM waveform, and corresponds to the “carrier_en” of this API.

- ✧ When “carrier_en” is 1, PWM will output carrier pulse.
- ✧ When “carrier_en” is 0, PWM will output Signal Frames with low level only.

The “pulse_num” specifies the number of Signal Frames for current PWM waveform.

The “pulse” supports two definitions below:

```
Typedef enum{
    PWM0_PULSE_NORMAL =      0,
    PWM0_PULSE_SHADOW =      BIT(14),
}Pwm0Pulse_SelectDef;
```

- ✧ When “pulse” is PWM0_PULSE_NORMAL, Signal Frame uses the configuration of the API “pwm_set_cycle_and_duty”.
- ✧ When “pulse” is PWM0_PULSE_SHADOW, Signal Frame uses the configuration of PWM shadow mode.
  PWM shadow mode enables more flexibility for PWM waveform configuration in IR DMA FIFO mode. Related API is shown as below, and its configuration is consistent with pwm_set_cycle_and_duty.

```
Void pwm_set_pwm0_shadow_cycle_and_duty(unsigned short cycle_tick,
                                        unsigned short cmp_tick);
```

10.1.10.2 Set DMA FIFO Buffer

After DMA FIFO buffer is configured, call the API below to set the starting address of the buffer to DMA module.

```
Void pwm_set_dma_address(void * pdat);
```

10.1.10.3 Start and Stop of IR DMA FIFO Mode

After DMA FIFO buffer is prepared, call the API below to start sending PWM waveforms.

```
void pwm_start_dma_ir_sending(void);
```

After all PWM waveforms in DMA FIFO buffer are sent, the PWM module will be stopped automatically. The API below can be called to manually stop the PWM module in advance.

```
void pwm_stop_dma_ir_sending(void);
```

10.2 IR Demo

Please refer to the IR demo code in SDK demo “5316_ble_remote”. Set the macro “REMOTE_IR_ENABLE” in “app_config.h” to 1.

10.2.1 PWM Mode Selection

As required by IR transmission, PWM output needs to switch at specific time with small error tolerance of switch time accuracy to avoid incorrect IR.

As described in Link Layer timing sequence (section 3.2.4), Link Layers uses system interrupt to process brx event. (In the new SDK, adv event is processed in the main_loop and does not occupy system interrupt time.) When IR is about to switch PWM output soon, if brx event related interrupt comes first and occupies MCU time, the time to switch PWM output may be delayed, thus to result in IR error.

Therefore IR cannot use PWM Normal mode.

For Telink 826x BLE SDK, PWM IR mode is used to implement IR. Please refer to “826x BLE SDK handbook”.

Since 826x PWM IR mode only supports data pre-storage of two IR FIFOs, if PWM Signal Frame takes very short time, even shorter than BLE interrupt handling time in irq_handler, PWM waveform may be delayed, thus it brings a risk for IR mode.

5316 introduces an extra IR DMA FIFO mode which is not supported by 826x. In IR DMA FIFO mode, since FIFO can be defined in SRAM, more FIFOs are available, which can effectively solve the shortcoming of PWM IR mode above.

IR DMA FIFO mode supports pre-storage of multiple PWM waveforms into SRAM. Once DMA is started, no software involvement is needed. This can save frequent SW processing time, and avoid PWM waveform delay caused by simultaneous response to multiple IRQs in interrupt system.

Only PWM0 with IR DMA FIFO mode can be used to implement IR. Therefore, in HW design, IR control GPIO must be PWM0 pin or PWM0_n pin.

10.2.2 Demo IR protocol

The figure below shows demo IR protocol in SDK.

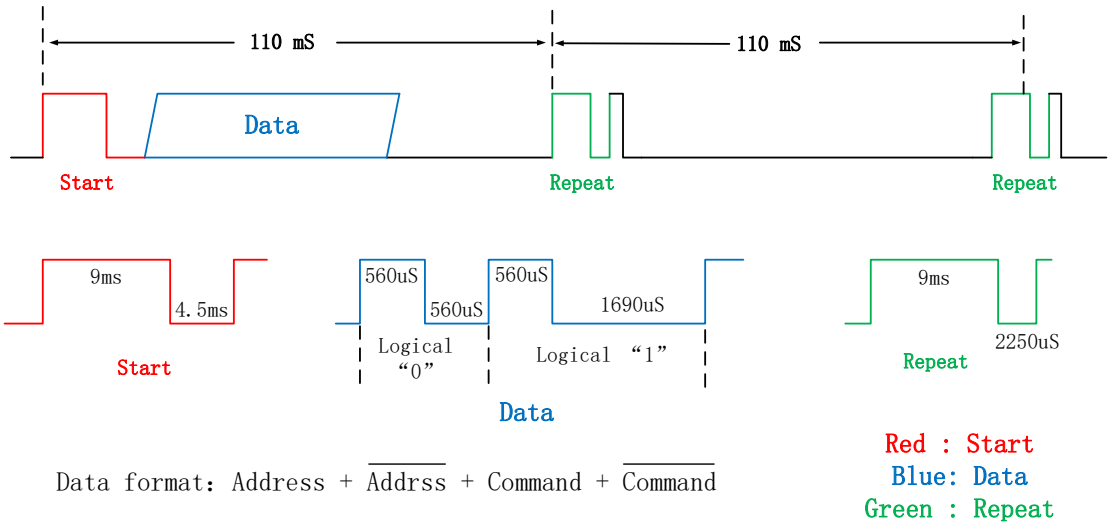


Figure 10-4 Demo IR protocol

10.2.3 IR Timing Design

The figure below shows basic IR timing abased demo IR protocol and feature of IR DMA FIFO mode.

In IR DMA FIFO mode, a complete task is defined as FifoTask. Herein the processing of IR repeat signal adopts the method of “add repeat one by one”, i.e. the macro below is defined as 1.

```
#define ADD_REPEAT_ONE_BY_ONE          1
```

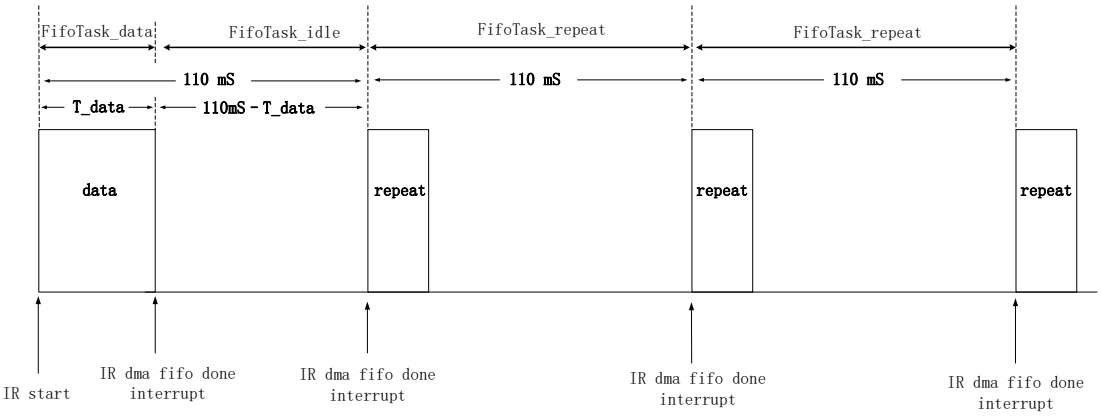


Figure 10-5 IR Timing 1

When a button is pressed to trigger IR transmission, IR is disassembled to FifoTasks as shown in Figure 10-5.

1) After IR is started, run FifoTask_data to send valid data. The duration of FifoTask_data, marked as T_data, is not certain due to the uncertainty of data. After FifoTask_data is finished, trigger IRQ_PWM0_IR_DMA_FIFO_DONE.
2) In interrupt function of IRQ_PWM0_IR_DMA_FIFO_DONE, start FifoTask_idle phase to send signal without carrier and it lasts for a duration of (110ms – T_data). This phase is designed to guarantee the time point the first FifoTask_repeat is 110ms later after IR is started. After FifoTask_idle is finished, trigger IRQ_PWM0_IR_DMA_FIFO_DONE.
3) In interrupt function of IRQ_PWM0_IR_DMA_FIFO_DONE, start the first FifoTask_repeat. Each FifoTask_repeat lasts for 110ms. By adding FifoTask_repeat in corresponding interrupt function, IR repeat signals can be sent continuously.
4) The time point to stop IR is not certain, and it depends on the time to release the button. After the APP layer detects key release, as long as FifoTask_data is correctly completed, IR transmission is finished by manually stoppng IR DMA FIFO mode.

Following shows some optimization steps for the IR timing design above.

1) Since FifoTask_repeat timing is fixed, and there are many DMA FIFOs in IR DMA FIFO mode, multiple FifoTask_repeat of 110ms can be assembled into one FifoTask_repeat*n, so as to reduce the number of times to process IRQ_PWM0_IR_DMA_FIFO_DONE in SW.
   Corresponding to the processing of “ADD_REPEAT_ONE_BY_ONE” macro defined as 0, the Demo herein assembles five IR repeat signals into one FifoTask_repeat*5. User can further optimize it according to the usage of DMA FIFOs.
2) Based on step 1), combine FifoTask_ilde and the first “FifoTask_repeat*n” to form “FifoTask_idle_repeat*n”.

The figure below shows optimized IR timing.

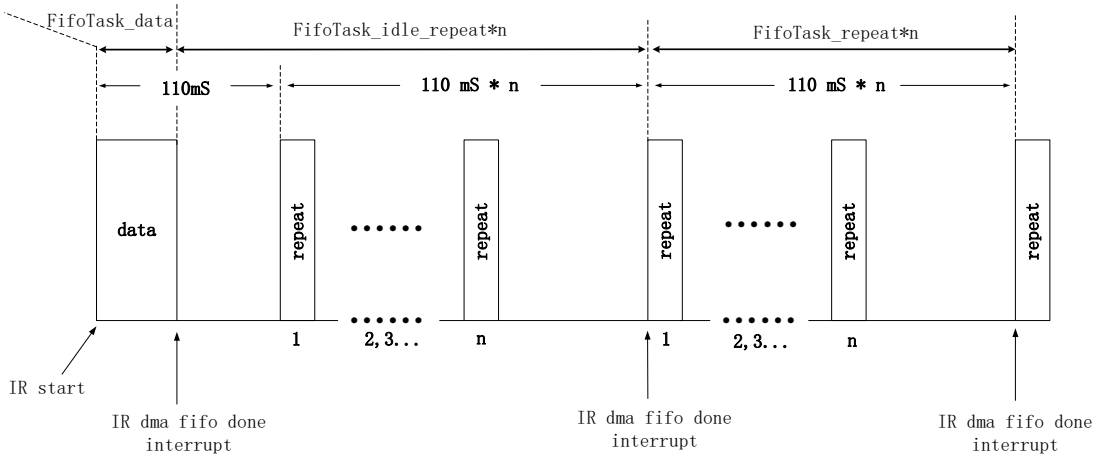


Figure 10-6 IR Timing 2

As per the IR timing design above, corresponding code in SW flow is shown as below:

At IR start, invoke the function “ir_nec_send”, enable FifoTask_data, and use interrupt to control the following flow. In the interrupt when FifoTask_data is finished, enable FifoTask_idle. In the interrupt when FifoTask_idle is finished, enable FifoTask_repeat. Before manually stopping IR DMA FIFO mode, FifoTask_repeat is executed continually.

```
void ir_nec_send(u8 addr1, u8 addr2, u8 cmd)
{
        //Add FifoTask_data to Dma
        ir_send_ctrl.is_sending = IR_SENDING_DATA;
        ir_send_ctrl.sending_start_time = clock_time();
        pwm_start_dma_ir_sending();
}


void rc_ir_irq_prc(void)
{
    if(reg_pwm_irq_sta & FLD_IRQ_PWM0_IR_DMA_FIFO_DONE)
    {
        reg_pwm_irq_sta = FLD_IRQ_PWM0_IR_DMA_FIFO_DONE;

        if(ir_send_ctrl.repeat_enable){
            if(ir_send_ctrl.is_sending == IR_SENDING_DATA){
                ir_send_ctrl.is_sending = IR_SENDING_REPEAT;
                //Add FifoTask_idle_repeat*n to Dma
                pwm_start_dma_ir_sending();
            }
```

```
            else if(ir_send_ctrl.is_sending == IR_SENDING_REPEAT){
                //Add FifoTask_repeat*n to Dma
                pwm_start_dma_ir_sending();
            }
        }
        else{
            ir_send_release();
        }
    }
}
```

10.2.4 IR Initialization

10.2.4.1 rc_ir_init

IR initialization function is shown as below. Please refer to demo code in SDK.

```
            void rc_ir_init(void)
```

10.2.4.2 IR Hardware Configuration

The demo code is as below:

```
    pwm_n_revert(PWM0_ID);
    gpio_set_func(GPIO_PA0, AS_PWM0);
    pwm_set_mode(PWM0_ID, PWM_IR_DMA_FIFO_MODE);
    pwm_set_phase(PWM0_ID, 0);    //no phase at pwm beginning
    pwm_set_cycle_and_duty(PWM0_ID, PWM_CARRIER_CYCLE_TICK,
                                    PWM_CARRIER_HIGH_TICK );
    pwm_set_dma_address(&T_dmaData_buf);
    reg_irq_mask |= FLD_IRQ_SW_PWM_EN;
    reg_pwm_irq_sta = FLD_IRQ_PWM0_IR_DMA_FIFO_DONE;
```

Since only PWM0 supports ID DMA FIFO mode, PA0 is selected to correspond to PWM0 herein.

In the demo, IR carrier frequency is 38K, cycle is 26.3us, and duty cycle is 1/3. The API “pwm_set_cycle_and_duty” should be used to configure the cycle and duty cycle. Since all FifoTasks share the same carrier frequency, the carrier of 38K can meet the configuration requirement, and PWM shadow mode is not needed.

DMA FIFO buffer is “T_dmaData_buf”。

Enable system interrupt mask “FLD_IRQ_SW_PWM_EN”.

Clear system status “FLD_IRQ_PWM0_IR_DMA_FIFO_DONE”.

10.2.4.3 IR Variable Initialization

Related variables in the SDK demo includes waveform_start_bit_1st, waveform_start_bit_2nd, and etc.

As introduced in IR timing design, FifoTask_data and FifoTask_repeat should be configured.

Start signal = 9ms carrier signal + 4.5ms low level signal (no carrier). Call “pwm_config_dma_fifo_waveform” to configure the two corresponding DMA FIFO data.

```
    //start bit, 9000 us carrier,   4500 us low

    waveform_start_bit_1st = pwm_config_dma_fifo_waveform(1,
PWM0_PULSE_NORMAL, 9000 *
CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

    waveform_start_bit_2nd = pwm_config_dma_fifo_waveform(0,
PWM0_PULSE_NORMAL, 4500 *
CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

 u16 waveform_stop_bit_2nd;
```

The method also applies to configure stop signal, repeat signal, data logic “1” signal, and data logic “0” signal.

10.2.5 FIFO Task Configuration

10.2.5.1 FIFO Task_data

As per demo IR protocol, to send a cmd (e.g. 7), first send start signal, i.e. 9ms carrier signal + 4.5ms low level signal (no carrier); then send “address+ ~address+ cmd + ~cmd”. In the demo code, address is 0x88.

When sending the final bit of “~cmd”, logical “0” or logical “1” always contains some non-carrier signals at the end. If “~cmd” is not followed by any data, there may be a problem on Rx side: Since there’s no boundary to differentiate carrier, the FW does not know whether the non-carrier signal duration of the final bit is 560us or 1690us, and fails to recognize whether it’s logical “0” or logical “1”.

To solve this problem, the Data signal should be followed by a “stop” signal which is defined as 560us carrier signal + 500us non-carrier signal.

Thus, the FifoTask_data mainly contains the three parts below:

1) start signal: 9ms carrier signal + 4.5ms low level signal (no carrier)
2) data signal: address+ ~address+ cmd + ~cmd
3) stop signal: 560us carrier signal + 500us non-carrier signal

The code below is used to configure DMA Fifo buffer and start IR transmission.

```
//// set waveform input in sequence //////

T_dmaData_buf.data_num = 0;


//waveform for start bit
```

```
T_dmaData_buf.data[T_dmaData_buf.data_num ++]=
waveform_start_bit_1st;

T_dmaData_buf.data[T_dmaData_buf.data_num ++]=
waveform_start_bit_2nd;


//add data

u32 data = (~cmd)<<24 | cmd<<16 | addr2<<8 | addr1;

for(int i=0;i<32;i++){

    if(data & BIT(i)){

                //waveform for logic_1

        T_dmaData_buf.data[T_dmaData_buf.data_num ++] =

                                            waveform_logic_1_1st;

        T_dmaData_buf.data[T_dmaData_buf.data_num ++] =

                                            waveform_logic_1_2nd;

            }

            else{

                //waveform for logic_0

                T_dmaData_buf.data[T_dmaData_buf.data_num ++] =

                                            waveform_logic_0_1st;

                T_dmaData_buf.data[T_dmaData_buf.data_num ++] =

                                            waveform_logic_0_2nd;

            }

        }


//waveform for stop bit

T_dmaData_buf.data[T_dmaData_buf.data_num ++] =
waveform_stop_bit_1st;

T_dmaData_buf.data[T_dmaData_buf.data_num ++] =
waveform_stop_bit_2nd;


T_dmaData_buf.dma_len = T_dmaData_buf.data_num * 2;

……

    pwm_start_dma_ir_sending();
```

10.2.5.2 FifoTask_idle

As introduced in IR timing design, FifoTask_idle lasts for the duration “110mS – T_data”.

Record the time when FifoTask_data starts:

```
    ir_send_ctrl.sending_start_time = clock_time();
```

Then calculate FifoTask_idle time in the interrupt triggered by completion of FifoTask_data:

110mS – (clock_time() - ir_send_ctrl.sending_start_time)

Demo code:

```
u32 tick_2_repeat_sysClockTimer16M = 110*CLOCK_16M_SYS_TIMER_CLK_1MS
-
                    (clock_time() - ir_send_ctrl.sending_start_time);
u32 tick_2_repeat_sysTimer =
              (tick_2_repeat_sysClockTimer16M*CLOCK_SYS_CLOCK_1US>>4);
```

Please pay attention to time unit switch. As introduced in Clock module, System Timer frequency used in software timer is fixed as 16MHz. Since PWM clock is derived from system clock, user needs to consider the case with system clock rather than 16MHz (e.g. 32MHz, 48MHz).

FifoTask_idle does not send PWM waveform, which can be considered to continually send non-carrier signal. It can be implemented by setting the first parameter “carrier_en” of the API “pwm_config_dma_fifo_waveform” to 0.

```
    waveform_wait_to_repeat = pwm_config_dma_fifo_waveform(0,
PWM0_PULSE_NORMAL, tick_2_repeat_sysTimer/PWM_CARRIER_CYCLE_TICK);
```

10.2.5.3 FifoTask_repeat

As per Demo IR protocol, repeat signal is 9ms carrier signal + 2.25ms non-carrier signal.

Similar to the processing of FifoTask_data, the end of repeat signal should be followed by 560us carrier signal as stop signal.

As introduced in IR timing design, repeat signal lasts for 110ms, so the duration of non-carrier signal after the 560us carrier signal should be:

110mS – 9mS – 2.25mS – 560uS = 99190uS

The code below shows the configuration for a complete repeat signal:

```
//repeat signal   first part, 9000 us carrier, 2250 us low

    waveform_repeat_1st = pwm_config_dma_fifo_waveform(1,
PWM0_PULSE_NORMAL, 9000 *
CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

    waveform_repeat_2nd = pwm_config_dma_fifo_waveform(0,
PWM0_PULSE_NORMAL, 2250 *
CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);


//repeat signal   second part,   560 us carrier, 99190 us low

    waveform_repeat_3rd = pwm_config_dma_fifo_waveform(1,
PWM0_PULSE_NORMAL, 560 * CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);

    waveform_repeat_4th = pwm_config_dma_fifo_waveform(0,
PWM0_PULSE_NORMAL, 99190 *
CLOCK_SYS_CLOCK_1US/PWM_CARRIER_CYCLE_TICK);
```

```
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_1st;
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_2nd;
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_3rd;
T_dmaData_buf.data[T_dmaData_buf.data_num ++] = waveform_repeat_4th;
```

10.2.5.4 FifoTask_repeat*n&FifoTask_idle_repeat*n

By simple superposition in DMA Fifo buffer, “FifoTask_repeat*n” and “FifoTask_idle_repeat*n” can be implemented on the basis of FifoTask_idle and FifoTask_repeat.

10.2.6 Check IR Busy Status in APP Layer

In the Application layer, user can use the variable “ir_send_ctrl.is_sending” to check whether IR is busy sending data or repeat signal.

ir_send_ctrl.is_sending

As shown in the demo code below, when IR is busy, MCU cannot enter suspend.

```
if( ir_send_ctrl.is_sending)
{
    bls_pm_setSuspendMask(SUSPEND_DISABLE);
}
```

11. Drivers in BLE SDK

11.1 External Capacitor for 24 MHz Crystal

By default, SDK uses internal capacitor of TLSR8232 MCU (i.e. cap corresponding to ana_81<4:0>) as matching capacitor of 24MHz crystal oscillator, which is measurable and adjustable in Telink jig system to reach optimal frequency point value of final application product.

If an external soldered capacitor needs to be use as the matching capacitor of 24MHz crystal oscillator instead, the API below should be called 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 called 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 Select 32kHz Clock Sources

By default SDK uses internal 32kHz crystal, i.e. 32kHz RC. The error of this crystal is large, so its accuracy will be influenced for applications with long suspend time. Currently 32kHz RC supports up to 3s suspend by default. Once the suspend time exceeds 3s, 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.

Call either of the two APIs below at the beginning of main function (must before “cpu_wakeup_init” function) to select 32kHz RC or 32kHz pad.

```
void blc_pm_select_internal_32k_crystal(void);

void blc_pm_select_external_32k_crystal(void);
```

SDK chooses 32k RC by default by calling “blc_pm_select_internal_32k_crystal”. For 32k pad, call “blc_pm_select_external_32k_crystal”.

11.3 EMI

11.3.1 EMI Test

5316 SDK provides project “EMI Test”. Users can see “app_emi.c” in the project folder of “5316_driver_test” , see “emi.c and emi.h” in driver/5316 files for related documents. To use EMI, users need to set the marco “DRIVER_TEST_MODE” in “app_config.h to “TEST_RF_EMI”.

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_s  ate_list[] = {
    {0x01,emicarrieronly},
    {0x02,emi_con_prbs9},
    {0x03,emirx},
    {0x04,emitxprbs9},
    {0x05,emitx55},
    {0x06,emitx0f},
};
```

11.3.1.1 Carrier Mode

Carrier mode is used to test the sending of carrier. The function is:

```
void emicarrieronly( RF_ModeTypeDef rf_mode,
                     RF_TxPowerTypeDef pwr,
                     signed char rf_chn);
```

Parameters:

rf_mode	Set RF mode, RF_MODE_BLE_1M and RF_MODE_BLE_2M are optional
pwr	Set tx power, see the definition of RF_TxPowerTypeDef for its values.
rf_chn	Set RF channel, the channel actually used is (2400+rf_chn).

11.3.1.2 CD Mode

CD mode is used to test the sending of carrirer with data. The function is:

```
void emi_con_prbs9(RF_ModeTypeDef rf_mode,
                   RF_TxPowerTypeDef pwr,
                   signed char rf_chn)
```

Parameters:

rf_mode	Set RF mode, RF_MODE_BLE_1M and RF_MODE_BLE_2M are optional
pwr	Set tx power, see the definition of RF_TxPowerTypeDef for its values.

rf_chn	Set RF channel, the channel actually used is (2400+rf_chn).

11.3.1.3 TX Mode

Tx mode can send packets of three different types, PRBS9 packet payload, 00001111 packet payload, 10101010 packet payload. Users can choose different Tx modes by cmd.

To send the three packets, the functions below should be called:

```
void emitxprbs9(RF_ModeTypeDef rf_mode,
                RF_TxPowerTypeDef pwr,
                signed char rf_chn);
void emitx0f( RF_ModeTypeDef rf_mode,
              RF_TxPowerTypeDef pwr,
              signed char rf_chn)
void emitx55( RF_ModeTypeDef rf_mode,
              RF_TxPowerTypeDef pwr,
              signed char rf_chn)
```

Parameters:

rf_mode	Set RF mode, RF_MODE_BLE_1M and RF_MODE_BLE_2M are optional.
pwr	Set tx power, see the definition of RF_TxPowerTypeDef for its values.
rf_chn	Set RF channel, the channel actually used is (2400+rf_chn).

11.3.1.4 RX Mode

Rx mode receives data and records the number of received data by poll. The function is:

```
void emirx(RF_ModeTypeDef rf_mode,
           RF_TxPowerTypeDef pwr,
          signed char rf_chn);
         signed char rf_chn)
```

Parameters:

rf_mode	Set RF mode, RF_MODE_BLE_1M and RF_MODE_BLE_2M are optional.
pwr	Set tx power, see the definition of RF_TxPowerTypeDef for its values.
rf_chn	Set RF channel, the channel actually used is 2400+rf_chn).

Note: EMI test is a demo project provided by Telink. Users can use this project to complete their own testing, and customize their own testing methods by calling related functions.

11.3.2 EMI Test Tool

Telink provides EMI Test Tool in order to facilitate testing. “EMI Test Tool” can be used with “EMI test” in project “5316_driver_test” to implement EMI test easily. The tool interface is shown as below.

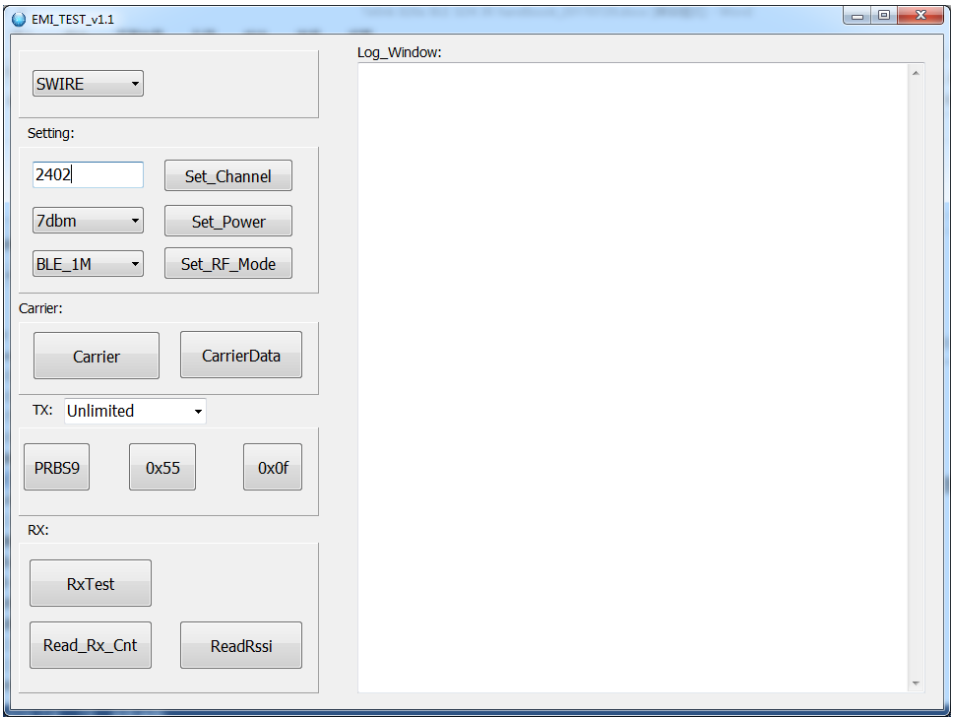


Figure 11-1 EMI Test Tool

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.

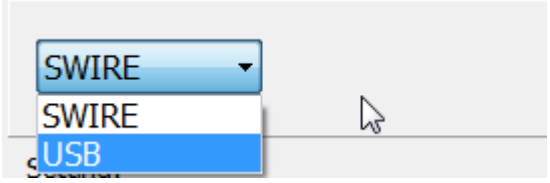


Figure 11-2 Select Data Bus

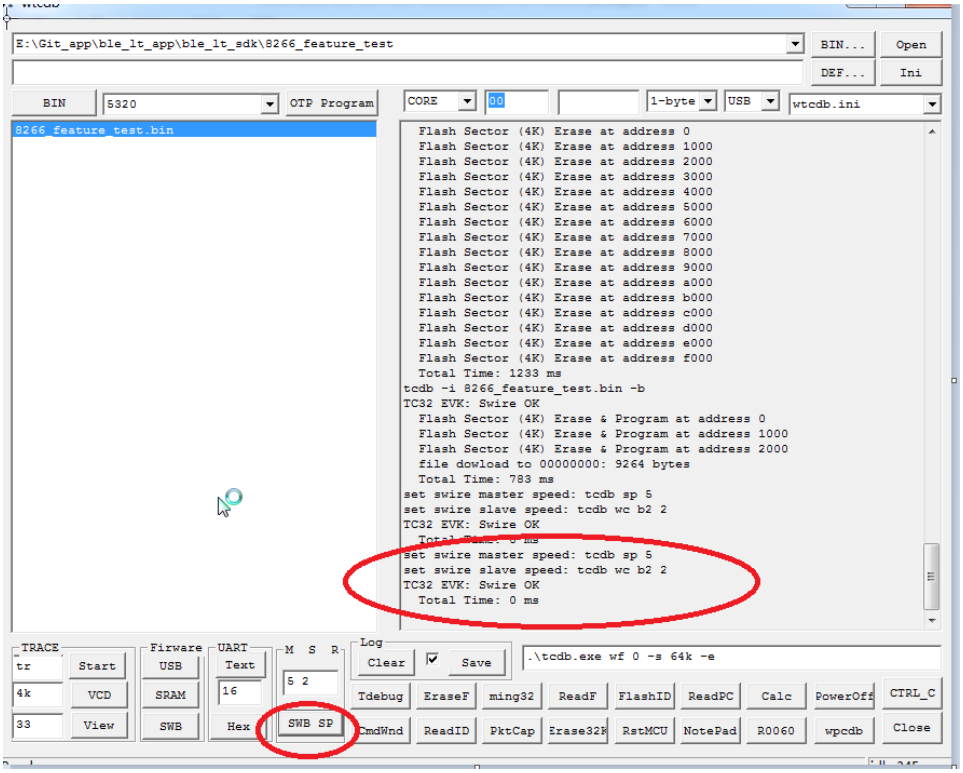


Figure 11-3 Swire synchronization operation

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.

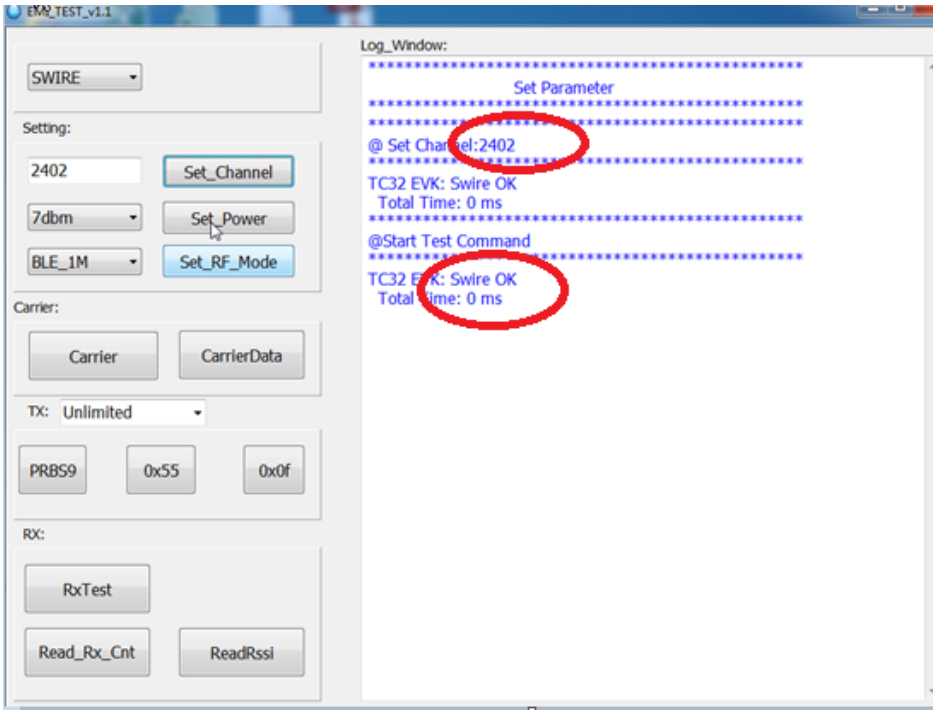


Figure 11-4 Set Channel

Step 3: Select power level and BLE mode via the corresponding drop-down box, and click “Set_Power”/”Set_RF_Mode”.

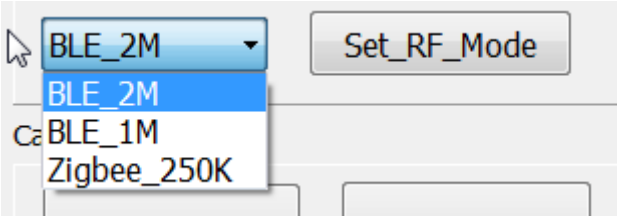


Figure 11-5 Select RF Mode

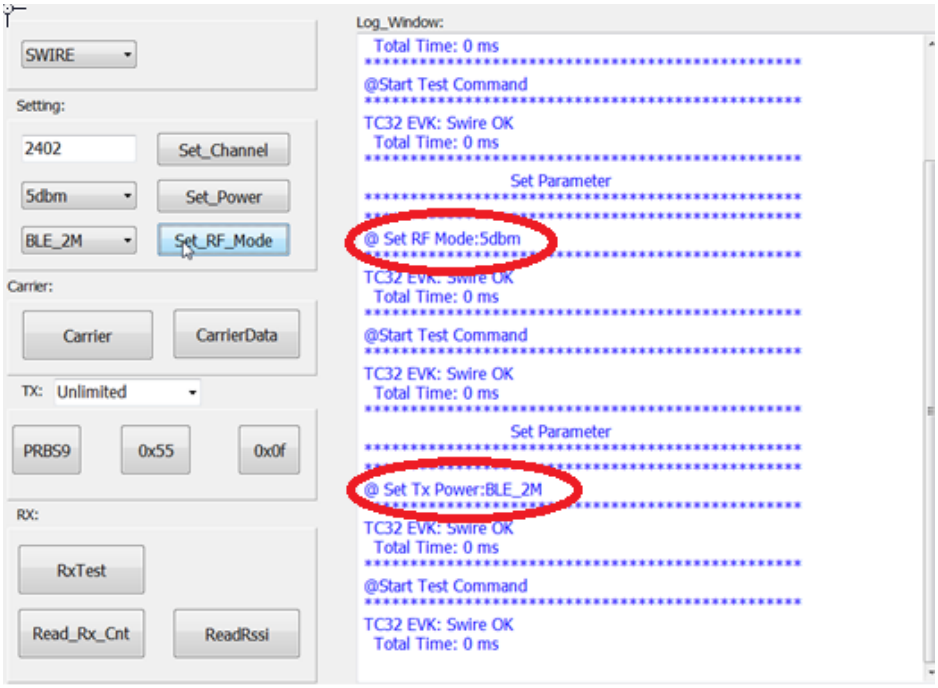


Figure 11-6 Interface After RF Mode Setting

Step 4: Click “Carrier”/“CarrierData”/“RXTest”/“PRBS9”/“0x55”/“0x0f” to enter corresponding test mode.

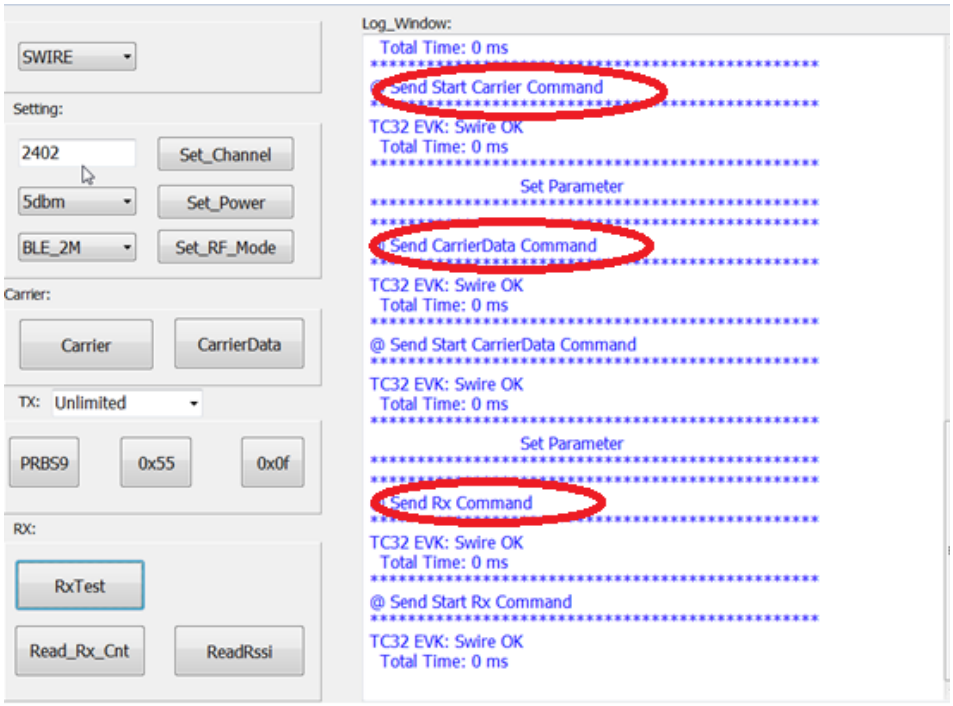


Figure 11-7 Select Test Mode

Step 5: In TX mode, user can select to send 1000 packets or unlimited packets.

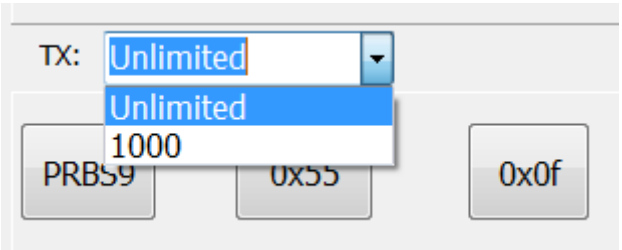


Figure 11-8 Set TX Packet Number

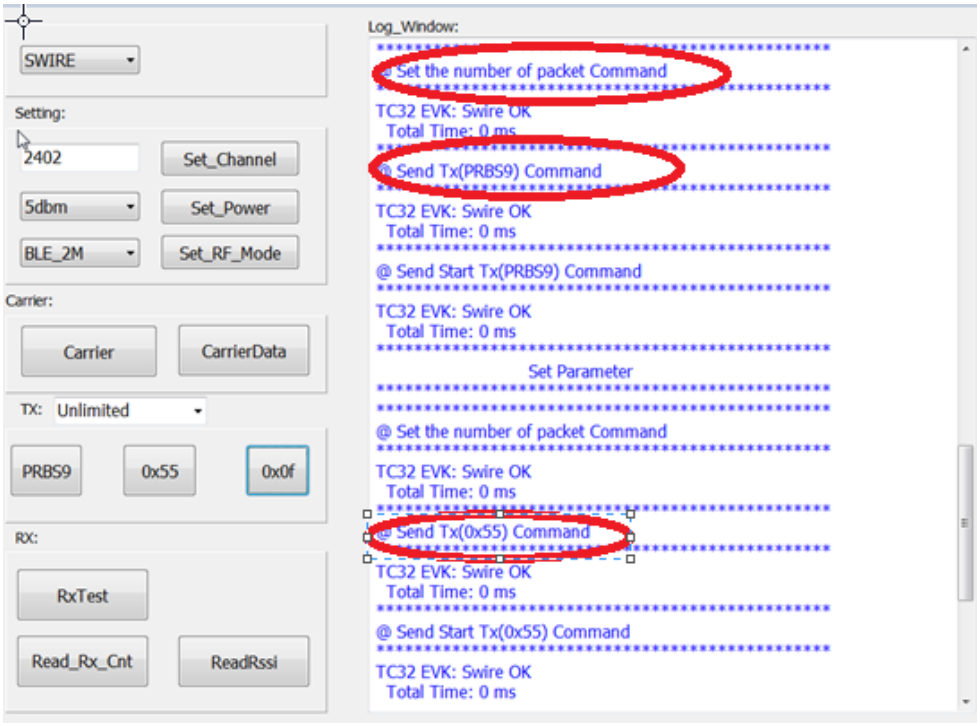


Figure 11-9 TX Mode Interface

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.

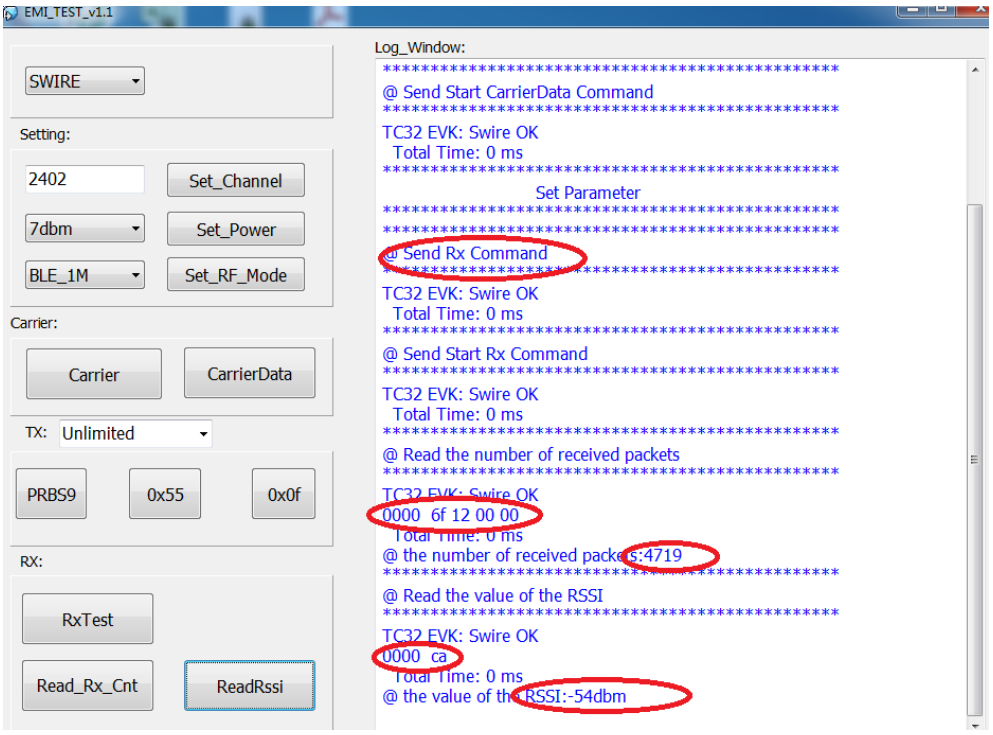


Figure 11-10 Read RX Packet Number and RSSI

11.4 PHY Test

To be added.

12. BLE SPP Module

Telink's BLE module provides SPP (Serial Port Profile) data transmission, therefore users do not need to care about the implementation details of BLE and can focus on the data.

12.1 Command and Data Packet Format

Telink BLE module use UART interface to communicate with Host. Telink provides corresponding commands and events for BLE module for users.

Table 12-1 BLE SPP Command (From Host to Telink BLE module)

Field	Bytes	Description
cmdID	2	Command ID
ParaLen	2	Length of parameters
Parameters	n	Parameter payload

Table 12-2 BLE SPP Events (From Telink BLE Module to Host)

Field	Bytes	Description
Token	1	Always be 0xFF
ParaLen	1	Length of parameters
eventId	2	Event ID
Parameters	n	Parameter payload

BLE SPP **command** indicates a command sent from the Host to Telink BLE module, while BLE SPP **event** indicates a synchronous event or an asynchronous event sent from Telink BLE module to the Host.

A **synchronous event** is the acknowledgement event corresponding to a command, e.g. "Command Complete event".

An **asynchronous event** is the report event to indicate that something has happened to Telink BLE module, e.g. data has been received ("Data Received event").

Please refer to Table 12-3 and Table 12-4 for general commands and events.

Table 12-3 General AT Command Set

CmdFunction	cmdID	ParaLen	Parameters	Format	Corresponding event
Set Advertising Interval	0xFF01	0x0002	Interval: e.g. 0x0050 Advertising interval = 80* 0.625ms = 50ms	e.g. 01 FF 02 00 50 00	Command Complete event
Set Advertising Data	0xFF02	<=0x0010 e.g. 0x0006	Data set: e.g. 01 02 03 04 05 06	e.g. 02 FF 06 00 01 02 03 04 05 06	Command Complete event
Enable/Disable Advertising	0xFF0A	0x0001	Enable: 0x01 Disable: 0x00	e.g. 0A FF 01 00 01	Command Complete event
Get Module Available Data Buffer	0xFF0C	0x0000	NA	0C FF 00 00	Command Complete event
Set Advertising Type	0xFF0D	0x0001	0x00: connectable undirected adv 0x01: connectable directed adv 0x02: scannable undirected adv 0x03: non-connectable adv	e.g. 0D FF 01 00 00	Command Complete event
Set Advertising Direct Address	0xFF0E	0x0007	**Parameter 1: dirAddrType** 0x00: Public Address 0x01: Random Address **Parameter 2: dirAddress** e.g. 01 02 03 04 05 06	e.g. 0E FF 07 00 00 01 02 03 04 05 06	Command Complete event

CmdFunction	cmdID	ParaLen	Parameters	Format	Corresponding event
Add White List Entry	0xFF0F	0x0007	**Parameter 1: addrType** 0x00: Public Address 0x01: Random Address **Parameter 2: address** e,g, 01 02 03 04 05 06	0F FF 07 00 00 01 02 03 04 05 06	Command Complete event
Delete White List Entry	0xFF10	0x0007	**Parameter 1: addrType** 0x00: Public Address 0x01: Random Address **Parameter 2: address** e.g. 01 02 03 04 05 06	e.g. 10 FF 07 00 00 01 02 03 04 05 06	Command Complete event
Reset White List Entry	0xFF11	0x0000	NA	11 FF 00 00	Command Complete event
Set Filter Policy	0xFF12	0x0001	00: All device 01: connReq from all device, scanReq from WL 02: scanReq from all device, connReq from WL 03: scanReq and connReq from WL	e.g. 12 FF 01 00 00	Command Complete event
Set device name	0xFF13	<=0x0012 e.g. 0x000A	Device name in hex format e.g. 01 02 03 04 05 06 07 08 09 0A	13 FF 0A 00 01 02 03 04 05 06 07 08 09 0A	Command Complete event
Get connection parameters	0xFF14	0x0000	NA	14 FF 00 00	Command Complete event

CmdFunction	cmdID	ParaLen	Parameters	Format	Corresponding event
Set connection parameters	0xFF15	0x0008	Parameters: u16 intervalMin; e.g. 0x00A0 means current device accepts minimum connection interval 0xA0*1.25ms=200ms u16 intervalMax; e.g. 0x00A2 means current device accepts maximum connection interval 0xA2*1.25ms=202.5ms u16 connLatency; e.g. 0x0000 means current device expects new latency 0x00 u16 connTimeout; e.g. 0x012C means current device expects new timeout 0x12C*10ms=3000ms	15 FF 08 00 A0 00 A2 00 00 00 2C 01	Command Complete event
Get module's current work state	0xFF16	0x0000	NA	16 FF 00 00	Command Complete Event
Terminate connection	0xFF17	0x0000	NA	17 FF 00 00	Command Complete Event
Restart Module	0xFF18	0x0000	NA	18 FF 00 00	No Command Complete Event
Enable or Disable MAC Binding Function	0xFF19	0x0001	**fEnable** '0x01': enable MAC binding function. After MAC binding is enabled, only the devices with MAC address in the MAC table	19 FF 01 00 01/00	Command Complete Event

CmdFunction	cmdID	ParaLen	Parameters	Format	Corresponding event
			can be connected with this module. '0x00': disable MAC binding function		
Add device's MAC address to MAC binding table	0xFF1A	0x0006	**MacAddr** MAC address to be added into MAC binding table. Note: MAC table supports public MAC address only. e.g. B4 CE BF 01 E7 60	e.g. 1A FF 06 00 B4 CE BF 01 E7 60	Command Complete Event
Delete a MAC item from MAC binding table	0xFF1B	0x0006	**MacAddr** MAC address to be deleted from MAC binding table. e.g. B4 CE BF 01 E7 60	e.g. 1B FF 06 00 B4 CE BF 01 E7 60	Command Complete Event
Send Data	0xFF1C	<=0x0016 e.g. 0x0007	**Handle（2 bytes）of the Attribute "Service to client"** E.g. 0x0011 **Data payload（Max Len: 20bytes）** E.g. 01,02,03,04,05	e.g. 1C FF 07 00 11 00 01 02 03 04 05	Command Complete Event

Table 12-4 General Events

Event name	Type	Token	ParaLen	EventID	Parameters	Format
Command Complete event	Synchronous	0xFF	0x03	corresponding to the command **Rule:** **eventID = (cmdID & 0x03FF) | 0x0400**	Indicates status information Success: 0x00 Others: ble error code, @ble_sts_t	e,g. FF 03 01 07 00

Event name	Type	Token	ParaLen	EventID	Parameters	Format
				e,g, 0x0701 (cmdID=0xFF01, corresponding to the “Set Advertising Interval” command)		
Data received event	Asynchronous	0xFF	Variable: n+2 Eg.If n=6, Len is 0x08	0x07A0	**data** (indicates received data, n bytes) Eg. 01, 02, 03, 04, 05, 06	e.g. FF 08 A0 07 01 02 03 04 05 06
Get Available Buffer Num Event	Asynchronous	0xFF	0x04	0x070C	**State**: 0x00: OK other: Fail **Buffer Size**: 1Byte Eg: state = 0x00 Buffer Size = 0x04	e.g. FF 03 0C 07 00 04
Connection Event	Asynchronous	0xFF	0x02	0x0783	NA	FF 02 83 07
Terminate Event	Asynchronous	0xFF	0x02	0x0784	NA	FF 02 84 07
Channel map change Event	Asynchronous	0xFF	0x02	0x078a	NA	FF 02 8a 07
Connection parameter update Event	Asynchronous	0xFF	0x02	0x078b	NA	FF 02 8b 07
Get Module Current Work State	Asynchronous	0xFF	0x04	0x0716	No connected: 0x0100	e.g. FF 04 0716 00 01

Event name	Type	Token	ParaLen	EventID	Parameters	Format
					Connected: 0x0800	

12.2 Function Description

To illustrate function of the module, conection between the module and a phone is taken an example. The app in the phone is LightBlue, the software in PC is a serial port assistant. The hardware connection is as below.

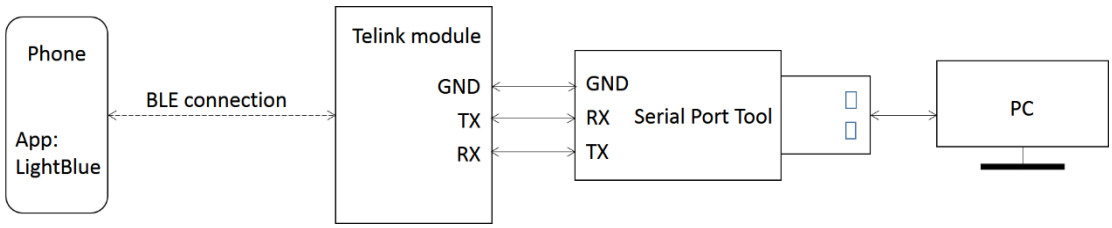


Figure 12-1 Module Hardware Connection

As Figure 12-1 shows, Telink module connects serial port tool via wires, serial port tool connects PC via USB, the serial port assistant in PC establishes soft connection with serial port tool, thus the working environment of the module is established. Before using the module function, the Firmware should be burned to the module. 5316 BLE SDK provides module project, users only need to compile the module project to obtain the bin file (Note: the bin file in this documents sets `BLE_MODULE_PM_ENABLE to 0 to disable low power, then` use EVK to burn the bin file to telink module. After the Module powers on, it sends Adv packets by default, the phone enables Bluetooth and enters LightBlue, starts scan. The phone scans the module device (see Figure 12-2) and establishes connection with BLE (see Figure 12-3).

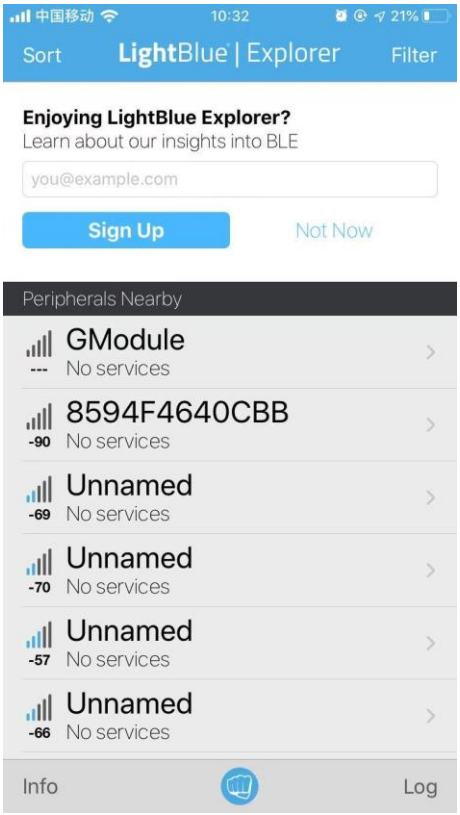


Figure 12-2 Scan Module Device

Figure 12-3 Connect Module Device

After Telink module establishes BLE connection with the phone, in the phone App, users can see two “Telink SPP”, one is “Module->Phone” used for sending data to the phone from the module, another “Phone->Module” used for sending data to the module from the phone. From the descriptors the data transfer directions are easy to know.

12.2.1 Module Sends Commands and Data

Data interconnection between the PC and the module is via the serial port. Users can modify the parameters of the serial port (baud rate, odd-even check, stop bit, etc.) in the module project as needed. The PC can transfer commands and data to the module, while the module can only report corresponding events to the PC. If it is commands that are transferred by the PC, the module will process the commands and report an event to the PC according to the processing result; if it is data that are transferred by the PC, the module will transfer the data to a remote device via BLE and report an event. Please see section 12.1 for formats of commands, data and events. Figure 12-4 and Figure 12-5 show the demo of the module sending data to the phone. The part marked by “1” in Figure 12-4 is the data sent by the module to the phone via command “Send Data” (cmd ID:0xFF1C); the part marked by “2” is the event reported by the module, which indicates the success of data transmission. Figure 12-5 shows the data received by the phone are the same as the data sent by the module.

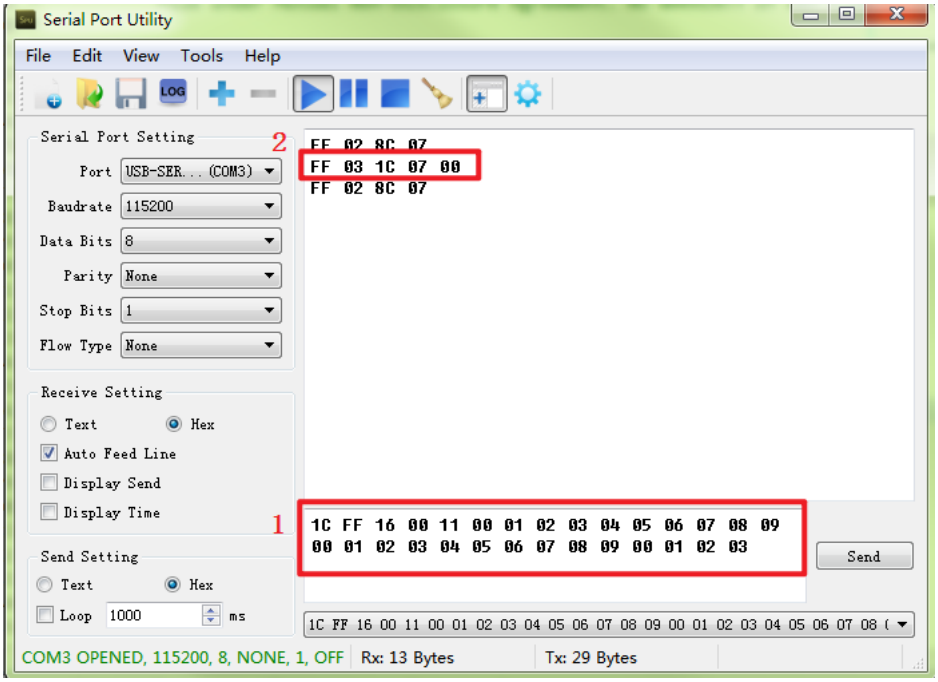


Figure 12-4 Module Sending Data

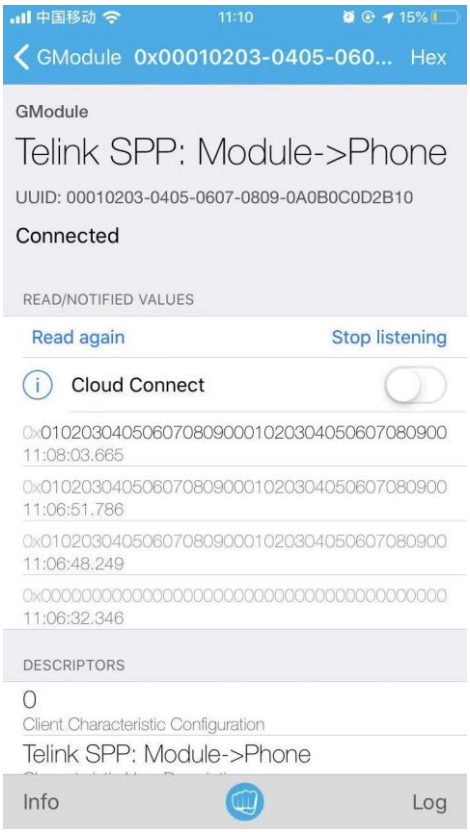


Figure 12-5 Phone Receiving Data

12.2.2 Module Receives Data

The phone can send data to the module, the module will report to the PC after it receives the data, as Figure 12-6 and Figure 12-7 show. The part highlighted by the red rectangle

in Figure 12-6 is the data sent by the phone; the part highlighted by the red rectangle in Figure 12-7 is the data received by the module.

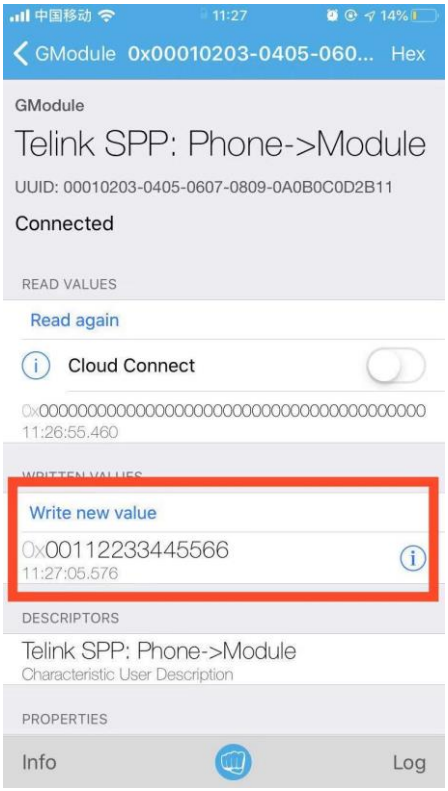


Figure 12-6 Phone Sending Data

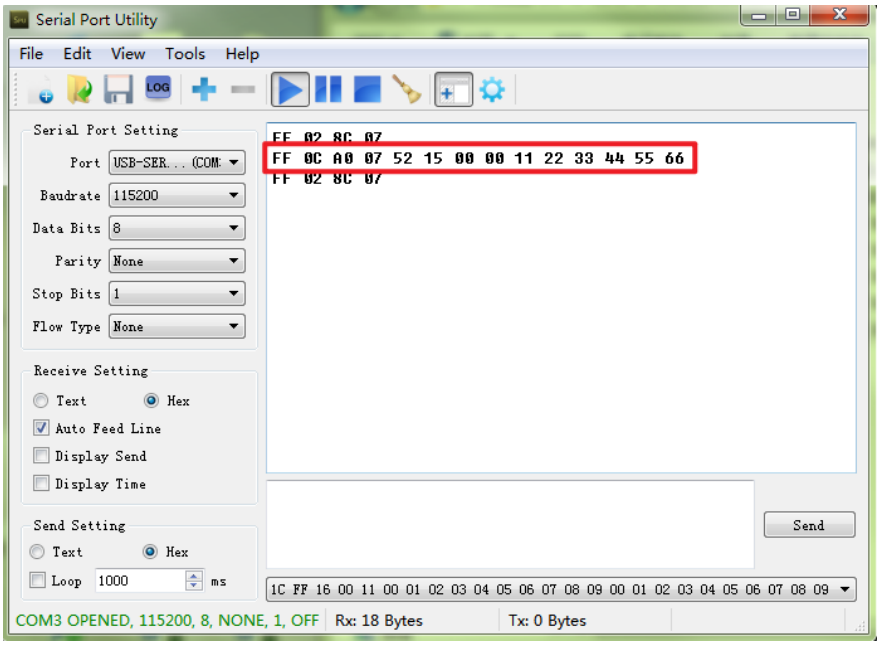


Figure 12-7 Module Receiving Data

12.3 Power Management of Module

The 5316 module project enables low power by default, as the sleep mode of Telink MCU can only be woken up by GPIO, to implement the low power and the normal interconnection of data and commands between the module and the Host, the module project uses two GPIOs as wakeup sources which can be selected according to users' requirements and hardware design. One GPIO is used to wake up the module so that the module can receive the commands and data sent by the Host; another GPIO is used to wake up the Host (if the Host enables low power) so that the Host can receive the events reported by the module.

If the module enables low power mode, the connection of hardware should be the same as below:

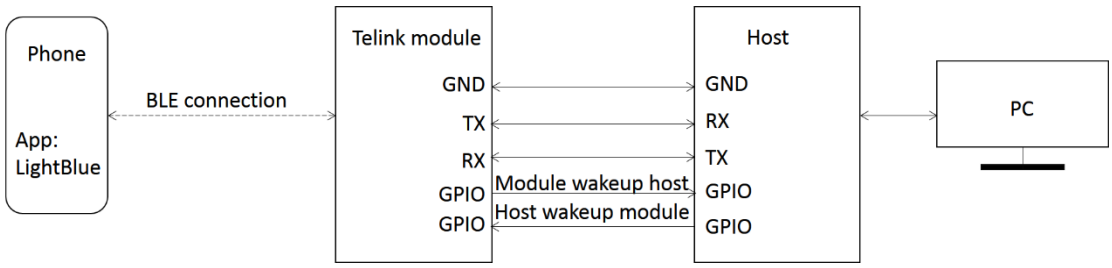


Figure 12-8 Connection of Hardware When Low Power is Enabled

If users do not use low power, the macro "BLE_MODULE_PM_ENABLE" in "app_config.h" can be set to 0, the connection of hardware is the same as the demo in section 12.2 Function Description.

Appendix

Appendix 1: crc16 Algorithm

```
unsigned shortcrc16 (unsigned char *pD, int len)
{
    static unsigned short poly[2]={0, 0xa001};
    unsigned short crc = 0xffff;
    unsigned char ds;
    int i,j;

    for(j=len; j>0; j--)
    {
        unsigned char ds = *pD++;
        for(i=0; i<8; i++)
        {
            crc = (crc >> 1) ^ poly[(crc ^ ds ) & 1];
            ds = ds >> 1;
        }
    }

    return crc;
}
```