RTL8762D Memory User Guide

V1.0

2020/06/05

Revision History

Date	Version	Comments	Author	Reviewer
2020/06/05	V1.0	First release version	Grace	Lory

Contents

Revision History 2
Figure List 4
Table List 5
1 Overview 6
2 ROM 6
3 RAM 6
3.1 Data RAM 7
3.2 Buffer RAM 9
3.3 APIs 10
3.4 Memory Usage Calculation 11
3.4.1 Statistics of Static Zone Size on Data RAM 11
3.4.2 Statistics of Static Zone Size on Cache Shared RAM Space 12
3.4.3 Statistics of Remaining Heap Size 12
3.5 Total RAM Size Available for APP Configuration 12
4 Cache 13
5 External Flash 14
5.1 Flash Layout 14
5.2 Flash APIs 22
5.3 FTL 23
5.3.1 FTL Storage Space 23
5.3.2 Adujust FTL Space Size 24
6 Flash Code and RAM Code Setting 25
7 eFuse 26

Figure List

Figure 3-1 Data RAM Layout 7
Figure 3-2 Adjust Data RAM Layout 9
Figure 3-3 Buffer RAM Layout 9
Figure 5-1 Flash Layout 14

Table List

Table 1-1 Memory Layout 6
Table 3-1 Data RAM Usage 7
Table 3-2 Buffer RAM Usage 9
Table 3-3 os_mem_alloc 10
Table 4-1 Configure Cache Usage 13
Table 5-1 Flash Section 14

1 Overview

This document describes memory system of RTL8762D and introduces how to use them. RTL8762D memory consists of ROM, RAM, external SPI Flash and eFuse, as is shown in Table 1-1. Cache has dedicated RAM, and the dedicated RAM also can be configured as general RAM. This flexible memory configuration mechanism makes RTL8762D support a wide range of applications.

Table 1-1 Memory Layout

Memory Type	Start Addr	End Addr	Size(bytes)
ROM	0x0	0x00040000	256K
Data RAM	0x00200000	0x00224000	144K
Cache (Shared as Data RAM)	0x00224000	0x00228000	16K
Buffer RAM	0x00280000	0x00288000	32K
SPIC Flash (Cacheable)	0x00800000	0x02800000	32M
SPIC Flash (Non Cacheable)	0x04800000	0x06800000	32M

2 ROM

The ROM code is located at [0x0, 0x40000), in which Bootloader, RTOS, BT Stack, Flash Driver and other platform modules are built in. RTL8762D opens some modules such as RTOS, BT Stack for application to use. RTL8762D SDK contains the header files of these ROM modules which enables users to access built in ROM functions. This reduces both application code size and RAM size.

3 RAM

RTL8762D has two pieces of RAM, the Data RAM located at [0x00200000, 0x00224000) and the Buffer RAM located at [0x00280000, 0x00288000). Both RAM memories can be used to store data and execute code. In the

current SDK, data RAM has been used for BT stack and running RAM code, while Buffer RAM has been used as heap for buffer of BT stack, log buffer, task stack and FTL mapping table.

3.1 Data RAM

In the SDK, Data RAM is divided into 6 parts by default, as is shown in Figure 3-1.

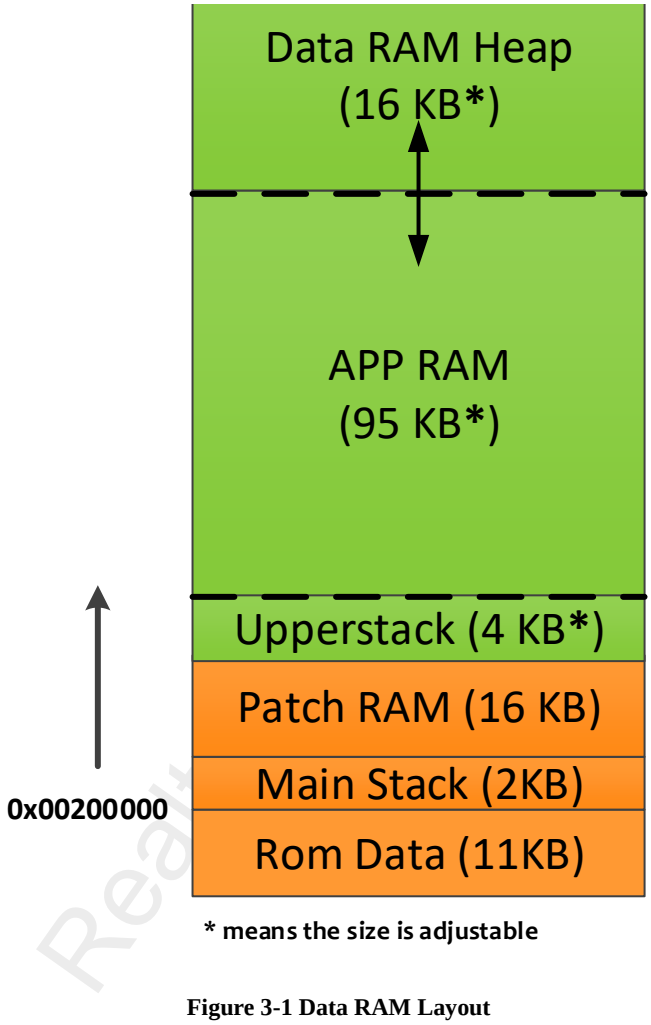


Figure 3-1 Data RAM Layout

Each part has its dedicated usage, and only APP RAM and Data RAM Heap have changeable size, as is shown in Table 3-1.

Table 3-1 Data RAM Usage

Memory Type	Memory Usage	Memory size

		changeable or not
ROM data	for all global and static variables used by ROM code	NO
Main Stack	For the stack of boot code and ISR (interrupt service routine)	NO
Patch RAM	for global/static variables and RAM code of patch	NO
Upperstack RAM	for global/static variables and RAM code of upperstack	YES
APP RAM	for global/static variables and RAM code of APP	YES
Data RAM Heap	for dynamic memory allocation of ROM code, Patch code and APP code 10K of the 16K data RAM heap has been used by rom code, so 6K is left for application	YES

The total size of Upperstack RAM, APP RAM and data RAM heap are 115K, which is unchangeable, while the size of each block can be adjusted by modifying mem_config.h, as is shown in Figure 3-2. **UPPERSTACK_GLOBAL_SIZE value must be set according to the version of the used upstack image. Different upperstack occupy different Data RAM size, please pay attention to the upperstack release files and notes.** Adjust the value of APP_GLOBAL_SIZE according to application, and the remain size equal to DATA Heap size. It is worth noting that data heap is shared by the whole system, including ROM, patch and app. Preliminary statistics, the system has used 10KB DATA Heap by default, so it is necessary to ensure that HEAP_DATA_ON_SIZE value is not less than 10KB.

```
/*============================================================================*
 *                        data ram layout configuration
 *============================================================================*/
/* Data RAM layout:                    144K
example:
   1) reserved for rom:                13K (fixed)
   2) patch ram code:                  16K (fixed)
   3) upperstack global + ram code:    4K  (adjustable, config UPPERSTACK_GLOBAL_SIZE, must consponding to used upperstack image)
   4) app global + ram code:           95K (adjustable, config APP_GLOBAL_SIZE)
   5) Heap ON:                         16K (adjustable, defined by UPPERSTACK_GLOBAL_SIZE and APP_GLOBAL_SIZE)
*/

/** @brief data ram size for upperstack global variables and code */
#define UPPERSTACK_GLOBAL_SIZE          (4 * 1024)

/** @brief data ram size for app global variables and code, could be changed */
#define APP_GLOBAL_SIZE                 (95 * 1024)

/** @brief data ram size for heap, could be changed,
but (UPPERSTACK_GLOBAL_SIZE + APP_GLOBAL_SIZE + HEAP_DATA_ON_SIZE) must be 115k, and can't less than 10KB  */
#define HEAP_DATA_ON_SIZE               (115 * 1024 - APP_GLOBAL_SIZE - UPPERSTACK_GLOBAL_SIZE)

/** @brief shared cache ram size (adjustable, config SHARE_CACHE_RAM_SIZE: 0/8KB/16KB) */
#define SHARE_CACHE_RAM_SIZE            (8 * 1024)
/*****************************************************/
```

Figure 3-2 Adjust Data RAM Layout

3.2 Buffer RAM

In the current SDK, the address space of Buffer RAM is located in [0x00280000, 0x00288000] which is divided into 2 parts by default, as is shown in Figure 3-3. The first 2KB is used for ROM Global Data, while the other 30 KB is used as heap. In the 30KB heap, ROM occupies about 26 KB dynamic spaces, while the remaining 4KB is used by APP, shown as table 3-2.

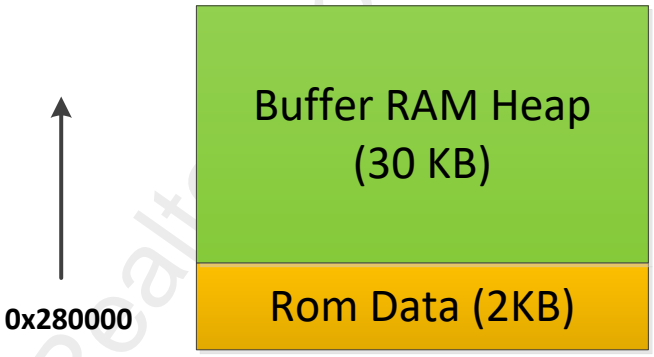


Figure 3-3 Buffer RAM Layout

Each part has its dedicated usage, as is shown in Table 3-2,

Table 3-2 Buffer RAM Usage

Memory Type	Memory Usage	Memory size changeable or not
ROM data	for all global and static variables used by ROM code	NO

Buffer RAM Heap	for dynamic memory allocation of ROM code, Patch code and APP code 26K of the 30K buffer RAM heap has been used by rom code, so 4K is left for application.	NO

3.3 APIs

os_mem_alloc can be used to allocate memory dynamically from Data RAM Heap or Buffer RAM heap according to the parameter of ram_type, as is shown in Table 3-3.

Table 3-3 os_mem_alloc

```
typedef enum
{
    RAM_TYPE_DATA_ON                = 0,
    RAM_TYPE_BUFFER_ON              = 1,
} RAM_TYPE;
/**
   * @brief    Allocate memory dynamically from Data RAM Heap or Buffer RAM heap
   * @param    ram_type : specify which heap to allocate memory from
*               size: memory size in bytes to be allocated
   * @retval pointer to the allocated memory
   */
#define os_mem_alloc(ram_type, size)    os_mem_alloc_intern(ram_type, size, __func__, __LINE__)
```

Other APIs are list as below:

- os_mem_zalloc: allocated memory will be initialized to 0
- os_mem_aligned_alloc: allocated memory will be aligned to the specified alignment
- os_mem_free: free a memory block that had been allocated from data ram heap or buffer ram heap

- os_mem_aligned_free: free an aligned memory block that had been allocated
- os_mem_peek: peek the unused memory size of the specified RAM type

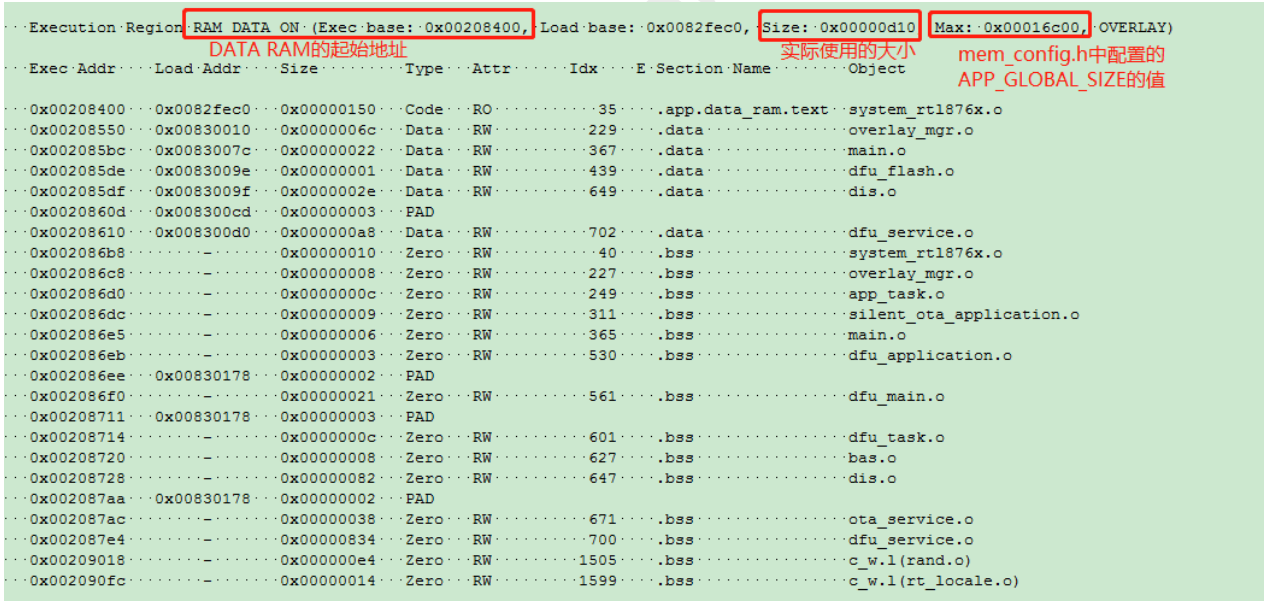
Refer to os_mem.h for more details.

3.4 Memory Usage Calculation

In order to make using of RAM more convenient, allocating memory dynamically from heap will always allocate memory from Buffer RAM firstly. If allocate fails, then it will allocate memory from Data RAM again. This allows static data and code centralization to be placed in Data RAM instead of spreading between Data RAM and Buffer RAM, thus easier to manage.

3.4.1 Statistics of Static Zone Size on Data RAM

Find the first address, the end address, and the size used in data RAM by the built "app.map" file. As shown in the figure below, checking the map of load area "RAM_DATA_ON" can determine the first address of app data RAM. Execution Region "OVERLAY_C" determines the end address of data RAM. In the following example, the actual usage size of data RAM is 0xd10, while the configured value of APP_GLOBAL_SIZE in "mem_config.h" file is 0x16c00 (91KB). APP configured Data RAM layout can be further optimized through map.



```
  Execution Region OVERLAY_A (Exec base: 0x00209110, Load base: 0x00830178, Size: 0x00000000, Max: 0xffffffff, OVERLAY)

  **** No section assigned to this execution region ****

  Execution Region OVERLAY_B (Exec base: 0x00209110, Load base: 0x00830178, Size: 0x00000000, Max: 0xffffffff, OVERLAY)

  **** No section assigned to this execution region ****

  Execution Region OVERLAY_C (Exec base: 0x00209110, Load base: 0x00830178, Size: 0x00000000, Max: 0xffffffff, OVERLAY)

  **** No section assigned to this execution region ****
```

3.4.2 Statistics of Static Zone Size on Cache Shared RAM Space

Through SHARE_CACHE_RAM_SECTION decoration can specify functions or static and global variables to share cache RAM. Find the first address (fixed to 0x00224000) and the end address allocated in the cache shared RAM by looking up built “app.map” file. In the following example, the actual use size of share cache RAM is 0x158, while the configured value of SHARE_CACHE_RAM_SIZE in “mem_config.h” file is 16KB, and the layout of APP configuration data RAM can be further optimized through map.

```
   Execution Region CACHE_DATA_ON (Exec base: 0x00224000, Load base: 0x0083001c, Size: 0x00000158, Max: 0x00004000, OVERLAY)

   Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

   0x00224000   0x0083001c   0x00000154   Code   RO          363    .app.sharecache_ram.text  main.o
   0x00224154   0x00830170   0x00000004   Data   RW          366    .app.sharecache_ram.text  main.o
```

3.4.3 Statistics of Remaining Heap Size

The remaining heap size of the specified RAM type can be obtained through the os_mem_peek function.

3.5 Total RAM Size Available for APP Configuration

Under the default config setting(max link num is 1, master num is 0, and slave num is 1), the total RAM size available for APP development is 95 (Data RAM Global) + 6 (Data RAM Heap) + 4 (Buffer RAM Heap) + 16 (Cache share ram) = 121 KB. If you want to further increase the total size of the RAM that can be used by the APP, there are two ways:

1） Configure Bluetooth as single link: Modify the “stack” page in the Config Set option in MPTool to modify the configuration as shown in Figure 3-4 to add 3KB Data RAM Heap and 1KB Buffer RAM Heap. Therefore, the total RAM size available for APP development is increased to 95 (Data RAM Global) + 9 (Data RAM Heap) + 5 (Buffer RAM Heap) + 16 (Cache share ram) =125 KB.

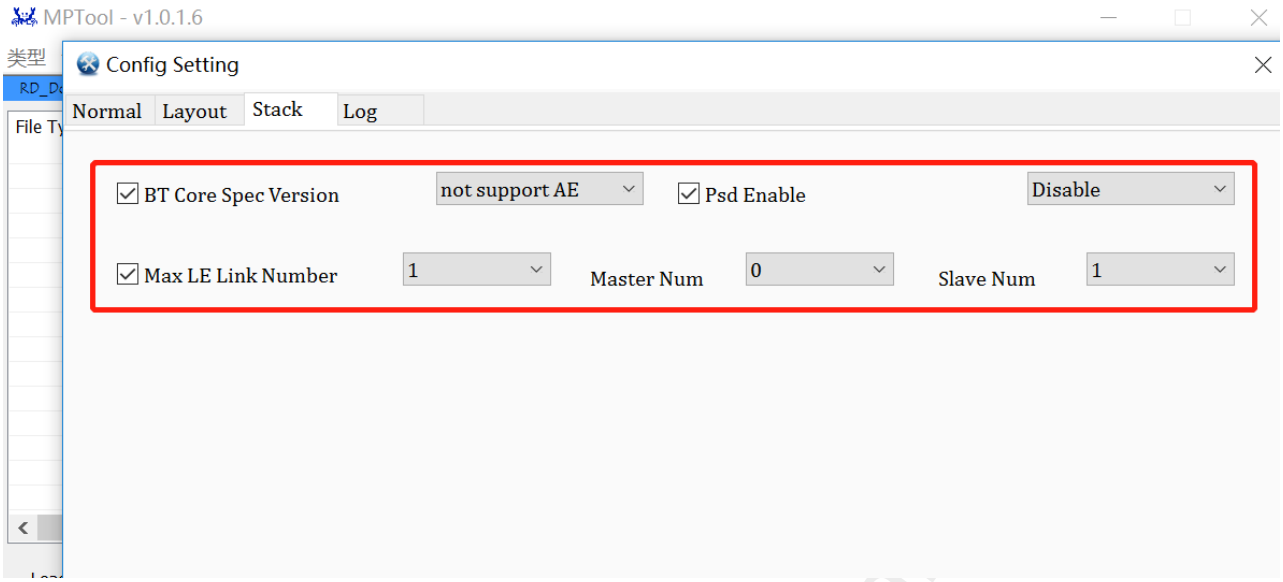


Figure 3-4 Total RAM Size Available for APP Configuration

2) Disable log function. Open the macro "RELEASE_VERSION" in the APP project header file “platform_autoconf.h” will disable log function, and APP can add an additional 3KB Buffer RAM Heap.

4 Cache

RTL8762D has a 16K bytes cache, it co-works with SPIC (SPI Flash Controller) to speed up the SPI Flash read and write operation. And it also can be used as data RAM. If it is configured as data RAM, it can be used for Data Storage or Code Execution. If Cache is configured as data RAM, its range is [0x00224000, 0x00228000). This range is just at the end of data RAM.

The Data RAM size of cache could be configured by setting SHARE_CACHE_RAM_SIZE micro in mem_config.h, as is shown in Table 4-1.

Table 4-1 Configure Cache Usage

SHARE_CACHE_RAM_SIZE	Flash Cache Size	Data RAM Size	Scenario
0 KB	16 KB	0 KB	Run large amounts of flash Code that requires a large piece of cache
8 KB	8 KB	8 KB	Run small amounts of flash code that requires a small piece of cache

16 KB	0 KB	16 KB	Not run flash code

5 External Flash

RTL8762D supports external SPI Flash by integrating a SPI Flash Controller (SPIC). Realtek offers the bottom level API of Flash driver, FTL for user application. SPIC supports memory mapping to SPI Flash on board and the maximum Flash size is 32M bytes. There are two range of memory mapping spaces. The range of [0x00800000, 0x02800000) is with cache, and [0x04800000, 0x068000000) is without cache. When enabled, cache will work while CPU accesses this space. This improves the data read and code execution efficiency of SPI Flash greatly.

5.1 Flash Layout

In the current SDK, the FLASH memory layout is summarized as in Figure 5-1 and consists of 7 fields : “Reserve”, “OEM Header”, “OTA Bank 0”, “OTA Bank 1”, “FLASH Transport Layer” , “OTA Tmp” and “APP Defined Section”. Note that the defined starting address of the FLASH memory accessible by MCU is 0x800000. Flash layout could be adjusted by MP tool.

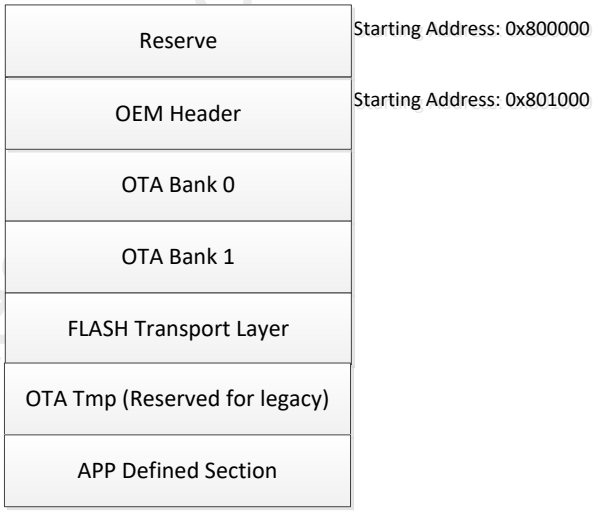


Figure 5-1 Flash Layout

The description of each field in the Flash layout is summarized in table below.

Table 5-1 Flash Section

Memory Segment	Starting Address	Size (Bytes)	Functions

Reserved	0x800000	0x1000	Reserved
OEM Header	0x801000	0x1000	Store configure information which includes BT Address, AES Key and user defined Flash layout
OTA Bank 0	Variable (defined in OEM Header)	Variable length (defined in OEM Header)	Store data and executable code. It can be divided into several sections: OTA Header, Secure boot, Patch, APP, APP Data1, APP Data2, APP Data3, APP Data4, APP Data5, APP Data6. If bank switch of OTA update is not supported, OTA_TMP is used for backup. If bank switch of OTA update is supported, one of bank0 and bank1 is executable zone, while the other is back-up zone.
OTA Bank 1	Variable (defined in OEM Header)	Variable length (defined in OEM Header)	The same as bank0, and the size of bank1 must also be the same with bank0
FLASH Transport layer	Variable (defined in OEM Header)	Variable length (defined in OEM Header)	Support accessing flash with logic address. User can read/write flash with unit size of 4 bytes at least
OTA_TMP	Variable (defined in OEM Header)	Variable length (defined in OEM Header)	Used as backups for OTA when bank switch of OTA update is not supported. Its size can't be less than the largest image of OTA bank0.
APP Defined Section	Variable (defined in OEM Header)	Variable length (defined in OEM Header)	The remaining zone of the Flash. User can use it freely except for OTA update.

There are 10 types of images in OTA Bank: OTA Header, Secure boot, Patch, APP, APP Data1, APP Data2, APP Data3, APP Data4, APP Data5 and APP Data6 shown in figure 5-2. The layout of OTA Bank is determined by OTA Header, which is generated by MP Pack Tool.

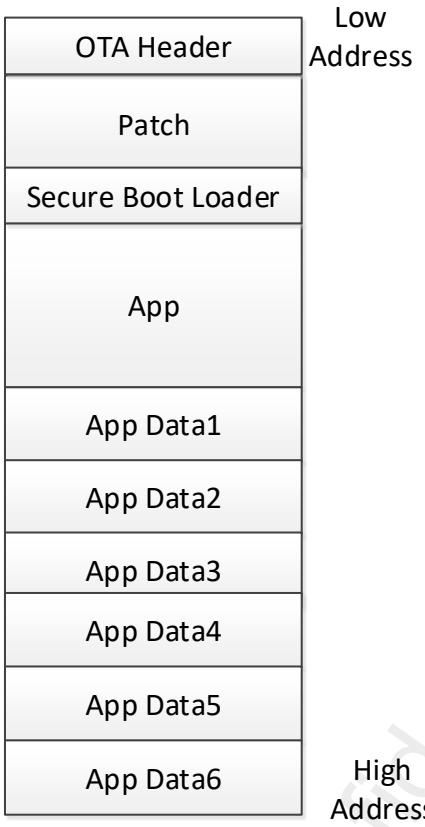


Figure 5-2 POTA Bank Layout

Table 5-2 Image Description of OTA Bank

Memory Segment	Starting Address	Size	Functions
OTA Header	Depend by OEM Header	4KB	OTA version, start address and size of each bank
Secure Boot Loader	Depend by OEM Header	changeable	Security check of code in boot process
Patch	Depend by OEM Header	changeable	Extended function of BT protocol stack and system in ROM
App	Depend by OEM Header	changeable	User application code
App Data1	Depend by OEM Header	changeable	APP Data need to be updated by OTA
App Data2	Depend by OEM	changeable	APP Data need to be updated by OTA

	Header		
App Data3	Depend by OEM Header	changeable	APP Data need to be updated by OTA
App Data4	Depend by OEM Header	changeable	APP Data need to be updated by OTA
App Data5	Depend by OEM Header	changeable	APP Data need to be updated by OTA
App Data6	Depend by OEM Header	changeable	APP Data need to be updated by OTA

RTL8762D supports flexible configuration of flash layout according to different application scenarios. User can customize flash layout through “config set” option of MP tool. Realtek offers FlashMapGenerateTool to generate files of flash_map.ini and flash_map.h. Flash_map.ini can be imported to MPTool and MPPack Tool to generate config file and OTA Header. Flash_map.h can be copied to APP project directory to build the target APP image.

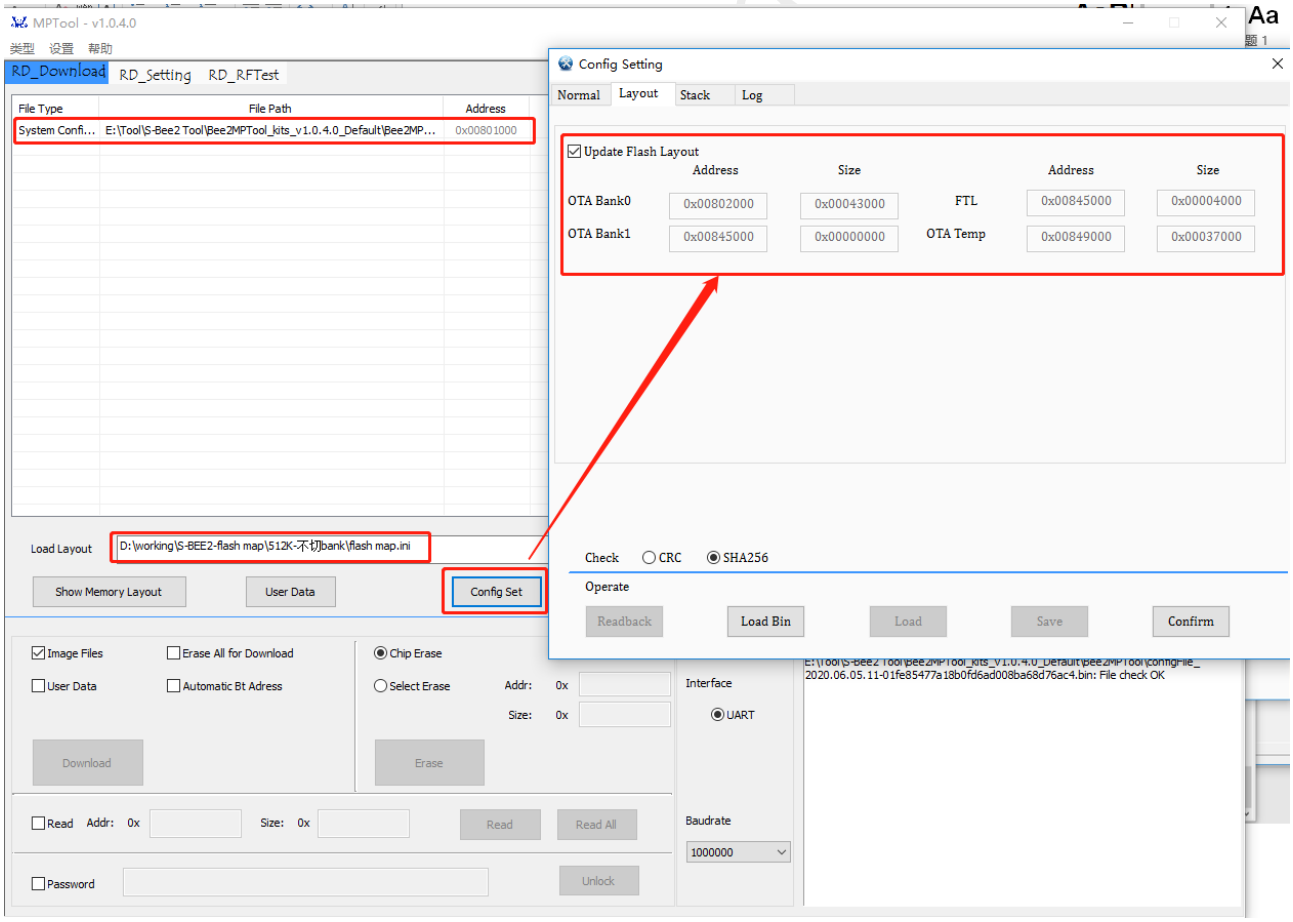


Figure 5-3 Generate config file by MPTool

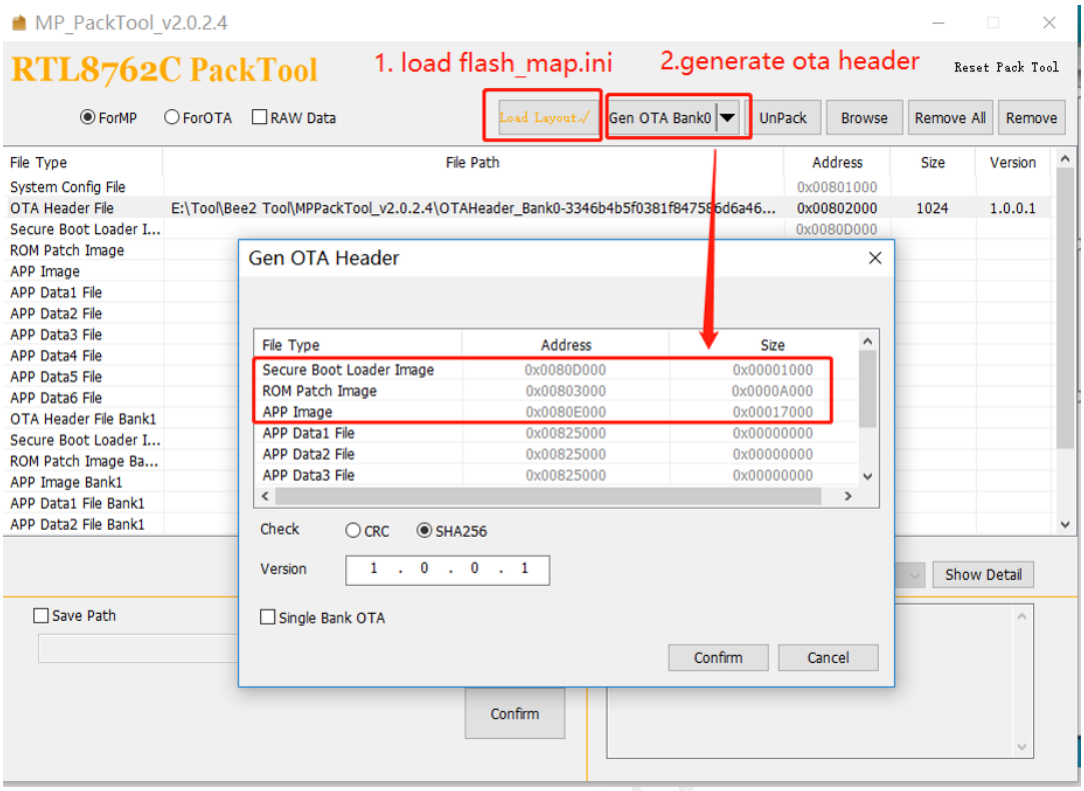


Figure 5-4 Generate OTA header file by MP PackTool

But you must pay attention to some of the principles of adjusting flash layout as described below.

1. If OTA supports bank switch, size of OTA bank0 and bank1 must be equal, while size of OTA tmp should be set to 0.
2. If OTA supports bank switch, layout of OTA bank0 and bank1 can be different. But the size of bank0 and bank1 cannot be changed and less than the total size of internal images.
3. If OTA doesn't support bank switch, size of OTA bank1 is set to 0. The size of OTA tmp can't be less than size of the largest image in OTA bank0.
4. **If OTA doesn't support bank switch, besides OTA Header, patch and APP, user should burn secure boot image. The address should be fixed to 0x80D0000, and size must be less than 4KB. The secure boot image is released by Realtek.**
5. After handled by tool, the original APP data bin file will be added a 1KB image header which meets some specific format. Note that, sections of APP Data1, APP Data2, APP Data3, APP Data4, APP Data5 and APP Data6 are used as storage of app data for OTA. If the data don't need to update, it should be placed in the APP Defined Section.
6. Make sure the offset between end address of OTA bank1 image and flash start address (0x800000) aligned to flash protected address range, such as 64KB, 128KB, 256KB and 512KB. This can lock all the code zone to protect from unexpected flash write and erase operation.
7. Because the data stored in the APP Data1, APP Data2, APP Data3, APP Data4, APP Data5 and APP Data6

areas in the OTA bank may fall within the flash lock range, the data stored in this area is generally read-only. If you need to store the data that needs to be overwritten, you should put it in the "app defined section" area.

As to the 6th principle, RTL8762D supports a mechanism named flash software Block Protect to lock flash to prevent writing and erasing operations. Flash software block protect uses some BP bits in flash status register to select the level (range) to protect as described below.

STATUS REGISTER(1)					W25Q16DV (16M-BIT) MEMORY PROTECTION(3)			
SEC	TB	BP2	BP1	BP0	PROTECTED BLOCK(S)	PROTECTED ADDRESSES	PROTECTED DENSITY	PROTECTED PORTION(2)
X	X	0	0	0	NONE	NONE	NONE	NONE
0	0	0	0	1	31	1F0000h – 1FFFFFh	64KB	Upper 1/32
0	0	0	1	0	30 and 31	1E0000h – 1FFFFFh	128KB	Upper 1/16
0	0	0	1	1	28 thru 31	1C0000h – 1FFFFFh	256KB	Upper 1/8
0	0	1	0	0	24 thru 31	180000h – 1FFFFFh	512KB	Upper 1/4
0	0	1	0	1	16 thru 31	100000h – 1FFFFFh	1MB	Upper 1/2
0	1	0	0	1	0	000000h – 00FFFFh	64KB	Lower 1/32
0	1	0	1	0	0 and 1	000000h – 01FFFFh	128KB	Lower 1/16
0	1	0	1	1	0 thru 3	000000h – 03FFFFh	256KB	Lower 1/8
0	1	1	0	0	0 thru 7	000000h – 07FFFFh	512KB	Lower 1/4
0	1	1	0	1	0 thru 15	000000h – 0FFFFFh	1MB	Lower 1/2
X	X	1	1	X	0 thru 31	000000h – 1FFFFFh	2MB	ALL

Figure 5-5 Flash Software Block Protect

Flash uses BP(x) bits in status register to identify number of blocks to lock, and TB bit to decide the direction to lock. However, Realtek only supports lock flash from low address to protect some important data such as configuration, security, and code sections. In order to support this feature, it is necessary to pass some checking rules to guarantee selected flash meets our requirement for software Block Protect. That is why Approved Vendor List (AVL) exists. Most flash in Qualified Vendor List supports protecting flash by level for different size, such as 64KB, 128KB, 256KB, 512KB, etc.

Therefore, when we divide flash layout, we need to ensure that the end address offset of OTA bank1 segment can be aligned to a certain level of protection that the selected flash supports as far as possible. Then RTL8762D will parse flash layout configuration parameters and query selected flash information to set block protect value. In order to maximize the use of BP, some flash layout examples are as follows. But what you have to notice is that there are two kinds of flash in AVL, which does not support protecting flash by block level. More details please

refer to Qualified Vendor List.

The protected flash zone can't be written and erased. If necessary, user can unlock flash first, and then write or erase, finally lock the flash to the previous level. But this operation is not recommended, as the flash status register is accessed by way of NVRAM. There is 100K times limit, so frequently unlocking may make the flash unavailable.

Table 5-3 Sample Flash layout (total flash size is 512KB)

Sample Flash Layout (total size is 512KB)	Size	Start Address	Block Protect size
1) Reserved	4K	0x800000	The front 256KB space starting from flash low address (OTA Bank1 end address offset is 276KB. It is not aligned so just lock 256KB.)
2) OEM Header	4K	0x801000	
3) OTA Bank0	268K	0x802000	
a) OTA Header	4K	0x802000	
b) Secure boot loader	4K	0x80D000	
c) Patch code	40K	0x803000	
d) APP code	220K	0x80E000	
e) APP data1	0K	0x00845000	
f) APP data2	0K	0x00845000	
g) APP data3	0K	0x00845000	
h) APP data4	0K	0x00845000	
i) APP data5	0K	0x00845000	
j) APP data6	0K	0x00845000	
4) OTA Bank1	0K	0x00845000	
5) FTL	16K	0x00845000	Unlocked region
6) OTA Temp	220K	0x00849000	
7) APP Defined Section	0K	0x00880000	

Table 5-4 Sample Flash layout (total flash size is 1MB)

Sample Flash Layout (Total size is 1MB)	Size	Start Address	Block Protect Size
1) Reserved	4K	0x00800000	The front 512KB space starting from flash low address
2) OEM Header	4K	0x00801000	
3) OTA Bank0	400K	0x00802000	

a) OTA Header	4K	0x00802000	(OTA Bank1 end address offset is 808KB. It is not aligned so just lock 512KB)
b) Secure boot loader	4K	0x0080D000	
c) Patch code	40K	0x00803000	
d) APP code	352K	0x0080E000	
e) APP data1	0K	0x00866000	
f) APP data2	0K	0x00866000	
g) APP data3	0K	0x00866000	
h) APP data4	0K	0x00866000	
i) APP data5	0K	0x00866000	
j) APP data6	0K	0x00866000	
4) OTA Bank1 (size must be same as OTA Bank0)	400K	0x00866000	
a) OTA Header	4K	0x00866000	
b) Secure boot loader	0K	0x00871000	
c) Patch code	40K	0x00867000	
d) APP code	352K	0x00872000	
e) APP data1	0K	0x008CA000	
f) APP data2	0K	0x008CA000	
g) APP data3	0K	0x008CA000	
h) APP data4	0K	0x008CA000	
i) APP data5	0K	0x008CA000	
j) APP data6	0K	0x008CA000	
5) FTL	16K	0x008CA000	Unlocked region
6) OTA Temp	0K	0x008CE000	
7) APP Defined Section	200K	0x008CE000	

Table 5-5 Sample Flash layout (total flash size is 2MB)

Sample flash layout (total size is 2MB)	Size	Start Addr	Block Protect Size
1) Reserved	4K	0x00800000	The front 1MB space starting from flash low address (OTA Bank1 end address offset is
2) OEM Header	4K	0x00801000	
3) OTA Bank0	508K	0x00802000	
a) OTA Header	4K	0x00802000	
b) Secure boot loader	0K	0x0080D000	

c) Patch code	40K	0x00803000	1MB.)
d) APP code	464K	0x0080E000	
e) APP data1	0K	0x00881000	
f) APP data2	0K	0x00881000	
g) APP data3	0K	0x00881000	
h) APP data4	0K	0x00881000	
i) APP data5	0K	0x00881000	
j) APP data6	0K	0x00881000	
4) OTA Bank1(size must be same as OTA Bank0)	508K	0x00881000	
a) OTA Header	4K	0x00881000	
b) Secure boot loader	0K	0x0088C000	
c) Patch code	40K	0x00882000	
d) APP code	464K	0x0088D000	
e) APP data1	0K	0x00900000	
f) APP data2	0K	0x00900000	
g) APP data3	0K	0x00900000	
h) APP data4	0K	0x00900000	
i) APP data5	0K	0x00900000	
j) APP data6	0K	0x00900000	
5) FTL	16K	0x00900000	
6) OTA Temp	0K	0x00904000	Unlocked region
7) APP Defined Section	200K	0x00904000	

5.2 Flash APIs

Flash operation APIs are listed as follows, refer to SBee2-SDK.chm and RTL8762D Flash User Guide for more details. The APIs with “auto” will access flash with auto mode, the others will access flash with user mode.

1, Basic Operation APIs:

bool flash_auto_read_locked(uint32_t addr, uint32_t *data);

bool flash_read_locked(uint32_t start_addr, uint32_t data_len, uint8_t *data);

bool flash_auto_write_locked(uint32_t start_addr, uint32_t data);

bool flash_auto_write_buffer_locked(uint32_t start_addr, uint32_t *data, uint32_t len);

bool flash_write_locked(uint32_t start_addr, uint32_t data_len, uint8_t *data);

bool flash_erase_locked(T_ERASE_TYPE type, uint32_t addr);

2, Flash High Speed Read APIs:

bool flash_auto_dma_read_locked(T_FLASH_DMA_TYPE dma_type, FlashCB flash_cb,

uint32_t src_addr, uint32_t dst_addr, uint32_t data_len);

bool flash_auto_seq_trans_dma_read_locked(T_FLASH_DMA_TYPE dma_type, FlashCB flash_cb,

uint32_t src_addr, uint32_t dst_addr, uint32_t data_len);

bool flash_split_read_locked(uint32_t start_addr, uint32_t data_len, uint8_t *data, uint32_t *counter);

If the three above APIs are used, user needs to copy file sdk\src\flash\flash_hs_read.c into the project and include header file flash_device.h.

5.3 FTL

FTL (flash transport layer) is used as abstraction layer for bt stack and user application to read/write data in flash. Through FTL interface, user can read or write the responding data in flash space for FTL by logic address.
Generally, FTL space is more suitable for storing data that needs to be rewritten frequently. On the one hand, it is simpler to use FTL interface. On the other hand, FTL operation will actually recycle the allocated physical space so as not to write multiple times to a flash sector and affect the life of flash. If you need to store a large amount of data and only need to overwrite occasionally, or store read-only data but do not need to upgrade, you can call flash APIs in the "app defined section" area for management. If you need to store read-only data and upgrade it, place it in App Data1, App Data2, App Data3, App Data4, App Data5 and App Data6 areas in OTA bank.

5.3.1 FTL Storage Space

The FTL space can be divided into 2 spaces according to functions. Take the default physical space size of FTL as 16K:

1. **BT storage space**

   (1) Logic address range: [0x0000, 0x0C00). But this space size can be changed by otp parameter.

(2) This region is used to store BT information such as device address, link key, etc.

(3) Refer to RTL8762D BLE Stack User Manual for more details.

2. **APP storage space**

(1) Logic address range: [0x0C00, 0x17f0)

(2) APP can use this region to store user defined information.

(3) The following APIs can be called to read/write data in this region, and they are defined in ftl.h. Please refer to SBee2-SDK.chm for more details.

```
uint32_t ftl_save(void * p_data, uint16_t offset, uint16_t size)

uint32_t ftl_load(void * p_data, uint16_t offset, uint16_t size)
```

5.3.2 Adujust FTL Space Size

The physical space size of FTL is configurable, which can be adjusted by modifying the configuration parameters of the config file. The operation steps are as follows:

(1) First, use FlashMapGenerateTool which released with MPackTool to generate “flash map.ini” and “flash_map.h” file.

(2) Copy the “flash_map.h” file to the app project directory, for example “\sdk\board\evb\ota”, so that the correct load address can be obtained when the app is compiled.

(3) Load “flash map.ini” into MPTool to generate config file for download. As shown in Figure 5-7, the physical space of FTL will be adjusted to 32K.

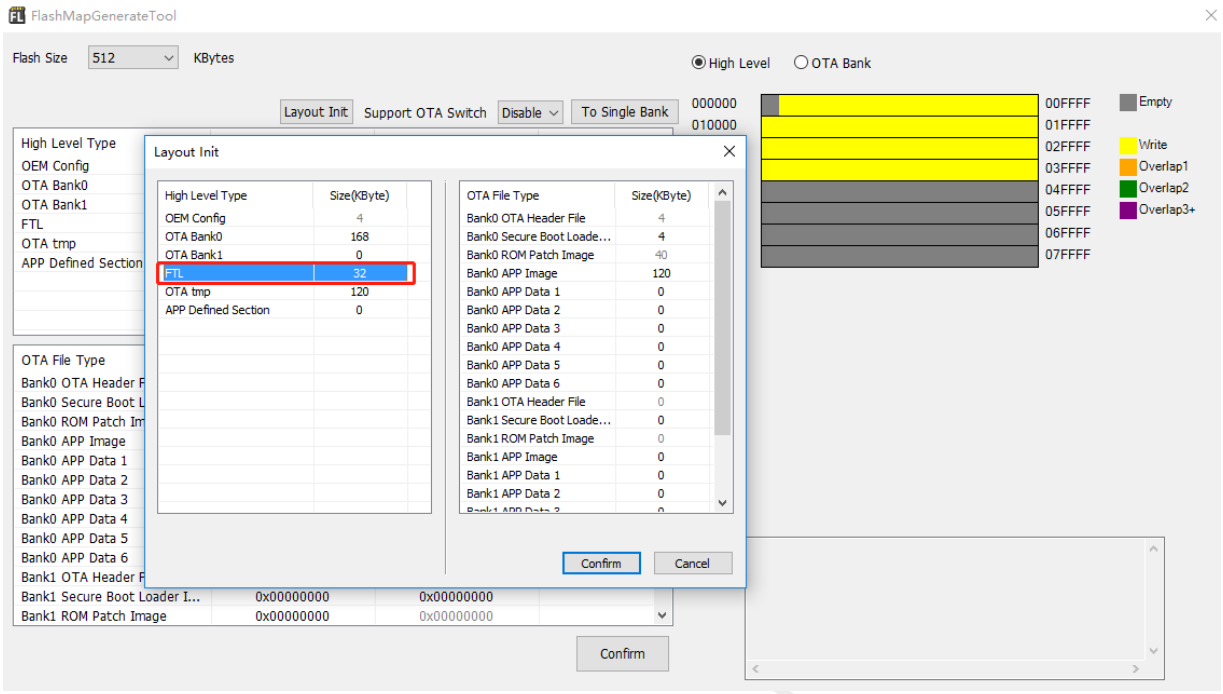


Figure 5-6 Adjust Flash Map Config File

When adjusting the size of FTL physical space, the logical space available to app will also change accordingly. Assuming that the physical space size of the actual FTL is set to MK (M is an integral multiple of 4), the logical space available for the corresponding app is equal to ((511 * (M-4) - 4) - 3072) bytes. In addition, in order to improve the efficiency of FTL reading, a set of mapping mechanism of physical address and logical address is designed in the bottom layer. This mapping table will occupy a certain amount of RAM space. When the default FTL physical space size is 16K, the mapping table takes up 2298 bytes of buffer RAM heap space. Assuming that the physical space size of the actual FTL is set to MK (M is an integral multiple of 4), the RAM space occupied by its mapping table is equal to ((511 * (M-4) - 4) * 0.375) bytes. Therefore, users need to adjust the size of FTL space reasonably according to specific application scenarios. If the size is too large, some RAM resources will be wasted. On the other hand, if you choose a smaller flash and want to compress the space occupied by FTL, you must ensure that the physical space of FTL is not smaller than 12K. At the same time, due to the limitation that each logical address in the mapping table only occupies 12 bits, the maximum physical space of FTL can be adjusted to 36K

6 Flash Code and RAM Code Setting

The code can run on Flash or on RAM. This section describes how to place code in a specific memory to execute.

1. Modify the macro FEATURE_RAM_CODE definition:
   (1) 1 indicates that the code without any section modified runs on RAM.
   (2) 0 indicates that the code without any section modified runs on Flash.

2. If you want to specify a function to place on a specific memory, use the section macro in app_section.h. For example:
   (1) APP_FLASH_TEXT_SECTION means putting the function into Flash to execute.
   (2) DATA_RAM_FUNCTION means putting the function into RAM to execute.
   (3) SHARE_CACHE_RAM_SECTION means putting functions, global and static variables on shared cache RAM

7 eFuse

eFuse is a block of one-time programming memory which is used to store the important and fixed information, such as UUID, security key and other one-time programming configuration. The single bit of eFuse cannot be changed from 0 to 1, and there is no erase operation to eFuse, so be careful to update eFuse. Realtek offers MP Tool to update certain eFuse sections.