Telink 802.154

SDK User Guide

AN_19052902-E2

Ver1.1.0

2021.6.9

Keyword

802.154

Brief

This document is the development guide for Telink 802.154 SDK.

Published by
Telink Semiconductor

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

Legal Disclaimer

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

Information

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

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

telinksales@telink-semi.com

telinksupport@telink-semi.com

Revision History

Version	Change Description
V1.0.0	Initial release.
V1.1.0	Refactor user guide

Contents

1 Overview

Based on IEEE 802.15.4(2006), Telink 802.15.4 SDK supports standard PHY(physical) layter and MAC(media access control) layter. In addition, the SDK realized part of application player function as a reference.

The SDK system structure is as follows：

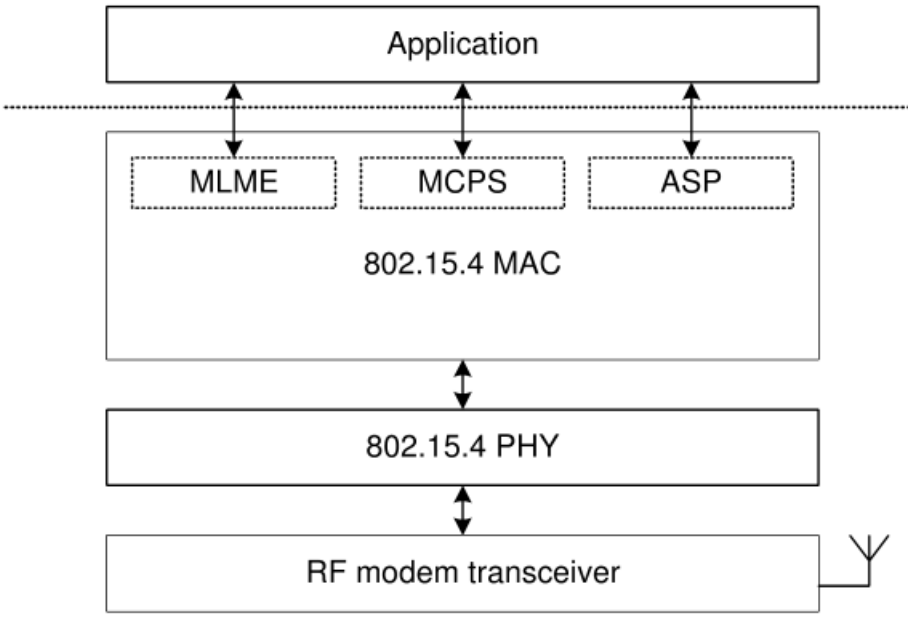

Figure 1: "SDK System Architecture"

- PHY (physical) layer, on the basis of the Telink hardware platform and driver, achieves a standard 2.4G wireless connection, such as modulation, supported frequency, transmit power, and so on, which can be referred to the corresponding chip datasheet.
- MAC (media access control) layer is implemented by software and is based on the IEEE 802.15.4 specification, which implements standard data formats, encryption mechanisms, and so on.
- APP (application) layer, the SDK provides some application examples, such as entering PAN, OTA and so on.

In order to interact with the 15.4 stack, the SDK provides three interfaces to the APP layer：MLME, MCPS, and ASP.

a) MAC sub layer includes MAC Layer Management Entity (MLME)：the application layer can send 802.15.4 MAC command through MLME. For example, the application layer can send MLME-ASSOCIATE.request primitive through this interface and also receive MLME- ASSOCIATE.confirm primitive.

b) MCPS interface：the application layer can send and receive 802.15.4 MAC data through MCPS.

c) ASP interface：the application layer can set the MAC layer PIB parameters through the ASP.

The use of the above interface will be described in detail in subsequent chapters.

1.1 Device type

From the perspective of the included functions, 15.4 contains the following two device types.

- **Full Function Device (FFD)：** Full functional devices with 15.4 can be either coordinator (PAN coordinator OR coordinator), routing device (router), or end node with full functionality (end device), full-function devices (FFD) generally do not go into hibernation.
- **Reduced Function Device (RFD)：** It can only be used as an end node, and generally has low power consumption requirements.

1.2 Network type

The 802.15.4 network is a central network, also called a star network.

- **Central network**: A network usually formed and managed by a PAN Coordinator.

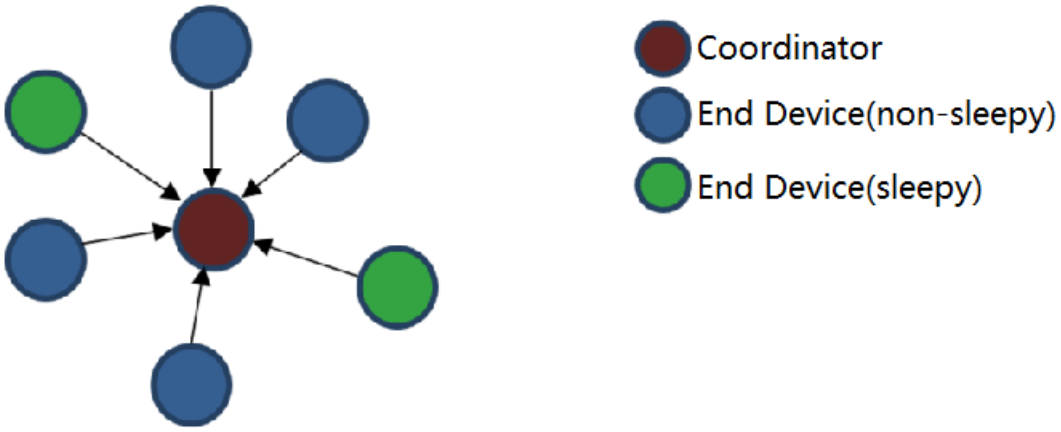

Figure 2: "Central network"

1.3 Basic concept

1) **PAN ID:** Personal Area Network ID

The PAN ID is to identify the formed network. The PAN ID is assigned by the node responsible for forming the network, and can be assigned directly or generated randomly, but active scan is required to avoid PAN ID conflicts.

2) **Channel:**

The operating frequency points permitted by 15.4 to work at (2.4G)：2405 + (N-11)*5 (mHz), (channel N = 11~26).

Once connected to a network, it uses a single frequency point of operation mode, will not actively jump frequency.

3) **Direct Data Transfer:**

Generally used to send data to FFD devices, for example, end device sends data to coordinator.

4) **Indirect Data Transfer:**

Generally used for FFD device to send data to RFD that enters low-power, for example, coordinator sends data to end device, coordinator will store data locally first, end device gets the packet by poll request.

14 Telink 802.154 SDK development brief

Telink 802.15.4 SDK development is realized on the basis of hardware and software provided by Telink.

14.1 Telink 802.154 SDK software development environment

1) **Essential software tools** (download at: http://wiki.telink-semi.cn/)

- Integrated development environment:

  TLSR8 Chips: Telink IDE for TC32

  TLSR9 Chips: Telink RDS IDE for RISC-V
- Download debugging tools: Telink Burning and Debugging Tools
- OTA code conversion tool: tl_ota_tool, the detailed procedure refers to Chapter 6.2.2 OTA.

2) **Auxiliary tool to capture and analyze packet** (download or purchase as you need)

- TI Packet Sniffer
- Ubiqua

3) **Software development kit** (download at: http://wiki.telink-semi.cn/)

- TLSR8_802_15_4_SDK.zip

14.2 Hardware platforms supported by Telink 802.154 SDK

- B85m (TC32 platform)

  826x: 8267 EVK Board and 8267 USB Dongle

  8269 EVK Board and 8269 USB Dongle

  8258: 8258 EVK Board and 8258 USB Dongle

  8278: 8278 EVK Board and 8278 USB Dongle
- B91m (RISC-V platform)

  9518: 9518 EVK Board and 9518 USB Dongle

2 Telink 802.154 SDK

Before installing the SDK, please install the appropriate Telink IDE for TC32 or Telink RDS IDE for RISC-V according to section 1.4.

2.1 TLSR8 TC32 SDK installation

2.1.1 Project import

1) Start the IDE and go to the interface File -> Import -> Existing Projects into Workspace in order.

2) Select tl_802154_sdk/build directory -> click "OK".

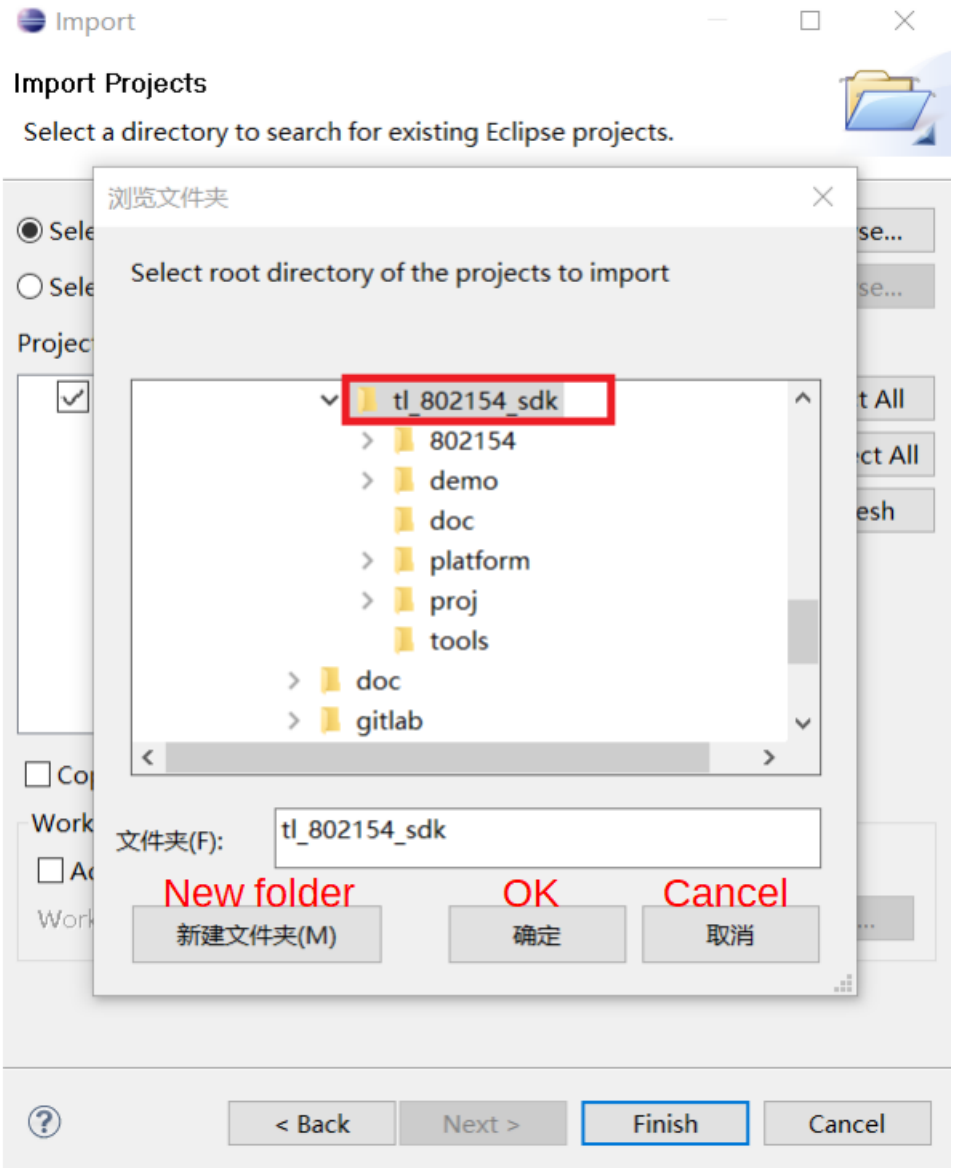

Figure 3: "Select import projects"

3) Click "Finish" to complete the project import.

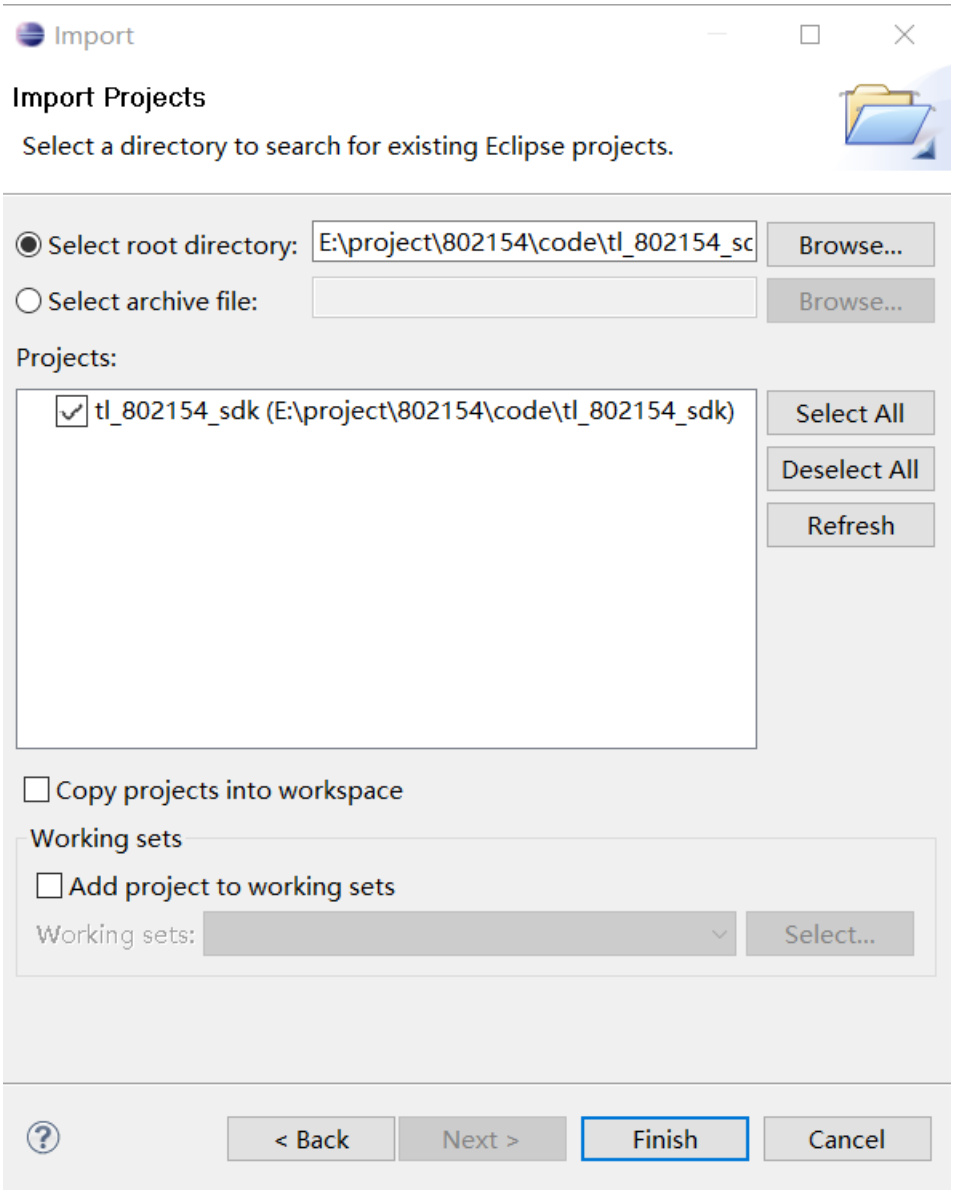

Figure 4: "Complete project import"

Copy project into workspace This box can be checked or unchecked according to the actual situation. If the project is already in this directory, you don't need to check it, but if it is in another directory and you want to edit and develop the project in the current directory, you can check it.

2.1.2 Project directory structure

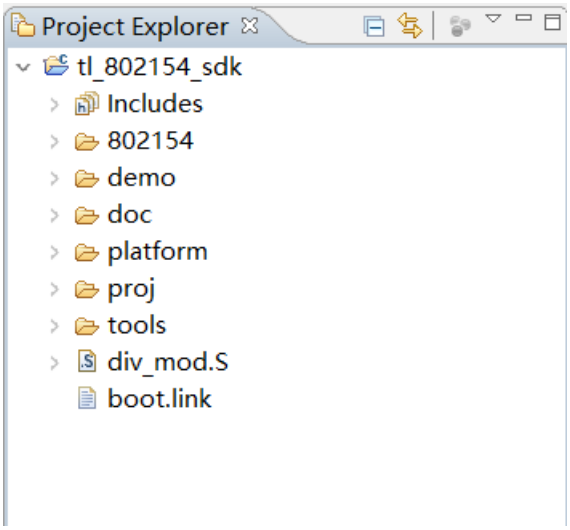

Figure 5: "Project directory structure"

- **/demo:** user project directory.
  - /app_common: application-level public code directory, including application-specific packet processing functions (tl_specific_data.c tl_specific_data.h), and application-specific OTAs in this processing function, which other applications can also improve on, or import their own application-layer code.
  - /associate_coor: PAN Coordinator samples.
  - /associate_dev: end device samples.
- **/platform:** operation platform directory.
  - /boot: boot and link files.
  - /chip_xx: chip driver files.
  - /services: interrupt service function files.
- **/proj**: project code directory.
  - /common: common code directory.
  - /drivers: abstraction layer driver file.
  - /os: task events, buffer management function files.
- **/802154:** protocol stack related directory.
- **/tools:** hci command processing related directory.
- **div_mod.S:** division and remainder related assembly functions. (TLSR8 does not support hardware dividers)

2.1.3 Compile options

Click the drop-down icon  , you can see all the current compiling options, select the sample project you need to compile as below, and wait for the compiling to complete.

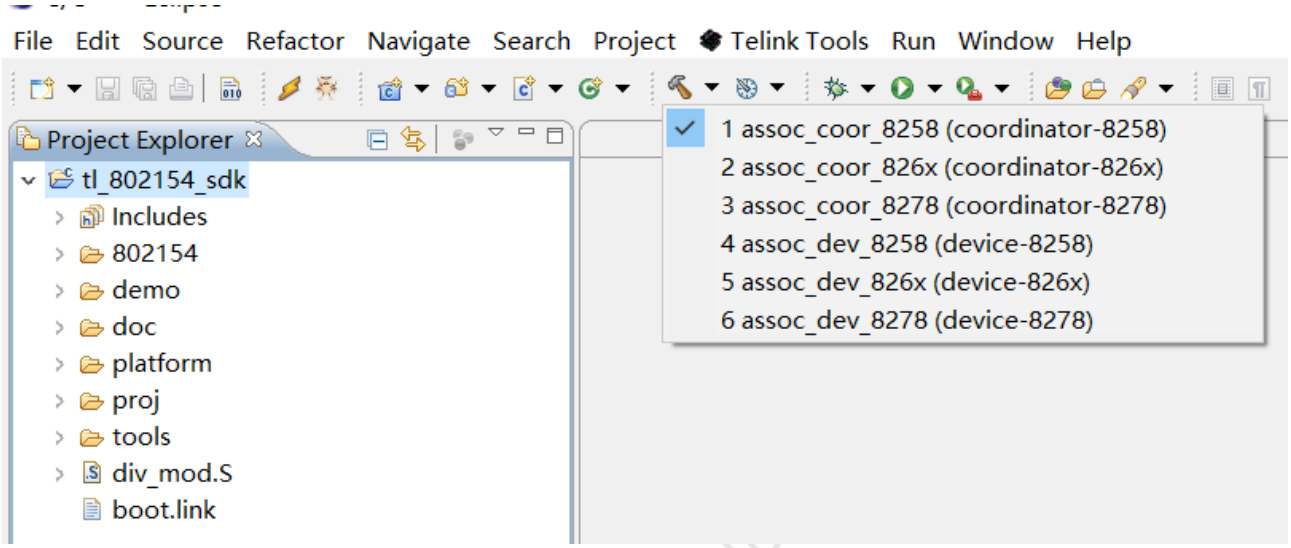

Figure 6: "Select and compile"

After the compiling is completed, the compiled folder will appear in the "Project Explorer" window, which contains the compiled firmware, as shown below.

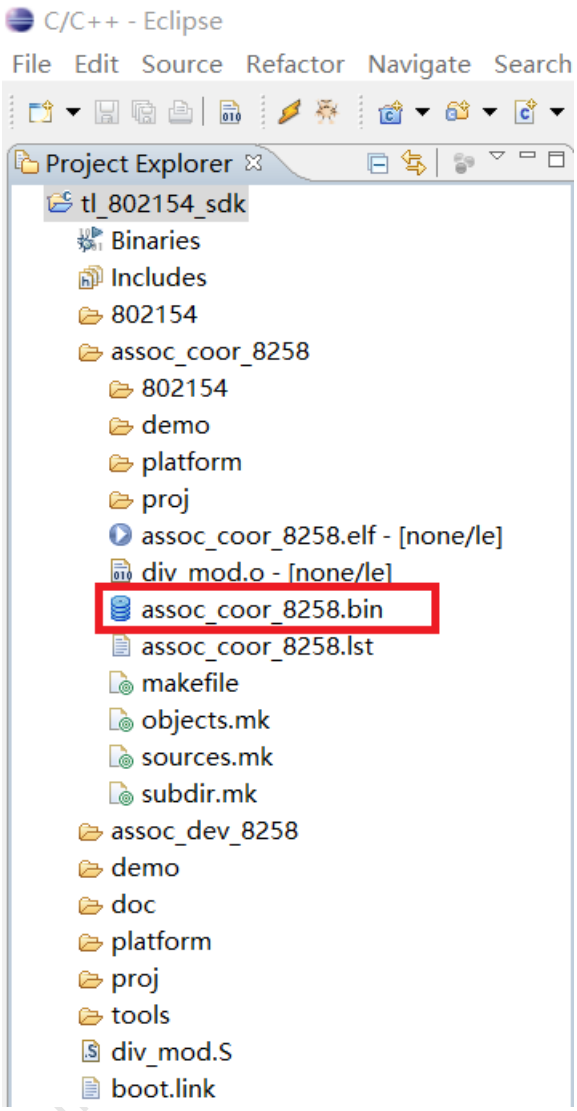

Figure 7: "assoc_coor_82xx output file"

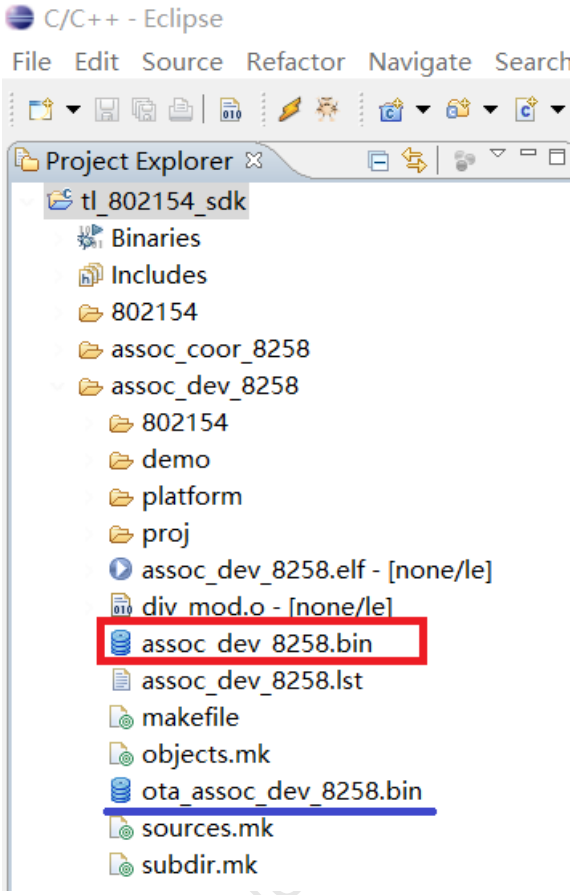

Figure 8: "assoc_dev_82xx output file"

The assoc_dev_82xx project will generate two bin files, the red boxed assoc_dev_82xx.bin is the running bin, which is used to burn directly to the target board; the blue underlined ota_assoc_dev_82xx.bin is the OTA file.

2.14 Add new project

In the provided SDK, only some simple use cases are listed. Users can add their own application projects and compiling options according to their needs.

The detailed steps are as follows.

1) Step 1: Project Explorer -> tl_802154_sdk -> Properties -> Manage Configurations

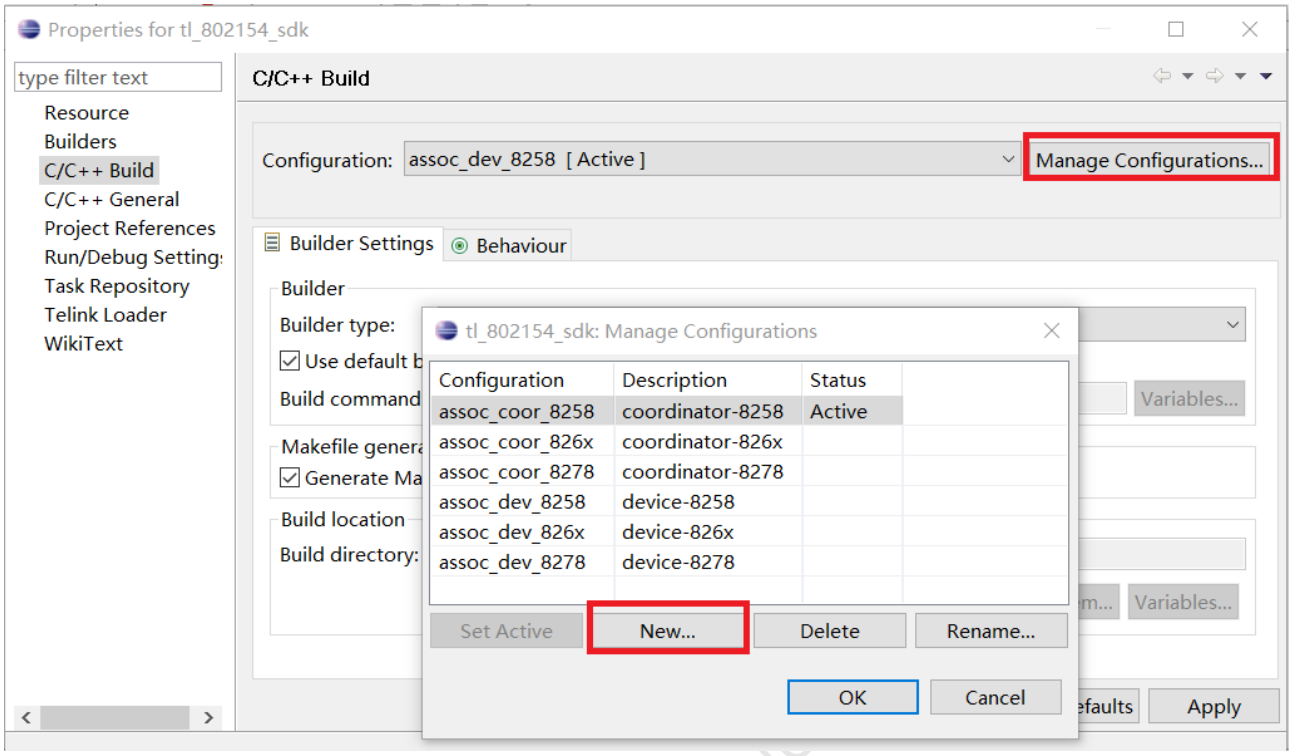

Figure 9: "Adding a new project"

2) Step 2: Click New, the following window will pop up.

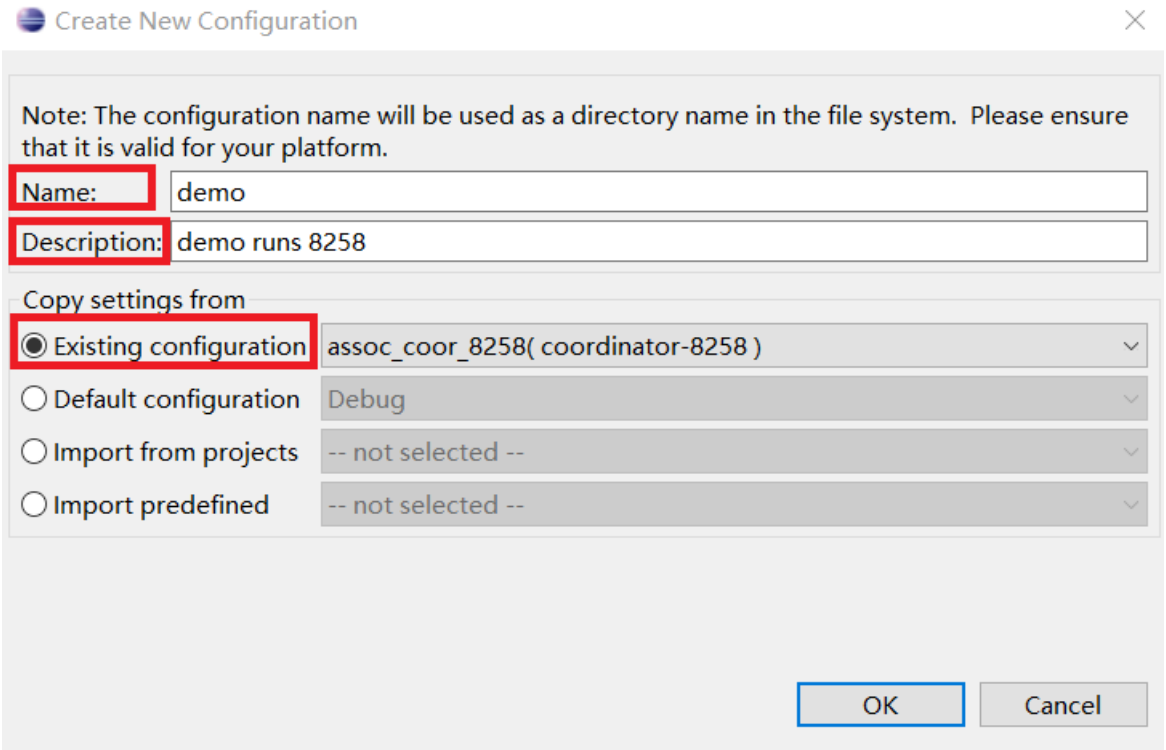

Figure 10: "Configuring a new project"

- Name: project name
- Description: brief project description
- Copy settings from: It is recommended to choose Existing configuration, which can simplify the configuration process.

The principle of selecting Existing configuration is as below.

- If using 8278 platform, select associate_xxx_8278; if using 8258 platform, select associate_xxx_8258.
- Project for coordinator development, select associate_coor_8258; project for end device development, select associate_dev_8258.

Assuming that a project is needed to develop a sensor with hibernation function using the 8258 chip, the configuration of the project "associate_dev_8258" should be selected as the configuration for this new project according to this principle.

If you need to modify the configuration, you can follow the next section (2.1.5 Project configuration description) and adjust accordingly.

2.1.5 Project configuration description

Enter in order: Project Explorer -> tl_802154_sdk -> Properties -> Settings (take the project associate_dev_8258 for example)

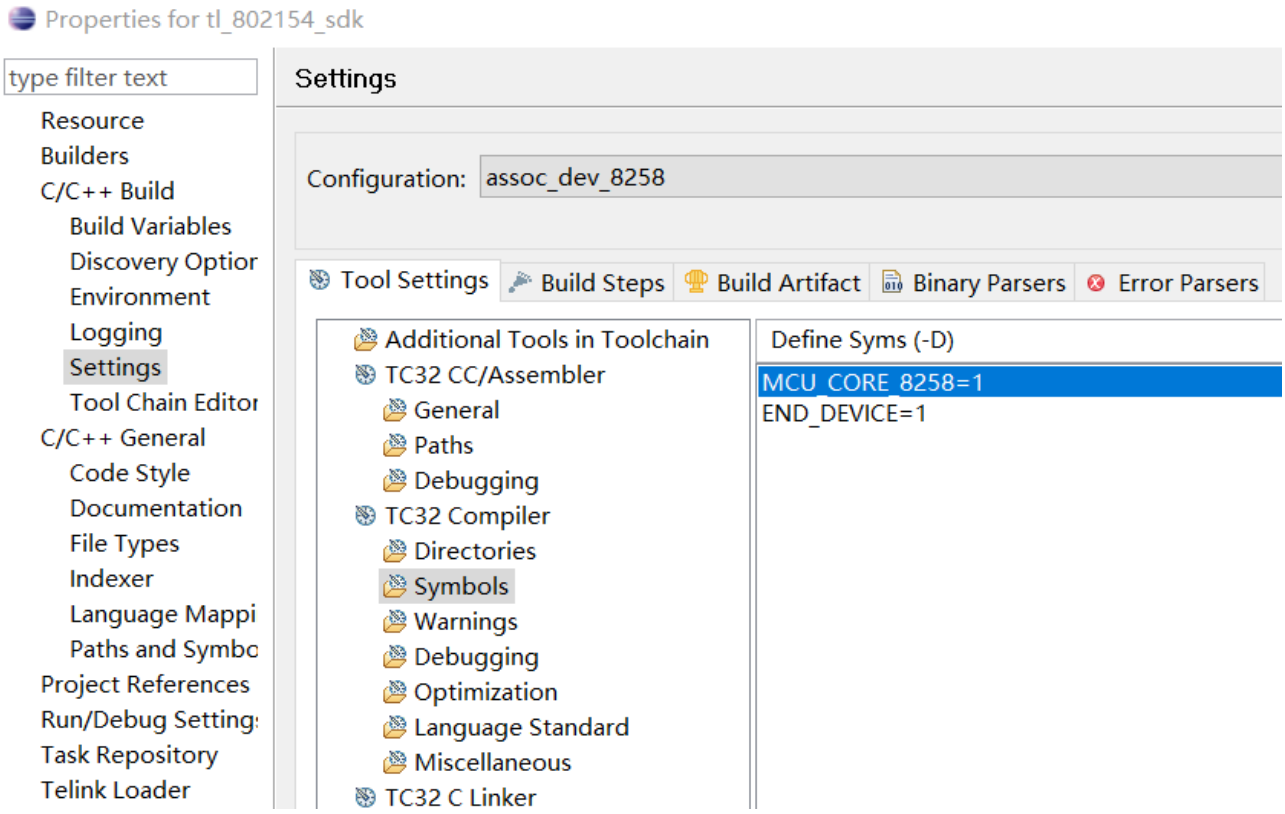

Figure 11: "Symbol definition"

The "Tool Settings" contains some pre-defined configurations for current project:

1) **Device type pre-definition**

END_DEVICE=1：indicates that the project is a device with end device function

Following shows the device setting for coordinator and end device:

- COORDINATOR=1：indicates that the project is a device with coordinator function
- END_DEVICE=1：indicates that the project is a device with end device function

2) **Platform selection**

- 8269 platform: -DMCU_CORE_826x=1 and -DCHIP_8269=1

  The startup code cstartup_826x.S is located in the tl_802154_sdk -> platform -> boot directory.

- 8258 platform: -DMCU_CORE_8258=1

  The startup code cstartup_8258.S is located in the tl_802154_sdk -> platform -> boot directory.

- 8278 platform: -DMCU_CORE_8278=1

  The startup code cstartup_8278.S is located in the tl_802154_sdk -> platform -> boot directory.

3) **Links for lib files**

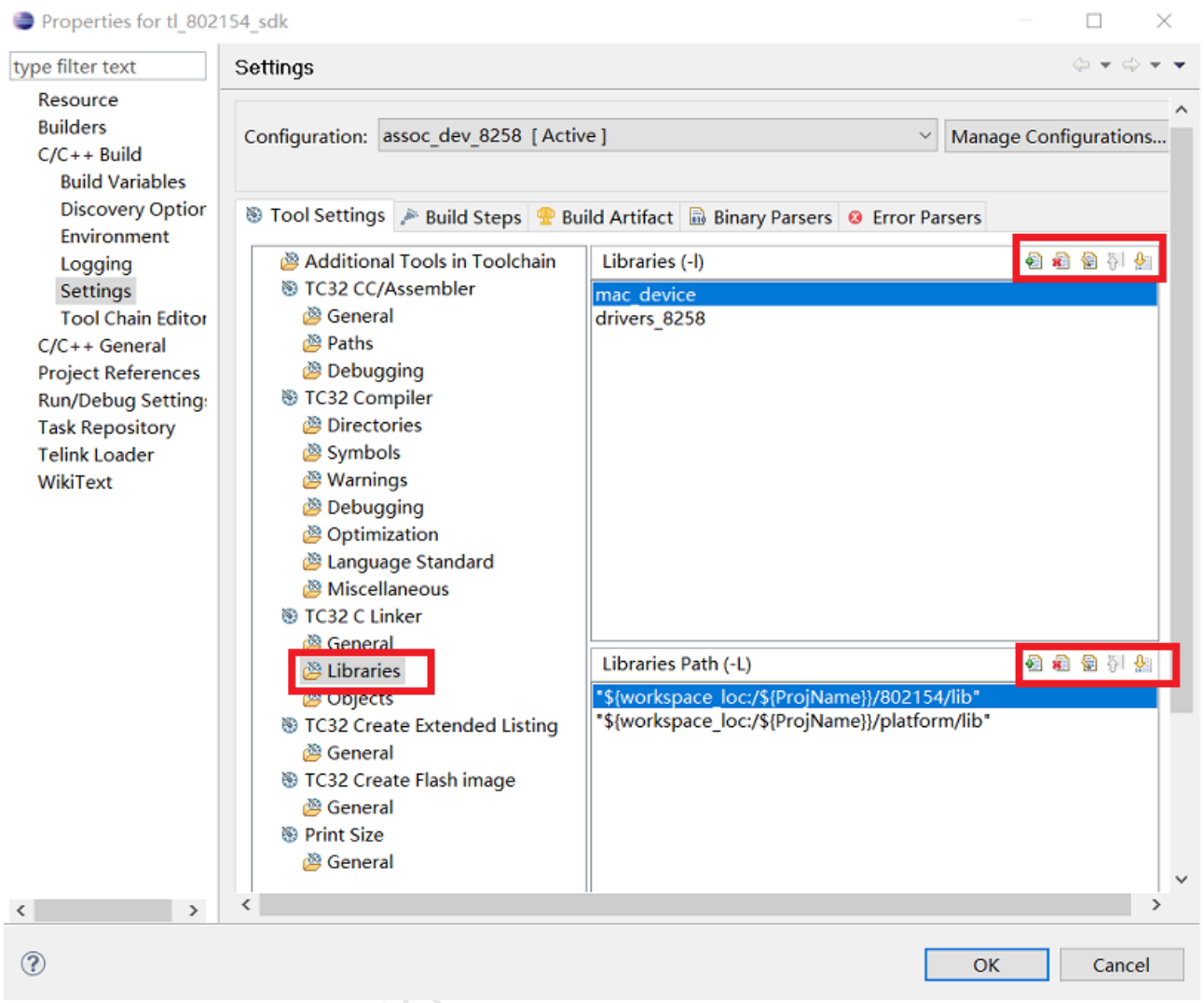

Figure 12: "Library files and paths"

The current SDK includes 2 types of library files: the 15.4 stack library and the platform driver library.

- 15.4 stack library: libmac_device.a，libmac_coor.a

  Available in the tl_802154_sdk -> 802154 -> lib directory.

- Platform driver library: libdrivers_826x.a，libdrivers_8258.a，libdrivers_8278.a

  Available in the tl_802154_sdk -> platform -> lib directory.

4) **Link file**

According to the actual application requirements, the user can adjust the appropriate link file as per the different platforms chosen and the memory requirements.

The current SDK provides the default link files boot_826x.link, boot_8258.link and boot_8278.link for the 826x, 8258 and 8278 platforms, which are available in the corresponding folder of tl_802154_sdk -> platform -> boot.

As shown in the figure below, the link file to be used can be selected by invoking the script of tl_link_load.sh (in the directory of tl_802154_sdk -> tools) during the pre-compiling.

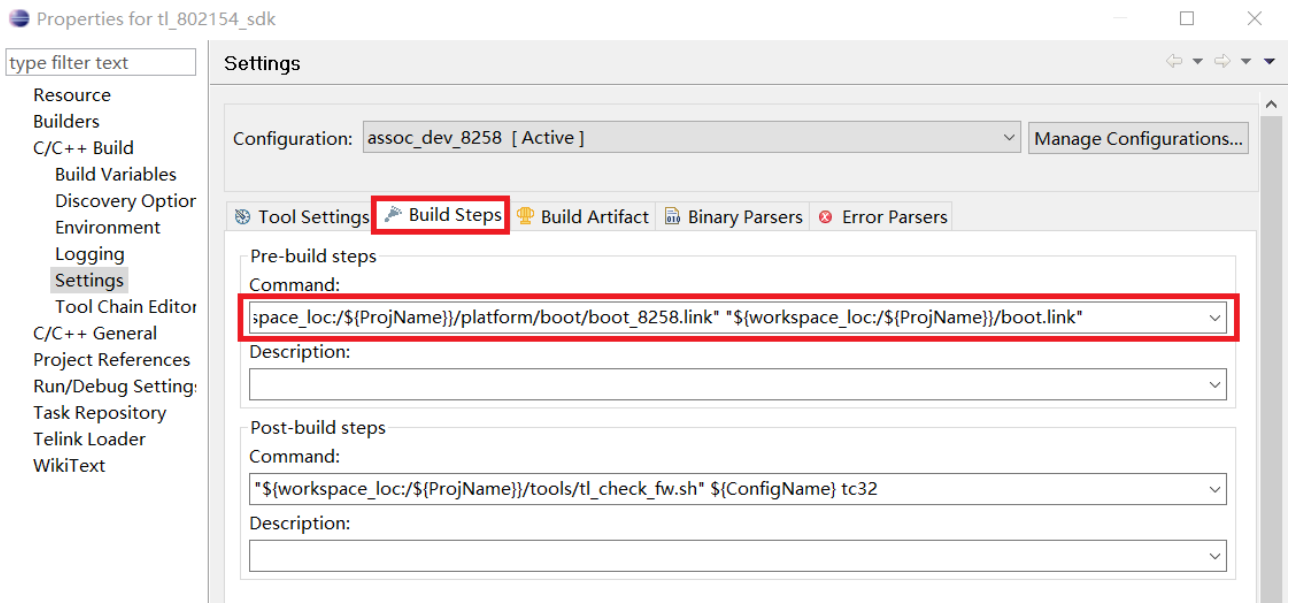

Figure 13: "Link file pre-compiling settings"

5) **.image file check**

In order to ensure the reliability of the downloaded file, a check field is added to the generated image file by the script tl_check_fw.sh (in the directory of tl_802154_sdk -> tools) and ota_bin_tool is executed by the non-coordinator device, and during the OTA process it will determine whether to update the image by checking whether the check field matches or not.

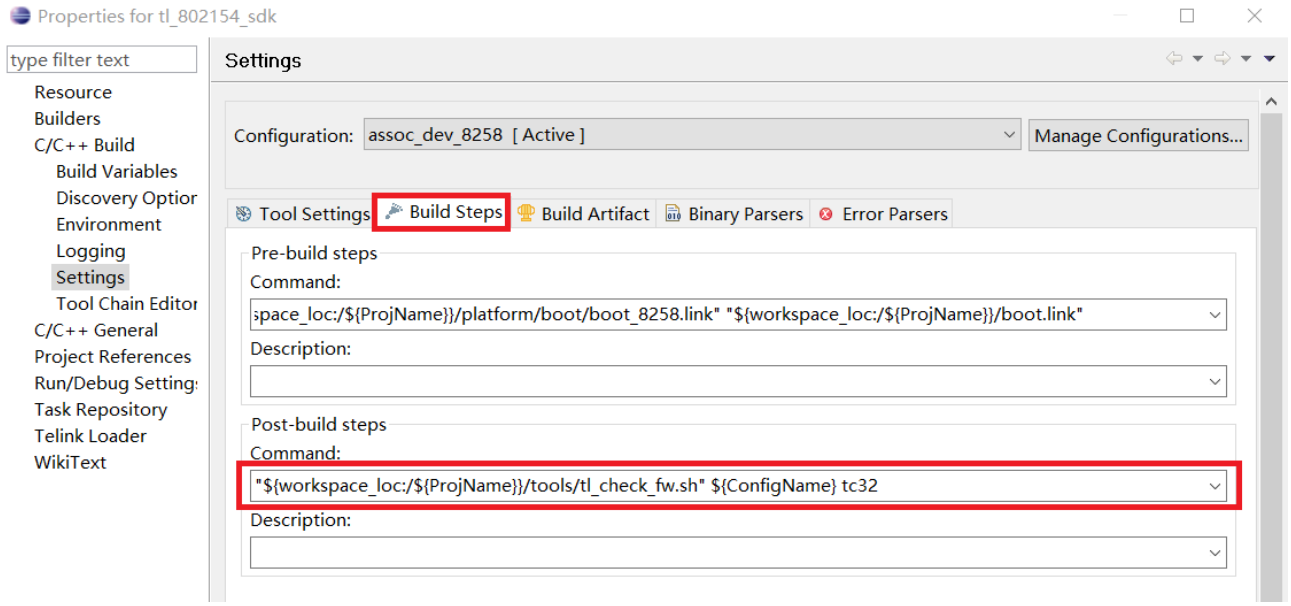

Figure 14: "image check settings"

2.2 TLSR9 RISC-V SDK installation

2.2.1 Project import

1) Start the IDE and go to the interface File -> Import -> Existing Projects into Workspace in order

2) Select tlsr_riscv in the tl_802154_sdk -> build directory, click "OK".

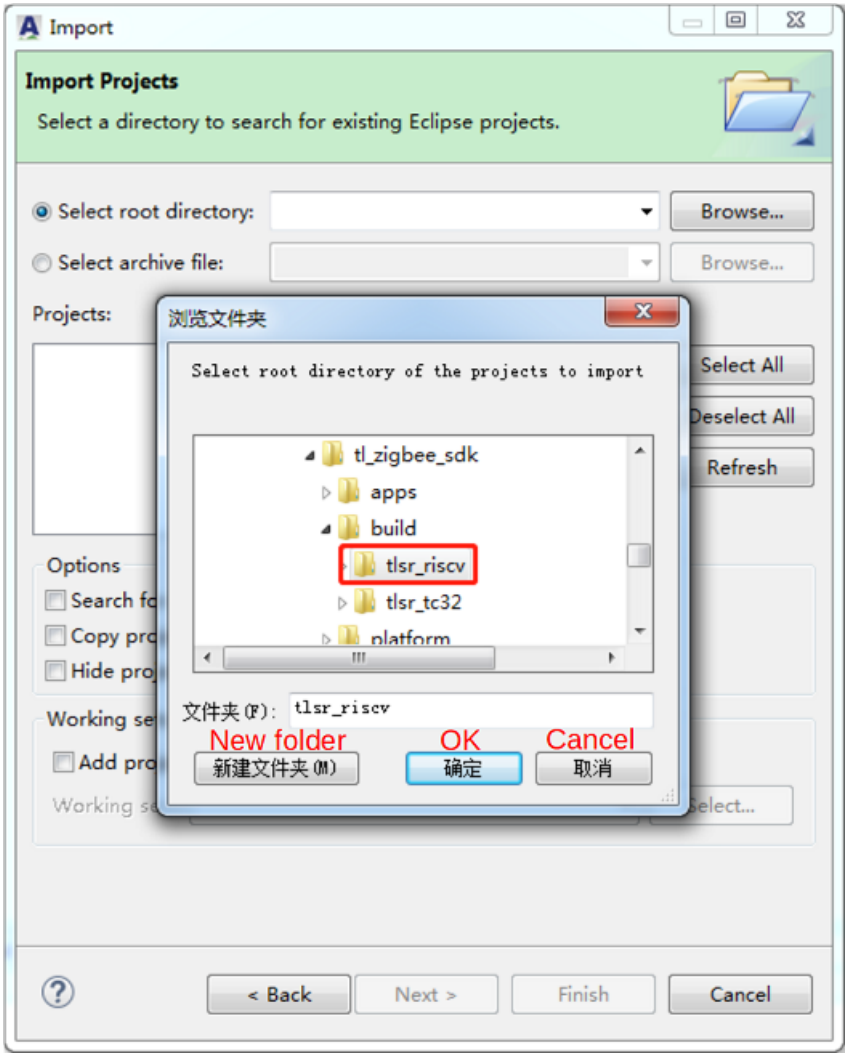

Figure 15: "Select import project file"

3) Click "Finish" to complete the project import.

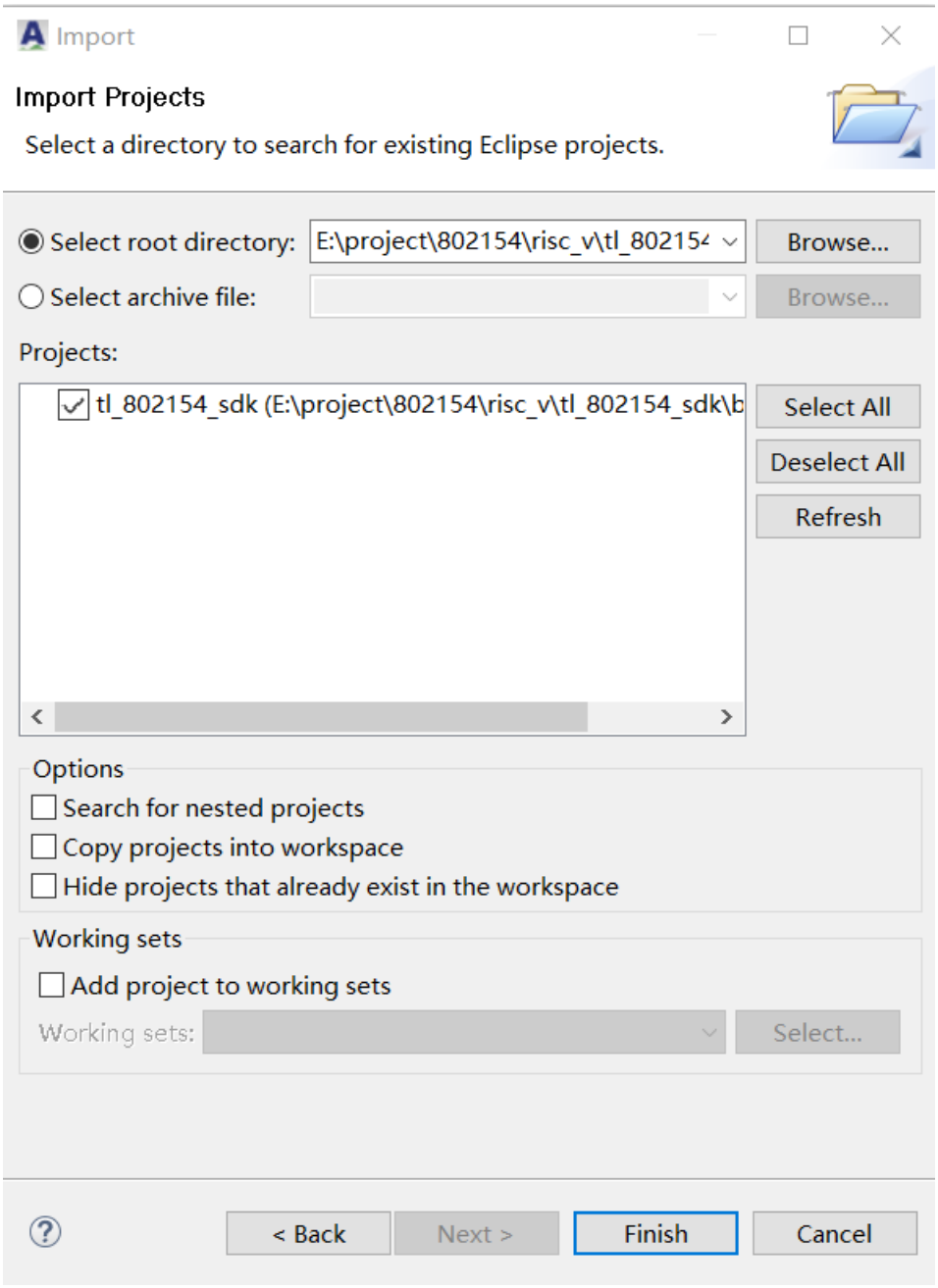

Figure 16: "Complete project import"

2.2.2 Project directory structure

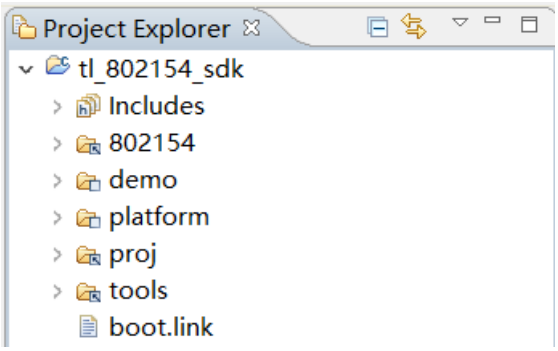

Figure 17: "Project directory structure"

- **/demo:** user project directory.
  - /app_common: application-level public code directory, including application-specific packet processing functions (tl_specific_data.c tl_specific_data.h), and application-specific OTAs in this processing function, which other applications can also improve on, or import their own application-layer code.
  - /associate_coor: PAN Coordinator samples.
  - /associate_dev: end device samples.
- **/platform**: operation platform directory.
  - /chip_xx: chip driver file.
  - /services: interrupt service function file.
- **/proj**: project code directory.
  - /common: common code directory.
  - /drivers: abstraction layer driver file.
  - /os: task events, buffer management function files.
- **/802154:** protocol stack related directory.

2.2.3 Compile options

Click the drop-down icon "  ", you can see all the current compiling options, select the sample project you need to compile as below, and wait for the compiling to complete.

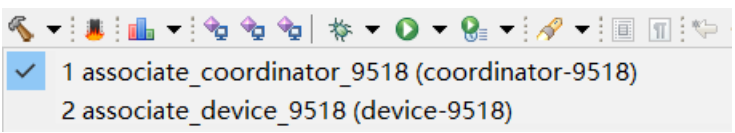

Figure 18: "Select and compile"

After the compiling is completed, the compiled folder will appear in the "Project Explorer" window, which contains the compiled firmware, as shown below.

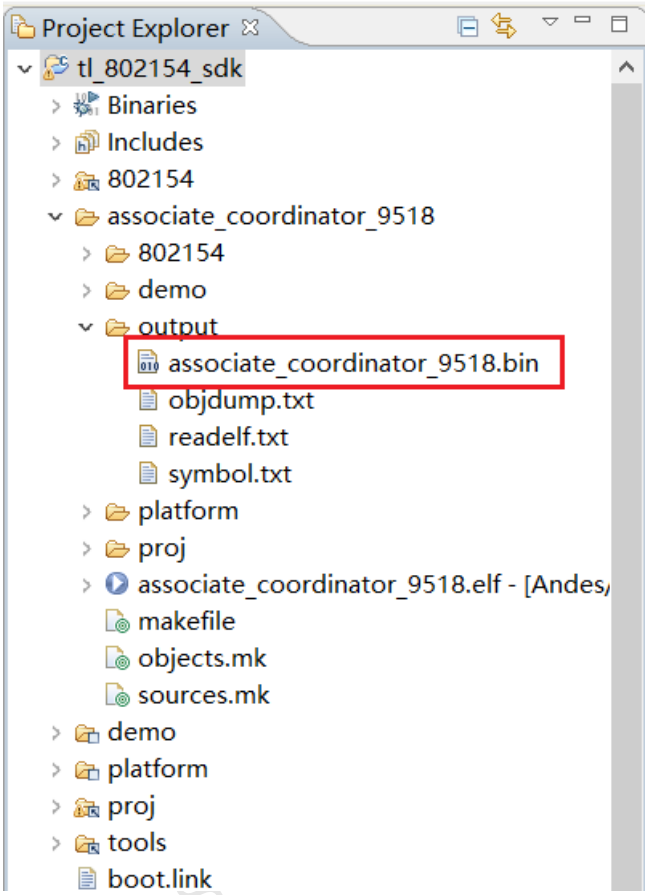

Figure 19: "Output file"

2.24 Add new project

In the provided SDK, only some simple use cases are listed. Users can add their own application projects and compiling options according to their needs.

The detailed steps are as follows.

1) Step 1: Project Explorer -> tl_802154_sdk -> Properties -> Manage Configurations

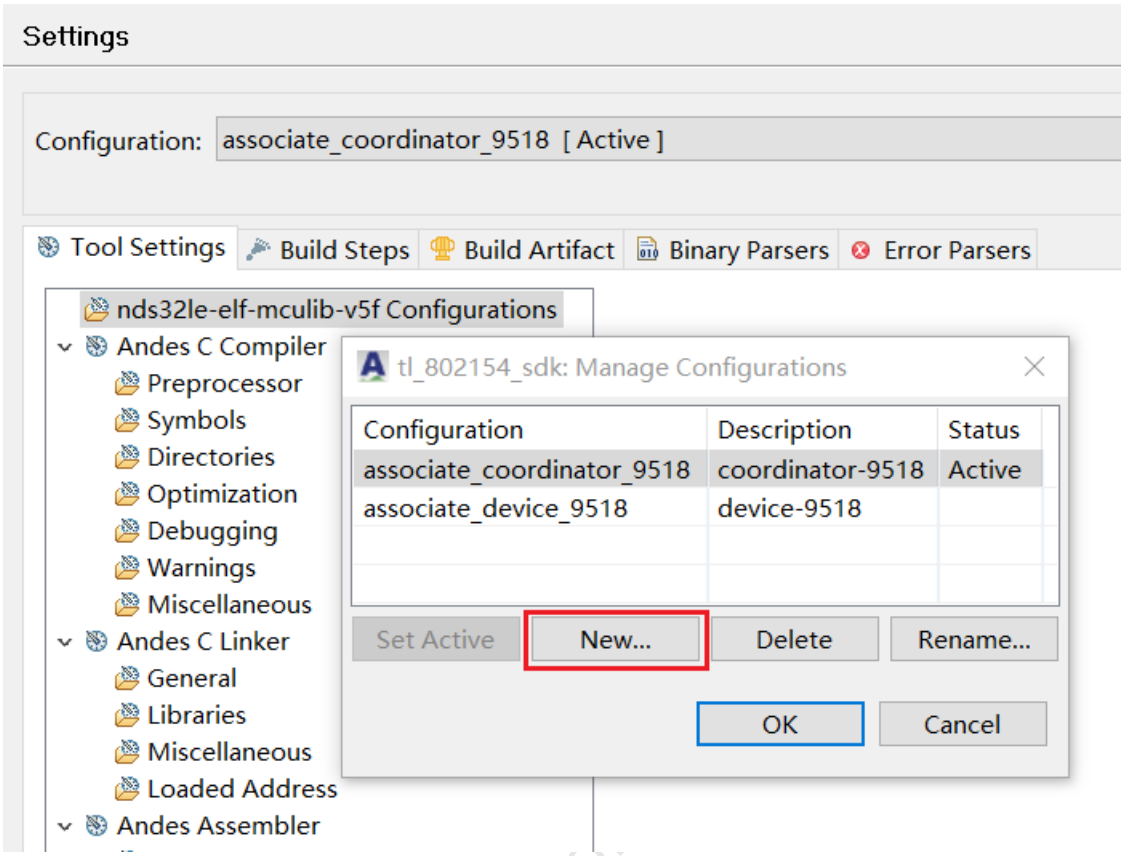

Figure 20: "Adding a new project"

2) Step 2: Click New, the following screen will pop up.

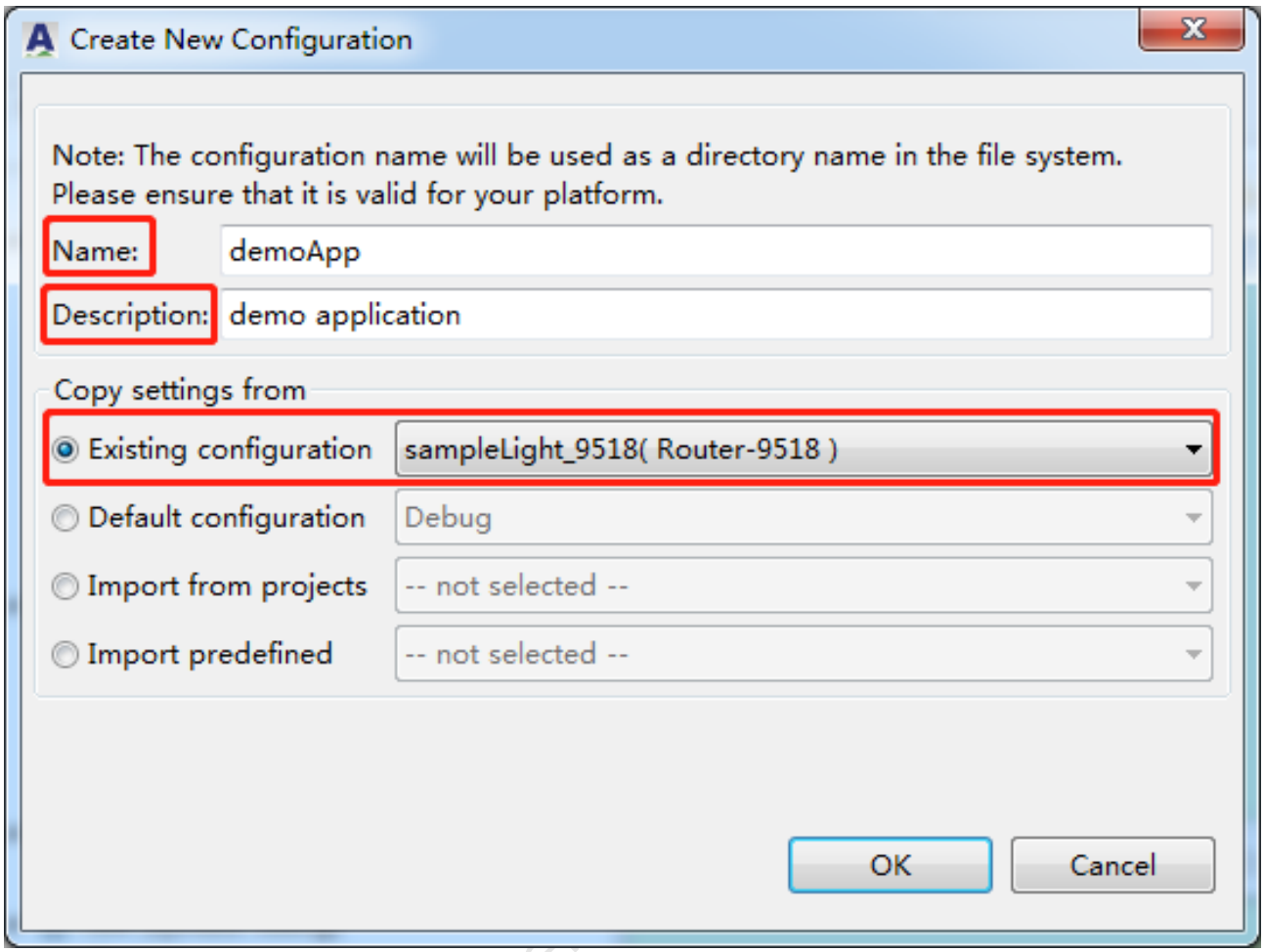

Figure 21: "Configuring a new project"

- Name: project name
- Description: brief project description
- Copy settings from: It is recommended to choose Existing configuration, which can simplify the configuration process.

The principle of selecting Existing configuration is as below.

- If using 9518 chip, select xxx_9518.
- Project for Gateway development, select sampleGw_xxxx; project for Router device development, select sampleLight_xxxx; project for End Device device development, select sampleSwitch_xxxx.

For example, suppose you need to develop a light project with routing function based on 9518, according to this principle, you should choose the configuration of the project "sampleLight_9518" as the configuration of this new project.

If you need to modify the configuration, you can follow the next section (2.2.5 Project configuration description) and adjust it accordingly.

2.2.5 Project configuration description

Enter in order: Project Explorer -> tl_802154_sdk -> Properties -> Settings (take the project associate_coordinator_9518 for example)

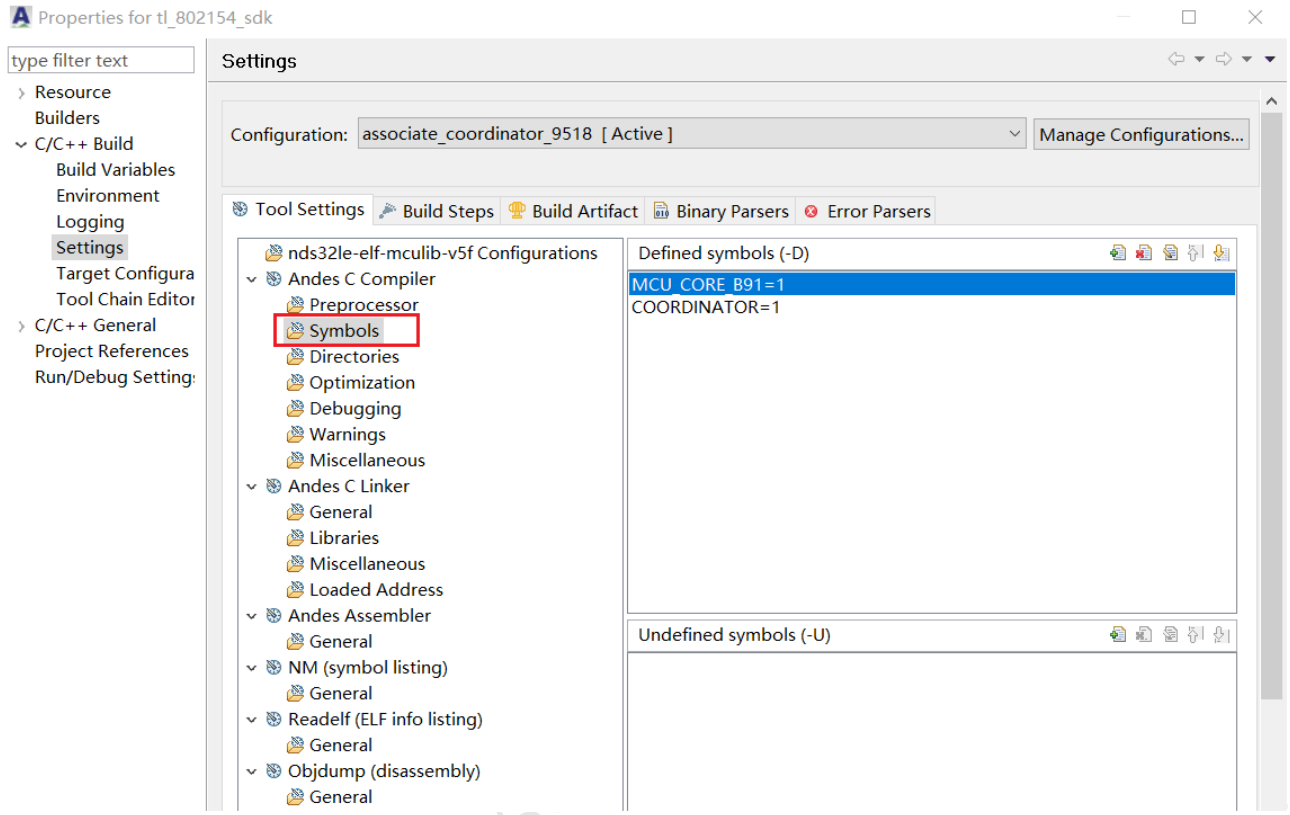

Figure 22: "Symbol definition"

The "Tool Settings" contains some pre-defined configurations for current project:

1) **Device type pre-definition**

Following shows the device setting for coordinator and end device:

-DEND_DEVICE=1：indicates that the project is a device with end device function

-DCOORDINATOR=1：indicates that the item is a device with coordinator function

2) **Platform selection**

- b91 platform：-DMCU_CORE_B91=1

  The startup code cstartup_b91.S 位于 tl_802154_sdk -> platform -> boot -> b91 directory.

3) **Links for lib files**

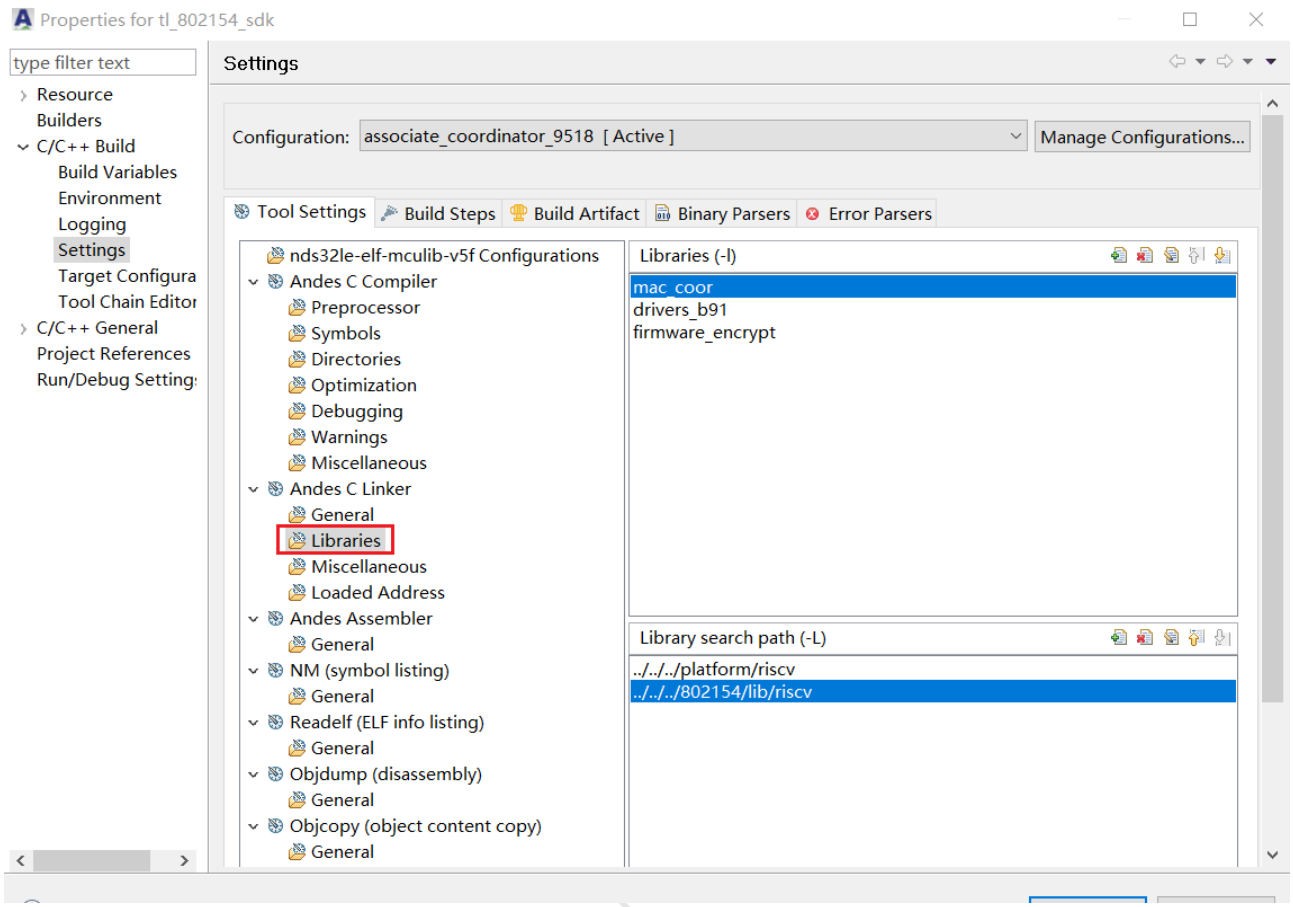

Figure 23: "Library files and paths"

The current SDK includes 2 types of library files: the 802154 stack library and the platform driver library.

- 802154 stack library: libmac_coor.a，libmac_device.a

  Available in the tl_802154_sdk -> 802154 -> lib -> riscv directory.

- Platform driver library: libdrivers_b91.a

  Available in the tl_802154_sdk -> platform -> lib directory.

4) **Link file**

According to the actual application requirements, the user can adjust the appropriate link file as per the different platforms chosen and the memory requirements.

The current SDK provides the default link file boot_b91.link for the b91 platform in the directory corresponding to tl_802154_sdk -> platform -> boot.

As shown in the figure below, the link file to be used can be selected by invoking the script tl_link_load.sh (in the directory of tl_802154_sdk -> tools) during the pre-compiling.

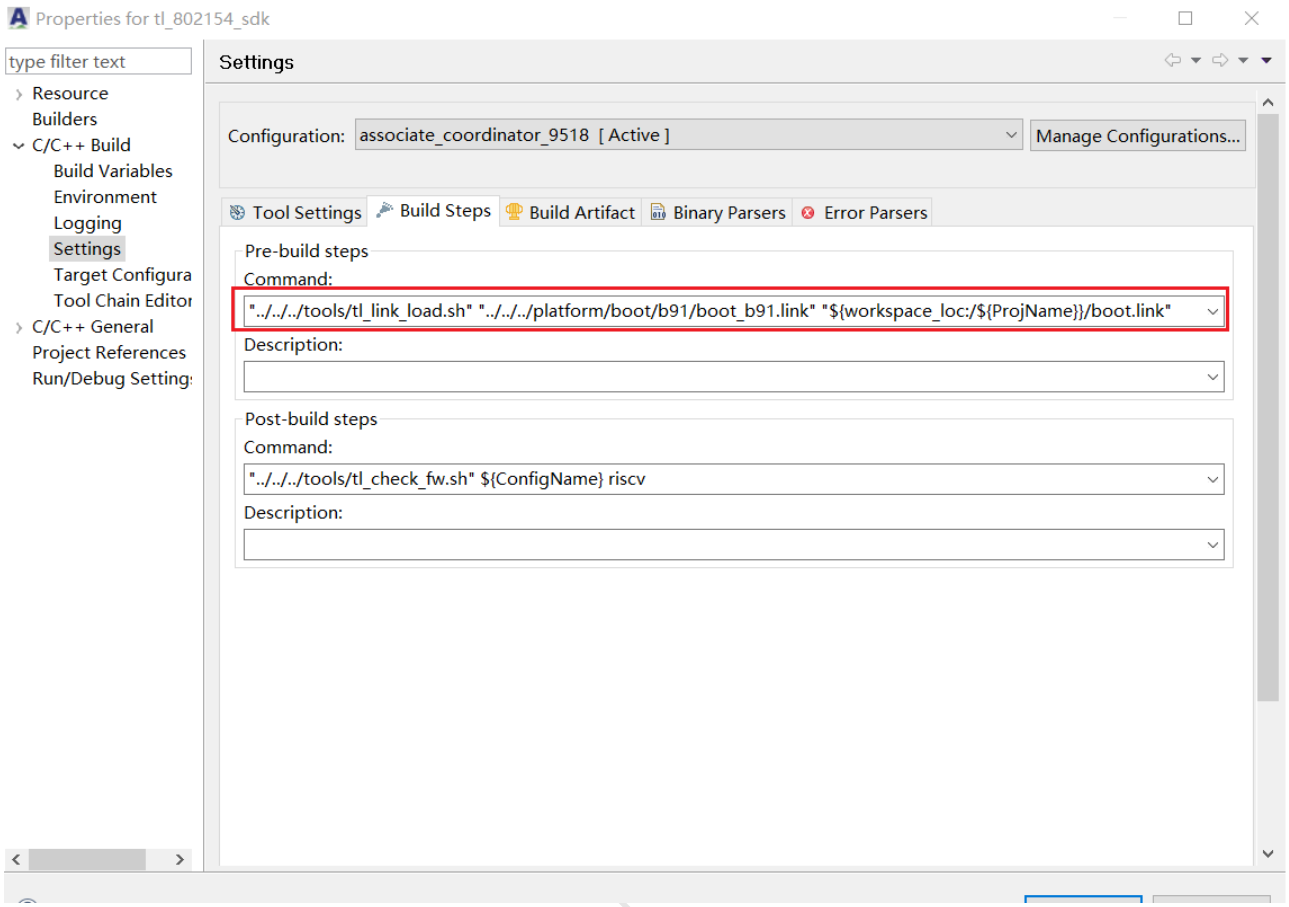

Figure 24: "Link file pre-compiling settings"

5) **.image file check**

In order to ensure the reliability of the downloaded file, a check field is added to the generated image file by the script tl_check_fw.sh (in the directory of tl_802154_sdk -> tools), and during download or the OTA process it will determine whether to update the image by checking whether the check field matches or not.

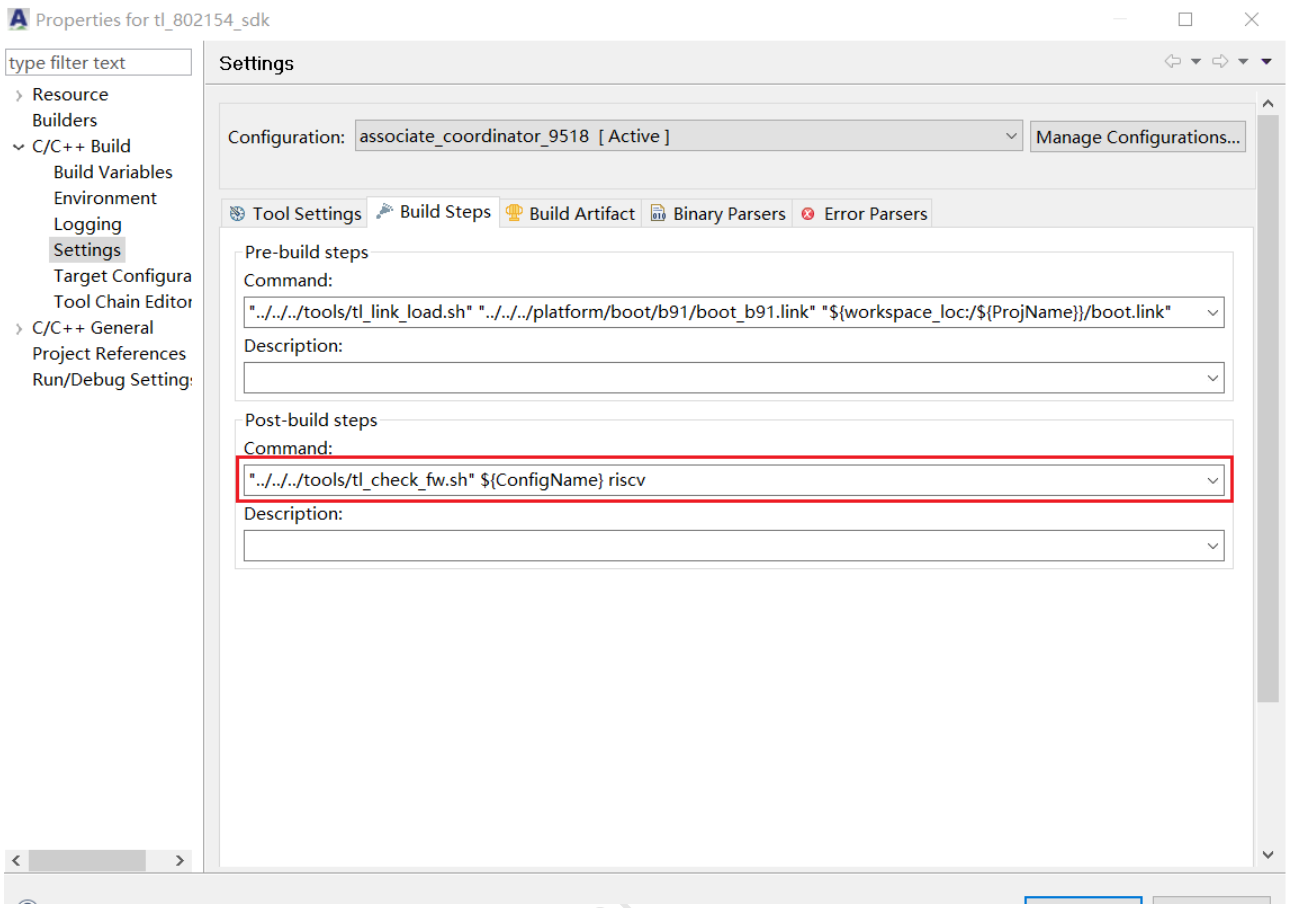

Figure 25: "image check settings"

2.3 App version management

There is a version_cfg.h file in each demo directory to manage the app version. It is very necessary to manage the version of the app, especially during OTA to prevent the risk of being bricked due to incorrect upgrade.

App version is composed of three key character fields, namely Manufacturer Code, Image Type and File Version, which will be stored in a fixed location in the firmware in the form of a small end, as shown below.

assoc_dev_8258.bin

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ASCII
00000000:	58	80	00	00	00	00	5D	02	4B	4E	4C	54	40	01	88	00	X€....
00000010:	06	81	00	00	00	00	30	10	E4	AB	00	00	41	11	52	58	.€....
00000020:	0C	64	81	A2	22	0B	1A	40	C0	06	C0	06	C0	06	C0	06	.d€...
00000030:	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	€.€...
00000040:	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	€.€...
00000050:	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	€.€...
00000060:	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	€.€...
00000070:	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	€.€...
00000080:	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	€.€...
00000090:	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	C0	06	€.€...
000000A0:	C0	06	C0	06	C0	06	C0	06	0C	6C	70	07	C0	46	C0	46	€.€...
000000B0:	6F	00	80	00	51	08	52	09	52	0A	91	02	02	CA	08	50	o.€...
000000C0:	04	B1	FA	87	31	08	C0	6B	32	08	85	06	30	08	C0	6B	.€€...
000000D0:	31	08	85	06	00	A0	34	09	34	0A	91	02	02	CA	08	50	1.€...
000000E0:	04	B1	FA	87	2F	09	32	08	00	FA	08	40	01	B0	48	40	.€€...
000000F0:	3A	08	3B	09	01	50	09	FE	01	41	41	41	3A	08	00	A1	:.;...
00000100:	41	40	AB	A1	01	40	00	A2	06	A3	01	B2	9A	02	FC	CD	A@€...
00000110:	01	A1	41	40	2F	08	30	09	02	EC	0A	40	40	A2	8A	40	.€A...
00000120:	8A	48	D2	F7	D2	FF	01	AA	FA	C0	4A	48	00	A0	82	02	€H€...
00000130:	04	C0	36	08	36	09	0A	48	02	40	11	80	22	09	23	0A	.€6...

Figure 26: "Binary file of image"

Green box: file version

Red box: MANUFACTURER_CODE

Blue box: image type

2.3.1 Manufacturer Code

MANUFACTURER_CODE: The manufacturer code is a 2-byte-long identifier assigned by the Zigbee Alliance to each member company, for example, 0x1141 is the manufacturer code of Telink. Users can modify it to their own manufacturer code.

2.3.2 Image Type

IMAGE_TYPE: The image type is a 2-byte constant, the high byte represents the Chip type and the low byte represents the Image type. The Chip type and Image type are defined in the version_comm.h file, which can be modified or added by users according to their needs.

2.3.3 File Version

FILE_VERSION: The file version is the version number that reflects the firmware release and compiling, and is a constant of 4 bytes in length. The file version should be managed in incremental form, e.g. the current file

version number should be greater than the previous release number, because when OTA, the upgrade will be performed only if the new version number is checked to be greater than the previous version number.

24 Operation mode

Telink 826x/8258/8278 hardware platform supports two boot modes: multi-address boot mode (boot from 0x0 or 0x40000 address) and BootLoader boot mode (boot from BootLoader and jump to App).

Telink 802154 SDK only supports multi-address boot mode, will support bootloader boot mode in the future.

24.1 Multi-address boot mode

Advantages: fast boot; no need to transfer image again after OTA, fast boot after correct verification

Disadvantages: image can only be located at address 0x0 or 0x40000, which will cause discontinuity in flash space allocation; in addition, the size of image (if OTA is supported) can only be less than 208KB.

24.2 Multi-address boot mode flash allocation

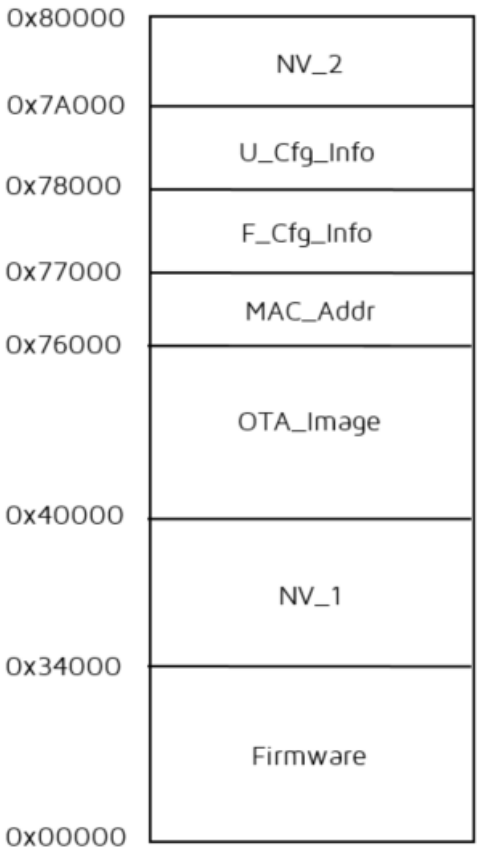

Figure 27: "512k Flash space allocation chart"

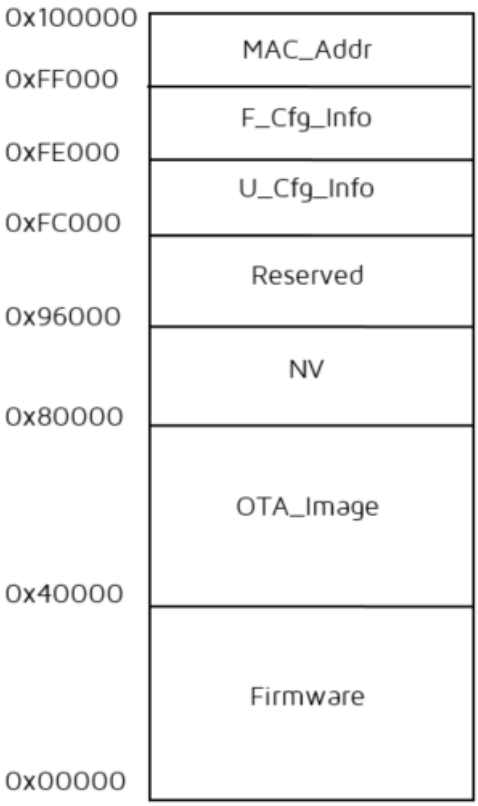

Figure 28: "1M Flash space allocation chart"

2.5 Flash allocation description

1) MAC_Addr

The MAC address is stored in 8 bytes location starting from the0x76000 or 0xFF000 in the flash. The system will check the MAC address information after startup, and if it is found to be 0xFFFFFFFFFFFFFFFF, a MAC address will be generated randomly. A MAC address is pre-written when the chip is out of factory, so erase it carefully.

2) F_Cfg_Info

Factory configuration parameter information. The chip will be pre-written with some calibration parameter information at the factory, such as RF frequency bias calibration, ADC calibration, etc. Please erase it carefully.

3) U_Cfg_Info

Information about user configuration parameters.

4) NV(NV_1, NV_2)

Network information is stored in the NV area after the node is on the network. The 512k Flash network information is stored in two parts, 48kB in NV_1 and 24kB in NV_2. The 1M Flash network information is stored in 88kB in NV.

So, if you only update the firmware or power off and reboot, the network information will not be lost.

5) Firmware and OTA_Image

TLSR8 supports Flash multi-address boot, you can boot from address 0x0 or 0x40000. The Telink 802154 SDK uses this feature with ping-pong mode to implement OTA functionality, using the Firmware and OTA_Image to switch between each other.

2.6 Firmware burning

1) Connect the Telink burning EVK to PC via mini USB cable, the EVK board indicator will blink once to indicate that the EVK is properly connected to the PC; then use three DuPont cables to connect the VCC, GND and SWM of the EVK to the VCC, GND and SWS of the target board to be burned, respectively.

Figure 29: "Burning connection"

2) After the hardware is connected, open the Telink BDT.exe burning tool and prepare to burn the firmware.

a) Select "8258" chip (this document takes 8258 for example).

b) Select "File" -> "Open" and choose the firmware to be burned.

c) Click "Erase" to erase 512K Flash.

d) Click "Download" to download the firmware.

(For detailed usage of the burning tool, please refer to the documentation on using the burning tool.)

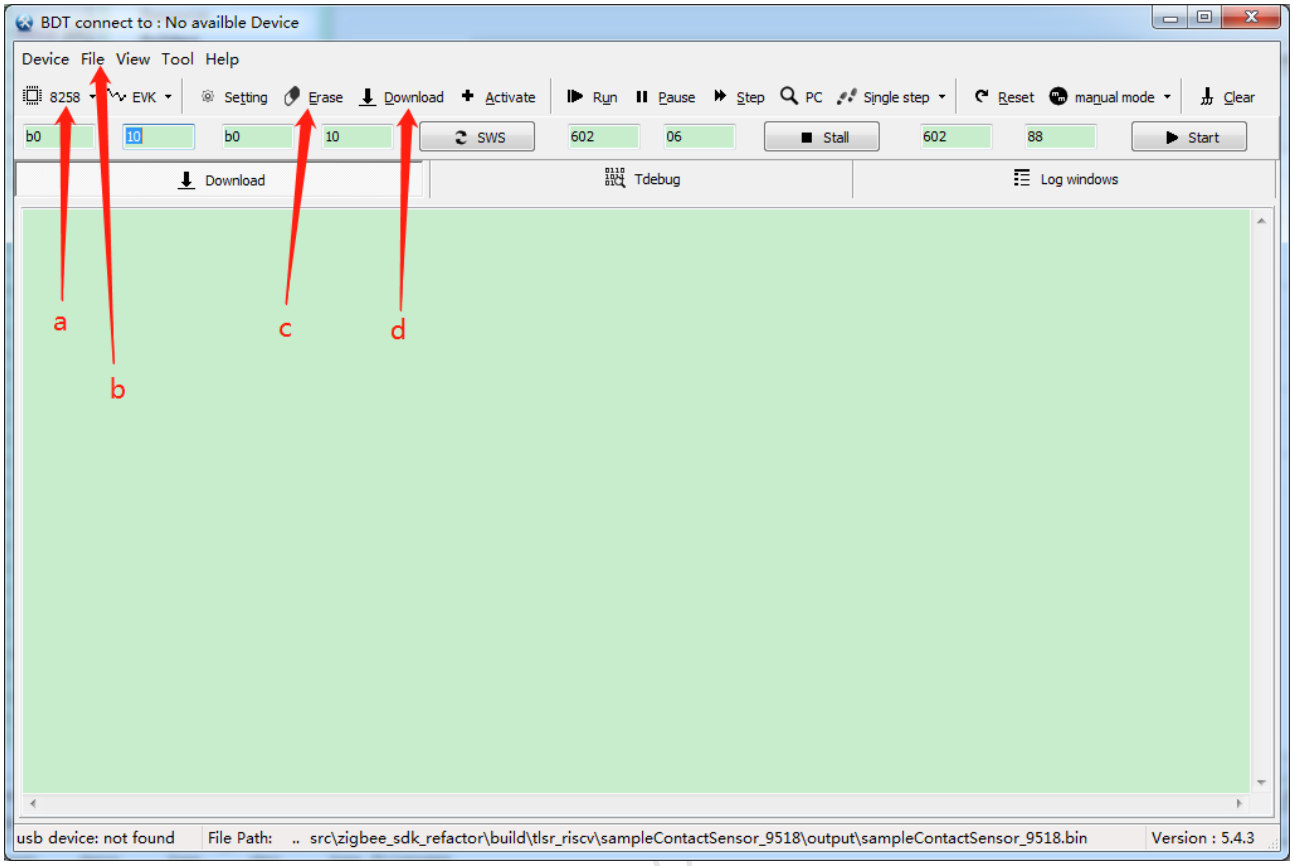

Figure 30: "Burning tool"

3 Software architecture

The directory structure of Telink 802154 SDK is listed in the section 2.1.2, and the files in each directory are described in this chapter.

3.1 Directory description

3.1.1 Hardware platform directory

The current SDK supports b85m (826x, 8258, 8278) and b91m (9518) for multiple platforms (tl_802154_sdk -> platform). Other hardware platforms will be added in the coming versions.

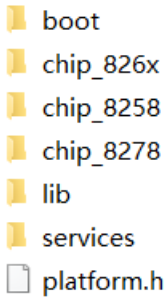

Figure 31: "Hardware platform directory"

- \boot: .S boot code and .link link files for different platforms
- \chip_826x: 826x platform hardware module driver header file
- \chip_8258: 8258 platform hardware module driver header file
- \chip_8278: 8278 platform hardware module driver header file
- \chip_b91: b91 platform hardware module driver header file
- \lib: driver library files for different platforms
- \services: interrupt handling files for different platforms

3.1.2 Common function directory

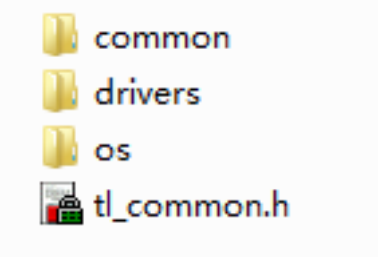

Figure 32: "Common functions directory"

- \common：some common functional functions, such as string, chain table, printing and other processing function files
- \drivers：abstraction layer driver files
- \os：task events, buffer management function files

3.1.3 802154 protocol stack directory

Figure 33: "Protocol stack directory"

In this directory, most are given in the form of .lib file, while ZCL and zbhci, which are closely related to the PHY layer and the application layer, are given as open source code.

- \common：802154 protocol stack configuration-related files
- \lib：802154 protocol stack library file
- \mac：Media Access Control layer related files, 802154 protocol stack core code

3.14 Application layer directory

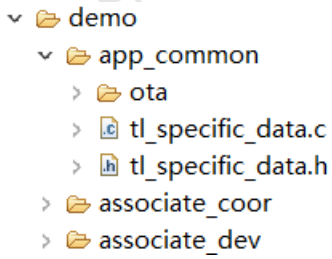

Figure 34: "Application layer directory"

- \app_common：application layer common file directory
- \ota：OTA application code for coordinator and end device

  tl_specific_data.c：Application code function entry

  tl_specific_data.h：Application code configuration files
- \associate_coor：coordinator application samples
- \associate_dev：end device application samples

3.1.5 Project compiling directory

Figure 35: "Project compiling directory"

- build -> tlsr_riscv: the project directory for risc-v platform import and compiling. Please choose to use this directory for TLSR9 series chips.
- build -> tlsr_tc32: the project directory for tc32 platform import and compiling. Please choose to use this directory for TLSR8 series chips.

3.2 Abstract layer driver

Telink 802154 sdk is compatible with many Telink chips, in order to facilitate the unification of the driver interface, a driver abstraction layer has been added, located under proj -> drvier, the specific use of each driver can be found in the following sections.

3.2.1 Platform initialization

- **Initialization**

```
drv_platform_init(void)
```

Complete the initialization of chip configuration, system clock configuration, gpio, radio frequency (RF), timer and other modules required in 802154 application development.

3.2.2 Radio Frequency (RF)

- **Initialization**

```
ZB_RADIO_INIT()
```

- **Mode switching**

```
ZB_RADIO_TRX_SWITCH(mode, chn)
```

mode: RF_MODE_TX = 0, transmit mode

RF_MODE_RX = 1, receive mode

RF_MODE_AUTO = 2, 802154 not used

RF_MODE_OFF, Turn off the RF module

Chn: 11-26 (16 channel values corresponding to 2.4G in 802.15.4)

- **Transmit power**

```
ZB_RADIO_TX_POWER_SET(level)
```

Set transmit power for RF module, different chips correspond to slightly different power values, please refer to the definition of RF_PowerIndexTypeDef (platform/chip/rf_drv.h).

- **Data sending**

```
ZB_RADIO_TX_START(txBuf)
```

Parameter txBuf: the memory address of the data to be sent. Data format dma length (4Bytes: payload length+ 1)+ len(1Byte: payload length+ 2)+ payload.

- **Set the RF receiving buffer**

```
ZB_RADIO_RX_BUF_SET(addr)
```

When setting the RF to Rx mode, you should ensure Rx buffer set to a valid and safe memory address.

- **RSSI obtain**

```
ZB_RADIO_RSSI_GET()
```

When invoking this function, make sure the RF is in Rx mode at this time.

- **Rssi to Lqi**

```
ZB_RADIO_RSSI_TO_LQI(mode, rssi, lqi)
```

mode: valid only for 8269

```
typedef enum{
 RF_GAIN_MODE_AUTO,
 RF_GAIN_MODE_MANU_MAX,
}rf_rxGainMode_t;
```

- **Receive interrupt**

```
rf_rx_irq_handler(void)
```

- **Send interrupt**

```
rf_tx_irq_handler(void)
```

Note: RF interrupt functions rf_rx_irq_handler/rf_tx_irq_handler can only be used in 802154 stack, if the user invokes interrupt generated by ZB_RADIO_TX_START, the user needs to register a new interrupt call-back function to avoid affecting the 802154 stack running state.

3.2.3 GPIO

- **Initialization and configuration**

```
gpio_init()
```

This function is used to initialize gpio by writing the default configuration under platform -> chip_xxxx -> gpio_default.h (which can be modified by board_xxxx.h in the application layer) to the gpio registers.

- **IO function setting**

```
void drv_gpio_func_set(u32 pin)
```

Set IO specific functions.

Parameter pin: See the definition of GPIO_PinTypeDef.

- **GPIO read**

```
void drv_gpio_read(u32 pin);
```

Read the high and low levels of gpio.

- **GPIO write**

```
void drv_gpio_write(u32 pin, bool value);
```

Set the high and low levels of gpio.

- **Output enable**

```
void drv_gpio_output_en(u32 pin, bool enable);
```

- **Input enable**

```
void drv_gpio_input_en(u32 pin, bool enable)
```

- **GPIO interrupt operation**

The chip can support up to 3 external GPIO interrupts at the same time, the interrupt modes are: GPIO_IRQ_MODE, GPIO_IRQ_RISC0_MODE and GPIO_IRQ_RISC1_MODE.

1) Register interrupt service functions

```
int drv_gpio_irq_conf(drv_gpio_irq_mode_t mode,u32 pin,
drv_gpioPoll_e polarity, irq_callback gpio_irq_callback);
```

2) Enable interrupt pin

```
int drv_gpio_irq_en(u32 pin);
int drv_gpio_irq_risc0_en(u32 pin);
int drv_gpio_irq_risc1_en(u32 pin);
```

3.24 UART

- **Pin setting**

```
void drv_uart_pin_set(u32 txPin, u32 rxPin)
```

Before using uart, you should select the corresponding IO as sending and receiving pin of uart.

- **Initialization**

```
void drv_uart_init(u32 baudrate, u8 *rxBuf, u16 rxBufLen, uart_irq_callback uart_recvCb)
```

Baudrate: Baud rate

rxBuf: Receiving buffer

rxBufLen: maximum length of received data

uart_recvCb: Callback function for the application layer when receiving interrupts

- **Sending**

```
u8 drv_uart_tx_start(u8 *data, u32 len)
```

- **Receive interrupt function**

```
void drv_uart_rx_irq_handler(void)
```

- **Send interrupt function**

```
void drv_uart_tx_irq_handler(void)
```

- **Exception processing function**

```
void drv_uart_exceptionProcess(void)
```

Note: When using uart, main_loop should be polled for this exception processing in order to avoid communication exceptions.

3.2.5 ADC

- **Initialization**

```
bool drv_adc_init(void);
```

- **Configuration**

```
void drv_adc_mode_pin_set(drv_adc_mode_t mode, GPIO_PinTypeDef pin)
```

Parameter mode：see drv_adc_mode_tdefinition

Parameter pin：for the pin number used, see GPIO_PinTypeDe definition

- **Get the sample value**

```
u16 drv_get_adc_data(void)
```

3.2.6 PWM

- **Initialization**

```
void drv_pwm_init(void)
```

- **Configuration**

```
void drv_pwm_cfg(u8 pwmId, u16 cmp_tick, u16 cycle_tick)
```

Parameter pwmId：pwm channel

Parameter cmp_tick：the number of ticks that are high in one cycle of PWM

Parameter cycle_tick：the number of ticks contained in one PWM cycle

3.2.7 TIMER

- **Initialization**

```
void drv_hwTmr_init(u8 tmrIdx, u8 mode)
```

Parameter tmrIdx：TIMER_IDX_0,TIMER_IDX_1, TIMER_IDX_2TIMER_IDX_3

Parameter mode：TIMER_MODE_SYSCLK, TIMER_MODE_GPIO_TRIGGER, TIMER_MODE_GPIO_WIDTH, TIMER_MODE_TICK

- **Setting**

```
void drv_hwTmr_set(u8 tmrIdx, u32 t_us, timerCb_t func, void *arg)
```

Parameter tmrIdx: The timer to be used

Parameter t_us: Timing interval, unit: us

Parameter func: The timed interrupt application layer callback function

Parameter arg: Parameters required by the application layer callback function

- **Cancel**

```
void drv_hwTmr_cancel(u8 tmrIdx)
```

- **Interrupt function**

```
void drv_timer_irq0_handler(void)
void drv_timer_irq1_handler(void)
void drv_timer_irq3_handler(void)
```

Where TIMER_IDX_2 is used as watchdog by default, users are advised not to use it.

TIMER_IDX_3 is used as MAC-CSMA and users are advised not to use it.

3.2.8 Watchdog

- **Initialize**

```
void drv_wd_setInterval(u32 ms)
```

Parameter ms: Timeout time

- **Start**

```
void drv_wd_start(void)
```

- **Feed the dog**

```
void drv_wd_clear(void)
```

3.2.9 System Tick

The System Tick counter is a 32-bit length readable counter that automatically adds one every clock cycle.

Since the clock source of System Timer of 826x and other series ICs are different, the System Timer of 8269 is 32M and the System Timer of 8258, 8278 and 9518 is 16M, so there is a difference in the maximum counting time.

- 8269 Maximum Timing Time: (1/32)us*(2^32) is approximately equal to 134 seconds, System Timer Tick runs one cycle every 134 seconds.
- 8258, 8278, 9518 Maximum Timing Time: (1/16)us*(2^32) is approximately equal to 268 seconds, System Timer Tick runs one cycle every 268 seconds.

The user can use the clock_time() interface to get the current tick.

3.2.10 Voltage detection

In order to prevent abnormal operation of low-voltage systems, the SDK provides a voltage detection function based on the ADC driver implementation, which requires below attention.

1) Needs to use the I/O port that supports ADC function, and the I/O port used for ADC detection cannot be used for other functions.

2) When using DRV_ADC_VBAT_MODE mode, the I/O port needs to be floating.

3) When using DRV_ADC_BASE_MODE mode, the I/O port needs to be connected to a voltage test point.

4) B91 can only use DRV_ADC_BASE_MODE mode, and I/O ports need to do external partial voltage processing

- **Initialization**

```
void voltage_detect_init(void)
```

- **Voltage detection**

```
voltage_detect(void)
```

3.2.11 Sleep and wake-up

Telink 802154 SDK provides related low-power management functions. The associate_dev_826x uses suspend mode; associate_dev_8258, associate_dev_8278 and associate_dev_B91 use deep with retention mode, i.e. RAM data can be retained while sleeping (8258 and 8278 support 32k RAM retention, B91 supports 64k RAM retention, please refer to the datasheet for details).

- **Wake-up source type**

Supports button wake-up and timer wake-up.

- **Configure wake-up pins**

If you need the button wake-up function, you need to configure the corresponding wake-up pins and wake-up voltage level first.

```
/**
* @brief Definition for wakeup source and level for PM
*/
drv_pm_pinCfg_t g_switchPmCfg[] =
{
{BUTTON1,   PM_WAKEUP_LEVEL},
{BUTTON2,   PM_WAKEUP_LEVEL},
};
drv_pm_wakeupPinConfig(g_switchPmCfg, sizeof(g_switchPmCfg)/sizeof(drv_pm_pinCfg_t));
```

- **Hibernation functions**

```
drv_pm_lowPowerEnter(void);
```

If button wake-up is enabled, when invoking this function, if the current level of the corresponding pin is the same as the wake-up level, the system cannot effectively enter the low-power state.

- **Hibernation function description**

1) Invoke ingtl_stackBusy() and zb_isTaskDone() to check whether the hibernation conditions are met.

2) Iterate through the software timed task list, check if a timed task exists, if so execute step 3), if not execute step 4).

3) Retrieve the time of the approaching task and use that time as the hibernation time to enter Suspend or DeepRetention hibernation mode, which can be timer wake-up and button wake-up.

4) Enter Deep hibernation mode, you can press the button to wake up.

3.3 Memory management

3.3.1 Dynamic memory management

Telink 802154 SDK provides an interface for dynamic memory allocation and release, which by default supports memory requests up to 142 bytes in length and can be invoked directly by users in development. The detailed description of the interface is as below.

1) Apply for memory

```
u8 *ev_buf_allocate(u16 size);
```

2) Release memory

```
buf_sts_t ev_buf_free(u8 *pBuf);
```

3) Resource allocation

By default it consists of four different sets of memory pools of different numbers and sizes, which can be modified by users according to product requirements and current memory usage.

```
MEMPOOL_DECLARE(size_x_pool, size_x_mem, BUFFER_GROUP_x, BUFFER_NUM_IN_GROUPx);
```

3.3.2 NV Management

Because 802154 needs to store the network information parameters of each layer to be able to recover the network after an abnormal power failure, the SDK divides an area from FLASH specifically for storing such information, which we invoke the NV area, i.e.,NV(NV_1, NV_2) area in section 2.4.

Since the chip FLASH supports only sector (1 sector= 4k) erasure, the minimum unit of information storage module in NV area is 1 sector.

The following information modules are used in the SDK, among them NV_MODULE_USER_INFO1 and NV_MODULE_USER_INFO2 are for users, if you need to add new information modules, please add them at the end, you cannot change the existing module serial number.

```
typedef enum {
    NV_MODULE_MAC_INFO              = 0,
    NV_MODULE_NWK_FRAME_COUNT       = 1,
    NV_MODULE_USER_INFO1            = 2,
    NV_MODULE_USER_INFO2            = 3,
    NV_MAX_MODULS
}nv_module_t;
```

The information module consists of multiple entry messages, for example NV_MODULE_USER_INFO1 consists of a series of NV_ITEM_USER_INFO. The entries are defined as follows. If you need to add new entry information, please add them at the end, you cannot change the existing entry serial number.

```
typedef enum {
    NV_ITEM_ID_INVALID      = 0,/* Item id 0 should not be used. */
    NV_ITEM_MAC_INFO        = 1,
    NV_ITEM_USER_INFO1      = 0x10,
    NV_ITEM_USER_INFO2      = 0x20,
    NV_ITEM_ID_MAX          = 0xFF,/* Item id 0xFF should not be used. */
}nv_item_t;
```

- Each module occupies 2 or (2*n) sectors, in the format of

  sector info + item index + item content

where the sector info identifies whether this module is valid, the length is sizeof(nv_sect_info_t), followed by the index of the item to be written, the item content starts at a 512Byte or 1024Byte offset from the module's first address.

- To avoid frequent erasure operations, NV uses a single-module append-write method until one sector is full, moves the valid information to another sector, and then erases the contents of the previous sector.
- The interface for NV area read and write operations is as follows.

```
nv_sts_t nv_flashWriteNew(u8 single, u16 id, u8 itemId, u16 len, u8 *buf);
nv_sts_t nv_flashReadNew(u8 single, u8 id, u8 itemId, u16 len, u8 *buf);
```

Note: For a certain item, if there is only one valid value, then single is 1; when this item is written again, previous valid item will cleared; otherwise, you need to retrieve the same item according to its content, set it to invalid, and then write a new item.

34 Task management

34.1 Single task queue

- Interface function

```
TL_SCHEDULE_TASK(tl_zb_callback_t func, void *arg)
```

Parameter func: Task callback function

Parameter arg: Required parameters for the task

- Execute only once, process in order without priority
- Recommended use occasions.

  1) Needs to avoid stack overflow problems caused by deep nesting of functions.

  2) In the interrupt function, avoid consuming too much time in the interrupt function by pressing data and events into the queue.
- Size: 32 (use to avoid pressing in too many tasks at once)

34.2 Standing task queue

- **Task registration** (start after task registration by default)

```
ev_on_poll(ev_poll_e e, ev_poll_callback_t cb)
```

- **Task suspension**

```
ev_disable_poll(ev_poll_e e, ev_poll_callback_t cb)
```

- **Task restart**

```
ev_enable_poll(ev_poll_e e, ev_poll_callback_t cb)
```

After this task is registered, it will be executed in the main loop all the time.

34.3 Software timed task

In order to facilitate users to implement timing tasks that do not require high time accuracy, Telink 802154 SDK provides a software timing task management mechanism.

It should be noted that from SDK V3.6.2 version, the time unit of software timing task management changed from original ticker to millisecond, which solves the problem of long time timing task demand.

34.3.1 Interface functions

- **Task registration:** TL_ZB_TIMER_SCHEDULE(cb, arg, timeout)

```
/**
* @param[in]        func - the callback of the timer event
*
* @param[in]        arg - the parameter to the callback
*
* @param        cycle - the timer interval
*
* @return       the status
*/
```

- **Task cancellation:** TL_ZB_TIMER_CANCEL(evt)

```
/**
 @param[in]         evt - the pointer to the timer event pointer
*
* @return       the status
*/
```

34.3.2 Attentions

1) Three uses of the return value of the timed task callback function.

- If the return value is less than 0, the task is automatically deleted after execution and the task ceases to exist.
- If the return value is equal to 0, the callback function always triggered periodically using the t_ms in previous start-up.
- If the return value is greater than 0, the return value is used as the new period to trigger the callback function at regular intervals, in ms.

Note: Avoid using TL_ZB_TIMER_CANCEL() in the interrupt function.

34.3.3 Examples

When VK_SW1 button is pressed, a 10-second timed task started or cancelled, command to broadcast On/Off Toggle executed once when the time is reached, and exits when the execution is finished. The code is as follows.

```
ev_timer_event_t *brc_toggleEvt = NULL;

s32 brc_toggleCb(void *arg)
{
    epInfo_t dstEpInfo;
    TL_SETSTRUCTCONTENT(dstEpInfo, 0);

    dstEpInfo.dstAddrMode = APS_SHORT_DSTADDR_WITHEP;
    dstEpInfo.dstEp = SAMPLE_GW_ENDPOINT;
    dstEpInfo.dstAddr.shortAddr = 0xffff;
    dstEpInfo.profileId = HA_PROFILE_ID;
    dstEpInfo.txOptions = 0;
    dstEpInfo.radius = 0;

    zcl_onOff_toggleCmd(SAMPLE_GW_ENDPOINT, &dstEpInfo, FALSE);

    brc_toggleEvt = NULL;
    return -1;
}

void buttonShortPressed(u8 btNum)
{
    if(btNum == VK_SW1){
        if(!brc_toggleEvt){
            brc_toggleEvt = TL_ZB_TIMER_SCHEDULE(brc_toggleCb,
                            NULL,
                            10 * 1000);
        }else{
            TL_ZB_TIMER_CANCEL(&brc_toggleEvt);
        }
}
}
```

4 MAC commonly used APIs

The MAC provides a series of APIs that enable the interaction of commands, data and status between the application layer (APP) and the MAC layer:

a) Request and Confirm: The application layer sends a command request and data request to the MAC layer, and receives a status confirmation (confirm) from the MAC layer.

b) Indication: The application layer receives MAC layer commands (COMMAND-indication) and data (DATA-indication).

The application layer is implemented through the MAC Layer Management Entity (MLME) and the MAC CommonLayer Service Access Point (MCPS).

4.1 MAC Layer Management Entity (MLME)

4.1.1 MLME-POLL

1) **MLME-POLL.request**

Implementation functions:

```
u8  tl_MacMlmePollRequestSend(mac_mlme_poll_req_t req)
```

- Req is a structure variable that points to the MLME-POLL.request primitive.

MLME-POLL.request primitive data structure: mac_mlme_poll_req_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

2) **MLME-POLL.confirm**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_POLL_CONFIRM, MyPollCnfCb);
```

- CALLBACK_POLL_CONFIRM is the MLME-POLL.confirm callback function ID and is a fixed value.
- void MyPollCnfCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-POLL.confirm primitive.

MLME-POLL.confirm primitive data structure: mac_mlme_poll_conf_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

3) **MLME-POLL.indication**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_POLL_INDICATION, MyPullIndCb);
```

- CALLBACK_POLL_INDICATION is the MLME-POLL.confirm callback function ID and is a fixed value.
- void MyPullIndCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-POLL.indication primitive.

MLME-POLL.indication primitive data structure: mac_mlme_poll_ind_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4.1.2 MLME-ASSOCIATE

1) **MLME-ASSOCIATE.request**

Implementation functions:

```
u8  tl_MacMlmeAssociateRequestSend(zb_mlme_associate_req_t req)
```

- Req is a structure variable that points to the MLME-ASSOCIATE.request primitive.

MLME-ASSOCIATE.request primitive data structure: zb_mlme_associate_req_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

2) **MLME_ASSOCIATE.confirm**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_ASSOCIATE_CONFIRM, MyAssocCnfCb);
```

- CALLBACK_ASSOCIATE_CONFIRM is the MLME-ASSOCIATE.confirm callback function ID and is a fixed value.
- void MyAssocCnfCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-ASSOCIATE.confirm primitive).

MLME-ASSOCIATE.confirm primitive data structure: zb_mlme_associate_conf_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

3) **MLME_ASSOCIATE.indication**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_ASSOCIATE_INDICATION, MyAssociateIndCb);
```

- CALLBACK_ASSOCIATE_INDICATION is the MLME-ASSOCIATE.indication callback function ID and is a fixed value.
- void MyAssociateIndCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-ASSOCIATE.indication primitive.

MLME- ASSOCIATE.indication primitive data structure: zb_mlme_associate_ind_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4) **MLME_ASSOCIATE.response**

Implementation functions:

```
u8  tl_MacMlmeAssociateResponseSend(zb_mlme_associate_resp_t req)
```

- Req is a structure variable that points to the MLME-ASSOCIATE.response primitive.

MLME-ASSOCIATE.response primitive data structure: zb_mlme_associate_resp_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4.1.3 MLME-SCAN

1) **MLME_SCAN.request**

Implementation functions:

```
u8  tl_MacMlmeScanRequest(zb_mac_mlme_scan_req_t req)
```

- Req is a structure variable that points to the MLME-SCAN.request primitive.

MLME-SCAN.request primitive data structure: zb_mac_mlme_scan_req_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

2) **MLME_SCAN.confirm**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_SCAN_CONFIRM, MyScanCnfCb);
```

- CALLBACK_SCAN_CONFIRM is the MLME-SCAN.confirm callback function ID and is a fixed value.
- void MyScanCnfCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-SCAN.confirm primitive.

MLME-SCAN.confirm primitive data structure: zb_mac_mlme_scan_conf_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4.1.4 MLME-START

1) **MLME_START.request**

Implementation functions:

```
u8  tl_MacMlmeStartRequest(zb_mac_mlme_start_req_t req)
```

- Req is a structure variable that points to the MLME-ASSOCIATE.request primitive.

MLME-ASSOCIATE.request primitive data structure: zb_mlme_associate_req_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

2) **MLME_START.confirm**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_START_CONFIRM, MyStartCnfCb);
```

- CALLBACK_START_CONFIRM is the MLME-START.confirm callback function ID and is a fixed value.
- void MyStartCnfCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME- START.confirm primitive.

MLME- START.confirm primitive data structure: mac_mlme_startCnf_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4.1.5 MLME-DISASSOCIATE

1) **MLME_DISASSOCIATE.request**

Implementation functions:

```
u8  tl_MacMlmeDisassociateRequestSend(zb_mlme_disassociate_req_t req)
```

- Req is a structure variable that points to the MLME-DISASSOCIATE.request primitive.

MLME- DISASSOCIATE.request primitive data structure: zb_mlme_disassociate_req_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

2) **MLME_DISASSOCIATE.confirm**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_DISASSOCIATE_CONFIRM, MyDisassocCnfCb);
```

- CALLBACK_ASSOCIATE_CONFIRM is the MLME-ASSOCIATE.confirm callback function ID and is a fixed value.
- void MyDisassocCnfCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-DISASSOCIATE.confirm primitive.

MLME- DISASSOCIATE.confirm primitive data structure: zb_mlme_disassociate_conf_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

3) **MLME_DISASSOCIATE.indication**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_DISASSOCIATE_INDICATION, MyDisassociateIndCb);
```

- CALLBACK_DISASSOCIATE_INDICATION is the MLME-DISASSOCIATE.indication callback function ID and is a fixed value.
- void MyDisassociateIndCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-DISASSOCIATE.indication primitive.

MLME-DISASSOCIATE.indication primitive data structure: zb_mlme_disassociate_ind_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4.1.6 MLME-BEACON-NOTIFY

1) **MLME-BEACON-NOTIFY.indication**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_BEACON_NOTIFY_INDICATION, MyDisassociateIndCb);
```

- CALLBACK_BEACON_NOTIFY_INDICATION is the MLME-BEACON-NOTIFY.indication callback function ID and is a fixed value.
- void MyDisassociateIndCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-BEACON-NOTIFY.indication primitive.

MLME-BEACON-NOTIFY.indication primitive data structure: zb_mlme_beacon_notify_ind_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4.1.7 MLME-COMM-STATUS

1) **MLME-COMM-STATUS.indication**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_COMM_STATUS_INDICATION, MyStateIndCb);
```

- CALLBACK_COMM_STATUS_INDICATION is the MLME-COMM-STATUS.indication callback function ID and is a fixed value.
- void MyStateIndCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MLME-COMM-STATUS.indication primitive.

MLME-COMM-STATUS.indication primitive data structure: zb_mlme_comm_status_ind_t.

Function declaration files:upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> tl_zb_mac.h).

4.2 MAC Common Layer Service Access Point (MCPS)

4.2.1 MCPS-DATA

1) **MCPS-DATA.request**

Implementation functions:

```
u8 tl_MacMcpsDataRequestSend(zb_mscp_data_req_t req,u8 *payload,u8 pay_len)
```

- Req is a structure variable that points to the MCPS-DATA.request primitive.
- Payload is a pointer to the data to be sent.
- Pay_len is the length of the data to be sent.

MCPS-DATA.request primitive data structure: zb_mscp_data_req_t.

Function declaration files: upper_layer.h (Pathtl_802154_sdk -> 802154 -> mac -> includes).

2) **MCPS-DATA.confirm**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_MCPS_DATA_CONFIRM, MyDataCnfCb);
```

- CALLBACK_MCPS_DATA_CONFIRM is the MCPS-DATA.confirm callback function ID and is a fixed value.
- void MyDataCnfCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MCPS-DATA.confirm primitive.

MCPS-DATA.confirm primitive data structure: zb_mscp_data_conf_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

3) **MCPS-DATA.indication**

Register callback functions:

```
UpperLayerCallbackSet(CALLBACK_DATA_INDICATION, MyDataIndCb);
```

- CALLBACK_DATA_INDICATION is the MCPS-DATA.indication callback function ID and is a fixed value.
- void MyDataIndCb (unsigned char *pData) is a callback function declared by the application layer. pData is a pointer to the MCPS-DATA.indication primitive.

MCPS-DATA.indication primitive data structure: zb_mscp_data_ind_t.

Function declaration files: upper_layer.h (Path: tl_802154_sdk -> 802154 -> mac -> includes).

4.3 ASP(ATTRIBUTE SETTING)

The MAC PIB attribute is set via ASP and this part of the code is open source.

File path: tl_802154_sdk -> 802154 -> mac -> mac_pib.c

4.3.1 Write MAC PIB attribute

```
u8 tl_zbMacAttrSet(u8 attribute, u8 *value, u8 index)
```

- attribute: MAC PIB attribute id
- value: pointer to write the value of the corresponding MAC PIB attribute.
- index: If attribute is written as an encryption-related attribute id, such as MAC_KEY_TABLE, MAC_DEVICE_TABLE, MAC_SECURITY_LEVEL_TABLE, the MAC PIB attributes corresponding to these attributes id are all arrays of a finite number, the index is the corresponding entry number to be written to; if the attribute has other values, the index is the length of the corresponding MAC PIB attribute value.
- Return value: it returns MAC_SUCCESS If the write succeeds, otherwise it returns MAC_INVALID_PARAMETER.

4.3.2 Read MAC PIB attribute

```
u8 tl_zbMacAttrGet(u8 attribute, u8* value, u8* index)
```

- attribute: MAC PIB attribute id
- value: pointer to read the value of the corresponding MAC PIB attribute
- index: If attribute is written as an encryption-related attribute id, such as MAC_KEY_TABLE, MAC_DEVICE_TABLE, MAC_SECURITY_LEVEL_TABLE, the MAC PIB attributes corresponding to these attributes id are all arrays of a finite number, the index is the corresponding entry number to be read; if the attribute has other values, the index is the length of the corresponding MAC PIB attribute value.
- Return value: it returns MAC_SUCCESS If the read succeeds, otherwise it returns MAC_INVALID_PARAMETER.

At the first power-up, the SDK defines the MAC PIB default value.

File path: tl_802154_sdk -> 802154 -> common -> zb_config.c

Default value variable: const tl_zb_mac_pib_t macPibDefault

4.3.3 Example of reading and writing a MAC PIB attribute

1) **associate device set PAN ID**

```
u8 len = 2;//pan id length is 2 bytes
u16 panid = 0xbeef;
tl_zbMacAttrSet(MAC_ATTR_PAN_ID, &panid ,len);
```

- MAC PIB attribute ID: attribute = MAC_ATTR_PAN_ID
- Write value pointer: value = &panid
- Write length: len = 2

2) **associate device read associate coordinator long address()**

```
u8 coor_ext_short[8] = {0};
u8 len=0;
tl_zbMacAttrGet(MAC_ATTR_COORDINATOR_EXTENDED_ADDRESS,(u8*)coor_ext_short,
&len);
```

- MAC PIB attribute ID:
  attribute = MAC_ATTR_COORDINATOR_EXTENDED_ADDRESS
- Read value pointer: value = &panid
- Read length pointer: index = &len

5 802.154 SDK application development

5.1 Hardware selection

Telink 802154 SDK demo can run on different series of chips and different hardware boards, users need to do the corresponding configuration for different hardware conditions.

5.1.1 Chip model confirmation

Telink 802154 SDK lists the following chip types in comm_cfg.h file. Before compiling project examples, users should check whether the chip type selected in version_cfg.h under the corresponding project is the same as the actual chip type used.

5.1.1.1 comm_cfg.h chip type definition

```
/* Chip IDs */
#define TLSR_8267           0x00
#define TLSR_8269           0x01
#define TLSR_8258_512K      0x02
#define TLSR_8258_1M        0x03
#define TLSR_8278           0x04
#define TLSR_9518           0x05
```

5.1.1.2 version_cfg.h chip type selection

```
#if defined(MCU_CORE_826x)
    #if (CHIP_8269)
        #define CHIP_TYPE        TLSR_8269
    #else
        #define CHIP_TYPE        TLSR_8267
    #endif
#elif defined(MCU_CORE_8258)
        #define CHIP_TYPE        TLSR_8258_512K//TLSR_8258_1M
#elif defined(MCU_CORE_8278)
        #define CHIP_TYPE        TLSR_8278
#elif defined(MCU_CORE_B91)
        #define CHIP_TYPE        TLSR_9518
#endif
```

5.1.2 Target board selection

Telink 802154 SDK provides demos that can run on EVK boards or USB Dongle, and the user can change the target board by selecting the app_cfg.h file under the corresponding demo.

```
/* Board ID */
#define BOARD_826x_EVK                  0
#define BOARD_826x_DONGLE               1
#define BOARD_826x_DONGLE_PA            2
#define BOARD_8258_EVK                  3
#defineBOARD_8258_EVK_V1P2              4//C1T139A30_V1.2
#define BOARD_8258_DONGLE               5
#define BOARD_8278_EVK                  6
#define BOARD_8278_DONGLE               7
#define BOARD_9518_EVK                  8
#define BOARD_9518_DONGLE               9

/* Board define */
#if defined(MCU_CORE_8258)
#if (CHIP_TYPE == TLSR_8258_1M)
    #define FLASH_CAP_SIZE_1M       1
#endif
    #define BOARD                   BOARD_8258_DONGLE
    #define CLOCK_SYS_CLOCK_HZ      48000000
#elif defined(MCU_CORE_8278)
    #define FLASH_CAP_SIZE_1M       1
    #define BOARD                   BOARD_8278_DONGLE
    #define CLOCK_SYS_CLOCK_HZ      48000000
#elif defined(MCU_CORE_B91)
    #define FLASH_CAP_SIZE_1M       1
    #define BOARD                   BOARD_9518_DONGLE
    #define CLOCK_SYS_CLOCK_HZ      48000000
#else
    #error "MCU is undefined!"
#endif
```

During the pre-compiling stage, we need to load corresponding board-level configuration depending on the previous chip type selection. The configuration file is board_xx.h in each project directory, and you need to pay attention to whether the FLASH capacity of the target board is consistent with the board configuration to prevent errors in loading data.

```
#defineFLASH_CAP_SIZE_1M      1
```

5.2 Print debug configuration

5.2.1 UART print

To avoid the waste of hardware UART resources, we implement the UART print function (printf() function) by simulating UART TX through GPIO, and the user can change the appropriate GPIO at will. The configuration method is as follows.

1) **Print enable configuration:**

```
#define UART_PRINTF_MODE     1
```

2) **Print port configuration:**

```
#define DEBUG_INFO_TX_PIN     GPIO_PC4//print
```

3) **Baud rate configuration:**

```
#define BAUDRATE              1000000//1M
```

Note:

1. 826x supports a maximum of 2M baud rate, while 8258, 8278 and 9518 supports a maximum of 1M baud rate.

2. Do not use the print in the interrupt processing function to prevent printing more data or function nesting too deep resulting in interrupt stack overflow.

5.2.2 USB print

USB print function can be used with Telink BDT tool. The configuration is as follows.

1) **Print enable configuration:**

```
#define USB_PRINTF_MODE     1
```

2) **Enables the USB port:**

```
#define HW_USB_CFG()        do{ \
                                usb_set_pin_en();\
                            }while(0)
```

Note:

The USB print function is disabled when the ZBHCI_USB_CDC or ZBHCI_USB_HID function is used.

5.3 802154 development process

5.3.1 Application layer initialization (user_init)

The application layer initialization mainly consists of 802154 stack and 802154 application layer initialization.

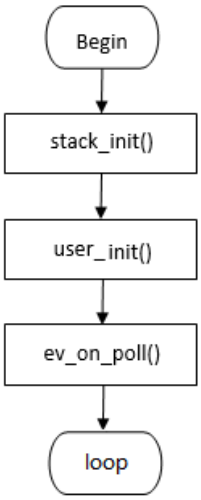

Figure 36: "Application layer initialization flow"

1) **os_init(u8 isRetention)**

For power-up devices with isRetention of 0, the os_init function completes the initialization of the task queue, as well as non-stack related work such as user memory.

2) **user_init()**

The user initialize the application code according to the specific product features.

- Protocol stack initialization:

```
zb_init();
```

The initialization of the mac layer. For not factory-new devices, it will read incoming network information from NV and restore network information and attributes of each layer.

Note: For devices that are already on the network, it is not possible to change the attributes by modifying the default attribute table configuration in zb_config.c. If you want to modify the attributes, you must do so through the u8 tl_zbMacAttrSet(u8 attribute, u8 *value, u8 index) function and attribute.

- Set exception handler callback function:

```
sys_exceptHandlerRegister(sys_exception_cb_t cb)
```

Register an exception status callback function, which defaults to an exception flashing light in the SDK.

- Register MAC callback function:

```
UpperLayerCallbackSet(unsigned char Index, UpperLayerCb_Type Callback)
```

Register the corresponding callback handler function with the primitive ID as required.

3) **Ev_on_poll()**

Register user polling events.

5.3.2 Parameters configuration

Users can modify the network parameters configuration in 802154 -> common -> zb_config.c according to their requirements.

- **macPibDefault**

The basic parameter information of the MAC layer.

5.3.3 Data security

The 15.4 SDK provides the encryption mode of the 802.15.4(2006) standard, and the 15.4 encryption function is enabled by default in the SDK. The application layer only needs to configure the MAC PIB encryption mode and set the encryption mode corresponding to the MAC PIB before sending.

- MAC PIB encryption mode: Encryption examples have been added to the demos provided with the SDK, please refer to the functions for details: void add_key_material(void)
- Set the encryption mode before sending: Before sending, set the value of the struct mac_sec_t sec. 15.4 SDK will do a mode and key check, and if it agrees with the MAC PIB, return MAC_SUCCES in the corresponding confirm callback function, otherwise return some other value. Reference function: void mac_send_data_indirect(void *arg):

app.c

```
void mac_send_data_indirect(void *arg)
{
    static unsigned int test_cnt = 0xaa000001;
    unsigned char *pData = ev_buf_allocate(sizeof(zb_mscp_data_req_t)+sizeof(test_cnt));
    if (NULL == pData) {
        while(1);
    }
    zb_mscp_data_req_t *req = (zb_mscp_data_req_t *)pData;
    memset(req, 0, sizeof(zb_mscp_data_req_t));
    req->srcAddr.addrMode = ZB_ADDR_16BIT_DEV_OR_BROADCAST; //16-bit short address mode
    req->dstAddr.addrMode = ZB_ADDR_16BIT_DEV_OR_BROADCAST; //16-bit short address mode
    req->dstAddr.addr.shortAddr = end_device.shortAddr;

    u8 len=0;
    tl_zbMacAttrGet(MAC_ATTR_PAN_ID,(u8 *)&req->dstPanId,&len);

    req->msduHandle = 0;
    req->txOptions = MAC_TX_OPTION_ACKNOWLEDGED_BIT|MAC_TX_OPTION_INDIRECT_TRANSMISSION_BIT;

    //security setting
    req->sec.key_id_mode = KEY_ID_MODE_KEY_EXPLICIT_8;
    req->sec.securityLevel = SECURITY_LEVEL_ENC_MIC_64;

    memcpy(req->sec.key_source,default_key_source,sizeof(default_key_source));
    req->sec.key_index = default_key_index;

    u8 *pay = (u8 *)&test_cnt;
    u8 pay_len = sizeof(test_cnt);
    test_cnt++;
    tl_MacDataRequestSend(req,pay,pay_len);
    ev_buf_free(pData);
}
```

Figure 37: "Application layer encryption functions"

Decryption is done by the MAC, the application layer only needs to configure the MAC PIB.

54 Work process

54.1 Network access process

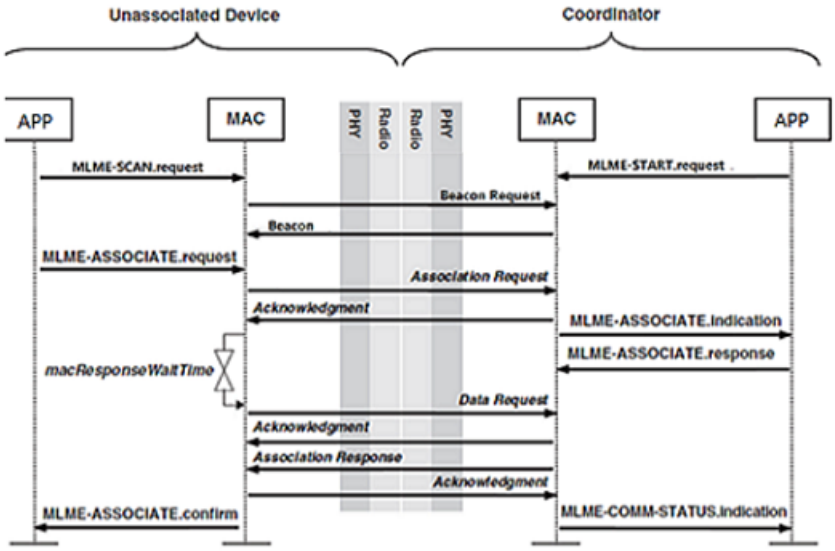

Figure 38: "Network access process"

The association process is the process by which an end device joins a network. The MAC layer provides the MLME-ASSOCIATE command to the application layer and upon completion of this process, the end device joins a network provided by the coordinator.

Associate process:

1) End device

   i) Start an active scan via MLME_SCAN.request and return the results to the application layer via the MLME-SCAN.confirm registration callback function after the scan is finished.

   MLME_SCAN.request api: MacMlmeScanRequest()

   MLME_SCAN.confirm api: UpperLayerCallbackSet(CALLBACK_SCAN_CONFIRM, MyScanCnfCb);

   ii) Based on the active scan result, initiate an association to the coordinator via MLME-ASSOCIATE.request and wait for an associate response packet from the coordinator (or timeout).

   MLME-ASSOCIATE.request api: tl_zbMacAssociateRequest

   iii) The association result is returned to the application layer via the registration callback function of MLME-ASSOCIATE.confirm.

   MLME-ASSOCIATE.confirm api: UpperLayerCallbackSet(CALLBACK_START_CONFIRM, MyStartCnfCb);

2) Coordinator

   i) Start energy scan/active scan with MLME_START.request, create PAN, and then it is in listening state.

MLME_START.request api：tl_zbMacStartRequest()

ii) When MLME-ASSOCIATE.request is received from the end device, it sends an association response, and then informs the application layer of the association result via MLME-COMM-STATUS.indication.

If successful, the status is MAC_SUCCESS, other status is failure, such as pending list full MAC_STA_TRANSACTION_OVERFLOW, etc.

Specific state refers to enumeration variable mac_sts_t：tl_802154_sdk -> 802154 -> mac -> includes -> tl_zb_mac.h

MLME-COMM-STATUS.indication api：UpperLayerCallbackSet(CALLBACK_COMM_STATUS_INDICATION, MyStateIndCb);

54.2 Data interaction

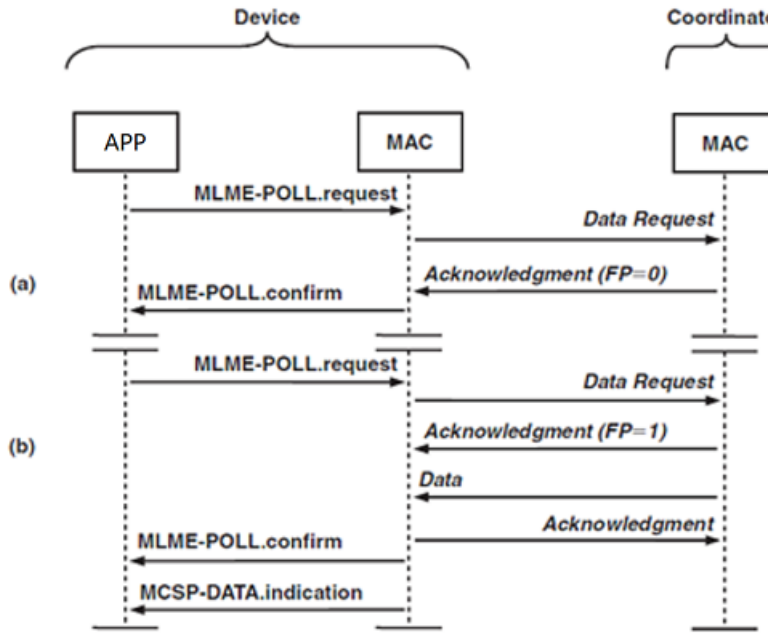

Figure 39: "Data interaction process"

(a) There is No Frame Pending (FP=0)

(b) There is a Frame Pending (FP=1)

After entering the network, for end devices that require low-power consumption functions, the periodic MLME-POLL.request is used to query and read the data on the coordinator side. The code of the application layer is implemented based on this process, such as OTA, end device as a switch to control lights and so on.

54.3 System exception processing

Invoke sys_exceptHandlerRegister() to register the exception callback function, triggered when buffer leak or status exception occurs, user can add the corresponding exception processing in this function.

Suggestion: In the development stage, the callback function directly use while(1) in order to locate the problem, and the variable T_evtExcept[1] can locate what kind of exception has occurred; in the product stage, it is better to do the reset process.

For example:

```
Volatileu16 T_debug_except_code = 0;

static void sampleLightSysException(void){
    T_debug_except_code = T_evtExcept[1];
    while(1)//or SYSTEM_RESET();
}

/* Register except handler for test */
sys_exceptHandlerRegister(sampleLightSysException);
```

6 OTA

The TLSR8 and TLSR9 series chips support flash multi-address booting: in addition to Flash address 0x00000, they also support reading firmware from 0x40000. This features is used by Telink 802154 SDK to implement OTA function.

As you can see from section 2.3, we have allocated two firmware areas Firmware and OTA-Image, and the firmware size should not exceed 208K.

Assuming that Firmware is currently running, when the device performs an OTA upgrade, the new firmware data will be stored in the OTA-Image. After the OTA is completed and verified, it will reboot and run the firmware in the OTA-Image. Subsequent OTAs will be executed alternately.

6.1 OTA query function

6.1.1 ota_queryStart()

Start the OTA query function, called by the end device, to periodically query the coordinator side for firmware updates.

- **Prototype**

```
void ota_queryStart(u16 seconds)
```

- **Return value** None

Name	Type	Description
seconds	u8	Query cycle, in second

6.2 OTA device type

OTA devices include service devices (Server) and end devices (Client). Therefore, it is necessary to pay attention to the type of service to be initialized by OTA during OTA initialization.

Generally, the device being upgraded is the end device (Client); the device providing the new firmware to the upgraded device is the service device (Server), usually the coordinator (PAN coordinator).

6.2.1 OTA Server

The OTA server needs to write the new firmware required by the upgraded device to the OTA-Image area for OTA use.

For example, if the firmware run by the server itself is in the Firmware area, then the OTA image of the target device (new firmware with OTA Header) can be temporarily stored in the OTA-Image area.

6.2.2 OTA Client

The successful network access will call void ota_queryStart(u16 seconds) to set the OTA query period of the OTA end device in seconds, that is, the OTA request will start after the set seconds.

The OTA image on the OTA Client side is generated at the same time as the normal bin, with ota_ prefixed to the normal bin to show the difference, only the Client side will execute tl_ota_tool to generate the OTA file.

As shown in 2.4.2 Flash space allocation chart, the OTA_Image and firmware are alternated, for example, if the current firmware is at address 0x0000, the OTA is stored to 0x40000, if the OTA is successful, the firmware at address 0x0000 will be manually set to fail, Client will start from 0x40000 each time and the next OTA file will be stored to 0x0000. Each time Client starts from 0x40000, the next OTA file is stored at 0x0000, and so on.

6.3 OTA process

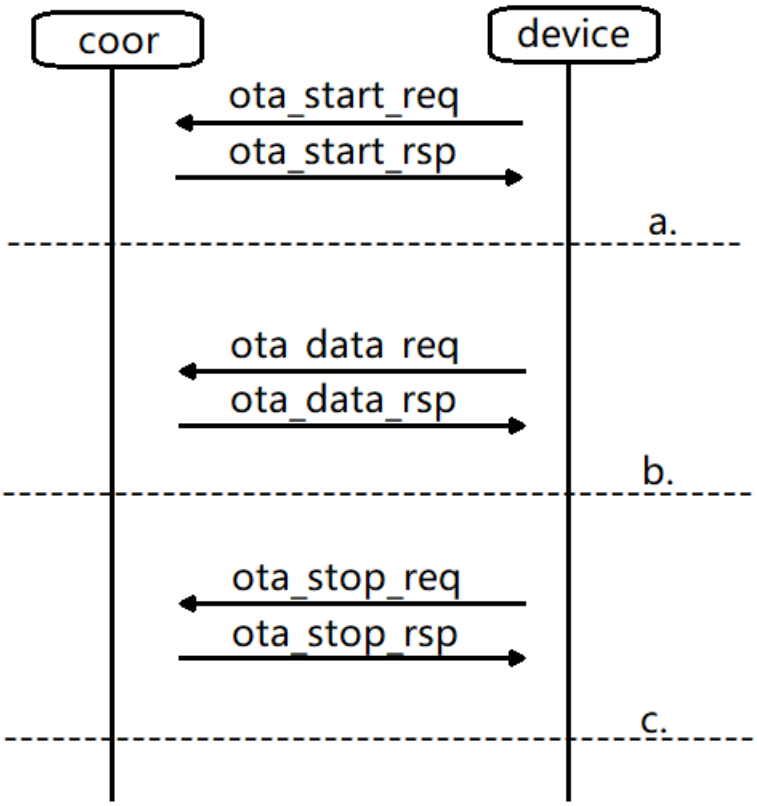

Figure 40: "OTA process"

- **OTA START** The OTA Client side periodically sends the ot_start_req packet to the OTA Server, the specific period is set by the function ota_queryStart() in seconds, the ot_start_req packet contains the current OTA Client side bin file information (file version, manufacturer code, image type); the OTA

Server side receives the ot_start_req and reads the OTA file information at 0x40000, and then sends the OTA file information to the OTA Client via sot_start_rsp packet;

- **OTA DATA** After OTA Client parses the start response from OTA Server, it will get the OTA file information from OTA Server, compare with the current bin of OTA Client, if the MANUFACTURER_CODE and image type are the same, and the OTA file version from OTA Server is larger than the current bin of OTA Client, then OTA Client sends OTA data request to OTA Server. Server side OTA file version is larger than OTA Client current bin file version, then OTA Client sends OTA data request to OTA Server, then Server side reads OTA file and sends it to OTA Client via OTA data response. This process will continue until the OTA file is sent.
- **OTA STOP** When the length of the data obtained by the OTA Client side is greater than or equal to the OTA size, the Client side will send an OTA STOP request to the Server to end the whole OTA process, then the Client side will compare the locally calculated CRC value with the CRC value of the last four bytes of the OTA file, restart if it is the same, otherwise ignore this OTA.