泰凌 802.15.4

SDK 开发指南

AN_19052903-C2

Ver1.1.0

2021.6.9

Keyword

802.15.4

Brief

本文档为泰凌微电子 802.15.4 SDK 的开发指南。

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

修订历史

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

Contents

1 概述

基于 IEEE 802.15.4 (2006)，Telink 802.15.4 SDK 支持标准的 PHY (physical) 层和 MAC (media access control) 层功能。此外，作为参考，SDK 中实现了部分应用层功能。

SDK 整体结构如下：

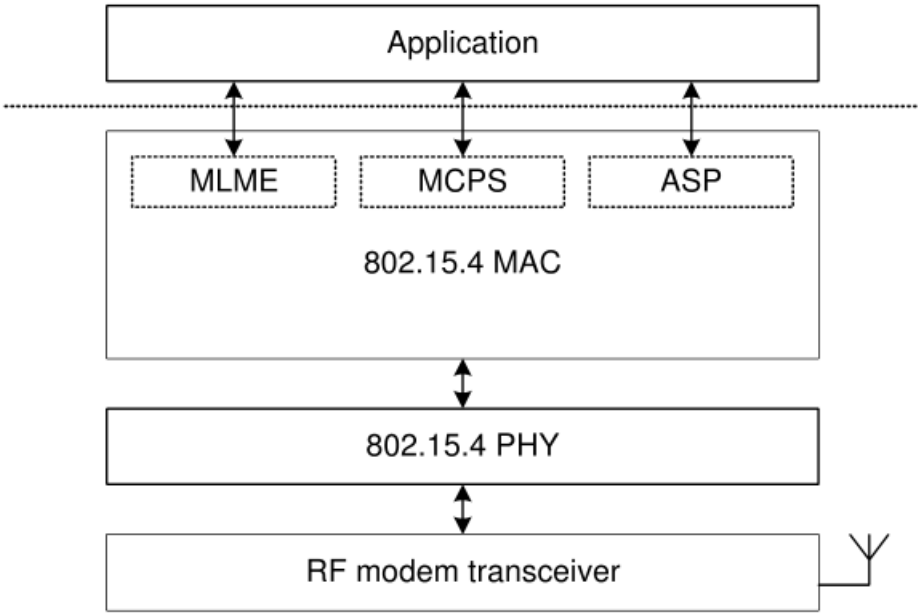

Figure 1: "SDK 系统结构图"

- PHY（physical）层在 telink 硬件平台及驱动的基础上，实现了标准的 2.4G 无线连接，例如调制方式、支持频率、发射功率等，具体可以参考相应芯片的数据手册；
- MAC（media access control）层由软件实现，基于 IEEE 802.15.4 specification，实现标准的数据格式和加密机制等。
- APP（application）层，SDK 提供了部分应用示例，例如加入 PAN、OTA 等。

为了和 15.4 协议栈交互，SDK 提供了三种接口 (interface) 给 APP 层：MLME、MCPS、ASP。

a) MAC 子层包含管理实体 (MAC Layer Management Entity，MLME)：应用层可以通过 MLME，发送 802.15.4 MAC COMMAND。例如，应用层通过此接口 (interface) 可以发送 MLME-ASSOCIATE.request 原语 (primitive)，同时也通过这个接口 (interface) 收到 MLME-ASSOCIATE.confirm 原语 (primitive)。

b) MCPS 接口 (interface)：应用层可以通过 MCPS，发送和接收 802.15.4 MAC DATA。

c) ASP 接口 (interface)：应用层可以通过 ASP 设置 MAC 层 PIB 参数。

以上接口 (interface) 的使用，后续章节会有详细的介绍。

1.1 设备类型

从包含的功能角度，15.4 包含以下两种设备类型：

- **全功能设备（Full Function Device，FFD）：**具有 15.4 的所有功能设备，可以是协调器 (PAN coordinator OR coordinator)，路由设备 (router)，或者具有全功能的终端节点 (end device)，全功能设备（FFD）一般不进入休眠。
- **非全功能设备（Reduced Function Device，RFD）：**只能作为终端节点，一般情况下有低功耗要求。

1.2 网络类型

802.15.4 的网络属于集中式 (Central) 网络，也叫星形网络。

- **集中式（Central）网络：**通常是由 PAN Coordinator 来组建和管理的网络。

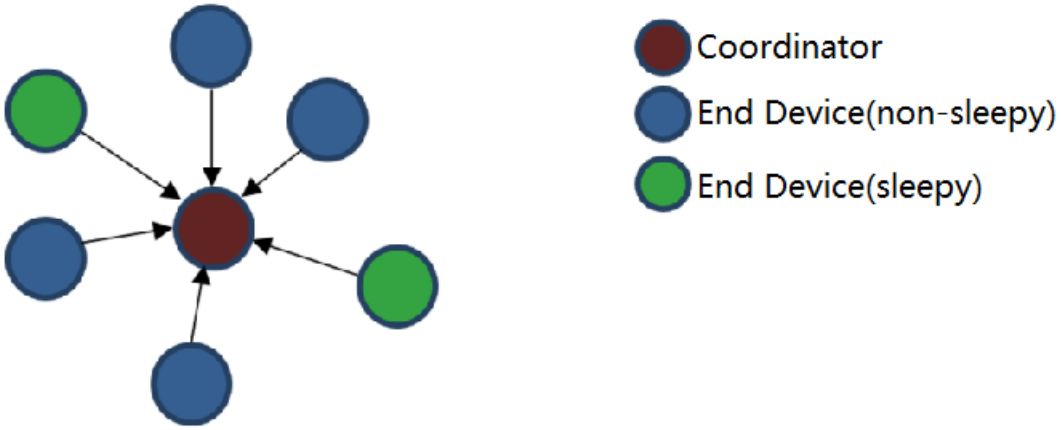

Figure 2: "集中式网络"

1.3 基本概念

1) **PAN ID**: 个人局域网（Personal Area Network）ID

由 PAN ID 来标识所组建的网络。PAN ID 由负责组建该网络的节点分配，可以直接指定，也可以随机生成，但需要通过 active scan 来避免 PAN ID 冲突。

2) **Channel**: 频点/信道

15.4 允许的工作频点 (2.4G)：2405 + (N-11)*5 (mHz), (信道 N = 11~26)。接入一个网络后采用单一频点的工作模式，不会主动跳频。

3) **Direct Data Transfer**: 直接数据发送

一般用于向 FFD 设备发送数据，例如 end device 发数据给 coordinator。

4) **Indirect Data Transfer**: 间接数据发送

一般用于 FFD 设备发送数据给进入低功耗的 RFD，例如 coordinator 发数据给 end device，coordinator 会先将数据存储在本地，end device 通过 poll request 的方式，获得数据包。

14 Telink 802.154 SDK 开发概述

在 Telink 提供的软硬件基础之上，实现 Telink 802.15.4 SDK 开发。

14.1 Telink 802.154 SDK 软件开发环境

1) **必备软件工具** (下载地址：http://wiki.telink-semi.cn/)

- 集成开发环境：

  TLSR8 Chips：Telink IDE for TC32

  TLSR9 Chips：Telink RDS IDE for RISC-V

- 下载调试工具：Telink Burning and Debugging Tools
- OTA 编码转换工具：tl_ota_tool，具体过程参考 OTA 章节 6.2.2

2) **抓包分析辅助工具** (自行下载或购买)

- TI Packet Sniffer
- Ubiqua

3) **软件开发包** (下载地址：http://wiki.telink-semi.cn/)

- TLSR8_802_15_4_SDK.zip

14.2 Telink 802.154 SDK 支持的硬件平台

- B85m (TC32 平台)

  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 平台)

  9518：9518 EVK Board and 9518 USB Dongle

2 泰凌 802.154 SDK

在安装 SDK 前，请根据 1.4 章节安装相应的集成开发环境 Telink IDE for TC32 或 Telink RDS IDE for RISC-V。

2.1 TLSR8 TC32 SDK 安装

2.1.1 项目导入

1) 打开 IDE，依次进入界面 File -> Import -> Existing Projects into Workspace。

2) 选择 tl_802154_sdk/build 目录点击"确定"。

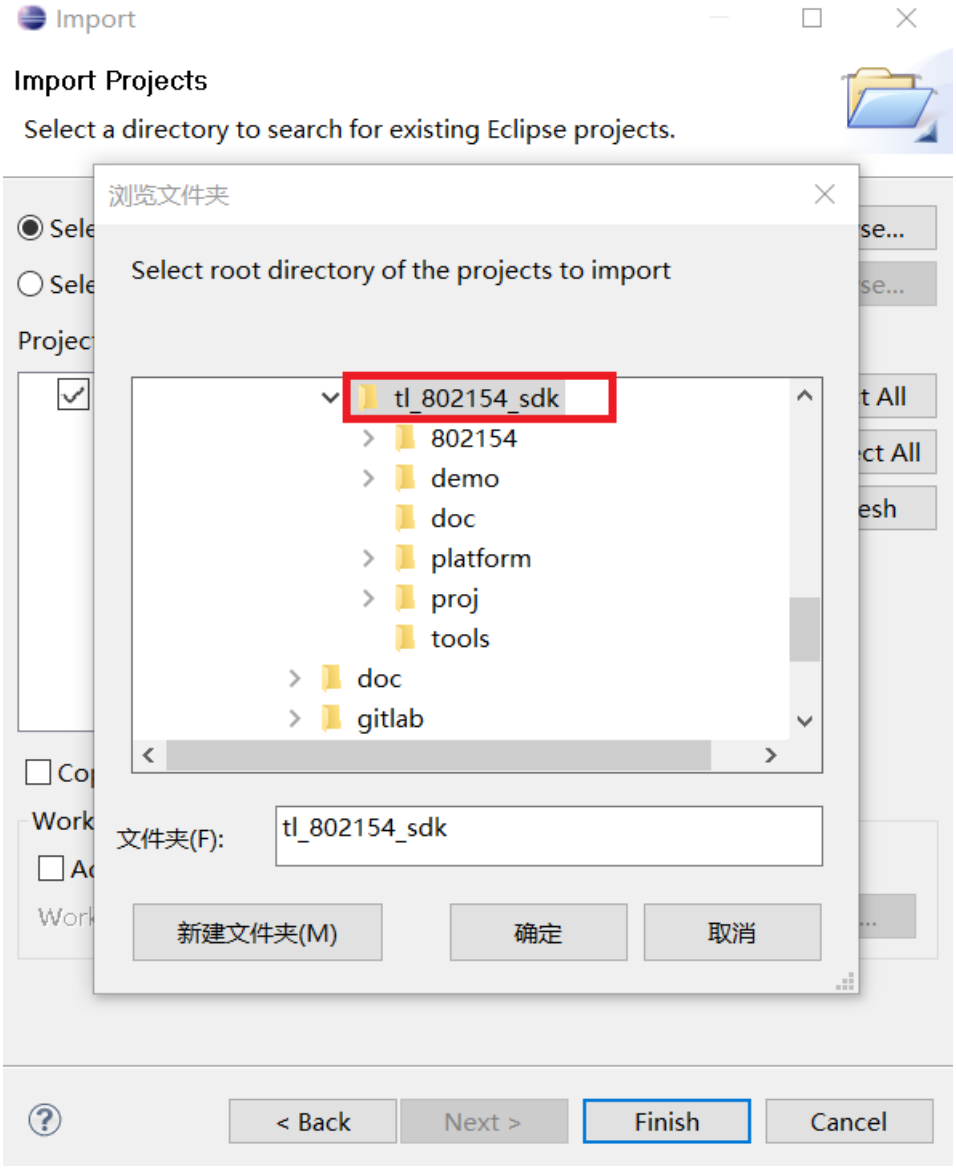

Figure 3: "选择导入工程文件"

3) 点击"Finish"，完成工程导入。

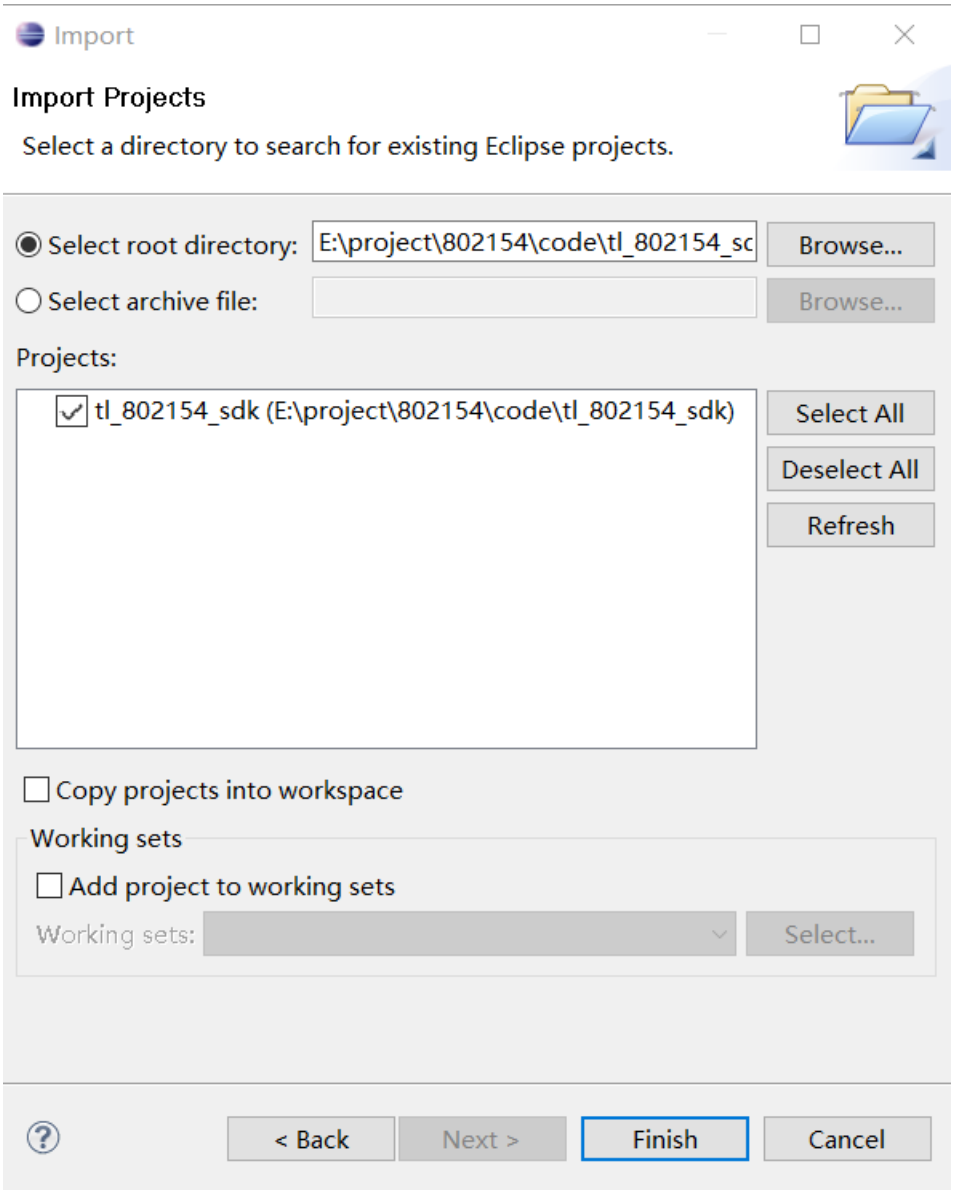

Figure 4: "完成工程导入"

Copy project into workspace 框根据实际情况决定是否勾选，如工程已经在此目录下不必勾选，如在其他目录下，希望将工程在当前目录编辑开发，则可以勾选。

2.1.2 工程目录结构

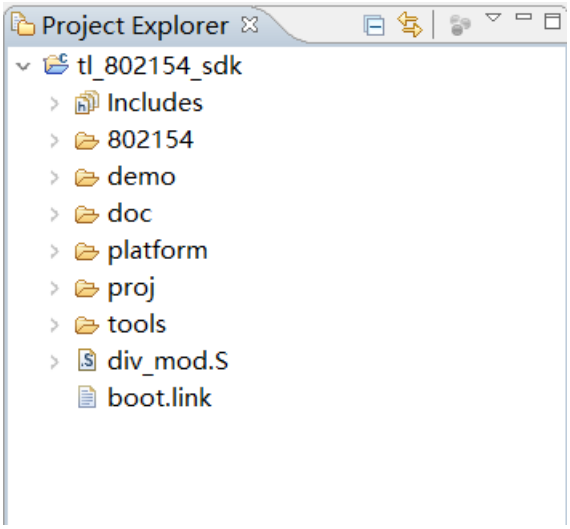

Figure 5: "工程目录结构"

- **/demo:** 用户工程目录。
  - /app_common: 应用层公共代码目录，包含包含应用数据包处理函数 (tl_specific_data.c tl_specific_data.h)、以及在此处理函数中的具体应用 OTA，其他的应用也可在此基础上完善，或者导入自己的应用层代码。
  - /associate_coor: 协调器 (PAN Coordinator) 例程。
  - /associate_dev: 终端节点（end device）例程。
- **/platform:** 运行平台目录。
  - /boot: 启动和链接文件。
  - /chip_xx: 芯片驱动文件。
  - /services: 中断服务函数文件。
- **/proj**: 工程代码目录。
  - /common: 通用代码目录。
  - /drivers: 抽象层驱动文件。
  - /os: 任务事件、buffer 管理函数文件。
- **/802154:** 协议栈相关目录。
- **/tools:** hci 命令处理相关目录。
- **div_mod.S:** 除法和取余相关的汇编函数。（TLSR8 不支持硬件除法器）

2.1.3 编译选项

点击下拉图标  ，可以看到当前所有的编译选项，选择需要编译的示例工程，如下图，等待编译完成。

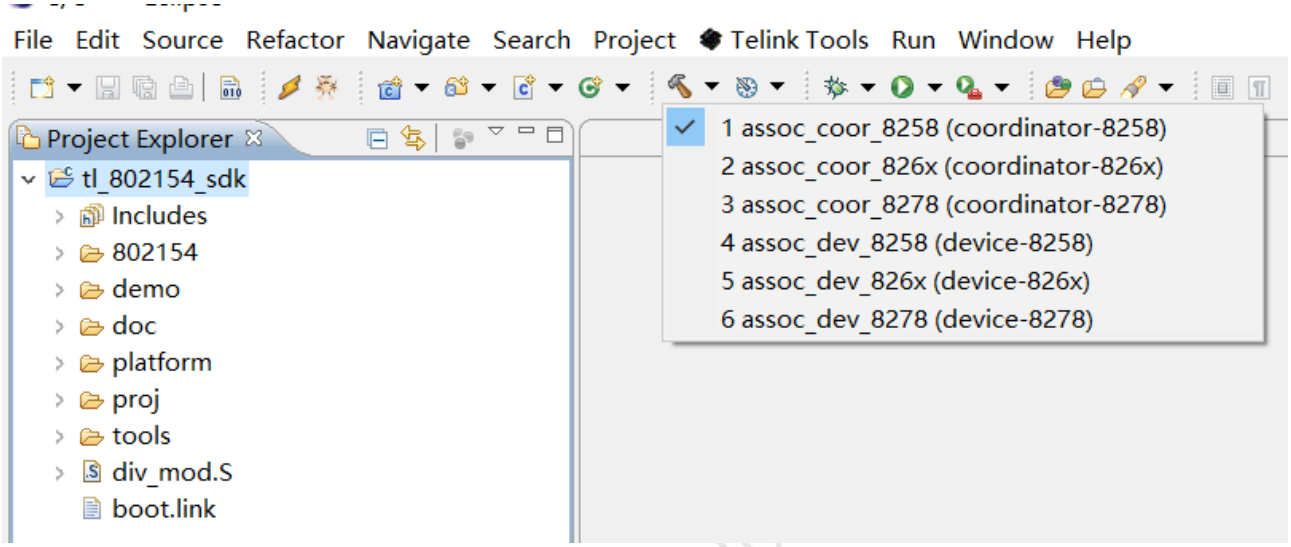

Figure 6: "选择编译"

编译完成后，"Project Explorer"窗口会出现编译出来的文件夹，其中包含了编译出来的固件，如下图。

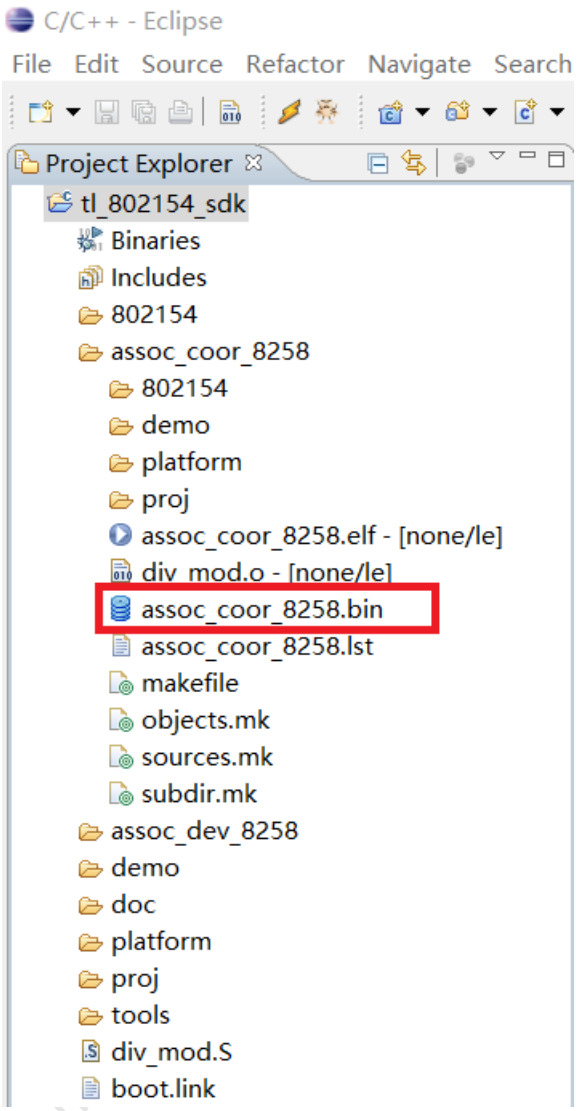

Figure 7: "assoc_coor_82xx 输出文件"

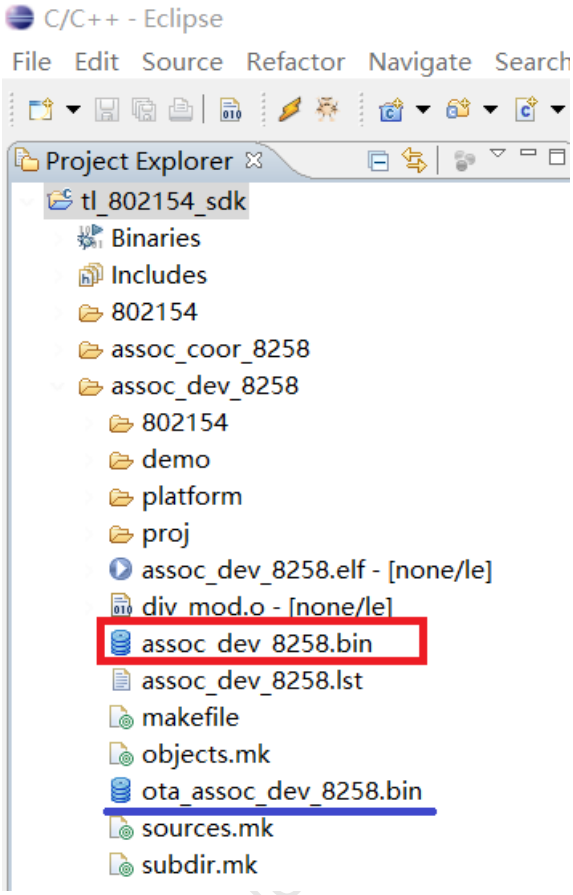

Figure 8: "assoc_dev_82xx 输出文件"

assoc_dev_82xx 工程会产生两个 bin 文件，红色框的 assoc_dev_82xx.bin 为运行 bin，用于直接烧录到目标板；蓝色下划线的 ota_assoc_dev_82xx.bin 为 OTA 文件。

2.14 添加新项目

在提供的 SDK 中，仅仅给出了一些简单的用例。用户可以根据需求，自行添加应用工程以及编译选项。

具体步骤如下：

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

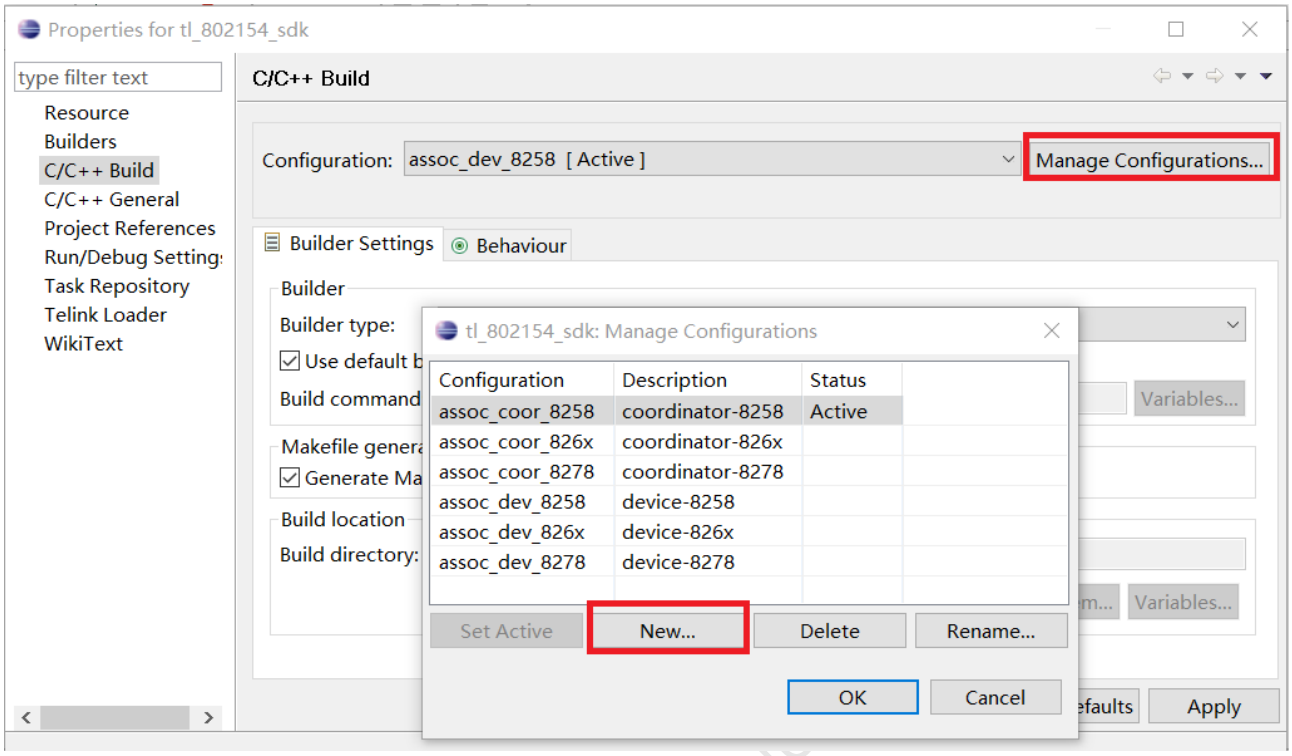

Figure 9: "添加新项目"

2) 步骤 2：单击 New，弹出如下界面

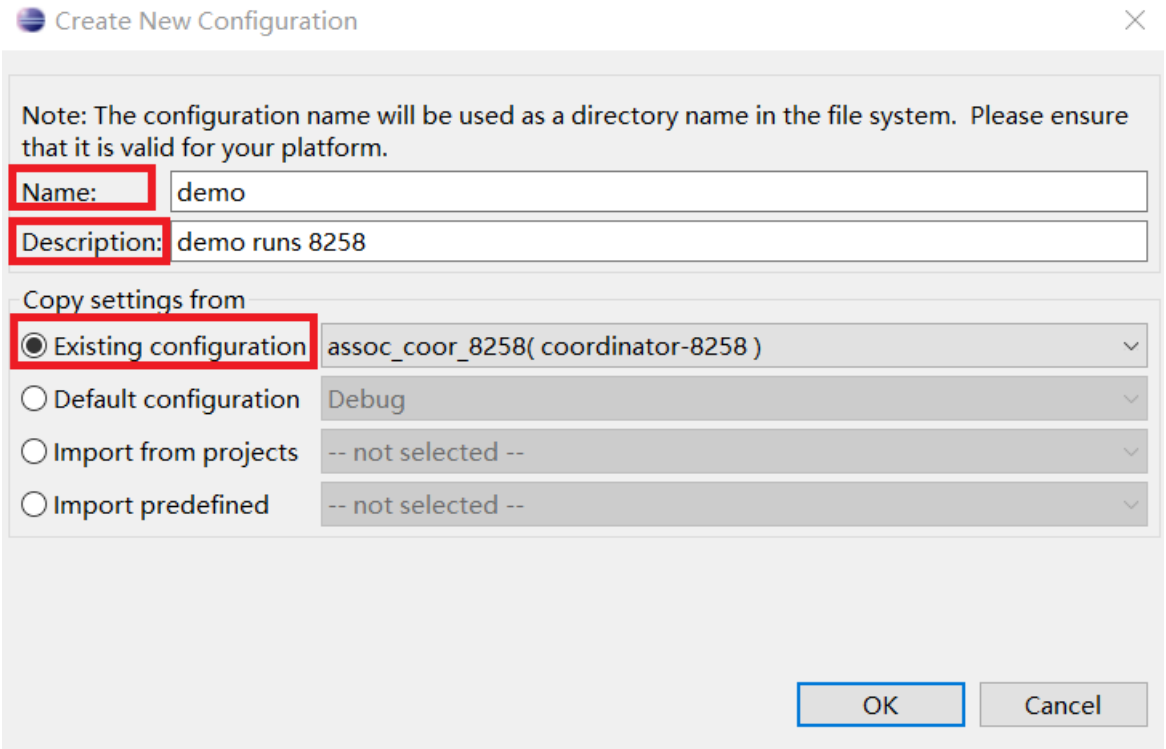

Figure 10: "配置新项目"

- Name：项目名称
- Description：项目简单描述
- Copy settings from：建议选用 Existing configuration，可以减少一些配置过程。

选用 Existing configuration 的原则如下所示：

- 使用 8278 平台，选择 associate_xxx_8278；使用 8258 平台，选择 associate_xxx_8258。
- 项目用于 coordinator 的开发，选择 associate_coor_8258；项目用于 end device 设备的开发，选择 associate_dev_8258。

假设需要用 8258 开发一个具有休眠功能的传感器的项目，根据该原则，应选择项目" associate_dev_8258"的配置作为这个新项目的配置。

如果需要修改配置，可按下节（2.1.5 项目配置说明），进行相应调整。

2.1.5 项目配置说明

依次进入：Project Explorer -> tl_802154_sdk -> Properties -> Settings (以项目 associate_dev_8258 为例说明)

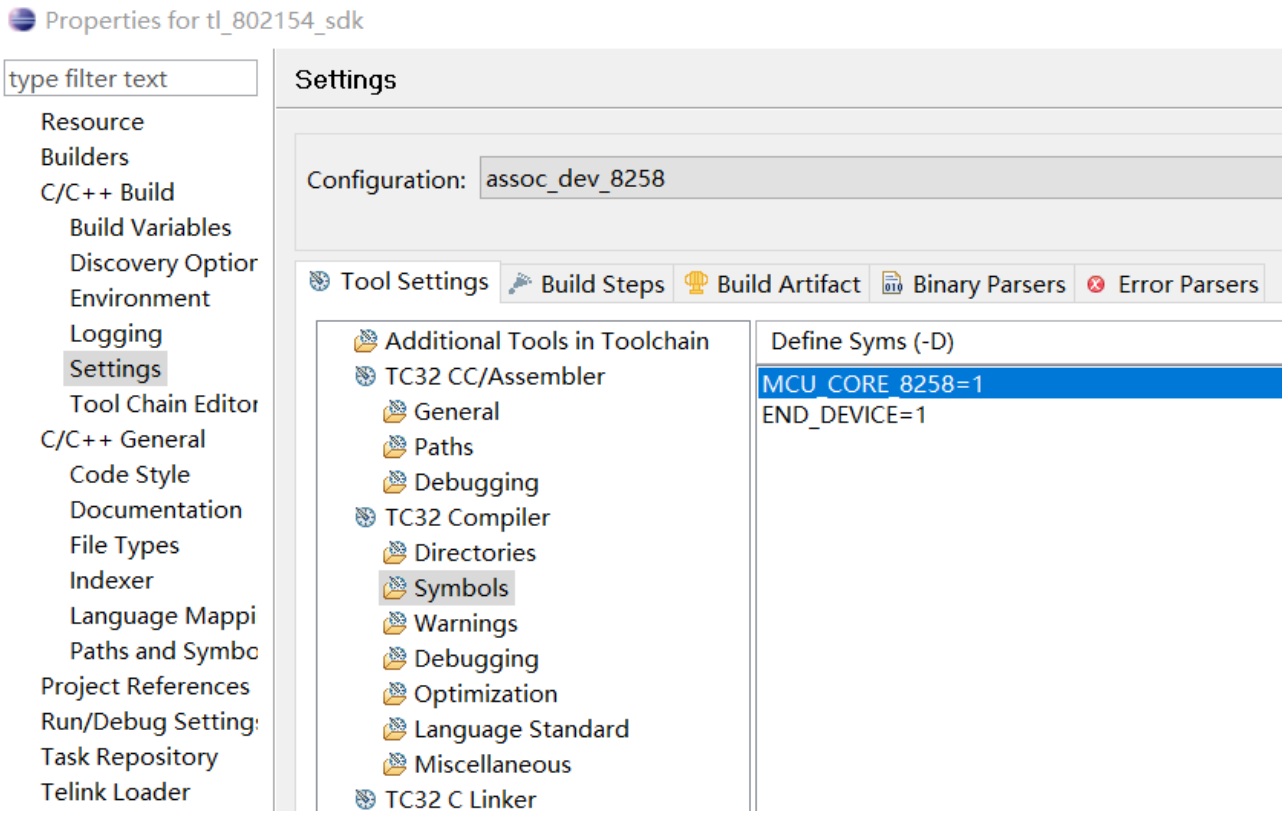

Figure 11: "符号定义"

在 Tool Settings 下，可以看到当前项目的一些预定义配置：

1) **设备类型预定义**

-END_DEVICE=1：表示项目是个终端节点 (end device) 设备

对于 coordinator 和 end device 的设备设置分别为：

-COORDINATOR=1：表示项目是个 Coordinator 功能的设备

-END_DEVICE=1：表示项目是个 end device 功能的设备

2) **平台选择**

- 8269 平台：-DMCU_CORE_826x=1 以及-DCHIP_8269=1

  启动代码 cstartup_826x.S 位于 tl_802154_sdk -> platform -> boot 目录下。

- 8258 平台：-DMCU_CORE_8258=1

  启动代码 cstartup_8258.S 位于 tl_802154_sdk -> platform -> boot 目录下。

- 8278 平台：-DMCU_CORE_8278=1

  启动代码 cstartup_8278.S 位于 tl_802154_sdk -> platform -> boot 目录下。

3) **lib 文件的链接**

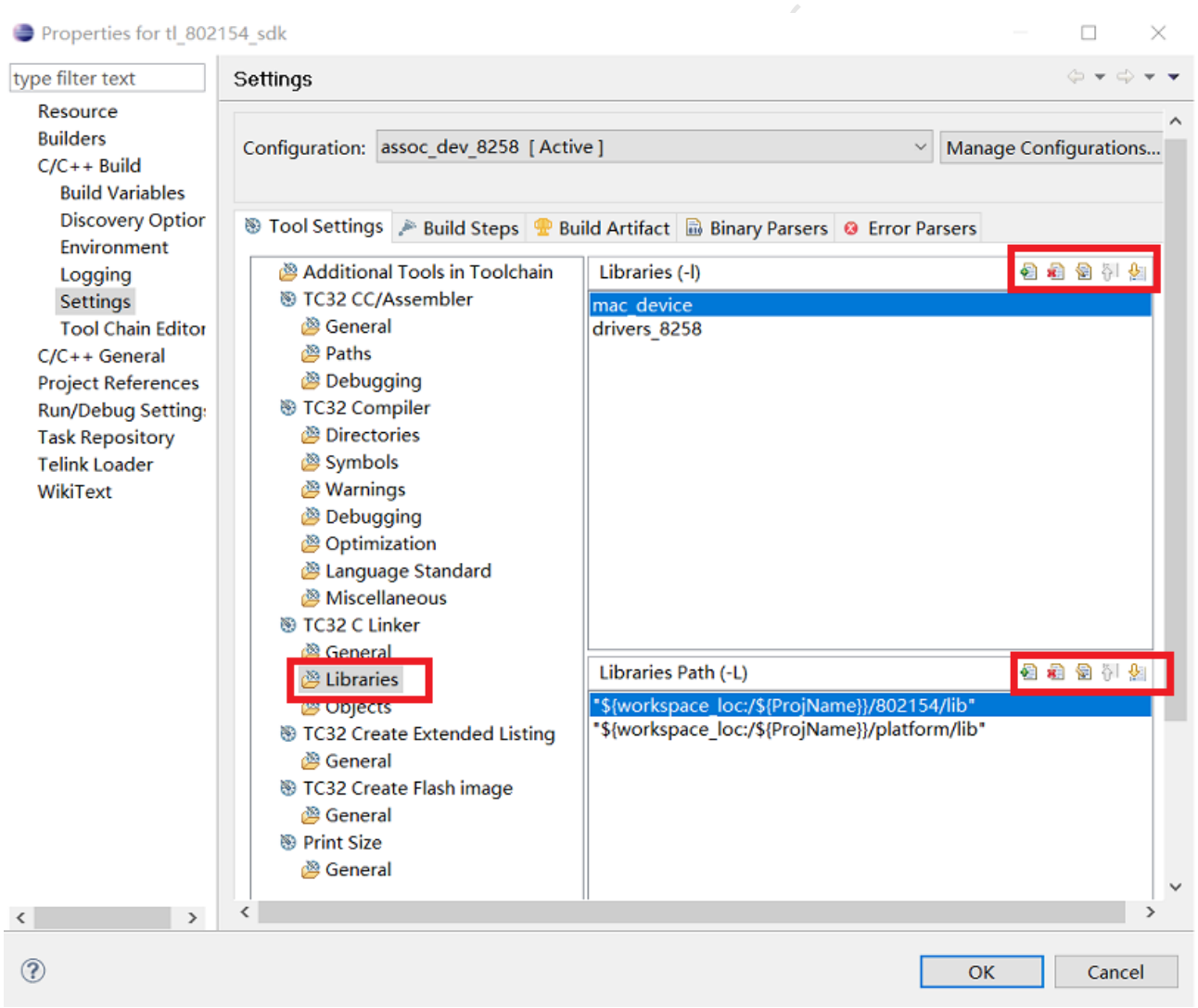

Figure 12: "库文件和路径"

当前 SDK 里包括 2 类库文件：15.4 stack 库和平台驱动库。

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

  位于 tl_802154_sdk -> 802154 -> lib 目录下。

- 平台驱动库：libdrivers_826x.a，libdrivers_8258.a，libdrivers_8278.a

  位于 tl_802154_sdk -> platform -> lib 目录下。

4) **链接文件**

用户可以依据实际应用需求，根据所选用的不同平台以及内存要求，调整合适的 link 文件。

当前 SDK 给出了 826x、8258 和 8278 平台的默认 link 文件 boot_826x.link、boot_8258.link 和 boot_8278.link，分别在 tl_802154_sdk -> platform -> boot 对应的文件。

如下图所示，通过预编译调用脚本 tl_link_load.sh（tl_802154_sdk -> tools 目录下）来选择使用的 link 文件。

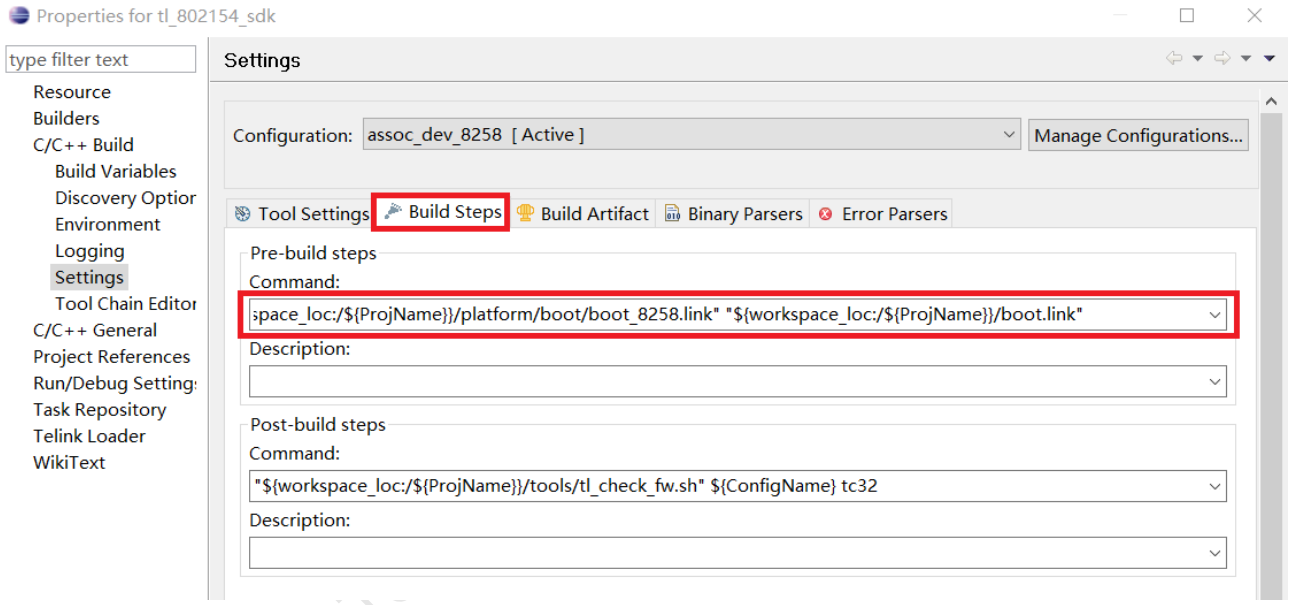

Figure 13: "link 文件预编译设置"

5) **.image 文件校验**

为了保证下载文件的可靠性，通过脚本 tl_check_fw.sh（tl_802154_sdk -> tools 目录下）在生成的 image 文件中加入校验字段以及非协调器设备执行 ota_bin_tool，OTA 过程中会通过检查校验字段是否匹配，来决定是否更新 image。

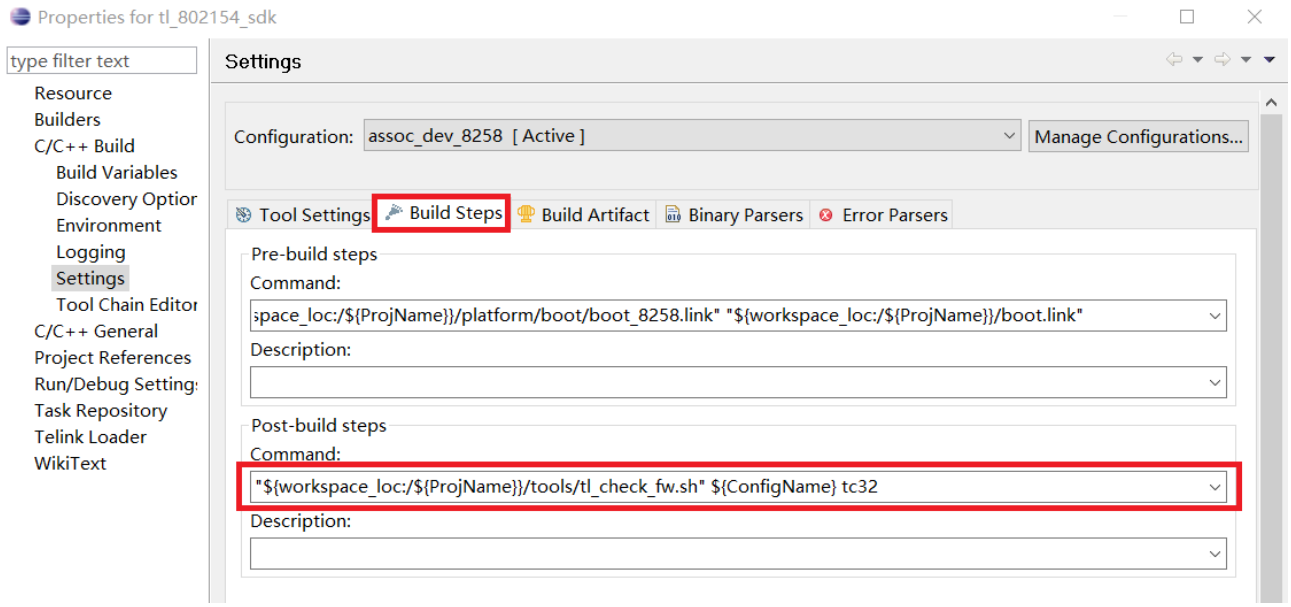

Figure 14: “image 校验设置”

2.2 TLSR9 RISC-V SDK **安装**

2.2.1 **项目导入**

1) 打开 IDE，依次进入界面 File -> Import -> Existing Projects into Workspace。

2) 选择 tl_802154_sdk -> build 目录下的 tlsr_riscv 点击“确定”。

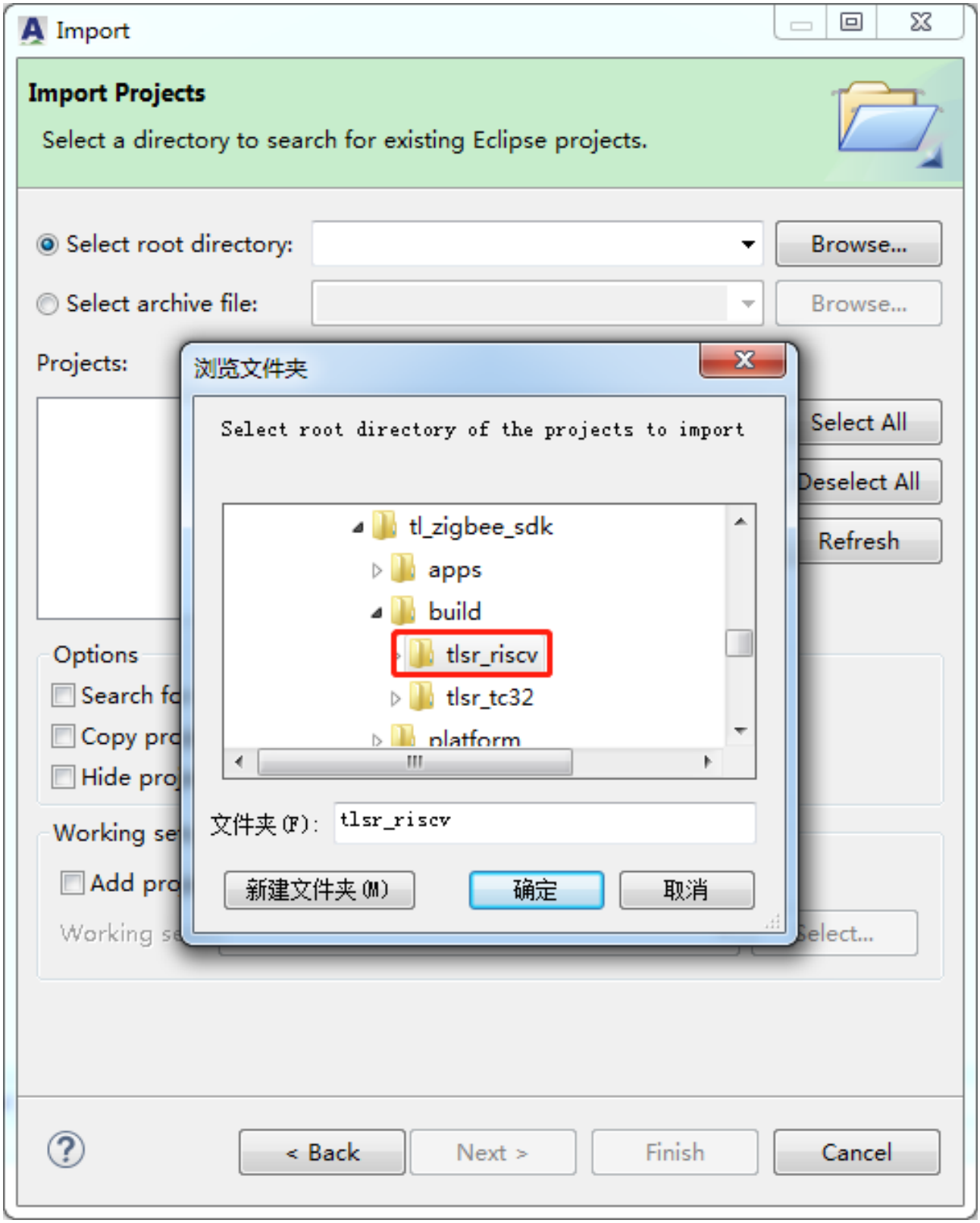

Figure 15: "选择导入工程文件"

3) 点击"Finish"，完成工程导入。

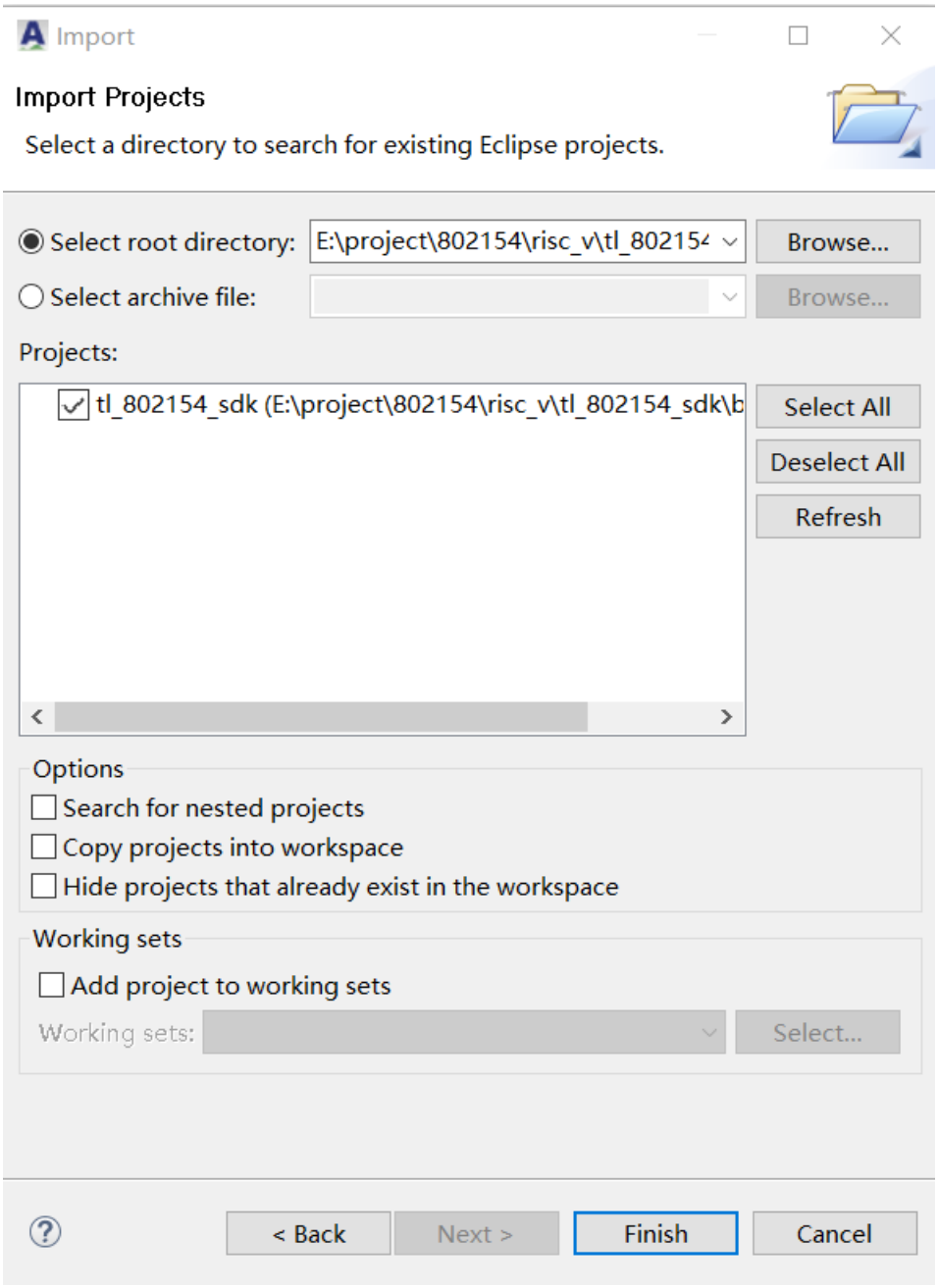

Figure 16: “完成工程导入”

2.2.2 工程目录结构

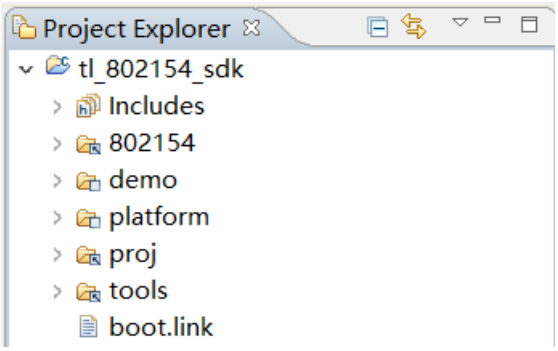

Figure 17: "工程目录结构"

- **/demo**: 用户工程目录。
  - / app_common: 应用层公共代码目录，包含应用数据包处理函数 (tl_specific_data.c tl_specific_data.h)、以及在此处理函数中的具体应用 OTA，其他的应用也可在此基础上完善，或者导入自己的应用层代码。
  - /associate_coor: 协调器 (PAN Coordinator) 例程。
  - /associate_dev: 终端节点（end device）例程。
- **/platform**: 运行平台目录。
  - /chip_xx: 芯片驱动文件。
  - /services: 中断服务函数文件。
- **/proj**: 工程代码目录。
  - /common: 通用代码目录。
  - /drivers: 抽象层驱动文件。
  - /os: 任务事件、buffer 管理函数文件。
- **/802154**: 协议栈相关目录。

2.2.3 编译选项

点击下拉图标 "  "，可以看到当前所有的编译选项，选择需要编译的示例工程，如下图，等待编译完成。

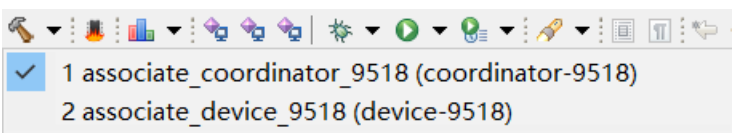

Figure 18: "选择编译"

编译完成后，在"Project Explorer"窗口会出现编译出来的文件夹，其中包含了编译出来的固件，如下图。

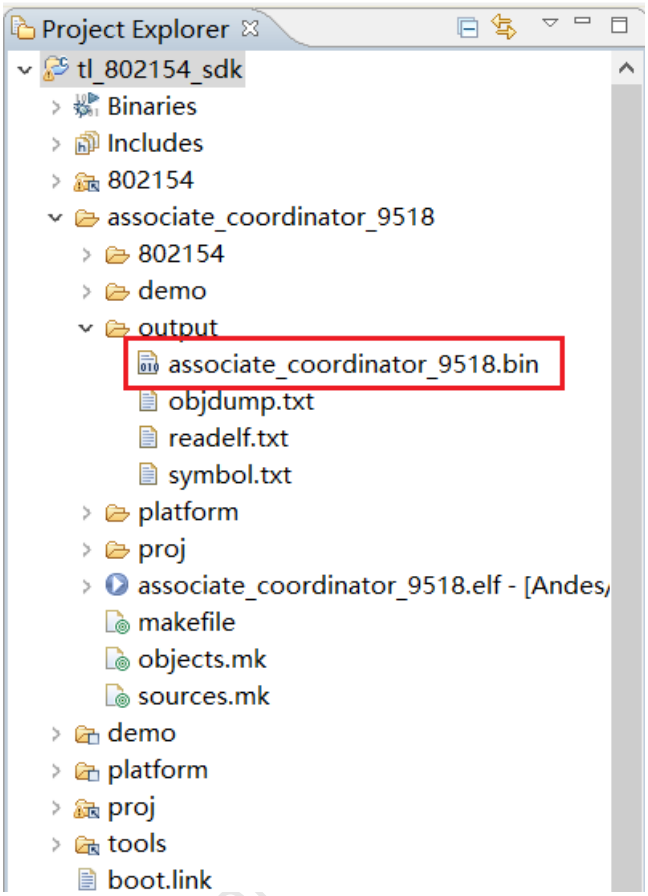

Figure 19: "输出文件"

2.24 添加新项目

在提供的 SDK 中，仅仅给出了一些简单的用例。用户可以根据需求，自行添加应用工程以及编译选项。

具体步骤如下：

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

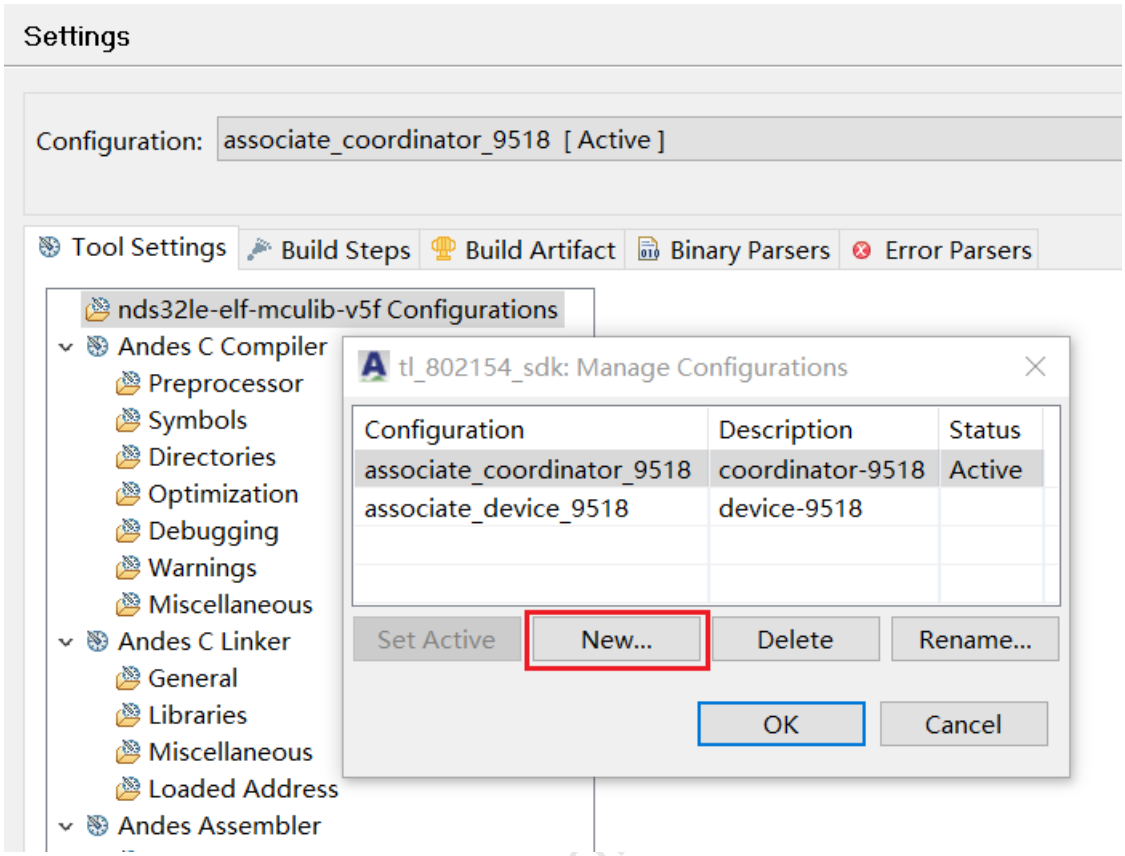

Figure 20: "添加新项目"

2) 步骤 2：单击 New，弹出如下界面

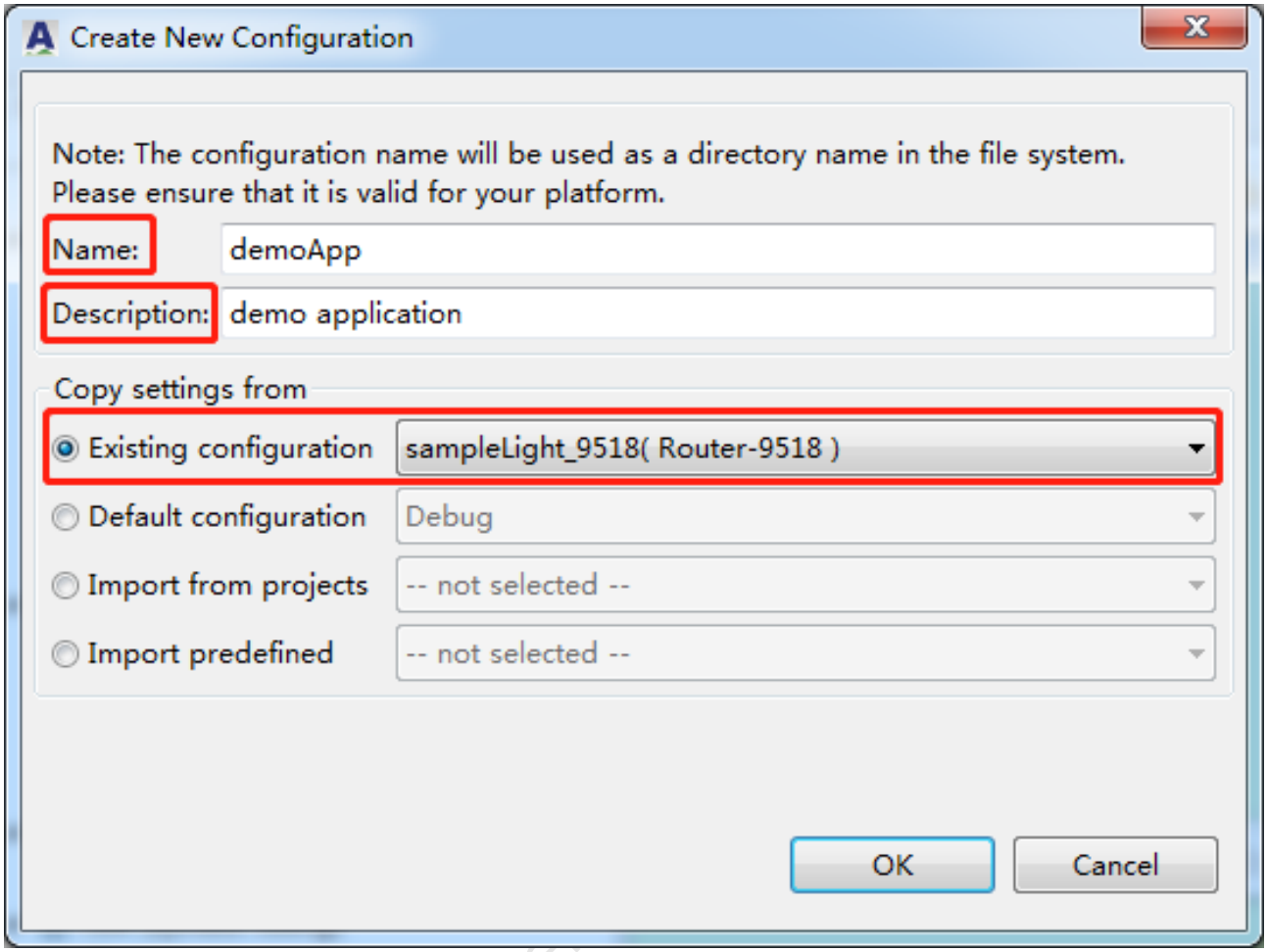

Figure 21: "配置新项目"

- Name：项目名称
- Description：项目简单描述
- Copy settings from：建议选用 Existing configuration，可以减少一些配置过程。

选用 Existing configuration 的原则如下所示：

- 使用 9518 芯片，选择 xxx_9518。
- 项目用于 Gateway 的开发，选择 sampleGw_xxxx；项目用于 Router 设备的开发，选择 sampleLight_xxxx；项目用于 End Device 设备的开发，选择 sampleSwitch_xxxx。

假设需要用 9518 开发一个具有路由功能的灯的项目，根据该原则，应选择项目"sampleLight_9518"的配置作为这个新项目的配置。

如果需要修改配置，可按下节（2.2.5 项目配置说明），进行相应调整。

2.2.5 项目配置说明

依次进入：Project Explorer -> tl_802154_sdk -> Properties -> Settings（以项目 associate_coordinator_9518 为例说明）

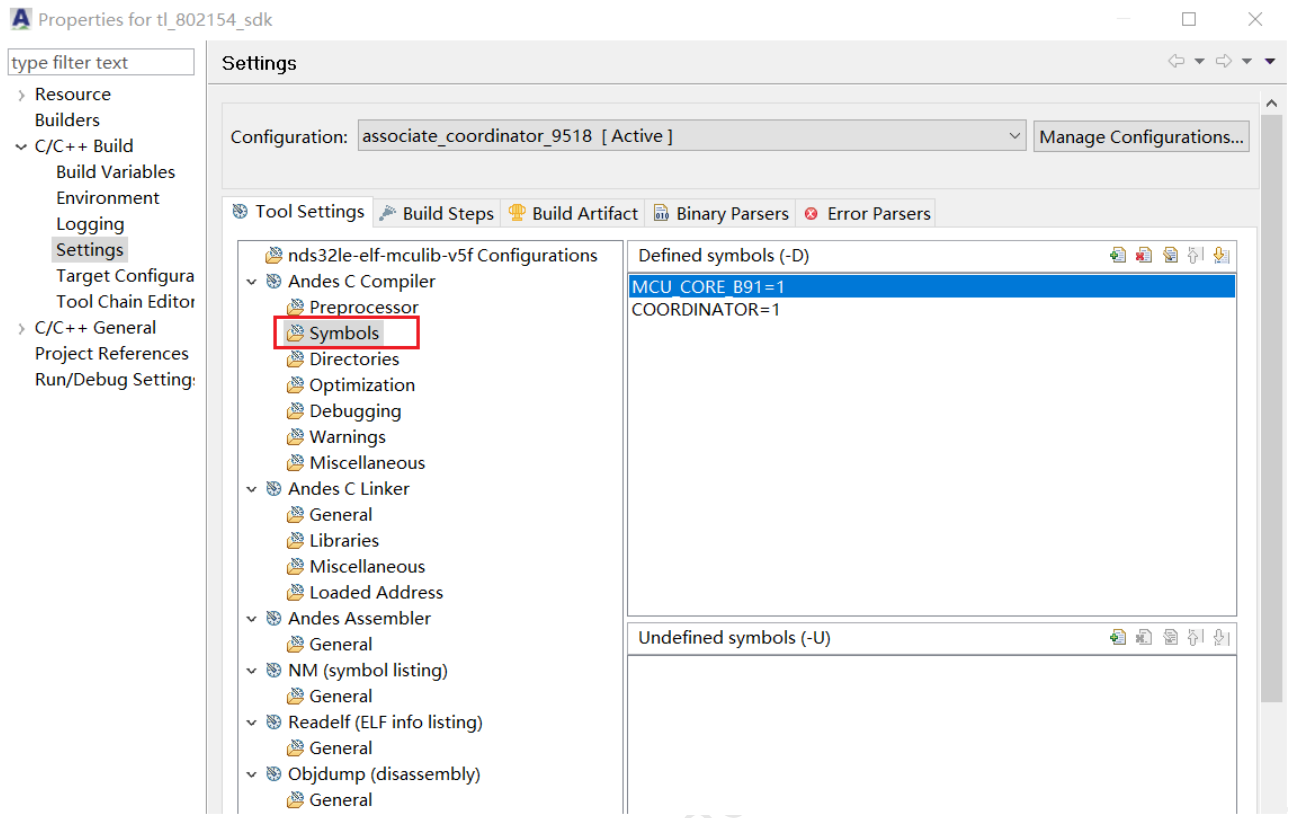

Figure 22: "符号定义"

在 Tool Settings 下，可以看到当前项目的一些预定义配置：

1) **设备类型预定义**

对于 coordinator 和 end device 的设备设置分别为：

-DEND_DEVICE=1：表示项目是个 end device 功能的设备

-DCOORDINATOR=1：表示项目是个 Coordinator 功能的设备

2) **平台选择**

- b91 平台：-DMCU_CORE_B91=1

  启动代码 cstartup_b91.S 位于 tl_802154_sdk -> platform -> boot -> b91 目录下。

3) **lib 文件的链接**

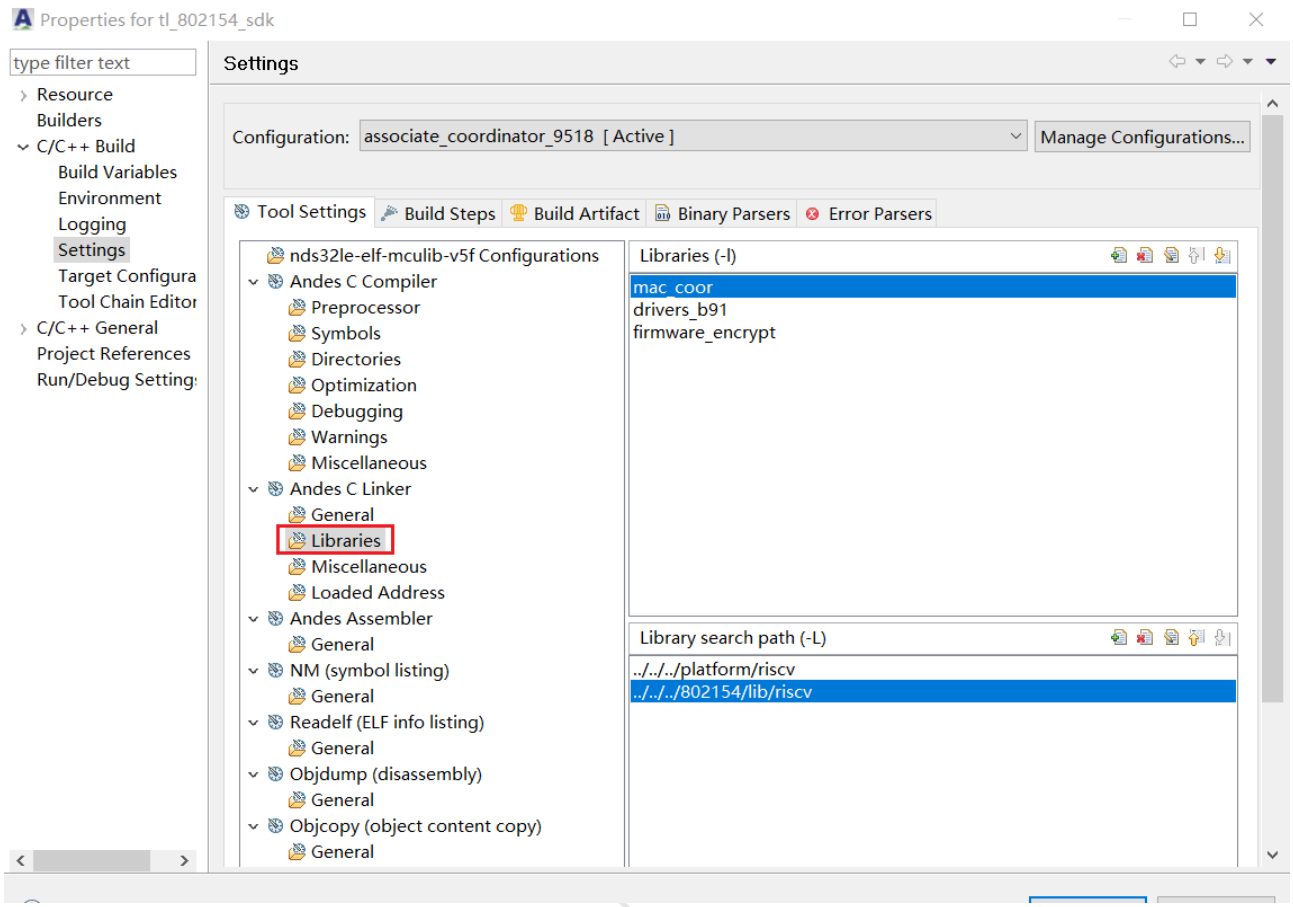

Figure 23: "库文件和路径"

当前 SDK 里包括 2 类库文件：802154 stack 库和平台驱动库。

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

  位于 tl_802154_sdk -> 802154 -> lib -> riscv 目录下。

- 平台驱动库：libdrivers_b91.a

  位于 tl_802154_sdk -> platform -> lib 目录下。

4) **链接文件**

用户可以依据实际应用需求，根据所选用的不同平台以及内存要求，调整合适的 link 文件。

当前 SDK 给出了 b91 平台的默认 link 文件 boot_b91.link，在 tl_802154_sdk -> platform -> boot 对应的目录下。

如下图所示，通过预编译调用脚本 tl_link_load.sh（tl_802154_sdk -> tools 目录下）来选择使用的 link 文件。

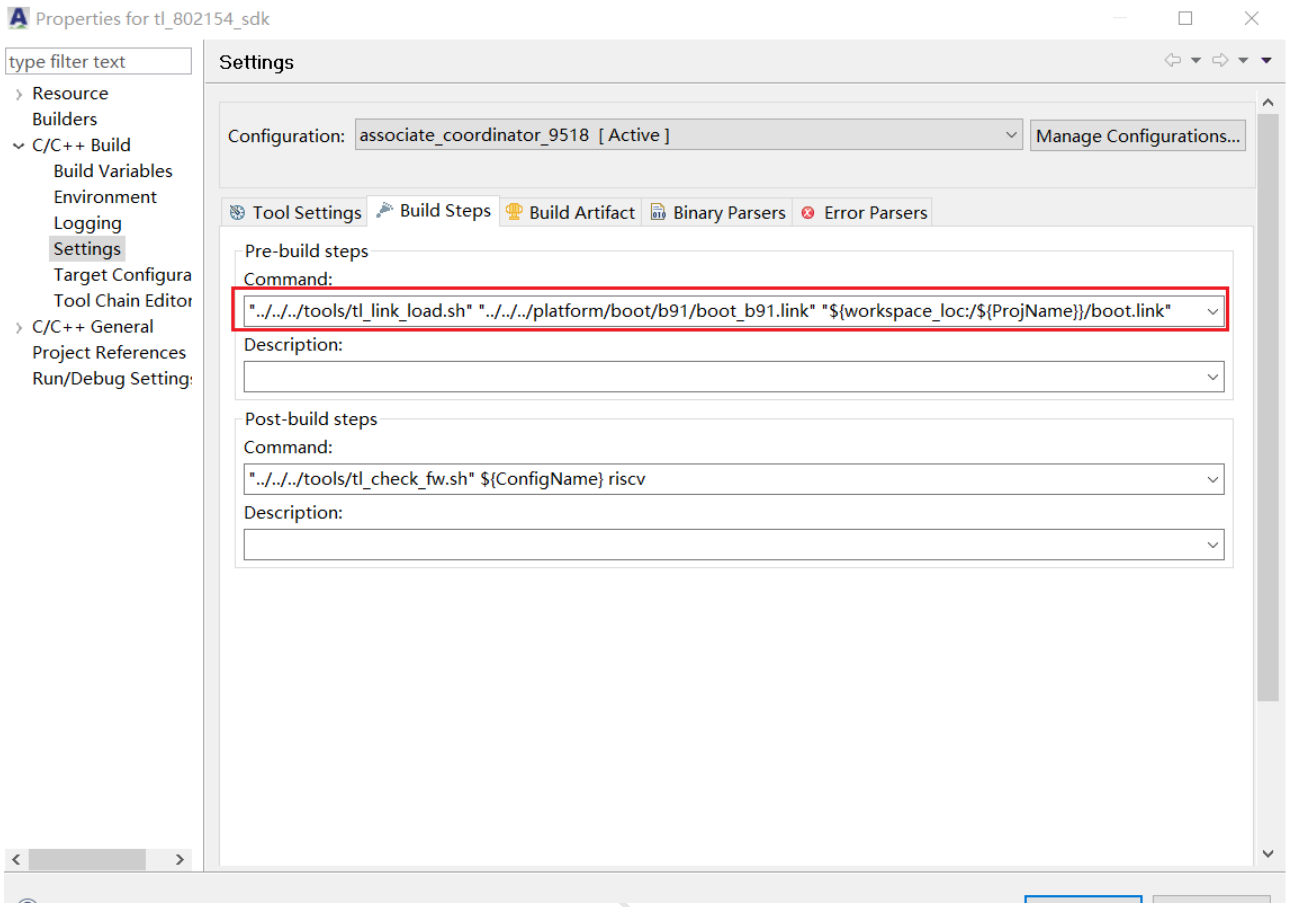

Figure 24: "link 文件预编译设置"

5) **.image 文件校验**

为了保证下载文件的可靠性，通过脚本 tl_check_fw.sh（tl_802154_sdk -> tools 目录下）在生成的 image 文件中加入校验字段，下载或 OTA 过程中会通过检查校验字段是否匹配，来决定是否更新 image。

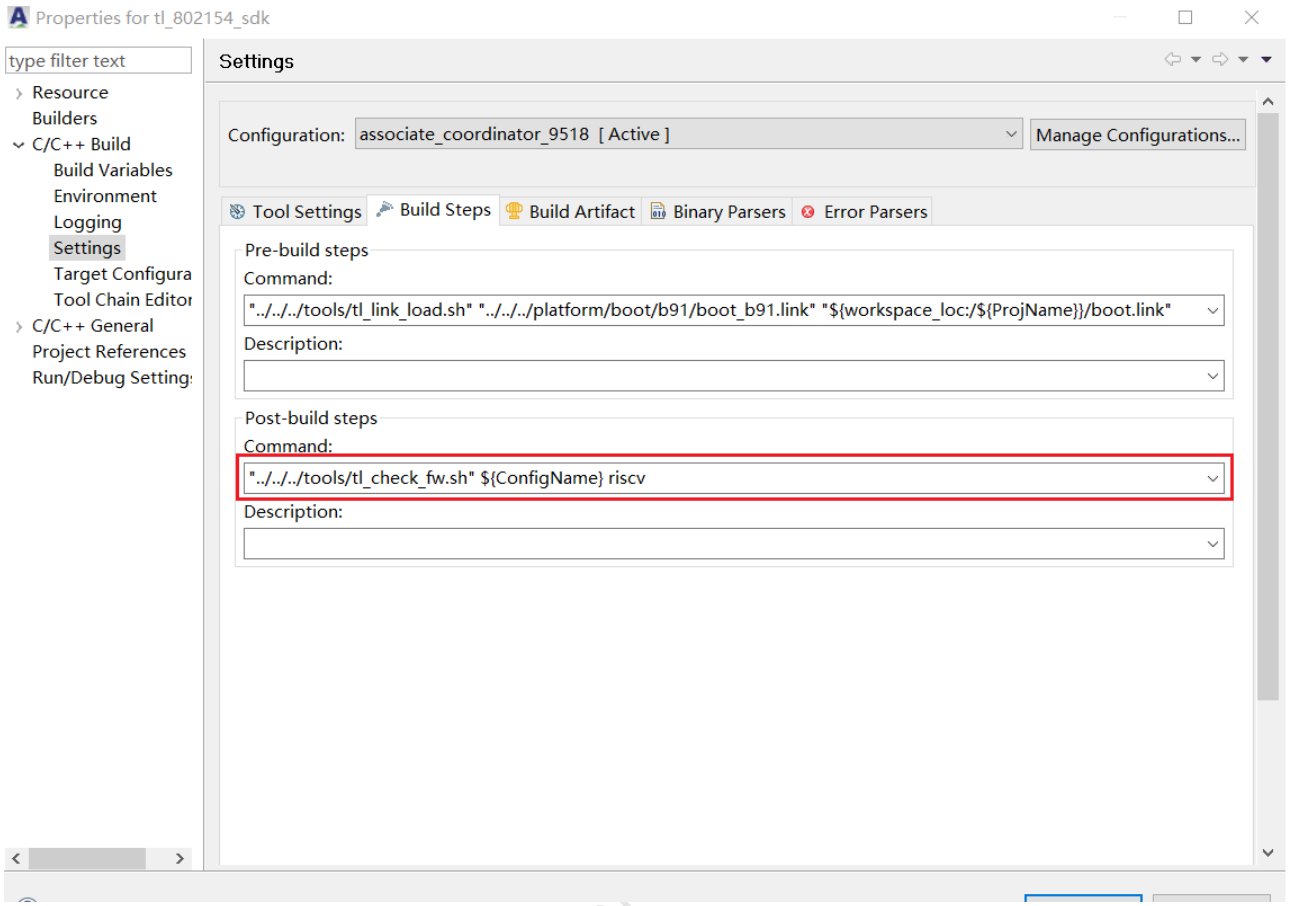

Figure 25: "image 校验设置"

2.3 App **版本管理**

每个 demo 目录下都对应有一个 version_cfg.h 文件用来管理 app 的版本，对 App 进行版本管理是非常有必要的，尤其是在 OTA 时可以有效的防止错误升级而导致变砖的风险。

App 版本是由三个关键字段组成的，分别是制造商代码（Manufacturer Code）、固件类型（Image Type）和文件版本（File Version），会以小端形式存放在固件的固定位置，如下图：

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: "image 二进制文件"

绿色框：文件版本 (file version)

红色框：制造商代码 (MANUFACTURER_CODE)

蓝色框：固件类型 (image type)

2.3.1 制造商代码

MANUFACTURER_CODE：制造商代码是 Zigbee 联盟为每个成员公司分配的一个 2 字节长度的标识符，比如 0x1141 是 Telink 的制造商代码。用户可以将其修改成自己的制造商代码。

2.3.2 固件类型

IMAGE_TYPE：固件类型是一个 2 字节长度的常量，高字节代表 Chip 类型，低字节代表 Image 类型。Chip 类型和 Image 类型定义在 version_comm.h 文件中，用户可以根据需求自行修改或添加。

2.3.3 文件版本

FILE_VERSION：文件版本是反应固件发行和编译的版本号，是一个 4 字节长度的常量。文件版本必须以递增形式进行管理，例如当前的文件版本号必须大于之前发行的版本号，因为当 OTA 时，只有检查到新的版本号大于之前的版本号时才会进行升级。

24 运行模式

Telink 826x/8258/8278 硬件平台支持两种启动模式：多地址启动模式 (从 0x0 或 0x40000 地址启动) 和 BootLoader 启动模式 (从 BootLoader 启动后跳转到 App)。

Telink 802154 SDK 仅支持多地址启动模式，后续会增加 boot loader 启动方式。

24.1 多地址启动模式

优点：启动速度快；OTA 结束后无需再次搬运 image，校验正确后可以快速启动。

缺点：image 只能位于地址 0x0 或 0x40000，会造成 flash 空间分配的不连续性；另外 image 的大小 (如果支持 OTA 的话) 只能小于 208KB。

24.2 多地址启动模式 Flash 分布

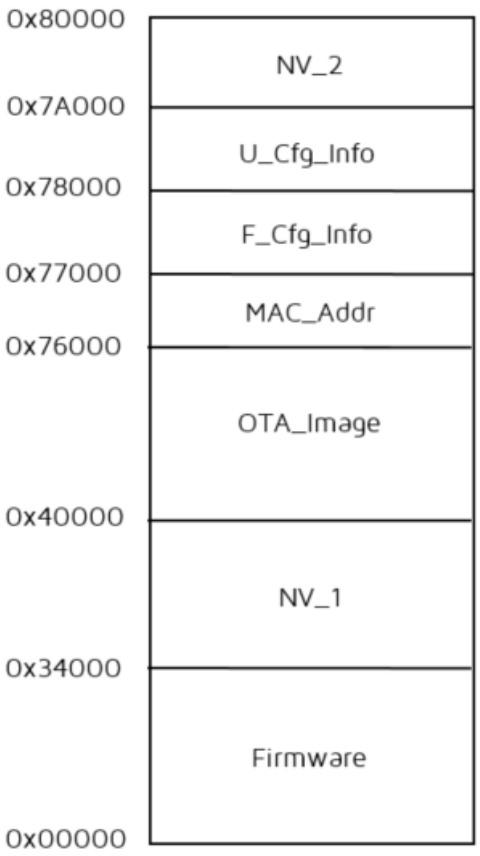

Figure 27: "512k Flash 空间分配图"

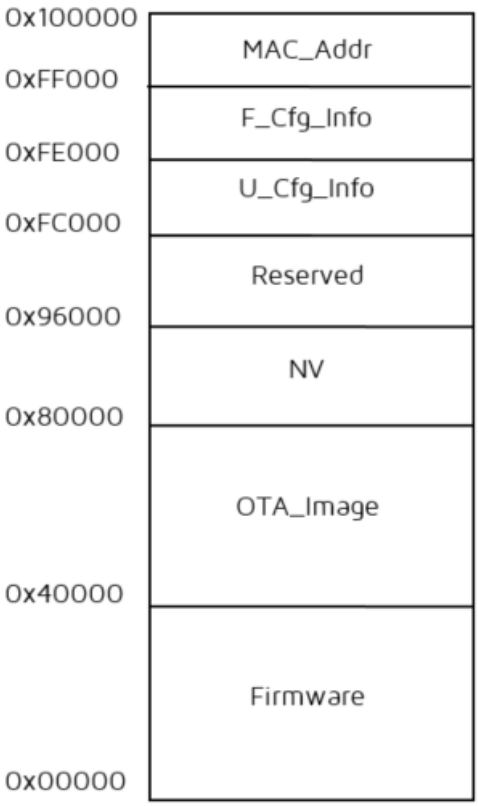

Figure 28: "1M Flash 空间分配图"

2.5 Flash **分布说明**

1) MAC_Addr

MAC 地址信息存放在 Flash 的 0x76000 或 0xFF000 地址开始的 8 个 bytes 位置。系统启动后会检查 MAC 地址信息，如果发现是 0xFFFFFFFFFFFFFFFF，将会随机产生一个 MAC 地址。芯片出厂时会预先写入 MAC 地址，请谨慎擦除。

2) F_Cfg_Info

出厂配置参数信息。芯片在出厂时会预先写入一些校准参数信息，比如射频频偏校准、ADC 校准等，请谨慎擦除。

3) U_Cfg_Info

用户配置参数信息。

4) NV(NV_1, NV_2)

节点入网后会将网络信息保存在 NV 区。512k Flash 网络信息分两部分保存，分别保存在 NV_1 的 48kB 空间和 NV_2 的 24kB 空间。1M Flash 网络信息保存在 NV 的 88kB 空间。

所以，如果只是更新固件或断电重启的话，网络信息不会丢失。

5) Firmware 和 OTA_Image

TLSR8 支持 Flash 多地址启动，既可以从 0x0 地址启动，也可以从 0x40000 地址启动。Telink 802154 SDK 使用该特性，使用乒乓模式，利用 Firmware 和 OTA_Image 相互切换来实现 OTA 功能。

2.6 固件烧录

1) 通过 mini USB 线将 Telink 烧录 EVK 与 PC 相连，EVK 板指示灯会闪烁一次，表示 EVK 与 PC 连接正常；再使用三根杜邦线将 EVK 的 VCC、GND 和 SWM 分别与待烧录目标板的 VCC、GND 和 SWS 相连。

Figure 29: "烧录连接"

2) 硬件连接好之后，打开 Telink BDT.exe 烧录工具，准备烧录固件。

a) 选择"8258"芯片型号（本文档以 8258 为例）。

b) 选择"File" -> "Open"，选择要烧录的固件。

c) 点击"Erase"擦除 512K Flash。

d) 点击"Download"下载固件。

(烧录工具详细的使用方法请参考烧录工具使用文档。)

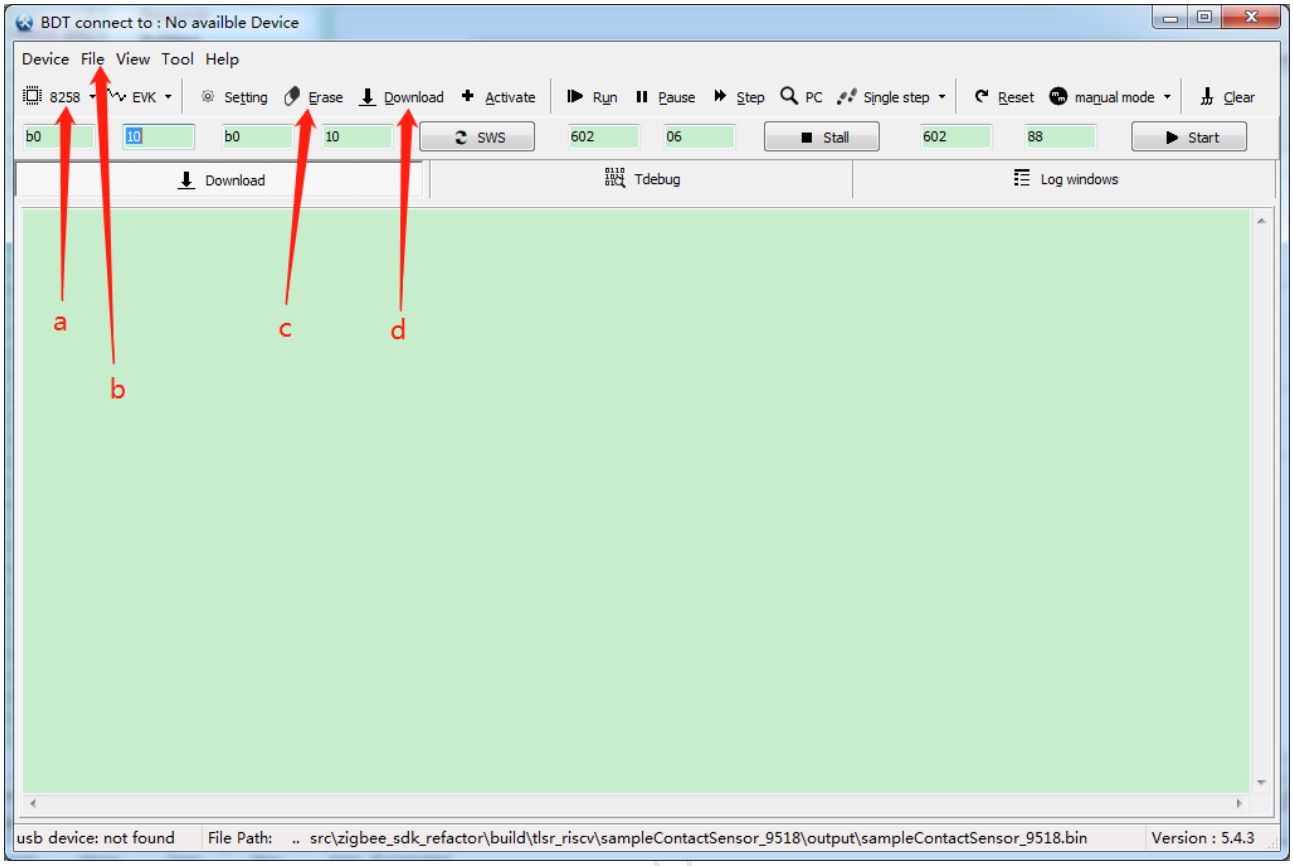

Figure 30: "烧录工具"

3 软件架构

Telink 802154 SDK 的目录结构已在"2.1.2 工程目录结构"章节中列出，本章节会对各个目录下的文件具体作下说明。

3.1 目录说明

3.1.1 硬件平台目录

当前 SDK 支持 b85m（826x、8258、8278）和 b91m（9518）多个平台（tl_802154_sdk -> platform），后续可以添加其他硬件平台。

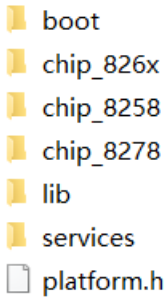

Figure 31: "硬件平台目录"

- \boot：不同平台的.S 启动代码和.link 链接文件
- \chip_826x：826x 平台硬件模块驱动头文件
- \chip_8258：8258 平台硬件模块驱动头文件
- \chip_8278：8278 平台硬件模块驱动头文件
- \chip_b91：b91 平台硬件模块驱动头文件
- \lib：不同平台的驱动库文件
- \services：不同平台中断处理文件

3.1.2 通用函数目录

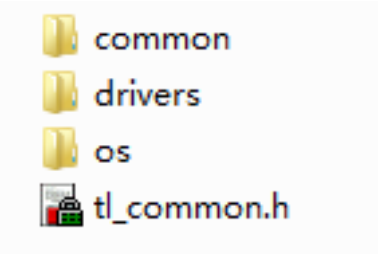

Figure 32: "通用函数目录"

- \common：一些通用的功能函数，如字符串、链表、打印等处理函数文件
- \drivers：抽象层驱动文件
- \os：任务事件、buffer 管理函数文件

3.1.3 802154 协议栈目录

Figure 33: "协议栈目录"

大多以.lib 文件的给出，其中 PHY 层以及与应用层密切相关的 ZCL 和 zbhci 以开源代码方式给出。

- \common：802154 协议栈配置相关文件
- \lib：802154 协议栈库文件
- \mac：媒介控制层（Media Access Control）相关文件，802154 协议栈核心代码

3.14 应用层目录

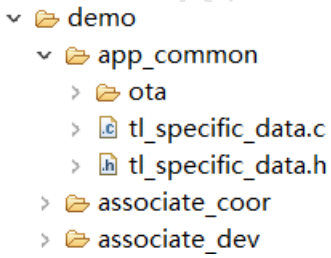

Figure 34: "应用层目录"

- \app_common：应用层通用文件目录
- \ota：协调器和终端节点的 OTA 应用代码

  tl_specific_data.c：应用代码函数入口

  tl_specific_data.h：应用代码配置文件
- \associate_coor：协调器应用例程
- \associate_dev：终端节点应用例程

3.1.5 工程编译目录

Figure 35: "工程编译目录"

- build -> tlsr_riscv：risc-v 平台导入和编译的工程目录，TLSR9 系列芯片请选择使用该目录
- build -> tlsr_tc32：tc32 平台导入和编译的工程目录，TLSR8 系列芯片请选择使用该目录

3.2 抽象层驱动

Telink 802154 sdk 兼容多款 Telink 芯片，为了便于驱动的接口统一，添加了驱动抽象层，位于 proj -> drvier 下，各驱动具体使用方法可参见后续章节。

3.2.1 平台初始化

- **初始化**

```
drv_platform_init(void)
```

完成芯片、系统 clock 的配置、gpio、射频（RF）、timer 等 802154 应用开发中所需的一些模块的初始化。

3.2.2 射频 (RF)

- **初始化**

```
ZB_RADIO_INIT()
```

- **模式切换**

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

mode：RF_MODE_TX = 0, 发送模式

RF_MODE_RX = 1, 接收模式

RF_MODE_AUTO = 2, 802154 未使用

RF_MODE_OFF, 关闭射频模块

Chn：11-26 (对应 802.15.4 中的 2.4G 的 16 个 channel 值)

- **发射功率**

```
ZB_RADIO_TX_POWER_SET(level)
```

设置 RF 模块发送功率，不同芯片对应的功率值稍有不同，具体指请参考 RF_PowerIndexTypeDef 的定义(platform/chip/rf_drv.h)。

- **数据发送**

```
ZB_RADIO_TX_START(txBuf)
```

参数 txBuf：待发送数据的内存地址，数据格式 dma length（4Bytes：payload 长度 +1）+ len（1Byte：payload 长度 +2）+ payload

- **设置射频接收 buffer**

```
ZB_RADIO_RX_BUF_SET(addr)
```

在设置射频为 Rx 模式时，必须确保将 Rx buffer 设置为有效安全的内存地址。

- **RSSI 获取**

```
ZB_RADIO_RSSI_GET()
```

调用该函数时，请确保射频此时处于 Rx 模式。

- **Rssi 转化为 Lqi**

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

mode：仅对 8269 有效

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

- **接收中断**

```
rf_rx_irq_handler(void)
```

- **发送中断**

```
rf_tx_irq_handler(void)
```

注意：射频中断函数 rf_rx_irq_handler / rf_tx_irq_handler 仅能被 802154 stack 中使用，如果用户自行调用 ZB_RADIO_TX_START 所产生的中断，需用户自行注册新的中断回调函数，以避免影响 802154 stack 的运行状态。

3.2.3 GPIO

- **初始化以及配置**

```
gpio_init()
```

此函数是通过将 platform -> chip_xxxx -> gpio_default.h 下的默认配置 (应用层的 board_xxxx.h 可以修改该默认配置）写入 gpio 寄存器，实现 gpio 初始化。

- **IO 功能设置**

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

设置 IO 具体功能。

参数 pin：参见 GPIO_PinTypeDef 的定义。

- **GPIO 读**

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

读取 gpio 的高低电平。

- **GPIO 写**

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

设置 gpio 的高低电平。

- **输出使能**

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

- **输入使能**

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

- **GPIO 中断操作**

芯片最多可以同时支持 3 路外部 GPIO 中断，中断模式分别为：GPIO_IRQ_MODE、GPIO_IRQ_RISC0_MODE 和 GPIO_IRQ_RISC1_MODE。

1) 注册中断服务函数

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

2) 使能中断管脚

```
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

- **管脚设置**

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

使用 uart 之前必须选择相应的 IO 作为 uart 的收发管脚。

- **初始化**

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

Baudrate：波特率

rxBuf：接收 buffer

rxBufLen：接收数据最大长度

uart_recvCb：接收中断时供应用层的回调函数

- **发送**

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

- **接收中断函数**

```
void drv_uart_rx_irq_handler(void)
```

- **发送中断函数**

```
void drv_uart_tx_irq_handler(void)
```

- **异常处理函数**

```
void drv_uart_exceptionProcess(void)
```

注意：当使用 uart 时，为避免通讯异常，main_loop 必须轮询该异常处理函数。

3.2.5 ADC

- **初始化**

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

- **配置**

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

参数 mode：参见 drv_adc_mode_t 定义

参数 pin：所使用的管脚号，参见 GPIO_PinTypeDe 定义

- **获取采样值**

```
u16 drv_get_adc_data(void)
```

3.2.6 PWM

- **初始化**

```
void drv_pwm_init(void)
```

- **配置**

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

参数 pwmId：pwm 通道

参数 cmp_tick：PWM 一个周期中处于高电平的 tick 数

参数 cycle_tick：PWM 一个周期包含的 tick 数

3.2.7 TIMER

- **初始化**

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

参数 tmrIdx：TIMER_IDX_0,TIMER_IDX_1，TIMER_IDX_2TIMER_IDX_3

参数 mode：TIMER_MODE_SYSCLK，TIMER_MODE_GPIO_TRIGGER，TIMER_MODE_GPIO_WIDTH，TIMER_MODE_TICK

- **设置**

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

参数 tmrIdx：需要使用的 timer

参数 t_us：定时间隔，单位：us

参数 func：该定时中断应用层回调函数

参数 arg：应用层回调函数所需参数

- **注销**

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

- **中断函数**

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

其中 TIMER_IDX_2 默认作为 watchdog 使用，建议用户不要使用；

TIMER_IDX_3 用作 MAC-CSMA，建议用户不要使用。

3.2.8 Watchdog

- **初始化**

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

参数 ms：超时时间

- **启动**

```
void drv_wd_start(void)
```

- **喂狗**

```
void drv_wd_clear(void)
```

3.2.9 System Tick

System Tick 计数器，它是一个 32bit 长度、每一个时钟周期自动加一的可读计数器。

由于 826x 和其他系列 IC 的 System Timer 的时钟源不同，8269 的 System Timer 是 32M，8258、8278 和 9518 的 System Timer 是 16M，所以在最大计数时间上存在区别。

- 8269 最大定时时间：(1/32)us*(2^32) 约等于 134 秒，每过 134 秒 System Timer Tick 转一圈。
- 8258、8278、9518 最大定时时间：(1/16)us*(2^32) 约等于 268 秒，每过 268 秒 System Timer Ticker 转一圈。

用户可以使用 clock_time() 接口获取当前 tick。

3.2.10 电压检测

为了防止低压系统运行异常，SDK 提供了基于 ADC 驱动实现的电压检测函数，需要注意的是：

1) 需要使用支持 ADC 功能的 I/O 口，且用于 ADC 检测的 I/O 口不可以用做其他功能

2) 使用 DRV_ADC_VBAT_MODE 模式时，I/O 口需要悬空

3) 使用 DRV_ADC_BASE_MODE 模式时，I/O 口需要连接电压测试点

4) B91 只能使用 DRV_ADC_BASE_MODE 模式，且 I/O 口需要做外部分压处理

- **初始化**

```
void voltage_detect_init(void)
```

- **电压检测**

```
voltage_detect(void)
```

3.2.11 睡眠和唤醒

Telink 802154 SDK 提供了相关的低功耗管理函数。associate_dev_826x 使用 suspend 模式；associate_dev_8258、associate_dev_8278 和 associate_dev_B91 使用 deep with retention 模式，即休眠时 RAM 数据可以保持（8258 和 8278 支持 32k RAM 保持，B91 支持 64k RAM 保持，详细请查阅数据手册）。

- **唤醒源类型**

支持按键以及 timer 定时唤醒。

- **配置唤醒引脚**

如果需要按键唤醒功能，首先需要配置对应的唤醒管脚以及唤醒电平。

```
/**
* @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));
```

- **休眠函数**

```
drv_pm_lowPowerEnter(void);
```

如果使能了按键唤醒，调用此函数时，若相应管脚当前的电平与唤醒电平一致，则系统不能有效进入低功耗状态。

- **休眠函数说明**

1）调用 tl_stackBusy() 和 zb_isTaskDone() 检查是否符合休眠条件；

2）遍历软件定时任务列表，检查是否存在定时任务，如果有执行 3），如果没有执行 4）；

3）检索出临近任务的时间，以该时间作为休眠时间，进入 Suspend 或 Deep Retention 休眠模式，可以定时器唤醒和按键唤醒；

4）进入 Deep 休眠模式，可以按键唤醒。

3.3 存储管理

3.3.1 动态内存管理

Telink 802154 SDK 提供了动态内存分配和释放的接口，默认最大支持 142 字节长度的内存申请，用户可以在开发中直接使用，接口如下。

1) 申请内存

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

2) 释放内存

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

3) 资源配置

默认由 4 组不同个数、不同大小的内存池构成，用户可以根据产品需求以及当前内存使用情况自行修改其个数以及大小。

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

3.3.2 NV 管理

因为 802154 需要保存各层的网络信息参数，在异常断电后能够恢复网络，所以 SDK 从 FLASH 中划分了一块区域专门用来存储此类信息，这块 FLASH 区域我们称它为 NV 区，即第 2.4 章节的 NV(NV_1, NV_2) 区。

由于芯片 FLASH 仅支持 sector（1 sector = 4k）擦除，所以 NV 区的信息存储模块的最小单位为 1 个 sector。

SDK 中使用了以下几个信息模块，其中 NV_MODULE_USER_INFO1 和 NV_MODULE_USER_INFO2 是供用户使用，如需添加新的信息模块，请在末尾添加，不能改变现有的模块序号。

```
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;
```

信息模块又由多个条目信息组成，例如 NV_MODULE_USER_INFO1 由一系列的 NV_ITEM_USER_INFO 组成。条目定义如下，如需添加新的条目信息，请在末尾添加，不能改变现有的条目序号。

```
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;
```

- 每个 module 占用 2 个或 (2*n)sectors，格式：

  sector info + item 索引 + item 内容

其中 sector info 标识该块是否有效，长度为 sizeof(nv_sect_info_t)，之后紧跟待写入的 item 索引，item 内容起始于该 module 首地址偏移 512Byte 或 1024Byte 的位置。

- 为了避免频繁进行擦除操作，NV 采用单 module 追加写入的方法，直到 1 个 Sector 写满后，将有效信息搬到另一个 sector，再将之前的 sector 内容清除。
- NV 区读写操作接口如下：

```
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);
```

注意：对于某个 item，如果只存在一个有效值的话，则 single 为 1，那么当再次写这个 item 时，会将之前有效的 item 清除；否则的话，需要根据内容检索相同的 item，将其置为无效，再写入新的 item。

34 任务管理

34.1 单次任务队列

- 接口函数

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

参数 func：任务回调函数

参数 arg：任务所需参数

- 只执行一次，没有按优先级，顺序依次处理
- 建议使用场合：

  1) 需要避免函数深度嵌套引起的堆栈溢出问题；

  2) 中断函数中，通过将数据、事件压入队列，避免在中断函数里消耗过多时间。
- 大小：32 个（使用时避免一次性压入过多任务）

34.2 常驻任务队列

- **任务注册** (任务注册后默认启动)

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

- **任务暂停**

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

- **任务重启**

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

此任务注册后，会在主循环中一直执行。

34.3 软件定时任务

为了方便用户实现对时间精度需求不高的定时任务，Telink 802154 SDK 提供了一个软件定时任务管理机制。

需要注意的是，从 SDK V3.6.2 版本开始，软件定时任务管理的时间单位从原来的 ticker 改为毫秒，这解决了长时间定时任务需求的问题。

34.3.1 接口函数

- **任务注册**: 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
*/
```

- **任务取消**: TL_ZB_TIMER_CANCEL(evt)

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

34.3.2 注意事项

1) 定时任务回调函数返回值的三种用法：

- 返回值小于 0，则该任务执行后被自动删除，任务不复存在。
- 返回值等于 0，则一直使用之前启动时的 t_ms 来周期定时触发回调函数。
- 返回值大于 0，则使用该返回值作为新的周期定时触发回调函数，单位 ms。

注意：在中断函数里避免使用 TL_ZB_TIMER_CANCEL()

34.3.3 使用实例

当 VK_SW1 按键按下后，启动或取消一个 10 秒的定时任务，时间到达后执行一次广播 On/Off Toggle 的命令，执行结束后退出。代码如下：

```
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 常用 APIs

MAC 提供了一系列 APIs，供应用层 (APP) 和 MAC 层实现命令、数据、状态的交互：

a) 请求 (Request) 及确认 (Confirm)：应用层对 MAC 层发出命令请求 (COMMAND-request) 和数据请求 (DATA-request)，以及收到 MAC 层回复的状态确认 (CONFIRM).

b) 接收 (indication)：应用层收到 MAC 层命令 (COMMAND-indication) 和数据 (DATA-indication).

应用层通过 MAC 层管理实体 (MAC Layer Management Entity，MLME) 和 MAC 层数据传输服务 (MAC CommonLayer Service Access Point，MCPS) 来实现。

4.1 MAC 层管理实体 (MAC Layer Management Entity，MLME)

4.1.1 MLME-POLL

1) **MLME-POLL.request**

实现函数：

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

- req 为指向 MLME-POLL.request 原语 (primitive) 的结构体变量。

MLME-POLL.request 原语数据结构：mac_mlme_poll_req_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

2) **MLME-POLL.confirm**

注册回调函数：

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

- CALLBACK_POLL_CONFIRM 为 MLME-POLL.confirm 回调函数 ID，为固定值。
- void MyPollCnfCb(unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-POLL.confirm 原语 (primitive) 的指针。

MLME-POLL.confirm 原语数据结构：mac_mlme_poll_conf_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

3) **MLME-POLL.indication**

注册回调函数：

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

- CALLBACK_POLL_INDICATION 为 MLME-POLL.confirm 回调函数 ID，为固定值。
- void MyPullIndCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-POLL.indication 原语 (primitive) 的指针。

MLME-POLL.indication 原语数据结构：mac_mlme_poll_ind_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4.1.2 MLME-ASSOCIATE

1) **MLME-ASSOCIATE.request**

实现函数：

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

- req 为指向 MLME-ASSOCIATE.request 原语 (primitive) 的结构体变量。

MLME-ASSOCIATE.request 原语数据结构：zb_mlme_associate_req_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

2) **MLME_ASSOCIATE.confirm**

注册回调函数：

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

- CALLBACK_ASSOCIATE_CONFIRM 为 MLME-ASSOCIATE.confirm 回调函数 ID，为固定值。
- void MyAssocCnfCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-ASSOCIATE.confirm 原语 (primitive) 的指针。

MLME-ASSOCIATE.confirm 原语数据结构：zb_mlme_associate_conf_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

3) **MLME_ASSOCIATE.indication**

注册回调函数：

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

- CALLBACK_ASSOCIATE_INDICATION 为 MLME-ASSOCIATE.indication 回调函数 ID，为固定值。
- void MyAssociateIndCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-ASSOCIATE.indication 原语 (primitive) 的指针。

MLME- ASSOCIATE.indication 原语数据结构：zb_mlme_associate_ind_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4) **MLME_ASSOCIATE.response**

实现函数：

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

- req 为指向 MLME-ASSOCIATE.response 原语 (primitive) 的结构体变量。

MLME-ASSOCIATE.response 原语数据结构：zb_mlme_associate_resp_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4.1.3 MLME-SCAN

1) **MLME_SCAN.request**

实现函数：

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

- req 为指向 MLME-SCAN.request 原语 (primitive) 的结构体变量。

MLME-SCAN.request 原语数据结构：zb_mac_mlme_scan_req_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

2) **MLME_SCAN.confirm**

注册回调函数：

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

- CALLBACK_SCAN_CONFIRM 为 MLME-SCAN.confirm 回调函数 ID，为固定值。
- void MyScanCnfCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-SCAN.confirm 原语 (primitive) 的指针。

MLME-SCAN.confirm 原语数据结构：zb_mac_mlme_scan_conf_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4.1.4 MLME-START

1) **MLME_START.request**

实现函数：

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

- req 为指向 MLME-ASSOCIATE.request 原语 (primitive) 的结构体变量。

MLME-ASSOCIATE.request 原语数据结构：zb_mlme_associate_req_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

2) **MLME_START.confirm**

注册回调函数：

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

- CALLBACK_START_CONFIRM 为 MLME-START.confirm 回调函数 ID，为固定值。
- void MyStartCnfCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-START.confirm 原语 (primitive) 的指针。

MLME- START.confirm 原语数据结构：mac_mlme_startCnf_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4.1.5 MLME-DISASSOCIATE

1) **MLME_DISASSOCIATE.request**

实现函数：

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

- req 为指向 MLME-DISASSOCIATE.request 原语 (primitive) 的结构体变量。

MLME- DISASSOCIATE.request 原语数据结构：zb_mlme_disassociate_req_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

2) **MLME_DISASSOCIATE.confirm**

注册回调函数：

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

- CALLBACK_ASSOCIATE_CONFIRM 为 MLME-ASSOCIATE.confirm 回调函数 ID，为固定值。
- void MyDisassocCnfCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-DISASSOCIATE.confirm 原语 (primitive) 的指针。

MLME- DISASSOCIATE.confirm 原语数据结构：zb_mlme_disassociate_conf_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

3) **MLME_DISASSOCIATE.indication**

注册回调函数：

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

- CALLBACK_DISASSOCIATE_INDICATION 为 MLME-DISASSOCIATE.indication 回调函数 ID，为固定值。
- void MyDisassociateIndCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-DISASSOCIATE.indication 原语 (primitive) 的指针。

MLME-DISASSOCIATE.indication 原语数据结构：zb_mlme_disassociate_ind_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4.1.6 MLME-BEACON-NOTIFY

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

注册回调函数：

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

- CALLBACK_BEACON_NOTIFY_INDICATION 为 MLME-BEACON-NOTIFY.indication 回调函数 ID，为固定值。

- void MyDisassociateIndCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-BEACON-NOTIFY.indication 原语 (primitive) 的指针。

MLME-BEACON-NOTIFY.indication 原语数据结构：zb_mlme_beacon_notify_ind_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4.1.7 MLME-COMM-STATUS

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

注册回调函数：

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

- CALLBACK_COMM_STATUS_INDICATION 为 MLME-COMM-STATUS.indication 回调函数 ID，为固定值。
- void MyStateIndCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MLME-COMM-STATUS.indication 原语 (primitive) 的指针。

MLME-COMM-STATUS.indication 原语数据结构：zb_mlme_comm_status_ind_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> tl_zb_mac.h)。

4.2 MAC **层数据传输服务** (MAC Common Layer Service Access Point，MCPS)

4.2.1 MCPS-DATA

1) **MCPS-DATA.request**

实现函数：

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

- req 为指向 MCPS-DATA.request 原语 (primitive) 的结构体变量。
- payload 为指向需要发送数据的指针。
- pay_len 为需要发送数据的长度。

MCPS-DATA.request 原语数据结构：zb_mscp_data_req_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

2) **MCPS-DATA.confirm**

注册回调函数：

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

- CALLBACK_MCPS_DATA_CONFIRM 为 MCPS-DATA.confirm 回调函数 ID，为固定值。
- void MyDataCnfCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MCPS-DATA.confirm 原语 (primitive) 的指针。

MCPS-DATA.confirm 原语数据结构：zb_mscp_data_conf_t。函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

3) **MCPS-DATA.indication**

注册回调函数：

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

- CALLBACK_DATA_INDICATION 为 MCPS-DATA.indication 回调函数 ID，为固定值。
- void MyDataIndCb (unsigned char *pData) 为应用层声明的回调函数，pData 为指向 MCPS-DATA.indication 原语 (primitive) 的指针。

MCPS-DATA.indication 原语数据结构：zb_mscp_data_ind_t。

函数声明文件：upper_layer.h (路径：tl_802154_sdk -> 802154 -> mac -> includes)。

4.3 ASP(ATTRIBUTE SETTING)

MAC PIB 属性 (attribute) 通过 ASP 来设置，这部分代码是开源的。

文件路径：tl_802154_sdk -> 802154 -> mac -> mac_pib.c

4.3.1 **写** MAC PIB **属性函数** (write MAC PIB attribute)

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

- attribute：MAC PIB 属性 (attribute) 号 (id)。
- value：写入对应 MAC PIB 属性 (attribute) 值的指针。
- index：如 attribute 写为加密相关的属性 (attribute) 号 (id)，例如 MAC_KEY_TABLE、MAC_DEVICE_TABLE、MAC_SECURITY_LEVEL_TABLE，这几个属性 (attribute) 号 (id) 对应的 MAC PIB 属性均为有限个数的数组，则 index 表示为需要写入的对应入口号；如 attribute 其他值，index 为对应 MAC PIB 属性 (attribute) 值的长度。
- 返回值：如果写成功返回 MAC_SUCCESS，否则返回 MAC_INVALID_PARAMETER。

4.3.2 **读** MAC PIB **属性函数** (read MAC PIB attribute)

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

- attribute ： MAC PIB 属性 (attribute) 号 id。
- value：读对应 MAC PIB 属性 (attribute) 值的指针。
- index：如 attribute 写为加密相关的属性 (attribute) 号 (id)，例如 MAC_KEY_TABLE、MAC_DEVICE_TABLE、MAC_SECURITY_LEVEL_TABLE，这几个属性 (attribute) 号 (id) 对应的 MAC PIB 属性均为有限个数的数组，则 index 表示为要读的对应入口号；如 attribute 其他值，index 为对应 MAC PIB 属性 (attribute) 值的长度。

- 返回值：如果读成功返回 MAC_SUCCESS，否则返回 MAC_INVALID_PARAMETER。

初次上电，SDK 定义了 MAC PIB 默认值。

文件路径：tl_802154_sdk -> 802154 -> common -> zb_config.c

默认值变量：const tl_zb_mac_pib_t macPibDefault

4.3.3 读写 MAC PIB 属性 (attribute) 示例

1) **associate device 设置 PAN ID**

```
u8 len = 2;//pan id 长度为 2 字节
u16 panid = 0xbeef;
tl_zbMacAttrSet(MAC_ATTR_PAN_ID, &panid ,len);
```

- MAC PIB 属性 (attribute)ID attribute = MAC_ATTR_PAN_ID
- 写入值指针 value = &panid
- 写入长度 len = 2

2) **associate device 读 associate coordinator 长地址 ()**

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

- MAC PIB 属性 (attribute) ID
  attribute = MAC_ATTR_COORDINATOR_EXTENDED_ADDRESS
- 读出值指针 value = &panid
- 读出长度指针 index = &len

5 802.154 SDK 应用开发

5.1 硬件选择

Telink 802154 SDK demo 可以运行在不同系列芯片、不同硬件板子上，用户需要针对不同的硬件条件，做相应的配置。

5.1.1 芯片型号确认

Telink 802154 SDK 在 comm_cfg.h 中列举了以下几种芯片类型，用户在编译工程实例前，请确认相应工程下 version_cfg.h 中选择的芯片类型是否与实际使用的芯片类型一致。

5.1.1.1 comm_cfg.h 芯片类型定义

```
/* 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 芯片类型选择

```
#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 目标板选择

Telink 802154 SDK 提供的 Demo 可以运行在 EVK 板或 USB Dongle 上的，用户可以在相应 Demo 下的 app_cfg.h 文件中选择更改目标板。

```
/* 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
```

在预编译阶段会根据先前的芯片类型选择加载对应的板级配置，配置文件是各工程目录下的 board_xx.h，同时需要注意一下目标板的 FLASH 容量是否与板级配置一致，以防加载数据出错。

```
#defineFLASH_CAP_SIZE_1M      1
```

5.2 打印调试配置

5.2.1 UART 打印

为避免硬件 UART 资源的浪费，我们通过 GPIO 模拟 UART TX 实现了 UART 打印功能（printf() 函数），用户可以任意更改合适的 GPIO。配置方法如下：

1) **打印使能配置：**

```
#define UART_PRINTF_MODE     1
```

2) **打印口配置：**

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

3) **波特率配置：**

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

注意：

1. 826x 最大支持 2M 波特率，8258、8278 和 9518 最大支持 1M 波特率；

2. 不要在中断处理函数中使用该打印，防止打印数据较多或函数嵌套过深导致中断栈溢出。

5.2.2 USB **打印**

可以配合 Telink BDT 工具使用 USB 打印。配置方法如下：

1) **打印使能配置：**

```
#define USB_PRINTF_MODE     1
```

2) **使能 USB 端口：**

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

注意：

当使用 ZBHCI_USB_CDC 或 ZBHCI_USB_HID 功能时，USB 打印功能失效。

5.3 802154 **开发流程**

5.3.1 **应用层初始化（**user_init**）**

应用层初始化主要由 802154 stack 以及 802154 应用层初始化组成。

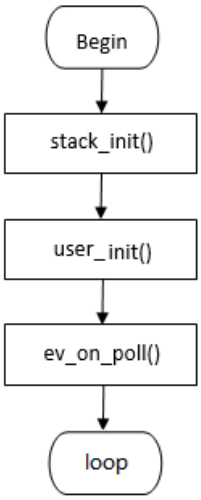

Figure 36: "应用层初始化流程"

1) **os_init(u8 isRetention)**

对于 isRetention 为 0 的上电设备来说，os_init 函数完成任务队列，以及用户内存等非协议栈相关的初始化工作。

2) **user_init()**

用户根据具体产品功能，初始化应用代码。

- 协议栈初始化:

```
zb_init();
```

初始化 mac 层的初始化，对于 not factory-new 设备，会从 NV 中读取入网信息，恢复网络信息以及各层属性。

注意: 对于已经入网的设备，无法通过修改 zb_config.c 里的默认属性表配置达到改变属性的目的，如果想修改属性的话，必须通过 u8 tl_zbMacAttrSet(u8 attribute, u8 *value, u8 index) 函数和 attribute 来修改。

- 设置异常处理回调函数:

```
sys_exceptHandlerRegister(sys_exception_cb_t cb)
```

注册一个异常状态回调函数，SDK 中默认为异常闪灯。

- 注册 MAC 回调函数:

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

根据实际需要用到原语标号 (primitive ID) 注册对应的回调处理函数。

3) **Ev_on_poll()**

注册用户轮询事件。

5.3.2 参数配置

用户可以根据需求在 802154 -> common -> zb_config.c 中修改网络参数配置。

- **macPibDefault**

MAC 层的基本参数信息。

5.3.3 数据安全

15.4 SDK 提供 802.15.4(2006) 标准的加密方式，SDK 中默认使能 15.4 加密功能，应用层只需要配置好 MAC PIB 加密模式，发送前设置和 MAC PIB 对应的加密模式即可。

- MAC PIB 加密模式：SDK 提供的 demo 中已经添加了加密示例，详细请参考函数：void add_key_material(void)
- 发送前设置加密模式：在发送前，设置好结构体 mac_sec_t sec 的值，15.4 SDK 会做模式和密钥的校对，如果和 MAC PIB 中一致，则在对应 confirm 回调函数中返回 MAC_SUCCES，否则返回其他值。参考函数 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: “应用层加密函数”

解密由 MAC 完成，应用层只要配置好 MAC PIB 即可。

54 工作流程

54.1 入网流程

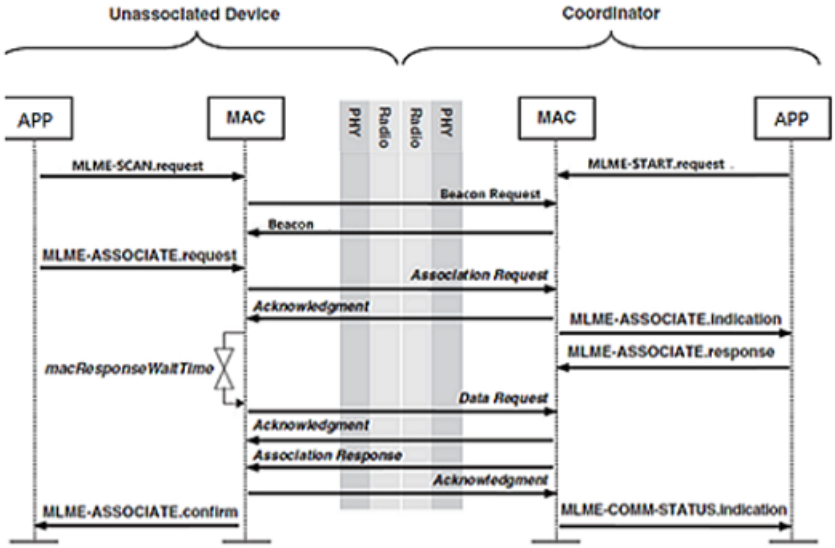

Figure 38: "设备入网流程"

Associate 流程：

1) 终端节点 (end device)

   i) 通过 MLME_SCAN.request 启动 active scan，scan 结束后通过 MLME-SCAN.confirm 的注册回调函数将结果返回到应用层

   MLME_SCAN.request api：MacMlmeScanRequest()

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

   ii) 根据 active scan 结果，通过 MLME-ASSOCIATE.request 向协调器发起 association，等待来自协调器的 associate response 数据包（或超时）

   MLME-ASSOCIATE.request api：tl_zbMacAssociateRequest

   iii) 通过 MLME-ASSOCIATE.confirm 的注册回调函数将 association 结果返回给应用层

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

2) 协调器节点 (coordinator)

   i) 通过 MLME_START.request 启动 energy scan/active scan, 建立 PAN, 后处于监听状态

   MLME_START.request api: tl_zbMacStartRequest()

   ii) 当接收到来自终端节点的 MLME-ASSOCIATE.request 后发送 associate response, 再通过 MLME-COMM-STATUS.indication 告知应用层 association 结果

   如成功, 状态为 MAC_SUCCESS, 其他状态则为失败, 如 pending 列表满 MAC_STA_TRANSACTION_OVERFLOW 等。

具体状态参考枚举变量 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 数据交互

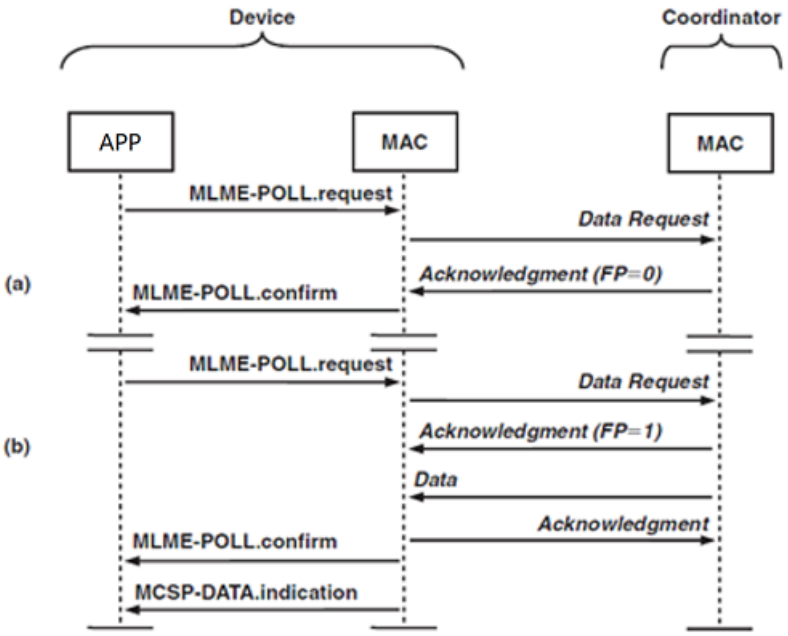

Figure 39: "数据交互流程"

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

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

入网后，对于需要低功耗功能的终端节点，是通过周期性的 MLME-POLL.request 来实现查询和读取 coordinator 端的数据，应用层的代码都是基于这个流程来实现，例如 OTA，终端节点作为开关控制灯等等应用。

54.3 系统异常处理

调用 sys_exceptHandlerRegister() 注册异常回调函数，当 stack 发生 buffer 泄露、状态异常时会触发该函数，在此函数里用户可以添加相应异常处理。

建议：开发阶段，回调函数直接 while(1)，以便定位问题，通过变量 T_evtExcept[1] 可以定位到发生了何种异常；产品阶段，最好做复位处理。

例如：

```
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

TLSR8 和 TLSR9 系列芯片支持 Flash 多地址启动：除了 Flash 地址 0x00000，还支持从 0x40000 读取固件运行。Telink 802154 SDK 使用了该特性来实现 OTA 的功能。

从 2.3 节可知，我们分配了两块固件区域 Firmware 和 OTA-Image，固件大小应不大于 208K。

假设当前正在运行 Firmware 的固件，当设备执行 OTA 升级时，新的固件数据将被存储到 OTA-Image，在 OTA 结束并且验证通过后，将重启并运行 OTA-Image 的固件。后续 OTA，将交替执行。

6.1 OTA 查询功能

6.1.1 ota_queryStart()

启动 OTA 查询功能，由终端节点 (end device) 调用，周期性的查询 coordinator 端是否由固件需要更新。

- **原型**

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

- **返回值** None

Name	Type	Description
seconds	u8	Query cycle, in second

6.2 OTA 设备类型

OTA 设备分为服务设备（Server）和终端设备（Client）。因此，在 OTA 初始化时需要注意 OTA 初始化的服务类型。

一般地，被升级设备为终端设备（Client），一般是指终端节点 (end device)；为被升级设备提供新固件的设备为服务设备（Server），一般是指协调器 (PAN coordinator)。

6.2.1 OTA Server

OTA 服务端需要将被升级设备所需要的新固件写入 OTA-Image 区域供 OTA 使用。

例如：服务端自己运行的固件在 Firmware 区，那么可以将目标设备的 OTA image（包含 OTA Header 的新固件）暂存到 OTA-Image 区域。

6.2.2 OTA Client

入网成功，将调用 void ota_queryStart(u16 seconds) 设置 OTA 终端设备的 OTA 查询周期，单位：秒，即在设置的 seconds 秒后开始 OTA 请求。

OTA Client 端的 OTA image 和正常 bin 同时生成，在正常 bin 前会加上 ota_ 以示区别，只有 CLIENT 端会执行 tl_ota_tool，生成 OTA 文件。

如 2.4.2 flash 分布图所示，OTA_Image 和 firmware 是轮流交替的，例如当前 firmware 在 0x0000 地址，则 OTA 被存储到 0x40000，如 OTA 成功，则 0x0000 地址的 firmware 会被手动置为失效，CLIENT 每次从 0x40000 启动，下一次 OTA 文件则存储到 0x0000，如此往复。

6.3 OTA 流程

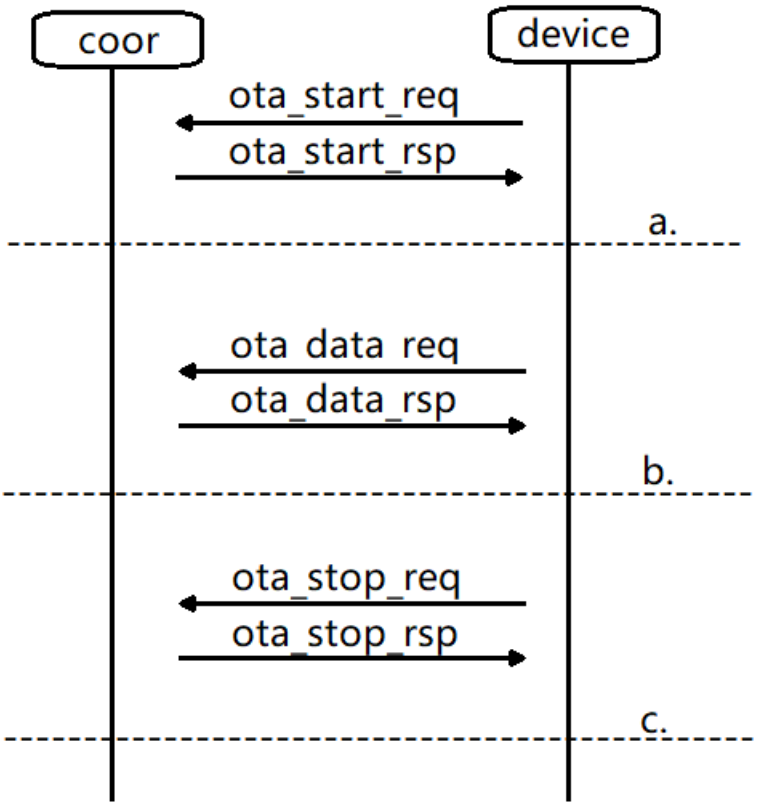

Figure 40: "OTA 流程"

- **OTA START** OTA CLIENT 端周期性的发送 ot_start_req 数据包给 OTA SERVER，具体周期通过函数 ota_queryStart() 设置，单位为秒，ot_start_req 数据包内包含当前 OTA CLIENT 端 bin 文件信息 (文件版本 (file version)、制造商代码 (MANUFACTURER_CODE)、固件类型 (image type));OTA SERVER 端收到 ot_start_req 后，读取 0x40000 位置的 OTA 文件信息，然后将 OTA 文件信息通过 sot_start_rsp 数据包发给 OTA CLIENT;
- **OTA DATA** OTA CLIENT 解析 OTA SERVER 端 start response 后，会得到 OTA SERVER 端 OTA 文件信息，与 OTA CLIENT 当前 bin 比较后，如果制造商代码 (MANUFACTURER_CODE)、固件类型 (image type) 一样，且 OTA SERVER 端 OTA 文件版本 (file version) 大于 OTA CLIENT 当前 bin 文件版本 (file version)，则 OTA CLIENT 发出 ota data request 给 OTA SERVER，接着 SERVER 端读取 OTA 文件，通过 ota data response 发送给 OTA CLIENT，此过程一直持续到 OTA 文件发送完。

- **OTA STOP** OTA CLIENT 端获取的数据长度大于等于 OTA 大小时，CLIENT 会发送 OTA STOP request 给 SERVER，结束整个 OTA 过程，接着 CLIENT 端会把本地计算的 CRC 值和 OTA 文件最后四个字节的 CRC 值做对比，如果一致则重启，否则忽略本次 OTA。