Telink Matter Developer's Guide

(TLSR9518)

Ver 1.0.3

2022/10/14

Keyword

TLSR9518, Matter, Zephyr, Thread, BLE

Brief

This developer's guide provides full information about Telink Matter project setup and usage

Acknowledgements

Published by
Telink Semiconductor

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

Legal Disclaimer

Information

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

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

telinkcnsales@telink-semi.com

telinkcnsupport@telink-semi.com

Revision History

- **1.0.0** Preliminary release
- **1.0.1** Change multicast binding steps
- **1.0.2** Add Usage of LinuxBDT
- **1.0.3** Add Usage of OTA feature, update some details

Contents

Acknowledgements **2**
Legal Disclaimer . . . 2
Information . . . 2

Revision History **2**
1 概述 **5**
1.1 Matter 概述 . . . 5
1.1.1 Matter 的出现 . . . 5
1.1.2 Matter 总览 . . . 6
1.1.3 Matter 功能 . . . 7
1.1.4 Matter 的未来 . . . 8
1.2 Telink Matter 概述 . . . 8
1.2.1 Telink Matter 目前支持的应用 . . . 8
1.2.2 所使用到的仓库和分支 . . . 8
1.2.3 此文档的目的 . . . 9
2 硬件和工具准备 **9**
2.1 所需硬件和工具 . . . 9
2.2 TLSR9518 评估板介绍 . . . 10
2.2.1 按键功能 . . . 10
2.2.2 LED 灯状态 . . . 10
2.2.3 连接 UART 模块串口输出 . . . 11
2.2.4 使用自带 PIN 脚进行 VBAT 供电 . . . 11
3 体验 Matter 的实现和功能 **13**
3.1 环境搭建 . . . 13
3.1.1 获取 Matter 源码 . . . 13
3.1.2 Docker 镜像安装 Zephyr 工程环境 . . . 13
3.2 编译固件 . . . 14
3.2.1 内存占用 . . . 14
3.2.2 设备的配置 . . . 15
3.2.3 编译 . . . 15
3.3 编译 chip-tool . . . 16
3.4 固件烧录 . . . 16
3.4.1 BDT 和 LinuxBDT 下载 . . . 16
3.4.2 BDT 的连接方式 . . . 17
3.4.3 BDT 在 Windows 平台上的使用步骤 . . . 17
3.4.4 BDT 在 Windows 平台上的常见问题 . . . 22
3.4.5 LinuxBDT 在 Ubuntu 平台上的使用步骤 . . . 23
3.5 边界路由配置 . . . 26
3.5.1 边界路由需要的硬件和固件 . . . 26
3.5.2 边界路由的树莓派镜像烧录 . . . 26
3.5.3 树莓派和 RCP 的连接方式 . . . 27
3.6 Thread 网络建立和通过 BLE 配网 . . . 28
3.6.1 建立 Thread 网络并获取 Dataset . . . 28
3.6.2 配网过程 . . . 29
3.7 使用 chip-tool 进行控制 . . . 31

3.7.1 开关灯 . . . 31
3.7.2 查看亮灯情况 . . . 31
3.7.3 查看亮度 . . . 32
3.7.4 改变亮度 . . . 32
3.7.5 连接灯和开关 . . . 32
3.8 体验 OTA 功能 . . . 34
3.8.1 启用 OTA 模块 . . . 34
3.8.2 OTA 的用法 . . . 35
4 如何基于/利用 Github 上的开源 SDK 进行开发 . . . 36
4.1 架构 . . . 36
4.2 数据模型 . . . 37
4.3 代码逻辑和重点源码 . . . 38
4.3.1 重要的回调函数 . . . 38
4.3.2 例程代码 . . . 40
4.3.3 硬件平台代码 . . . 40
4.3.4 Matter 配置代码 . . . 41
4.3.5 Matter 平台代码 . . . 41
4.3.6 Telink BLE 单连接 SDK 代码 . . . 42

1 概述

1.1 Matter 概述

1.1.1 Matter 的出现

- 困境

  物联网（IoT）产品"碎片化"的问题不仅让消费者和开发者们头疼不已，也阻碍了家庭智能硬件产品的进一步发展。目前，主导物联网产品发展的科技巨头们，都有各自独立的智能家居生态系统，如 Amazon Alexa，Apple HomeKit 和 Google Home 等等。

- 对于消费者

  消费者在挑选智能产品的过程中，不仅需要关注产品的功能、特性和价格，还需要去考虑它是否兼容家中已有的生态系统，这造成了他们在选择上的困难以及使用中的不便。

- 对于商家

  第三方开发者们在开发一款智能产品之前，也必须考虑该产品需要支持哪一种生态系统，以便于满足该产品目标用户的需求。如果开发者们不满足于将自己的产品限制于某一种生态系统中，想要满足不同的消费者的多样化的需求并拓展自己的市场占有率，他们可能需要改造已有的设备来支持更多的生态系统，或者重新开发支持更多智能生态系统的设备，而这两种选项都可能会让他们面临极大的困境。一方面，不得不花费更多的时间和精力去完成产品的软件层和不同智能生态系统的适配工作；另一方面，甚至要为了满足某些生态系统的要求，去改动底层硬件或者外观的设计。

因此，各方都希望能形成一套共同接受并遵循的"标准"去解决以上的"碎片化"的问题，便于智能家居品牌和制造商的开发，同时也便于消费者的选择。为了满足这一共同需求，包括 Amazon，Apple，Google 等许多物联网生态系统构建者们，与 Zigbee 联盟走到了一起，在 2019 年 12 月宣布成立了 Connected Home over IP（CHIP）项目工作组，致力于打造一个基于开源生态的全新智能家居协议。2021 年 5 月，随着 Zigbee 联盟更名为连接标准联盟（Connectivity Standards Alliance），项目工作组协商制定出这个局域网内的应用层标准协议的初稿，并将 CHIP 改名为：Matter。

1.1.2 Matter 总览

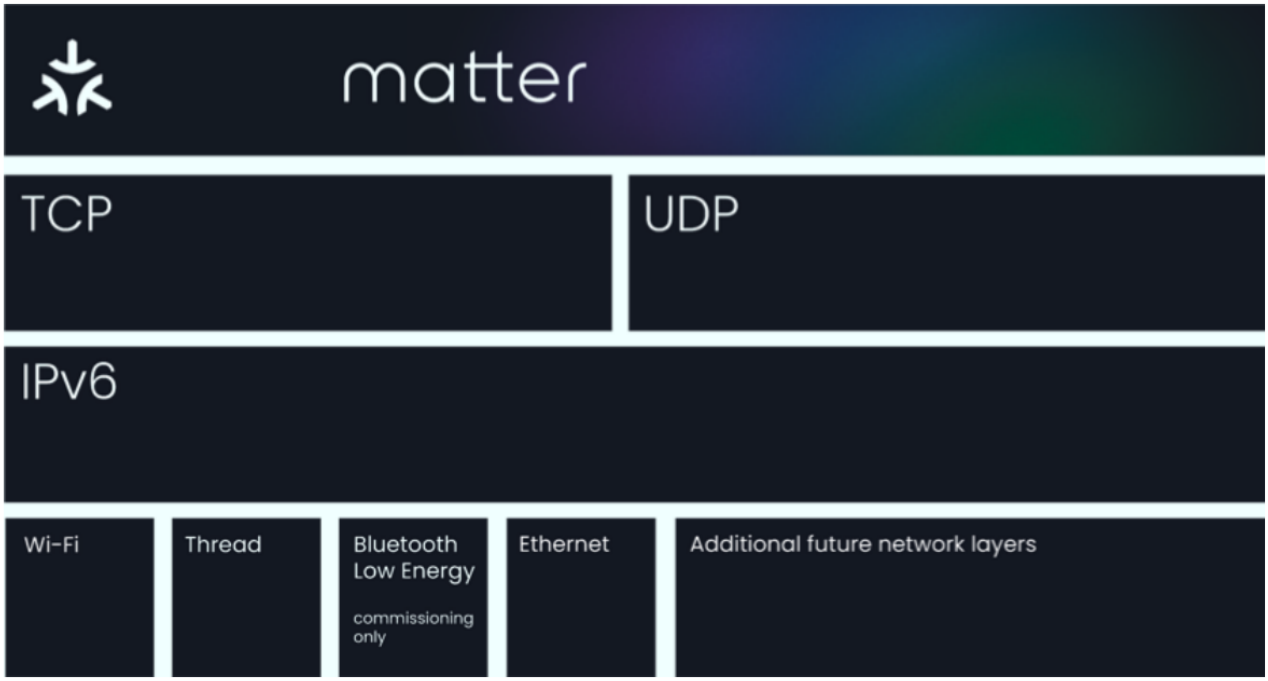

Figure 1.1: Matter

Matter 设备将工作在统一的应用层中，且仅仅依赖传输层中的 IPv6 标准所构成的 TCP/IP 和 UDP 协议，从而兼容不同的物理介质和数据链路标准。因为 IP 网络是一个 Mesh 结构，所以 Matter 也将呈现为 Mesh 拓扑结构（由不同通信技术的子网络组成）。

在计划于今年秋季正式发布的第一版 Matter 协议中，它将首先支持以 Ethernet，Wi-Fi 和 Thread 网络进行设备间通信；并利用蓝牙低功耗（BLE）技术作为 Matter 设备入网的通道，以简化 Matter 设备的配置步骤。其中，Thread 协议基于 IEEE 802.15.4 技术，其网络中的设备无法直接与 Wi-Fi 或以太网设备通信，因此在 Matter 拓扑结构中需要增加 Thread 边界路由器，使 Thread 网络中的 Matter 设备与其他网络中的设备可以互联。比如，一个仅支持 Thread 网络的 Matter 设备，可以通过 Border Router（如 HomePod Mini），来和其他的仅支持 Wi-Fi 网络的设备（如 iPhone）进行通信。

在 Matter 网络中，将拥有统一的数据模型和交互模型。Matter 协议把网络当作共享资源处理，没有制定排他性的网络所有权和访问权，这就使得多个不同协议的网络可以共存于同一组 IP 网络中。在以往，两个智能设备需要处于同一个物理网络中的才可以实现互相通讯，而 Matter 协议将构建起多个虚拟网络，允许不同物理网络中的智能设备实现互相通讯。这里的一个虚拟网络是一群 Matter 设备的集合，被称作一个 Fabric。在实现中，一个 Fabric 往往对应一个智能生态系统所构成的网络。

一个物理设备被叫做 Node（相当于 HomeKit 中的 Accessory），一个 Node 可以被加入到一个或多个 Fabrics 之中。Node 下面用逻辑功能单元 Endpoint 来表示不同功能模块，比如下图中 Endpoint 1 和 Endpoint 2 来表示一个空调的不同功能模块；Endpoint 0 是必须保留的根端点，用来描述设备特性。Endpoint 中用若干个 Cluster（继承于 Zigbee Cluster Library，ZCL）来表述具体功能，如开关，风力，温度测量和温度控制。而 Matter 设备之间的交互（Interaction），则是由一个设备的某个 Endpoint 和另一个设备的某个 Endpoint 之间的通讯来完成的。

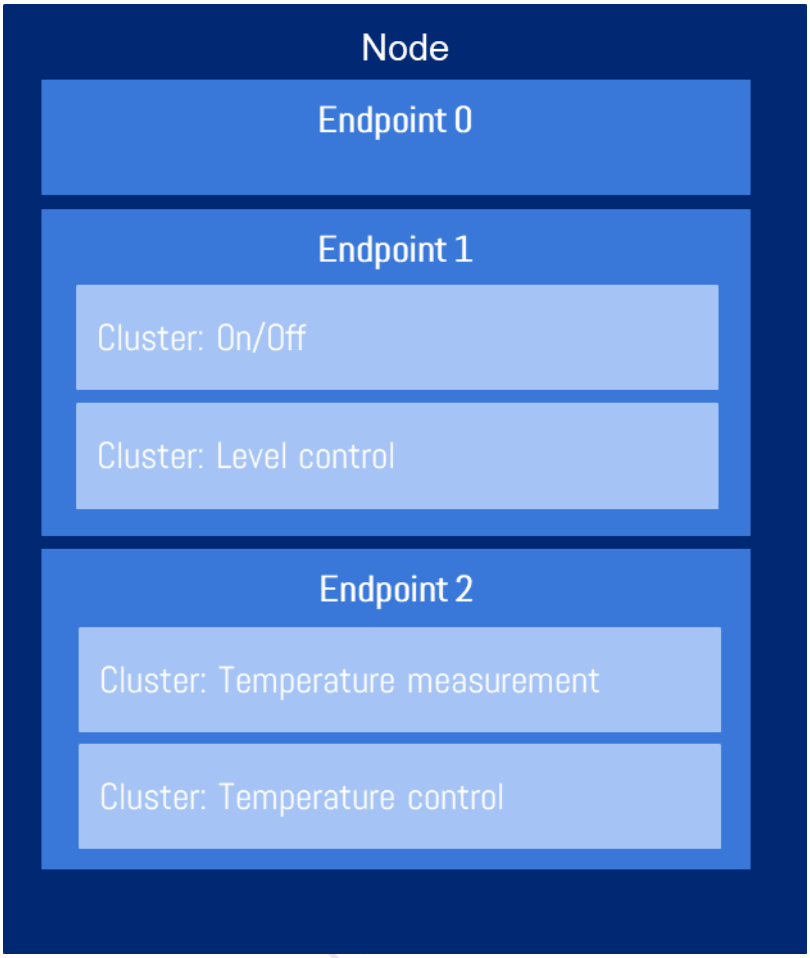

Figure 1.2: Node 示例

除此之外，Matter 协议所构建的网络还具有以下特性：

- Multi-Admin，支持把 Matter 设备同时加入到不同生态系统中的能力，被不同 Fabrics 中的管理员（Administrator）所管理，从而实现广泛的兼容性。
- 通过分布式合规设备总账（Distributed Compliance Ledger, DCL）来共享 Matter 厂商及设备的信息。每个生态系统都可以向 DCL 查询 Matter 配网，OTA 等过程中所需要的信息。
- 允许使用者在无需连接到云端的情况下，进行本地设备的控制。
- 已经大量存在的非 Matter 的智能设备，也有机会通过 Matter Bridge 设备被添加到 Matter Fabric 之中。这个 Matter Bridge 设备负责和非 Matter 设备通信，它会将非 Matter 设备虚拟成对应的 Endpoint，就像一个 Node 里面有多个不同功能的 Endpoints，从而让 Matter 网络中的 Matter 设备可以和非 Matter 设备实现通信。

1.1.3 Matter 功能

Matter 协议主要是面向智能家居市场，其主要支持的设备类型有：

- 照明，开关等照明设备
- 加热，制冷等空气处理设备

- 探测器，报警器等安全设备
- 门禁，门锁等进入控制设备
- 音箱，电视等影音娱乐设备
- 窗户，窗帘等采光通风设备
- 热点，网桥等网络中继设备

随着 Matter 协议的发展和演化，在未来还会支持更多的智能设备。

1.14 Matter 的未来

Matter 项目获得如此高的关注，并不仅仅是因为它概念和标准上的先进，而是源于三位智能家居巨头的承诺。在 Matter 项目立项之初，Amazon，Apple 和 Google 就承诺使用该协议的设备将可以兼容他们的生态。在 Matter 协议推出之后，IoT 产品的开发者们将能够做到一次开发同时支持多个生态系统（Amazon Alexa，Apple HomeKit 和 Google Home 等）的接入协议，这将大大简化开发者的工作，使得智能设备能无缝地连接到任何 Matter 兼容的智能生态系统中。消费者也可以更容易地选购产品，而不用特别担心买到的设备和已有的生态系统的适配问题。

1.2 Telink Matter 概述

泰凌微电子积极参与了 Matter 协议中的 Matter 设备的功能开发，Matter 设备的测试与认证，以及 Matter 标准中文解读等方面的工作。作为致力于低功耗高性能无线连接 SoC 芯片解决方案的提供商，我们推出了基于 TLSR9 系列芯片的 Matter Over Thread 解决方案，可以用于开发 Matter Thread 终端设备。

1.2.1 Telink Matter 目前支持的应用

目前 Telink Matter 支持的例程如下：

1. connectedhomeip/example/lighting-app/telink

   lighting-app 灯泡应用

2. connectedhomeip/example/ligth-switch-app/telink

   light-switch-app 开关应用

为了便于客户体验上述应用，我们提供了预制的 lighting-app 固件，light-switch-app 固件和对应的 chip-tool 工具，可以在下面的联想网盘地址中找到：

https://console.box.lenovo.com/l/NJioaS

提取码：surz

1.2.2 所使用到的仓库和分支

最新的 Docker Image 仍然在更新中，截至目前其中使用的 Zephyr 环境为 V3.1.99，如果您想要了解，可以查看：

https://hub.docker.com/r/connectedhomeip/chip-build-telink

Matter 的 custom branch 分支 telink_matter 已经合并进了官方的 master 分支，因此可以使用以下仓库：

https://github.com/project-chip/connectedhomeip

1.2.3 此文档的目的

本文档提供 Telink Matter 解决方案的完整指导，包括环境搭建、Matter 设备固件构建和烧录、边界路由器设置（包括 RCP 构建和烧录）、chip-tool 的构建和使用等，帮助用户了解 Telink Matter 相关事宜，更好的体验 Telink Matter 应用的功能。

2 硬件和工具准备

2.1 所需硬件和工具

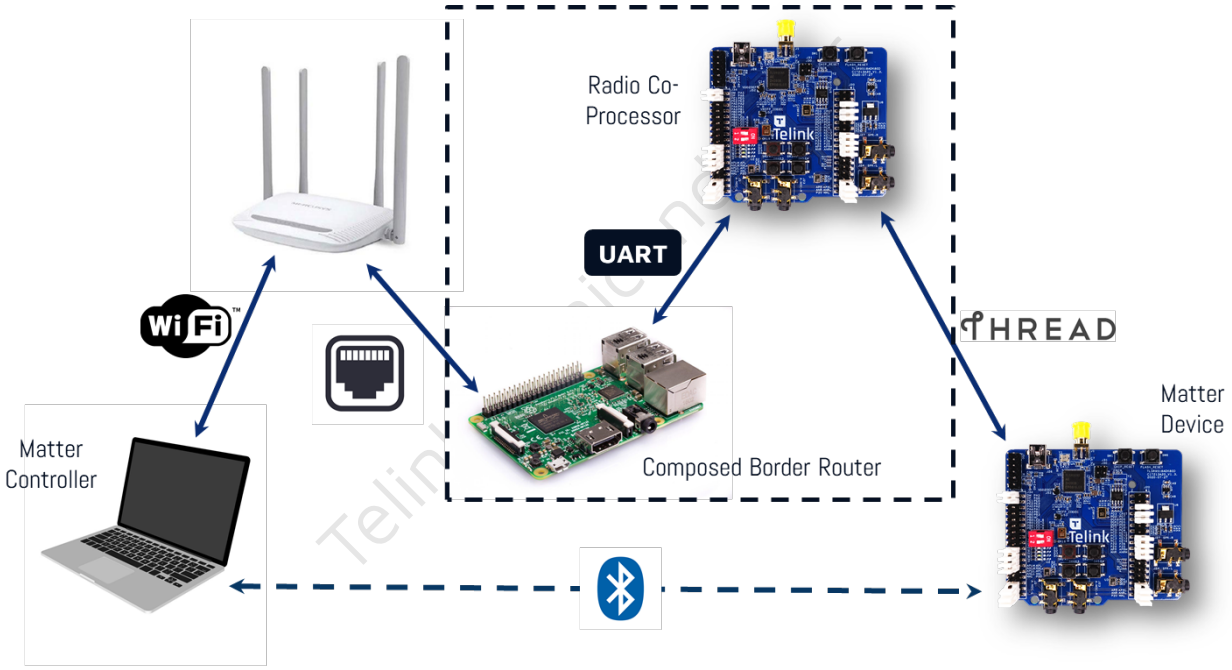

Figure 2.1: Telink-Matter

- **TLSR9518ADK80D** 作为 Matter 设备
- **TLSR9518ADK80D** 作为 RCP
- **Raspberry Pi 3B** 或者更高版本，作为边界路由的一部分
- **SD 卡**给 Raspberry Pi 3B 使用，至少需要 16GB
- **主机 PC** 需要系统为 Ubuntu v20.04 LTS，作为一个构建机器和 Matter 设备的主机
- **Telink 烧录器**用于烧录 Matter 设备和 RCP 固件
- **Wi-Fi 路由器**充当 Wi-Fi 接入点

2.2 TLSR9518 评估板介绍

2.2.1 按键功能

TLSR9518 评估板上的四个机械按键布局和功能如下：

Pin	PC3	PC1
PC2	Key1	Key2
PC0	Key3	Key4

Figure 2.2: 机械按键布局

按钮序号	功能	描述
Key1	恢复出厂设置	清除当前配网的 Thread 网络并返回未配网状态
Key2	灯光控制	对于灯泡，可以手动控制开关灯。对于开关，可以控制连接灯泡的开关灯
Key3	组建 Thread 网络	设备以默认配置加入 Thread 网络，仅用于测试目的
Key4	触发 BLE 广播	进入配网状态，设备开始 BLE 广播，进入可被发现的状态

2.2.2 LED 灯状态

红灯指示设备当前的网络状态：

状态	描述
红灯短亮并闪烁	设备未配网于 Thread 网络，Thread 未使能
红灯频繁闪烁	设备已配网，Thread 使能，设备正在尝试加入 Thread 网络
红灯长亮并闪烁	设备已配网且已作为 child 加入 Thread 网络

蓝灯表示灯的状态：

- 开

亮度 0 - 255 (最大值)

- 关

2.2.3 连接 UART 模块串口输出

UART 模块可以帮助我们获得 Matter 设备的串口调试信息，我们可以按照以下管脚位置进行接线：

UART 模块	TLSR9518 评估板
TXD	PB3(pin 15 of J34)
RXD	PB2(pin 18 of J34)
GND	GND(e.g.pin 23 of J50 or pin 3 of J56)

串口的配置信息：115200，8N1。

2.24 使用自带 PIN 脚进行 VBAT 供电

在没有 USB 线的情况下，TLSR9518 评估板可以用自带 PIN 脚进行供电，但是需要做一些改动。

TLSR9518 评估板按下图做改动：

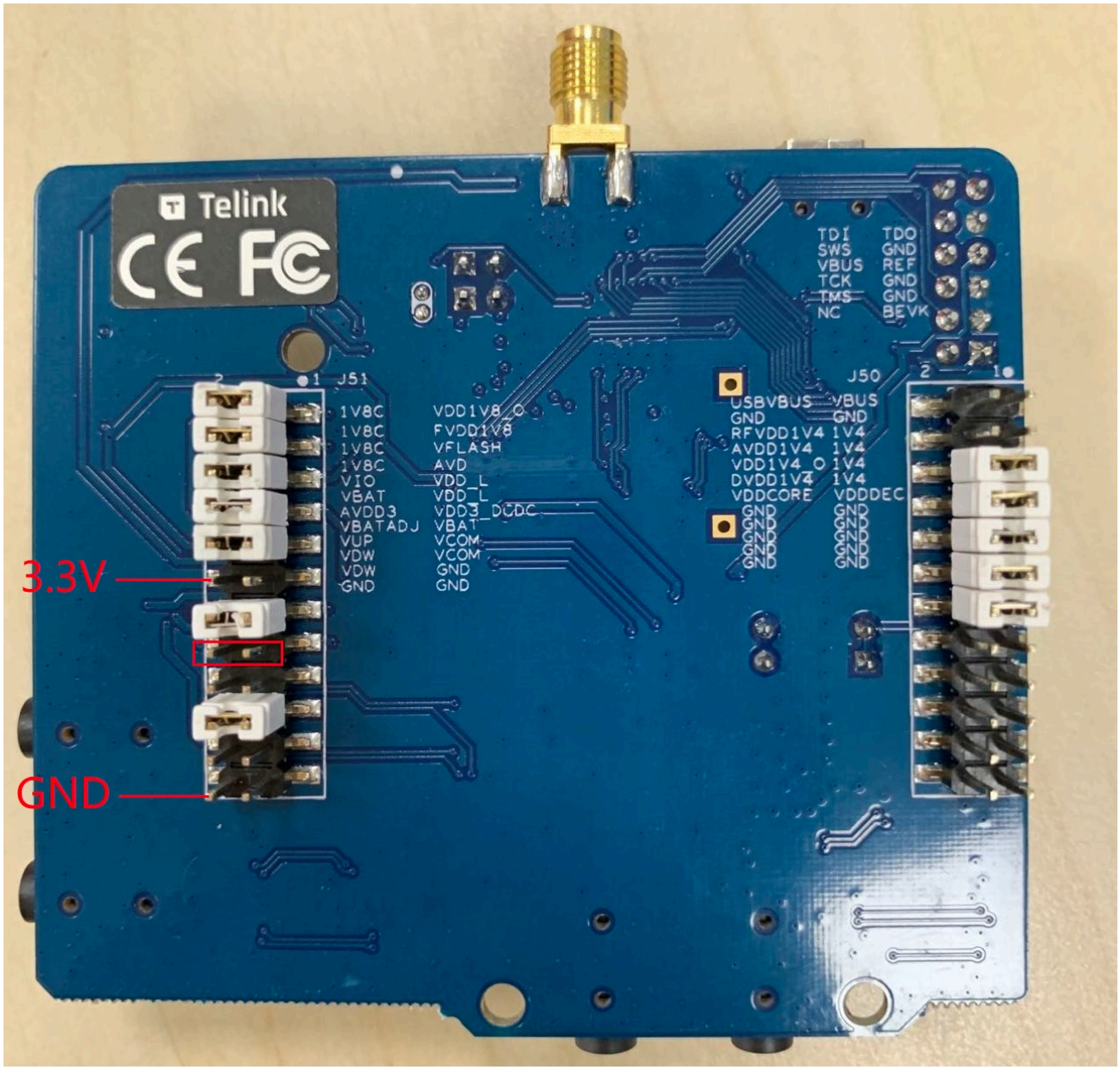

Figure 2.3: 通过 VBAT 脚直接供电

1. 移除上图中红色框标注的跳线帽。
2. J51 的 VBAT 脚接入 3.3V 电源。
3. J51 的 GND 脚接地。

3 体验 Matter 的实现和功能

3.1 环境搭建

3.1.1 获取 Matter 源码

1. 安装依赖项：

```
$ sudo apt-get install git gcc g++ python pkg-config libssl-dev libdbus-1-dev \
libglib2.0-dev libavahi-client-dev ninja-build python3-venv python3-dev \
python3-pip unzip libgirepository1.0-dev libcairo2-dev
```

2. 克隆 Matter 项目：

   将 Matter 项目克隆到本地目录，例如 /home/${YOUR_USERNAME}/workspace/matter。

```
git clone https://github.com/project-chip/connectedhomeip
```

> 其中 **${YOUR_USERNAME}** 是您的用户名文件夹

3. 更新子模块

```
cd ./connectedhomeip
git submodule update --init --recursive
```

3.1.2 Docker 镜像安装 Zephyr 工程环境

Matter 的应用的实现依赖于 Zephyr RTOS，如果您对 Zephyr 比较熟悉，可以根据 Zephyr 3.0.0 去配置本地环境。我们提供了 Docker 镜像，方便获取 Telink Matter 工程所需要的 Zephyr 环境。

1. 获取 Docker 镜像：

```
sudo docker pull connectedhomeip/chip-build-telink
```

2. 运行 Docker 容器：

   在运行 Docker 容器之前，请确认已经通过3.1.1 获取 Matter 源码 中的步骤 1-3 将 Matter 存储库克隆到一个干净的文件夹中。

   使用以下命令来运行 Docker 容器：

```
sudo docker run -it --rm -v ${MATTER_BASE}:/root/chip -v /dev/bus/usb:/dev/bus/usb --
↪  device-cgroup-rule "c 189:* rmw" connectedhomeip/chip-build-telink
```

> 其中 **${MATTER_BASE}** 是 Matter 项目根目录的绝对路径, 例如：

```
/home/${YOUR_USERNAME}/connectedhomeip
```

> 其中 **connectedhomeip** 是 Matter 项目文件夹名称

此处使用的命令会将 Matter 项目根目录映射到 Docker 容器中的 /root/chip，因此即使退出容器，您也会得到生成的 bin 文件。

Docker 容器启动后，请通过以下命令进入当前 Matter 根目录：

```
cd /root/chip
```

3. 运行引导程序

   执行 bootstrap，准备 Matter 的环境，第一次运行通常需要很长时间。

   ```
   source scripts/bootstrap.sh
   ```

   > 注意：每次切换 commit、改变环境都要重新运行引导程序

   此步骤将生成一个在 Matter 根目录 **connectedhomeip** 下的叫做 **.environment** 的隐藏文件夹。在中国大陆它可能会花费额外的时间或遭遇失败。

> 提示：如果 Matter 构建环境有任何问题，您可以尝试：

1. 移除环境（在 Matter 项目的根目录中）：

   ```
   rm -rf .environment
   ```

2. 再次重做引导程序：

   ```
   source scripts/bootstrap.sh
   ```

您可以在这里找到更多信息：https://github.com/project-chip/connectedhomeip/blob/master/docs/guides/BUILDING.md

3.2 编译固件

3.2.1 内存占用

下面是编译出的 lighting-app 的内存占用情况：

内存区	已使用大小	总大小	使用%
RAM ILM	62632 B	128 K	47.78%
RAM DLM	74968 B	128 K	57.20%
FLASH	752888 B	1 M	71.80%

> 注意：表格内的信息仅具有参考价值，最准确的内存占用情况请下载最新的 Matter 代码进行确认。

3.2.2 设备的配置

如果一个或多个设备被配置为 FTD (Full Thread Device)，例如灯泡，则设备到设备之间的通信可以不需要边界路由器。

> 注意：默认情况下，所有设备都配置为 MTD（Minimal Thread Device)。

应用程序的配置文件位于：

```
examples/app/telink/prj.conf
```

MTD 配置示例：

```
# OpenThread configs
CONFIG_OPENTHREAD_MTD=y
CONFIG_OPENTHREAD_FTD=n
```

FTD 配置示例：

```
# OpenThread configs
CONFIG_OPENTHREAD_MTD=n
CONFIG_OPENTHREAD_FTD=y
```

3.2.3 编译

在 Matter 根目录执行以下操作，如果使用 Docker 镜像的话则在 /root/chip 中进行操作：

1. 启动 Matter 环境：

   ```
   source scripts/activate.sh
   ```

2. 转到示例所在目录：

   ```
   cd examples/*app*/telink
   ```

   app：lighting-app 或 light-switch-app

3. 若已经存在构建，则删除原有构建时产生的目录：

   ```
   rm -rf build/
   ```

4. 构建示例：

   ```
   west build
   ```

   您可以在 **build/zephyr** 目录下找到名为 **zephyr.bin** 的目标构建文件。

> 注意：若您不采用我们提供的 docker，而是选择手动配置 Zephyr 的工程，请确保您的 PC 中已经安装并配置了 gn 构建工具，否则可能出现构建错误。另外，请记得执行 zephyr 路径下的 zephyr-env.sh，否则，可能出现 west build 不存在的报错。

3.3 编译 chip-tool

1. 启动 Matter 环境：

```
source scripts/activate.sh
```

2. 转到示例所在目录：

```
cd examples/chip-tool
```

3. 若已经存在构建，则删除原有构建时产生的目录：

```
rm -rf out/
```

4. 构建 chip-tool

```
gn gen out
ninja -C out
```

构建完成的文件可以在 **{MATTER_CHIP_TOOL_EXAMPLE_FOLDER}/out/chip-tool** 找到。

如果想了解更多和 chip-tool 相关的内容，可以点击以下链接进行查看：https://github.com/project-chip/connectedhomeip/blob/master/examples/chip-tool/README.md

34 固件烧录

本烧录说明的实现适用于 Windows 和 Ubuntu 平台，烧录一块 B91 EVB 所需要准备的硬件，至少需要有：

1. 一块 B91 EVB 评估板；
2. 一块 Burning Evk 烧录器；
3. 两根 Mini-USB 线；

如需同时烧录多块 B91 EVB 评估板，请准备多套以上的硬件。另外，可能需要足够接口的 USB Hub。

34.1 BDT 和 LinuxBDT 下载

BDT 是烧录调试工具，可以通过下面链接获取到最新的烧录工具：

Windows 10：http://wiki.telink-semi.cn/tools_and_sdk/Tools/BDT/BDT.zip

Ubuntu 20.04：http://wiki.telink-semi.cn/tools_and_sdk/Tools/BDT/LinuxBDT.tar.bz2

34.2 BDT 的连接方式

请按照下面的图示进行硬件连接，如图是完成硬件连接后状态：

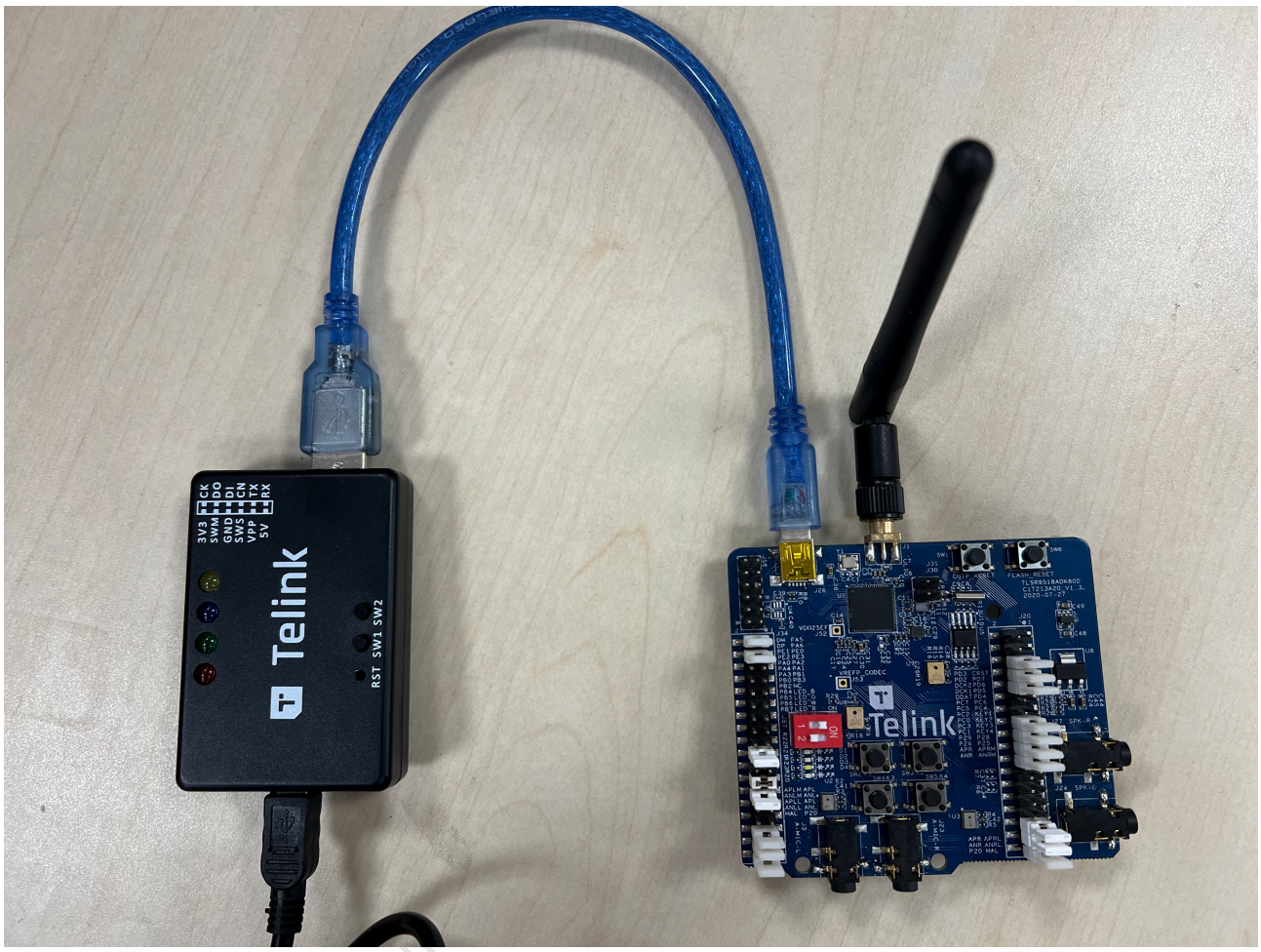

Figure 3.1: 硬件连接实例

> 注意：务必采用默认的跳帽配置。

34.3 BDT 在 Windows 平台上的使用步骤

1. 用 USB 线连接烧录器到电脑的 USB 口。
2. 下载 BDT 烧录软件，解压到本地文件夹，双击可执行文件"Telink BDT.exe"。如果一切正常，可以看到如下的窗口显示，在系统标题栏中可以看到已被连接的烧录器的设备信息（见图中红色框）。

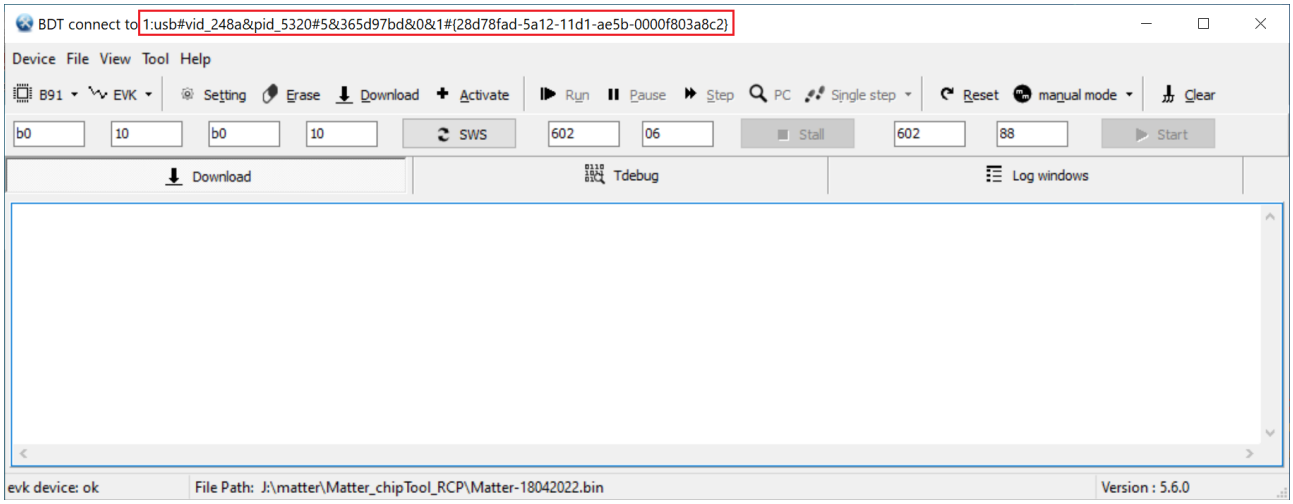

Figure 3.2: BDT 正常启动并连接后的页面

3. 点击工具栏中的"SWS"按钮，如果看到下图中的信息，则表明所有的硬件连接都没有问题。

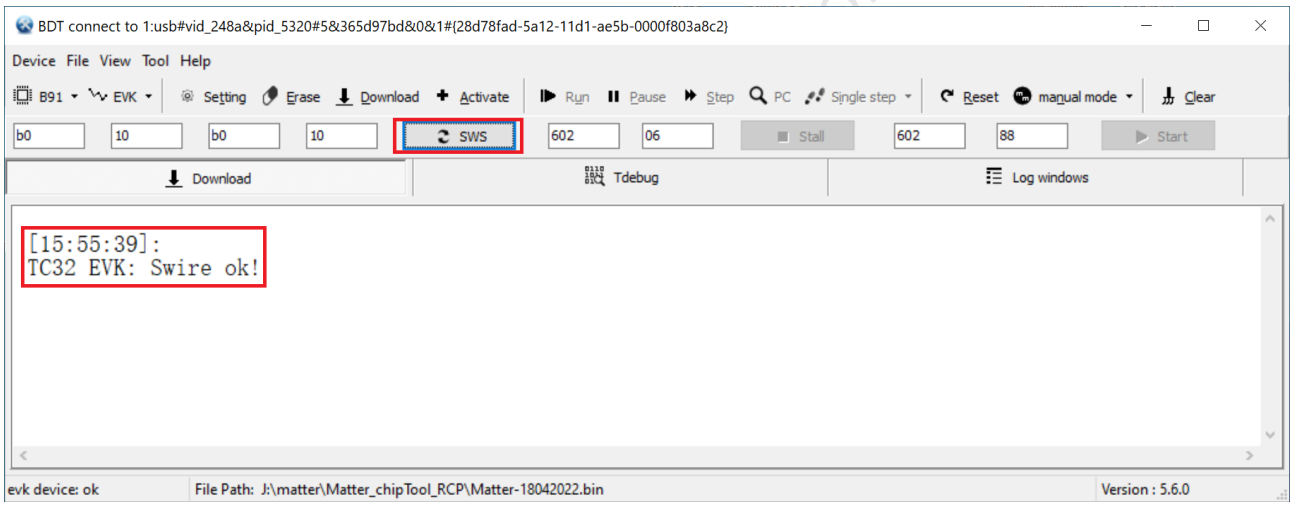

Figure 3.3: 按下 SWS 按钮后显示的信息

4. 设置 Flash 擦除的区域大小。点击工具栏中的"Setting"按钮，在弹出的"Setting"窗口中可以看到默认的 Flash 擦除的区域大小是 512kB。

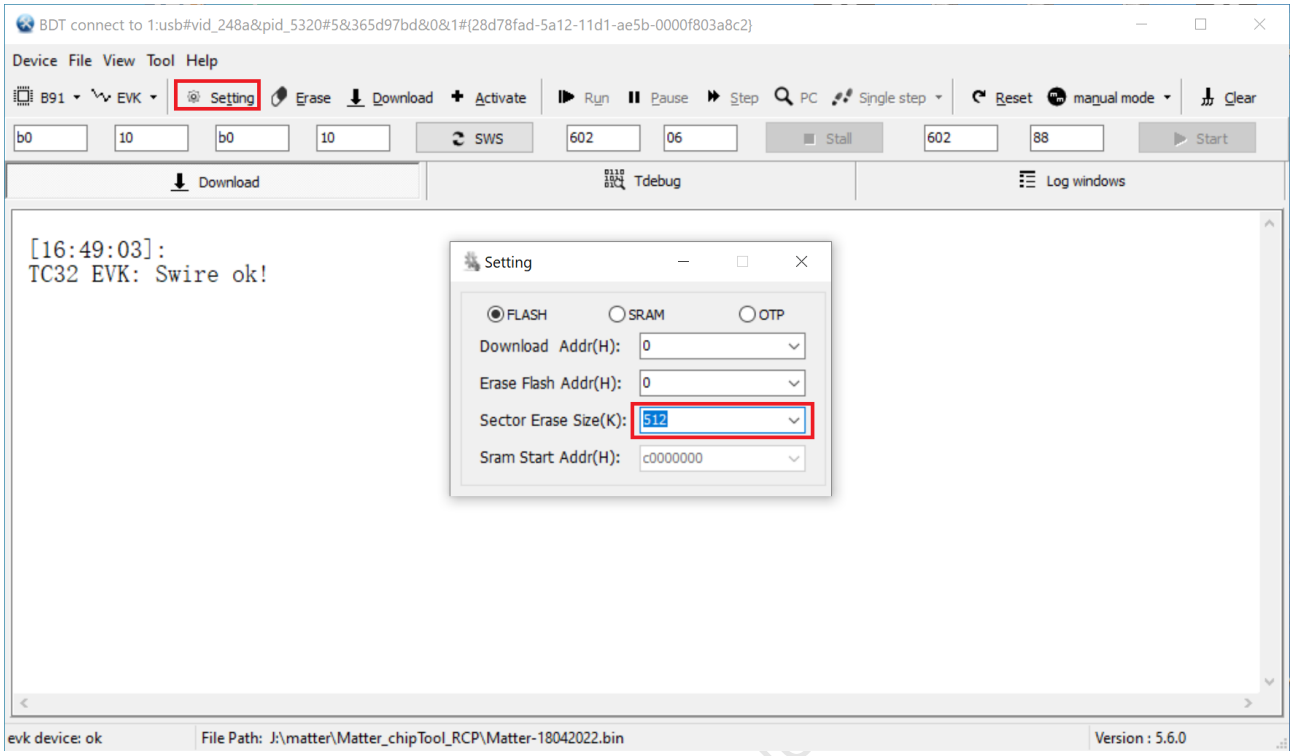

Figure 3.4: 默认的 Flash 擦除的区域大小

将 Flash 擦除的区域大小设置为"2040"，如下图所示：

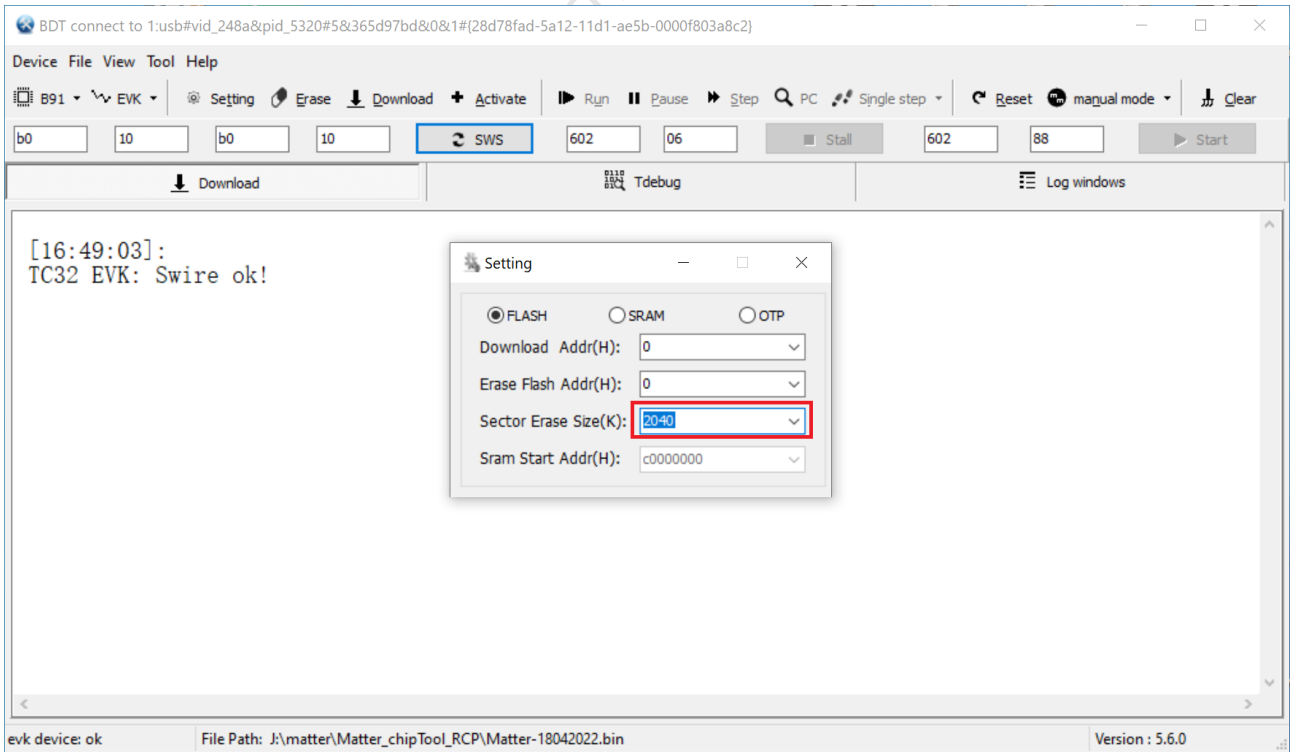

Figure 3.5: 设置 Flash 擦除的区域大小

> 注意：对于外挂 2MB Flash 的 TLSR9518 开发板，Flash 最后的 8kB 空间预留用于保存重要的 SoC 信息，因此最多可以擦除 2040kB 的 Flash 区域。

5. 点击工具栏中的"Erase"按钮，等待 Flash 擦除操作完成。

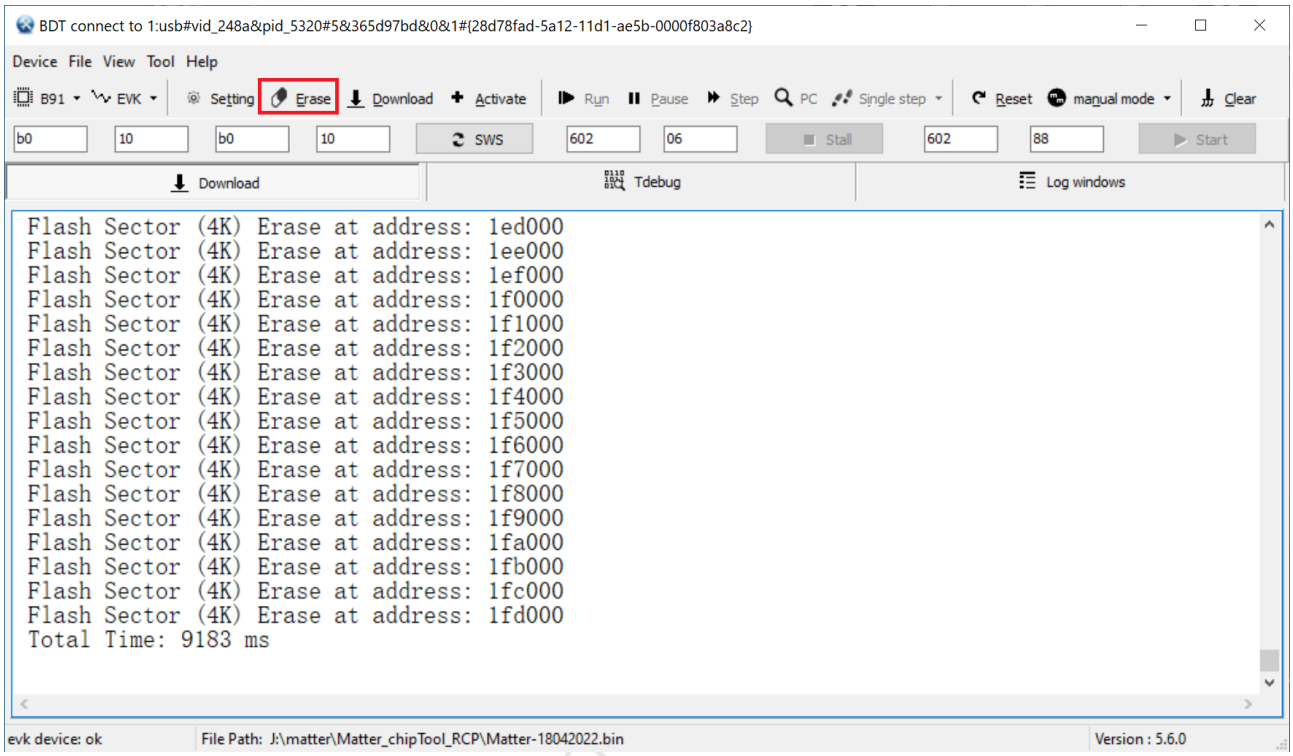

Figure 3.6: Flash 擦除操作

6. 选择需要烧录的 BIN 文件。点击"File"菜单里的"Open"子菜单，在弹出的文件选择对话框中选中需要烧录的 BIN 文件。选中后的 BIN 文件将显示在底部的状态栏中。

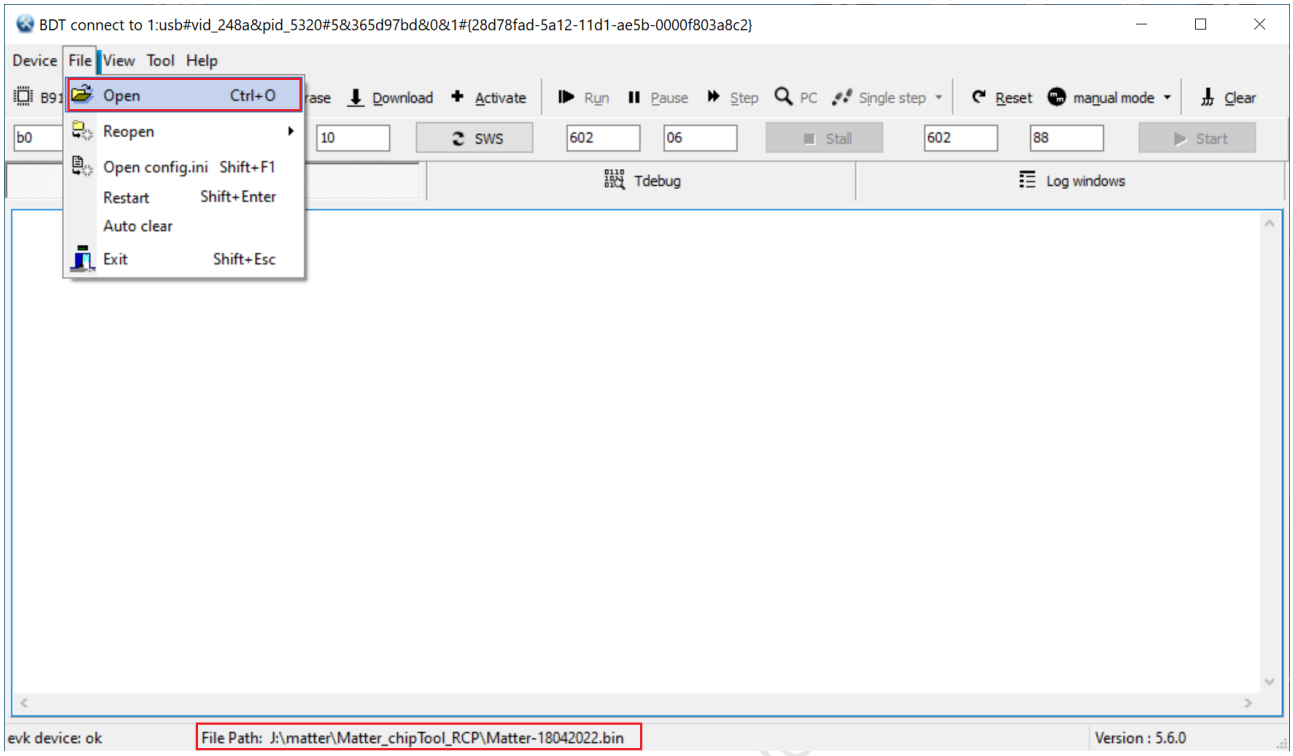

Figure 3.7: 选择 BIN 文件

7. 点击工具栏中的"Download"按钮，等待 Flash 烧录完成。

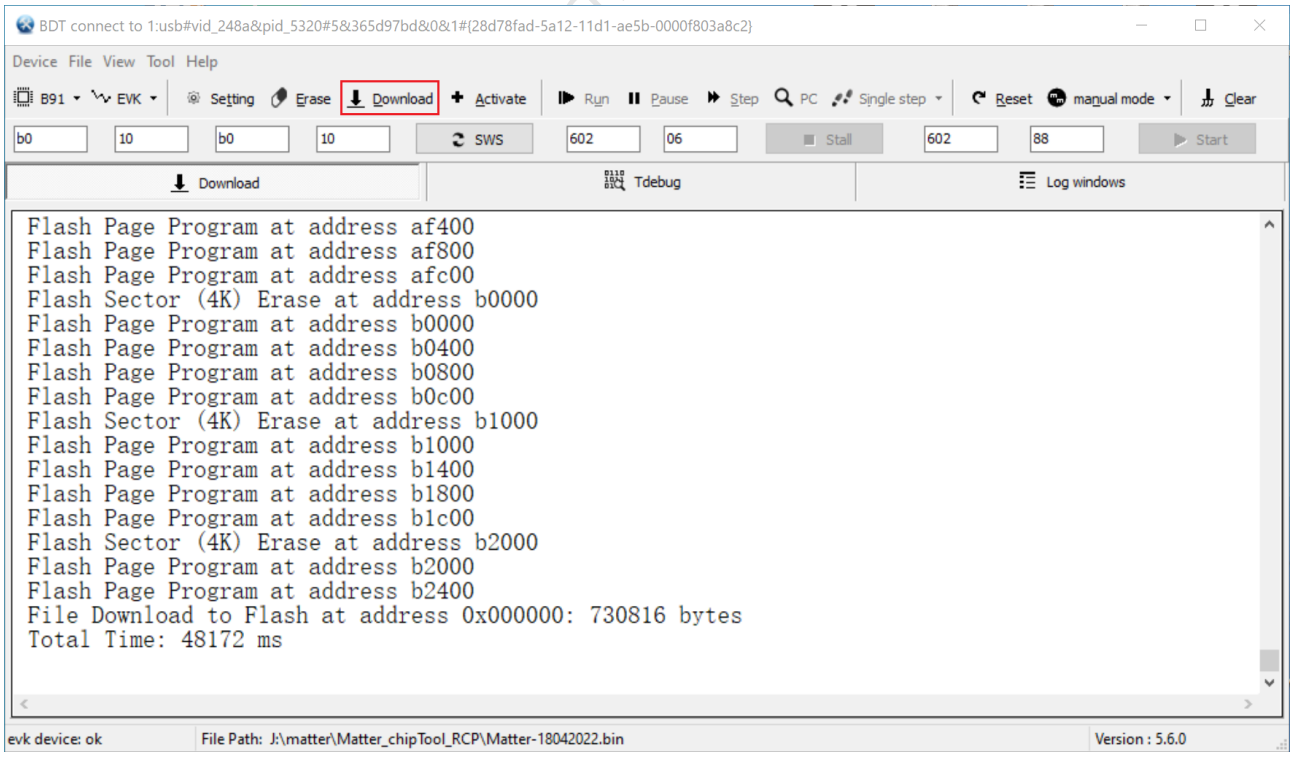

Figure 3.8: Flash 烧录操作

对于更多命令的详细说明，请参考 `BDT\release_v5.6.0\doc` 文件夹下的文档。

3.4.4 BDT 在 Windows 平台上的常见问题

1. 烧录器插入电脑后，可以被 Windows 设备管理器正确识别，但是烧录工具软件没有识别到，即在系统标题栏中看不到烧录器的设备信息（见图 3.3）。请检查电脑是否用了 AMD 平台的处理器，换一台 Intel 平台处理器的电脑重新尝试。
2. 最常见的问题是，在点击工具栏中的"SWS"按钮后，出现下图中的错误信息：

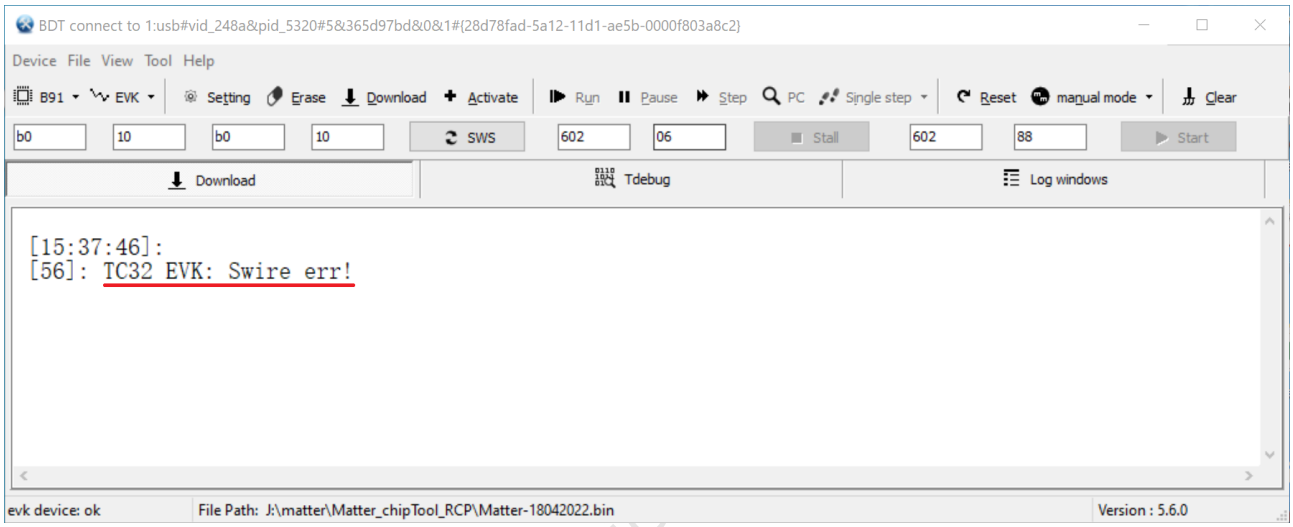

Figure 3.9: 按下 SWS 按钮后的错误信息

主要有两种原因：

- 硬件连接不正确。请参照前面的说明仔细核对所有的硬件连接，确认没有遗漏或错误的连接。
- 烧录器的固件版本太低。请按照以下步骤查看烧录器固件的版本：

1. 点击 Help 菜单下的 Upgrade 子菜单。
2. 在弹出的 Upgrade Evk 窗口中，点击"Read FW version"按钮。在旁边的"Firmware Version"区域将会显示烧录器的固件版本号，例如下图所示。如果固件版本号低于 V3.4，可以确认是由于固件版本过低导致了通讯错误。请继续下面的步骤去完成固件升级。

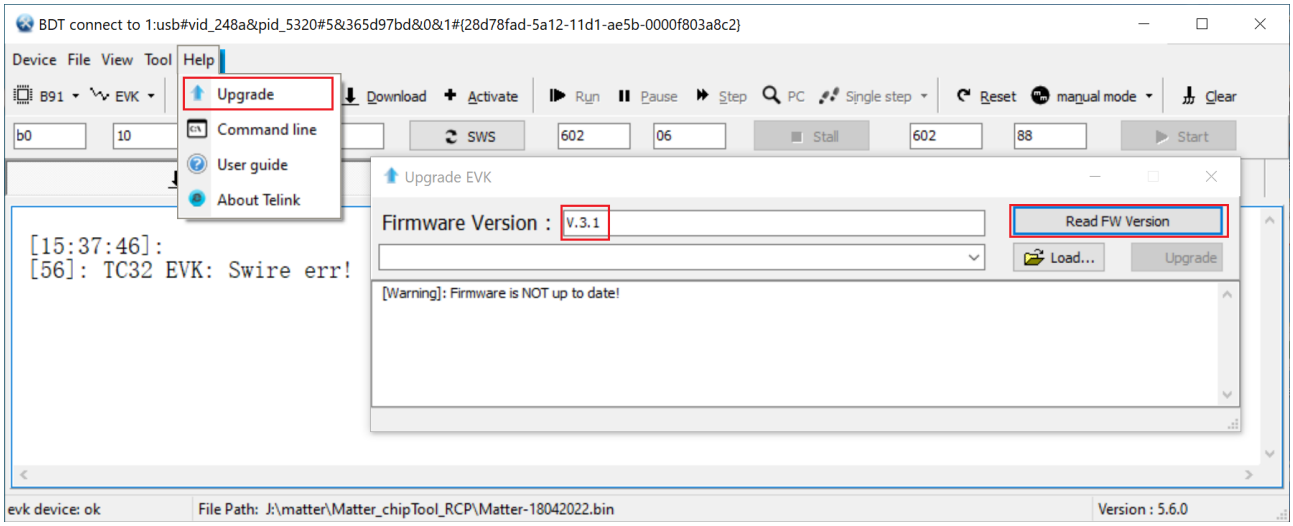

Figure 3.10: 查看烧录器固件版本

3. 点击窗口中的"Load..."按钮，在 BDT 工具所在目录下的 config 目录下的 fw 子目录找到最新的烧录器固件，如下图中的 Firmware_v3.5.bin 文件。

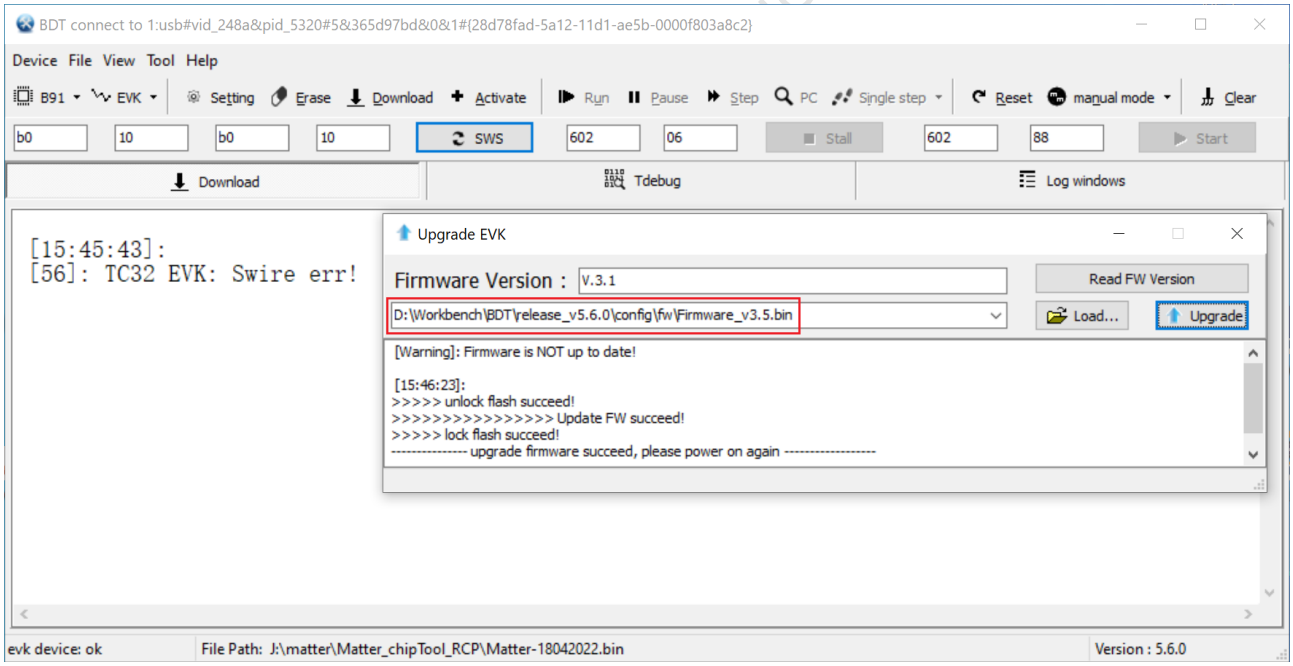

Figure 3.11: 升级烧录器固件

4. 点击 Upgrade 按钮完成烧录器固件升级。
5. 插拔烧录器的 USB 线，使烧录器重新上电。

34.5 LinuxBDT 在 Ubuntu 平台上的使用步骤

在 Ubuntu 20.04 上，打开终端，下载 `LinuxBDT`。

```
wget http://wiki.telink-semi.cn/tools_and_sdk/Tools/BDT/LinuxBDT.tar.bz2
```

对下载到本地的 LinuxBDT.tar.bz2 进行解压缩：

```
tar -xvf LinuxBDT.tar.bz2
```

打开 LinuxBDT 文件夹，可以使用图形化的烧录调试工具 bdt_gui 或者命令行的 bdt，去操纵 Burning Evk 对 B91 EVB 进行烧录。bdt_gui 的使用方法和 windows 平台下的 BDT.exe 相似，因此下面主要介绍命令行 bdt 的使用步骤。

1. 用 Mini-USB 线连接烧录器到电脑的 USB 口
2. 检测 Burning Evk 与目标板 B91 EVB 的连接是否正常

   进入烧录工具 LinuxBDT 的根目录下的，执行下面命令。

   ```
   ./bdt 9518 sws
   ```

   可以查看已连接的烧录器和 B91 EVB，如下图：

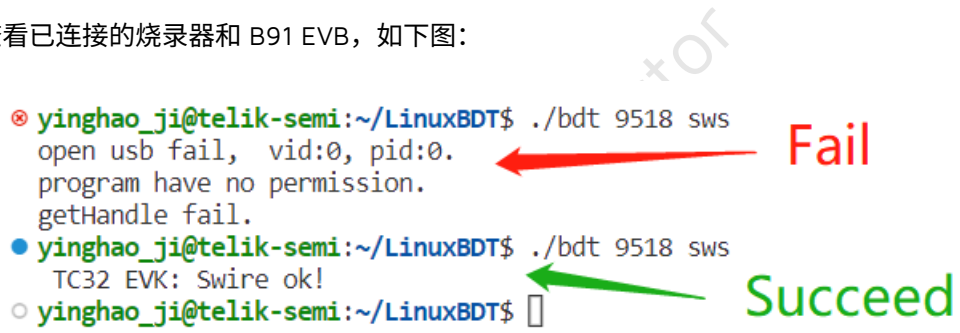

Figure 3.12: 检测连接

3. 激活 MCU

   B91 EVB 上已有固件在运行时，直接使用 Burning Evk 烧录器进行擦除或者烧录，可能会遇到 "Swire err!" 错误。为了避免类似错误，请执行以下命令先激活 MCU。

   ```
   ./bdt 9518 ac
   ```

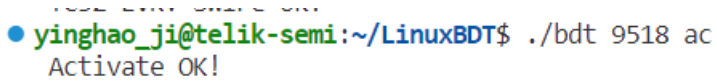

Figure 3.13: 激活 MCU

4. 擦除 Flash 扇区

   为避免旧的固件所占的尺寸大于新的固件，导致新刷入的固件后，Flash 上存在以往的数据残留，请先执行下面的擦除命令。

   ```
   ./bdt 9518 wf 0 -s 2040k -e
   ```

- MCU 种类 – 9518

- 可选命令 – `wf` (for write flash)
- 将要被擦除的起始地址 – `0`
- 可选命令 – `-s` (指定将要擦除的扇区大小)
- 将要擦除的扇区大小 – `512k` (64k, 128k, 256k, 512k... B91 EVB 最多 2040k)
- 可选命令 – `-e` (erase flash)

  等待约数十秒，擦除成功后，会显示：

```
yinghao_ji@telik-semi:~/LinuxBDT$ ./bdt 9518 wf 0 -s 2040k -e
EraseSectorsize...
Total Time: 26034 ms
```

Figure 3.14: 擦除 Flash

5. 将固件下载到 MCU 的 Flash 中

```
./bdt 9518 wf 0 -i \ ~/workspace/matter/binaries_220922_718934487/lighting-app.bin
```

   - MCU 种类– `9518`
   - 可选命令 – `wf` (for write flash)
   - 将要被写入的起始地址 – `0`
   - 可选命令 – `-i` (指定将要写入的固件)
   - 将要写入的固件的路径 – 如这里的 `~/workspace/matter/binaries_220922_718934487/lighting-app.bin`

   执行上述命令可以所选的固件刷入 Flash 当中，成功后的输出如下：

```
[99%]Flash Bytes Program at address ba400
[99%]Flash Bytes Program at address ba500
[100%]Flash Bytes Program at address ba600
 File Download to Flash at address 0x000000: 763560 bytes
 Total Time: 187059 ms
yinghao_ji@telik-semi:~/LinuxBDT$ ./bdt 9518 wf 0 -i ~/workspace/matter/binaries_220922_718934487/lighting-app.bin
```

Figure 3.15: 烧入固件

6. 重置 MCU

```
./bdt 9518 rst -f
```

   - MCU 种类– `9518`
   - 可选命令 – `rst` (for reset)
   - 可选命令 – `-f` (for Flash or OTP)

   使用上述重置命令成功后的输出如下：

```
yinghao_ji@telik-semi:~/LinuxBDT$ ./bdt 9518 rst -f
Total Time: 69 ms
reset mcu
```

Figure 3.16: 重置 MCU

重置 B91 EVB 评估板后，整个 MCU 将被重新上电，随即开始运行刚刚烧录的固件。

对于更多命令的详细说明，请参考 `LinuxBDT/doc` 文件夹下的文档。

3.5 边界路由配置

3.5.1 边界路由需要的硬件和固件

OTBR 边界路由需要由以下两个部分组成：

1. **Raspberry Pi 3B+**：树莓派 3B 或更高版本。

   树莓派的预置镜像可以在下面的联想网盘中找到：

   https://console.box.lenovo.com/l/NJioaS

   提取码：surz

2. **Radio Co-Processor**：RCP 负责 Thread 通讯，由一个 TLSR9518ADK80D 评估板来实现。

   RCP 固件也在上述的网盘地址中，解压后即可。

3.5.2 边界路由的树莓派镜像烧录

在 Windows 下可以通过 Win32 Disk Image 进行树莓派镜像的烧录，具体步骤如下：

1. 下载 Win32 Disk Image 的安装包。下面是安装包的下载地址：

   https://sourceforge.net/projects/win32diskimager/files/Archive/

2. 安装 Win32 Disk Image。如果上面的网址中下载了 zip 文件，则不需要安装，只需要把压缩文件解压到电脑某个目录中。

3. 把 Mirco SD 卡插入读卡器中，再将读卡器插入 Windows 电脑。

   注意：请确保用到的 Mirco SD 卡的容量是 16G 或以上。

4. 打开 Win32 Disk Image。在"映像文件"中选择树莓派的镜像文件，在"设备"中选择 Micro SD 卡所在的驱动盘符。

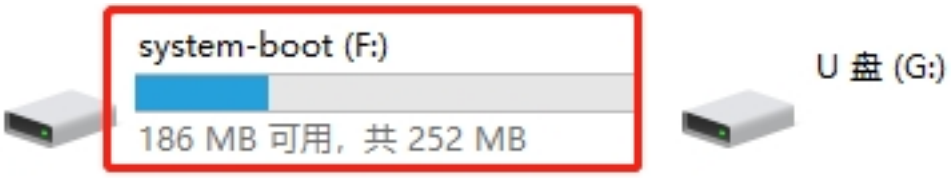

Figure 3.17: 选择盘符

5. 点击写入按钮，等待镜像写入完成。

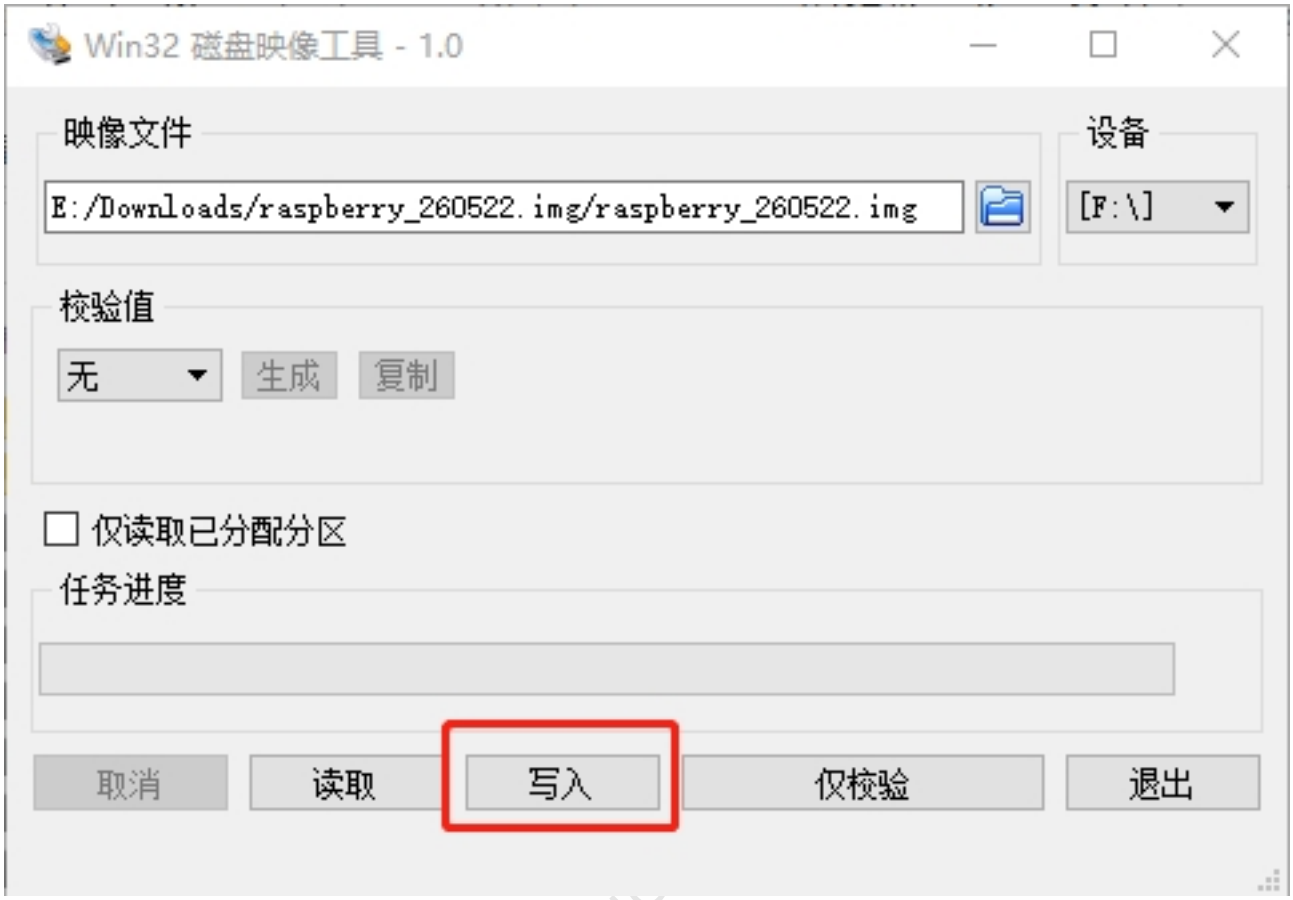

Figure 3.18: 写入镜像

3.5.3 树莓派和 RCP 的连接方式

按照下面的管脚位置进行接线：

RCP	树莓派
TX(PB2 pin 18 of J34)	RXD1(GPIO15 pin 10)
RX(PB3 pin 15 of J34)	TXD1(GPIO14 pin 8)
GND (e.g. pin 23 of J50 or pin 3 of J56)	GND (e.g. pin 6 or 9)

> 注意：波特率为 57600

RCP 和树莓派上的管脚位置可以参考下图：

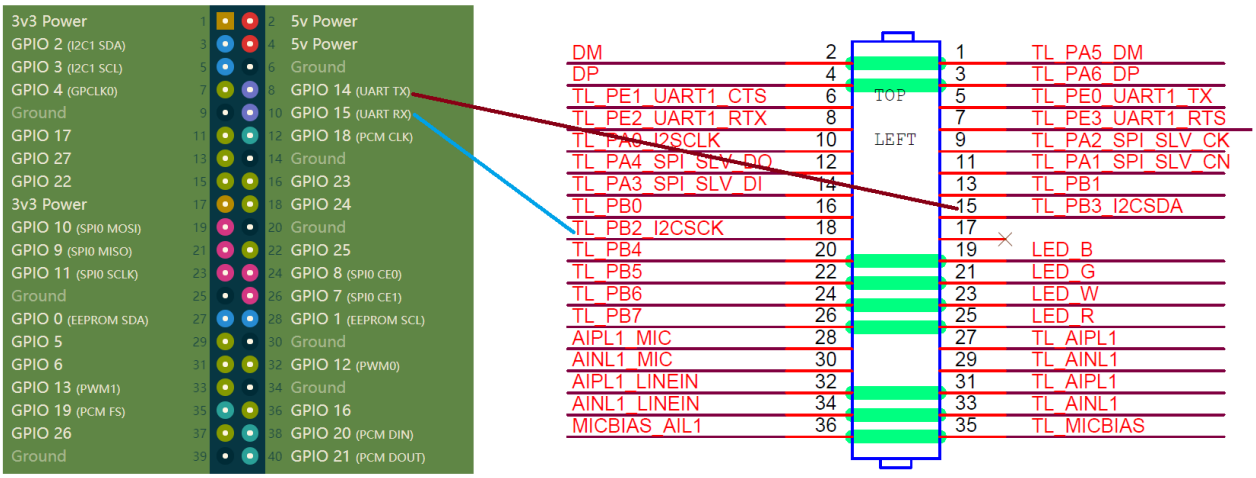

Figure 3.19: RCP-树莓派管脚位置

3.6 Thread **网络建立和通过** BLE **配网**

3.6.1 **建立** Thread **网络并获取** Dataset

1. 在浏览器中输入树莓派的 IP 地址，点击 Form 按钮，默认设置不用更改，点击 FORM 建立 Thread 网络。

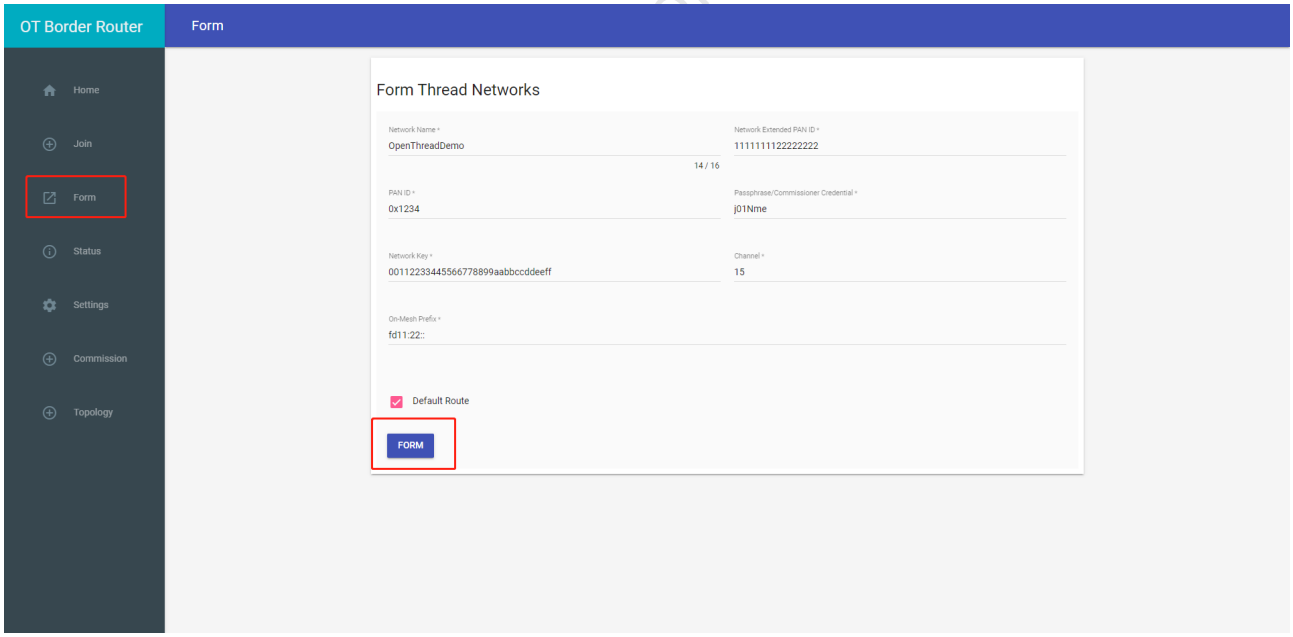

Figure 3.20: 建立 Thread 网络

2. Thread 网络建立后可以在 Status 下查看状态

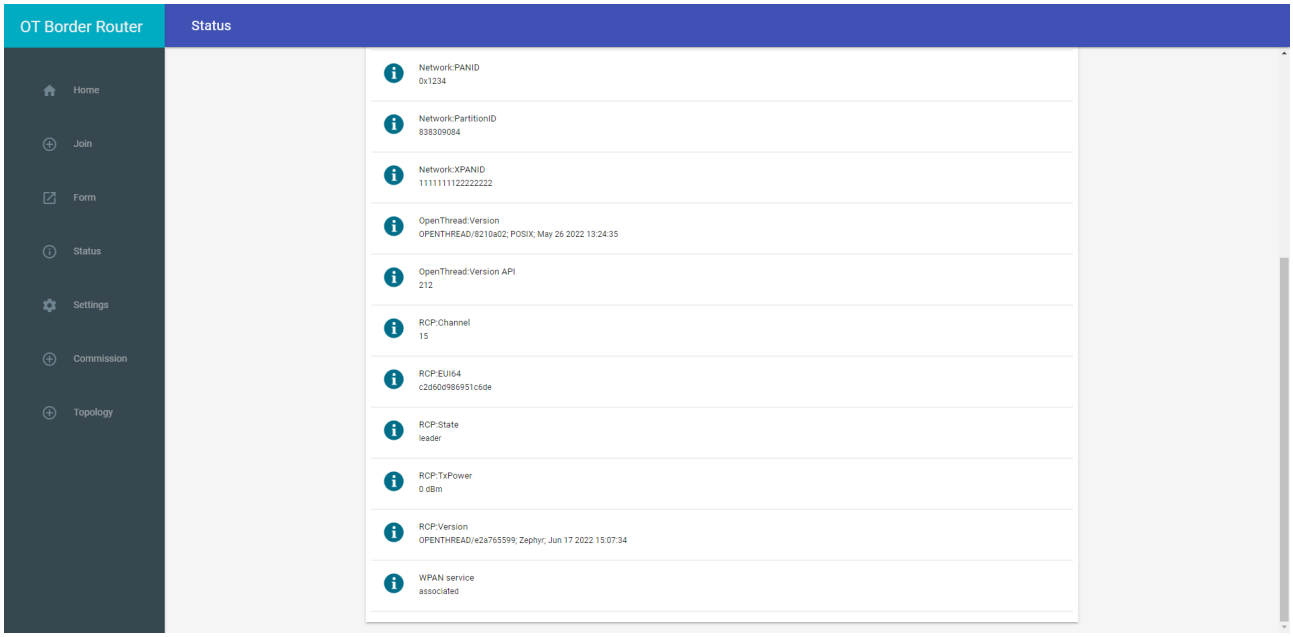

Figure 3.21: Thread 网络建立后状态

3. 获取 DATASET。请以 SSH 方式登录树莓派（预置镜像中的用户名 username：pi，密码 password：raspberry），执行以下命令：

```
$sudo ot-ctl dataset active -x
```

DATASET 是类似于以下形式的一串十六进制的字符串，将其保存好。

```
pi@raspberrypi:~ $ sudo ot-ctl dataset active -x
0e080000000000010000000300000f35060004001fffe00208111111112222222220708fd0b448cf7
918bcf0510001122334455667788 99aabbccddeeff03094f5444323730353232010212340410dc2f
4eef10c505a7f22fd2a35b11e73f0c0402a0f7f8
Done
```

Figure 3.22: DATASET

> **注意**：每次形成新的 Thread 网络，上面的 DATASET 将会被重新生成。即使每次生成 Thread 网络所设置的参数相同，其中间的部分...0708fd0b448cf7918bcf051000...也会不同。

3.6.2 配网过程

1. 在主机上进行配网之前，请检查主机与树莓派之间的网络连接状态。
   - 如果主机与树莓派之间是由带防火墙的路由器做转发，暂时关闭路由器上的防火墙，尤其是其禁止端口监听、端口扫描等功能。
   - 如果使用运营商的光猫作为路由，可能会导致 mDNS 服务无法发现的错误，尝试将主机与树莓派用仅开启 DHCP 服务的其他路由器进行网线直连。
   - 确保主机是独立的 Ubuntu 主机；若使用 Windows 上的 VirtualBox 等虚拟机充当主机，则需要给它提供并配置额外的蓝牙适配器。

2. 检查 Matter 固件版本与 chip-tool 的是否相符

   编译 Matter 设备的固件和 chip-tool 需要相同的 Zephyr 环境，否则进行配网时会出错。

   > 重要提醒：若要使用自己构建的 chip-tool 和 Matter 设备的固件，必须保证它们使用了相同的 commit 的 connectedhomeip 工程目录进行构建，以避免出现兼容性问题。

3. 在主机上的 shell 中配置好以下命令：

```
./chip-tool pairing ble-thread ${NODE_ID} hex:${DATASET} ${PIN_CODE} ${DISCRIMINATOR}
```

   > 注意：运行 chip-tool 需退出镜像，并检查 chip-tool 的执行权限。

   NODE_ID 可以是 RCP 初始化之后，未使用过的任何非零值，chip-tool 将使用它来操作特定的 Matter 设备。

   DATASET 即为树莓派上获取的字符串。

   示例：

```
ubuntu@ubuntu2004:~/workspace/chip-tool$ NODE_ID=1252
ubuntu@ubuntu2004:~/workspace/chip-tool$ DATASET=0e080000000000010000000300000f35060004001fffe00208111111112222222220708fd1ca5c6b42145030510001
56d6f0102123404104 45f2b5ca6f2a93a55ce570a70efeecb0c0402a0f7f8
ubuntu@ubuntu2004:~/workspace/chip-tool$ PIN_CODE=20202021
ubuntu@ubuntu2004:~/workspace/chip-tool$ DISCRIMINATOR=3840
ubuntu@ubuntu2004:~/workspace/chip-tool$ ./chip-tool_ubuntu_06072022 pairing ble-thread ${NODE_ID} hex:${DATASET} ${PIN_CODE} ${DISCRIMINATOR}
```

Figure 3.23: 配对和配网

4. Matter 设备上电后，红灯闪烁，进入 BLE 广播状态，在主机上的 shell 中输入上面命令并运行，会让 Matter 设备与 RCP 所在的边界路由开始配对并配网。

   这个过程会持续一段时间，如果一切顺利，Matter 设备加入 Thread 网络后，你将能够从主机的 shell 中看到类似下面的信息：

```
[1657851241.806316][3215:3220] CHIP:DMG: },
[1657851241.806336][3215:3220] CHIP:DMG: Received Command Response Data, Endpoint=0 Cluster=0x0000_0030 Command=0x0000_0005
[1657851241.806366][3215:3220] CHIP:CTL: Received CommissioningComplete response, errorCode=0
[1657851241.806381][3215:3220] CHIP:CTL: Successfully finished commissioning step 'SendComplete'
[1657851241.806389][3215:3220] CHIP:CTL: Commissioning stage next step: 'SendComplete' -> 'Cleanup'
[1657851241.806438][3215:3220] CHIP:CTL: Performing next commissioning step 'Cleanup'
[1657851241.806469][3215:3220] CHIP:DIS: Closing all BLE connections
[1657851241.806518][3215:3220] CHIP:CTL: Successfully finished commissioning step 'Cleanup'
[1657851241.806542][3215:3220] CHIP:TOO: Device commissioning completed with success
[1657851241.806593][3215:3220] CHIP:DMG: ICR moving to [AwaitingDe]
[1657851241.806608][3215:3220] CHIP:EM: Sending Standalone Ack for MessageCounter:188298949 on exchange 24262i
[1657851241.806636][3215:3220] CHIP:IN: Prepared secure message 0x7f53d9c55080 to 0x0000000000004E4 (1) of type 0x10 and protocolId (0, 0) on exchange 24262i with MessageCounter:118995674.
[1657851241.806652][3215:3220] CHIP:IN: Sending encrypted msg 0x7f53d9c55080 with MessageCounter:118995674 to 0x0000000000004E4 (1) at monotonic time: 000000000001BB54 msec
[1657851241.806714][3215:3220] CHIP:EM: Flushed pending ack for MessageCounter:188298949 on exchange 24262i
[1657851241.806881][3215:3215] CHIP:CTL: Shutting down the commissioner
[1657851241.806945][3215:3215] CHIP:CTL: Shutting down the controller
[1657851241.806959][3215:3215] CHIP:CTL: Shutting down the commissioner
[1657851241.806973][3215:3215] CHIP:CTL: Shutting down the controller
[1657851241.806987][3215:3215] CHIP:IN: Expiring all connections for fabric 1!!
[1657851241.806992][3215:3215] CHIP:IN: SecureSession[0x7f53c40105f0]: MarkForEviction Type:2 LSID:2789
[1657851241.807002][3215:3215] CHIP:SC: SecureSession[0x7f53c40105f0]: Moving from state 'kActive' --> 'kPendingEviction'
[1657851241.807006][3215:3215] CHIP:IN: SecureSession[0x7f53c40105f0]: Released - Type:2 LSID:2789
[1657851241.807012][3215:3215] CHIP:FP: Forgetting fabric 0x1
[1657851241.807028][3215:3215] CHIP:TS: Pending Last Known Good Time: 2022-07-05T23:17:37
[1657851241.807145][3215:3215] CHIP:TS: Previous Last Known Good Time: 2022-07-05T23:17:37
[1657851241.807160][3215:3215] CHIP:TS: Reverted Last Known Good Time to previous value
[1657851241.807190][3215:3215] CHIP:CTL: Shutting down the commissioner
[1657851241.807215][3215:3215] CHIP:CTL: Shutting down the controller
[1657851241.807228][3215:3215] CHIP:IN: Expiring all connections for fabric 2!!
[1657851241.807240][3215:3215] CHIP:FP: Forgetting fabric 0x2
[1657851241.807253][3215:3215] CHIP:TS: Pending Last Known Good Time: 2022-07-05T23:17:37
[1657851241.807317][3215:3215] CHIP:TS: Previous Last Known Good Time: 2022-07-05T23:17:37
[1657851241.807328][3215:3215] CHIP:TS: Reverted Last Known Good Time to previous value
[1657851241.807343][3215:3215] CHIP:CTL: Shutting down the commissioner
[1657851241.807386][3215:3215] CHIP:CTL: Shutting down the controller
[1657851241.807404][3215:3215] CHIP:IN: Expiring all connections for fabric 3!!
[1657851241.807415][3215:3215] CHIP:FP: Forgetting fabric 0x3
[1657851241.807435][3215:3215] CHIP:TS: Pending Last Known Good Time: 2022-07-05T23:17:37
[1657851241.807500][3215:3215] CHIP:TS: Previous Last Known Good Time: 2022-07-05T23:17:37
[1657851241.807514][3215:3215] CHIP:TS: Reverted Last Known Good Time to previous value
[1657851241.807526][3215:3215] CHIP:CTL: Shutting down the System State, this will teardown the CHIP Stack
[1657851241.807585][3215:3215] CHIP:DMG: IM WH moving to [Uninitialized]
[1657851241.807602][3215:3215] CHIP:DMG: IM WH moving to [Uninitialized]
[1657851241.807610][3215:3215] CHIP:DMG: IM WH moving to [Uninitialized]
[1657851241.807617][3215:3215] CHIP:DMG: IM WH moving to [Uninitialized]
[1657851241.807625][3215:3215] CHIP:DMG: All ReadHandler-s are clean, clear GlobalDirtySet
[1657851241.807645][3215:3215] CHIP:BLE: BleConnectionDelegate::CancelConnection is not implemented.
[1657851241.807756][3215:3215] CHIP:DL: writing settings to file (/tmp/chip_counters.ini-EFpUkj)
[1657851241.808081][3215:3215] CHIP:DL: renamed tmp file to file (/tmp/chip_counters.ini)
[1657851241.808175][3215:3215] CHIP:DL: NVS set: chip-counters/total-operational-hours = 0 (0x0)
[1657851241.808185][3215:3215] CHIP:DL: Inet Layer shutdown
[1657851241.808190][3215:3215] CHIP:DL: BLE shutdown
[1657851241.808513][3215:3215] CHIP:DL: System Layer shutdown
[1657851241.808569][3215:3215] CHIP:IN: SecureSession[0x7f53c4001a30]: Released - Type:1 LSID:2788
ubuntu@ubuntu2004:~/workspace/chip-tool$
```

Figure 3.24: 配网成功

3.7 使用 chip-tool 进行控制

在配网成功之后，可以使用 chip-tool 对 Matter 设备进行控制，主要的几个控制命令如下。

3.7.1 开关灯

```
./chip-tool onoff command_name [param1 param2 ...]
```

- command_name 是命令的名字，此处开关灯可以用到 on（开灯）、off（关灯）、toggle（切换状态）
- [param1 param 2 ...] 是所使用的多个参数，此处开关灯用到的是节点 ID <node_id> 和端点 ID <endpoint_id>
- <node_id> 是之前设备进行配网时使用的节点 ID, 可以用 **${NODE_ID}** 这样的 shell 变量表示
- <endpoint_id> 是实现了 OnOff Cluster 的端点的 ID

开灯命令示例：

```
./chip-tool onoff on ${NODE_ID} 1
```

关灯命令示例：

```
./chip-tool onoff off ${NODE_ID} 1
```

灯泡状态翻转命令示例：

```
./chip-tool onoff toggle ${NODE_ID} 1
```

3.7.2 查看亮灯情况

标准的读取属性的命令为：

```
./chip-tool onoff read attribute-name [param1 param2 ...]
```

- attribute-name 是要读取的属性名
- [param1 param 2 ...] 是所使用的多个参数

读取亮灯情况的命令示例：

```
./chip-tool onoff read on-off ${NODE_ID} 1
```

- 属性名为 on-off
- 此处的两个参数为节点 ID 和端点 ID

3.7.3 查看亮度

同样使用读取属性的命令，标准的读取属性的命令为：

```
./chip-tool onoff read attribute-name [param1 param2 ...]
```

- attribute-name 是要读取的属性名
- [param1 param 2 ...] 是所使用的多个参数

读取亮度属性的命令示例：

```
./chip-tool levelcontrol read current-level ${NODE_ID} 1
```

- 属性名为 current-level
- 此处的两个参数为节点 ID 和端点 ID

3.74 改变亮度

标准的命令为：

```
./chip-tool levelcontrol move-to-level <level> <transition_time> <option_mask> <option_override>
↪  <node_id> <endpoint_id>
```

- <level> 是亮度值
- <transition_time> 是过渡时间
- <option_mask> 是 option mask
- <option_override> 是 option override
- <node_id> 是之前设备进行配网时使用的节点 ID
- <endpoint_id> 是实现了 LevelControl Cluster 的端点的 ID

改变亮度的示例命令：

```
./chip-tool levelcontrol move-to-level 32 0 0 0 ${NODE_ID} 1
```

3.7.5 连接灯和开关

对于配网在同一 Matter 网络的 light-switch 开关和 lighting 灯泡，可以通过以下步骤进行绑定。

使用 chip-tool 将单一设备绑定：

1. 使用 chip-tool 将访问控制列表 ACL(Accsee Control List) 写入 accesscontrol Cluster，为灯泡设备添加合适的访问控制列表，执行如下命令:

```
./chip-tool accesscontrol write acl <acl_data> <node_id> <endpoint_id>
```

- <acl_data> 是格式为 JSON 数组的 ACL 数据。
- <node_id> 是要接收 ACL 的节点的 ID。
- <endpoint_id> 是实现 accesscontrol Cluster 的端点的 ID。

命令示例：

```
./chip-tool accesscontrol write acl '[{"fabricIndex": 1, "privilege": 5, "authMode": 2,
↪  "subjects": [112233], "targets": null}, {"fabricIndex": 1, "privilege": 3, "authMode":
↪  2, "subjects": [<light-switch-node-id>], "targets": [{"cluster": 6, "endpoint": 1,
↪  "deviceType": null}, {"cluster": 8, "endpoint": 1, "deviceType": null}]}]' <lighting-
↪  node-id> 0
```

在上面的命令中：

- <lighting-node-id> 是之前灯泡设备进行配网时使用的节点 ID, 可以用 **${NODE_ID}** 这样的 shell 变量表示。
- <light-switch-node-id> 是之前开关设备进行配网时使用的节点 ID，此处需要用数字表示。
- {"fabricIndex": 1, "privilege": 5, "authMode": 2, "subjects": [112233], "targets": null} 是和 chip-tool 通讯的访问控制列表。
- {"fabricIndex": 1, "privilege": 5, "authMode": 2, "subjects": [<light- switch- node-id>], "targets": [{"cluster": 6, "endpoint": 1, "deviceType": null}, {"cluster": 8, "endpoint": 1, "deviceType": null}]} 是一个绑定的访问控制列表（cluster NO.6 是 On/Off cluster，cluster NO.8 是 Level Control cluster)。

此命令为照明应用设备添加权限，允许其接收来自开关设备的命令。

2. 将绑定表添加到开关以通知设备端点：

```
./chip-tool binding write binding  <binding_data> <node_id> <endpoint_id>
```

- <binding_data> 是格式为 JSON 数组的绑定数据。
- <node_id> 是要接受绑定的节点的 ID。
- <endpoint_id> 是实现 binding Cluster 的端点的 ID。

命令示例：

```
./chip-tool binding write binding '[{"fabricIndex": 1, "node": <lighting-node-id>,
↪  "endpoint": 1, "cluster": 6}, {"fabricIndex": 1, "node": <lighting-node-id>,
↪  "endpoint": 1, "cluster": 8}]' <light-switch-node-id> 1
```

在上面的命令中：

- <light-switch-node-id> 是之前开关设备进行配网时使用的节点 ID，可以用 **${SWITCH_NODE_ID}** 这样的 shell 变量表示。
- <lighting-node-id> 是之前灯泡设备进行配网时使用的节点 ID，此处需要用数字表示。

- {"fabricIndex": 1, "node": <lighting-node-id>, "endpoint": 1, "cluster": 6} 是绑定 On/Off Cluster。
- {"fabricIndex": 1, "node": <lighting-node-id>, "endpoint": 1, "cluster": 8} 是绑定 Level Control Cluster。

使用 chip-tool 绑定多个设备：

1. 通过运行以下命令将开关设备添加到多播组：

```
./chip-tool tests <test_name>
```

   - <test_name> 是特定测试的名称

   命令示例：

```
./chip-tool tests TestGroupDemoConfig --nodeId <light-switch-node-id>
```

   - <light-switch-node-id> 是之前开关设备进行配网使用的节点 ID，可以用 **${SWITCH_NODE_ID}** 这样的 shell 变量表示。

2. 通过对每个灯泡应用下面的命令将所有灯泡加入到同一个组中，对每个灯泡使用相应的 <lighting-node-id> （用户进行配网时定义的节点 ID）：

```
./chip-tool tests TestGroupDemoConfig --nodeId <lighting-node-id>
```

   - <lighting-node-id> 可以用 **${NODE_ID}** 这样的 shell 变量表示。

3. 添加绑定命令：

```
./chip-tool binding write binding  <binding_data> <node_id> <endpoint_id>
```

   - <binding_data> 是格式为 JSON 数组的绑定数据。
   - <node_id> 是要接受绑定的节点的 ID。
   - <endpoint_id> 是实现 binding Cluster 的端点的 ID。

   命令示例：

```
./chip-tool binding write binding '[{"fabricIndex": 1, "group": 257}]' <light-switch-node-
↪  id> 1
```

   - <light-switch-node-id> 可以用 **${SWITCH_NODE_ID}** 这样的 shell 变量表示。

3.8 体验 OTA 功能

3.8.1 启用 OTA 模块

OTA 模块只在 ota-requestor-app 示例中默认启用，若要在其他 Telink Matter 示例中启用 OTA 模块，需要按照以下步骤：

- 在相应的 prj.conf 配置文件中设置 "CONFIG_CHIP_OTA_REQUESTOR=y"

在通过以上配置启用 OTA 模块后，编译得到固件：

- zephyr.bin - 烧录到开发板上的 bin 文件
- zephyr-ota.bin - 用于 OTA Provide 的 bin 文件

所有的 bin 文件都拥有相同的 SW 版本，为了测试 OTA，"zephyr-ota.bin"应该有比基础的 SW 更高的 SW 版本。在对应的 `prj.conf` 配置文件中设置 "CONFIG_CHIP_DEVICE_SOFTWARE_VERSION=2"

3.8.2 OTA 的用法

1. 构建 Linux OTA Provider

```
./scripts/examples/gn_build_example.sh examples/ota-provider-app/linux out/ota-provider-app
↪  chip_config_network_layer_ble=false
```

2. 运行 Linux OTA Provider

```
./chip-ota-provider-app -f zephyr-ota.bin
```

此处的 zephyr-ota.bin 是用来升级的固件。

3. 打开另一个终端，用 chip-tool 准备 Linux OTA Provider

```
./chip-tool pairing onnetwork ${OTA_PROVIDER_NODE_ID} 20202021
```

在这个命令中：

- ${OTA_PROVIDER_NODE_ID} 是 Linux OTA Provider 的 node id，类似于 lighting-app 的 **NODE_ID**，应该是一个之前没使用过的非零值。

4. 配置 ota-provider-app 的 ACL 来允许访问

```
./chip-tool accesscontrol write acl '[{"fabricIndex": 1, "privilege": 5, "authMode": 2,
↪  "subjects": [112233], "targets": null}, {"fabricIndex": 1, "privilege": 3, "authMode":
↪  2, "subjects": null, "targets": null}]' ${OTA_PROVIDER_NODE_ID} 0
```

- 此处的 ${OTA_PROVIDER_NODE_ID} 是 Linux OTA Provider 的 node id。

5. 使用 chip-tool 通知 ota-provider-app 开始 OTA 进程。

```
./chip-tool otasoftwareupdaterequestor announce-ota-provider ${OTA_PROVIDER_NODE_ID} 0 0 0
↪  ${DEVICE_NODE_ID} 0
```

在这个命令中：

- ${OTA_PROVIDER_NODE_ID} 是 Linux OTA Provider 的 node id
- ${DEVICE_NODE_ID} 是配对设备的 node id

4 如何基于/利用 Github 上的开源 SDK 进行开发

4.1 架构

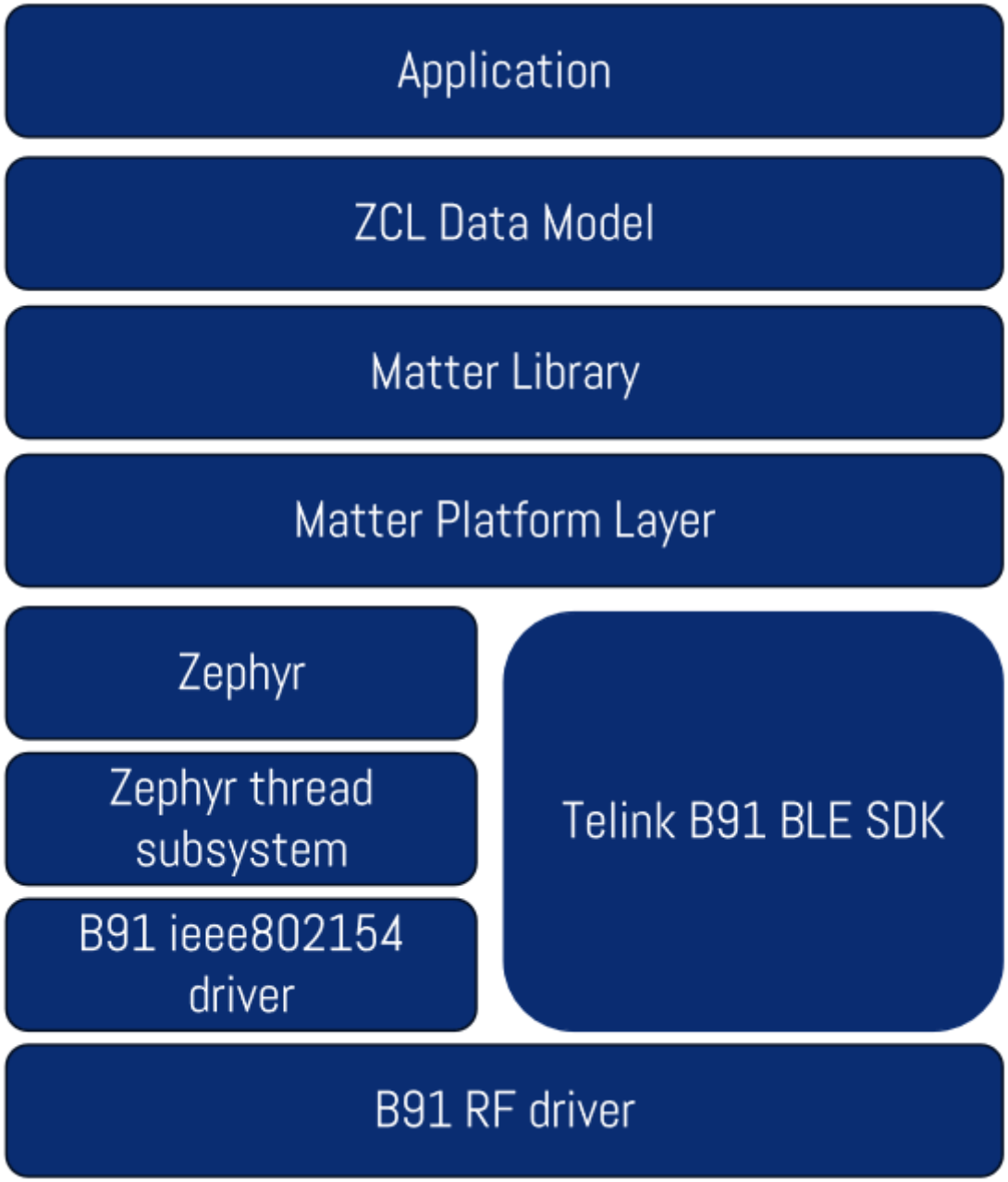

Figure 4.1: Telink Matter 架构

- Application：包含特定 Matter 设备的业务逻辑
- ZCL Data Model：数据模型层，将应用程序与周围的环境连接起来。Matter 中借鉴了 ZCL(ZigBee Cluster Library) 的数据格式
- Matter Library：负责 Matter 信息的通信，加密/解密，与数据模型的交互，配网管理等工作

- Matter Platform Layer：连接 Matter 软件库和特定的硬件平台
- Telink B91 BLE SDK：BLE SDK 主要负责 B91 评估板配网过程
- Zephyr：Zepyhr 实时操作系统
- Zephyr thread subsystem：Zephyr 包含的 OpenThread 的实现
- B91 ieee80214 driver：被 OpenThread 用于实现通讯的 Zephyr 的驱动
- B91 RF driver：被 Zephyr OpenThread 和 Telink BLE SDK 用于实现射频通讯

4.2 数据模型

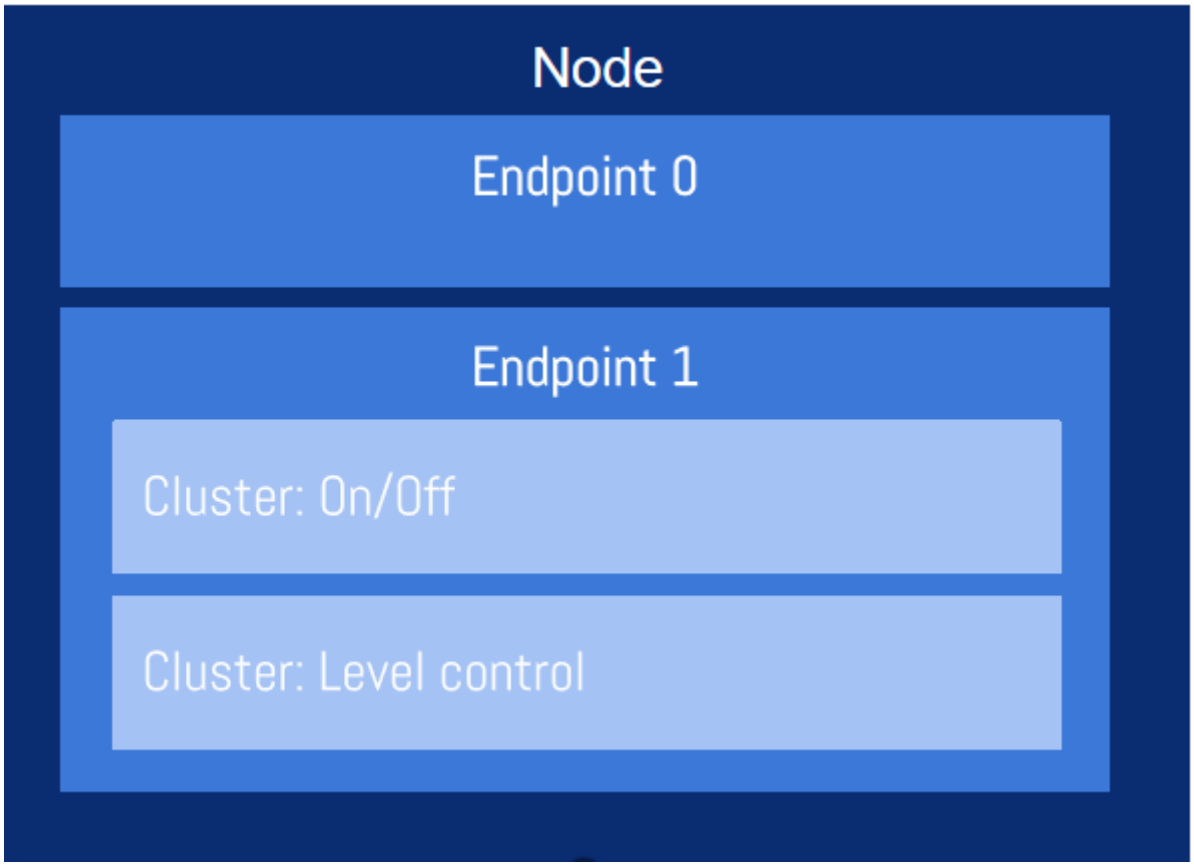

Figure 4.2: Data model

这里以一个具备开关灯和亮度调节功能的灯泡设备为例，介绍 Matter 的数据模型。

- Node 即为节点，表示物理设备，此处的灯泡设备就是一个节点。
- Endpoint 是端点，表示设备类的逻辑功能单元，通讯就是靠本地的端点和设备上的端点进行交互来完成的。每个节点都有许多端点，在每个端点中又有许多 Cluster，这些 Cluster 就说明了这个端点的功能。图中以灯泡为例，Endpoint 0 是需要保留的，提供整个 Matter 节点的一些基础功能，例如 Basic Information Cluster（提供节点的一些基本信息，如固件版本、制造商等等），Access Control Cluster（允许为此节点配置访问控制列表），Network Commissioning Cluster（允许在此节点上配置网络，如 Wi-Fi、以太网、Thread 网络）。在 Endpoint 1 里有 On/Off 和 Level Control 两个 Cluster，它们共同组成了 Endpoint 1 的功能。

- Cluster 用来实现具体的功能，不同的 Cluster 可以组合实现不同的功能。图中的两个 Cluster：On/Off 实现开关灯功能，Level Control 实现调节亮度的功能。Cluster 里定义了一些属性、命令和事件。
- Cluster 使用的是 Client/Sever 的通讯模型，属性通常在 Server，说明了设备的属性，例如灯的开关状态、灯的亮度等。
- Client 和 Server 之间通过预设好的命令进行控制，通常是 Client 发起请求，Server 进行应答的模式。例如，On/Off Cluster 用 On 命令来控制开灯，用 Off 命令来控制关灯，Level Control Cluster 也有 MoveToLevel、Move 等命令。
- 在某些情况下，当 Server 上的属性发生了变化，Server 会通过事件 (Event) 主动通知 Client，以便同步状态。

4.3 代码逻辑和重点源码

4.3.1 重要的回调函数

Matter 代码中预留了很多的回调函数，好处是可以在不修改源码的情况下，最大程度地支持代码扩展。这里以 ligting-app 为例，介绍代码中数据模型涉及的两个重要回调函数，它们的实现都在 examples/lighting-app/telink/src/ZclCallback.cpp 这个文件中。

第一个重要的回调函数是 MatterPostAttributeChangeCallback() 函数，他的默认实现在 zzz_generated/lighting-app/zap-generated/callback-stub.cpp 这个文件中。如下面的代码所示，Matter 提供的默认实现是一个不包含任何代码的空函数。当用户提供了新的实现代码后，新的回调函数将替换该默认实现。

```
void __attribute__((weak)) MatterPostAttributeChangeCallback(const
 ↪  chip::app::ConcreteAttributePath & attributePath, uint8_t type,
                                                        uint16_t size, uint8_t * value)
{}
```

只要一个属性（可以是任何 Cluster 的属性）的内容被更新后，MatterPostAttributeChangeCallback() 这个回调函数就会被调用。在 ligting-app 中，我们实现了如下的回调函数，作用是将属性的最新状态同步到硬件中。例如，用户通过 chip-tool 更新了 On/Off Cluster 的 OnOff 属性值后，开发板上的 LED 状态需要更新为 OnOff 属性对应的状态，这个动作就是在回调函数中完成的（具体是调用了下面代码中的 LightingMgr().InitiateAction() 函数）。同样的，如果用户通过 chip-tool 跟新了 Level Control Cluster 的 Current Level 属性值后，下面的代码也通过调用 LightingMgr().InitiateAction() 更新了 LED 的亮度。

如果用户需要对其他属性的状态变化进行响应，可以在这个回调函数中添加代码进行处理。

```
void MatterPostAttributeChangeCallback(const chip::app::ConcreteAttributePath & attributePath,
 ↪  uint8_t type, uint16_t size, uint8_t * value) {
   ClusterId clusterId     = attributePath.mClusterId;
   AttributeId attributeId = attributePath.mAttributeId;

   if (clusterId == OnOff::Id && attributeId == OnOff::Attributes::OnOff::Id){
       LightingMgr().InitiateAction(*value ? LightingManager::ON_ACTION :
 ↪  LightingManager::OFF_ACTION,
                                    AppEvent::kEventType_Lighting, size, value);
   }
```

```
    else if (clusterId == LevelControl::Id && attributeId ==
    ↪ LevelControl::Attributes::CurrentLevel::Id){
        if (size == 1) {
            LightingMgr().InitiateAction(LightingManager::LEVEL_ACTION,
↪   AppEvent::kEventType_Lighting, size, value);
        }
        else {
            ChipLogError(Zcl, "wrong length for level: %d", size);
        }
    }
}
```

第二个重要的回调函数是 emberAfOnOffClusterInitCallback() 函数，它的默认实现也同样在 zzz_generated/lighting-app/zap-generated/callback-stub.cpp 这个文件中。如下面的代码所示，Matter 提供的默认实现也是一个不包含任何实际代码的空函数。当用户提供了新的实现代码后，新的回调函数将替换该默认实现。

```
void __attribute__((weak)) emberAfOnOffClusterInitCallback(EndpointId endpoint)
{
    // To prevent warning
    (void) endpoint;
}
```

emberAfOnOffClusterInitCallback() 函数在初始化所有 Cluster 时被间接调用到。下面是我们在 ligting-app 中实现的新回调函数。

```
void emberAfOnOffClusterInitCallback(EndpointId endpoint)
{
    GetAppTask().UpdateClusterState();
}
```

如下面的代码所示，UpdateClusterState() 函数的作用是更新 On/Off Cluster 中 OnOff 属性值和 Level Control Cluster 中 Current Level 属性值。

> 注意：emberAfWriteAttribute() 函数被调用后，将最终触发 MatterPostAttributeChangeCallback() 回调函数被调用，进而把硬件状态更新为 Cluster 的属性值对应的状态。

```
void AppTask::UpdateClusterState()
{
    uint8_t onoff = LightingMgr().IsTurnedOn();

    // write the new on/off value
    EmberAfStatus status =
        emberAfWriteAttribute(1, ZCL_ON_OFF_CLUSTER_ID, ZCL_ON_OFF_ATTRIBUTE_ID, &onoff,
↪   ZCL_BOOLEAN_ATTRIBUTE_TYPE);
    if (status != EMBER_ZCL_STATUS_SUCCESS)
```

```
    {
        LOG_ERR("Updating on/off cluster failed: %x", status);
    }

    uint8_t level = LightingMgr().GetLevel();

    status =
        emberAfWriteAttribute(1, ZCL_LEVEL_CONTROL_CLUSTER_ID, ZCL_CURRENT_LEVEL_ATTRIBUTE_ID,
↪   &level, ZCL_INT8U_ATTRIBUTE_TYPE);

    if (status != EMBER_ZCL_STATUS_SUCCESS)
    {
        LOG_ERR("Updating level cluster failed: %x", status);
    }
}
```

4.3.2 例程代码

目前支持的两个例程：

- examples/lighting-app/telink
- examples/light-switch-app/telink

文件	描述
src/main.cpp	主入口函数。包含 Matter 栈的初始化和其他组件，如 Thread，BLE 功能
src/AppTask.cpp	包含应用程序的主循环和事件处理程序
src/LightingManager.cpp	lighting-app 的管理类。包含管理亮灯状态的方法
src/ZclCallbacks.cpp	包含 ZCL 数据模型的回调函数的事件处理程序
src/binding-handler.cpp	light-switch-app 具体相关的事件处理程序
prj.conf	包含关于具体例程的 Zephyr 配置
CMakeLists.txt	包含要生成的源代码列表及其路径

4.3.3 硬件平台代码

Telink B91 评估板的相关代码：

- examples/patform/telink

文件	描述
project_include/OpenThreadConfig.h	包含 OpenThread 的配置的定义
util/src/ButtonManager.cpp	包含负责 TLSR9518ADK80D 按键设置的代码
util/src/LEDWidget.cpp	LED 管理器的相关代码
util/src/ThreadUtil.cpp	用静态密钥配置 Thread 网络的相关代码

4.3.4 Matter **配置代码**

Matter 的配置代码：

- config/telink

文件	描述
app/enable-gnu-std.cmake	包括负责支持 gnu++17 标准的 CMake
app/zephyr.conf	Telink Matter SDK 的 Zephyr 配置文件
chip-gn/	包含 gn 特定文件的文件夹，以构建 Telink Matter SDK
chip-module/CMakeLists.txt	CMake 配置文件，包含 Telink Matter SDK 的设置规则和依赖项。还包含 gn 设置到 CMake 的映射，以链接 SDK 与 Zephyr
chip-module/Kconig	包含 Matter SDK 的 Zephyr 的 Telink 特定配置，也包括 Matter 的一般 Zepyhr 配置

4.3.5 Matter **平台代码**

Matter 的平台代码：

- src/platform/telink

文件	描述
crypto/	包含硬件加密所需源代码的文件夹
BLEManagerImpl.cpp	包含 BLE Manager for Matter SDK 实现的代码，基于 Telink 单连接 BLE SDK
BUILD.gn	用于构建的 gn 文件
../Zephyr	Zephyr 平台层

4.3.6 Telink BLE **单连接** SDK **代码**

Telink BLE 单连接 SDK：

- third_party/telink_sdk

文件	描述
BUILD.gn	构建 Telink BLE SDK 的 gn 文件
repo/	包含 Telink BLE SDK 源代码的目录