Telink Matter Developer's Guide

(TLSR9518)

Ver 0.2.9

2022/10/28

Keyword

TLSR9518, Matter, Zephyr, Thread, BLE

Brief

This guidance provide 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 email to the address of:

telinkcnsales@telink-semi.com

telinkcnsupport@telink-semi.com

Revision History

- **0.1.0** Preliminary release
- **0.2.0** Add section about docker image
- **0.2.1** Remove temporary step for Matter project setup
- **0.2.2** Added Light-Switch-App
- **0.2.3** Fix markdown lint warnings
- **0.2.4** Update Raspberry Pi image
- **0.2.5** Update docker image
- **0.2.6** Update minimum Ubuntu version to 20.04 LTS
- **0.2.7** Update repos url and added Form Thread network via CLI
- **0.2.8** Added OTA section
- **0.2.9** Added factory data section

Contents

Acknowledgements 2
Legal Disclaimer 2
Information 2

Revision History 2
1 Overview 5
2 Required Equipment 6
3 Environment setup 7
3.1 Docker image 7
3.2 Manual environment setup 7
3.2.1 Zephyr project setup 7
3.2.2 Matter project setup 10
3.2.3 Telink tools setup 11
4 Matter firmware 12
4.1 Memory footprint 12
4.2 Device configuration 12
4.3 Build and flash 12
4.4 Logging 13
4.5 UI 13
4.5.1 Buttons 13
4.5.2 LEDs 14
5 Border Router 15
5.1 Radio Co-Processor (RCP) 15
5.1.1 Build and flash 15
5.2 Raspberry Pi 16
5.2.1 Setup 16
5.2.1.1 Write image 16
5.2.1.2 Setup border router software 21
5.2.1.3 Setup from prebuild image 22
5.3 Usage 23
5.3.1 Form Thread network via GUI 23
5.3.2 Form Thread network via CLI 25
5.3.3 Get active dataset 25
6 chip-tool 27
6.1 Build 27
6.2 Usage 27
6.2.1 Commissioning 27
6.2.1.1 BLE-Thread commissioning 27
6.2.2 Lightbulb control 28
6.2.3 Binding cluster and endpoints 30
6.2.3.1 Unicast binding to a remote endpoint using the CHIP Tool 30
6.2.3.2 Group multicast binding to the group of remote endpoints using the CHIP Tool 31
6.2.4 Testing the communication 32
7 OTA with Linux OTA Provider 33

8 chip-device-ctrl.py **34**
8.1 Build 34
8.2 Usage 34
8.2.1 Run 34
8.2.2 Commissioning 34
8.2.3 Lightbulb control 35
9 Configuring factory data for the Telink examples **35**
9.1 Overview 36
9.1.1 Factory data component table 36
9.1.2 Factory data format 39
9.2 Enabling factory data support 40
9.3 Generating factory data 40
9.3.1 Creating factory data JSON file with the first script 40
9.3.2 Verifying using the JSON Schema tool 43
9.3.2.1 Option 1: Using the php-json-schema tool 43
9.3.2.2 Option 2: Using a website validator 43
9.3.2.3 Option 3: Using the Telink Python script 44
9.3.3 Preparing factory data partition on a device 44
9.3.4 Creating a factory data partition with the second script 44
9.4 Building an example with factory data 45
9.4.1 Providing factory data parameters as a build argument list 46
9.5 Programming factory data 46
9.6 Using own factory data implementation 47

1 Overview

This document provides full guidance of Telink matter solution which includes such topics as environment setup, Matter device firmware building and flashing, Border Router setup including RCP building and flashing, building and usage of chip-tool etc.

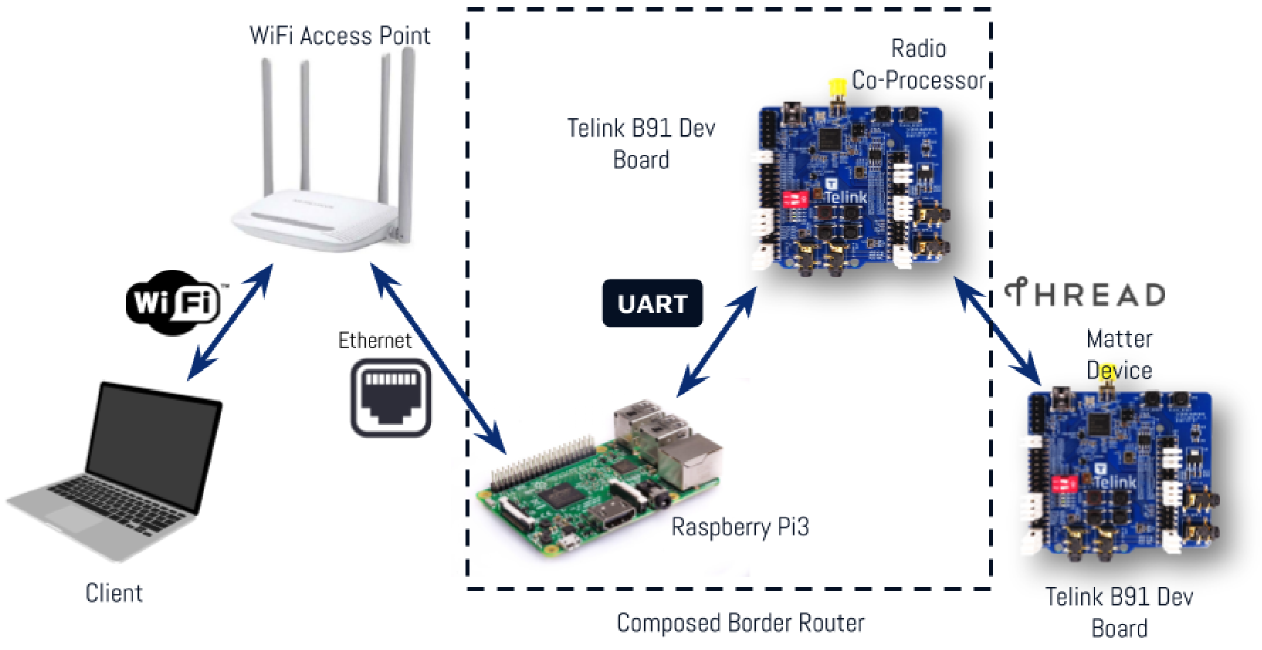


Figure 1.1: Solution Structure

2 Required Equipment

- **TLSR9518ADK80D** as Matter device.
- **TLSR9518ADK80D** as RCP.
- **RaspberryPi3** or higher, as part of border router.
- **SD card** for RPi3. At least 8 GB.
- **Host PC** with Debian based distro (like Ubuntu v20.04 LTS and later) which will be used as a build machine and as host for Matter device.
- **Telink JTAG programmer** to program Matter device and RCP.
- **Wi-Fi Router** that act as Wi-Fi Access Point.

3 Environment setup

3.1 Docker image

To avoid routine with Zephyr environment setup in section 3.2.1 and 3.2.3, user can just pull and run existing docker image that contains ready to use environment.

1. Pull docker image:

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

2. Run docker container:

   Please go through Step 1-3 in 3.2.2 Matter project setup to clone Matter repository into a clean folder before running docker container, because it needs to configure Matter project root directory.

   If you want to set up Zephyr project manually later, you can also back up this folder for now since the environment configuration of Matter project in Docker container may be different from that in local machine.

   Use the following line to run Docker container:

   ```
   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
   ```

   ${MATTER_BASE} is absolute path to Matter project root directory, e.g.

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

   ${YOUR_USERNAME} is your username folder, and **connectedhomeip** is the Matter project folder name.

   The command used here will map Matter project root directory to **/root/chip** in Docker container, so you will get generated bin file even if you exit container.

   After docker container starts, please enter current Matter root directory by the following command.

   ```
   cd /root/chip
   ```

   Continue to perform Step 4 in 3.2.2 Matter project setup to do bootstrap.

Then, follow instruction in Chapter 4. Matter Firmware to build firmware.

3.2 Manual environment setup

3.2.1 Zephyr project setup

Please execute APT update and upgrade before the following steps.

```
sudo apt update
sudo apt upgrade
```

1. Install dependencies:

   ```
   wget https://apt.kitware.com/kitware-archive.sh
   sudo bash kitware-archive.sh
   sudo apt install --no-install-recommends git cmake ninja-build gperf \
   ccache dfu-util device-tree-compiler \
   python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils file \
   make gcc gcc-multilib g++-multilib libsdl2-dev
   ```

   Zephyr requires minimum version for main dependencies for now, such as CMake (3.20.0), Python3 (3.6), Devicetree compiler (1.4.6).

   ```
   cmake --version
   python3 --version
   dtc --version
   ```

   Please verify versions installed on your system before next steps; Otherwise please switch APT mirror to stable and latest one, or update these dependencies manually.

2. Install west:

   ```
   pip3 install --user -U west
   echo 'export PATH=~/.local/bin:"$PATH"' >> ~/.bashrc
   source ~/.bashrc
   ```

   Make sure ~/.local/bin is on $PATH environment variable:

3. Get the Zephyr source code:

   ```
   west init ~/zephyrproject
   cd ~/zephyrproject
   west update
   west zephyr-export
   ```

   It usually costs extra time to get the Zephyr source code using **west init ~/zephyrproject** and **west update** within Chinese mainland. Moreover, some project may fail to update from foreign servers. Please find alternative methods to download the latest source code.

4. Install additional Python dependencies for Zephyr:

   ```
   pip3 install --user -r ~/zephyrproject/zephyr/scripts/requirements.txt
   ```

5. Setup toolchain:

   Download Zephyr toolchain (about 1.2 GB) into local directory to allow you to flash most boards. It may take extra time within Chinese mainland.

   ```
   wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.13.2/zephyr-
   ↪  sdk-0.13.2-linux-x86_64-setup.run
   chmod +x zephyr-sdk-0.13.2-linux-x86_64-setup.run
   ./zephyr-sdk-0.13.2-linux-x86_64-setup.run -- -d ~/.local/zephyr-sdk-0.13.2
   ```

   Download Zephyr SDK and install it in recommended path as below.

```
$HOME/zephyr-sdk[-x.y.z]
$HOME/.local/zephyr-sdk[-x.y.z]
$HOME/.local/opt/zephyr-sdk[-x.y.z]
$HOME/bin/zephyr-sdk[-x.y.z]
/opt/zephyr-sdk[-x.y.z]
/usr/zephyr-sdk[-x.y.z]
/usr/local/zephyr-sdk[-x.y.z]
```

Where [-x.y.z] is optional text, and can be any text, for example, -0.13.2. You cannot move the SDK directory after you have installed it.

6. Build Hello World Sample

   Please verify the official Zephyr project configure is correct using the Hello World sample before continue to set up custom project.

   ```
   cd ~/zephyrproject/zephyr
   west build -p auto -b tlsr9518adk80d samples/hello_world
   ```

   Build the hello_world example with west build command from the root of the Zephyr repository. You can find firmware called **zephyr.bin** under **build/zephyr** directory.

7. Add environment Zephyr script to ~/.bashrc.

   Here is the difference between prior added and added:

   ```
   + source ~/zephyrproject/zephyr/zephyr-env.sh
   ```

   You can add the above line using editor such as vi/vim or execute the following command in bash.

   ```
   echo "source ~/zephyrproject/zephyr/zephyr-env.sh" >> ~/.bashrc
   ```

   Execute the following line to active updated shell environment immediately.

   ```
   source ~/.bashrc
   ```

8. TEMPORARY STEP. Add custom Zephyr remote repository:

   Download custom repo to local as telink_matter branch and update this branch.

   ```
   cd ~/zephyrproject/zephyr
   git remote add custom https://github.com/telink-semi/zephyr
   git fetch custom telink_matter
   git checkout telink_matter
   cd ..
   west update
   ```

   This update may cost extra time within Chinese mainland.

More info you could find here: https://docs.zephyrproject.org/latest/getting_started/index.html

3.2.2 Matter project setup

1. Setup dependencies:

```
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. Clone Matter project:

   Clone Matter project to your local directory, e.g., /home/${YOUR_USERNAME}/workspace/matter.

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

   This clone may cost extra time within Chinese mainland.

3. Update submodules:

   Enter the repo root directory and Update submodule:

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

   This update may cost extra time within Chinese mainland.

   > Optional: The official repository `connectedhomeip` on GitHub has been updated frequently, so there may be compatibility issues between Matter firmware and chip-tool built on the latest commit.

   If Matter devices encounter the issues in commissioning using chip-tool, please consider switching to this commit. Redo step 3~4 as well as rebuild firmware and chip-tool to resolve it.

4. Do bootstrap:

   Download and install packages into local for Matter. It usually takes long time when we run it the first time.

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

   This step will generate an invisible folder called ***.environment*** under the Matter root directory **connectedhomeip**. It may cost extra time or encounter failure within Chinese mainland.

> INFO: In case of any troubles with Matter build environment you may try:

1. Remove the environment (in root directory of Matter project):

```
rm -rf .environment
```

2. Redo bootstrap once again:

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

More info you could find here: https://github.com/project-chip/connectedhomeip/blob/master/docs/guides/BUILDING.md

3.2.3 Telink tools setup

1. Download toolchain:

   Download and unzip Telink toolchain into your local directory, e.g. ~, to allow you flash Zephyr into Telink board.

   ```
   wget http://wiki.telink-semi.cn/tools_and_sdk/Tools/IDE/telink_riscv_linux_toolchain.zip
   unzip telink_riscv_linux_toolchain.zip
   ```

   You may spend several minutes on the download because this zipped file is around hundreds MB. The download may cost extra time outside Chinese mainland.

2. Setup dependencies:

   ```
   sudo dpkg --add-architecture i386
   sudo apt-get update
   sudo apt-get install -y libc6:i386 libncurses5:i386 libstdc++6:i386
   ```

3. Run ICEman.sh script to change udev rules:

   The following step 3 and 4 are set for flashing firmware to developement boards on Ubuntu platform. If you would like to flash firmware on Windows platform, please igonre the following setup.

   ```
   sudo sh ${TELINK_TOOLCHAIN_BASE_DIR}/ice/ICEman.sh
   ```

   `${TELINK_TOOLCHAIN_BASE_DIR}` is Telink SDK root directory, e.g. `~/telink_riscv_linux_toolchain`.

   If you meet issues related to path or permissions, or disconnection from your ubuntu host at current shell, you can enter the above directory and then try to execute this script:

   ```
   cd ${TELINK_TOOLCHAIN_BASE_DIR}/ice
   source ICEman.sh
   ```

4. In `~/.bashrc` add SPI_burn and ICEman to `PATH`.

   Here is the difference between prior added and added:

   ```
   + export PATH=${TELINK_TOOLCHAIN_BASE_DIR}/flash/bin:"$PATH"
   + export PATH=${TELINK_TOOLCHAIN_BASE_DIR}/ice:"$PATH"
   ```

4 Matter firmware

4.1 Memory footprint

Memory Region	Used Size	Total Size	% Used
RAM ILM	62632 B	128 K	47.78%
RAM DLM	74968 B	128 K	57.20%
FLASH	752888 B	1 M	71.80%

4.2 Device configuration

Device-to-device communication without border router available in case if one or several devices configured as FTD (Full Thread Device) for example Light Bulb.

> **Note:** By default all devices configured as MTD (Minimal Thread Device).

Application configuration file location (relative path):

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

MTD (Minimal Thread Device) configuration example:

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

FTD (Full Thread Device) configuration example:

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

4.3 Build and flash

In Matter root folder or `/root/chip/` if using Docker image:

1. Activate Matter environment

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

2. Go to directory with example:

```
cd examples/${app}/telink
```

`${app}`: lighting-app or light-switch-app

3. Remove previous build if exists:

```
rm -rf build/
```

4. Build the example:

```
west build
```

You can find target built file called **zephyr.bin** under **build/zephyr** directory.

5. Flash the example (for ubuntu platform):

```
west flash --erase
```

4.4 Logging

To get output from device, connect UART to following pins:

Name	Pin
RX	PB3 (pin 15 of J34)
TX	PB2 (pin 18 of J34)
GND	GND (pin 23 of J50)

4.5 UI

4.5.1 Buttons

The following buttons are available on **TLSR9518ADK80D** board:

Name	Function	Description
Button 1	Factory reset	Perform factory reset to forget currently commissioned Thread network and back to decommissioned state
Button 2	Lighting control	Manually triggers the lighting state (only for lightning-app)
	LightSwitch control	Triggers the light switch state (only for light-switch-app)
Button 3	Thread start	Commission thread with static credentials and enables the Thread on device

Name	Function	Description
Button 4	Start BLE (optional)	Initiate BLE stack and start BLE advertisement

> Note: The lighting-app and light-switch-app will turn on BLE advertising automatically when powered on. Just need to be careful when you have flashed more than one board at the same time.

4.5.2 LEDs

Red LED indicates current state of Thread network. It is able to be in following states:

State	Description
Blinks with short pulses	Device is not commissioned to Thread, Thread is disabled
Blinks with frequent pulses	Device is commissioned, Thread enabled. Device trying to JOIN thread network
Blinks with wide pulses	Device commissioned and joined to thread network as CHILD

Blue LED shows current state of lightbulb (only for lightning-app)

5 Border Router

Open Thread border router is composed device which contains two main parts:

- **Raspberry Pi** contains all necessary services and firmware to act as a Border Router
- **Radio Co-Processor** is responsible for Thread communication

5.1 Radio Co-Processor (RCP)

5.1.1 Build and flash

As RCP, you may use additional TLSR9518ADK80D board.

1. Go to zephyr project openthread rcp sample directory under zephyr root path.

   ```
   cd ~/zephyrproject/zephyr/samples/net/openthread/coprocessor
   ```

   Or through $ZEPHYR_BASE if you jump from section 3.1 and implement Docker image.

   ```
   cd $ZEPHYR_BASE/samples/net/openthread/coprocessor
   ```

2. Add **CONFIG_OPENTHREAD_THREAD_VERSION_1_2=y** in prj.conf.

   ```
   echo "CONFIG_OPENTHREAD_THREAD_VERSION_1_2=y" >> prj.conf
   ```

3. Build firmware:

   ```
   west build -b tlsr9518adk80d -- -DCONF_FILE="prj.conf overlay-rcp.conf"
   ```

   You can find target built file called **zephyr.bin** under **build/zephyr** directory to flash into the board as RCP.

   If there's a warning says "west: command not found", please active Matter Environment before building.

   ```
   source ${MATTER_BASE}/scripts/activate.sh
   ```

   ${MATTER_BASE} is absolute path to Matter project root directory, e.g.

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

   ${YOUR_USERNAME} is your username folder, and **connectedhomeip** is the Matter project folder name.

   Or the following path if you jump here using Docker.

   ```
   /root/chip
   ```

   Note: The bin file generated outside the above path in Docker container will not exist after you exit the container. Please copy it to somewhere under this Matter project root path if you want to keep it.

4. Flash firmware (for ubuntu platform)

```
west flash
```

5.2 Raspberry Pi

5.2.1 Setup

5.2.1.1 Write image

1. Download Imager
2. Insert your SD card into PC
3. Open Imager and press "CHOOSE OS" button

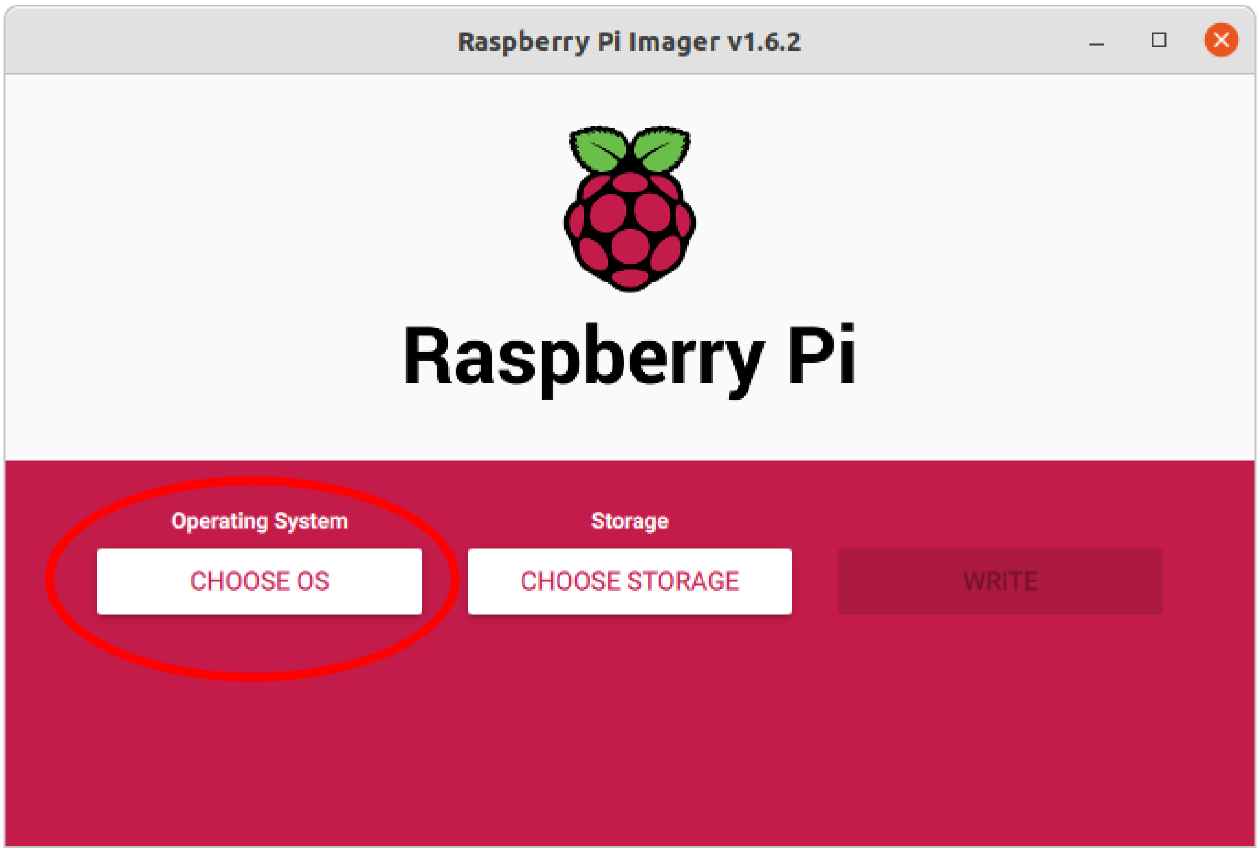
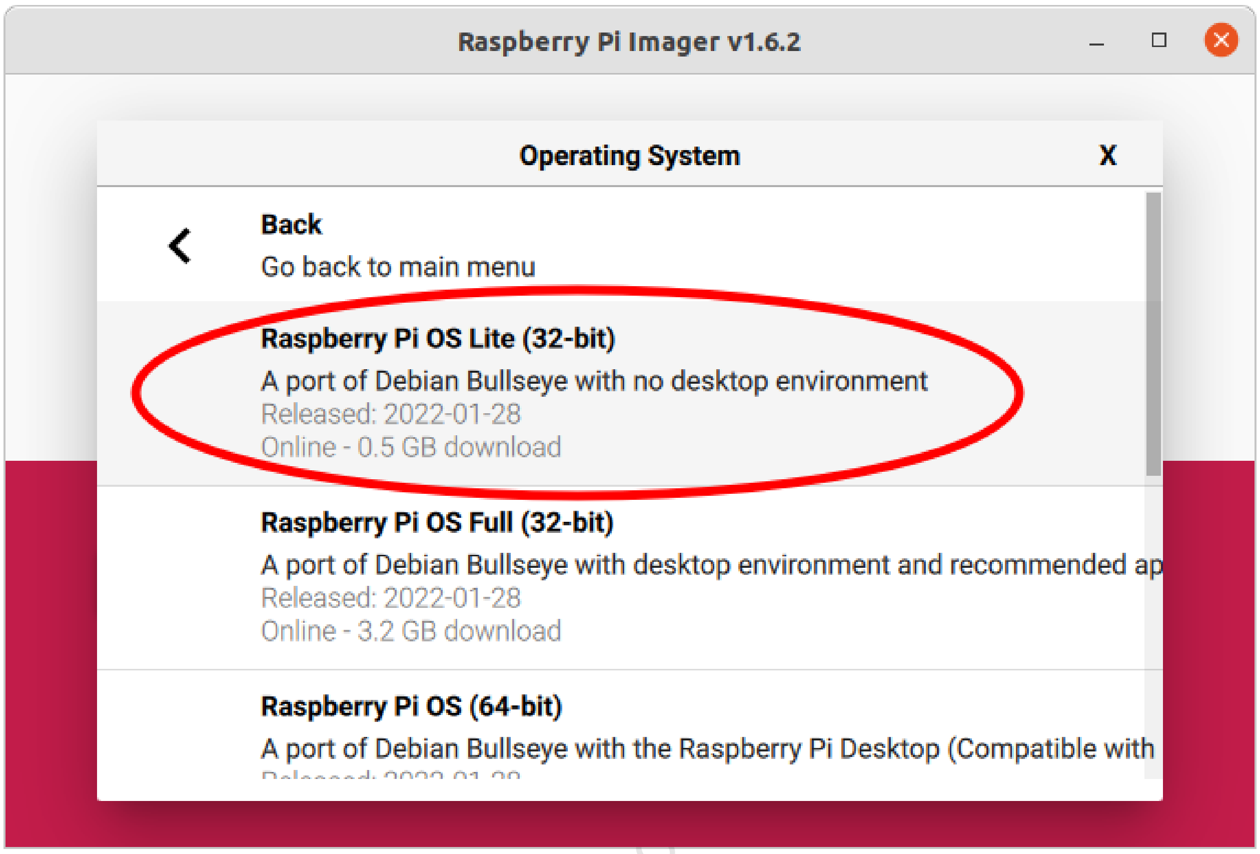


Figure 5.1: Imager step 3 - Choose OS

4. Choose "Raspberry Pi OS (other)" > "Raspberry Pi OS Lite (32-bit)"


Raspberry Pi Imager v1.6.2

Operating System

Back
Go back to main menu

Raspberry Pi OS Lite (32-bit)
A port of Debian Bullseye with no desktop environment
Released: 2022-01-28
Online - 0.5 GB download

Raspberry Pi OS Full (32-bit)
A port of Debian Bullseye with desktop environment and recommended ap
Released: 2022-01-28
Online - 3.2 GB download

Raspberry Pi OS (64-bit)
A port of Debian Bullseye with the Raspberry Pi Desktop (Compatible with

Figure 5.2: Imager step 4

5. Press "CHOOSE STORAGE" button and choose your SD card

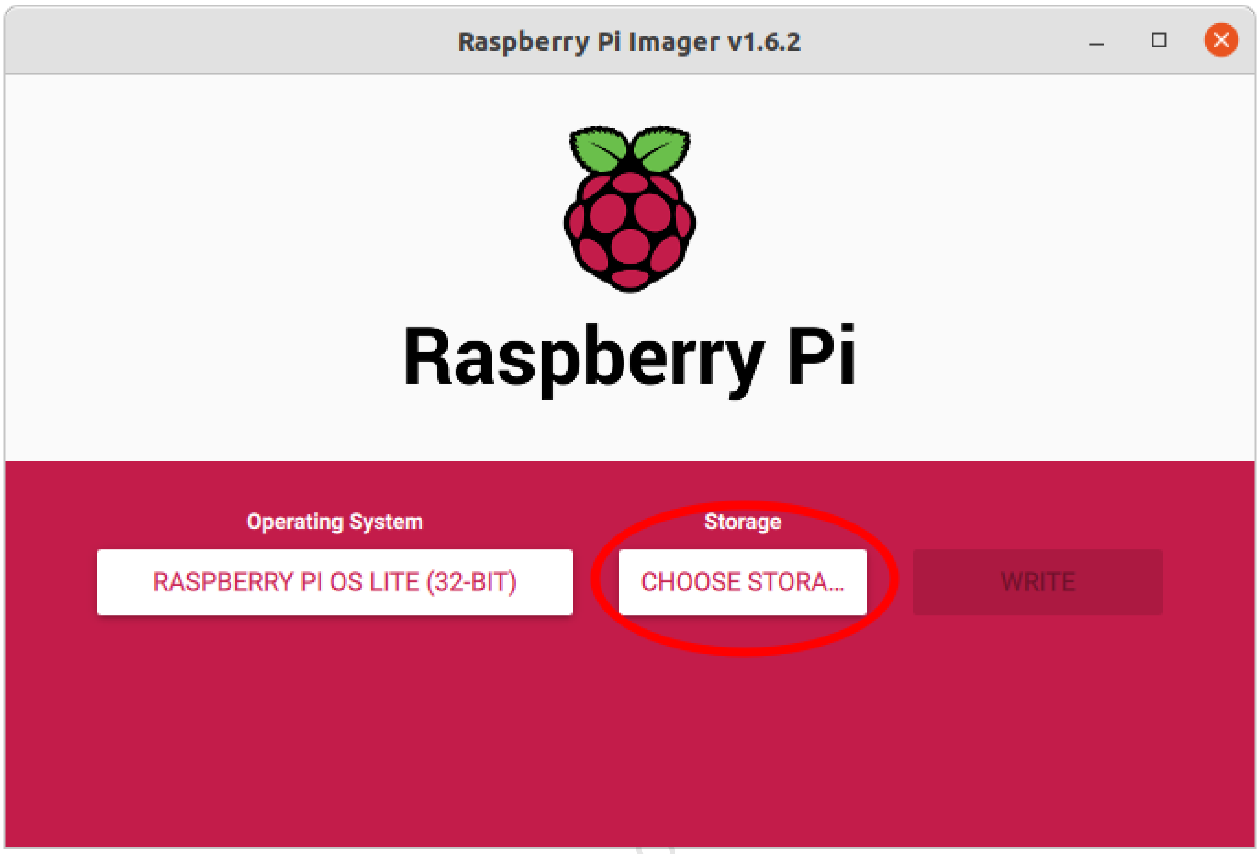
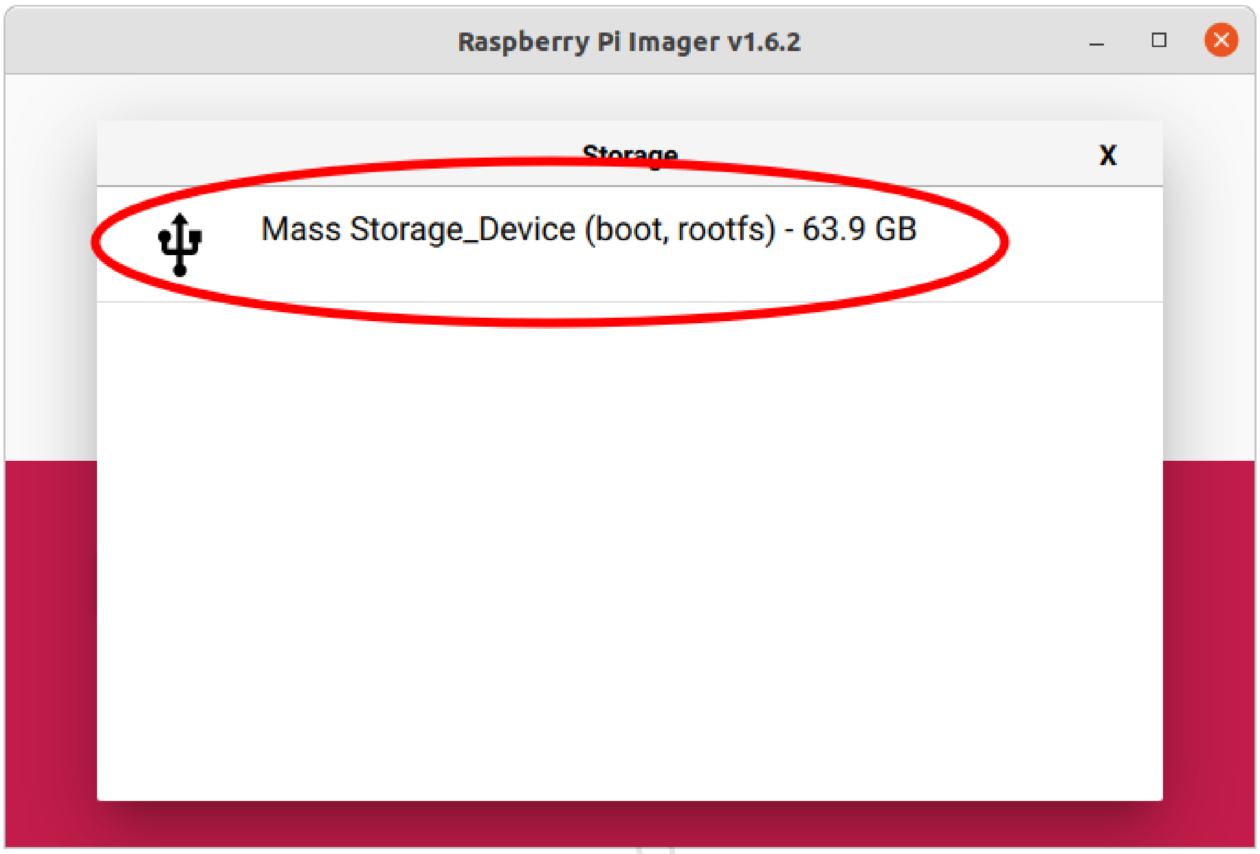


Figure 5.3: Imager step 5 - Storage button



Mass Storage_Device (boot, rootfs) - 63.9 GB

Figure 5.4: Imager step 5 - Choose SD card

6. Press Write

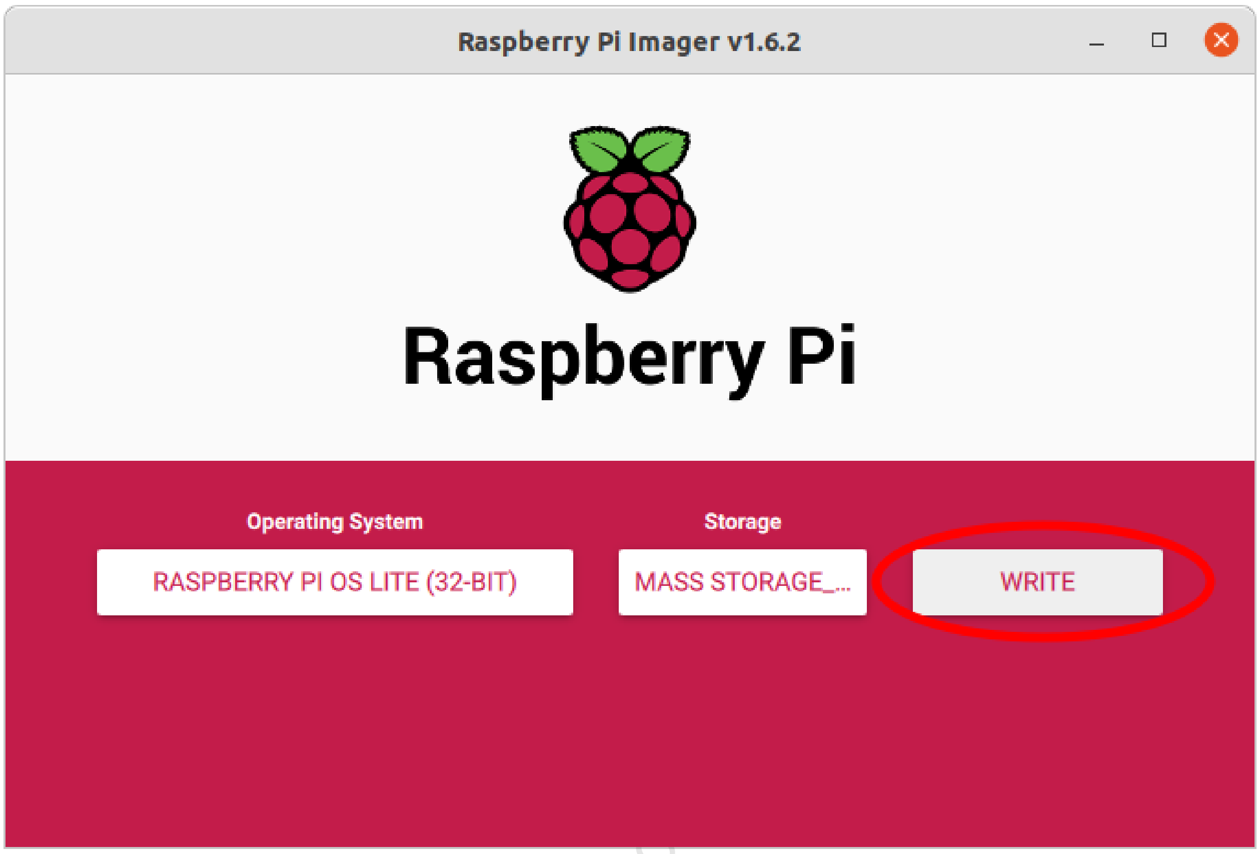
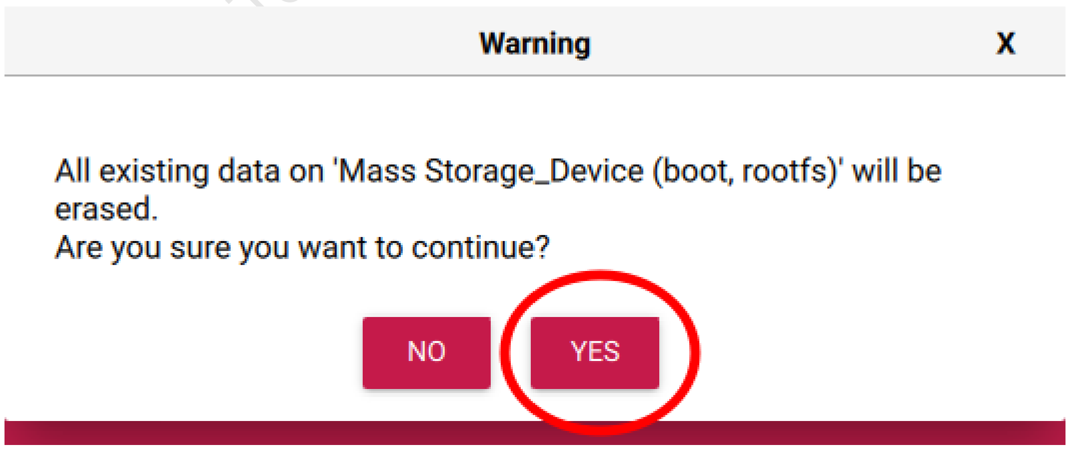


Figure 5.5: Imager step 6 - Write

7. Confirm that you want to continue writhing.

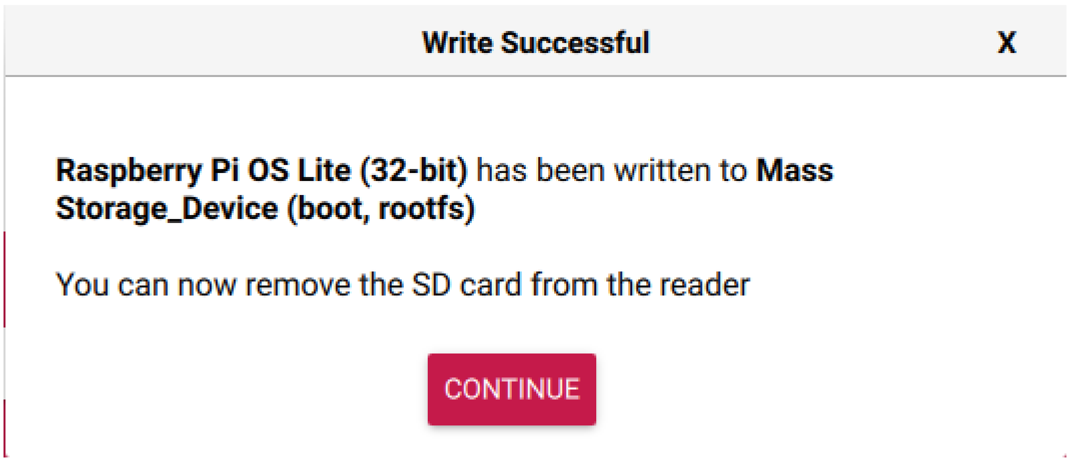
> WARNING: All data from SD card will be erased.



All existing data on 'Mass Storage_Device (boot, rootfs)' will be erased.
Are you sure you want to continue?

Figure 5.6: Imager step 7 - Confirmation message

8. Wait for write completion. If everything is ok, you will see following message.


Write Successful X

Raspberry Pi OS Lite (32-bit) has been written to **Mass Storage_Device (boot, rootfs)**

You can now remove the SD card from the reader

Figure 5.7: Imager step 8 - Write complete

5.2.1.2 Setup border router software

> WARNING: Before you continue, make sure your configured hardware platform is connected to the internet using Ethernet. The bootstrap script disables the platform's Wi-Fi interface and the setup script requires internet connectivity to download and install several packages.

1. Attach keyboard and monitor to Raspberry Pi. Power on Border router. All further steps should be performed directly on PRi.
2. Clone OpenThread Border Router:

```
git clone https://github.com/openthread/ot-br-posix
```

3. Do bootstrap:

```
cd ot-br-posix
./script/bootstrap
```

4. Setup:

```
INFRA_IF_NAME=eth0 ./script/setup
```

5. Attach RCP according to connection map:

RCP	Raspberry
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)

6. Modify /etc/default/otbr-agent file by replacing default string with following difference:

```
- OTBR_AGENT_OPTS="-I wpan0 -B eth0 spinel+hdlc+uart:///dev/ttyACM0 trel://eth0"
+ OTBR_AGENT_OPTS="-I wpan0 -B eth0 spinel+hdlc+uart:///dev/ttyS0?uart-baudrate=57600
↪  trel://eth0"
```

7. Enable UART (ttyS0) on the RPi via raspi-config:

   ```
   sudo raspi-config
   ```

   Interface Options   Serial Port   No (Shell)   Yes (Serial)

8. Config ttyS0 source clock.

   ```
   sudo nano /boot/config.txt
   ```

   do change in file and here is the difference:

   ```
   enable_uart=1
   + core_freq=250
   ```

9. Switch on SSH:

   ```
   sudo raspi-config
   ```

   Interface Options   SSH   Yes (Enable)

10. Reboot the RPi

    ```
    sudo reboot
    ```

More info you can find here: https://openthread.io/guides/border-router/build

5.2.1.3 Setup from prebuild image

1. Insert SD card into PC.
2. Unzip **raspberry_260522.img.zip** archive.
3. Use dd command to copy image to SD card:

   ```
   sudo dd if=<path_to_image> of=<path_to_sd_card_device> status=progress
   ```

   Example:

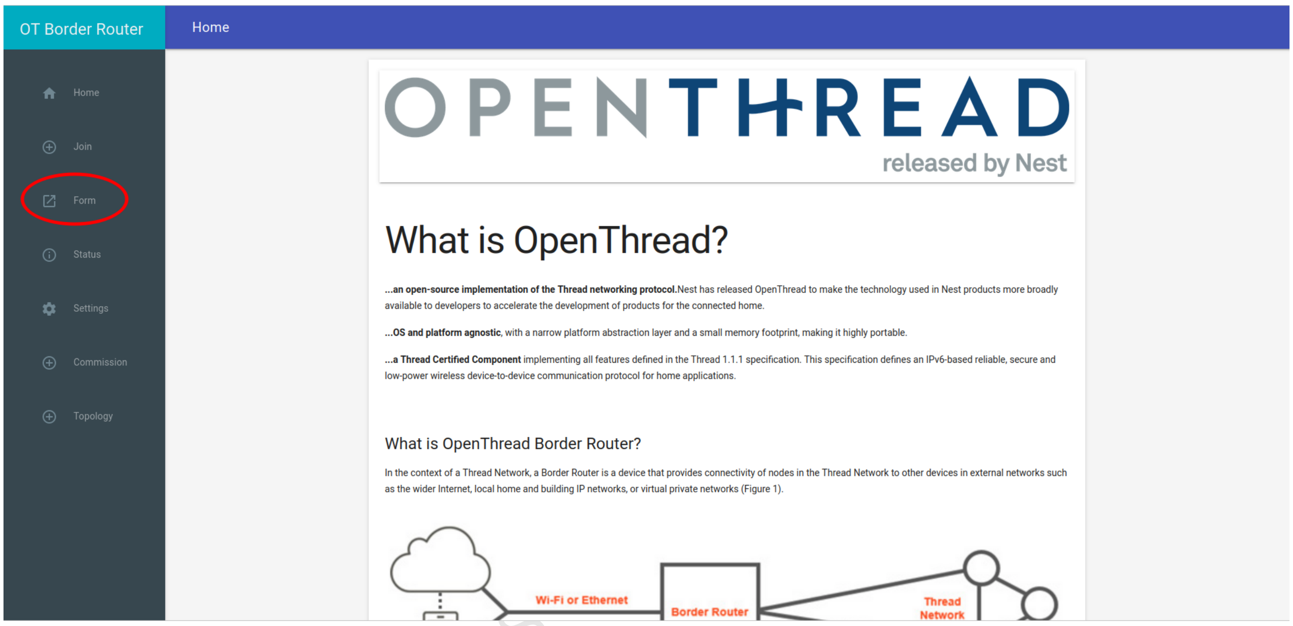
   ```
   sudo dd if=~/Telink/Doc/ZigBee/Matter/LightfairMaterialsPack/raspberry_260522.img of=/dev/
   ↪  sda status=progress
   ```

4. Wait till copy process finishing.
5. Insert SD into Raspberry Pi.
6. Connect RCP.
7. Plug Raspberry Pi to power source.
8. Wait for a few minutes till it loads… Done.

5.3 Usage

5.3.1 Form Thread network via GUI

1. Open your Internet browser.
2. In address line type IP address of your Border Router.
3. If everything is ok, you shall see Border Router Home page.
4. Go to the "Form" page:



What is OpenThread?

...an open-source implementation of the Thread networking protocol. Nest has released OpenThread to make the technology used in Nest products more broadly available to developers to accelerate the development of products for the connected home.

...OS and platform agnostic, with a narrow platform abstraction layer and a small memory footprint, making it highly portable.

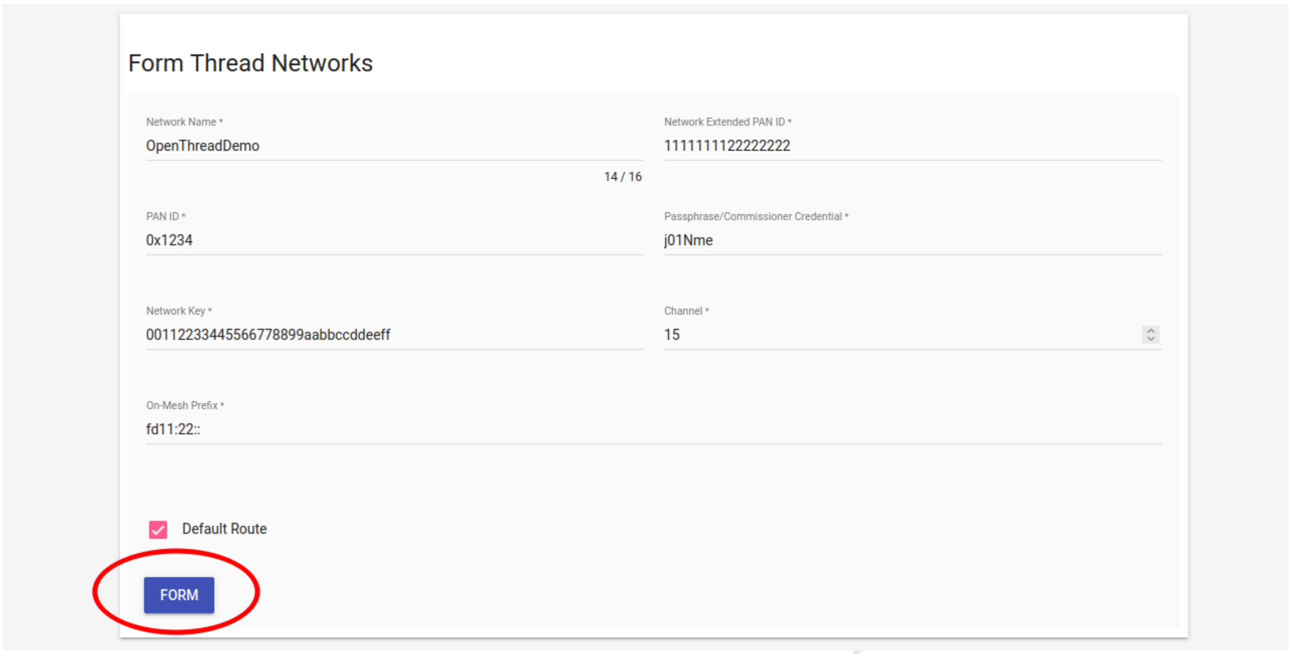
...a Thread Certified Component implementing all features defined in the Thread 1.1.1 specification. This specification defines an IPv6-based reliable, secure and low-power wireless device-to-device communication protocol for home applications.

What is OpenThread Border Router?

In the context of a Thread Network, a Border Router is a device that provides connectivity of nodes in the Thread Network to other devices in external networks such as the wider Internet, local home and building IP networks, or virtual private networks (Figure 1).

Figure 5.8: Form step 4 - Go to form page

5. Input desirable Thread credentials. You may leave it default as well.
6. Press the "Form" button.



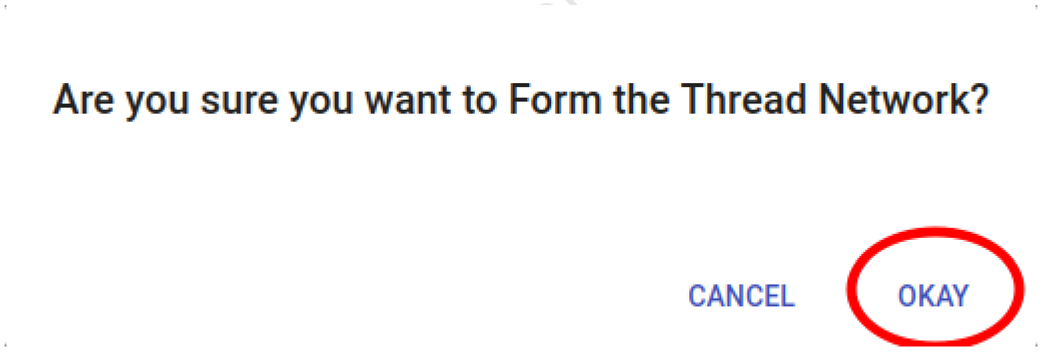
Form Thread Networks

Network Name * OpenThreadDemo 14 / 16	Network Extended PAN ID * 1111111122222222
PAN ID * 0x1234	Passphrase/Commissioner Credential * j01Nme
Network Key * 00112233445566778899aabbccddeeff	Channel * 15
On-Mesh Prefix * fd11:22::	

☑ Default Route

Figure 5.9: Form step 6 - Start form new network

7. Confirm that you want to Form the Thread Network by pressing the "OKAY" button.



Are you sure you want to Form the Thread Network?

Figure 5.10: Form step 7 - Confirmation

8. Wait till "Form operation successful" message.

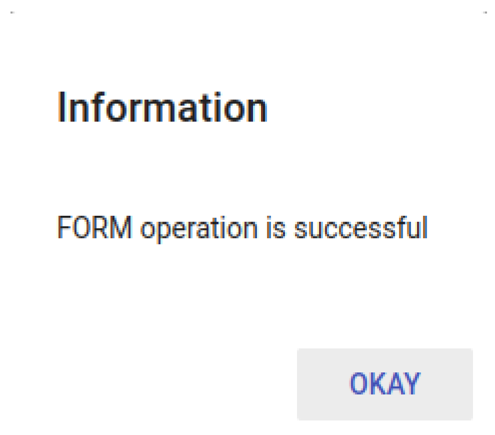


Figure 5.11: Form step 8 - Success

5.3.2 Form Thread network via CLI

1. Connect to OpenThread Border Router via SSH (default password: raspberry):

   ```
   ssh pi@${OTBR_IP_ADDRESS}
   ```

2. Set Thread network operational dataset:

   ```
   sudo ot-ctl dataset set active
   0e080000000000010000000300000f35060004001fffe0020811111111222222220708
   fd7302e133ca932d051000112233445566778899aabbccddeeff030e4f70656e546872
   65616444656d6f010212340410445f2b5ca6f2a93a55ce570a70efeecb0c0402a0fff8
   ```

3. Init Thread network operational dataset:

   ```
   sudo ot-ctl dataset init active
   ```

4. Commit Thread network operational dataset:

   ```
   sudo ot-ctl dataset commit active
   ```

5. Bring interfaces up:

   ```
   sudo ifconfig wpan0 up
   ```

6. Start Thread network:

   ```
   sudo ot-ctl thread start
   ```

5.3.3 Get active dataset

1. Connect to OpenThread Border Router via SSH (default password: raspberry):

   ```
   ssh pi@${OTBR_IP_ADDRESS}
   ```

2. Get Thread network operational dataset:

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

3. As output, you will get dataset which looks like that:

```
0e080000000000010000000300000f35060004001fffe00208111111112222222220708
fd7302e133ca932d051000112233445566778899aabbccddeeff030e4f70656e546872
65616444656d6f010212340410445f2b5ca6f2a93a55ce570a70efeecb0c0402a0fff8
```

4. Store it for further commissioning steps

> WARNING: Thread border router creates new active dataset on each Form operation.

6 chip-tool

6.1 Build

> WARNING: It is really important to build chip tool on same commit as Matter firmware to avoid compatibility issues.

1. Activate environment under Matter project root directory:

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

2. Go to the example folder:

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

3. Remove previous build if necessary:

   ```
   rm -rf out/
   ```

4. Build:

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

5. chip-tool binary located here:

   ```
   ${MATTER_CHIP_TOOL_EXAMPLE_FOLDER}/out/chip-tool
   ```

More info you could find here: https://github.com/project-chip/connectedhomeip/blob/master/examples/chip-tool/README.md

6.2 Usage

6.2.1 Commissioning

6.2.1.1 BLE-Thread commissioning

1. Commission device with the latest active dataset (See Get active dataset paragraph on Border router section):

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

NODE_ID could be any non-zero value which in not used before. It is handler that will be used to perform other chip-tool operations that refer to specific Matter device.

DATASETs will be re-generated by Thread border router after new Thread networks was formed. They and slight different in the middle of the hex string so **DO NOT** use the dataset in following command directly. Please go back to the Get active dataset and replace **DATASET** with the current dataset if you forget it.

Example:

```
$ ./chip-tool pairing ble-thread 1234
hex:0e080000000000010000000300000f35060004001fffe00208111111112222222207 0
8fd61f77bd3df233e051000112233445566778899aabbccddeeff030e4f70656e54687265
616444656d6f010212340410445f2b5ca6f2a93a55ce570a70efeecb0c0402a0fff8
20202021 3840
```

Commission could take some time. If commissioning is successful, you should have the following message:

```
Device commissioning completed with success
```

6.2.2 Lightbulb control

1. Switch on the light:

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

Argument	Description
onoff	Cluster name
on	Command to the cluster
${NODE_ID}	Unique node ID of device. Shall be greater than 0
1	ID of endpoint

2. Switch off the light:

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

Argument	Description
onoff	Cluster name
off	Command to the cluster
${NODE_ID}	Unique node ID of device. Shall be greater than 0
1	ID of endpoint

3. Read the light state:

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

Argument	Description
onoff	Cluster name
read	Command to the cluster
${NODE_ID}	Unique node ID of device. Shall be greater than 0
1	ID of endpoint

4. Change brightness of light:

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

Argument	Description
levelcontrol	Cluster name
move-to-level	Command to the cluster
32	Brightness value
0	Transition time
0	Option mask
0	Option override
${NODE_ID}	Unique node ID of device. Shall be greater than 0
1	ID of endpoint

5. Read brightness level:

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

Argument	Description
levelcontrol	Cluster name
read	Command to the cluster
current-level	Attribute to read
${NODE_ID}	Unique node ID of device. Shall be greater than 0
1	ID of endpoint

6.2.3 Binding cluster and endpoints

Binding links clusters and endpoints on both devices, which enables them to communicate with each other.

To perform binding, you need a controller that can write the binding table to the light switch device and write proper ACL to the endpoint light bulb on the Lighting Example application. For example, you can use the CHIP Tool as the controller. The ACL should contain information about all clusters that can be called by the light switch application. See the section about interacting with ZCL clusters in the CHIP Tool's user guide for more information about ACLs.

You can perform the binding process to a single remote endpoint (unicast binding) or to a group of remote endpoints (group multicast).

> **Note:** To use a light switch without brightness dimmer, apply only the first binding command with cluster no. 6.

6.2.3.1 Unicast binding to a remote endpoint using the CHIP Tool

In this scenario, commands are provided for a light switch device with the `nodeId = <light-switch-node-id>` and a light bulb device with `nodeId = <lighting-node-id>`, both commissioned to the same Matter network.

To perform the unicast binding process, complete the following steps:

1. Add an ACL to the development kit that is programmed with the Lighting Application Example by running the following command:

   ```
   ./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
   ```

   In this command:

   - `[...]` is JSON format message for attr-value so `<light-switch-node-id>` must be a real number when the command is executed.
   - `<lighting-node-id>` can be a shell variable as **${NODE_ID}** used for commissioning before.
   - `{"fabricIndex": 1, "privilege": 5, "authMode": 2, "subjects": [112233], "targets": null}` is an ACL for the communication with the CHIP Tool.
   - `{"fabricIndex": 1, "privilege": 5, "authMode": 2, "subjects": [<light- switch- node-id>], "targets": [{"cluster": 6, "endpoint": 1, "deviceType": null}, {"cluster": 8, "endpoint": 1, "deviceType": null}]}` is an ACL for binding (cluster no. 6 is the On/Off cluster, and the cluster no. 8 is the Level Control cluster).

   This command adds permissions on the lighting application device that allows it to receive commands from the light switch device.

2. Add a binding table to the Light Switch binding cluster:

```
./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
```

In this command:

- `[...]` is JSON format message for attr-value so `<lighting-node-id>` must be real numbers when the command is executed.
- `<light-switch-node-id>` can be a shell variable such as **${SWITCH_NODE_ID}** used by chip-tool to do commissioning with Lighting Switch App.
- `{"fabricIndex": 1, "node": <lighting-node-id>, "endpoint": 1, "cluster": 6}` is a binding for the On/Off cluster.
- `{"fabricIndex": 1, "node": <lighting-node-id>, "endpoint": 1, "cluster": 8}` is a binding for the Level Control cluster.

6.2.3.2 Group multicast binding to the group of remote endpoints using the CHIP Tool

The group multicast binding lets you control more than one lighting device at a time using a single light switch.

The group multicast binding targets all development kits that are programmed with the Lighting Application Example and added to the same multicast group. After the binding is established, the light switch device can send multicast requests, and all the devices in the bound groups can run the received command.

In this scenario, commands are provided for a light switch device with the `nodeId = <light-switch-node-id>` and light bulb devices with `nodeId = <lighting-node-id>`, all commissioned to the same Matter network.

To perform the multicast binding process, complete the following steps:

1. Add the light switch device to the multicast group by running the following command:

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

   - `<light-switch-node-id>` can be a shell variable such as **${SWITCH_NODE_ID}** used by chip-tool to do commissioning with Lighting Switch App.

2. Add all light bulbs to the same multicast group by applying command below for each of the light bulbs, using the appropriate `<lighting-node-id>` (the user-defined ID of the node being commissioned except `<light-switch-node-id>` due to use this `<light-switch-node-id>` for light-switch) for each of them:

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

   - `<lighting-node-id>` can be shell variables as **${NODE_ID}**s used for commissioning before.

3. Add Binding commands for group multicast:

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

   - `<light-switch-node-id>` can be a shell variable such as **${SWITCH_NODE_ID}** used for commissioning before.

6.24 Testing the communication

To test the communication between the light switch device and the bound devices, use light switch buttons.

7 OTA with Linux OTA Provider

OTA feature enabled by default only for ota-requestor-app example. To enable OTA feature for another Telink example:

- set CONFIG_CHIP_OTA_REQUESTOR=y in corresponding "prj.conf" configuration file.

After build application with enabled OTA feature, use next binary files:

- zephyr.bin - main binary to flash PCB (Use 2MB PCB).
- zephyr-ota.bin - binary for OTA Provider

All binaries has the same SW version. To test OTA "zephyr-ota.bin" should have higher SW version than base SW. Set CONFIG_CHIP_DEVICE_SOFTWARE_VERSION=2 in corresponding "prj.conf" configuration file.

Usage of OTA:

1. Build the 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. Run the Linux OTA Provider with OTA image.

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

here:

- `zephyr-ota.bin` is the firmware needs to be updated to

Please keep this terminal window till the end of test, for chip-tool use separate terminal window.

3. Open another terminal and provision the Linux OTA Provider using chip-tool

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

here:

- ${OTA_PROVIDER_NODE_ID} is the node id of Linux OTA Provider. It is similar to **NODE_ID** for lighting-app. You need to set it to any non-zero value which in not used before, .

4. Configure the ACL of the ota-provider-app to allow access

```
./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
```

here:

- ${OTA_PROVIDER_NODE_ID} is the node id of Linux OTA Provider

5. Use the chip-tool to announce the ota-provider-app to start the OTA process

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

here:

- ${OTA_PROVIDER_NODE_ID} is the node id of Linux OTA Provider
- ${DEVICE_NODE_ID} is the node id of paired device

Once the transfer is complete, OTA requestor sends ApplyUpdateRequest command to OTA provider for applying the image. Device will restart on successful application of OTA image.

8 chip-device-ctrl.py

8.1 Build

1. Activate environment under Matter project root directory:

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

2. Build:

```
scripts/build_python.sh -m platform
```

8.2 Usage

8.2.1 Run

1. Activate python environment:

```
source out/python_env/bin/activate
```

2. Launch:

```
sudo out/python_env/bin/chip-device-ctrl
```

8.2.2 Commissioning

1. Set active dataset (**Note**: They are different once a new Thread network is generated. Please ensure to replace the dataset in the following command with the latest one. See Get active dataset paragraph on Border router section):

```
chip-device-ctrl > set-pairing-thread-credential 0e080000000000001000000
0300001335060004001fffe002084fe76e9a8b5edaf50708fde46f999f0698e20510d47
f5027a414ffeebaefa92285cc84fa030f4f70656e5468726561642d653439630102e49c
0410b92f8c7fbb4f9f3e08492ee3915fbd2f0c0402a0fff8
```

2. connect via BLE:

```
chip-device-ctrl > connect -ble 3840 20202021 ${NODE_ID}
```

3. Wait till commissioning completion. If everything is ok you shall see following message.

```
Commissioning complete
```

8.2.3 Lightbulb control

1. Toggle light:

```
chip-device-ctrl > zcl OnOff Toggle ${NODE_ID} 1 0
```

9 Configuring factory data for the Telink examples

Factory data is a set of device parameters written to the non-volatile memory during the manufacturing process. This guide describes the process of creating and programming factory data using Matter and the Telink platform from Telink Semiconductor.

The factory data parameter set includes different types of information, for example about device certificates, cryptographic keys, device identifiers, and hardware. All those parameters are vendor-specific and must be inserted into a device's persistent storage during the manufacturing process. The factory data parameters are read at the boot time of a device. Then, they can be used in the Matter stack and user application (for example during commissioning).

All of the factory data parameters are protected against modifications by the software, and the firmware data parameter set must be kept unchanged during the lifetime of the device. When implementing your firmware, you must make sure that the factory data parameters are not re-written or overwritten during the Device Firmware Update (DFU) or factory resets, except in some vendor-defined cases.

For the Telink platform, the factory data is stored by default in a separate partition of the internal flash memory.

- Overview
- Factory data components
- Factory data format
- Enabling factory data support
- Generating factory data
- Creating factory data JSON file with the first script
- Verifying using the JSON Schema tool
  - Option 1: Using the php-json-schema tool
  - Option 2: Using a website validator
  - Option 3: Using the Telink Python script
- Creating a factory data partition with the second script
- Building an example with factory data
- Providing factory data parameters as a build argument list
- Programming factory data
- Using own factory data implementation

9.1 Overview

You can implement the factory data set described in the factory data component table in various ways, as long as the final HEX/BIN file contains all mandatory components defined in the table. In this guide, the generating factory data and the building an example with factory data sections describe one of the implementations of the factory data set created by the Telink platform's maintainers. At the end of the process, you get a HEX file that contains the factory data partition in the CBOR format.

The factory data accessor is a component that reads and decodes factory data parameters from the device's persistent storage and creates an interface to provide all of them to the Matter stack and to the user application.

The default implementation of the factory data accessor assumes that the factory data stored in the device's flash memory is provided in the CBOR format. However, it is possible to generate the factory data set without using the Telink scripts and implement another parser and a factory data accessor. This is possible if the newly provided implementation is consistent with the Factory Data Provider. For more information about preparing a factory data accessor, see the section about using own factory data implementation.

> Note: Encryption and security of the factory data partition is not provided yet for this feature.

9.1.1 Factory data component table

The following table lists the parameters of a factory data set:

Key name	Full name	Length	Format	Conformance	Description
`version`	factory data version	2 B	uint16	mandatory	A version of the current factory data set. It cannot be changed by a user and it must be coherent with current version of the Factory Data Provider on device side.
`sn`	serial number	<1, 32> B	ASCII string	mandatory	A serial number parameter defines an unique number of manufactured device. The maximum length of the serial number is 32 characters.
`vendor_id`	vendor ID	2 B	uint16	mandatory	A CSA-assigned ID for the organization responsible for producing the device.
`product_id`	product ID	2 B	uint16	mandatory	A unique ID assigned by the device vendor to identify the product. It defaults to a CSA-assigned ID that designates a non-production or test product.

Key name	Full name	Length	Format	Conformance	Description
`vendor_name`	vendor name	<1, 32> B	ASCII string	mandatory	A human-readable vendor name that provides a simple string containing identification of device's vendor for the application and Matter stack purposes.
`product_name`	product name	<1, 32> B	ASCII string	mandatory	A human-readable product name that provides a simple string containing identification of the product for the application and the Matter stack purposes.
`date`	manufacturing date	<8, 10> B	ISO 8601	mandatory	A manufacturing date specifies the date that the device was manufactured. The date format used is ISO 8601, for example `YYYY-MM-DD`.
`hw_ver`	hardware version	2 B	uint16	mandatory	A hardware version number that specifies the version number of the hardware of the device. The value meaning and the versioning scheme is defined by the vendor.
`hw_ver_str`	hardware version string	<1, 64> B	uint16	mandatory	A hardware version string parameter that specifies the version of the hardware of the device as a more user-friendly value than that presented by the hardware version integer value. The value meaning and the versioning scheme is defined by the vendor.
`rd_uid`	rotating device ID unique ID	<16, 32> B	byte string	mandatory	The unique ID for rotating device ID, which consists of a randomly-generated 128-bit (or longer) octet string. This parameter should be protected against reading or writing over-the-air after initial introduction into the device, and stay fixed during the lifetime of the device.

Key name	Full name	Length	Format	Conformance	Description
`dac_cert`	(DAC) Device Attestation Certificate	<1, 602> B	byte string	mandatory	The Device Attestation Certificate (DAC) and the corresponding private key are unique to each Matter device. The DAC is used for the Device Attestation process and to perform commissioning into a fabric. The DAC is a DER-encoded X. 509v3-compliant certificate, as defined in RFC 5280.
`dac_key`	DAC private key	68 B	byte string	mandatory	The private key associated with the Device Attestation Certificate (DAC). This key should be encrypted and maximum security should be guaranteed while generating and providing it to factory data.
`pai_cert`	Product Attestation Intermediate	<1, 602> B	byte string	mandatory	An intermediate certificate is an X. 509 certificate, which has been signed by the root certificate. The last intermediate certificate in a chain is used to sign the leaf (the Matter device) certificate. The PAI is a DER-encoded X.509v3-compliant certificate as defined in RFC 5280.
`spake2_it`	SPAKE2+ iteration counter	4 B	uint32	mandatory	A SPAKE2+ iteration counter is the amount of PBKDF2 (a key derivation function) interactions in a cryptographic process used during SPAKE2+ Verifier generation.
`spake2_salt`	SPAKE2+ salt	<32, 64> B	byte string	mandatory	The SPAKE2+ salt is a random piece of data, at least 32 byte long. It is used as an additional input to a one-way function that performs the cryptographic operations. A new salt should be randomly generated for each password.
`spake2_verifier`	SPAKE2+ verifier	97 B	byte string	mandatory	The SPAKE2+ verifier generated using SPAKE2+ salt, iteration counter, and passcode.
`discriminator`	Discriminator	2 B	uint16	mandatory	A 12-bit value matching the field of the same name in the setup code. The discriminator is used during the discovery process.

Key name	Full name	Length	Format	Conformance	Description
passcode	SPAKE passcode	4 B	uint32	optional	A pairing passcode is a 27-bit unsigned integer which serves as a proof of possession during the commissioning. Its value must be restricted to the values from `0x0000001` to `0x5F5E0FE` (`00000001` to `99999998` in decimal), excluding the following invalid passcode values: `00000000`, `11111111`, `22222222`, `33333333`, `44444444`, `55555555`, `66666666`, `77777777`, `88888888`, `99999999`, `12345678`, `87654321`.
user	User data	variable	JSON string	max 1024 B	The user data is provided in the JSON format. This parameter is optional and depends on user's or manufacturer's purpose (or both). It is provided as a string from persistent storage and should be parsed in the user application. This data is not used by the Matter stack.

9.1.2 Factory data format

The factory data set must be saved into a HEX/BIN file that can be written to the flash memory of the Matter device.

In the Telink example, the factory data set is represented in the CBOR format and is stored in a HEX/BIN file. The file is then programmed to a device. The JSON format is used as an intermediate, human-readable representation of the data. The format is regulated by the JSON Schema file.

All parameters of the factory data set are either mandatory or optional:

- Mandatory parameters must always be provided, as they are required for example to perform commissioning to the Matter network.
- Optional parameters can be used for development and testing purposes. For example, the `user` data parameter consists of all data that is needed by a specific manufacturer and that is not included in the mandatory parameters.

In the factory data set, the following formats are used:

- uint16 and uint32 – These are the numeric formats representing, respectively, two-bytes length unsigned integer and four-bytes length unsigned integer. This value is stored in a HEX file in the big-endian order.
- Byte string - This parameter represents the sequence of integers between `0` and `255`(inclusive), without any encoding. Because the JSON format does not allow to use of byte strings, the `hex:` prefix is added to a parameter, and its representation is converted to a HEX string. For example, an ASCII string *abba*

is represented as *hex:61626261* in the JSON file and then stored in the HEX file as `0x61626261`. The HEX string length in the JSON file is two times greater than the byte string plus the size of the prefix.
- ASCII string is a string representation in ASCII encoding without null-terminating.
- ISO 8601 format is a date format that represents a date provided in the `YYYY-MM-DD` or `YYYYMMDD` format.
- All certificates stored in factory data are provided in the X.509 format.

9.2 Enabling factory data support

By default, the factory data support is disabled in all Telink examples and the Telink device uses predefined parameters from the Matter core, which you should not change. To start using factory data stored in the flash memory and the **Factory Data Provider** from the Telink platform, build an example with the following option:

```
west build -- -DCONFIG_CHIP_FACTORY_DATA=y
```

9.3 Generating factory data

This section describes generating factory data using the following Telink Python scripts:

- The first script creates a JSON file that contains a user-friendly representation of the factory data.
- The second script uses the JSON file to create a factory data partition and save it to a HEX/BIN file.

After these operations, you will program a HEX/BIN file containing factory data partition into the device's flash memory.

You can use the second script without invoking the first one by providing a JSON file written in another way. To make sure that the JSON file is correct and the device is able to read out parameters, verify the file using the JSON schema.

9.3.1 Creating factory data JSON file with the first script

A Matter device needs a proper factory data partition stored in the flash memory to read out all required parameters during startup. To simplify the factory data generation, you can use the generate_telink_chip_factory_data.py Python script to provide all required parameters and generate a human-readable JSON file.

To use this script, complete the following steps:

1. Navigate to the `connectedhomeip` root directory.
2. Run the script with `-h` option to see all possible options:

   ```
   python scripts/tools/telink/generate_telink_chip_factory_data.py -h
   ```

3. Prepare a list of arguments:

   a. Fill up all mandatory arguments:

```
--sn --vendor_id, --product_id, --vendor_name, --product_name, --date, --hw_ver, --
↪  hw_ver_str, --spake2_it, --spake2_salt, --discriminator
```

b. Add output file path:

```
-o <output_dir>
```

c. Generate SPAKE2 verifier using one of the following methods:

- Automatic:

```
--passcode <pass_code> --spake2p_path <path to spake2p executable>
```

> Note: To generate new SPAKE2+ verifier you need `spake2p` executable. See the note at the end of this section to learn how to get it.

- Manual:

```
--spake2_verifier <verifier>
```

d. Add paths to `.der` files that contain PAI and DAC certificates and the DAC private key (replace the respective variables with the file names) using one of the following methods:

- Automatic:

```
--chip_cert_path <path to chip-cert executable>
```

> Note: To generate new certificates, you need the `chip-cert` executable. See the note at the end of this section to learn how to get it.

- Manual:

```
--dac_cert <path to DAC certificate>.der --dac_key <path to DAC key>.der --pai_cert <path
↪  to PAI certificate>.der
```

e. (optional) Add the new unique ID for rotating device ID using one of the following options:

- Provide an existing ID:

```
--rd_uid <rotating device ID unique ID>
```

- Generate a new ID and provide it ():

```
--generate_rd_uid
--rd_uid <rotating device ID unique ID>
```

You can find a newly generated unique ID in the console output.

f. (optional) Add the JSON schema to verify the JSON file (replace the respective variable with the file path):

```
--schema <path to JSON Schema file>
```

g. (optional) Add a request to include a pairing passcode in the JSON file:

```
--include_passcode
```

h. (optional) Add the request to overwrite existing the JSON file:

```
--overwrite
```

4. Run the script using the prepared list of arguments:

```
python generate_telink_chip_factory_data.py <arguments>
```

For example, a final invocation of the Python script can look similar to the following one:

```
$ python scripts/tools/telink/generate_telink_chip_factory_data.py \
--sn "11223344556677889900" \
--vendor_id 65521 \
--product_id 32774 \
--vendor_name "Telink Semiconductor" \
--product_name "not-specified" \
--date "2022-02-02" \
--hw_ver 1 \
--hw_ver_str "prerelase" \
--dac_cert "credentials/development/attestation/Matter-Development-DAC-8006-Cert.der" \
--dac_key "credentials/development/attestation/Matter-Development-DAC-8006-Key.der" \
--pai_cert "credentials/development/attestation/Matter-Development-PAI-noPID-Cert.der" \
--spake2_it 1000 \
--spake2_salt "U1BBS0UyUCBLZXkgU2FsdA==" \
--discriminator 0xF00 \
--generate_rd_uid \
--passcode 20202021 \
--spake2p_path "src/tools/spake2p/out/spake2p" \
--out "build.json" \
--schema "scripts/tools/telink/telink_factory_data.schema"
```

As the result of the above example, a unique ID for the rotating device ID is created, SPAKE2+ verifier is generated using the `spake2p` executable, and the JSON file is verified using the prepared JSON Schema.

If the script finishes successfully, go to the location you provided with the `-o` argument. Use the JSON file you find there when generating the factory data partition.

Note: Generating the SPAKE2+ verifier is optional and requires providing a path to the `spake2p` executable. To get it, complete the following steps:

1. Navigate to the `connectedhomeip` root directory.
2. In a terminal, run the command: `cd src/tools/spake2p && gn gen out && ninja -C out spake2p` to build the executable.

3. Add the `connectedhomeip/src/tools/spake2p/out/spake2p` path as an argument of `--spake2p_path` for the Python script.

> Note: Generating new certificates is optional if default vendor and product IDs are used and requires providing a path to the `chip-cert` executable. To get it, complete the following steps:
>
> 1. Navigate to the `connectedhomeip` root directory.
> 2. In a terminal, run the command: `cd src/tools/chip-cert && gn gen out && ninja -C out chip-cert` to build the executable.
> 3. Add the `connectedhomeip/src/tools/chip-cert/out/chip-cert` path as an argument of `--chip_cert_path` for the Python script.

> Note: By default, overwriting the existing JSON file is disabled. This means that you cannot create a new JSON file with the same name in the exact location as an existing file. To allow overwriting, add the `--overwrite` option to the argument list of the Python script.

9.3.2 Verifying using the JSON Schema tool

The JSON file that contains factory data can be verified using the JSON Schema file. You can use one of three options to validate the structure and contents of the JSON data.

9.3.2.1 Option 1: Using the php-json-schema tool

To check the JSON file using a JSON Schema verification tool manually on a Linux machine, complete the following steps:

1. Install the `php-json-schema` package:

```
sudo apt install php-json-schema
```

2. Run the following command, with and replaced with the paths to the JSON file and the Schema file, respectively:

```
validate-json <path_to_JSON_file> <path_to_schema_file>
```

The tool returns empty output in case of success.

9.3.2.2 Option 2: Using a website validator

You can also use external websites instead of the `php-json-schema` tool to verify a factory data JSON file. For example, go to the JSON Schema Validator website, copy-paste the content of the JSON Schema file to the first window and a JSON file to the second one. A message under the window indicates the validation status.

9.3.2.3 Option 3: Using the Telink Python script

You can have the JSON file checked automatically by the Python script during the file generation. For this to happen, provide the path to the JSON schema file as an additional argument, which should replace the variable in the following command:

```
python generate_telink_chip_factory_data.py --schema <path_to_schema>
```

> Note: To learn more about the JSON schema, visit this unofficial JSON Schema tool usage website.

9.3.3 Preparing factory data partition on a device

The factory data partition is an area in the device's persistent storage where a factory data set is stored. This area is configured in DTS file `zephyr/boards/riscv/tlsr9518adk80d/tlsr9518adk80d.dts`, within which all partitions are declared.

To prepare an example that supports factory data, add a partition called `factory- data` to the `tlsr9518adk80d.dts` file. The partition size should be a multiple of one flash page (for B91 SoCs, a single page size equals 4 kB).

See the following code snippet for an example of a factory data partition in the `tlsr9518adk80d.dts` file. The snippet is based on the `tlsr9518adk80d.dts` file from telink-semi's custom branch telink_factory_data:

```
...
  scratch_partition: partition@f0000 {
   label = "image-scratch";
   reg = <0xf0000 0x4000>;
  };
  factory_partition: partition@f4000 {
   label = "factory-data";
   reg = <0xf4000 0x1000>;
  };
  storage_partition: partition@f5000 {
   label = "storage";
   reg = <0xf5000 0xa000>;
  /* region <0xff000 0x1000> is reserved for Telink B91 SDK's data */
  };
...
```

In this example, a `factory-data` partition has been placed between the partition (image-scratch) and the storage. Its size has been set to one flash page (4 kB).

9.3.4 Creating a factory data partition with the second script

To store the factory data set in the device's persistent storage, convert the data from the JSON file to its binary representation in the CBOR format. To do this, use the telink_generate_partition.py to generate the factory data partition:

1. Navigate to the connectedhomeip root directory
2. Run the following command pattern:

```
python scripts/tools/telink/telink_generate_partition.py -i <path_to_JSON_file> -o
↪  <path_to_output> --offset <partition_address_in_memory> --size <partition_size>
```

In this command:

- is a path to the JSON file containing appropriate factory data.
- is a path to an output file without any prefix. For example, providing `/build/output` as an argument will result in creating `/build/output.hex` and `/build/output.bin`.
- is an address in the device's persistent storage area where a partition data set is to be stored.
- is a size of partition in the device's persistent storage area. New data is checked according to this value of the JSON data to see if it fits the size.

To see the optional arguments for the script, use the following command:

```
python scripts/tools/telink/telink_generate_partition.py -h
```

Example of the command for the Telink DK:

```
python scripts/tools/telink/telink_generate_partition.py -i build/zephyr/factory_data.json -o
↪  build/zephyr/factory_data --offset 0xf4000 --size 0x1000
```

As a result, `factory_data.hex` and `factory_data.bin` files are created in the `/build/zephyr/` directory. The first file contains the memory offset. For this reason, it can be programmed directly to the device using a programmer.

9.4 Building an example with factory data

You can manually generate the factory data set using the instructions described in the Generating factory data section. Another way is to use the Telink platform build system that creates factory data content automatically using Kconfig options and includes the content in the final firmware binary.

To enable generating the factory data set automatically, go to the example's directory and build the example with the following option:

```
west build -- -DCONFIG_CHIP_FACTORY_DATA=y -DCONFIG_CHIP_FACTORY_DATA_BUILD=y
```

Alternatively, you can also add `CONFIG_CHIP_FACTORY_DATA_BUILD=y` Kconfig setting to the example's `prj.conf` file.

Each factory data parameter has a default value. These are described in the Kconfig file. Setting a new value for the factory data parameter can be done either by providing it as a build argument list or by using interactive Kconfig interfaces.

9.4.1 Providing factory data parameters as a build argument list

This way for providing factory data can be used with third-party build script, as it uses only one command. All parameters can be edited manually by providing them as an additional option for the west command:

```
west build -- -DCONFIG_CHIP_FACTORY_DATA=y --DCONFIG_CHIP_FACTORY_DATA_BUILD=y --
↪  DCONFIG_CHIP_DEVICE_DISCRIMINATOR=0xF11
```

Alternatively, you can add the relevant Kconfig option lines to the example's `prj.conf` file.

9.5 Programming factory data

The HEX/BIN file containing factory data can be programmed into the device's flash memory using `BDT Tool` and the Telink burning key.

Another way to program the factory data to a device is to use the Telink platform build system described in Building an example with factory data, and build an example with the additional option `-DCONFIG_CHIP_FACTORY_DATA_MERGE_WITH_FIRMWARE=y`:

```
$ west build -- \
-DCONFIG_CHIP_FACTORY_DATA=y \
-DCONFIG_CHIP_FACTORY_DATA_BUILD=y \
-DCONFIG_CHIP_FACTORY_DATA_MERGE_WITH_FIRMWARE=y
```

You can also build an example with auto-generation of new CD, DAC and PAI certificates. The newly generated certificates will be added to factory data set automatically. To generate new certificates disable using default certificates by building an example with the additional option `-DCHIP_FACTORY_DATA_USE_DEFAULT_CERTS=n`:

```
$ west build -- \
-DCONFIG_CHIP_FACTORY_DATA=y \
-DCONFIG_CHIP_FACTORY_DATA_BUILD=y \
-DCONFIG_CHIP_FACTORY_DATA_MERGE_WITH_FIRMWARE=y \
-DCONFIG_CHIP_FACTORY_DATA_USE_DEFAULT_CERTS=n
```

> Note: To generate new certificates using the Telink platform build system, you need the `chip-cert` executable in your system variable PATH. To learn how to get `chip-cert`, go to the note at the end of Creating a factory data partition with the second script section, and then add the newly built executable to the system variable PATH. The Cmake build system will find this executable automatically.

After that, use the following command from the example's directory to write firmware and newly generated factory data at the same time:

```
west flash
```

9.6 Using own factory data implementation

The factory data generation process described above is only an example valid for the Telink platform. You can well create a HEX file containing all Factory data component table in any format and then implement a parser to read out all parameters and pass them to a provider. Each manufacturer can implement a factory data set on its own by implementing a parser and a factory data accessor inside the Matter stack. Use the Telink Provider and FactoryDataParser as examples.

You can read the factory data set from the device's flash memory in different ways, depending on the purpose and the format. In the Telink example, the factory data is stored in the CBOR format. The device uses the Factory Data Parser to read out raw data, decode it, and store it in the `FactoryData` structure. The Factor Data Provider implementation uses this parser to get all needed factory data parameters and provide them to the Matter core.

In the Telink example, the `FactoryDataProvider` is a template class that inherits from `DeviceAttestationCredentialsProvi` `CommissionableDataProvider`, and `DeviceInstanceInfoProvider` classes. Your custom implementation must also inherit from these classes and implement their functions to get all factory data parameters from the device's flash memory. These classes are virtual and need to be overridden by the derived class. To override the inherited classes, complete the following steps:

1. Override the following methods:

```
// ===== Members functions that implement the DeviceAttestationCredentialsProvider
CHIP_ERROR GetCertificationDeclaration(MutableByteSpan & outBuffer) override;
CHIP_ERROR GetFirmwareInformation(MutableByteSpan & out_firmware_info_buffer) override;
CHIP_ERROR GetDeviceAttestationCert(MutableByteSpan & outBuffer) override;
CHIP_ERROR GetProductAttestationIntermediateCert(MutableByteSpan & outBuffer) override;
CHIP_ERROR SignWithDeviceAttestationKey(const ByteSpan & messageToSign, MutableByteSpan
↪   & outSignBuffer) override;

// ===== Members functions that implement the CommissionableDataProvider
CHIP_ERROR GetSetupDiscriminator(uint16_t & setupDiscriminator) override;
CHIP_ERROR SetSetupDiscriminator(uint16_t setupDiscriminator) override;
CHIP_ERROR GetSpake2pIterationCount(uint32_t & iterationCount) override;
CHIP_ERROR GetSpake2pSalt(MutableByteSpan & saltBuf) override;
CHIP_ERROR GetSpake2pVerifier(MutableByteSpan & verifierBuf, size_t & verifierLen)
↪   override;
CHIP_ERROR GetSetupPasscode(uint32_t & setupPasscode) override;
CHIP_ERROR SetSetupPasscode(uint32_t setupPasscode) override;

// ===== Members functions that implement the DeviceInstanceInfoProvider
CHIP_ERROR GetVendorName(char * buf, size_t bufSize) override;
CHIP_ERROR GetVendorId(uint16_t & vendorId) override;
CHIP_ERROR GetProductName(char * buf, size_t bufSize) override;
CHIP_ERROR GetProductId(uint16_t & productId) override;
CHIP_ERROR GetSerialNumber(char * buf, size_t bufSize) override;
CHIP_ERROR GetManufacturingDate(uint16_t & year, uint8_t & month, uint8_t & day)
↪   override;
CHIP_ERROR GetHardwareVersion(uint16_t & hardwareVersion) override;
```

```
    CHIP_ERROR GetHardwareVersionString(char * buf, size_t bufSize) override;
    CHIP_ERROR GetRotatingDeviceIdUniqueId(MutableByteSpan & uniqueIdSpan) override;
```

2. Move the newly created parser and provider files to your project directory.
3. Add the files to the `CMakeList.txt` file.
4. Disable building both the default and the Telink implementations of factory data providers to start using your own implementation of factory data parser and provider. This can be done in one of the following ways:

- Add `CONFIG_CHIP_FACTORY_DATA_CUSTOM_BACKEND=y` Kconfig setting to `prj.conf` file.
- Build an example with following option:

```
west build -- -DCONFIG_CHIP_FACTORY_DATA_CUSTOM_BACKEND=y
```