ESPRESSIF

ESP-Matter Programming Guide

Release latest

Espressif Systems

Nov 13, 2026

CONTENTS

1 Table of Contents **3**

1.1 Introduction . . . 3
1.2 Developing with the SDK . . . 6
1.3 Matter Controller . . . 34
1.4 Matter Certification . . . 49
1.5 Production Considerations . . . 59
1.6 Espressif Matter Services . . . 63
1.7 Security Considerations . . . 64
1.8 Configuration options to optimize RAM and Flash . . . 66
1.9 API Reference . . . 73
1.10 Enabling ESP-Insights in ESP-Matter . . . 77
1.11 Application User Guide . . . 77
1.12 Copyrights and Licenses . . . 87
1.13 A1 FAQs . . . 88

Matter is a unified IP-based connectivity protocol that is designed to connect and build open, reliable and secure IoT ecosystems. This new technology and connectivity standard enables communication among a wide range of smart devices. Matter supports IP connectivity over Wi-Fi, Thread and Ethernet.

Espressif's SDK for Matter is the official Matter development framework for Espressif's ESP32 series SoCs.

We have put together a series of blog posts that introduces various aspects of Matter. We recommend that you go through this Espressif Matter Blog.

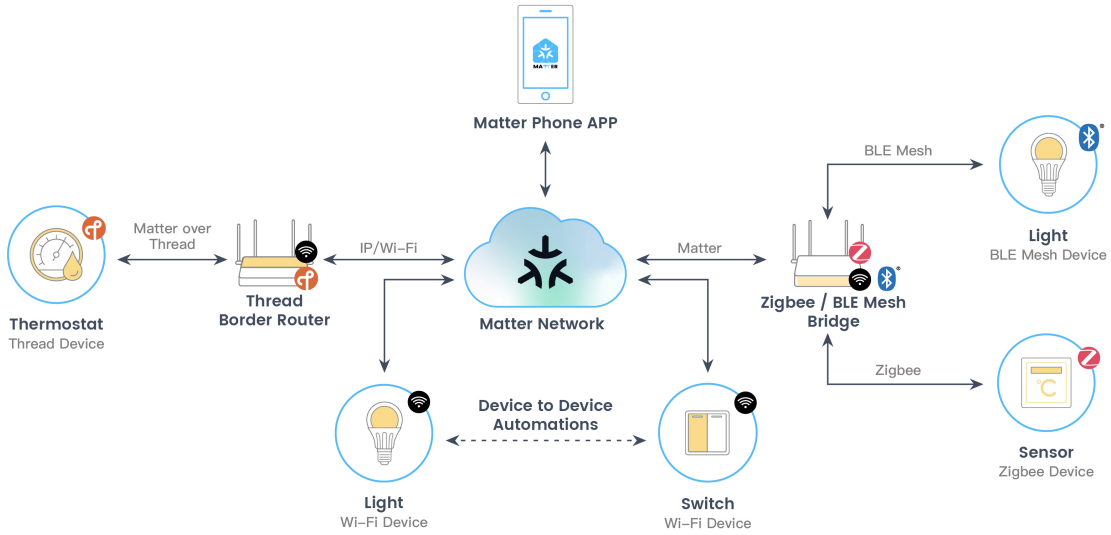
CHAPTER

ONE

TABLE OF CONTENTS

1.1 Introduction

1.1.1 Espressif Matter Solutions

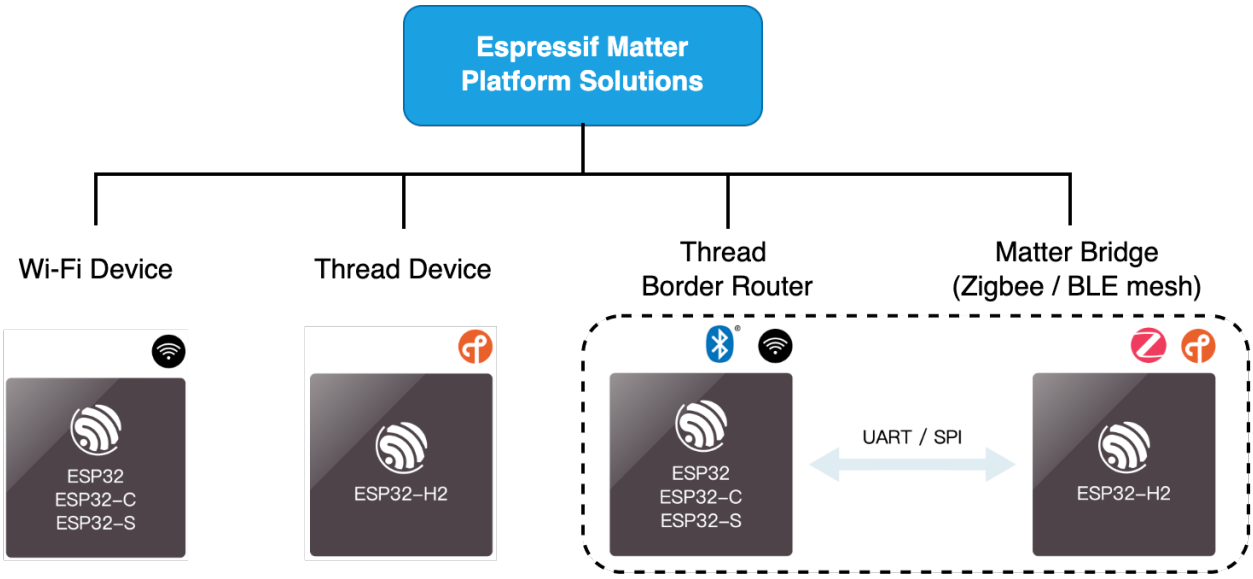


Espressif's Matter Solutions consist of:

- A full spectrum of Matter device platforms
- Production ready SDK for Matter
- Matter and ESP RainMaker integration

Espressif Matter Platforms

Espressif platform solutions are as shown below:

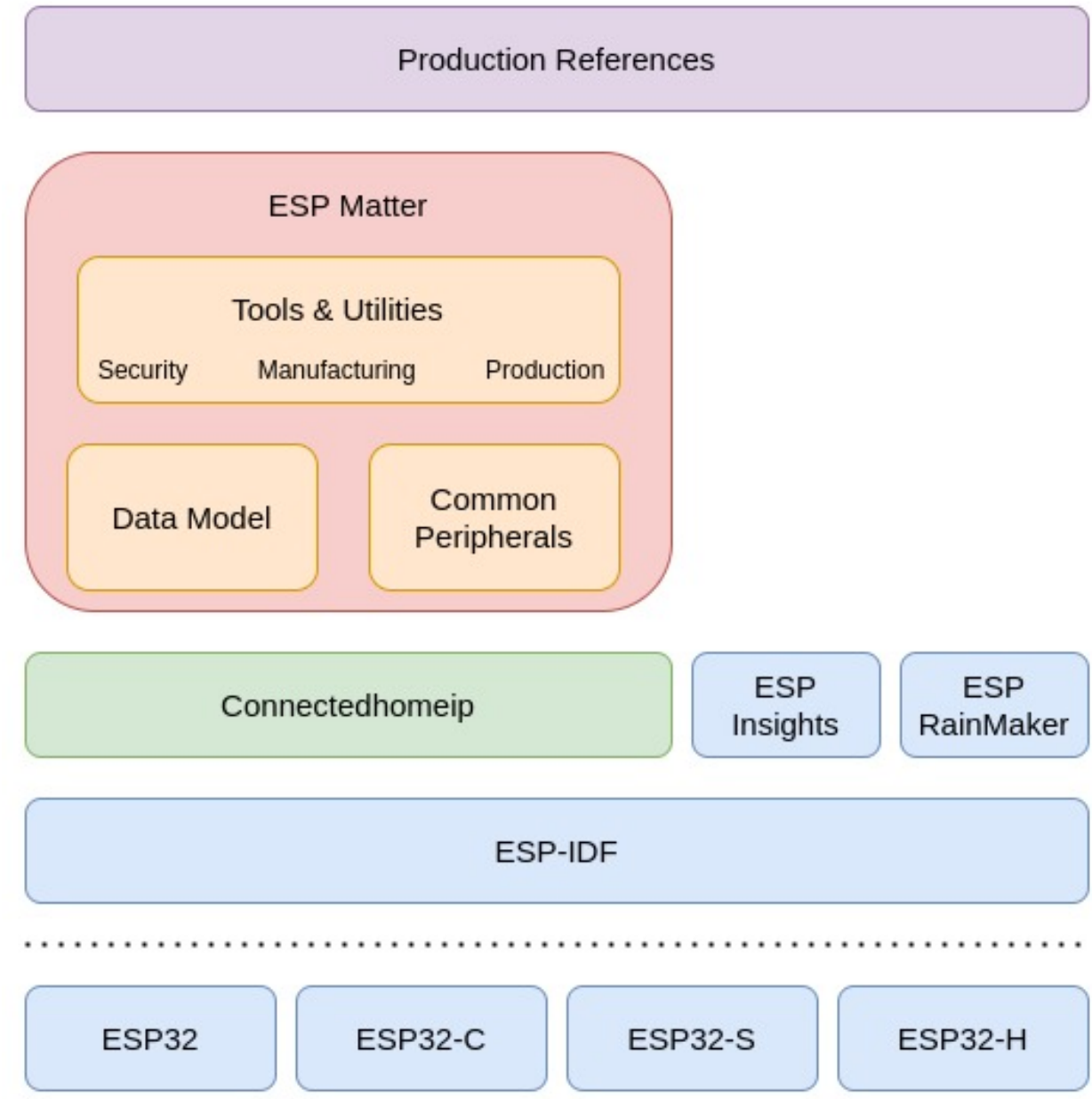


- The Wi-Fi-enabled SoCs and modules, such as ESP32, ESP32-C and ESP32-S series can be used to build **Matter Wi-Fi devices**.
- SoCs and modules(ESP32-H series, ESP32-C5, and ESP32-C6) with 802.15.4 can be used to build **Matter Thread devices**.
- By efficiently combining ESP32-H and our Wi-Fi SoCs, a **Thread Border Router** can be built to connect the Thread network with the Wi-Fi network. We provide hardware devkits, reference designs and production-ready SDK, which supports the latest Thread 1.3 feature for Matter.
- We also provide Matter-Zigbee and Matter-BLE Mesh bridge solutions that enable non-Matter devices based on Zigbee, Bluetooth LE Mesh and other protocols to connect to the Matter ecosystem. A **Matter-Zigbee Bridge** uses ESP32-H and another Wi-Fi SoC, while a **Matter-BLE Mesh Bridge** can be done on a single SoC with both Wi-Fi and Bluetooth LE interfaces.

Espressif's SDK for Matter

Espressif's SDK for Matter is built on top of the open source Matter SDK, and provides simplified APIs, commonly used peripherals, tools and utilities for security, manufacturing and production accompanied by exhaustive documentation. It includes rich production references, aimed to simplify the development process of Matter products and enable the users to go to production in the shortest possible time.

In addition, the SDK also integrates ESP RainMaker and ESP Insights for cloud services and remote diagnostics.


Production References
ESP Matter
Tools & Utilities
Security
Manufacturing
Production
Data Model
Common Peripherals
Connectedhomeip
ESP Insights
ESP RainMaker
ESP-IDF
ESP32
ESP32-C
ESP32-S
ESP32-H

Matter and ESP RainMaker Integration

Espressif's AIoT cloud platform ESP RainMaker can provide remote control and enable cloud-based device management for Matter devices.

By combining the above-mentioned Matter hardware and software solutions with ESP RainMaker, this one-stop Matter ecosystem solution provides a full-fledged cloud deployment through your own private account with advanced device management features.

1.1.2 Try it yourself

If you want to test Matter on Espressif devices without any additional setup:

1.1.3 3. Try Data Model Validator

Espressif provides a web UI tool to validate the data model of the Matter devices. It is available at *https://espressif.github.io/esp-matter-tools/*.

1.2 Developing with the SDK

Please refer the Release Notes to know more about the releases

You can develop with ESP-IDF and ESP-Matter in either of the following ways:

- Set up ESP-IDF and ESP-Matter on the host.
- Use the official Docker image, which already includes both.

1.2.1 Setup on the Host

Note: Skip this step if you prefer to use the Docker image. See *Using Docker Image*.

Host Setup

You should install drivers and support packages for your development host. Linux and Mac OS-X are the supported development hosts in Matter, the recommended host versions:

- Ubuntu 20.04 or 22.04 or 24.04 LTS
- macOS 10.15 or later

Additionally, we also support developing on Windows Host using WSL.

Windows 10

Development on Windows is supported using Windows Subsystem for Linux (WSL). Please follow the below instructions to set up host.

- Install and enable Windows Subsystem for Linux 2 (WSL2).
- Install Ubuntu 20.04 or 22.04 from the Windows App Store.
- Start Ubuntu (search into start menu) and run command `uname -a`, it should report a kernel version of `5.10.60.1` or later. If not please upgrade the WSL2. To upgrade the kernel, run `wsl --upgrade` from Windows Power Shell.
- Windows does not support exposing COM ports to WSL distros. Install usbipd-win and WSL (usbipd-win WSL Support).
- Here onwards process for setting esp-matter and building examples is same as other hosts.
- Please clone the repositories from inside the WSL environment and not inside a mounted directory.

For using CHIP tool on WSL, please check *Using CHIP-tool in WSL*.

For using VSCode for development, please check Developing in WSL.

Working with the CHIP Tool in WSL2

The CHIP Tool (chip-tool) is a Matter controller implementation that allows to commission a Matter device into the network and to communicate with it using Matter messages, which may encode Data Model actions, such as cluster commands.

The tool also provides other utilities specific to Matter, such as parsing of the setup payload or performing discovery actions.

The CHIP Tool requires access to the local network and bluetooth.

Requirements

- Windows 11 64-Bit Pro/Enterprise/Education [for Hyper-V Manager]

Providing access to local network

WSL2 does not share an IP address with your computer. Because WSL2 was implemented with Hyper-V, it runs with a virtualized ethernet adapter. Your computer hides WSL2 behind a NAT where WSL2 has its own unique IP.

To provide an IP address from the local network, WSL2 instance needs to be connected to a virtual Bridge through Hyper-V Manager.

Enabling Hyper-V for use on Windows 10

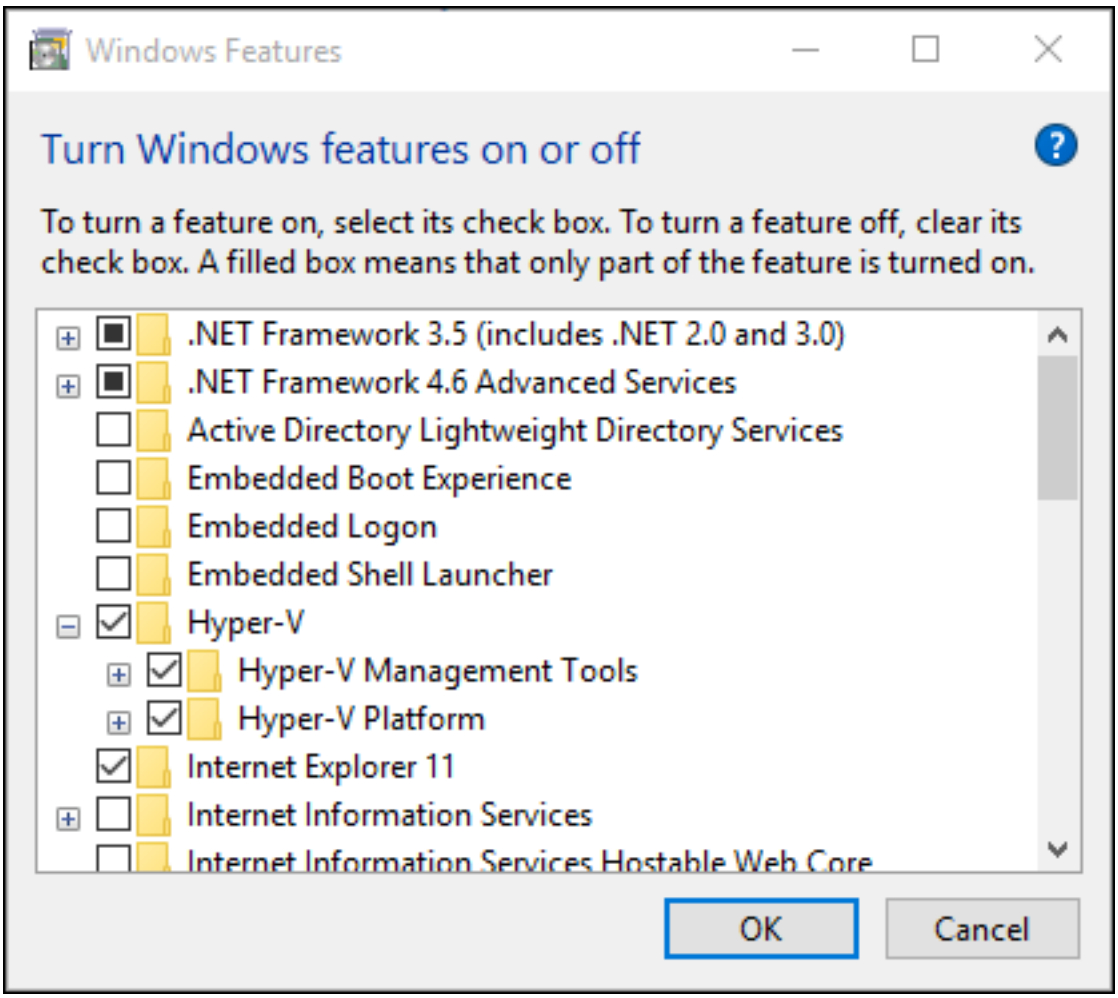
Enable Hardware Virtualization in BIOS

- In the Startup Menu, enter the BIOS setup.
- In the BIOS Setup Utility,open the Configuration or Security tab.
- Enable the Virtualization Technology option

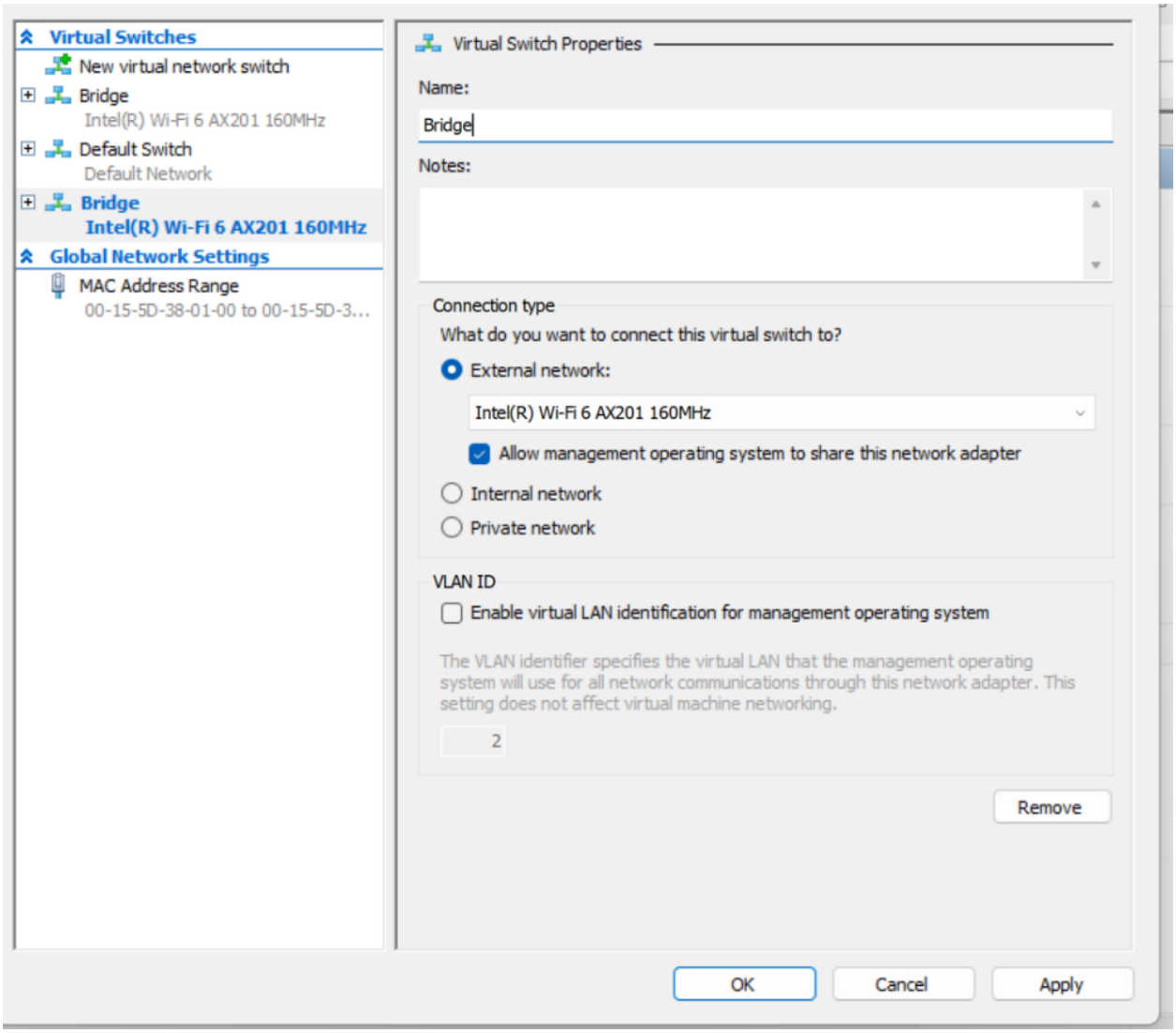
Setting Up Hyper-V

Ensure that hardware virtualization support is turned on in the BIOS settings Save the BIOS settings and boot up the machine normally.

- Right click on the Windows button and select 'Apps and Features'.
- Select Programs and Features on the right under related settings.
- Select Turn Windows Features on or off.
- Select Hyper-V and click OK.



Configure Hyper-V for Bridge Network



- Open Hyper-V Manager.
- From the Actions pane, select Virtual Switch Manager.
- Choose the type of virtual switch as **External**, then select Create Virtual Switch.
- Enter a name for the virtual switch as **Bridge**.
- Choose the wifi network adapter (NIC) that you want to use, then select OK.

You'll be prompted with a warning that the change may disrupt your network connectivity; select Yes if you're happy to continue.

Create .wslconfig file on C:/Users/user-name/ and add the following lines to connect to configured virtual Bridge Network.

```
[wsl2]
networkingMode = bridged
vmSwitch = Bridge
ipv6 = true
```

Checking the configuration

```
ifconfig
```

The IPv4 Address assigned to the eth0 will be from the local network and IPv6 Address will also be assigned to it.

```
avahi-browse _matter._tcp -r
```

If the configuration was done correctly, all the matter operational nodes performing mDns advertisement on the local network will be shown.

Troubleshoot

If no IP address assigned to WSL or same IP address as host, uspipd fails with tcp connect error, not able to ping external network.

1) Run *wsl –shutdown* and wait for WSL instance to close.
2) Run Hyper-V Manager and open Virtual Switch Manager from Actions Pane.
3) Disconnect and Reconnect to Wi-Fi.
4) Start WSL instance.
5) If still no IP is assigned, run *sudo dhclient*

Providing access to bluetooth

Bluetooth support is missing in default WSL kernel. To add support for bluetooth, WSL kernel needs to be recompiled with right drivers.

Building custom kernel for bluetooth access in WSL2

```
git clone --depth 1 --branch linux-msft-wsl-6.1.21.2 https://github.com/microsoft/
↪WSL2-Linux-Kernel.git
```

Replace branch with the latest available WSL-Linux-Kernel tag.

```
cd WSL2-Linux-Kernel
git checkout linux-msft-wsl-6.1.21.2
cp /proc/config.gz config.gz
gunzip config.gz
mv config .config
sudo make menuconfig
```

Select the features to be enabled in the kernel:

1) Enable Networking support ->Bluetooth subsystem support.
2) Enable Networking Support ->Bluetooth Subsystem Support ->Bluetooth device drivers -> HCI USB driver.
3) Save the config file.

```
sudo make -j$(getconf _NPROCESSORS_ONLN) && sudo make modules_install -j$(getconf _
↪NPROCESSORS_ONLN) && sudo make install -j$(getconf _NPROCESSORS_ONLN)
```

The new kernel image will be built.

Copy the new kernel

```
cp arch/x86/boot/bzImage /mnt/path/to/kernel/bluetooth-bzImage
```

Configure WSL to use new custom kernel image

Add the following line to the created *.wslconfig* file.

```
[wsl2]
kernel=c:\\users\\<user>\\bluetooth-bzImage
```

Replace the path with the path of new custom kernel built.

Attaching Bluetooth module to WSL2

Get the BUSID of the bluetooth module. [*Tested using usbipd-win 4.0.0*]

```
usbipd list
```

Attach the bluetooth module to WSL2 instance using usbipd.

```
usbipd attach --wsl --busid=<BUSID>
```

```
$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 003: ID 8087:0029 Intel Corp. AX201 Bluetooth
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

The bluetooth module should be available to WSL.

Testing Bluetooth

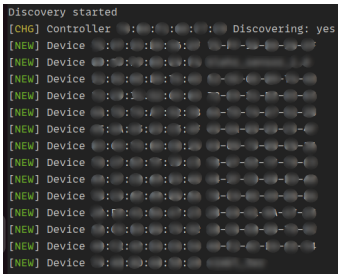
Install bluez library and scan for bluetooth devices.

```
sudo apt install bluez
```

Start scanning for available Bluetooth devices.

```
bluetoothctl scan on
```

The bluetooth discovery should start.



Tested Bluetooth modules:

- Intel AX201 series and above
- Marvell AVASTAR Bluetooth Radio Adapter.

Final .wslconfig file

```
[wsl2]
kernel = D:\\custom-kernel\\bluetooth-bzImage
networkingMode = bridged
vmSwitch = Bridge
ipv6 = true
```

Replace the kernel path appropriately.

ESP-IDF

The Prerequisites for ESP-IDF:

- Please get the Prerequisites for ESP-IDF. For beginners, please check step by step installation guide for esp-idf.

Note: `git clone` command accepts the optional argument `--jobs N`, which can significantly speed up the process by parallelizing submodule cloning. Consider using this option when cloning repositories.

ESP Matter

There are two options to setup ESP-Matter. Choose one according to your needs:

- ESP Matter component: SDK only, added with `idf.py add-dependency`. Lighter setup for product applications; no clone of the esp-matter repository or host tools.
- ESP Matter repository: SDK, examples, and host tools (CHIP-tool, CHIP-cert, ZAP, …). Use this for full development, commissioning, certification, and data-model work.

ESP Matter Component

You can check the component in Espressif Component Registry.

To add the esp_matter component to your project, run:

```
idf.py add-dependency "espressif/esp_matter^1.4.0"
```

An example with esp_matter component is offered:

- Managed Component Light

Note: To use this component, the version of IDF component management should be `1.4.*` or `>= 2.0`. Use `compote version` to show the version. Use `pip install 'idf-component-manager~=1.4.0'` or `pip install 'idf-component-manager~=2.0.0'` to install.

ESP-Matter Repository

The Prerequisites for Matter:

- Please get the Prerequisites for Matter.

Cloning the esp-matter repository takes a while due to a lot of submodules in the upstream connectedhomeip, so if you want to do a shallow clone use the following command:

- For Linux host:

```
cd esp-idf
source ./export.sh
cd ..

git clone --depth 1 https://github.com/espressif/esp-matter.git
cd esp-matter
git submodule update --init --depth 1
cd ./connectedhomeip/connectedhomeip
./scripts/checkout_submodules.py --platform esp32 linux --shallow
cd ../..
./install.sh
cd ..
```

- For Mac OS-X host:

```
cd esp-idf
source ./export.sh
cd ..

git clone --depth 1 https://github.com/espressif/esp-matter.git
cd esp-matter
git submodule update --init --depth 1
cd ./connectedhomeip/connectedhomeip
./scripts/checkout_submodules.py --platform esp32 darwin --shallow
cd ../..
./install.sh
cd ..
```

Note: The modules for platform `linux` or `darwin` are required for the host tools building.

Note: If you don't want to install host tools (chip-tool, chip-cert etc.) you can use `./install.sh --no-host-tool`.

To clone the esp-matter repository with all the submodules, use the following command:

```
cd esp-idf
source ./export.sh
cd ..

git clone --recursive https://github.com/espressif/esp-matter.git
cd esp-matter
./install.sh
cd ..
```

Note: If it runs into some errors like:

```
dial tcp 108.160.167.174:443: connect: connection refused
```

```
ConnectionResetError: [Errno 104] Connection reset by peer
```

It's probably caused by some network connectivity issue, a VPN is required for most of the cases.

Configuring the Environment

This should be done each time a new terminal is opened

```
cd esp-idf; source ./export.sh; cd ..
cd esp-matter; source ./export.sh; cd ..
```

Enable Ccache for faster IDF builds.

Ccache is a compiler cache. Matter builds are very slow and takes a lot of time. Ccache caches the previous compilations and speeds up recompilation in subsequent builds.

```
export IDF_CCACHE_ENABLE=1
```

Above can also be added to your shell's profile file (.profile, .bashrc, .zprofile, etc.) to enable ccache every time you open a new terminal.

1.2.2 Using Docker Image

Note: Skip this step if you have already set up ESP-IDF and ESP-Matter on the host.

The official image `espressif/esp-matter` on Docker Hub already contains ESP-IDF, ESP-Matter, and host tools such as `chip-tool` and `chip-cert`. The container entrypoint sources the IDF and ESP-Matter environment, so you can run `idf.py` without extra setup.

By default, use the `latest` tag:

```
docker pull espressif/esp-matter:latest
```

You can pin another release if needed, for example v1.6:

```
docker pull espressif/esp-matter:release-v1.6
```

Build an example that is already in the image:

```
docker run --rm -it espressif/esp-matter:latest \
    idf.py -C examples/light build
```

Build a project on the host by mounting the current directory:

```
docker run --rm -it -v $PWD:/project -w /project \
    espressif/esp-matter:latest \
    idf.py build
```

Open an interactive shell (IDF and ESP-Matter are already exported):

```
docker run --rm -it -v $PWD:/project -w /project \
    espressif/esp-matter:latest
```

See tools/docker/README.md for build arguments if you need to build a custom image.

1.2.3 Build and Flash

Building Applications

- Light
- Light Switch
- Zap Light
- Zigbee Bridge
- BLE Mesh Bridge

Thread MeshCoP Commissioning

The light example includes an additional configuration for Thread MeshCoP commissioning. After setting up the ESP-IDF and ESP-Matter environment, run the following commands from the `examples/light` directory.

`sdkconfig.defaults.thread_commission` enables the OpenThread joiner, MeshCoP crypto, and Matter Thread MeshCoP commissioning, with Matter-over-BLE disabled.

Flashing the Firmware

Choose IDF target.

- If IDF target has not been set explicitly, then `esp32` is considered as default.
- The default device for `esp32`/`esp32c3` is `esp32-devkit-c`/`esp32c3-devkit-m`. If you want to use another device, you can export `ESP_MATTER_DEVICE_PATH` after choosing the correct target.
  - If the device that you have is of a different revision, and is not working as expected, you can create a new device and export your device path.
  - The other peripheral components like led_driver, button_driver, etc. are selected based on the device selected.
  - The configuration of the peripheral components can be found in `$ESP_MATTER_DEVICE_PATH/esp_matter_device.cmake`.

(When flashing the SDK for the first time, it is recommended to do `idf.py erase_flash` to wipe out entire flash and start out fresh.)

```
idf.py flash monitor
```

Note: If you are getting build errors like:

```
ERROR: This script was called from a virtual environment, can not create a virtual␣
↪environment again
```

It can be fixed by running below command:

```
pip install -r $IDF_PATH/requirements.txt
```

1.2.4 Commissioning and Control

There are a few implementations of Matter commissioners present in the connectedhomeip repository.

CHIP Tool is an example implementation of Matter commissioner and used for development purposes. An in-depth guide on how to use chip-tool can be found in the CHIP Tool User Guide.

Espressif's ESP RainMaker iOS and Android applications support commissioning and control of Matter devices.

- ESP-RainMaker Android App
- ESP-RainMaker iOS App

Test Setup (CHIP Tool)

A host-based chip-tool can be used as a commissioner to commission and control a Matter device. During the previous `install.sh` step, the `chip-tool` is generated under the folder:

```
${ESP_MATTER_PATH}/connectedhomeip/connectedhomeip/out/host
```

Note: **macOS Users**: To use chip-tool with BLE commissioning on macOS, you must install the Bluetooth Central Matter Client Developer Mode Profile. It enables Matter commissioning, and may require periodic re-installation.

Instructions to download the profile can be found in the profile installation section

Commissioning

Use `chip-tool` in interactive mode to commission the device:

```
chip-tool interactive start
```

In the above commands:

- `0x7283` is the randomly chosen `node_id`
- `20202021` is the `setup_passcode`
- `3840` is the `discriminator`

Above method commissions the device using setup passcode and discriminator. Device can also be commissioned using manual pairing code or QR code.

To Commission the device using manual pairing code 34970112332

Above default manual pairing code contains following values:

```
Version:              0
Custom flow:          0       (STANDARD)
Discriminator:        3840
Passcode:             20202021
```

To commission the device using QR code MT:Y.K9042C00KA0648G00

Above QR Code contains the below default values:

```
Version:              0
Vendor ID:            65521    (0xFFF1)
ProductID:            32768    (0x8000)
Custom flow:          0        (STANDARD)
Discovery Bitmask:    0x02     (BLE)
Long discriminator:   3840     (0xf00)
Passcode:             20202021
```

Alternatively, you can scan the below QR code image using Matter commissioners.

If QR code is not visible, paste the below link into the browser and scan the QR code.

```
https://project-chip.github.io/connectedhomeip/qrcode.html?data=MT:Y.K9042C00KA0648G00
```

If you want to use different values for commissioning the device, please use the esp-matter-mfg-tool to generate the factory partition which has to be flashed on the device. It also generates the new pairing code and QR code image using which you can commission the device.

Post Commissioning Setup

The device would need additional configuration depending on the example, for it to work. Check the "Post Commissioning Setup" section in examples for more information.

- Light
- Light Switch
- Zap Light
- Zigbee Bridge
- BLE Mesh Bridge

Cluster Control

Use the cluster commands to control the attributes.

```
onoff toggle 0x7283 0x1
```

```
onoff on 0x7283 0x1
```

```
levelcontrol move-to-level 10 0 0 0 0x7283 0x1
```

```
levelcontrol move-to-level 100 0 0 0 0x7283 0x1
```

```
colorcontrol move-to-color-temperature 0 10 0 0 0x7283 0x1
```

chip-tool when used in interactive mode uses CASE resumption as against establishing CASE for cluster control commands. This results into shorter execution times, thereby improving the overall experience.

For more details about the commands, please check chip-tool usage guide

1.2.5 Device console

The console on the device can be used to run commands for testing. It is configurable through menuconfig and enabled by default in the firmware. Here are some useful commands:

- BLE commands: Start and stop BLE advertisement:

```
matter ble [start|stop|state]
```

- Wi-Fi commands: Set and get the Wi-Fi mode:

```
matter wifi mode [disable|ap|sta]
```

- Device configuration: Dump the device static configuration:

```
matter config
```

- Factory reset:

```
matter esp factoryreset
```

- On-boarding codes: Dump the on-boarding pairing code payloads:

```
matter onboardingcodes
```

Additional Matter specific commands:

- Get attribute: (The IDs are in hex):

```
matter esp attribute get <endpoint_id> <cluster_id> <attribute_id>
```

  - Example: on_off::on_off:

```
matter esp attribute get 0x1 0x6 0x0
```

- Set attribute: (The IDs are in hex):

```
matter esp attribute set <endpoint_id> <cluster_id> <attribute_id> <attribute␣
↪value>
```

  - Example: on_off::on_off:

```
matter esp attribute set 0x1 0x6 0x0 1
```

- Diagnostics:

```
matter esp diagnostics mem-dump
```

- Wi-Fi

```
matter esp wifi connect <ssid> <password>
```

- Bridge device:

```
matter esp bridge <command>
```

- Example: add (Parent endpoint should have aggregator device type):

```
matter esp bridge add <parent_endpoint_id> <device_type_id>
```

1.2.6 Developing your Product

Understanding the structure before actually modifying and customising the device is helpful.

Building a Color Temperature Lightbulb

A device is represented in Matter in terms of its data model. As a first step of building your product, you will have to define the data model for your device. Matter has a standard set of device types already defined that you can use. Please refer to the Espressif Matter Blog for clarity on the terms like endpoints, clusters, etc. that are used in this section.

Data Model

- Typically, the data model is defined in the example's *app_main.cpp*. First off we start by creating a Matter node, which is the root of the Data Model.

```
node::config_t node_config;
node_t *node = node::create(&node_config, app_attribute_update_cb, app_
↪identification_cb, NULL /* optional priv data */);
```

- We will use the `color_temperature_light` standard device type in this case. All standard device types are available in esp_matter_endpoint.h header file. Each device type has a set of default configuration that can be specific as well.

```
color_temperature_light::config_t light_config;
light_config.on_off.on_off = DEFAULT_POWER;
light_config.level_control.current_level = DEFAULT_BRIGHTNESS;
endpoint_t *endpoint = color_temperature_light::create(node, &light_config,␣
↪ENDPOINT_FLAG_NONE, NULL /* priv data */);
```

  In this case, we create the light using the `color_temperature_light::create()` function. Similarly, multiple endpoints can be created on the same node. Check the following sections for more info.

Attribute Callback

- Whenever a Matter client makes changes to the device, they end up updating the attributes in the data model.
- When an attribute is updated, the attribute_update_cb is used to notify the application of this change. You would typically call device driver specific APIs for executing the required action. Here, if the callback type is `PRE_UPDATE`, the driver is updated first. If that is a success, only then the attribute value is actually updated in the database.

```
esp_err_t app_attribute_update_cb(callback_type_t type, uint16_t endpoint_id,␣
↪uint32_t cluster_id,
                                  uint32_t attribute_id, esp_matter_attr_val_t␣
↪*val, void *priv_data)
```

(continues on next page)

(continued from previous page)

```
{
    esp_err_t err = ESP_OK;

    if (type == PRE_UPDATE) {
        /* Driver update */
        err = app_driver_attribute_update(endpoint_id, cluster_id, attribute_id,␣
↪val);
    }

    return err;
}
```

Identify Callback

- This callback is invoked when clients interact with the Identify Cluster. In the callback implementation, an endpoint can identify itself. (e.g., by flashing an LED or light).

```
esp_err_t app_identification_cb(identification::callback_type_t type,␣
↪uint16_t endpoint_id, uint8_t effect_id,
                                        uint8_t effect_variant, void *priv_
↪data)
{
    ESP_LOGI(TAG, "Identification callback: type: %u, effect: %u,␣
↪variant: %u", type, effect_id, effect_variant);
    return ESP_OK;
}
```

Device Drivers

- The drivers, depending on the device, are typically initialized and updated in the example's *app_driver.cpp*.

```
esp_err_t app_driver_init()
{
    ESP_LOGI(TAG, "Initialising driver");

    /* Initialize button */
    button_config_t button_config = button_driver_get_config();
    button_handle_t handle = iot_button_create(&button_config);
    iot_button_register_cb(handle, BUTTON_PRESS_DOWN, app_driver_button_toggle_
↪cb);
    app_reset_button_register(handle);

    /* Initialize led */
    led_driver_config_t led_config = led_driver_get_config();
    led_driver_init(&led_config);

    app_driver_attribute_set_defaults();
    return ESP_OK;
}
```

- The driver's attribute update API just handles the attributes that are actually relevant for the device. For example, a color_temperature_light handles the power, brightness, hue, saturation and temperature.

```
esp_err_t app_driver_attribute_update(uint16_t endpoint_id, uint32_t cluster_id,␣
↪uint32_t attribute_id,
                                      esp_matter_attr_val_t *val)
{
    esp_err_t err = ESP_OK;
    if (endpoint_id == light_endpoint_id) {
        if (cluster_id == OnOff::Id) {
            if (attribute_id == OnOff::Attributes::OnOff::Id) {
                err = app_driver_light_set_power(val);
            }
        } else if (cluster_id == LevelControl::Id) {
            if (attribute_id == LevelControl::Attributes::CurrentLevel::Id) {
                err = app_driver_light_set_brightness(val);
            }
        } else if (cluster_id == ColorControl::Id) {
            if (attribute_id == ColorControl::Attributes::CurrentHue::Id) {
                err = app_driver_light_set_hue(val);
            } else if (attribute_id ==␣
↪ColorControl::Attributes::CurrentSaturation::Id) {
                err = app_driver_light_set_saturation(val);
            } else if (attribute_id ==␣
↪ColorControl::Attributes::ColorTemperature::Id) {
                err = app_driver_light_set_temperature(val);
            }
        }
    }
    return err;
}
```

Defining your own data model

This section demonstrates creating standard endpoints, clusters, attributes, and commands that are defined in the Matter specification

Endpoints

The device can be customized by editing the endpoint/device_type creating in the *app_main.cpp* of the example. Examples:

- on_off_light:

```
on_off_light::config_t light_config;
endpoint_t *endpoint = on_off_light::create(node, &light_config, ENDPOINT_FLAG_
↪NONE, NULL /* priv data */);
```

- fan:

```
fan::config_t fan_config;
endpoint_t *endpoint = fan::create(node, &fan_config, ENDPOINT_FLAG_NONE, NULL /*␣
↪priv data */);
```

- door_lock:

```
door_lock::config_t door_lock_config;
endpoint_t *endpoint = door_lock::create(node, &door_lock_config, ENDPOINT_FLAG_
↪NONE, NULL /* priv data */);
```

- window_covering:

```
window_covering::config_t window_covering_config(static_cast<uint8_t>
↪(chip::app::Clusters::WindowCovering::EndProductType::kTiltOnlyInteriorBlind));
endpoint_t *endpoint = window_covering_device::create(node, &window_covering_
↪config, ENDPOINT_FLAG_NONE, NULL /* priv data */);
```

The `window_covering` `config_t` structure includes a constructor that allows specifying an end product type different than the default one, which is "Roller shade". Once a `config_t` instance has been instantiated, its end product type cannot be modified.

- pump

```
pump::config_t pump_config(1, 10, 20);
endpoint_t *endpoint = pump::create(node, &pump_config, ENDPOINT_FLAG_
↪NONE, NULL /* priv data */);
```

The `pump` `config_t` structure includes a constructor that allows specifying maximum pressure, maximum speed and maximum flow values. If they aren't set, they will be set to null by default. Once a `config_t` instance has been instantiated, these three values cannot be modified.

Clusters

Additional clusters can also be added to an endpoint. Examples:

- on_off:

```
on_off::config_t on_off_config;
cluster_t *cluster = on_off::create(endpoint, &on_off_config, CLUSTER_FLAG_
↪SERVER);
```

- temperature_measurement:

```
temperature_measurement::config_t temperature_measurement_config;
cluster_t *cluster = temperature_measurement::create(endpoint, &temperature_
↪measurement_config, CLUSTER_FLAG_SERVER);
```

- window_covering:

```
window_covering::config_t window_covering_config(static_cast<uint8_
↪t>
↪(chip::app::Clusters::WindowCovering::EndProductType::kTiltOnlyInteriorBlind));
↪
window_covering_config.feature_flags = window_
↪covering::feature::lift::get_id();
cluster_t *cluster = window_covering::create(endpoint, &window_
↪covering_config, CLUSTER_FLAG_SERVER);
```

The `window_covering` `config_t` structure includes a constructor that allows specifying an end product type different than the default one, which is "Roller shade". Once a `config_t` instance has been instantiated, its end product type cannot be modified.

- pump_configuration_and_control:

```
pump_configuration_and_control::config_t pump_configuration_and_control_
↪config(1, 10, 20);
pump_configuration_and_control_config..feature_flags = pump_configuration_
↪and_control::feature::constant_pressure::get_id();
cluster_t *cluster = pump_configuration_and_control::create(endpoint, &
↪pump_configuration_and_control_config, CLUSTER_FLAG_SERVER);
```

The `pump_configuration_and_control` `config_t` structure includes a constructor that allows specifying maximum pressure, maximum speed and maximum flow values. If they aren't set, they will be set to null by default. Once a `config_t` instance has been instantiated, these three values cannot be modified.

Attributes and Commands

Additional attributes and commands can also be added to a cluster. Examples:

- attribute: on_off:

```
bool default_global_scene_control = true;
attribute_t *attribute = on_off::attribute::create_global_scene_control(cluster,␣
↪default_global_scene_control);
```

- command: toggle:

```
command_t *command = on_off::command::create_toggle(cluster);
```

- command: move_to_level:

```
command_t *command = level_control::command::create_move_to_level(cluster);
```

Features

Mandatory features for a device type or endpoint can be configured at endpoint level.

- feature: lighting: On/Off cluster:

```
extended_color_light::config_t light_config;
light_config.on_off_lighting.start_up_on_off = nullptr;
endpoint_t *endpoint = extended_color_light::create(node, &light_config, ENDPOINT_
↪FLAG_NONE, nullptr /* priv data */);
```

Few of some mandatory feature for a cluster (i.e. cluster having O.a/O.a+ feature conformance) can be configured at cluster level. For example: Thermostat cluster has O.a+ conformance for Heating and Cooling features, that means at least one of them should be present on the thermostat cluster while creating it.

- feature: heating: Thermostat cluster:

```
thermostat::config_t thermostat_config;
thermostat_config.features.heating.occupied_heating_setpoint = 2200;
thermostat_config.feature_flags = thermostat::feature::heating::get_id();
cluster::thermostat::create(endpoint, &(config->thermostat_config), CLUSTER_FLAG_
↪SERVER);
```

Optional features which are applicable to a cluster can also be added.

- feature: tag_list: Descriptor cluster:

```
cluster_t* cluster = cluster::get(endpoint, Descriptor::Id);
descriptor::feature::tag_list::add(cluster);
```

Adding custom data model fields

This section demonstrates creating custom endpoints, clusters, attributes, and commands that are not defined in the Matter specification and can be specific to the vendor.

Endpoints

Non-Standard endpoint can be created, without any clusters.

- Endpoint create:

```
endpoint_t *endpoint = endpoint::create(node, ENDPOINT_FLAG_NONE);
```

Clusters

Non-Standard/Custom clusters can also be created:

- Cluster create:

```
uint32_t custom_cluster_id = 0x131bfc00;
cluster_t *cluster = cluster::create(endpoint, custom_cluster_id, CLUSTER_FLAG_
↪SERVER);
```

Attributes and Commands

Non-Standard/Custom attributes can also be created on any cluster:

- Attribute create:

```
uint32_t custom_attribute_id = 0x0;
uint16_t default_value = 100;
attribute_t *attribute = attribute::create(cluster, custom_attribute_id,␣
↪ATTRIBUTE_FLAG_NONE, esp_matter_uint16(default_value);
```

- Command create:

```
static esp_err_t command_callback(const ConcreteCommandPath &command_path,␣
↪TLVReader &tlv_data, void
*opaque_ptr)
{
   ESP_LOGI(TAG, "Custom command callback");
   return ESP_OK;
}

uint32_t custom_command_id = 0x0;
command_t *command = command::create(cluster, custom_command_id, COMMAND_FLAG_
↪ACCEPTED, command_callback);
```

Advanced Setup

This section explains adding external platforms for Matter. This step is **optional** for most devices. Espressif's SDK for Matter provides support for overriding the default platform layer, so the BLE and Wi-Fi implementations can be customized. Here are the required steps for adding an external platform layer.

Creating the external platform directory

Create a directory `platform/${NEW_PLATFORM_NAME}` in your codebase. You can typically copy `${ESP_MATTER_PATH}/connectedhomeip/connectedhomeip/src/platform/ESP32` as a start. Note that the new platform name should be something other than `ESP32`. In this article we'll use `ESP32_custom` as an example. The directory must be under `platform` folder to meet the Matter include path conventions.

Modifying the BUILD.gn target

There is an example BUILD.gn file for the `ESP32_custom` example platform. It simply compiles the ESP32 platform in Matter without any modifications.

- The new platform directory must be added to the Matter include path. See the `ESP32_custom_include` config in the above mentioned file.
- Multiple build configs must be exported to the build system. See the `buildconfig_header` section in the file for the required definitions.

Editing Kconfigs

- Enable `CONFIG_CHIP_ENABLE_EXTERNAL_PLATFORM`.
- Set `CONFIG_CHIP_EXTERNAL_PLATFORM_DIR` to the relative path from `${ESP_MATTER_PATH}/connectedhomeip/connectedhomeip/config/esp32` to the external platform directory. For instance, if your source tree is:

```
my_project
├── esp-matter
└── platform
    └── ESP32_custom
```

  Then `CONFIG_CHIP_EXTERNAL_PLATFORM_DIR` would be `../../../../../platform/ESP32_custom`.
- Disable `CONFIG_BUILD_CHIP_TESTS`.
- If your external platform does not support the *connectedhomeip/connectedhomeip/src/lib/shell/* provided in the Matter shell library, then disable `CONFIG_ENABLE_CHIP_SHELL`.

Example Usage

As an example, you can build *light* example on `ESP32_custom` platform with following steps:

```
mkdir $ESP_MATTER_PATH/../platform
cp -r $ESP_MATTER_PATH/connectedhomeip/connectedhomeip/src/platform/ESP32 $ESP_MATTER_
↪PATH/../platform/ESP32_custom
cp $ESP_MATTER_PATH/examples/common/external_platform/BUILD.gn $ESP_MATTER_PATH/../
↪platform/ESP32_custom
cd $ESP_MATTER_PATH/examples/light
cp sdkconfig.defaults.ext_plat sdkconfig.defaults
idf.py build
```

1.2.7 Factory Data Providers

Providers Introduction

There are four factory data providers, each with its own implementation, that need to be configured. These providers supply the device with necessary factory data, which is then read by the device according to their respective implementations.

- `Commissionable Data Provider`

  This particular provider is responsible for retrieving commissionable data, which includes information such as setup-discriminator, spake2p-iteration-count, spake2p-salt, spake2p-verifier, and setup-passcode.

- `Device Attestation Credentials(DAC) Provider`

  This particular provider is responsible for retrieving device attestation credentials, which includes information such as CD, firmware-information, DAC, and PAI certificate. And it can also sign message with the DAC private key.

- `Device Instance Info Provider`

  This particular provider is responsible for retrieving device instance information, which includes vendor-name, vendor-id, product-name, product-id, product-url, product-label, hardware-version-string, hardware-version, rotating-device-id-unique-id, serial-number, manufacturing-data, and part-number.

- `Device Info Provider`

  This particular provider is responsible for retrieving device information, which includes fixed-labels, user-labels, supported-locales, and supported-calendar-types.

Configuration Options

Different implementations of the four providers can be chosen in meuconfig:

- `Commissionable Data Provider options` in → `Component config` → `ESP Matter`
  - `Commissionable Data - Test`, the device will use the hardcoded Commissionable Data. This uses the legacy commissionable data provider and provides the test values. These test values are enclosed in `CONFIG_ENABLE_TEST_SETUP_PARAMS` option and enabled by default.
  - `Commissionable Data - Factory`, the device will use commissionable data information from the factory partition.This option is visible only when `CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER` is selected.

  - `Commissionable Data - Secure Cert`, the device will use commissionable data information from the secure cert partition. This option is only visible when `CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER` and `CONFIG_SEC_CERT_DAC_PROVIDER` is enabled.
  - `Commissionable Data - Custom`, the device will use the custom defined commissionable data provider to obtain commissionable data information. `esp_matter::set_custom_commissionable_data_provider()` should be called before `esp_matter::start()` to set the custom provider.

Note: If you are using `Factory`, `Secure Cert` or `Custom` commissionable data provides, then disable the `CONFIG_ENABLE_TEST_SETUP_PARAMS` option.

- `DAC Provider options` in → `Component config → ESP Matter`
  - `Attestation - Test`, the device will use the hardcoded Device Attestation Credentials.
  - `Attestation - Factory`, the device will use the Device Attestation Credentials in the factory partition binary. This option is visible only when `CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER` is selected.
  - `Attestation - Secure Cert`, the device will use the Device Attestation Credentials in the secure cert partition. This option is for the Pre-Provisioned Modules. And the original vendor ID and product ID should be added to the CD file for the Pre-Provisioned Modules. Please contact your Espressif contact person for more information.
  - `Attestation - Custom`, the device will use the custom defined DAC provider to obtain the Device Attestation Credentials. `esp_matter::set_custom_dac_provider()` should be called before `esp_matter::start()` to set the custom provider.
- `Device Instance Info Provider options` in → `Component config → ESP Matter`
  - `Device Instance Info - Test`, the device will use the hardcoded Device Instance Information.
  - `Device Instance Info - Factory`, the device will use device instance information from the factory partition. This option is visible only when `CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER` and `ENABLE_ESP32_DEVICE_INSTANCE_INFO_PROVIDER` is selected.
  - `Device Instance Info - Secure Cert`, the device will use the unique identifier for generating the rotating device identifier from the secure cert partition and all other details will be read from the factory partition. This option is only visible when `CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER` and `CONFIG_SEC_CERT_DAC_PROVIDER` is enabled.
  - `Device Instance Info - Custom`, the device will use custom defined Device Instance Info Provider to obtain the Device Instance Information. `esp_matter::set_custom_device_instance_info_provider` should be called before `esp_matter::start()` to set the custom provider.
- `Device Info Provider options` in → `Component config → ESP Matter`
  - `Device Info - None`, the device will not use any device information provider. It should be selected when there are not related clusters on the device.
  - `Device Info - Factory`, the device will use device information from the factory partition. This option is visible only when `CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER` and `ENABLE_ESP32_DEVICE_INFO_PROVIDER` is selected.
  - `Device Info - Custom`, the device will use custom defined Device Info Provider to obtain the Device Information. `esp_matter::set_custom_device_info_provider` should be called before `esp_matter::start()` to set the custom provider.

Custom Providers

In order to use custom providers, you need to define implementations of the four base classes of the providers and override the functions within them. And the custom providers should be set before `esp_matter::start()` is called.

1.2.8 Using esp_secure_cert partition

Configuration Options

Build the firmware with below configuration options

```
# Disable the DS Peripheral support
CONFIG_ESP_SECURE_CERT_DS_PERIPHERAL=n

# Use DAC Provider implementation which reads attestation data from secure cert␣
↪partition
CONFIG_SEC_CERT_DAC_PROVIDER=y

# Enable some options which reads CD and other basic info from the factory partition
CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER=y
CONFIG_ENABLE_ESP32_DEVICE_INSTANCE_INFO_PROVIDER=y
CONFIG_FACTORY_COMMISSIONABLE_DATA_PROVIDER=y
CONFIG_FACTORY_DEVICE_INSTANCE_INFO_PROVIDER=y
```

Certification Declaration

If you do not have an certification declaration file then you can generate the test CD with the help of below mentioned steps. We need to generate the new CD because it SHALL match the VID, PID in DAC and the ones reported by basic cluster.

- Build the host tools if not done already

```
cd connectedhomeip/connectedhomeip
gn gen out/host
ninja -C build
```

Generate the Test CD, please make sure to change the `-V` (vendor_id) and `-p` (product-id) options based on the ones that are being used. For more info about the arguments, please check chip-cert's gen-cd command in the connectedhomeip repository.

```
out/host/chip-cert gen-cd -f 1 -V 0xFFF1 -p 0x8001 -d 0x0016 \
                          -c "CSA00000SWC00000-01" -l 0 -i 0 -n 1 -t 0 \
                          -K credentials/test/certification-declaration/Chip-Test-CD-
↪Signing-Key.pem \
                          -C credentials/test/certification-declaration/Chip-Test-CD-
↪Signing-Cert.pem \
                          -O TEST_CD_FFF1_8001.der
```

Factory Partition

Factory partition contains basic information like VID, PID, etc.

By default, the CD(Certification Declaration) is stored in the factory partition and we need to add the `-cd` option when generating the factory partition.

Alternatively, if you'd like to embed the CD in the firmware, you can enable the `CONFIG_ENABLE_SET_CERT_DECLARATION_API` option and use the `SetCertificationDeclaration()` API to set the CD. You can refer to the reference implementation in :project_file: light example.

Export the dependent tools path

```
cd esp-matter
export PATH=$PATH:$PWD/connectedhomeip/connectedhomeip/out/host
```

Generate the factory partition, please use the APPROPRIATE values for `-v` (Vendor Id), `-p` (Product Id), and `-cd` (Certification Declaration).

```
esp-matter-mfg-tool --passcode 89674523 \
              --discriminator 2245 \
              -cd TEST_CD_FFF1_8001.der \
              -v 0xFFF1 --vendor-name Espressif \
              -p 0x8001 --product-name Bulb \
              --hw-ver 1 --hw-ver-str DevKit
```

Few important output lines are mentioned below. Please take a note of onboarding codes, these can be used for commissioning the device.

```
[2022-12-02 11:18:12,059] [   INFO] - Generated QR code: MT:-24J06PF150QJ850Y10
[2022-12-02 11:18:12,059] [   INFO] - Generated manual code: 20489154736
```

Factory partition binary will be generated at the below path. Please check for <uuid>.bin file in this directory.

```
[2022-12-02 11:18:12,381] [   INFO] - Generated output files at: out/fff1_8001/
↪e17c95e1-521e-4979-b90b-04da648e21bb
```

Flashing firmware, secure cert and factory partition

Flashing secure cert partition. Please check partition table for `esp_secure_cert` partition address.

Note: Flash only if not flashed on manufacturing line.

```
esptool.py -p (PORT) write_flash 0xd000 secure_cert_partition.bin
```

Flashing factory partition, Please check the `CONFIG_CHIP_FACTORY_NAMESPACE_PARTITION_LABEL` for factory partition label. Then check the partition table for address and flash at that address.

```
esptool.py -p (PORT) write_flash 0x10000 path/to/partition/generated/using/mfg_tool/
↪uuid.bin
```

Flash application

```
idf.py flash
```

Test commissioning using chip-tool

If using the DACs signed by custom PAA that is not present in connectedhomeip repository, then download the PAA certificate, please make sure it is in DER format.

Run the following command from host to commission the device.

```
./chip-tool pairing ble-wifi 1234 my_SSID my_PASSPHRASE my_PASSCODE my_DISCRIMINATOR -
↪-paa-trust-store-path /path/to/PAA-Certificates/
```

1.2.9 Matter OTA

- Enable the `CONFIG_ENABLE_OTA_REQUESTOR` option to enable Matter OTA Requestor functionality.

Please follow the OTA guide in the connectedhomeip repository for generating a Matter OTA image and performing OTA.

Encrypted Matter OTA

The esp-matter SDK supports using a pre-encrypted application image for OTA upgrades. Please follow the steps below to enable and use encrypted application images for OTA upgrades.

- Enable the `CONFIG_ENABLE_OTA_REQUESTOR` and `CONFIG_ENABLE_ENCRYPTED_OTA` options
- The application code must make an API call to `esp_matter_ota_requestor_encrypted_init()` after calling `esp_matter::start()`. You can use the following code as a reference:

```
#include <esp_matter_ota.h>

{
    const char *rsa_private_key;     // Please set this to the buffer containing RSA␣
↪3072 private key in PEM format
    uint16_t rsa_private_key_len;    // Please set this to the length of RSA 3072␣
↪private key

    esp_err_t err = esp_matter_ota_requestor_encrypted_init(rsa_private_key, rsa_
↪private_key_len);
}
```

- Please refer to the encrypted OTA guide in the connectedhomeip repository for instructions on how to generate a private key, encrypted OTA image, and Matter OTA image.

Note: There are several ways to store the private key, such as hardcoding it in the firmware, embedding it as a text file, or reading it from the NVS. We have demonstrated the use of the private key by embedding it as a text file in the light example.

1.2.10 Mode Select

This cluster provides an interface for controlling a characteristic of a device that can be set to one of several predefined values. For example, the light pattern of a disco ball, the mode of a massage chair, or the wash cycle of a laundry machine.

Attribute Supported Modes

This attribute is the list of supported modes that may be selected for the CurrentMode attribute. Each item in this list represents a unique mode as indicated by the Mode field of the ModeOptionStruct. Each entry in this list SHALL have a unique value for the Mode field. ESP_MATTER uses factory partition to set the values of Supported Modes attribute.

Generate Factory Partition Using esp-matter-mfg-tool

Use esp-matter-mfg-tool to generate factory partition of the supported modes attribute.

Usage

```
esp-matter-mfg-tool -cn "My bulb" -v 0xFFF2 -p 0x8001 --pai \
-k path/to/esp-matter/connectedhomeip/connectedhomeip/credentials/test/attestation/
↪Chip-Test-PAI-FFF2-8001-Key.pem \
-c path/to/esp-matter/connectedhomeip/connectedhomeip/credentials/test/attestation/
↪Chip-Test-PAI-FFF2-8001-Cert.pem \
-cd path/to/esp-matter/connectedhomeip/connectedhomeip/credentials/test/certification-
↪declaration/Chip-Test-CD-FFF2-8001.der \
--supported-modes mode1/label1/endpointId/"value\mfgCode, value\mfgCode"  mode2/
↪label2/endpointId/"value\mfgCode, value\mfgCode"
```

- For empty Semantic Tags list

```
--supported-modes mode1/label1/endpointId  mode2/label2/endpointId
```

Build example

For example we want to use mode_select cluster in light example.

- Add source and include path to example/light/main/CMakeList.txt

```
Append "${MATTER_SDK_PATH}/examples/platform/esp32/mode-support" to SRC_DIRS and PRIV_
↪INCLUDE_DIRS
```

- In file example/light/app_main.cpp.

```
#include <static-supported-modes-manager.h>

static ModeSelect::StaticSupportedModesManager sStaticSupportedModesManager;
{
    cluster::mode_select::config_t ms_config;
    cluster_t *ms_cluster = cluster::mode_select::create(endpoint, &ms_config,␣
↪CLUSTER_FLAG_SERVER);

    sStaticSupportedModesManager.InitEndpointArray(get_count(node));
    ModeSelect::setSupportedModesManager(&sStaticSupportedModesManager);
}
```

1.2.11 Custom Cluster

Matter enables users to implement custom clusters for unique features. This section introduces how to add a custom cluster.

Cluster XML Template

Before adding a custom cluster, you should design the attributes, commands, and events it will include, and create the cluster XML template file based on your design.

Example:

```
<?xml version="1.0"?>
<configurator>
  <domain name="CHIP"/>
  <cluster>
    <domain>General</domain>
    <name>Sample ESP</name>
    <!-- The MSB 16 bits of <code> are the VendorID. Replace this with your
         VendorID. 0x131B is the VendorId of Espressif.
         The LSB 16 bits of <code> are a self-assigned ClusterID -->
    <code>0x131BFC20</code>
    <define>SAMPLE_ESP_CLUSTER</define>
    <description>The Sample ESP cluster showcases a manufacturer custom cluster</
↪description>

    <!-- Attributes -->
    <!-- A simple test boolean attribute -->
    <attribute side="server" code="0x0000" define="SAMPLE_BOOLEAN" type="boolean"␣
↪writable="true" default="false" optional="false">SampleBoolean</attribute>
    <attribute side="server" code="0x0001" define="SAMPLE_CHAR_STR" type="char_string
↪" writable="false" optional="false">SampleCharStr</attribute>

    <!-- Commands -->
    <command source="client" code="0x00" name="CommandwithoutArgs" optional="false">
      <description>
        Simple command without any parameters and without a response.
      </description>
    </command>

    <command source="client" code="0x01" name="CommandWithArgs" response=
↪"CommandWithArgsResponse" optional="false">
      <description>
        Command that takes two uint8 arguments.
      </description>
      <arg name="Arg1" type="int8u"/>
      <arg name="Arg2" type="int8u"/>
    </command>

    <!-- Command Responses -->
    <command source="server" code="0x02" name="CommandWithArgsResponse" optional=
↪"false" disableDefaultResponse="true">
      <description>
        Response for CommandwithArgs.
      </description>
      <arg name="ResponseArg" type="int8u"/>
    </command>
```

(continues on next page)

(continued from previous page)

```
    <!-- Events -->
    <event side="server" code="0x0000" name="TestEvent" priority="info"␣
↪isFabricSensitive="true" optional="false">
      <description>
        Example event with a event data
      </description>
      <field id="1" name="EventData" type="int32u"/>
    </event>
  </cluster>
</configurator>
```

The example XML file above illustrates a cluster with two attributes, two accepted commands, one generated command(command response), and one event.

After creating the custom cluster XML template file, add the root directory of your template file to the `xmlRoot` array and the template file name to the `xmlFile` array in both the zcl configuration file and the zcl test configuration file.

Run `zap_regen_all.py` in Matter virtual environment to generate common code and client code for the custom cluster.

```
cd esp_matter/connectedhomeip/connectedhomeip
source ./scripts/activate.sh
./scripts/tools/zap_regen_all.py
```

The code generation script will create client code for the custom cluster, supporting Android, Darwin, and Python controllers, as well as the chip-tool. It will also generate app-common code for the new custom cluster. The chip-tool can be used to test the custom cluster after recompiling.

Cluster Implementation

The custom cluster should be implemented after the app-common code has been generated.

Custom Cluster Attributes

The attributes in a cluster can be managed with the two following methods. A cluster can utilize both the methods to manage its attributes.

- Attribute Accessors

  By default, all the attributes are stored in the ZCL data model and can be managed with the Attribute Accessors generated in the app-common code. You can set/get the attribute values with the Accessors APIs.

- Attribute Access Interface (AAI)

  Matter provides a virtual class, `AttributeAccessInterface`, which can be inherited by the custom cluster to manage its attributes. Attributes managed by AAI should be added to `attributeAccessInterfaceAttributes` array in both the zcl configuration file and the zcl test configuration file. Then, run the `zap_regen_all.py` to regenerate the app-common code. Once the code is regenerated, the Attribute Accessors APIs for attributes managed by AAI will be removed.

  Notes that attributes of complex types(structure or array) cannot be handled by Attribute Accessors and MUST be managed using AAI.

Custom Cluster Commands

The commands in a cluster can be handled with one of the two following methods. A cluster can only choose one method to handle its commands.

- Ember Command callbacks

  By default, all the commands are handled using Ember command callbacks. The `zap` tool generates declarations for these callbacks in the app-common code. And the corresponding definitions should be implemented to use the commands within the clusters.

- Command Handler Interface (CHI)

  Matter also provides a virtual class, `CommandHandlerInterface`, which can be inherited to handle commands within the cluster. If the commands in a cluster are handled by CHI. The cluster should be added to the `CommandHandlerInterfaceOnlyClusters` array in the zap configuration data file. After modifying the zap configuration data, the code should be regenerated, which will remove the Ember command callback declarations.

Custom Cluster Events

All the events are managed by the EventLogging. If an event is triggered, `chip::app::LogEvent()` can be called to record it. The event will then be reported to the subscriber that has subscribed to it.

Custom Cluster Functions

A custom cluster might requires special functions to handle initialization, attribute changes, shutdown, and pre-attribute changes. For instance, the AAI and CHI need to be registered so that they can be accessed by the Matter data model. Therefore, the cluster requires an initialization function to register them. To enable these functions, the cluster should be added to the appropriate entry in the zap configuration data file.

Example Usage

- Zap Example

  If the example uses `zap` tool to generate its data model, the custom cluster should be added to the example's zap file. The `zap` tool will then generate the data model code, including the custom cluster, during the building process.

- ESP-Matter Example

  If the example uses ESP-Matter APIs to define its data model, the custom data model should be created and added to the data model using the esp-matter APIs, following the instructions in Adding custom data model fields

1.3 Matter Controller

This section introduces the `esp_matter_controller` component, which can be used to build Matter controller and commissioner on Espressif SoCs.

Several controller/commissioner examples are provided.

- Basic Commissioner

This example implements a basic Matter commissioner which can be used for commissioning Matter commissionees to its Matter Fabric.

- Controller with Server Instance

This example implements a Matter controller in RainMaker Matter Fabric. It is also a Matter server so it is commissioned to RainMaker Matter Fabic via standard Matter commissioning flow. And it uses a custom Matter cluster to update its Node Operational Credentials (NOC) and obtain the device list of the Matter Fabric.

- Client-only Controller

This example implements a Matter controller in RainMaker Matter Fabric without a Matter server instance. It can be provisioned to a Wi-Fi network via network_provisioning. And it uses a custom RainMaker service to obtain Administer NOC and device list of the Matter Fabric.

The device console uses these command prefixes:

- `matter esp controller` runs controller commands.
- `matter esp group` configures groups and keys.
- `matter esp device` manages devices in the controller example. See the example README for its commands.

1.3.1 Console modules

Include `esp_matter_controller_console.h` and register each module you need once, before starting the shell. The first controller module registers the command root.

```
esp_matter::controller::console::register_read_commands();
esp_matter::controller::console::register_write_commands();
esp_matter::controller::console::register_invoke_commands();
esp_matter::controller::console::register_subscribe_commands();
esp_matter::controller::console::register_groups_commands();
ESP_ERROR_CHECK_WITHOUT_ABORT(esp_matter::console::init());
```

The Read module handles attributes and events. The Subscribe module creates, lists, and removes subscriptions. Register window, pairing, or ICD commands with `register_window_commands()`, `register_pairing_commands()`, or `register_icd_commands()`.

Pairing requires `CONFIG_ESP_MATTER_COMMISSIONER_ENABLE`. ICD and group-settings registration require a build without Matter server support. Group settings use the separate `matter esp group` root. Group Invoke and Write belong to the Invoke and Write modules and work with or without a Matter server.

Registration can fail if the console has no command-set capacity left. Modules use separate translation units and do not register themselves automatically. The linker can remove unreferenced handlers, help strings, and state. Referenced APIs and shared dependencies remain linked.

An application with its own pairing CLI can call `pairing::interaction` without registering the generic pairing console.

Console input

Only registered modules provide commands. Put options such as `--timed-ms` before `--`, followed by one JSON argument. Write takes a tagged attribute value. Invoke takes a command-fields object, or `{}` for no fields. Use shell quoting and escape embedded double quotes and backslashes.

The ESP-IDF line editor filters non-ASCII input. Use JSON Unicode escapes such as `\uXXXX`, entered as `\\uXXXX` through shell escaping. `CONFIG_CHIP_SHELL_CMD_LINE_BUF_MAX_LENGTH` limits line length, and `CONFIG_CHIP_SHELL_CMD_LINE_ARG_MAX_COUNT` limits the argument count.

1.3.2 Controller features

The controller is the role of a node that has permissions to enable it to control one or more nodes. It has a complete chain of Node Operational Credentials (NOC) and acts as an administer or an operator in its Matter fabric. And the NodeId of the controller or CASE Authenticated Tag (CAT) of the controller's NOC should in the Access Control List of other end-devices with administer/operator privilege.

The controller should support the following features:

- Send Cluster Invoking commands to other nodes.
- Send Read Attribute/Event commands to other nodes.
- Send Write Attribute commands to other nodes
- Send Subscribe Attribute/Event commands to other nodes.
- Join Matter Groups and manage the Group Key Set.
- Register and unregister ICD clients.

Cluster Invoking commands

Use `invoke` to send unicast or group cluster commands. In C++, use `invoke_command::interaction` with a request and an explicit delegate pointer. Pass `nullptr` to discard successful responses while retaining failure logs. `OnResponse()` receives the concrete command path, status, and optional response TLV; `OnDone()` reports terminal completion. See *Ownership and callbacks* for ownership and error-handling requirements.

- Send the cluster command:

```
matter esp controller invoke <node-id> <endpoint-id> <cluster-id> <command-id> [--
↪timed-ms N] -- <command-data>
matter esp controller invoke group <group-id> <cluster-id> <command-id> --
↪<command-data>
```

Note:

- The `command-data` should utilize a JSON object string and the name of each item in this object should be `\"<TagNumber>:<DataType>\"` or `\"<TagName>:<TagNumber>:<DataType>\"`. The TagNumber should be the same as the command parameter ID in Matter SPEC and the supported DataTypes are listed in `$ESP_MATTER_PATH/components/esp_matter/utils/json_to_tlv.h`
- For the DataType `bytes`, the value should be a Base64-Encoded string.

Here are some examples of the `command-data` format.

- For MoveToLevel command in LevelControl cluster, the `command-data` (`{"level": 10, "transitionTime": 0, "optionsMask": 0, "optionsOverride": 0}`) should be:

```
matter esp controller invoke <node-id> <endpoint-id> 8 0 -- "{\"0:U8\": 10, \
↪"1:U16\": 0, \"2:U8\": 0, \"3:U8\": 0}"
```

- For KeySetWrite command in GroupKeyManagement cluster, the `command-data` (`{"groupKeySet":{"groupKeySetID": 42, "groupKeySecurityPolicy": 0, "epochKey0": d0d1d2d3d4d5d6d7d8d9dadbdcdddedf, "epochStartTime0": 2220000, "epochKey1": null, "epochStartTime1": null, "epochKey2": null, "epochStartTime2": null}}`) should be:

```
matter esp controller invoke <node-id> <endpoint-id> 63 0 -- "{\"0:OBJ\": {\
↪"0:U16\": 42, \"1:U8\": 0, \"2:BYT\": \"0NHS09TV1tfY2drb3N3e3w==\", \"3:U64\":␣
↪2220000, \"4:NULL\": null, \"5:NULL\": null, \"6:NULL\": null, \"7:NULL\": null}
↪}"
```

- For AddGroup command in Groups cluster, the `command-data` (`{"groupID": 1, "groupName": "grp1"}`) should be:

```
matter esp controller invoke <node-id> <endpoint-id> 0x4 0 -- "{\"0:U16\": 1, \
↪"1:STR\": \"grp1\"}"
```

Read commands

Use `read_command::interaction` with attribute and/or event path spans. Its delegate receives `OnReportBegin()`, `OnAttribute()`, `OnEvent()`, `OnReportEnd(err)`, and terminal `OnDone(err)` callbacks. A synchronous `start()` failure produces no callbacks.

Read attribute commands

Use `read attr` to read attributes from end-devices.

- Attribute read syntax:

```
matter esp controller read attr <node-id> <endpoint-ids> <cluster-ids> <attribute-
↪ids>
```

Note:

- endpoint-ids can represent a single or multiple endpoints, e.g. '0' or '0,1'. And the same applies to cluster-ids, attribute-ids, and event-ids below.

Read event commands

Use `read event` to read events from end-devices.

- Event read syntax:

```
matter esp controller read event <node-id> <endpoint-ids> <cluster-ids> <event-
↪ids>
```

Write attribute commands

`write attr` writes one complete attribute per command. In C++, use `write_command::interaction` with a concrete attribute path, value encoder, and explicit delegate pointer. Pass `nullptr` to discard successful responses while retaining failure logs. A delegate gets the path status in `OnResponse()` and terminal completion in `OnDone(err)`.

- Attribute write syntax:

```
matter esp controller write attr <node-id> <endpoint-id> <cluster-id> <attribute-
↪id> [--timed-ms N] -- <attribute-value>
matter esp controller write attr group <group-id> <cluster-id> <attribute-id> --
↪<attribute-value>
```

Note:

- `attribute_value` should utilize a JSON object string. And the format of this string is the same as the `command_data` in cluster commands. This JSON object should contain only one item that represents the attribute value.

Here are some examples of the `attribute_value` format.

For StartUpOnOff attribute of OnOff Cluster, you should use the following JSON structures as the `attribute_value` to represent the StartUpOnOff `2` and `null`:

```
matter esp controller write attr <node_id> <endpoint_id> 6 0x4003 -- "{\
↪"0:U8\": 2}"
matter esp controller write attr <node_id> <endpoint_id> 6 0x4003 -- "{\
↪"0:NULL\": null}"
```

For Binding attribute of Binding cluster, you should use the following JSON structure as the `attribute_value` to represent the binding list `[{"node":1, "endpoint":1, "cluster":6}]`:

```
matter esp controller write attr <node_id> <endpoint_id> 30 0 -- "{\"0:ARR-
↪OBJ\":[{\"1:U64\":1, \"3:U16\":1, \"4:U32\": 6}]}"
```

For ACL attribute of AccessControl cluster, you should use the following JSON structure as the `attribute_value` to represent the AccessControlList `[{"privilege": 5, "authMode": 2, "subjects": [112233], "targets": null}, {"privilege": 4, "authMode": 3, "subjects": [1], "targets": null}]`:

```
matter esp controller write attr <node_id> <endpoint_id> 31 0 -- "{\"0:ARR-
↪OBJ\":[{\"1:U8\": 5, \"2:U8\": 2, \"3:ARR-U64\": [112233], \"4:NULL\":␣
↪null}, {\"1:U8\": 4, \"2:U8\": 3, \"3:ARR-U64\": [1], \"4:NULL\": null}]}"
```

To write multiple attributes, issue separate commands. For example, write the ACL and Binding attributes in order with:

```
matter esp controller write attr <node_id> <endpoint_id1> 31 0 -- "{\"0:ARR-
↪OBJ\":[{\"1:U8\": 5, \"2:U8\": 2, \"3:ARR-U64\": [112233], \"4:NULL\":␣
↪null}, {\"1:U8\": 4, \"2:U8\": 3, \"3:ARR-U64\": [1], \"4:NULL\": null}]}"
matter esp controller write attr <node_id> <endpoint_id2> 30 0 -- "{\"0:ARR-
↪OBJ\":[{\"1:U64\":1, \"3:U16\":1, \"4:U32\": 6}]}"
```

Wait for the first Write's completion before submitting the second if ordering is required; console submission success alone does not indicate completion.

For attributes of type uint64_t or int64_t, if the absolute value is greater than (2^53), you should use string to represent number in JSON structure for precision

```
matter esp controller write attr <node_id> <endpoint_id> 42 0 -- "{\"0:ARR-
↪OBJ\":[{\"1:U64\": \"9007199254740993\", \"2:U8\": 0}]}"
```

Subscribe commands

Use `subscribe_command::subscription` for remote attribute and event reports. Its delegate receives report callbacks and `OnStateChanged(state, cause)` for connection, retry, and termination. Keep the subscription alive until stopped or terminated, and perform cleanup outside its callbacks under the CHIP lock.

Subscribe attribute commands

Use `sub add attr` to subscribe to attribute reports.

- Attribute subscription syntax:

```
matter esp controller sub add attr <node-id> <endpoint-ids> <cluster-ids>
↪<attribute-ids> <min-interval> <max-interval>
```

Subscribe event commands

Use `sub add event` to subscribe to event reports.

- Send the subscribe-event command:

```
matter esp controller sub add event <node-id> <endpoint-ids> <cluster-ids> <event-
↪ids> <min-interval> <max-interval>
```

List and remove subscriptions

`sub list` shows console-owned subscriptions, including connecting and retrying entries. Each entry shows the fabric, node, state, and subscription ID when available.

```
matter esp controller sub list
matter esp controller sub remove <node-id> <subscription-id>
matter esp controller sub remove <node-id> all
matter esp controller sub remove all
```

Node-specific removal applies to the active fabric. The literal `all` must be provided explicitly: missing arguments, invalid IDs, and extra arguments are rejected without removing anything. Node and subscription IDs accept decimal or `0x`-prefixed hexadecimal notation.

> **Warning:** `sub remove all` removes subscriptions across all nodes and fabrics. Removal also affects subscriptions created outside the console, whereas `sub list` displays only console-owned subscriptions.

Group commands

Group Invoke and Write accept `group <id>` or an encoded node ID `0xFFFFFFFFFFFFXXXX`, where `XXXX` is the nonzero 16-bit group ID. Both use the active fabric at submission time. The `group` form has no endpoint argument; the encoded node form ignores the endpoint argument. Read and Subscribe reject group destinations.

Completion reports dispatch, not member acknowledgments. Group commands do not support timed options. Configure matching groups and keys on the controller and targets before sending requests. Manage remote membership through each target's Groups cluster.

Equivalent group destinations:

```
matter esp controller invoke group 1 6 1 -- {}
matter esp controller invoke 0xFFFFFFFFFFFF0001 0 6 1 -- {}
matter esp controller write attr group 1 6 0x4003 -- "{\"0:U8\":2}"
matter esp controller write attr 0xFFFFFFFFFFFF0001 0 6 0x4003 -- "{\"0:U8\":2}"
```

Configure controller-local groups and keys in builds without a Matter server:

```
matter esp group list
matter esp group add <group-id> <group-name>
matter esp group remove <group-id>
matter esp group keyset list
matter esp group keyset add <ketset-id> <policy> <validity-time> <epoch-key-oct-str>
matter esp group keyset remove <ketset-id>
matter esp group bind <group-id> <ketset-id>
matter esp group unbind <group-id> <ketset-id>
```

ICD client management

Use ICD client management with `CONFIG_ESP_MATTER_CONTROLLER_ENABLE=y` and `CONFIG_ESP_MATTER_ENABLE_MATTER_SERVER=n`. The controller example registers the console commands when `CONFIG_ENABLE_CHIP_SHELL=y`. In a custom application, call `esp_matter::controller::console::register_icd_commands()` to register them.

ICD console commands

```
matter esp controller icd list
matter esp controller icd register <node> <check-in-node> <subject> <key-hex> --
↪client-type permanent|ephemeral [--verification-key HEX]
matter esp controller icd unregister <node> <check-in-node> [--verification-key HEX]
matter esp controller icd status
matter esp controller icd stop
```

`node` identifies the ICD server. `check-in-node` identifies the check-in recipient on the controller's current fabric; use the controller's operational Node ID to receive check-ins on this controller. `subject` identifies the subject whose activity the ICD monitors. Node IDs and subjects accept decimal or `0x`-prefixed hexadecimal notation. Supply each key as 32 hexadecimal digits, without a `0x` prefix. The verification key is optional.

`register` and `unregister` use the current fabric. Unregistration requires a local registration with the same ICD server and check-in node. `list` shows registration metadata across stored fabrics, including the fabric index, node IDs, counters, monitored subject, and client type. It omits key material.

The console observes one ICD request at a time. Submission success means the request was accepted. Wait for the completion log to confirm that both the remote command and local storage update succeeded. `status` reports whether the console still observes the request.

`stop` detaches observation and stops work before command dispatch. An already-sent command can still complete and update local storage. A request for the same peer can return busy while that work finishes, even after `status` reports inactive.

ICD client API

Include `esp_matter_controller_icd_client.h` and use the `esp_matter::controller::icd_client` namespace. Replace calls to the removed `controller::list_registered_icd()` with `icd_client::for_each_registered_icd(callback, context)`. Its callback receives an `entry_t` containing registration metadata. Return `CHIP_NO_ERROR` to continue iteration, or another error to stop and return that error to the caller. Keep storage unchanged during iteration; entry references remain valid for the callback duration.

For registration and unregistration, keep an `icd_client::interaction` and its `icd_client::delegate` alive while observing the operation. Populate a `request_t` with `METHOD_REGISTER` or `METHOD_UNREGISTER` and call `start(request, &delegate)`. Pass the required registration key as 16 bytes; an omitted verification key uses `nullptr` and size zero. `start()` copies borrowed key data before returning. A synchronous error produces no callbacks. An accepted request produces deferred callbacks and one terminal `OnDone()`, unless you detach the delegate with `stop()`.

`OnResponse()` provides a valid remote status, including rejection, and an ICD counter for a successful registration. The response reference remains valid for the callback duration. `OnDone()` reports connection, command, or local storage errors; `CHIP_NO_ERROR` confirms both remote success and the local storage update. Remote success can precede a local storage failure.

Use the CHIP stack context for API calls, storage iteration, and destruction of an active interaction. `OnDone()` may restart, stop, or destroy the interaction; `OnResponse()` may not. `stop()` detaches the delegate and lets an already-sent command finish updating storage. `active()` reports whether the interaction still observes the request. Pending work for the same scoped peer can cause `CHIP_ERROR_BUSY` even after observation stops.

1.3.3 Commissioner features

The commissioner is an enhanced controller that can perform commissioning which is the sequence of operations to bring a Node into a Fabric by assigning an Operational Node ID and Node Operational credentials.

The commissioner should support the additional features:

- Obtain the onboarding payload (QR code or manual code) and use it to starting commissioning.
- Verify the commissionee's Device Attestation Certificate (DAC) chain and Certificate Declaration (CD) during commissioning.
- Receive the Certificate Signing Request (CSR) and issue NOC for it during commissioning.

Pairing commands

Enable `CONFIG_ESP_MATTER_COMMISSIONER_ENABLE` and connect the controller to its operational IP network before pairing. Supply a QR payload or manual setup code. The SDK chooses the discovery transport by default; forcing BLE requires `CONFIG_ENABLE_ESP32_BLE_CONTROLLER`.

Supplying a resolved address skips discovery. Link-local IPv6 addresses require an address scope or `--interface`. Optional Wi-Fi and Thread credentials provision the device network independently of transport selection.

```
matter esp controller pair <node> <setup-code> [--transport auto|ble|on-network] [--
↪wifi SSID PASSWORD] [--thread HEX]
matter esp controller pair <node> --address IP --pin PIN [--port N] [--interface NAME]
matter esp controller unpair <node>
```

UDC pairing requires commissioner discovery. It uses the peer address from a pending UDC entry; supply the node ID and setup PIN. Successful request submission consumes the entry. An inactive entry expires after 30 seconds.

```
matter esp controller pair udc status
matter esp controller pair udc <node> <pin>
```

Attestation Verification

Attestation Trust Storage

The commissioner offers four options for the Attestation Trust Storage which is used to store and utilize the PAA certificates for the Device Attestation verification. This feature is available when the `Enable matter commissioner` option is enabled in menuconfig. You can modify this setting in menuconfig `Components` -> `ESP Matter Controller` -> `Attestation Trust Store`.

- `Attestation Trust Store - Test`

  Use two hardcoded PAA certificates (Chip-Test-PAA-FFF1-Cert&Chip-Test-PAA-NoVID-Cert) in the firmware.

- `Attestation Trust Store - SPIFFS`

  Read the PAA root certificates from the SPIFFS partition. Please refer to the SPIFFS FileSystem to configure the SPIFFS partition and directory. After this, please configure the partition label in menuconfig `Components` -> `ESP Matter Controller` -> `SPIFFS Attestation Trust Store Partition Label`.

- `Attestation Trust Store - DCL`

  Read the PAA root certificates from the DCL(Distributed Compliance Ledger). You can choose to use MainNet or TestNet by calling `chip::Credentials::dcl_attestation_trust_store::get_instance().SetDCLNetType()`.

- `Attestation Trust Store - Custom`

  Read the PAA root certificates with the custom method. You should call `chip::Credentials::set_custom_attestation_trust_store()` before `esp_matter::controller::matter_controller_client::setup_commissioner()` to set the custom attestation trust store.

Revoked DAC Chain Check

The commissioner can check whether the DAC chain on the commissionee is revoked during commissioning. This feature is available when the `Enable matter commissioner` option is enabled in menuconfig. You can configure the DAC Revocation Delegate in menuconfig `Components` -> `ESP Matter Controller` -> `Check for Revoked DAC Chain`.

- `Revoked DAC Chain Check - None` None check for revoked DAC chain.
- `Revoked DAC Chain Check - DCL Revocation Points` Check the Revoked DAC chain with the revocation points on DCL. The controller will fetch revocation points from DCL. You can choose to use MainNet or TestNet by calling `chip::Credentials::dcl_revocation_point_da_revocation_delegate::get_instance().set_dcl_net_type()`.
- `Revoked DAC Chain Check - Custom` Check the Revoked DAC chain with the custom method. You should call `chip::Credentials::set_custom_dac_revocation_delegate()` before `esp_matter::controller::matter_controller_client::setup_commissioner()` to set the custom DAC revocation delegate.

Certification Declaration Verification

The commissoner can commission a device that is not certified and using test CD signed by Test CD signing Cert. This feature is available when the `Enable matter commissioner` option is enabled in menuconfig. You can modify this setting in menuconfig `Components` -> `ESP Matter Controller` -> `Support Test Certification Declaration`. If the option is not enable, only certified device can be commissioned by the commissioner.

NOC Issuer

In the esp_matter commissioner example, the commissioner offers two options to issue the NOC chains for itself and other operational nodes.

- `Operational Credentials Issuer - Test`

  Generate Root CA Certificate (RCAC) and RCAC Private Key and issue a Commissioner NOC when setting up the commissioner, and then issue NOCs for other nodes during commissioning with the generated RCAC Certificate and Key.

- `Operational Credentials Issuer - Custom`

  Obtain the NOC chains for the commissioner and other operational nodes with a custom issuer class. The NOC chains can be issued from the cloud with the custom issuer.

1.3.4 Production Considerations

Controller Production

The Matter Controller should always work with a commissioner which is typically a mobile application that assists with its setup and onboarding.

Access Control Privilege

The controller should possess either Administrator or Operator privileges in order to access other nodes within the same Matter fabric. A common approach is to issue a NOC containing an Administrator/Operator CAT for the controller, and to initialize the Access Control List (ACL) of Matter end devices with the corresponding CATs.

Controller NOC

To access other nodes within a Matter fabric, the controller must be part of the same fabric. Therefore, there must be a mechanism to deliver the NOC chain to the controller. Upon receiving the chain, the controller can add or update it in its Fabric Table.

Device List

The controller should be able to retrieve the list of devices within the same Matter fabric. This list should include the Node IDs and device type information of each node, enabling the controller to determine how to interact with and control the respective Matter end devices.

Commissioner Production

Onboarding Payload

The Matter Commissioner should be able to get the QR Code or Manual Code of Matter end-device so that it can start commissioning that device.

Device Attestation Verification

The Matter Commissioner should be able to process the following DA verification during commissioning:

- Verify that the DAC chain of the commissionee is issued by a trusted Product Attestation Authority (PAA) Certificate in the Connectivity Standards Alliance (CSA) 's Distributed Compliance Ledger(DCL).
- Verify that the CD of the commissionee is issued by CSA.
- Verify that the DAC or PAI Certificate is not revoked in CSA's DCL.

NOC Issuer

The Matter Commissioner should be able to install NOC chain on the commissionee during commissioning. The custom NOC issuer should be implemented so that the Matter Commissioner could generate or otherwise obtain NOC chain after receiving CSRResponse command from the commissionee. In production, the NOC issuer is typically a cloud service: the Commissioner retrieves the CSR from the commissionee and forwards it to the cloud service for signing. The local RCAC is intended for testing only and must not be used in production.

Access Control List Configuration

The Matter Commissioner should configure the ACL on the Commissionee over PASE session to grant Administer/Operator privilege over CASE authentication type for all the controllers in the Matter fabric.

1.3.5 Commissioning operation APIs

Include `esp_matter_controller_pairing_command.h` for `esp_matter::controller::pairing`, or `esp_matter_controller_commissioning_window_opener.h` for `esp_matter::controller::commissioning_window`.

Keep each caller-owned operation and its delegate alive until completion or `stop()`. Call APIs, inspect state, and destroy active operations in CHIP context or while holding the CHIP stack lock. `start()` copies any request data it retains before returning. A synchronous rejection produces no callbacks.

Pairing

Pairing defaults to `method_t::setup_code`. Set `request_t::params.setup_code.payload` to a manual code or `MT:` QR payload. For a resolved address, call `request.set_address({peer_address, setup_pin})` to select `method_t::address` and initialize its union member.

Wi-Fi and Thread credentials are independent `chip::Optional` values used to provision the device network, not to choose a discovery transport. An empty Wi-Fi password selects an open network. A present but empty Thread dataset is invalid.

Choose the discovery and PASE transport with `request_t::params.setup_code.transport`:

- `transport_t::automatic` is the default. The SDK discovers the device using payload capabilities, including on-network fallback.
- `transport_t::ble` uses the payload PIN and discriminator over BLE, regardless of advertised transport flags. It requires `CONFIG_ENABLE_ESP32_BLE_CONTROLLER` and rejects compound QR payloads with `CHIP_ERROR_INVALID_ARGUMENT`.
- `transport_t::on_network` uses IP discovery regardless of payload BLE flags.

Forced transports do not fall back to another transport. Payload validation, authentication, and the configured attestation policy still apply. Operational Matter connections use IP.

```
namespace pairing = esp_matter::controller::pairing;

pairing::interaction operation;
pairing::request_t request;
request.node_id = node_id;
request.params.setup_code.payload = setup_code;
CHIP_ERROR error = operation.start(request, pairing::options_t(), &delegate);
```

`OnProgress()` and `OnCommissioningStage()` forward SDK progress. Callbacks may run before `start()` or `cancel()` returns, so initialize any state they access before either call. Once the operation accepts a request, it reports SDK errors through `OnDone()`, including errors encountered during `start()`. Completion means commissioning has finished, not just PASE establishment.

The result contains the node, error, last observed work phase, and SDK commissioning failure details when available. Result references and pointers are valid only during the callback. Do not start, cancel, stop, or destroy a pairing operation from a delegate callback.

A commissioner allows one pairing reservation at a time, including cancellation cleanup. `options_t::timeout_ms` defaults to 300000 ms and must be nonzero. Expiry initiates cancellation. `cancel()` retains the delegate and reports `CHIP_ERROR_CANCELLED` after cleanup. `stop()` detaches the delegate immediately, but holds the commissioner reservation until cleanup finishes. Starts during cleanup return `CHIP_ERROR_BUSY`. Neither cancellation nor stop rolls back remote changes.

If the SDK reports a discovery or PASE failure only through a status update, the operation cancels and reports `CHIP_ERROR_CONNECTION_ABORTED` without waiting for the overall timeout. A later, more specific PASE error may be unavailable.

`options_t::register_icd` enables ICD registration by default. Each operation generates its own registration key and persists the completed registration.

Commissioning windows

Choose `method_t::basic`, `method_t::passcode`, or `method_t::verifier` and set an explicit `chip::ScopedNodeId` destination. The operation does not look up an application or console fabric. Passcode requests generate a PIN and salt when omitted, but reject supplied values that are invalid. Basic windows require target support for Basic Commissioning.

```
namespace window = esp_matter::controller::commissioning_window;

window::interaction operation;
window::request_t request(window::method_t::passcode);
request.node = chip::ScopedNodeId(node_id, fabric_index);
request.window_timeout_seconds = 180;
request.params.passcode.discriminator = discriminator;
request.params.passcode.iterations = 10000;
CHIP_ERROR error = operation.start(request, &delegate);
```

`OnResponse()` reports remote status and includes the setup payload after a successful passcode-based opening. `OnDone()` reports transport or remote rejection errors. Callbacks run after submission returns. `OnDone()` may restart, stop, or destroy the operation; `OnResponse()` may not.

Each object allows one pending request. Separate objects may operate concurrently. `active()` reports local request state, not whether the remote window is open. `stop()` does not close an already-open window, and a timeout does not rule out remote execution.

Unpairing

Call `matter_controller_client::unpair(node, callback)` to unpair a node. On success, it removes the peer's ICD registration before calling the application callback.

1.3.6 Client interactions

`esp_matter` provides the generic interaction APIs. Include `esp_matter_client.h` and use the `invoke_command`, `read_command`, `write_command`, and `subscribe_command` namespaces under `esp_matter::client`. Keep each caller-owned operation and its delegate alive until completion or `stop()`.

Targets and authorization

Use `chip::ScopedNodeId(node_id, fabric_index)` for unicast destinations. The node must be a remote operational node, and the fabric must exist in the initialized Interaction Model runtime. Self-targets are rejected. For group Invoke and Write, use `start(group_id, fabric_index, request, &delegate)`.

Apply application authorization policy before submitting a request. The remote peer enforces Matter ACL checks.

Ownership and callbacks

`start()` copies path spans and encodes values and command fields before returning. A synchronous error rejects the request and schedules no delegate callbacks.

Invoke and Write require an explicit delegate pointer. Pass `nullptr` to log asynchronous failures and discard successful responses, or pass a delegate address to receive callbacks. Read and Subscribe require a delegate. TLV readers are borrowed only for the callback duration.

Starting another Invoke, Read, or Write releases completed state. An active operation rejects restart with `CHIP_ERROR_INCORRECT_STATE` for Invoke or `CHIP_ERROR_BUSY` for Read and Write. `stop()` synchronously detaches the delegate and prevents later callbacks. Repeated calls are safe, but stopping cannot undo work already sent to the network.

Invoke, Read, and Write allow stop, destruction, and reuse in terminal `OnDone()` callbacks. Do not perform these actions in report, response, or subscription callbacks. Subscription states are `connecting`, `active`, `backoff`, and `terminated`. Call `retry_now()` only during automatic resubscription backoff.

Invoke and JSON fields

An Invoke request with `fields == nullptr` encodes an empty command structure. `client::json_encodable` encodes exactly one tagged JSON value, whether a scalar, an `OBJ` structure, or an `ARR-*` array. The encoder replaces the outer JSON tag with the tag supplied by the interaction. Invoke fields must be an `OBJ` value. Use `{"0:OBJ":{}}` for empty fields. The temporary TLV buffer is limited to `chip::kMaxAppMessageLen`. For example:

```
client::json_encodable fields(R"({"0:OBJ":{"0:U8":2}})");
client::invoke_command::request_t request = {
    .endpoint_id = 1,
    .cluster_id = 6,
    .command_id = 1,
    .fields = &fields,
};
CHIP_ERROR err = invoke.start(target, request, &delegate);
```

The JSON text need only remain valid through `start()`. Group Invoke does not support timed or response-timeout options. Its `OnDone(CHIP_NO_ERROR)` callback means the Interaction Model accepted the groupcast. It does not confirm that group members received it. Group Invoke does not call `OnResponse()`.

Read, Write, and Subscribe

Write encoders emit the complete attribute value, including any structure or array container. `client::json_encodable` accepts, for example, `{"0:U16":42}`, `{"0:OBJ":{"1:U16":42}}`, or `{"0:ARR-U8":[1,2]}`.

Read and Subscribe requests take spans of Matter attribute and event paths and copy them during `start()`. Read supports wildcard paths and is fabric-filtered by default. Subscribe uses CHIP automatic resubscription according to its request options.

Use `write.start(group_id, fabric_index, request, &delegate)` for Group Write. Group Write omits endpoints and does not support data versions, timed writes, or chunking. `OnDone()` reports dispatch after `start()` returns. There are no path responses or member acknowledgments.

Write always uses a span, including the one-attribute case:

```
client::json_encodable value("{\"0:U8\":2}");
client::write_command::attribute_write_t attribute = {
    .path = chip::app::ConcreteDataAttributePath(1, 6, 0x4003),
    .value = &value,
};
client::write_command::request_t request = {
    .attributes = chip::Span<const client::write_command::attribute_write_t>(&
↪attribute, 1),
};
CHIP_ERROR err = write.start(target, request, &delegate);
```

Write prepares all paths and values before accepting the request, then sends it through one `WriteClient`. Complete lists use ReplaceAll and may use Matter chunking for non-timed requests. A timed request that requires chunking fails synchronously.

A multi-path Write is not atomic. Inspect every `OnResponse(path, status)`. Later paths may proceed after a recoverable path failure, and earlier successful writes are not rolled back. `OnDone()` reports interaction completion, not success for every path. Supply an explicit JSON value for each write, including `{"0:NULL":null}` for null.

Binding fan-out

Server applications use `client::binding::dispatcher` after Binding Manager initialization. `dispatch(local_endpoint, request, delegate)` accepts native Invoke, Read, Write and Subscribe requests selecting one concrete cluster. Requests are copied before return; bindings supply the fabric and remote endpoint. Read and Subscribe skip group bindings. Invoke and Write accept a nullable delegate pointer; Read and Subscribe require a delegate reference.

Native callbacks identify each destination through `source`. `OnDone()` is per destination, not aggregate completion. Dispatches may overlap on one dispatcher. `stop()` cancels all its pending dispatches and native operations across local endpoints, detaches delegates and suppresses later callbacks. Keep delegates alive until then. The next `dispatch()` or `stop()` releases completed handles.

Removed bindings stop subscriptions; dispatched finite operations may finish. Fabric removal cancels affected work. New bindings apply on the next dispatch. SDK notifications may be replaced, CASE failures may produce no callback, and native start errors are logged. Returned errors do not undo dispatched work.

Call methods in CHIP context or under the stack lock. `OnDone()` may dispatch, stop or destroy the dispatcher. Other callbacks may not stop or destroy it; `dispatch()` returns `CHIP_ERROR_INCORRECT_STATE` during nonterminal native client callbacks. Subscription state callbacks may occur during `dispatch()`.

Controller console interactions

Controller commands retain the `matter esp controller` prefix. The supported subscription commands are `sub add attr`, `sub add event`, `sub list`, and `sub remove`. Removal takes `<node-id> <subscription-id>`, `<node-id> all`, or `all`. Missing arguments never imply all subscriptions. `sub list` shows console-owned subscriptions; removal can affect other owners. Node-specific removal uses the active fabric, while `sub remove all` covers every fabric.

1.4 Matter Certification

The Matter Certification denotes compliance to a Connectivity Standards Alliance (CSA) specification for the product and allows the use of Certified Product logos and listing of the product on the Alliance website for verification.

You need to become a member of CSA and request a Vendor ID code from CSA Certification before you apply for a Matter Certification. Then you need to choose an authorized test provider (must be validated for Matter testing) and submit your product for testing. Below are the steps for the Matter Certification.

1.4.1 Firmware Development

Choose an esp-matter branch

Use the branch corresponding to the Matter version you plan to certify for firmware development.

Table 1: esp-matter Branches Mapping

Matter Version	esp-matter Branch
Matter 1.1	release/v1.1
Matter 1.2	release/v1.2
Matter 1.3	release/v1.3
Matter 1.4	release/v1.4
Matter 1.4.2	release/v1.4.2
Matter 1.5	release/v1.5

Product Attestation Authority (PAA) Generation

For Matter certification testing, vendors should generate the test Product Attestation Authority (PAA) certificate, Product Attestation Intermediate (PAI), and Device Attestation Certificate (DAC), rather than using the default test PAA certificate from the connectedhomeip SDK repository. Therefore, you need to generate a PAA certificate and use it to sign the PAI certificate, which is then used to sign the DAC. The PAI certificate, DAC certificate, and DAC private key should be stored in the product under test. The chip-cert tool which is built in esp-matter setup process can be used to generate the PAA certificate and Certification Declaration (CD).

```
cd $ESP_MATTER_PATH/connectedhomeip/connectedhomeip/out/host/
./chip-cert gen-att-cert --type a --subject-cn "Example PAA CN" --subject-vid 0x131B \
                         --valid-from "2021-06-28 14:23:43" --lifetime 4294967295 \
                         --out-key /path/to/PAA_key \
                         --out /path/to/PAA_certificate
```

Certification Declaration (CD) Generation

Certification Declaration (CD) is a cryptographic document that proves a Matter device complies with protocol standards. For Matter certification testing, users should issue a test CD based on the test CD signing keys in the connectedhomeip SDK repository. This test CD should match the Vendor ID and Product ID in the test DAC, as well as the information in the BasicInformation cluster. During certification testing, the CD's –certification-type should be set to 1 (provisional), while the official CD issued by CSA after passing certification will have –certification-type set to 2 (official).

```
cd $ESP_MATTER_PATH/connectedhomeip/connectedhomeip/out/host/
./chip-cert gen-cd --format-version 1 --vendor-id 0x131B --product-id 0x1234 \
                            --device-type-id 0x010c --certificate-id CSA00000SWC00000-
↪01 \
                            --security-level 0 --security-info 0 --version-number 1 \
                            --certification-type 1 \
                            --key $ESP_MATTER_PATH/connectedhomeip/connectedhomeip/
↪credentials/test/certification-declaration/Chip-Test-CD-Signing-Key.pem \
                            --cert $ESP_MATTER_PATH/connectedhomeip/connectedhomeip/
↪credentials/test/certification-declaration/Chip-Test-CD-Signing-Cert.pem \
                            --out path/to/test_CD_file
```

Factory Bin File Generation

Based on the test PAA and CD generated above, users can use esp-matter-mfg-tool to generate the corresponding factory partition bin file. This file contains commissioning information (discriminator, salt, iteration count, and SPAKE2+ verifier), device attestation information (Certification Declaration (CD), Product Attestation Intermediate (PAI) certificate, Device Attestation Certificate (DAC), and DAC private key), device instance information (vendor ID, vendor name, product ID, product name, etc.), and device information (fixed label, supported locales, etc.). This information is used to identify the product and ensure secure commissioning.

In the reference command below, the -n option specifies the number of factory bin files to generate. esp-matter-mfg-tool will first generate the PAI certificate and key, then use them to generate the specified number of different DAC certificates and corresponding keys. The tool will use the generated certificates and keys to create multiple factory partition binary files with different DACs, discriminators, and setup pincodes.

```
python3 -m pip install esp-matter-mfg-tool
esp-matter-mfg-tool -n <count> -cn Espressif --paa -c /path/to/PAA_certificate -k /
↪path/to/PAA_key \
                    -cd /path/to/CD_file -v 0x131B --vendor-name Espressif -p 0x1234 \
                    --product-name Test-light --hw-ver 1 --hw-ver-str v1.0
```

Firmware Configuration

When generating test product firmware, the following configurations can be set in menuconfig to enable accessing the information from the factory bin file generated above:

```
CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER=y
CONFIG_ENABLE_ESP32_DEVICE_INSTANCE_INFO_PROVIDER=y
CONFIG_ENABLE_ESP32_DEVICE_INFO_PROVIDER=y
CONFIG_FACTORY_PARTITION_DAC_PROVIDER=y
CONFIG_FACTORY_COMMISSIONABLE_DATA_PROVIDER=y
CONFIG_FACTORY_DEVICE_INSTANCE_INFO_PROVIDER=y
```

OTA File Generation

If the test device supports standard Matter OTA, you need to prepare OTA files for certification testing. (Note: You can add OTA rollback functionality to facilitate certification testing)

In esp-matter projects, change the firmware version number by modifying the PROJECT_VER and PROJECT_VER_NUMBER values in your application's CMakeLists.txt file. When generating OTA files, you need to use firmware with a higher version number than the one currently running on the device.

Use ota_image_tool.py to generate Matter OTA files, where the `-vn` parameter is the software version number, and the `-vs` parameter is the software version string:

```
cd $ESP_MATTER_PATH/connectedhomeip/connectedhomeip/src/app
./ota_image_tool.py create -v <vendor-id> -p <product-id> -vn 2 -vs 1.1 -da sha256 \
                        /path/to/original_app_bin /path/to/out_ota_bin
```

Other Notes

For products that support the Identify cluster, you need to set the identify type according to the specific product form:

Table 2: Presentation Enumeration Values

Value	Name	Summary	Conformance
0x00	None	No presentation.	M
0x01	LightOutput	Light output of a lighting product.	M
0x02	VisibleIndicator	Typically a small LED.	M
0x03	AudibleBeep	Audible beep sound.	M
0x04	Display	Presentation will be visible on display screen.	M
0x05	Actuator	Presentation via actuator (e.g., window blind operation or relay).	M

Implement the corresponding identify command and trigger effect command handlers:

```
esp_err_t app_identification_cb(identification::callback_type_t type, uint16_t␣
↪endpoint_id, uint8_t effect_id, uint8_t effect_variant, void *priv_data)
{
    esp_err_t err = ESP_OK;
    ESP_LOGI(TAG, "Identification type: %d, effect_id: %d", type, effect_id);
    switch (type) {
        case esp_matter::identification::START:
            break;
        case esp_matter::identification::STOP:
            break;
        case esp_matter::identification::EFFECT:
            switch (effect_id) {
                case (int)Identify::EffectIdentifierEnum::kBlink:
                    break;
                case (int)Identify::EffectIdentifierEnum::kBreathe:
                    break;
                case (int)Identify::EffectIdentifierEnum::kOkay:
                    break;
                case (int)Identify::EffectIdentifierEnum::kChannelChange:
                    break;
                case (int)Identify::EffectIdentifierEnum::kFinishEffect:
                    break;
```

(continues on next page)

(continued from previous page)

```
                case (int)Identify::EffectIdentifierEnum::kStopEffect:
                    break;
                default:
                    break;
            }
            break;
        default:
            err = ESP_FAIL;
            break;
    }
    return err;
}
```

1.4.2 Preparation of Files Required for Certification Testing

After preparing the device under test, users need to prepare the following files to submit to the testing organization along with the device.

PICS File

The PICS file is used to define all Matter features of the product, including endpoints, device types, clusters, attributes, commands, etc. The testing organization will determine the test cases to execute during certification testing based on the PICS file submitted by the user.

The PICS Tool website is a tool for opening, modifying, validating, and saving XML format PICS files. Pics_xml_template_files contains all PICS file templates, with each XML file defining the feature characteristics of one or more clusters.

The PICS-generator tool can generate PICS files based on Pics_xml_template_files. This tool reads the clusters, attributes, commands, features, and events supported by the device under test and outputs a PICS file. Note: This tool cannot automatically generate the Base.xml file, which needs to be opened with PICS TOOL and manually configured. Reference commands are as follows:

```
# WiFi Device:
python3 PICSGenerator.py --pics-template path/to/XML_template --pics-output out_
↪folder --commissioning-method ble-wifi \
                        --wifi-ssid ssid-name --wifi-passphrase password --passcode␣
↪setup-pin-code --discriminator discriminator \
                        --paa-trust-store-path path/to/paa --dm-xml path/to/data_model
# Thread Device:
python3 PICSGenerator.py --pics-template path/to/XML_template --pics-output out_
↪folder --commissioning-method ble-thread \
                        --thread-dataset-hex thread_dataset --passcode setup-pin-code␣
↪--discriminator discriminator \
                        --paa-trust-store-path path/to/paa --dm-xml path/to/data_model
```

Test PAA Certificate

Use the PAA file generated above.

OTA File

Use the OTA file generated above.

QR Code for Device Under Test

Use the QR code for the device under test.

Operating Instructions for Device Under Test

Provide operating instructions for powering the device, performing factory reset, OTA updates, rollback, and other operations.

1.4.3 Testing Environment Setup and Testing Methods

After preparing the device under test, users can perform self-testing using the Test Harness tool.

Different Matter versions use different Test Harness tool versions for certification.

Table 3: TH Version Mapping

Matter Version	TH Version	SDK Commit
Matter 1.2	TH-Fall2023	19771ed
Matter 1.3	v2.10+spring2024	11f94c3
Matter 1.4	v2.11+fall2024	f2e5de7
Matter 1.4.1	v2.12+spring2025	91eab26
Matter 1.4.2	v2.13+summer2025	1b2b3fd
Matter 1.5	v2.14+fall2025	ca9d111

Test Harness Setup

After selecting the Test Harness version according to the table above, refer to the Matter_TH_User_Guide.pdf document for each Matter version setup, for different test harness versions, the Matter_TH_User_Guide.pdf file may be different.

1.4.4 Submitting Certification Application Online

After the device under test passes all tests from the testing organization, users can fill out the application on the CSA certification application website . The following explanation uses Add New Product Application as an example.

Information to Fill In

Table 4: Application Parameters Description

Parameter Name	Description
Your Company	Manufacturer name, usually just select from the dropdown list
Product Name	Product name, defined by the manufacturer
Product Short Description	Custom short product description
Product Long Description	Custom long product description
Product Link URL	Optional, product description link
Product Image	Optional, product image
Date Product can be Displayed	Date when the product will be publicly displayed on the CSA website
Product Type	Product certification type, select end device or platform certification. Device manufacturers generally select End Device
Completed optional TIS/TRP testing	Optional, default is No
Application Type	Application protocol type to be certified, select Matter
Application Type Version	Matter protocol version, select Matter 1.1, Matter 1.2, etc. as needed
Compliant Platform	Optional, if applying for certification based on Espressif's certified platform, select ESP32-C-Series or ESP32-H-Series from the dropdown list
Firmware Version	Firmware version number, consistent with software version string in BasicInformation Cluster
Hardware Version	Hardware version number, consistent with hardware version string in BasicInformation Cluster
SKU	Optional, manufacturer-defined, usually the product model
Product ID (PID)	Product ID, consistent with Product ID in BasicInformation Cluster
Vendor ID (VID)	Vendor ID, the ID assigned to the manufacturer by CSA
Transport Used	Check the transport protocols supported by the product. For example, for a WiFi product that supports BLE commissioning, check Wi-Fi and Bluetooth
Use 3rd DAC	Optional, if using a third-party DAC, select Yes and fill in dac_origin_vendor_id and dac_origin_product_id
dac_origin_vendor_id	If using a third-party DAC, fill in the DAC's Vendor ID
dac_origin_product_id	If using a third-party DAC, fill in the DAC's Product ID
authorized_paa_list	Optional, default is empty
SoftwareVersion Attribute (integer)	Software version number, consistent with software version in BasicInformation Cluster
Technical Category	Technical category, select Matter
Primary Technical Sub Category	Primary technical subcategory (device type). If the certified device includes multiple device types, select the primary device type
Technical Sub Categories	Technical subcategories (device types), select all supported device types
Parent Functional Categories	Parent functional category. For end products, select a category that matches the actual functionality
Functional Sub Categories	Functional subcategories. For end products, select one or more categories that match the actual product functionality
Submit for processing	Select submit or save the filled record
Declaration of Conformance Document	Doc document

continues on next page

Table 4 – continued from previous page

Parameter Name	Description
PICS Document	PICS file in zip format
Supplemental PICS Document	Optional, supplemental PICS file, default is not filled
External Certificate	Can be left blank for Matter certification
Transport Attestations	Transport attestation documents consistent with the transport protocols checked in Transport Used. Can be declared based on Espressif certificates or certificates derived from Espressif certificates
Security Attestation	Security attestation document

Declaration of Conformance (Doc) Instructions

Based on the Doc template provided by Espressif, fill in the relevant content. After the manufacturer signs this document, send it to the certification testing organization. After the testing organization signs it, the manufacturer uploads the signed Doc.

Transport Attestations (Bluetooth, Wi-Fi, Thread, Ethernet) Instructions:

Fill in based on the Transport Attestation template provided by Espressif (version description). Different chip models may have different certificate numbers, which should be verified before filling in. After completion, sign and upload yourself.

Security Attestation Instructions

Fill in based on the Security Attestation template provided by Espressif. After completion, sign and upload yourself.

Get the AID and test report

After completing and submitting the application, you will receive an Application ID (AID). Provide this ID to the certification testing organization, and the testing organization will upload the product test report to the CSA certification body.

Other steps

After completing the application and payment, the CSA certification team will review the application and test report submitted by the manufacturer. If there are issues, the manufacturer will be notified by email. If there are no issues, the certification is passed, and the official CD file will be issued.

1.4.5 Filling in Certified Product Information on DCL Website

After the product passes certification, manufacturers also need to add Model, Model Version, and Compliance on the Distributed Compliance Ledger (DCL). Refer to the documentation: HowTo - Writing to the DCL.pdf .

1.4.6 Other notes for some certification test cases

Route Information Option (RIO) notes

For Wi-Fi products using LwIP, TC-SC-4.9 should be tested in order to verify that the product can receive Router Advertisement (RA) message with RIO and add route table that indicates whether the prefix can be reached by way of the router. It can be tested with a Thread Border Router (BR) which sends RA message periodically and a Thread End Device that is used to verify the Wi-Fi product can reach the Thread network via Thread BR. Some Wi-Fi Routers might have the issue that they cannot forward RA message sent by the Thread BR, so please use a Wi-Fi Router that can forward RA message when you are testing TC-SC-4.9.

Ensure that the following Kconfig options are enabled in your device:

```
# Enable SLAAC so that the unique local address can be added to the device's network␣
↪interface.
CONFIG_LWIP_IPV6_AUTOCONFIG=y
# Ensure that IPv6 address number is enough for Matter usage, can be increased if␣
↪your device requires more IPv6 addresses.
CONFIG_LWIP_IPV6_NUM_ADDRESSES=6
# Enable IP6 Route Hook so that the device can know which netif should be chosen for␣
↪the messages to Thread devices.
CONFIG_LWIP_HOOK_IP6_ROUTE_DEFAULT=y
# Enable ND6 Get Gateway Hook so that the device can know the gateway address for␣
↪Thread devices.
CONFIG_LWIP_HOOK_ND6_GET_GW_DEFAULT=y
# Install the hook to your netif so that the RA message with RIO can be handled
CONFIG_ENABLE_ROUTE_HOOK=y
```

For ESP-IDF v5.4.3+, v5.5.2+, v6.x, the RIO support has been added to LwIP component in ESP-IDF. With the RIO support enabled, the route hook in Matter and the two LwIP hooks can be disabled.

```
# Enable Route Info Option support
CONFIG_LWIP_IPV6_ND6_ROUTE_INFO_OPTION_SUPPORT=y
# The three options can be disabled if LwIP supports RIO
CONFIG_LWIP_HOOK_IP6_ROUTE_DEFAULT=n
CONFIG_LWIP_HOOK_ND6_GET_GW_DEFAULT=n
CONFIG_ENABLE_ROUTE_HOOK=n
```

Here are the steps to set up the Thread BR and Thread End Device. You should prepare 2 Radio Co-Processors (RCP) to set up the ot-br-posix and ot-cli-posix. The RCP on ESP32-H2 is suggested to be used here. And you can also use other platforms (such as nrf52840, efr32, etc.) as the RCPs.

Setup Thread BR

The otbr-posix can be run on RaspberryPi or Ubuntu machine. Connecting an RCP to the host, the port `RCP_PORT1` for it will be `/dev/ttyUSBX` or `/dev/ttyACMX`.

- Build the otbr-posix on the host

```
git clone https://github.com/openthread/ot-br-posix
cd ot-br-posix
./script/bootstrap
./script/setup
```

Then the otbr-posix will be built and a service named otbr-agent will be created on the host. You can disable the service and start the otbr-posix manually.

```
sudo systemctl disable otbr-agent.service
sudo ./build/otbr/src/agent/otbr-agent -I wpan0 -B eth0 -v spinel+hdlc+uart:///{RCP_
↪PORT1}
```

In the above commands:

- `wpan0` is the infra network interface. The network interface named wpan0 will be created on the host as the thread network interface.
- `eth0` is the backbone network interface, which is always the ethernet or wifi network interface on the host, please ensure that the backbone network interface is connected to the AP which the Wi-Fi product is also connected to.
- `RCP_PORT1` is the port of RCP for Thread BR.

The otbr-posix is running on the host now. Open another terminal, start console for otbr-posix, form Thread network, and get dataset.

```
sudo ot-ctl
> ifconfig up
> thread start
> dataset active -x
```

Please record the dataset you get with the last command, it will be used by ot-cli-posix to join the BR's network in the next step.

Setup Thread End Device

We use the Posix Thread Command-Line Interface (CLI) as the Thread End Device. Connect another RCP to the host and get the port *RCP_PORT2* for it.

- Build the otcli on the host

```
git clone --recursive https://github.com/openthread/openthread.git
cd openthread/
./script/bootstrap
./bootstrap
./script/cmake-build posix
./build/posix/src/posix/ot-cli 'spinel+hdlc+uart:///dev/{RCP_PORT2}?uart-
↪baudrate=115200' -v
```

The console for the ot-cli will be started. Connect the ot-cli to the otbr's Thread network with the dataset you got in the above step.

```
> dataset set active <PROVIDE THE DATASET OF THE BR THAT YOU NEED TO JOIN>
> dataset commit active
> ifconfig up
> thread start
> srp client autostart enable
```

In the console of ot-cli, discover the product IP address.

```
> dns service 177AC531F48BE736-0000000000000190 _matter._tcp.default.service.arpa.
DNS service resolution response for 177AC531F48BE736-0000000000000190 for service _
↪matter._tcp.default.service.arpa.
Port:5540, Priority:0, Weight:0, TTL:6913
Host:72FF282E7739731F.default.service.arpa.
```

(continues on next page)

(continued from previous page)

```
HostAddress:fd11:66:0:0:22ae:27fe:13ac:54df TTL:6915
TXT:[SII=35303030, SAI=333030, T=30] TTL:6913
```

Note: `177AC531F48BE736-0000000000000190` can be get with command `avahi-browse -rt _matter._tcp`. `177AC531F48BE736` is the compressed Fabric ID and `0000000000000190` is the node ID.

Ping the IP address of the Wi-Fi device.

```
> ping fd11:66:0:0:22ae:27fe:13ac:54df
16 bytes from fd11:66:0:0:22ae:27fe:13ac:54df : icmp_seq=2 hlim=64 time=14ms
1 packets transmitted, 1 packets received. Packet loss = 0.0%. Round-trip min/avg/max
↪= 14/14.0/14 ms.
Done
```

The ping command should be successful.

TC-CNET-3.11

No response on step 7 is expected (Related issue).

All the NetworkCommissioning commands are fail-safe required. If the commands fail with a `FAILSAFE_REQUIRED` status code. You need to send `arm-fail-safe` command and then send the NetworkCommissioning commands.

TC-RR-1.1

For more application endpoints with group cluster, need more nvs size to store group table, so if the `TC-RR-1.1` failed, can try to increase the nvs size. (Related issue)

Please note that the minimum NVS size required is 48 KB (0xC000) when using a single endpoint with a group cluster.

1.4.7 FW/SDK configuration notes

- `Enable OTA Requestor` in → `Component config` → `CHIP Core` → `System Options`

  The option to enable OTA requestor. This option should be enabled if the OTA requestor feature is selected in PICS files.

- `Enable Extended discovery Support` in → `Component config` → `CHIP Device Layer` → `General Options`

  This option should be enabled if the PICS option `MCORE.DD.EXTENDED_DISCOVERY` is selected.

- `Enable Device type in commissionable node discovery` in → `Component config` → `CHIP Device Layer` → `General Options`

  This option should be enabled if the PICS option `MCORE.SC.EXTENDED_DISCOVERY` is selected.

- `LOG_DEFAULT_LEVEL` in → `Component config` → `Log output`

  It is suggested to set log level to `No output` for passing the test cases of OnOff, LevelControl, and ColorControl clusters. Here is related issue.

1.5 Production Considerations

1.5.1 Prerequisites

All Matter examples use certain test or evaluation values that enables you to quickly build and test Matter. As you get ready to go to production, these must be replaced with the actual values. These values are typically a part of the manufacturing partition in your device.

Vendor ID and Product ID

A **Vendor Identifier (VID)** is a 16-bit number that uniquely identifies a particular product manufacturer or a vendor. It is allocated by the Connectivity Standards Alliance (CSA). Please reach out to CSA for this.

A **Product Identifier (PID)** is a 16-bit number that uniquely identifies a product of a vendor. It is assigned by the vendor (you).

A VID-PID combination uniquely identifies a Matter product.

Certificates

A **Device Attestation Certificate (DAC)** proves the authenticity of the device manufacturer and the certification status of the device's hardware and software. Every Matter device must have a DAC and corresponding private key, unique to it. The device should also have a Product Attestation Intermediate (PAI) certificate that was used to sign and attest the DAC. The PAI certificate in turn is signed and attested by Product Attestation Authority (PAA). The PAA certificate is an implicitly trusted self-signed root certificate.

Please reach out to your Espressif representative for the details about how to procure the DAC.

Certification Declaration (CD)

A **Certification Declaration (CD)** is a cryptographic document that allows a Matter device to assert its protocol compliance. Once your product is certified, the CSA creates a CD for that device. The CD should then be included in the device firmware by the device manufacturer.

Setup Passcode, Discriminator and Onboarding Payload

The unique **setup passcode** serves as the proof of possession and is also used to compute the shared secret during commissioning. The corresponding SPAKE2+ verifier of the passcode is installed on the device and not the actual passcode.

The **discriminator** is used to easily distinguish between devices to provide a seamless experience during commissioning.

The onboarding payload is the **QR code** and the **manual pairing code** that assists a commissioner (like a phone app) to allow onboarding a device into the Matter network. The QR code and/or the manual pairing code are generally printed on the packaging of the device.

Manufacturing Partition

Espressif's SDK for Matter uses a separate manufacturing partition to store all the information mentioned above. Because the DACs are unique to every device, the manufacturing partition will also be unique per device. Thus by moving all the typical per device unique fields into the manufacturing partition, the rest of the components like the bootloader, firmware image are common across all your devices. You can refer the Manufacturing section below for creating a large number of manufacturing partition images.

Your manufacturing line needs to ensure that these unique manufacturing partition images are correctly written to each device and the appropriate QR code images associated with each device. You may also opt for Espressif's pre-provisioning service that pre-provisions these unique images before shipping the modules and provides a manifest (CSV file) along with QR code images bundle.

1.5.2 Over-the-Air (OTA) Updates

Matter devices must support OTA firmware updates, either by using Matter-based OTA or vendor specific means.

In case of Matter OTA, there's an *OTA provider* that assists an *OTA requestor* to get upgraded. The SDK examples support Matter OTA requestor role out of the box. The OTA provider could be a manufacturer specific phone app or any Matter node that has internet connectivity.

Alternatively, ESP RainMaker OTA service can also be used to upgrade the firmware on the devices remotely. As opposed to the Matter OTA, ESP RainMaker OTA allows you the flexibility of delivering the OTA upgrades incrementally or to groups of devices.

1.5.3 Manufacturing

Mass Manufacturing Utility

For commissioning a device into the Matter Fabric, the device requires the following information:

- **Device Attestation Certificate (DAC) and Certification Declaration (CD)**: verified by commissioner to determine whether a device is a Matter certified product or not.
- **Discriminator**: advertised during commissioning to easily distinguish between advertising devices.
- **Spake2+ parameters**: work as a proof of possession.

These details are generally programmed in the manufacturing partition that is unique per device. ESP-Matter provides a utility (esp-matter-mfg-tool) to create these partition images on a per-device basis for mass manufacturing purposes.

When using the utility, by default, the above details will be included in the generated manufacturing partition image. The utility also has a provision to include additional details in the same image by using CSV files.

Details about using the mass manufacturing utility can be found here: esp-matter-mfg-tool

Pre-Provisioned Modules

ESP32-C6 modules can be pre-flashed with the manufacturing partition images during module manufacturing itself and then be shipped to you.

This saves you the overhead of securely generating, encrypting and then programming the partition into the device at your end.

Please contact your Espressif contact person for more information.

The esp-matter-mfg-tool Example

In Espressif Matter Prep-provisioning modules, the DAC key pair, DAC and PAI certificates are pre-flashed by default.

This section gives some examples on how to generate factory partition binary which includes :

Device unique data (Discriminator, Verifier, Serial Number, etc)
Manufacturing information (Vendor name, Product name, Hardware version, etc)

Note: The items listed in the examples are all mandatory, some common manufacturing information could be removed if they are hard coded in the firmware.

This is the example to generate factory images after pre-provisioning:

- **Generate generic factory image**

```
esp-matter-mfg-tool -cd ~/test_cert/CD/Chip-CD-131B-1000.der -v 0x131B --
↪vendor-name ESP -p 0x1000 --product-name light --hw-ver 1 --hw-ver-str␣
↪v1.0 --mfg-date 20221025 --passcode 19861989 --discriminator 601 --
↪serial-num esp32c_dev3
```

- **Generate multiple generic factory images**

```
esp-matter-mfg-tool -n 10 -cd ~/test_cert/CD/Chip-CD-131B-1000.der -v␣
↪0x131B --vendor-name ESP -p 0x1000 --product-name light --hw-ver 1 --hw-
↪ver-str v1.0 --mfg-date 20221025
```

- **Generate factory image with rotating device unique identify**

```
esp-matter-mfg-tool -cd ~/test_cert/CD/Chip-CD-131B-1000.der -v 0x131B --
↪vendor-name ESP -p 0x1000 --product-name light --hw-ver 1 --hw-ver-str␣
↪v1.0 --mfg-date 20221025 --passcode 19861989 --discriminator 601 --
↪serial-num esp32c_dev3 --enable-rotating-device-id --rd-id-uid␣
↪c0398f4980b07c9460f71c5421e1a3c5
```

- **Generate multiple factory images with csv and mcsv**

```
esp-matter-mfg-tool -cd ~/test_cert/CD/Chip-CD-131B-1000.der -v 0x131B --
↪vendor-name ESP -p 0x1000 --product-name light --hw-ver 1 --hw-ver-str␣
↪v1.0 --enable-rotating-device-id --mfg-date 20221025 --csv mfg.csv --
↪mcsv mfg_m.csv
```

- **The example of csv and mcsv file**
- **CSV:**

serial-num,data,string
rd-id-uid,data,hex2bin
discriminator,data,u32

- **MCSV:**

serial-num,rd-id-uid,discriminator
esp32c_dev3,c0398f4980b07c9460f71c5421e1a3c5,1234
esp32c_dev4,c0398f4980b07c9460f71c5421e1a3c6,1235
esp32c_dev5,c0398f4980b07c9460f71c5421e1a3c7,1236
esp32c_dev6,c0398f4980b07c9460f71c5421e1a3c8,1237
esp32c_dev7,c0398f4980b07c9460f71c5421e1a3c9,1238

Recommended Providers to Use

Note: WARNING: These options are not recommended for devices that are already in field or modules that reads data from the factory partition or some other source.

We recommend using the following providers:

- Commissionable data provider: secure cert
- Device attestation data provider: secure cert
- Device instance info provider: secure cert

Below are the configuration options that should be enabled. These can be appended to `sdkconfig.defaults`.

In the following example, we demonstrate a different approach that places the configurations in a separate file, which is then used with the `idf.py build` command.

```
cat > sdkconfig.defaults.prod <<EOF
# Enable the implementations in the connectedhomeip repo
CONFIG_ENABLE_ESP32_FACTORY_DATA_PROVIDER=y
CONFIG_ENABLE_ESP32_DEVICE_INSTANCE_INFO_PROVIDER=y

# Set the appropriate providers
CONFIG_SEC_CERT_DAC_PROVIDER=y
CONFIG_SEC_CERT_COMMISSIONABLE_DATA_PROVIDER=y
CONFIG_SEC_CERT_DEVICE_INSTANCE_INFO_PROVIDER=y
CONFIG_NONE_DEVICE_INFO_PROVIDER=y
EOF

idf.py -D SDKCONFIG_DEFAULTS="sdkconfig.defaults.prod" set-target esp32c3 build
```

1.6 Espressif Matter Services

Beyond the SDK, Espressif offers production services that help you take a Matter product from prototype to mass production. This page summarizes what is available and how to engage.

1.6.1 Matter Pre-Provisioning Service

Every Matter device needs a unique, CSA-anchored identity:

- Device Attestation Certificate (DAC) and its private key
- Product Attestation Intermediate (PAI) certificate
- Unique commissioning data (discriminator, setup passcode, QR code)

Provisioning these securely at scale involves two challenges:

- You need a CSA-authorized CA vendor, known as Product Attestation Authority (PAA), that can issue certificates for every device you manufacture.
- Each certificate must be programmed uniquely onto each device. Programming something unique per device at scale on the factory line can be a hassle for some customers.

Espressif's Pre-Provisioning Service handles both of these for you.

What you get:

- **Device Attestation Authority:** Espressif is a CSA-authorized PAA who can issue unique DACs for each module
- **Device Attestation:** each module is provisioned with a unique DAC and its PAI, so it can prove authenticity to any Matter commissioner.
- **Uniqueness:** per-device credentials (discriminator, passcode, QR code), generated by the esp-matter-mfg-tool command you provide for Espressif to run. A manifest of the per-device data your command generates (e.g. MAC address, common name, DAC, and more) is available on request.
- **Security:** modules ship with Secure Boot and Flash Encryption enabled, and the DAC private key never leaves the module.

Matter Data Provisioning Options

The service supports three flows:

1. Espressif factory pre-provisioning

Espressif programs the Matter Provisioning Data onto the modules in Espressif's factory; along with your signed bootloader and firmware and ships modules that are ready to use. No Matter-specific tooling is required on your side. These modules will have flash encryption and secure boot enabled to prevent tampering in-transit when shipped.

2. Customer factory provisioning

You program Matter Provisioning Data at your own manufacturing line using Espressif's factory tool, which fetches DACs securely from Espressif's DAC Service cloud during production. It provisions the DAC, DAC private key, and PAI into the `esp_secure_cert` partition; you flash your own firmware afterwards.

3. In-field provisioning

Devices already deployed in the field can be upgraded to Matter. The DAC is delivered over a mutually authenticated cloud connection (with your cloud acting as a trust broker) and applied via an OTA update, so pre-Matter devices can gain Matter support.

1.6.2 Certification Assistance Service

Espressif's Matter Certification Assistance Service helps you carry your product through certification, drawing on Espressif's own experience with Matter Certification:

- Espressif can provide guidance on application documentation, certification workflows, and implementation best practices to help you prepare for certification.
- Espressif can run pre-certification test runs in its own lab to ensure your product passes all tests before formal submission.
- Espressif can work with the Authorized Test Laboratory (ATL) to assist and complete certification for your product.
- If certification issues are identified, Espressif can help debug the failure cases and provide recommendations for resolving firmware issues to achieve certification successfully.

See *Matter Certification* for the certification workflow with the SDK.

1.6.3 Getting Started

To get started or learn more, reach out to:

- **Sales / service requests:** sales@espressif.com
- **Matter PKI / DAC service:** matter-pki@espressif.com

1.6.4 Further Reading

- Accelerating Matter Device Manufacturing
- Matter: Improvements to Espressif's DAC Provisioning Service

1.7 Security Considerations

1.7.1 Overview

This guide provides an overview of the overall security features that should be considered while designing the products with Matter framework on ESP32 SoCs.

High level security goals are as follows:

1. Preventing untrustworthy code from being executed
2. Securing device identity (e.g., Matter DAC Private Key)
3. Secure storage for confidential data

1.7.2 Platform Security

Secure Boot

The Secure Boot feature ensures that only authenticated software can execute on the device. The Secure Boot process forms a chain of trust by verifying all **mutable** software entities involved in the boot-up process. Signature verification happens during both boot-up as well as in OTA updates.

Please refer to Secure Boot V2 guide for detailed documentation about this feature in ESP32-C6.

Flash Encryption

The Flash Encryption feature helps to encrypt the contents on the off-chip flash memory and thus provides the confidentiality aspect to the software or data stored in the flash memory.

Please refer to Flash Encryption guide for detailed documentation about this feature in ESP32-C6.

1.7.3 Product Security

Secure Storage

Secure storage refers to the application-specific data that can be stored in a secure manner on the device, i.e., off-chip flash memory. This is typically a read-write flash partition and holds device specific configuration data, e.g., Wi-Fi credentials.

ESP-IDF provides the **NVS (Non-volatile Storage)** management component which allows encrypted data partitions. This feature is tied with the platform flash encryption feature described earlier.

Please refer to the NVS Encryption for detailed documentation on the working and instructions to enable this feature in ESP32-C6.

Device Identity

Matter specification requires a unique Device Attestation Key (DAC) per device. This is a private ECDSA (secp256r1 curve) key that establishes the device identity to the Matter Ecosystem. DAC private needs to be protected from remote as well as physical attacks in the best possible way.

Recommended ways for DAC private key protection:

- DAC private key can be protected using *Flash Encryption* or *Secure Storage* schemes.

Important: Support for DAC private key protection mechanisms described above is available in the Matter crypto port layer for ESP32 platform.

Note: Espressif provides pre-provisioning service to build Matter-Compatible devices. This service also ensures the security of the DAC private key and configuration data. Please contact Espressif Sales for more information.

1.7.4 More Security Considerations

Please refer to the overall ESP-IDF Security Guide for more considerations related to the debug interfaces, network, transport and OTA updates related security.

1.7.5 Security Policy

The ESP-Matter GitHub repository has attached Security Policy Brief.

Advisories

- Espressif publishes critical Security Advisories, which includes security advisories regarding both hardware and software.
- The specific advisories of the ESP-Matter software components shall be published through the GitHub repository.

Software Updates

Critical security issues in the ESP-Matter components, ESP-IDF components and dependent third-party libraries are fixed as and when we find them or when they are reported to us. Gradually, we make the fixes available in all applicable release branches in ESP-Matter.

Important: We recommend periodically updating to the latest bugfix version of the ESP-Matter release to have all critical security fixes available.

1.8 Configuration options to optimize RAM and Flash

1.8.1 Overview

There are several configuration options available to optimize Flash and RAM storage. The following list highlights key options that significantly increase the free DRAM, heap, and reduce the flash footprint.

For more optimizations, we've also listed the reference links to esp-idf's optimization guide.

1.8.2 Configurations

Test Environment setup

All numbers mentioned below are collected in the following environment:

Note:

- These numbers may vary slightly in a different environment.
- All numbers are in bytes
- As we are using BLE only for commissioning, BLE memory is freed post commissioning, hence there is an increase in the free heap post commissioning. (`CONFIG_USE_BLE_ONLY_FOR_COMMISSIONING=y`)

- After building an example, some DRAM will be utilized, and the remaining DRAM will be allocated as heap. Therefore, a direct increase in the free DRAM will reflect as an increase in free heap.

Default Configuration

We have used the default light example here, and below listed are the static and dynamic sizes.

Disable the chip-shell

Console shell is helpful when developing/debugging the application, but may not be necessary in production. Disabling the shell can save space. Disable the below configuration option.

```
CONFIG_ENABLE_CHIP_SHELL=n
```

Adjust the dynamic endpoint count

The default dynamic endpoint count and default device type count is 16, which may be excessive for a normal application creating only 2 endpoints. eg: light, only has two endpoints, one for root endpoint and one for actual light. Adjusting this to a lower value, corresponding to the actual number of endpoints the application will create, can save DRAM.

Here, we have set the dynamic endpoint count and device type count to 2. Increase in the DRAM per endpoint/count is ~550 bytes.

```
CONFIG_ESP_MATTER_MAX_DYNAMIC_ENDPOINT_COUNT=2
CONFIG_ESP_MATTER_MAX_DEVICE_TYPE_COUNT=2
```

Use the newlib nano formatting

This optimization saves approximately 25-50K of flash, depending on the target. In our case, it results in a flash reduction of 47 KB.

Additionally, it lowers the high watermark of task stack for functions that call printf() or other string formatting functions. Fore more details please take a look at esp-idf's newlib nano formatting guide.

```
CONFIG_NEWLIB_NANO_FORMAT=y
```

BLE Optimizations

Since most devices will primarily operate as BLE peripherals and typically won't need more than one connection (especially if it's just a Matter app), we can optimize by reducing the maximum allowed connections, thereby saving DRAM. Additionally, given the peripheral nature of these devices, we can disable the central and observer roles, for further optimization. In current implementation, BLE is disabled once commissioning succeeds, so these optimizations do not contribute to free heap post-commissioning.

Below are the configuration options that can be set to achieve these optimizations.

```
CONFIG_NIMBLE_MAX_CONNECTIONS=1
CONFIG_BTDM_CTRL_BLE_MAX_CONN=1
CONFIG_BT_NIMBLE_MAX_CONNECTIONS=1
CONFIG_BT_NIMBLE_ROLE_CENTRAL=n
```

(continues on next page)

(continued from previous page)

```
CONFIG_BT_NIMBLE_ROLE_OBSERVER=n
CONFIG_BT_NIMBLE_MAX_BONDS=2
CONFIG_BT_NIMBLE_MAX_CCCDS=2
CONFIG_BT_NIMBLE_SECURITY_ENABLE=n
CONFIG_BT_NIMBLE_50_FEATURE_SUPPORT=n
CONFIG_BT_NIMBLE_WHITELIST_SIZE=1
CONFIG_BT_NIMBLE_GATT_MAX_PROCS=1
CONFIG_BT_NIMBLE_MSYS_1_BLOCK_COUNT=10
CONFIG_BT_NIMBLE_MSYS_1_BLOCK_SIZE=100
CONFIG_BT_NIMBLE_MSYS_2_BLOCK_COUNT=4
CONFIG_BT_NIMBLE_MSYS_2_BLOCK_SIZE=320
CONFIG_BT_NIMBLE_ACL_BUF_COUNT=5
CONFIG_BT_NIMBLE_HCI_EVT_HI_BUF_COUNT=5
CONFIG_BT_NIMBLE_HCI_EVT_LO_BUF_COUNT=3
CONFIG_BT_NIMBLE_ENABLE_CONN_REATTEMPT=n
```

Configuring logging event buffer

Matter events serve as a historical record, stored in chronological order in the logging event buffer. By reducing the buffer size we can potentially save the DRAM. However, it's important to note that this reduction could lead to the omission of events.

For instance, reducing the critical log buffer from 4K to 256 bytes could save 3K+ DRAM, but it comes with the trade-off of potentially missing critical events.

```
CONFIG_EVENT_LOGGING_CRIT_BUFFER_SIZE=256
CONFIG_EVENT_LOGGING_INFO_BUFFER_SIZE=256
CONFIG_EVENT_LOGGING_DEBUG_BUFFER_SIZE=256
CONFIG_MAX_EVENT_QUEUE_SIZE=20
```

Reduce ESP system event queue size and event task stack size can increase free heap size.

```
CONFIG_ESP_SYSTEM_EVENT_QUEUE_SIZE=16
CONFIG_ESP_SYSTEM_EVENT_TASK_STACK_SIZE=2048
CONFIG_MAX_EVENT_QUEUE_SIZE=20
```

Reduce the chip device event queue size can reduce IRAM size usage, lead to free heap increase.

```
CONFIG_MAX_EVENT_QUEUE_SIZE=20
```

Decrease pools size in Matter stack

Users can provide a project-specific configuration header file by setting the *CONFIG_CHIP_PROJECT_CONFIG* option in menuconfig. This configuration file can be used to define the sizes of certain Matter stack resource pools.

```
#define CHIP_CONFIG_SECURE_SESSION_POOL_SIZE 16
#define CHIP_CONFIG_UNAUTHENTICATED_CONNECTION_POOL_SIZE 2
#define CHIP_CONFIG_DEVICE_MAX_ACTIVE_DEVICES 1
#define CHIP_CONFIG_DEVICE_MAX_ACTIVE_CASE_CLIENTS 1
#define CHIP_IM_MAX_NUM_COMMAND_HANDLER   2
#define CHIP_IM_MAX_NUM_WRITE_HANDLER     2
#define CHIP_IM_MAX_NUM_TIMED_HANDLER     2
#define CHIP_IM_MAX_REPORTS_IN_FLIGHT     2
#define CHIP_IM_SERVER_MAX_NUM_DIRTY_SET  4
```

Relocate certain code from IRAM to flash memory

Relocating certain code from IRAM to flash can reduce IRAM usage, so increase available heap size. However, this may increase execution time.

Note: The options in this section may impact performance. Please perform thorough testing before using them in production.

Reduce BLE IRAM usage

Move most IRAM into flash. This will increase the usage of flash and reduce ble performance. Because the code is moved to the flash, the execution speed of the code is reduced. To have a small impact on performance, you need to enable flash suspend (SPI_FLASH_AUTO_SUSPEND).

```
CONFIG_BT_CTRL_RUN_IN_FLASH_ONLY=y
```

Place FreeRTOS functions into Flash

When enabled the selected Non-ISR FreeRTOS functions will be placed into Flash memory instead of IRAM. This saves up to 8KB of IRAM depending on which functions are used.

```
CONFIG_FREERTOS_PLACE_FUNCTIONS_INTO_FLASH=y
```

Place non-ISR ringbuf functions into flash

Place non-ISR ringbuf functions (like xRingbufferCreate/xRingbufferSend) into flash. This frees up IRAM, but the functions can no longer be called when the cache is disabled.

```
CONFIG_RINGBUF_PLACE_FUNCTIONS_INTO_FLASH=y
```

Use esp_flash implementation in ROM

Enable this flag to use new SPI flash driver functions from ROM instead of ESP-IDF. After enable CONFIG_SPI_FLASH_ROM_IMPL, will increase free IRAM. But may miss out on some flash features and support for new flash chips.

```
CONFIG_SPI_FLASH_ROM_IMPL=y
CONFIG_SPI_MASTER_ISR_IN_IRAM=n
CONFIG_SPI_SLAVE_ISR_IN_IRAM=n
```

Force the entire heap component to be placed in flash memory

Enable this flag to save up RAM space by placing the heap component in the flash memory Note that it is only safe to enable this configuration if no functions from esp_heap_caps.h or esp_heap_trace.h are called from IRAM ISR which runs when cache is disabled.

```
CONFIG_HEAP_PLACE_FUNCTION_INTO_FLASH=y
```

Reduce Task Stack Size

Reduce some task stack size can increase free heap size.

```
CONFIG_ESP_MAIN_TASK_STACK_SIZE=3072
CONFIG_ESP_TIMER_TASK_STACK_SIZE=2048
CONFIG_CHIP_TASK_STACK_SIZE=6144
```

Excluding Unused Matter Clusters

If the cluster implementation source files use a class derived from another class with virtual functions and instantiate a global object of this class, the linker may keep all the related symbols that may be used for this class in the vtable. To eliminate these symbols, you can deselect the unused Matter clusters under → `Component config` → `ESP Matter` → `Select Supported Matter Clusters`. Excluding unused clusters will help reduce flash and memory usage. The default configuration disables all unused clusters.

```
CONFIG_SUPPORT_ACCOUNT_LOGIN_CLUSTER=n
CONFIG_SUPPORT_ACTIVATED_CARBON_FILTER_MONITORING_CLUSTER=n
CONFIG_SUPPORT_AIR_QUALITY_CLUSTER=n
CONFIG_SUPPORT_APPLICATION_BASIC_CLUSTER=n
CONFIG_SUPPORT_APPLICATION_LAUNCHER_CLUSTER=n
CONFIG_SUPPORT_AUDIO_OUTPUT_CLUSTER=n
CONFIG_SUPPORT_BOOLEAN_STATE_CONFIGURATION_CLUSTER=n
CONFIG_SUPPORT_BRIDGED_DEVICE_BASIC_INFORMATION_CLUSTER=n
CONFIG_SUPPORT_CARBON_DIOXIDE_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_CARBON_MONOXIDE_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_CHANNEL_CLUSTER=n
CONFIG_SUPPORT_CHIME_CLUSTER=n
CONFIG_SUPPORT_COMMISSIONER_CONTROL_CLUSTER=n
CONFIG_SUPPORT_CONTENT_LAUNCHER_CLUSTER=n
CONFIG_SUPPORT_CONTENT_CONTROL_CLUSTER=n
CONFIG_SUPPORT_CONTENT_APP_OBSERVER_CLUSTER=n
CONFIG_SUPPORT_DEVICE_ENERGY_MANAGEMENT_CLUSTER=n
CONFIG_SUPPORT_DEVICE_ENERGY_MANAGEMENT_MODE_CLUSTER=n
CONFIG_SUPPORT_DIAGNOSTIC_LOGS_CLUSTER=n
CONFIG_SUPPORT_DISHWASHER_ALARM_CLUSTER=n
CONFIG_SUPPORT_DISHWASHER_MODE_CLUSTER=n
CONFIG_SUPPORT_MICROWAVE_OVEN_MODE_CLUSTER=n
CONFIG_SUPPORT_DOOR_LOCK_CLUSTER=n
CONFIG_SUPPORT_ECOSYSTEM_INFORMATION_CLUSTER=n
CONFIG_SUPPORT_ELECTRICAL_ENERGY_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_ELECTRICAL_POWER_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_ENERGY_EVSE_CLUSTER=n
CONFIG_SUPPORT_ENERGY_EVSE_MODE_CLUSTER=n
CONFIG_SUPPORT_ENERGY_PREFERENCE_CLUSTER=n
CONFIG_SUPPORT_FAN_CONTROL_CLUSTER=n
```

(continues on next page)

(continued from previous page)

```
CONFIG_SUPPORT_FAULT_INJECTION_CLUSTER=n
CONFIG_SUPPORT_FIXED_LABEL_CLUSTER=n
CONFIG_SUPPORT_FORMALDEHYDE_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_HEPA_FILTER_MONITORING_CLUSTER=n
CONFIG_SUPPORT_ICD_MANAGEMENT_CLUSTER=n
CONFIG_SUPPORT_KEYPAD_INPUT_CLUSTER=n
CONFIG_SUPPORT_LAUNDRY_WASHER_MODE_CLUSTER=n
CONFIG_SUPPORT_LOCALIZATION_CONFIGURATION_CLUSTER=n
CONFIG_SUPPORT_LOW_POWER_CLUSTER=n
CONFIG_SUPPORT_MEDIA_INPUT_CLUSTER=n
CONFIG_SUPPORT_MEDIA_PLAYBACK_CLUSTER=n
CONFIG_SUPPORT_MICROWAVE_OVEN_CONTROL_CLUSTER=n
CONFIG_SUPPORT_MESSAGES_CLUSTER=n
CONFIG_SUPPORT_MODE_SELECT_CLUSTER=n
CONFIG_SUPPORT_NITROGEN_DIOXIDE_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_SAMPLE_MEI_CLUSTER=n
CONFIG_SUPPORT_OCCUPANCY_SENSING_CLUSTER=n
CONFIG_SUPPORT_POWER_TOPOLOGY_CLUSTER=n
CONFIG_SUPPORT_OPERATIONAL_STATE_CLUSTER=n
CONFIG_SUPPORT_OPERATIONAL_STATE_OVEN_CLUSTER=n
CONFIG_SUPPORT_OPERATIONAL_STATE_RVC_CLUSTER=n
CONFIG_SUPPORT_OVEN_MODE_CLUSTER=n
CONFIG_SUPPORT_OZONE_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_PM10_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_PM1_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_PM2_5_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_POWER_SOURCE_CLUSTER=n
CONFIG_SUPPORT_POWER_SOURCE_CONFIGURATION_CLUSTER=n
CONFIG_SUPPORT_PUMP_CONFIGURATION_AND_CONTROL_CLUSTER=n
CONFIG_SUPPORT_RADON_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_REFRIGERATOR_ALARM_CLUSTER=n
CONFIG_SUPPORT_REFRIGERATOR_AND_TEMPERATURE_CONTROLLED_CABINET_MODE_CLUSTER=n
CONFIG_SUPPORT_RVC_CLEAN_MODE_CLUSTER=n
CONFIG_SUPPORT_RVC_RUN_MODE_CLUSTER=n
CONFIG_SUPPORT_SERVICE_AREA_CLUSTER=n
CONFIG_SUPPORT_SMOKE_CO_ALARM_CLUSTER=n
CONFIG_SUPPORT_SOFTWARE_DIAGNOSTICS_CLUSTER=n
CONFIG_SUPPORT_SWITCH_CLUSTER=n
CONFIG_SUPPORT_TARGET_NAVIGATOR_CLUSTER=n
CONFIG_SUPPORT_TEMPERATURE_CONTROL_CLUSTER=n
CONFIG_SUPPORT_THERMOSTAT_CLUSTER=n
CONFIG_SUPPORT_THERMOSTAT_USER_INTERFACE_CONFIGURATION_CLUSTER=n
CONFIG_SUPPORT_THREAD_BORDER_ROUTER_MANAGEMENT_CLUSTER=n
CONFIG_SUPPORT_THREAD_NETWORK_DIRECTORY_CLUSTER=n
CONFIG_SUPPORT_TIME_FORMAT_LOCALIZATION_CLUSTER=n
CONFIG_SUPPORT_TIME_SYNCHRONIZATION_CLUSTER=n
CONFIG_SUPPORT_TIMER_CLUSTER=n
CONFIG_SUPPORT_TVOC_CONCENTRATION_MEASUREMENT_CLUSTER=n
CONFIG_SUPPORT_UNIT_TESTING_CLUSTER=n
CONFIG_SUPPORT_USER_LABEL_CLUSTER=n
CONFIG_SUPPORT_VALVE_CONFIGURATION_AND_CONTROL_CLUSTER=n
CONFIG_SUPPORT_WAKE_ON_LAN_CLUSTER=n
CONFIG_SUPPORT_LAUNDRY_WASHER_CONTROLS_CLUSTER=n
CONFIG_SUPPORT_LAUNDRY_DRYER_CONTROLS_CLUSTER=n
CONFIG_SUPPORT_WIFI_NETWORK_MANAGEMENT_CLUSTER=n
CONFIG_SUPPORT_WINDOW_COVERING_CLUSTER=n
CONFIG_SUPPORT_WATER_HEATER_MANAGEMENT_CLUSTER=n
```

(continues on next page)

(continued from previous page)

```
CONFIG_SUPPORT_WATER_HEATER_MODE_CLUSTER=n
```

Table 5: Static memory stats

	Size	Decreased by
Used D/IRAM	179487	3736
Used Flash	1576436	36938

Table 6: Dynamic memory stats

	Free Heap	Increased by
On Bootup	44256	3876
Post Commissioning	77976	4164

Link Time Optimization (LTO)

Link Time Optimization (LTO) helps further optimize both binary size and runtime performance. You can read more about LTO in GCC's LTO documentation.

For details on enabling LTO in ESP-IDF, along with its effects and known limitations, please refer to ESP-IoT-Solution's LTO documentation.

As demonstrated in the example listed in ESP-IoT-Solution's LTO documentation, enabling LTO can result in around ~90 KB of flash savings, though it also increases stack usage by ~1700 bytes.

Moving BSS Segments to PSRAM to Reduce Memory Usage

The BSS section of the Matter libraries can consume significant internal memory. For devices with PSRAM, such as Matter Controller or Thread Border Router applications, moving these BSS segments to PSRAM significantly reduces the internal memory footprint.

The `esp_matter` component does this automatically. Enable `CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY` in menuconfig (under `Component config` → `SPI RAM config` → `Allow .bss segment placed in external memory`), and its built-in linker fragment places the Matter BSS in PSRAM instead of internal RAM. This covers `libCHIP.a`, `libesp_matter.a` and `libesp_matter_controller.a` when esp-matter is used from a local checkout, and `libespressif__esp_matter.a` when it is used from the Component Registry.

To move the BSS segments of additional libraries (for example your application's `libmain.a`), add your own `linker.lf` fragment and register it via `LDFRAGMENTS`. See the reference linker.lf and CMakeLists.txt :

```
idf_component_register(
    ...
    LDFRAGMENTS "linker.lf")
```

1.8.3 References for further optimizations

- RAM optimization
- Binary size optimization
- Speed Optimization
- ESP32 Memory Analysis — Case Study
- Optimizing IRAM can provide additional heap area but at the cost of execution speed. Relocating frequently-called functions from IRAM to flash may result in increased execution time

1.9 API Reference

1.9.1 Data Model

This has the high level APIs for Data Model.

1.9.2 Generated Data Model

When **Enable Generated Data Model** is enabled in `menuconfig` (`CONFIG_ESP_MATTER_ENABLE_GENERATED_DATA_MODEL`), ESP-Matter uses the auto-generated data model instead of the legacy (manually implemented) one.

Source Selection

- Generated implementation is used from:

```
components/esp_matter/data_model/generated/
```

- Legacy implementation (used when disabled) is located at:

```
components/esp_matter/data_model/legacy/
```

Directory Structure

- **Clusters**

```
generated/clusters/<cluster>/
```

Example (On/Off cluster):

```
on_off/on_off.h
on_off/on_off.cpp
on_off/on_off_ids.h
```

- **Device Types**

```
generated/device_types/<device>_device/
```

Example (Extended Color Light):

```
extended_color_light_device/extended_color_light_device.h
extended_color_light_device/extended_color_light_device.cpp
```

Regenerating the Data Model

To regenerate the data model files, run the following command from the ESP-Matter repository root:

```
python tools/data_model_gen/data_model_gen.py
```

For more details, refer to:

```
tools/data_model_gen/README.md
```

Summary

- Enable `CONFIG_ESP_MATTER_ENABLE_GENERATED_DATA_MODEL` to use generated clusters and device types.
- Disable it to use the legacy implementation.
- The generated model improves maintainability and alignment with the data model definition as per the Matter specification.

API reference

Header File

- components/esp_matter/data_model/esp_matter_data_model.h

Macros

ESP_MATTER_NVS_PART_NAME

1.9.3 Endpoint/Device Type

This has the high level APIs for Endpoint/Device Type.

API reference

Header File

- components/esp_matter/data_model/esp_matter_endpoint.h

1.9.4 Cluster

This has the high level APIs for Clusters.

API reference

Header File

- components/esp_matter/data_model/esp_matter_cluster.h

1.9.5 Attribute

This has the high level APIs for Attributes.

API reference

Header File

- components/esp_matter/data_model/esp_matter_attribute.h

1.9.6 Command

This has the high level APIs for Commands.

API reference

Header File

- components/esp_matter/data_model/esp_matter_command.h

1.9.7 Core Low Level

This has the Core Low Level APIs.

API reference

Header File

- components/esp_matter/esp_matter_core.h

Classes

class **ScopedChipStackLock**

RAII-style scoped lock for the Matter (CHIP) stack.

This class provides automatic locking and unlocking of the Matter stack using the RAII (Resource Acquisition Is Initialization) pattern. The lock is acquired when the object is constructed and automatically released when it goes out of scope.

Use this class whenever you need to access Matter stack APIs from a non-Matter context (e.g., from application tasks, callbacks, or interrupt handlers). The Matter stack is single-threaded and requires synchronization when accessed from external contexts.

Advantages

- **Exception Safety**: The lock is automatically released even if an exception occurs or the function returns early, preventing deadlocks.
- **Simplified Code**: No need to manually call lock/unlock functions or worry about missing unlock calls in error paths.
- **Scope-based Lifetime**: The lock duration is clearly defined by the variable's scope, making the code easier to read and maintain.
- **Copy Prevention**: The class is non-copyable to prevent accidental lock duplication which could lead to undefined behavior.

Note: Always use the smallest possible scope for the lock to minimize blocking time.

1.9.8 Event

This has the APIs for Events.

API reference

Header File

- components/esp_matter/data_model/esp_matter_event.h

1.9.9 Client

This has the APIs for Clients such as connect() and send_command().

API reference

Header File

- components/esp_matter/esp_matter_client.h

Classes

class **json_encodable** : public chip::app::DataModel::EncodableToTLV

Borrowed tagged JSON containing exactly one TLV value. The supplied tag replaces the JSON member's tag.

1.10 Enabling ESP-Insights in ESP-Matter

- To learn more about esp-insights and get started, please refer project README.md.
- Before building the app, enable the option *ESP_INSIGHTS_ENABLED* through menuconfig.
- Follow the steps present set up esp-insights account , and create an auth key.
- Create a file named *insights_auth_key.txt* in the project directory of the example.
- Download the auth key and copy Auth Key to the example.

```
cp /path/to/auth/key.txt path/to/esp-matter/examples/generic_switch/
↪insights_auth_key.txt
```

- Refer the esp-matter Generic Switch example to enable the traces and metrics reported by the esp32 tracing backend in the chip SDK on the insights dashboard and about how to use the auth key for enabling insights.
- Enable the option *ENABLE_ESP_INSIGHTS_SYSTEM_STATS* to get a report of the system metrics in the chip SDK on the insights dashboard.

1.11 Application User Guide

1.11.1 Delegate Implementation

As per the implementation in the connectedhomeip repository, some of the clusters require an application defined delegate to consume specific data and actions. In order to provide this flexibity to the application, esp-matter facilitates delegate initialization callbacks in the cluster create API. It is expected that application will define it's data and actions in the form of delegate-impl class and set the delegate while creating cluster/device type.

List of clusters with delegate:

- Account Login Cluster.
- Actions Cluster.
- Application Basic Cluster.

- Application Launcher Cluster.
- Audio Output Cluster.
- Boolean State Configuration Cluster.
- Camera AV Settings User Level Management Cluster.
- Channel Cluster.
- Chime Cluster.
- Closure Control Cluster.
- Closure Dimension Cluster.
- Commissioner Control Cluster.
- Commodity Price Cluster.
- Commodity Tariff Cluster.
- Content App Observer Cluster.
- Content Control Cluster.
- Content Launcher Cluster.
- Device Energy Management Cluster.
- Dishwasher Alarm Cluster.
- Door Lock Cluster.
- Electrical Grid Conditions Cluster.
- Electrical Power Measurement Cluster.
- Energy EVSE Cluster.
- Energy Preference Cluster.
- Fan Control Cluster.
- Keypad Input Cluster.
- Laundry Dryer Controls Cluster.
- Laundry Washer Controls Cluster.
- Low Power Cluster.
- Media Input Cluster.
- Media Playback Cluster.
- Messages Cluster.
- Microwave Oven Control Cluster.
- Mode Base Cluster (all derived types of clusters).
- Mode Select Cluster.
- Operational State Cluster.
- Power Topology Cluster.
- Push AV Stream Transport Cluster.
- Resource Monitoring Cluster.

- Service Area Cluster.
- Target Navigator Cluster.
- Thermostat Cluster.
- Time Synchronization Cluster.
- Valve Configuration And Control Cluster.
- Wake On Lan Cluster.
- Water Heater Management Cluster.
- WebRTC Transport Provider Cluster.
- Window Covering Cluster.

Below is the list of clusters with delegate and their reference implementation header files:

Account Login Cluster

Delegate Class	Reference Implementation
Account Login	Account Login Delegate

Actions Cluster

Delegate Class	Reference Implementation
Actions	None

Application Basic Cluster

Delegate Class	Reference Implementation
Application Basic	Application Basic Delegate

Application Launcher Cluster

Delegate Class	Reference Implementation
Application Launcher	Application Launcher Delegate

Audio Output Cluster

Delegate Class	Reference Implementation
Audio Output	Audio Output Delegate

Boolean State Configuration Cluster

Delegate Class	Reference Implementation
Boolean State Configuration	Boolean State Configuration Delegate

Camera AV Settings User Level Management Cluster

Delegate Class	Reference Implementation
Camera AV Settings User Level Management	Camera AV Settings User Level Management Delegate

Channel Cluster

Delegate Class	Reference Implementation
Channel	Channel Delegate

Chime Cluster

Delegate Class	Reference Implementation
Chime	Chime Delegate

Closure Control Cluster

Delegate Class	Reference Implementation
Closure Control	Closure Control Delegate

Closure Dimension Cluster

Delegate Class	Reference Implementation
Closure Dimension	Closure Dimension Delegate

Commissioner Control Cluster

Delegate Class	Reference Implementation
Commissioner Control	Commissioner Control Delegate

Commodity Price Cluster

Delegate Class	Reference Implementation
Commodity Price	Commodity Price Delegate

Commodity Tariff Cluster

Delegate Class	Reference Implementation
Commodity Tariff	Commodity Tariff Delegate

Content App Observer Cluster

Delegate Class	Reference Implementation
Content App Observer	None

Content Control Cluster

Delegate Class	Reference Implementation
Content Control	None

Content Launcher Cluster

Delegate Class	Reference Implementation
Content Launcher	Content Launcher Delegate

Device Energy Management Cluster

Delegate Class	Reference Implementation
Device Energy Management	Device Energy Management Delegate

Dishwasher Alarm Cluster

Delegate Class	Reference Implementation
Dishwasher Alarm	Dishwasher Alarm Delegate

Door Lock Cluster

Delegate Class	Reference Implementation
Door Lock	Door Lock Delegate

Electrical Grid Conditions Cluster

Delegate Class	Reference Implementation
Electrical Grid Conditions	Electrical Grid Conditions Delegate

Note:

Make sure that after implementing delegate class, you set the delegate class pointer at the time of creating cluster.

```
robotic_vacuum_cleaner::config_t rvc_config;
rvc_config.rvc_run_mode.delegate = object_of_delegate_class;
endpoint_t *endpoint = robotic_vacuum_cleaner::create(node, & rvc_config, ENDPOINT_
↪FLAG_NONE);
```

Electrical Power Measurement Cluster

Delegate Class	Reference Implementation
Electrical Power Measurement	Electrical Power Measurement Delegate

Energy Evse Cluster

Delegate Class	Reference Implementation
Energy Evse	Energy Evse Delegate

Energy Preference Cluster

Delegate Class	Reference Implementation
Energy Preference	Energy Preference Delegate

Fan Control Cluster

Delegate Class	Reference Implementation
Fan Control	Fan Control Delegate

Keypad Input Cluster

Delegate Class	Reference Implementation
Keypad Input	Keypad Input Delegate

Laundry Dryer Controls Cluster

Delegate Class	Reference Implementation
Laundry Dryer Controls	Laundry Dryer Controls Delegate

Laundry Washer Controls Cluster

Delegate Class	Reference Implementation
Laundry Washer Controls	Laundry Washer Controls Delegate

Low Power Cluster

Delegate Class	Reference Implementation
Low Power	Low Power Delegate

Media Input Cluster

Delegate Class	Reference Implementation
Media Input	Media Input Delegate

Media Playback Cluster

Delegate Class	Reference Implementation
Media Playback	Media Playback Delegate

Messages Cluster

Delegate Class	Reference Implementation
Messages	Messages Delegate

Microwave Oven Control Cluster

Delegate Class	Reference Implementation
Microwave Oven Control	Microwave Oven Control Delegate

Mode Base Cluster

It is a base cluster for ModeEVSE, ModeOven, ModeRVSRun, ModeRVSClean, ModeDishwasher, ModeWaterHeater, ModeRefrigerator, ModeLaundryWasher and ModeMicrowaveOven.

Delegate Class	Reference Implementation
Mode Base	Refrigeratore And TCC Mode
	Laundry Washer Mode
	Dish Washer Mode
	Rvc Run And Rvc Clean Mode
	Energy Evse Mode
	Microwave Oven Mode
	Device Energy Management Mode
	Water Heater Mode

Mode Select Cluster

Delegate Class	Reference Implementation
Mode Select	Mode Select Delegate

Operational State Cluster

Delegate Class	Reference Implementation
Operational State	Operational State Delegate

Power Topology Cluster

Delegate Class	Reference Implementation
Power Topology	Power Topology Delegate

Push AV Stream Transport Cluster

Delegate Class	Reference Implementation
Push AV Stream Transport	Push AV Stream Transport Delegate

Resource Monitoring Cluster

Delegate Class	Reference Implementation
Resource Monitoring	HEPA Filter Monitoring Delegate
	Activated Carbon Filter Monitoring Delegate
	Water Tank Level Monitoring Delegate

Service Area Cluster

Delegate Class	Reference Implementation
Service Area	Service Area Delegate

Target Navigator Cluster

Delegate Class	Reference Implementation
Target Navigator	Target Navigator Delegate

Thermostat Cluster

Delegate Class	Reference Implementation
Thermostat	Thermostat Delegate

Time Synchronization Cluster

Delegate Class	Reference Implementation
Time Synchronization	Time Synchronization Delegate

Valve Configuration And Control Cluster

Delegate Class	Reference Implementation
Valve Configuration And Control	Valve Configuration And Control Delegate

Wake On LAN Cluster

Delegate Class	Reference Implementation
Wake On LAN	Wake On LAN Delegate

Water Heater Management Cluster

Delegate Class	Reference Implementation
Water Heater Management	Water Heater Management Delegate

WebRTC Transport Provider Cluster

Delegate Class	Reference Implementation
WebRTC Transport Provider	WebRTC Transport Provider Delegate

Window Covering Cluster

Delegate Class	Reference Implementation
Window Covering	Window Covering Delegate

1.12 Copyrights and Licenses

1.12.1 Software Copyrights

All original source code in this repository is Copyright (C) 2021-2025 Espressif Systems. This source code is licensed under the Apache License 2.0 as described in the file LICENSE .

Additional third party copyrighted code is included under the following licenses.

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

Some examples use external components which are not Apache licensed, please check the copyright description in each example source code.

ESP-IDF

- ESP-IDF is licensed under the Apache License 2.0, as described in its copyright and license document.

Matter SDK

- The Matter SDK is licensed under the Apache License 2.0 as described in the file LICENSE file.

Managed Components from Espressif's Component Registry

- espressif/mdns is licensed under the Apache License 2.0.
- espressif/esp_secure_cert_mgr is licensed under the Apache License 2.0.
- espressif/esp_encrypted_img is licensed under the Apache License 2.0.
- espressif/esp_insights is licensed under the Apache License 2.0.
- espressif/led_strip is licensed under the Apache License 2.0.
- espressif/button is licensed under the Apache License 2.0.

Documentation

- HTML version of the ESP-Matter Programming Guide uses the Sphinx theme sphinx_idf_theme, which is Copyright (c) 2013-2020 Dave Snider, Read the Docs, Inc. & contributors, and Espressif Systems (Shanghai) CO., LTD. It is based on sphinx_rtd_theme. Both are licensed under MIT license.

1.13 A1 FAQs

1.13.1 A1.1 Compilation errors

I cannot build the application:

- Make sure you are on the correct esp-idf branch/release.
- Run `git submodule update --init --recursive` to make sure the submodules are at the correct heads.
- Make sure you have the correct `ESP_MATTER_PATH` (and any other required paths).
- Delete the `build/` directory and also `sdkconfig` and `sdkconfig.old` and then build again.
- If you are still facing issues, reproduce it on the default example and then raise a Github issue.

1.13.2 A1.2 Device commissioning using chip-tool

I cannot commission a new device through the chip-tool:

- If the `chip-tool pairing ble-wifi` command is failing, make sure the arguments are correct.
- Please check `chip-tool pairing ble-wifi --help` for argument help.
- Make sure Bluetooth is turned on, on your client (host).

Bluetooth/BLE does not work on by device:

- There is a known issues #13303 where BLE does not work on MacOS.
- In this case, the following can be done:
  - Run the device console command: `matter esp wifi connect <ssid> <password>`.
  - Run the chip-tool command for commissioning over ip: `chip-tool pairing onnetwork 0x7283 20202021`.
- If you are still facing issues, reproduce it on the default example for the device and then raise a Github issue.

1.13.3 A1.3 Device crashing

My device is crashing:

- Given the tight footprint requirements of the device, please make sure any issues in your code have been ruled out. If you believe the issue is with the Espressif SDK itself, please recreate the issue on the default example application (without any changes) and go through the following steps:
- Make sure you are on the correct esp-idf branch. Run `git submodule update —init —recursive` to make sure the submodules are at the correct heads.
- Make sure you have the correct `ESP_MATTER_PATH` (and any other paths) is (are) exported.
- Delete the `build/` directory and also `sdkconfig` and `sdkconfig.old` and then build and flash again.
- If you are still facing issues, reproduce it on the default example for the device and then raise a Github issue. Along with the details mentioned in the issue template, please share the following details:
  - The steps you followed to reproduce the issue.
  - The complete device logs taken over UART.
  - The .elf file from the build/ directory.
  - If you have gdb enabled, run the command `backtrace` and share the output of gdb too.

1.13.4 A1.4 Device not crashed but not responding

My device is not responding to commands:

- Make sure your device is commissioned successfully and is connected to the Wi-Fi.
- Make sure the node_id and the endpoint_id are correct in the command from chip-tool.
- If you are still facing issues, reproduce it on the default example for the device and then raise a Github issue. Along with the details mentioned in the issue template, please share the following details:
  - The steps you followed to reproduce the issue.
  - The complete device logs taken over UART.

1.13.5 A1.5 mDNS advertising failures: ESP-IDF mdns vs CHIP Minimal mDNS conflict

Rainmaker+Matter device logs show mDNS / DNS-SD advertising errors such as:

```
I (7785) chip[DIS]: Updating services using commissioning mode 0
E (7785) chip[DIS]: Failed to initialize advertiser: 3000008
E (7795) chip[DIS]: Failed to remove advertised services: 3
E (7795) chip[DIS]: Failed to advertise operational node: ...
E (7815) chip[DIS]: Failed to finalize service update: 3
```

What is happening:

- Error `3000008` in the log is the hex value `0x03000008`. This encodes an LwIP `ERR_USE` error: UDP port *5353* is already in use on the same network interface.
- Error `3` is `CHIP_ERROR_INCORRECT_STATE`. It is a follow-on error because the mDNS advertiser failed to initialize and later `RemoveServices` / `Advertise` / `FinalizeServiceUpdate` calls are rejected.

Why ESP-IDF mdns and CHIP Minimal mDNS conflict:

- The ESP-IDF `mdns` component (used by ESP RainMaker Local Control, on-network challenge-response, and other features on Wi-Fi/Ethernet) binds a UDP socket on port **5353** when `mdns_init()` is called.
- When `CONFIG_USE_MINIMAL_MDNS` is enabled, Matter uses CHIP **Minimal mDNS**, which also tries to bind UDP port **5353** through LwIP on the same interface.
- Only one listener can bind port 5353 on a given network interface. Whichever stack binds first prevents the other from binding the same port, and the second initialization fails with `ERR_USE`.
- `CONFIG_USE_MINIMAL_MDNS` defaults to `y`. Any firmware that also links the ESP-IDF `mdns` component (not only RainMaker + Matter) can hit this conflict unless Minimal mDNS is disabled.

This is especially common in **ESP RainMaker + Matter** firmware on Wi-Fi/Ethernet because RainMaker may call `mdns_init()` before Matter starts DNS-SD advertising.

Recommended configuration for RainMaker + Matter:

- Set `CONFIG_USE_MINIMAL_MDNS=n` (menuconfig: **Component config → CHIP Core → General Options → Use the minimal mDNS implementation shipped in the CHIP library**).
- Matter will then use **Platform mDNS** (`EspDnssdInit()`). Both stacks use the same ESP-IDF `mdns` component (`mdns_init()` is safe to call more than once). A single UDP socket on port 5353 is shared. Each side registers its own service records via `mdns_service_add()` (e.g. `_matter._tcp`, `_matterc._udp`, `_esp_local_ctrl._tcp`) without competing for the port.

1.13.6 A1.6 Onboard LED not working

The LED on my devkit is not working:

- Make sure you have selected the proper `device`. You can explicitly do that by exporting the `ESP_MATTER_DEVICE_PATH` to the correct path.
- Check the version of your board, and if it has the LED connected to a different pin. If it is different, you can change the `led_driver_config_t` accordingly in the `device.c` file.
- If you are still facing issues, reproduce it on the default example for the device and then raise a Github issue.

1.13.7 A1.7 Using Rotating Device Identifier

What is Rotating Device Identifier:

- The Rotating Device Identifier provides a non-trackable identifier which is unique per-device and that can be used in one or more of the following ways:
  - Provided to the vendor's customer support for help in pairing or establishing Node provenance;
  - Used programmatically to obtain a Node's Passcode or other information in order to provide a simplified setup flow. Note that the mechanism by which the Passcode may be obtained is outside of this specification. If the Rotating Device Identifier is to be used for this purpose, the system implementing this feature SHALL require proof of possession by the user at least once before providing the Passcode. The mechanism for this proof of possession, and validation of it, is outside of this specification.

How to use Rotating Device Identifier

- Enable the Rotating Device Identifier support in menuconfig.
- Add the `--enable-rotating-device-id` and add the `--rd-id-uid` to specify the `Rotating ID Unique ID` when use the `esp-matter-mfg-tool` to generate partition.bin file.

Difference between Rotating ID Unique ID and Unique ID

- The `Rotating ID Unique ID` is a parameter used to generate `Rotating Device Identifier`, it is a unique per-device identifier and shall consist of a randomly-generated 128-bit or longer octet string which shall be programmed during factory provisioning or delivered to the device by the vendor using secure means after a software update, it shall stay fixed during the lifetime of the device.
- The `Unique ID` is an attribute in `Basic Information Cluster`, it shall indicate a unique identifier for the device, which is constructed in a manufacturer specific manner. It may be constructed using a permanent device identifier (such as device MAC address) as basis. In order to prevent tracking:
  - it SHOULD NOT be identical to (or easily derived from) such permanent device identifier
  - it SHOULD be updated when the device is factory reset
  - it SHALL not be identical to the SerialNumber attribute
  - it SHALL not be printed on the product or delivered with the product

1.13.8 A1.8 ModuleNotFoundError: No module named 'lark'

Encountering the above error while building the esp-matter example could indicate that the steps outlined in the getting the repository section of the documentation were not properly followed.

The esp-matter example relies on several python dependencies that can be found in the requirements.txt . These dependencies must be installed into the python environment of the esp-idf to ensure that the example builds successfully.

One recommended approach to installing these requirements is by running the command `source $IDF_PATH/export.sh` before running `esp-matter/install.sh`, as suggested in the programming guide. However, if the error persists, you can try the following steps to resolve it:

```
cd esp-idf
source ./export.sh

cd esp-matter
python3 -m pip install -r requirements.txt

# Now examples will build without any error
```

(continues on next page)

(continued from previous page)

```
cd examples/...
idf.py build
```

1.13.9 A1.9 Why does free RAM increase after first commissioning

After the first commissioning, you may notice that the free RAM increases. This is because, by default, BLE is only used for the commissioning process. Once the commissioning is complete, BLE is deinitialized, and all the memory allocated to it is recovered. Here's the link to the implementation which frees the BLE memory .

However, if you want to continue using the BLE even after the commissioning process, you can disable the `CONFIG_USE_BLE_ONLY_FOR_COMMISSIONING`. This will ensure that the memory allocated to the BLE functionality is not released after the commissioning process, and the free RAM won't go up.

1.13.10 A1.10 How to generate Matter Onboarding Codes (QR Code and Manual Pairing Code)

When creating a factory partition using `esp-matter-mfg-tool`, both the QR code and manual pairing codes are generated.

Along with that, there are two more methods for generating Matter onboarding codes:

- Python script: generate_setup_payload.py

```
./generate_setup_payload.py --discriminator 3131 --passcode 20201111 \
                            --vendor-id 65521 --product-id 32768 \
                            --commissioning-flow 0 --discovery-cap-
↪bitmask 2
```

- chip-tool

```
// Generate the QR Code
chip-tool payload generate-qrcode --discriminator 3131 --setup-pin-code␣
↪20201111 \
                                  --vendor-id 0xFFF1 --product-id 0x8004 \
                                  --version 0 --commissioning-mode 0 --
↪rendezvous 2

// Generates the short manual pairing code (11-digit).
chip-tool payload generate-manualcode --discriminator 3131 --setup-pin-
↪code 20201111 \
                                      --version 0 --commissioning-mode 0

// To generate a long manual pairing code (21-digit) that includes both␣
↪the vendor ID and product ID,
// --commissioning-mode parameter must be set to either 1 or 2,␣
↪indicating a non-standard commissioning flow.
chip-tool payload generate-manualcode --discriminator 3131 --setup-pin-
↪code 20201111 \
                                      --vendor-id 0xFFF1 --product-id␣
↪0x8004 \
                                      --version 0 --commissioning-mode 1
```

To create a QR code image, copy the QR code text and paste it into CHIP QR Code.

1.13.11 A1.11 Chip stack locking error ... Code is unsafe/racy

```
E (84728) chip[DL]: Chip stack locking error at 'src/system/
↪SystemLayerImplFreeRTOS.cpp:55'. Code is unsafe/racy
E (84728) chip[-]: chipDie chipDie chipDie
abort() was called at PC 0x40139b7f on core 0
0x40139b7f:␣
↪chip::Platform::Internal::AssertChipStackLockedByCurrentThread(char const*,
↪ int) at /home/jonathan/Desktop/Workspace/firmware/build/esp-idf/chip/../..
↪/../../esp-matter/connectedhomeip/connectedhomeip/config/esp32/third_party/
↪connectedhomeip/src/lib/support/CodeUtils.h:508
 (inlined by) chipDie at /home/jonathan/Desktop/Workspace/firmware/build/esp-
↪idf/chip/../../../../esp-matter/connectedhomeip/connectedhomeip/config/
↪esp32/third_party/connectedhomeip/src/lib/support/CodeUtils.h:518
 (inlined by)␣
↪chip::Platform::Internal::AssertChipStackLockedByCurrentThread(char const*,
↪ int) at /home/jonathan/Desktop/Workspace/firmware/build/esp-idf/chip/../..
↪/../../esp-matter/connectedhomeip/connectedhomeip/config/esp32/third_party/
↪connectedhomeip/src/platform/LockTracker.cpp:36
```

When interacting with Matter resources, it is necessary to perform the operations from within the Matter thread to avoid assertion errors. This applies to tasks such as getting and setting attributes, invoking commands, and performing operations using the server's object, such as opening or closing the commissioning window.

To address this, there are two possible approaches:

- Using ScopedChipStackLock to lock the Matter thread

  ```
  lock::ScopedChipStackLock lock(portMAX_DELAY);
  ... // eg: access Matter attribute, open/close commissioning window.
  ```

- Scheduling the work on Matter thread

  ```
  static void WorkHandler(intptr_t context);
  {
      ... // Do the stuff
  }
  chip::DeviceLayer::PlatformMgr().ScheduleWork(WorkHandler, <intptr_t>
  ↪(nullptr));
  ```

1.13.12 A1.12 Firmware Version Number

Similar to the ESP-IDF's application versioning scheme, the ESP-Matter SDK provides two options for setting the firmware version. It depends on CONFIG_APP_PROJECT_VER_FROM_CONFIG option and by default option is disabled.

If the `CONFIG_APP_PROJECT_VER_FROM_CONFIG` option is disabled, you need to set the version and version string by defining the CMake variables in the project's `CMakeLists.txt` file. All the examples use this scheme and have these variables set. Here's an example:

```
set(PROJECT_VER "1.0")
set(PROJECT_VER_NUMBER 1)
```

On the other hand, if the `CONFIG_APP_PROJECT_VER_FROM_CONFIG` option is enabled, you need to set the version using the following configuration options:

- **Software Version**
  Set the `CONFIG_DEVICE_SOFTWARE_VERSION_NUMBER` option. (Component config -> CHIP Device Layer -> Device Identification Options -> Device Software Version Number)
- **Software Version String**
  Set the `CONFIG_APP_PROJECT_VER` option. (Application manager -> Get the project version from Kconfig)

Note:

- Ensure you use the correct versioning scheme when building the OTA image.
- Verify that the software version number in the firmware matches the one specified in the Matter OTA header.
- The software version number of the OTA image must be numerically higher.
- If you need to perform a functional rollback, the version number in the OTA image must be higher than the current version, even though the binary content may match the previous OTA image.

1.13.13 A1.13 Stuck at "Solving dependencies requirements ....."

When building an example, if it is stuck at "Solving dependencies requirements…" you can resolve this issue by clearing the component manager cache.

```
# On Linux
rm -rf ~/.cache/Espressif/ComponentManager

# On macOS
rm -rf ~/Library/Caches/Espressif/ComponentManager
```

1.13.14 A1.14 ESP32-C2 log garbled, unable to perform Matter commissioning and other abnormal issues

When encountering the above issues, the following possible causes may exist: 1. Incorrect baud rate settings. See UART console baud rate 2. Incorrect XTAL crystal frequency settings. The default XTAL crystal frequency in the SDK examples is 26 Mhz, if the ESP32-C2 board used for testing is 40 MHz, please change the configuration as *CONFIG_XTAL_FREQ_40=y*. See Main XTAL frequency You can check the XTAL frequency with this command.

```
$ esptool.py flash_id
esptool.py v4.7.0
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP32-C2
Chip is ESP32-C2 (revision v1.0)
Features: WiFi, BLE
Crystal is 26MHz
MAC: 08:3a:8d:49:b3:90
```

1.13.15 A1.15 Generating Matter Onboarding Codes on the device itself

The Passcode serves as both proof of possession for the device and the shared secret needed to establish the initial secure channel for onboarding.

For best practices in Passcode generation and storage on the device, refer to **Section 5.1.7: Generation of the Passcode** in the Core Matter Specification.

Ideally, devices should only store the Spake2p verifier, not the Passcode itself. If the Passcode is stored on the device, it must be physically separated from the Spake2p verifier's location and must be accessible only through local interface and must not be accessible to the unit handling the Spake2p verifier.

For devices capable of displaying the onboarding payload, the use of a dynamic Passcode is recommended.

The Light Switch example in the SDK demonstrates the use of a dynamic Passcode. It implements a custom Commissionable Data Provider that generates the dynamic Passcode, along with the corresponding Spake2p verifier and onboarding payload, directly on the device.

Please check #1128 and #1126 for relevant discussion on Github issue

1.13.16 A1.16 Using BLE after Matter commissioning

Most Matter applications do not require BLE after commissioning. By default, BLE is deinitialized after commissioning to reclaim RAM and increase the available free heap. Refer to *Why does free RAM increase after first commissioning* for more details.

However, if BLE functionality is needed even after commissioning, you can disable the `CONFIG_USE_BLE_ONLY_FOR_COMMISSIONING` option. This ensures that the memory allocated to BLE functionality is retained, allowing BLE to be used for other purposes post-commissioning.

After commissioning is complete, Matter will stop advertising, but the application can utilize BLE for other roles or operations. e.g. BLE Peripheral, BLE Central, etc.

To learn more, refer to the bleprph and blecent examples in `esp-idf/examples/bluetooth/nimble`. These examples demonstrate BLE Peripheral and BLE Central roles. It also provides the step-by-step tutorial for building such devices.

For implementation details on Peripheral and Central roles, refer to the bleprph_advertise() and blecent_scan() functions in the respective examples.

BLE Central role is disabled by default in the esp-matter SDK's default example configurations. Please enable `CONFIG_BT_NIMBLE_ROLE_CENTRAL` option if you plan to use BLE Central role.

Note: Above mentioned details apply specifically to the NimBLE host.

For more advanced BLE usage, you can use the external platform feature. It also serves as a way to integrate custom BLE usage with Matter.

Please refer to the advance setup section in the programming guide. This has been demonstrated in the blemesh_bridge examples.

1.13.17 A1.17 TC-SC-4.9 (Route Information Option / RIO) failure

For Wi-Fi products using LwIP, Matter certification test TC-SC-4.9 checks that the device can receive Router Advertisements (RA) with Route Information Option (RIO) and install routes so traffic to Thread can use the correct path. Please ensure the Kconfig options under **Route Information Option (RIO) notes** in *Matter Certification* are enabled.

1.13.18 A1.18 Updating attribute marked as ATTRIBUTE_FLAG_MANAGED_INTERNALLY

When an attribute is marked with the flag `ATTRIBUTE_FLAG_MANAGED_INTERNALLY`, application can not directly modify the attribute's value using `esp_matter::attribute::update`. To update such attributes, retrieve the corresponding delegate or instance within the cluster implementation and perform the update through it. For example, to update `DefaultOTAProviders` attribute in `OTASoftwareUpdateRequestor` cluster, use the following code:

```
chip::OTARequestorInterface * request = chip::GetRequestorInstance();
if (request) {
    chip::OTARequestorInterface::ProviderLocationType provider;
    provider.providerNodeID = 123;
    provider.endpoint = 0;
    provider.fabricIndex = 1;
    request->AddDefaultOtaProvider(provider);
}
```

1.13.19 A1.19 attribute::update() returns ESP_ERR_NOT_SUPPORTED for some attributes

Some clusters (e.g. `TemperatureMeasurement`) have been migrated to code-driven implementations, where the cluster instance owns the attribute values instead of the esp-matter attribute store. For such clusters, calling `esp_matter::attribute::update()` or `esp_matter::attribute::set_val()` on a **non-writable** attribute (e.g. `MeasuredValue`) returns `ESP_ERR_NOT_SUPPORTED`, because the write would land in the legacy store and never reach the controller.

Earlier versions silently returned `ESP_OK` while the value was never actually updated; the error return makes this failure visible. See #1798 for details.

What to do instead:

- **Writable attributes** are unaffected — `update()` and `set_val()` continue to work for them.
- For non-writable attributes of a code-driven cluster, update the value through the cluster implementation itself (its instance or delegate), similar to attributes marked `ATTRIBUTE_FLAG_MANAGED_INTERNALLY` described above. For example, to update `MeasuredValue` of the `TemperatureMeasurement` cluster, retrieve the cluster instance from the provider registry and use its setter:

```
#include <app/clusters/temperature-measurement-server/
↪TemperatureMeasurementCluster.h>

using namespace chip::app;
using namespace chip::app::Clusters;

auto *cluster = esp_matter::data_model::provider::get_instance().registry().Get(
    ConcreteClusterPath(endpoint_id, TemperatureMeasurement::Id));
if (cluster) {
    static_cast<TemperatureMeasurementCluster *>(cluster)->SetMeasuredValue(
        DataModel::MakeNullable<int16_t>(2550) /* 25.50 °C */);
}
```

As with any Matter data-model access, call this from the Matter thread or with the chip stack lock held (see *Chip stack locking error … Code is unsafe/racy* above).

1.13.20 A1.20 Devices with custom means of network configuration (CustomNetworkConfig)

Some devices configure their network through custom means — such as a rich user interface, manufacturer-specific provisioning, custom commissioning flows, or a future IP-compliant network technology not yet directly supported by the Network Commissioning cluster — rather than through the standard Matter Network Commissioning cluster.

The Matter specification defines a `CustomNetworkConfig` condition for such devices. When this condition is set, the Network Commissioning cluster is not required on the Root Node endpoint.

To enable this in the ESP-Matter SDK:

1. Enable the `CONFIG_CUSTOM_NETWORK_CONFIG` option in menuconfig Menu path: (Top) -> Component config -> ESP Matter -> Use custom network commissioning (skip Network Commissioning cluster)
2. When enabled, the Network Commissioning cluster will not be added to the root node endpoint (endpoint 0). The device is expected to handle network configuration through its own out-of-band mechanism.

1.13.21 A1.21 Certification test TC-RR-1.1 fails with "MaxGroupKeysPerFabric must be at least 4"

The Groupcast test plan update raised the certification floor for `MaxGroupKeysPerFabric` to 4 (3 group key sets plus the Identity Protection Key), matching the connectedhomeip core default. However, the ESP32 build reads this limit from the `CONFIG_MAX_GROUP_KEYS_PER_FABRIC` option, which defaults to 3 in some connectedhomeip versions.

If TC-RR-1.1 reports this failure, add the following to your project's `sdkconfig.defaults` (or set it via menuconfig, Menu path: (Top) -> Component config -> CHIP Core -> General Options -> Maximum Number of Group Key Sets per Fabric):

```
CONFIG_MAX_GROUP_KEYS_PER_FABRIC=4
```

Note that raising the limit slightly increases the group key storage reserved per fabric.