Table of Contents

Introduction

Overview

This project aims to develop a **decentralized automation system** for **Controlled Environment Agriculture** (CEA).

The system follows a modular structure, where each module has its own subnetwork of accessory devices, such as sensors and actuators, that autonomously monitor and control environmental conditions without relying on a central hub.

The system prioritizes interoperability, enabling the integration of devices from different manufacturers and the replacement of existing ones without major updates. It also implements security measures to ensure safe data transmission and access control, maintaining system functionality and scalability.

Motivation

Advancements in IoT-based monitoring and control systems have improved automation in agriculture, increasing efficiency and productivity.

Precision Agriculture enhances field farming by using sensor data, GPS, and automated equipment to precisely manage inputs like water, fertilizers, and pesticides. *Controlled-Environment Agriculture* extends this approach by eliminating limitations such as external environmental factors, soil variability, and large-scale deployment challenges through enclosed systems where temperature, humidity, light, and CO_2 are actively regulated. This allows for year-round production, reduces reliance on weather conditions, and ensures consistent crop yields, making it more reliable than traditional open-field farming.

However, many farming automation startups struggle to succeed, primarily due to the high cost of research and development. Building advanced automation systems requires substantial investment, which raises food production costs and makes it difficult for startups to compete with traditional farms that operate at lower expenses.

A major factor driving these costs is the reliance on centralized servers or cloud resources for data processing and control. This setup requires complex communication infrastructure, increases operational expenses, and complicates scalability, particularly for larger farms. Additionally, centralized systems create a single point of failure - if the

central server or cloud service fails or is compromised, the entire operation can be disrupted. For example, an AWS IoT outage in 2020 caused significant downtime for applications relying on AWS IoT Core, demonstrating the risks of centralized dependency.

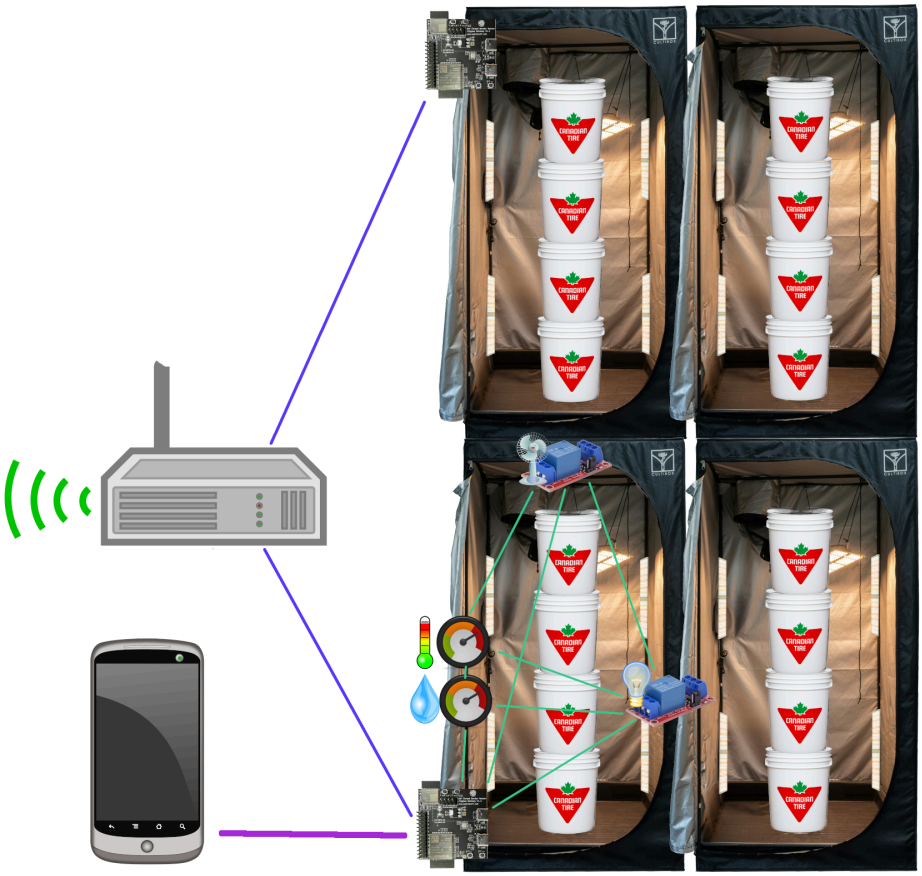
Another issue is the difficulty of integrating components from different manufacturers. Automated farms use a variety of sensors, actuators, and equipment, but differences in APIs and communication protocols can make it challenging to connect these systems smoothly.

Discussion

The system leverages the Matter protocol, which ensures compatibility between devices from different manufacturers. This standardisation simplifies integration, replacement, and future expansion of devices. However, the high costs associated with Matter certification – including a $7,000 annual membership fee for the Connectivity Standards Alliance (CSA) and $3,000 per product certification – pose significant challenges for small manufacturers and farms operating on tight budgets.

Future advancements in this system could include the adoption of microfluidics within hydroponic systems. Such developments would enhance resource efficiency and allow for precise control over environmental conditions tailored to specific crops. These innovations align with the system's modular design philosophy, which prioritises adaptability and scalability in agricultural automation.

System Design



The system is a modular IoT solution designed for decentralized automation in **Controlled Environment Agriculture** (CEA).

Each module, called a **Grow Chamber** (Grow Chamber), independently maintains optimal environmental conditions for specific plants and includes the following components:

- **Root Chamber** contains the plant's root system and nutrient solution.
- **Lighting** provides artificial light for photosynthesis.
- **Ventilation** regulates temperature, humidity, and CO_2 levels.

Environmental conditions are managed through a mesh network of **Accessory Devices** (Accessory Devices):

- **Sensors** monitor temperature, humidity, light intensity, and CO_2 levels.
- **Actuators** adjust conditions by controlling fog generation, ventilation, and lighting.

The **Orchestrator** (Orchestrator) devices allow farmers to set up and manage Grow Chambers, while the **Dashboard** (Dashboard) provides a graphical user interface for monitoring and control.

Grow Chamber

Overview

A **Grow Chamber** is a self-contained module within the system, designed to provide an independent, controlled environment for plant growth.

System Design

Commercially available grow tents for indoor plant cultivation function as individual **Grow Chambers**, providing structure and isolating the growing environment.

A bucket at the centre of the tent acts as a **Root Chamber**. It contains multiple **Plant Slots** that hold net pots filled with growing medium, where seeds are planted.

As the seeds germinate, their stems grow upward to form a canopy, while roots develop below. The **Fogponics System** delivers a fine, nutrient-rich mist to the roots, generated by an ultrasonic mist maker submerged in a water-nutrient solution.

Lighting panels positioned in the corners provide the necessary wavelengths for growth. The walls of the Tent and the Root Chamber are lined with Mylar, a reflective material that enhances light distribution for the plants.

The **Ventilation system** maintains airflow and regulates the temperature within the Grow chamber.

Hardware Components

The hardware components were sourced from easily accessible suppliers, such as Canadian Tire and Amazon.

Grow Chamber Assembly

Required Components:

- Grow Tent for indoor plant cultivation (24" x 24" x 48", 61cm x 61cm x 122cm)
- Full Spectrum Grow Light for Indoor Plants
- Inline Fan: 4-inch diameter
- Carbon Filter: 4-inch flange diameter, 14 inches long
- Ducting: 4-inch diameter, 8 feet long
- Rubber Coupler Connector: Compatible with a 4-inch diameter

Root Chamber Assembly

Required Components:

- Canadian Tire Plastic Food Grade Safe Bucket, 5-Gal/19-L (37.20 cm x 31.00 cm) or 7.5-L
- Mylar Films
- Ultrasonic Mist Maker, 400ml/h
- Brushless DC Fan, 12V, IP67, 120mm x 25mm
- Peristaltic Pump, 12V 3x5mm
- Silicone tube, 3x5mm 5m
- Aquarium air pump
- FS400-SHT3X Soil Temperature and Humidity Sensor (SHT35)
- Ultrasonic Distance Sensor, water-resistant
- 2" Net pots

Required Tools:

- Heat Gun
- Hole Saw, 1.5" diameter
- Wire Stripper
- 12V 1A AC/DC Power Adapter
- 12V DC Power Jack Connector Adapter, Male Plug & Female Socket Set, 5.5 X 2.1mm

Seed Planting

Required Components:

- Fertilizer
- Clay Pellets
- Organic cotton balls
- Arugula Seeds

Ventilation

The tent's internal volume is approximately 64,512 cubic inches, which converts to about 37.34 cubic feet: 32" × 32" × 63" ÷ 1,728 ≈ 37.34 ft^3

The 130 CFM inline duct fan can replace 37 cubic feet of air in about 17 seconds: 37 ft^3 ÷ 130 ft^3/min ≈ 0.28 minutes

For effective growing conditions, it's recommended to perform a full air exchange every one to three minutes. This ensures fresh air circulation, stable temperature, and proper humidity levels.

Automation

Overview

The Automation System regulates the environment and resource availability based on plant growth stages, ensuring optimal conditions from germination to maturity.

Hardware

The system uses development boards from **Espressif** (Espressif).

The **ESP32-H2** board is used for low-power sensors that monitor temperature, humidity, and CO_2 levels.

Actuators such as mist makers, exhaust fans, and LED grow lights require more power and can be run on more affordable boards like the **ESP32-DevKitM-1**.

The system also uses a **Thread Border Router board** to enable communication between Thread-enabled devices and external networks.

Software

Development is based on ESP-IDF ("ESP-IDF Framework" in "Espressif"), the official framework for ESP32, with ESP-Matter ("ESP Matter Solution" in "Espressif") integrated for Matter device support.

The environment is set up on **Ubuntu Desktop**.

Network Communication

The **Orchestrator** (Orchestrator) uses the **Thread** (Thread) protocol to create a low-power, wireless mesh network for **accessory devices** (Accessory Devices). All accessory devices also implement the **Matter** (Matter) protocol, allowing any Matter Controller to control and communicate with them, regardless of manufacturer.

The Orchestrator also acts as a **Thread Border Router** (Thread Border Routers), bridging Thread and Wi-Fi so that Matter controllers on Wi-Fi—such as the CHIP tool or mobile apps—can communicate with Matter accessory devices on the Thread network.

The **Dashboard** (Dashboard) connects to the Orchestrator over Wi-Fi using WebSocket, either by joining the Orchestrator's access point or by connecting through the same Wi-Fi network when the Orchestrator is in station mode.

The Orchestrator also supports connection to a cloud WebSocket relay, which the Dashboard uses to send remote commands to the Orchestrator.

Background

This section provides an overview of the relevant background information necessary to understand the project's context, including key concepts and technologies.

Thread

Overview

Thread is an IPv6-based networking protocol designed for low-power Internet of Things devices in an IEEE 802.15.4-2006 wireless mesh network.

Matter (Matter) devices can communicate with each other using Thread as a transport protocol.

OpenThread (OT) is an open-source implementation of the **Thread** networking protocol.

The Thread Network prevents **single points of failure** through device redundancy and autonomous role transitions, but in topologies like a Thread Network Partition with only one Border Router, failure of that Border Router can disrupt external network access due to the lack of a backup.

Devices in a Thread Network can communicate directly without an active Border Router. Although, having multiple Border Routers allows connections to one or more external networks, improving reliability and redundancy.

Architecture

OpenThread runs on various OS or bare-metal systems, including Linux and ESP32, due to its narrow abstraction layer and portable C/C++ design. It supports the following designs:

- **System-on-chip (SoC):** a single chip combines a processor and an 802.15.4 radio for Thread. In this design, both OpenThread and the device's main software (application layer) run on the same processor. This mode is used in **end devices** like sensors and actuators (ESP32-H2, ESP32-C6).
- **Radio Co-Processor (RCP):** the host processor runs the full OpenThread stack, while a separate Thread radio chip handles only the PHY and MAC layers of IEEE 802.15.4. They communicate using the Spinel protocol over SPI or UART. Since the host processor manages all networking tasks, it typically remains powered on, making this setup ideal for devices that prioritize performance over power efficiency. This mode is

used in **Thread Border Routers**, where a host processor, such as an ESP32-S3 or ESP32, is paired with an ESP32-H2 or ESP32-C6 as the Thread radio co-processor.

- **Network Co-Processor (NCP):** OpenThread runs on a Thread-enabled SoC, while the application software runs separately on a **host processor**. These two processors communicate using wpantund over SPI or UART, with the Spinel protocol managing the connection. Unlike RCP, this design allows the host processor to sleep, while the Thread-enabled chip stays active to maintain network connectivity. This design is beneficial for power-saving devices that require a constant network connection but do not need the host processor to remain active at all times.

Device Types

There are two types of Thread devices:

- **Minimal Thread Device (MTD)** communicates only with its FTD parent.
- **Full Thread Device (FTD)** communicates with other FTDs and its MTD children.

Device Roles

A Thread device can have multiple roles. For example, it can act as a Router, Border Router, and Commissioner simultaneously. See also Commissioning Roles ("Commissioning Roles" in "Thread").

Router

A **Router** is an FTD ("Device Types" in "Thread") that provides routing services in a Thread Network. It forwards packets, maintains routing information, and serves as a Parent for End Devices ("End Device" in "Thread"). Additionally, Routers support Commissioning ("Commissioning" in "Thread"), handling device joining and security services.

A **Leader** is a special Router that manages a Thread Network Partition. Each Partition has one Leader, elected dynamically from the available Routers. If the Leader fails, another Router automatically takes over. The Leader is responsible for:

- Assigning and managing Router IDs.
- Maintaining Thread Network Data.

- Coordinating network operations.

End Device

An **End Device** is a device that connects to the Thread Network but does not forward packets like a Router ("Router" in "Thread"). Instead, it relies on a Parent Router for communication. End Devices are typically low-power devices, such as sensors or actuators, that do not need to maintain full network topology information.

Depending on their capabilities and power management strategies, End Devices are classified into the following types:

- **Router-Eligible End Device (REED)** is an FTD ("Device Types" in "Thread") that operates as an End Device but can be promoted to a Router ("Router" in "Thread") if the network requires additional routing capacity. It does not forward messages but maintains connections with Routers and supports Commissioning ("Commissioning" in "Thread").
- A **Full End Device (FED)** is an FTD ("Device Types" in "Thread") that operates as an End Device and will never become a Router. Unlike a REED, a FED cannot be promoted to a Router.
- A **Minimal End Device (MED)** is an MTD ("Device Types" in "Thread") that keeps its radio on at all times and can communicate with its Parent Router whenever needed. MEDs do not forward messages or participate in routing.
- A **Sleepy End Device (SED)** is an MTD ("Device Types" in "Thread") that turns off its radio when idle to save power. It periodically wakes up to communicate with its Parent Router but does not forward messages.

Border Router

A **Thread Border Router** is a device that connects a Thread Network to external IP networks, such as Wi-Fi or Ethernet. It allows Thread devices to communicate with the internet or other smart home ecosystems.

OpenThread's implementation of a Border Router is called **OpenThread Border Router (OTBR)**.

Espressif (Espressif) provides the ESP Thread Border Router SDK ("ESP Thread Border Router Solution" in "Espressif") and hardware platforms ("ESP Thread Border Router

Solution" in "Espressif") for building Thread Border Routers.

Network

Thread Network Data

Thread Network Data is a collection of network-related information managed and distributed by the Leader ("Router" in "Thread") in a Thread network.

It includes the following details:

- Border Routers
- On-mesh prefixes
- External routes
- 6LoWPAN contexts
- Network commissioning parameters

The Leader ("Router" in "Thread") collects and updates Thread Network Data, distributing changes using MLE (Mesh Link Establishment) messages. Routers and REEDs store the full data, while End Devices (MTDs) ("End Device" in "Thread") can store either the full set or only a stable subset to save resources.

Active Operational Dataset

The **Active Operational Dataset** contains the configuration settings that Thread devices use to connect and operate within a specific Thread network:

- **Active Timestamp** determines dataset priority.
- **Channel** is a PHY-layer channel for network communication.
- **Channel Mask** defines channels for network discovery and scanning.
- **Extended PAN ID** is a unique identifier for the Thread network.
- **Mesh-Local Prefix** is an IPv6 prefix for local device communication.

- **Network Key** is a 128-bit key used to secure communication within the Thread network.
- **Network Name** is a human-readable network identifier.
- **PAN ID** is a MAC-layer identifier for data transmissions.
- **PSKc** is a security key for network authentication.
- **Security Policy** specifies allowed and restricted security operations.

Commissioning

The **Joiner ID** is derived from the Joiner Discerner if one is set; otherwise, it is derived from the device's factory-assigned EUI-64 using SHA-256. This ID also serves as the device's IEEE 802.15.4 Extended Address during commissioning. When the device joins a Thread network, it automatically receives the Network Key.

Commissioning Roles

The Thread Specification defines several commissioning roles within the **Mesh Commissioning Protocol (MeshCoP):**

- **Commissioner** manages device authentication and onboarding in a Thread network. It provides network credentials to new devices (**Joiners**) and can update network parameters or perform diagnostics. Types of Commissioners:
  - **On-Mesh Commissioner** operates inside the Thread network and has full control over commissioning.
  - **External Commissioner** resides outside the Thread network and connects via a **Border Agent** (e.g., a mobile app or cloud service).
  - **Native Commissioner** uses the same Thread interface as the mesh to manage commissioning.
- **Joiner** is a Thread device attempting to join a commissioned Thread network. It does not have network credentials and must authenticate with a Commissioner to gain access.

- **Border Agent (BA)** relays messages between an External or Native Commissioner and the Thread Network, ensuring secure communication between external networks and the Thread network.
- **Joiner Router** is a Router or REED that assists a Joiner in communicating with a Commissioner when the Joiner is not within direct range. It does not perform full routing but forwards Joiner messages to facilitate secure commissioning.

OpenThread CLI

The **OpenThread CLI** is a command-line interface that provides configuration and management APIs for OpenThread.

It is used on OpenThread Border Routers (OTBR), Commissioners, and other Thread devices. The available commands depend on the device type. The `help` command prints all the supported commands.

References

- OpenThread Platforms (https://openthread.io/platforms)
- OpenThread Node Roles and Types (https://openthread.io/guides/thread-primer/node-roles-and-types)
- OpenThread Border Router (https://openthread.io/guides/border-router)
- Espressif OpenThread API (https://docs.espressif.com/projects/esp-idf/en/stable/esp32s2/api-guides/openthread.html)
- OpenThread CLI Overview (https://openthread.io/reference/cli)
- Espressif OpenThread CLI (https://github.com/espressif/esp-idf/tree/v5.4/examples/openthread/ot_cli)
- ESP Thread Border Router SDK (https://docs.espressif.com/projects/esp-thread-br/en/latest/)
- Thread Commissioning (https://www.threadgroup.org/Portals/0/documents/support/CommissioningWhitePaper_658_2.pdf)

- OpenThread commissioning (https://docs.nordicsemi.com/bundle/ncs-latest/page/nrf/protocols/thread/overview/commissioning.html)

Matter

Overview

Matter is an **IP-based** IoT connectivity standard that defines a common application layer for secure and interoperable communication over **Wi-Fi**, **Thread (Thread)**, and **Ethernet** networks.

References

- CSA - Specification Download Request (https://csa-iot.org/developer-resource/specifications-download-request/)

Data Model

Overview

Matter (Matter) devices follow a hierarchical **Data Model**.

At the top of this hierarchy is the **Device**, which represents a physical entity.

Each Device contains one or more **Nodes** - addressable entities that support the Matter protocol stack. Once commissioned, each Node has its own **Operational Node ID** and **Node Operational Credentials**. Nodes can have different roles:

- **Controllers** (e.g., Google Home app) manage other nodes.
- **Commissioners** are responsible for commissioning devices by assigning Fabric credentials to **Commissionee** devices.
- **OTA Providers** supply software updates to **OTA Requesters**.

Nodes can contain one or more individually addressable **Endpoints**, each representing a specific feature set.

Endpoints contain one or more **Clusters** (**"Clusters" in "Data Model"**), which enable independent utility or application functions. The *Endpoint 0* is reserved for the Utility Clusters. Each Cluster may include the following:

- **Attributes**, which represent a physical quantity or state.
- **Commands**, which are actions.
- **Events**, which represent notifications of state changes or significant conditions.

Example: a Smart Thermostat Device contains two Nodes:

- A Thermostat Node with a Thermostat Cluster (Attributes: Current Temperature, Commands: Set Heating Setpoint, Events: Temperature Changed).
- A Humidity Sensor Node with a Humidity Measurement Cluster (Attributes: Humidity Percentage, Commands: Read Humidity, Events: Humidity Changed).

Device Types

Thread Border Router (0x0091)

Clusters:

- Thread Network Diagnostics (0x0035)
- Thread Border Router Management (0x0452)
- Thread Network Directory (0x0453)

Temperature Sensor (0x0302)

Clusters:

- Identify (0x0003)
- Groups (0x0004)
- Temperature Measurement (0x0402)

Secondary Network Interface (0x0019)

Clusters:

- Network Commissioning (0x0031)
- Thread Network Diagnostics (0x0035)
- Wi-Fi Network Diagnostics (0x0036)
- Ethernet Network Diagnostics (0x0037)

Root Node (0x0016)

Conditions:

- CustomNetworkConfig: The node only supports out-of-band-configured networking (e.g., rich user interface, manufacturer-specific means, custom commissioning flows,

or future IP-compliant network technology not yet directly supported by the `NetworkCommissioning` cluster).

- ManagedAclAllowed: The node has at least one endpoint where some Device Type present on the endpoint has a Device Library element requirement table entry that sets this condition to true.

Clusters:

- Access Control (0x001F)
- Basic Information (0x0028)
- Localization Configuration (0x002B)
- Time Format Localization (0x002C)
- Unit Localization (0x002D)
- Power Source Configuration (0x002E)
- General Commissioning (0x0030)
- Network Commissioning (0x0031)
- Diagnostic Logs (0x0032)
- General Diagnostics (0x0033)
- Software Diagnostics (0x0034)
- Thread Network Diagnostics (0x0035)
- Wi-Fi Network Diagnostics (0x0036)
- Ethernet Network Diagnostics (0x0037)
- Time Synchronization (0x0038)
- Administrator Commissioning (0x003C)
- Node Operational Credentials (0x003E)
- Group Key Management (0x003F)

- ICD Management (0x0046)

Pressure Sensor (0x0305)

Clusters:

- Identify (0x0003)
- Groups (0x0004)
- Pressure Measurement (0x0403)

Network Infrastructure Manager (0x0090)

Clusters:

- Wi-Fi Network Management (0x0451)
- Thread Border Router Management (0x0452)
- Thread Network Directory (0x0453)

Clusters

This section describes the **Clusters** in the Matter data model used in this project.

Descriptor Cluster (0x001D)

Features:

- TagList: The TagList attribute is present.

Attributes:

- DeviceTypeList (0x0000): List of device types associated with this endpoint.
- ServerList (0x0001): List of server clusters implemented on this endpoint.
- ClientList (0x0002): List of client clusters implemented on this endpoint.
- PartsList (0x0003): List of endpoints that are part of this endpoint.

- TagList (0x0004): List of semantic tags (available if the **TagList** feature is supported).

Basic Information Cluster (0x0028)

Attributes:

- DataModelRevision (0x0000): Data model revision, default is "MS"
- VendorName (0x0001): Vendor name (max length: 32 characters)
- VendorID (0x0002): Vendor ID
- ProductName (0x0003): Product name (max length: 32 characters)
- ProductID (0x0004): Product ID
- NodeLabel (0x0005): Node label (max length: 32 characters)
- Location (0x0006): Location (2-character code)
- HardwareVersion (0x0007): Hardware version (default is 0)
- HardwareVersionString (0x0008): Hardware version as a string (1 to 64 characters)
- SoftwareVersion (0x0009): Software version
- SoftwareVersionString (0x000A): Software version as a string (1 to 64 characters)
- ManufacturingDate (0x000B): Manufacturing date (8 to 16 characters)
- PartNumber (0x000C): Part number (max length: 32 characters)
- ProductURL (0x000D): Product URL (max length: 256 characters)
- ProductLabel (0x000E): Product label (max length: 64 characters)
- SerialNumber (0x000F): Serial number (max length: 32 characters)
- LocalConfigDisabled (0x0010): Local configuration status (true/false)
- Reachable (0x0011): Device reachability status (true/false)
- UniqueID (0x0012): Unique ID (max length: 32 characters)
- CapabilityMinima (0x0013): Minimum capabilities (Case sessions and subscriptions)

- ProductAppearance (0x0014): Product appearance (Finish and PrimaryColor)
- SpecificationVersion (0x0015): Specification version
- MaxPathsPerInvoke (0x0016): Max paths per invocation (default is 1)

Events:

- StartUp (0x00): Triggered on startup with the SoftwareVersion field
- ShutDown (0x01): Triggered on shutdown
- Leave (0x02): Triggered when a device leaves, with a FabricIndex field (1 to 254)
- ReachableChanged (0x03): Triggered when the reachability status changes, with a ReachableNewValue field (true/false)

Commissioner Control Cluster (0x0751)

Attributes:

- SupportedDeviceCategories (0x0000): Supported Device Categories bitmap

Commands:

- RequestCommissioningApproval (0x00): Triggered by a request for commissioning approval from the server
  - RequestID (uint64)
  - VendorID (vendor-id)
  - ProductID (uint16)
  - Label (string, max length 64)
- CommissionNode (0x01): Command to commission a node, expects a response from the server
  - RequestID (uint64)
  - ResponseTimeoutSeconds (uint16, default 30, range 30 to 120)

- ReverseOpenCommissioningWindow (0x02): Response from the server after a commissioning request
  - CommissioningTimeout (uint16)
  - PAKEPasscodeVerifier (octstr)
  - Discriminator (uint16, max value 4095)
  - Iterations (uint32, range 1000 to 100000)
  - Salt (octstr, length between 16 to 32)

Events:

- CommissioningRequestResult (0x00): Event indicating the result of a commissioning request
  - RequestID (uint64)
  - ClientNodeID (node-id)
  - StatusCode (status)

Thread Network Diagnostics Cluster (0x0035)

Attributes:

- Channel (0x0000): Channel number
- RoutingRole (0x0001): Routing role of the node
- NetworkName (0x0002): Network name (max length 16)
- PanId (0x0003): PAN ID
- ExtendedPanId (0x0004): Extended PAN ID
- MeshLocalPrefix (0x0005): Mesh local prefix
- OverrunCount (0x0006): Overrun count (default 0)
- NeighborTable (0x0007): List of neighbor table entries

- RouteTable (0x0008): List of route table entries
- PartitionId (0x0009): Partition ID
- Weighting (0x000A): Weighting (max value 255)
- DataVersion (0x000B): Data version (max value 255)
- StableDataVersion (0x000C): Stable data version (max value 255)
- LeaderRouterId (0x000D): Leader router ID (max value 62)
- DetachedRoleCount (0x000E): Detached role count (default 0)
- ChildRoleCount (0x000F): Child role count (default 0)
- RouterRoleCount (0x0010): Router role count (default 0)
- LeaderRoleCount (0x0011): Leader role count (default 0)
- AttachAttemptCount (0x0012): Attach attempt count (default 0)
- PartitionIdChangeCount (0x0013): Partition ID change count (default 0)
- BetterPartitionAttachAttemptCount (0x0014): Better partition attach attempt count (default 0)
- ParentChangeCount (0x0015): Parent change count (default 0)
- TxTotalCount (0x0016): Total TX count (default 0)
- TxUnicastCount (0x0017): Unicast TX count (default 0)
- TxBroadcastCount (0x0018): Broadcast TX count (default 0)
- TxAckRequestedCount (0x0019): TX with ACK requested (default 0)
- TxAckedCount (0x001A): TX with ACK received (default 0)
- TxNoAckRequestedCount (0x001B): TX with no ACK requested (default 0)
- TxDataCount (0x001C): Data TX count (default 0)
- TxDataPollCount (0x001D): Data poll TX count (default 0)

- TxBeaconCount (0x001E): Beacon TX count (default 0)
- TxBeaconRequestCount (0x001F): Beacon request TX count (default 0)
- TxOtherCount (0x0020): Other TX count (default 0)
- TxRetryCount (0x0021): Retry TX count (default 0)
- TxDirectMaxRetryExpiryCount (0x0022): Direct max retry expiry TX count (default 0)
- TxIndirectMaxRetryExpiryCount (0x0023): Indirect max retry expiry TX count (default 0)
- TxErrCcaCount (0x0024): CCA error TX count (default 0)
- TxErrAbortCount (0x0025): Abort error TX count (default 0)
- TxErrBusyChannelCount (0x0026): Busy channel error TX count (default 0)
- RxTotalCount (0x0027): Total RX count (default 0)
- RxUnicastCount (0x0028): Unicast RX count (default 0)
- RxBroadcastCount (0x0029): Broadcast RX count (default 0)
- RxDataCount (0x002A): Data RX count (default 0)
- RxDataPollCount (0x002B): Data poll RX count (default 0)
- RxBeaconCount (0x002C): Beacon RX count (default 0)
- RxBeaconRequestCount (0x002D): Beacon request RX count (default 0)
- RxOtherCount (0x002E): Other RX count (default 0)
- RxAddressFilteredCount (0x002F): Address filtered RX count (default 0)
- RxDestAddrFilteredCount (0x0030): Destination address filtered RX count (default 0)
- RxDuplicatedCount (0x0031): Duplicated RX count (default 0)
- RxErrNoFrameCount (0x0032): No frame error RX count (default 0)
- RxErrUnknownNeighborCount (0x0033): Unknown neighbor error RX count (default 0)

- RxErrInvalidSrcAddrCount (0x0034): Invalid source address error RX count (default 0)
- RxErrSecCount (0x0035): Security error RX count (default 0)
- RxErrFcsCount (0x0036): FCS error RX count (default 0)
- RxErrOtherCount (0x0037): Other error RX count (default 0)
- ActiveTimestamp (0x0038): Active timestamp (default 0)
- PendingTimestamp (0x0039): Pending timestamp (default 0)
- Delay (0x003A): Delay (default 0)
- SecurityPolicy (0x003B): Security policy

Commands:

- ResetCounts (0x00): Resets error counts and statistics, invokes on the server

Events:

- ConnectionStatus (0x00): Event indicating connection status
- NetworkFaultChange (0x01): Event indicating network fault change

Wi-Fi Network Diagnostics Cluster (0x0036)

Attributes:

- BSSID (0x0000): Basic Service Set Identifier (BSSID)
- SecurityType (0x0001): Wi-Fi security type
- WiFiVersion (0x0002): Wi-Fi version
- ChannelNumber (0x0003): Channel number
- RSSI (0x0004): Received Signal Strength Indicator (RSSI)
- BeaconLostCount (0x0005): Number of beacon losses (default 0)
- BeaconRxCount (0x0006): Number of beacon packets received (default 0)

- PacketMulticastRxCount (0x0007): Multicast RX packet count (default 0)
- PacketMulticastTxCount (0x0008): Multicast TX packet count (default 0)
- PacketUnicastRxCount (0x0009): Unicast RX packet count (default 0)
- PacketUnicastTxCount (0x000A): Unicast TX packet count (default 0)
- CurrentMaxRate (0x000B): Current maximum rate (default 0)
- OverrunCount (0x000C): Overrun count (default 0)

Commands:

- ResetCounts (0x00): Resets the counts for errors and statistics

Events:

- Disconnection (0x00): Event triggered when a disconnection occurs
  - ReasonCode (uint16)
- AssociationFailure (0x01): Event triggered on association failure
  - AssociationFailureCause (AssociationFailureCauseEnum)
  - Status (uint16)
- ConnectionStatus (0x02): Event triggered when the connection status changes
  - ConnectionStatus (ConnectionStatusEnum)

Network Commissioning Cluster (0x0031)

Attributes:

- MaxNetworks (0x0000): Maximum number of networks (min 1)
- Networks (0x0001): List of network information
- ScanMaxTimeSeconds (0x0002): Max scan time for networks
- ConnectMaxTimeSeconds (0x0003): Max connection time

- InterfaceEnabled (0x0004): Whether the interface is enabled (default true)
- LastNetworkingStatus (0x0005): Last network commissioning status
- LastNetworkID (0x0006): Last network ID
- LastConnectErrorValue (0x0007): Last connection error value
- SupportedWiFiBands (0x0008): List of supported Wi-Fi bands
- SupportedThreadFeatures (0x0009): Thread features supported
- ThreadVersion (0x000A): Thread version

Commands:

- ScanNetworks (0x00): Scan for available networks
- ScanNetworksResponse (0x01): Response to scan networks request
- AddOrUpdateWiFiNetwork (0x02): Add or update a Wi-Fi network
- AddOrUpdateThreadNetwork (0x03): Add or update a Thread network
- RemoveNetwork (0x04): Remove a network
- NetworkConfigResponse (0x05): Response to network configuration commands
- ConnectNetwork (0x06): Connect to a network
- ConnectNetworkResponse (0x07): Response to connect to network
- ReorderNetwork (0x08): Reorder a network in the list

Events:

- Disconnection (0x00): Event indicating network disconnection
- AssociationFailure (0x01): Event indicating a failure in association
- ConnectionStatus (0x02): Event indicating the connection status

On/Off Cluster (0x0006)

Attributes:

- OnOff (0x0000): On/Off state (default false)
- GlobalSceneControl (0x4000): Global scene control (default true)
- OnTime (0x4001): On time duration (default 0)
- OffWaitTime (0x4002): Off wait time (default 0)
- StartUpOnOff (0x4003): Start-up behavior for OnOff state (default "MS")

Commands:

- Off (0x00): Command to turn off
- On (0x01): Command to turn on
- Toggle (0x02): Command to toggle the On/Off state
- OffWithEffect (0x40): Turn off with an effect
  - EffectIdentifier (EffectIdentifierEnum)
  - EffectVariant (enum8)
- OnWithRecallGlobalScene (0x41): Turn on with the recall of the global scene
- OnWithTimedOff (0x42): Turn on with timed off
  - OnOffControl (OnOffControlBitmap)
  - OnTime (uint16)
  - OffWaitTime (uint16)

Events:

- No events listed for this cluster

Features:

- Lighting (LT): Supports lighting applications
- DeadFrontBehavior (DF): Device has Dead Front behavior
- OffOnly (OFFONLY): Device supports the OffOnly feature

Data Types:

- DelayedAllOffEffectVariantEnum: Defines variants for delayed off effects
- DyingLightEffectVariantEnum: Defines variants for dying light effect
- EffectIdentifierEnum: Identifiers for effects like DelayedAllOff and DyingLight
- StartUpOnOffEnum: Defines start-up behavior for OnOff (Off, On, or Toggle)
- OnOffControlBitmap: Bitmap for controlling On/Off state behavior

Pressure Measurement Cluster (0x0403)

Attributes:

- MeasuredValue (0x0000): The current measured pressure value (int16)
- MinMeasuredValue (0x0001): Minimum measured pressure value (int16, max 32766)
- MaxMeasuredValue (0x0002): Maximum measured pressure value (int16)
- Tolerance (0x0003): Tolerance for the measurement (uint16, default 0, max 2048)
- ScaledValue (0x0010): Scaled pressure value (int16)
- MinScaledValue (0x0011): Minimum scaled value (int16, max 32766)
- MaxScaledValue (0x0012): Maximum scaled value (int16)
- ScaledTolerance (0x0013): Tolerance for the scaled value (uint16, default 0, max 2048)
- Scale (0x0014): Scaling factor for the measurement (int8, default 0, min -127)

Features:

- Extended (EXT): Indicates extended range and resolution support

Temperature Measurement Cluster (0x0402)

Attributes:

- MeasuredValue (0x0000): The current measured temperature value (type: temperature)
- MinMeasuredValue (0x0001): Minimum measured temperature value (type: temperature, range: -273.15°C to 32766°C)
- MaxMeasuredValue (0x0002): Maximum measured temperature value (type: temperature, greater than MinMeasuredValue)
- Tolerance (0x0003): Tolerance for the measurement (uint16, default 0, max 2048)

Thread Border Router Management Cluster (0x0452)

Attributes:

- BorderRouterName (0x0000): The name of the Border Router (1 to 63 characters)
- BorderAgentID (0x0001): The Border Agent ID (16 bytes)
- ThreadVersion (0x0002): The Thread version (default "MS")
- InterfaceEnabled (0x0003): Whether the interface is enabled (default false)
- ActiveDatasetTimestamp (0x0004): Timestamp for the active dataset (default 0)
- PendingDatasetTimestamp (0x0005): Timestamp for the pending dataset (default 0)

Commands:

- GetActiveDatasetRequest (0x00): Request to get the active dataset
- GetPendingDatasetRequest (0x01): Request to get the pending dataset
- DatasetResponse (0x02): Response containing the dataset (max length: 254 bytes)
- SetActiveDatasetRequest (0x03): Request to set the active dataset (max length: 254 bytes)

- SetPendingDatasetRequest (0x04): Request to set the pending dataset (max length: 254 bytes) with PAN change feature

Thread Network Directory Cluster (0x0453)

Attributes:

- PreferredExtendedPanID (0x0000): Preferred Extended PAN ID (8 bytes, nullable)
- ThreadNetworks (0x0001): List of Thread networks (max count defined by `ThreadNetworkTableSize`)
- ThreadNetworkTableSize (0x0002): Maximum number of Thread networks (default: 10)

Commands:

- AddNetwork (0x00): Add a network by providing the Operational Dataset (max length: 254 bytes)
- RemoveNetwork (0x01): Remove a network by specifying the Extended PAN ID (8 bytes)
- GetOperationalDataset (0x02): Get the Operational Dataset of a specified network by providing the Extended PAN ID (8 bytes)
- OperationalDatasetResponse (0x03): Response containing the Operational Dataset (max length: 254 bytes)

References

Device Types are defined in the **Device Library** rather than the main **Matter specification** document. Application Clusters are specified in the **Application Cluster Library**. These three documents can be requested from the **Connectivity Standards Alliance** (**CSA**) members' website. The **Matter Data Model** is also available on GitHub in the **Connected Home over IP** repository.

- Google Developer Centre - Matter - The Device Data Model (https://developers.home.google.com/matter/primer/device-data-model)

- GitHub - Matter Data Model (https://github.com/project-chip/connectedhomeip/tree/master/data_model/)
- Thread Border Router Device (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/device_types/ThreadBorderRouter.xml)
- Temperature Sensor Device (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/device_types/TemperatureSensor.xml)
- Secondary Network Interface Device (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/device_types/SecondaryNetworkInterface.xml)
- Root Node Device (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/device_types/RootNodeDeviceType.xml)
- Pressure Sensor Device (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/device_types/PressureSensor.xml)
- Network Infrastructure Manager Device (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/device_types/NetworkInfraManager.xml)
- Basic Information Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/BasicInformationCluster.xml)
- Commissioner Control Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/CommissionerControlCluster.xml)
- Thread Diagnostics Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/DiagnosticsThread.xml)

- Wi-Fi Diagnostics Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/DiagnosticsWiFi.xml)
- Network Commissioning Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/NetworkCommissioningCluster.xml)
- On/Off Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/OnOff.xml)
- Pressure Measurement Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/PressureMeasurement.xml)
- Temperature Measurement Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/TemperatureMeasurement.xml)
- Thread Border Router Management Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/ThreadBorderRouterManagement.xml)
- Thread Network Directory Cluster (https://github.com/project-chip/connectedhomeip/blob/master/data_model/1.4.1/clusters/ThreadNetworkDirectory.xml)

Matter Commissioning

Overview

Commissioners add a **Commissionee** device to a **Fabric**.

A **Fabric** is a private virtual network that connects Matter Devices and extends across Wi-Fi, Thread, and Ethernet physical networks. During the Matter commissioning process, controllers assign Fabric credentials to ensure secure integration.

Commissioning Process

Commissioning involves the following steps:

1. The Commissioner retrieves the Onboarding Payload ("Onboarding Payload" in "Matter Commissioning") and discovers ("Device Discovery" in "Matter Commissioning") the Commissionee device.
2. Passcode-Authenticated Session Establishment (PASE) is used to establish encryption keys, securing communication between the Commissioner and Commissionee. This process also sets up an attestation challenge for verifying the device's authenticity.
3. The Commissioner verifies the Commissionee's authenticity through Device Attestation, ensuring it is a certified Matter device.
4. The Commissioner configures the Commissionee by providing essential information including network configuration settings, such as Wi-Fi credentials (SSID and passphrase) or Thread network credentials.
5. The Commissionee joins the operational network, unless it is already connected. The Commissioner or Administrator identifies or discovers the device's IPv6 address to enable further communication.
6. Certificate-Authenticated Session Establishment (CASE) is used to derive long-term encryption keys, securing all future unicast communication between the Commissioner or Administrator and the Commissionee.
7. The Commissioning process completes with an encrypted message exchange, confirming successful onboarding using CASE-derived encryption keys on the

operational network.

Onboarding Payload

The **Commissionee** shares the **Onboarding Payload** with the **Commissioner** through a *QR Code*, *Manual Pairing Code*, or *NFC Tag*. It is composed of required and optional information. The following information may be included:

- **Version** specifies the payload format version, allowing future updates. It is 3 bits (0b000) for machine-readable formats and 1 bit (0b0) for Manual Pairing Codes.
- **Vendor ID** is a 16-bit identifier assigned by the Connectivity Standards Alliance (CSA) to uniquely identify a device manufacturer. It ensures that devices from different manufacturers can be distinguished within the Matter ecosystem.
- **Product ID** is a 16-bit identifier assigned by the manufacturer to differentiate between their products. It helps identify specific device models and is used in commissioning and attestation processes.
- **Custom Flow** is a 2-bit enumeration that specifies whether additional steps are required before commissioning. It informs the Commissioner if the device is ready for commissioning immediately, requires user interaction (such as pressing a button), or needs an external service interaction before being available for setup.
- **Discovery Capabilities Bitmask** is an 8-bit bitmask included in machine-readable formats. It indicates the discovery technologies (e.g., BLE, Wi-Fi, Thread) supported by the device, helping the Commissioner determine how to find and connect to it.
- **Discriminator** is a 12-bit identifier used to distinguish between multiple commissionable device advertisements. When a device enters commissioning mode, it broadcasts this value over BLE, Wi-Fi, or Thread, and it must match the value the device advertises. Each device should have a unique Discriminator to improve discovery and setup reliability. In machine-readable formats, the full 12-bit Discriminator is used, while in Manual Pairing Codes, only the upper 4 bits are included.
- **Passcode** is a 27-bit numeric value used to authenticate the device during commissioning, serving as proof of possession. It is encoded as an 8-digit decimal number ranging from 00000001 to 99999998, excluding invalid values. The Passcode

is also used as a shared secret to establish a secure channel between the Commissioner and the device for further onboarding steps.

- **TLV Data** is optional, variable-length information stored in machine-readable formats using the Tag-Length-Value encoding. This data provides additional commissioning details.

Device Discovery

Device Discovery is the process where a **Commissioner** identifies a **Commissionee** before onboarding it to a **Fabric**. Devices announce their presence through the following methods:

- **BLE** is used by devices without network credentials to advertise their presence. The Commissioner scans for advertisements containing the Discriminator, Vendor ID, and Product ID to identify the correct device.
- **Wi-Fi / Ethernet** is used by devices already connected to a network, announcing themselves via mDNS.
- **Thread** is used by Thread-enabled devices, which register with a Thread Border Router using Service Registration Protocol (SRP).
- **User-Directed Commissioning (UDC)** allows a device to actively search for Commissioners, letting the user select one for onboarding.

Access Control

All Matter Interaction Model operations require verification by the Access Control mechanism. Before a client can read, subscribe, write attributes, or invoke commands on a server, Access Control must confirm sufficient privileges. If not granted, the operation is denied (status 0x7E: Access Denied).

Matter enforces access levels for secure operations:

- **View** (1) – Can read and subscribe to all attributes and events, except the Access Control Cluster.
- Proxy (2) - *not yet supported in Matter SDK* – Can read and subscribe to all attributes and events when acting as a Proxy device.
- **Operate** (3) – Can perform the main function of the Node, plus all View privileges (except Access Control Cluster).
- **Manage** (4) – Can modify the Node's settings and configuration, plus all Operate privileges (except Access Control Cluster).
- **Administer** (5) – Can manage the Access Control Cluster, plus all Manage privileges.

The Administer privilege is initially granted to the commissioner over the PASE commissioning channel, allowing it to invoke commands during commissioning.

Administrators manage ACLs by reading and writing the fabric-scoped ACL attribute in the Access Control Cluster (always on endpoint 0). These ACLs control which Interaction Model operations are allowed or denied for fabric nodes via CASE and group messaging.

References

- GitHub - Matter - Access Control Guide (https://github.com/project-chip/connectedhomeip/blob/master/docs/guides/access-control-guide.md)

Espressif

Overview

Espressif is a semiconductor company that develops SoCs, microcontrollers, and development boards for IoT applications.

Hardware

Espressif's SoCs (System-on-Chips) enable wireless communication for IoT applications. Development boards integrate these SoCs with power circuits, USB interfaces, and breakout pins, making them easy to use for prototyping and automation projects.

The following development boards are used in this system:

- ESP32-H2-DevKitM-1 (https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32h2/esp32-h2-devkitm-1/index.html) is a development board with Thread and Bluetooth LE connectivity, designed for low-power applications.
- ESP32-C6-DevKitC-1 (https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32c6/esp32-c6-devkitc-1/index.html) is a development board with Wi-Fi 6, Bluetooth LE, and Thread connectivity.
- ESP32-DevKitM-1 (https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32/esp32-devkitm-1/index.html) is a development board with Wi-Fi and Bluetooth LE connectivity, designed for general-purpose applications.
- ESP Thread Border Router (https://docs.espressif.com/projects/esp-thread-br/en/latest/hardware_platforms.html) is a development board that acts as a Thread Border Router, enabling communication between Thread networks and Wi-Fi or Ethernet for external connectivity.

ESP-IDF Framework

ESP-IDF (**Espressif IoT Development Framework**) is Espressif's official IoT Development Framework for the ESP32, ESP32-S, ESP32-C and ESP32-H series of SoCs.

ESP Matter Solution

The **Espressif's SDK for Matter** is the official Matter (Matter) development framework for ESP32 series SoCs.

Wi-Fi-enabled ESP32, ESP32-C, and ESP32-S series can be used to build Matter Wi-Fi devices, with the ESP32-S series also supporting Ethernet connectivity via an external controller.

ESP32-H series SoCs, which support IEEE 802.15.4, are used for Matter-compatible Thread end devices.

ESP Thread Border Router Solution

The **Espressif Thread Border Router SDK** is a FreeRTOS-based solution built on the ESP-IDF ("ESP-IDF Framework" in "Espressif") and OpenThread (Thread) stack. It supports both Wi-Fi and Ethernet interfaces as the backbone link, combined with 802.15.4 SoCs for Thread communication.

The Wi-Fi-based Espressif Thread Border runs on two SoCs:

- The host Wi-Fi SoC, which runs OpenThread Border Router (ESP32, ESP32-S, or ESP32-C series SoC).
- The Radio Co-Processor (RCP), which enables the Border Router to access the 802.15.4 physical and MAC layers (ESP32-H series SoC).

Espressif also provides a single **ESP THREAD BR-ZIGBEE GW** board that integrates both the host SoC and the RCP into a single board.

Espressif also provides a **ESP Thread BR-Zigbee GW_SUB** daughter board for building an Ethernet-enabled Thread Border Router. It must be connected to a Wi-Fi-based ESP Thread Border Router.

ESP-Mesh-Lite

ESP-Mesh-Lite is a Wi-Fi Mesh networking solution based on the Wi-Fi protocol. It supports Espressif's Wi-Fi SoCs ("Hardware" in "Espressif"), including ESP32, ESP32-C, and ESP32-S.

In a traditional Wi-Fi network, all devices must connect directly to the router, limiting coverage and the number of connected devices based on the router's capacity. With **ESP-Mesh-Lite**, devices can connect to nearby nodes instead of relying solely on the router. This extends Wi-Fi coverage and allows more devices to join the network without router limitations.

References

- About Espressif (https://www.espressif.com/en/about)
- Espressif Development Boards (https://www.espressif.com/en/products/devkits/)
- ESP IoT Development Framework (https://www.espressif.com/en/products/sdks/esp-idf)
- ESP-Mesh-Lite (https://www.espressif.com/en/sdks/esp-mesh-lite)
- ESP-Mesh-Lite User Guide (https://github.com/espressif/esp-mesh-lite/blob/master/components/mesh_lite/User_Guide.md)

Development Environment

This chapter describes the development environment setup for building and deploying projects on ESP32 series SoCs.

Prerequisites:

- **Ubuntu Desktop** 24.04 LTS is used as the development environment for this guide.

The following software has to be installed:

- ESP-IDF framework ("ESP-IDF Framework" in "Espressif")
- ESP-Matter framework ("ESP Matter Solution" in "Espressif")

ESP-IDF Setup

Overview

This section demonstrates how to set up the ESP-IDF ("ESP-IDF Framework" in "Espressif") development environment for building and running applications on ESP32 SoCs.

This project uses ESP-IDF v5.3.2 (https://docs.espressif.com/projects/esp-idf/en/v5.3.2/), locked to commit fb25eb0 (https://github.com/espressif/esp-idf/commit/fb25eb02ebcf78a78b4c34a839238a4a56accec7).

Step 1: Install Prerequisites

The following packages are required for the ESP-IDF development environment:

```
sudo apt-get install git wget flex bison gperf python3 python3-pip
python3-venv cmake ninja-build ccache libffi-dev libssl-dev dfu-util
libusb-1.0-0
```

`idf.py` uses `#!/usr/bin/env python` in its shebang, which expects the `python` command to be available. On Ubuntu 20.04 and later, `python` is not included by default, as Python 2 has been deprecated. Check with:

```
which python
which python3
```

If `python` is missing, a symlink can be created to point to `python3`:

```
sudo ln -s /usr/bin/python3 /usr/bin/python
```

Step 2: Get ESP-IDF

```
mkdir -p ~/esp
cd ~/esp
git clone -b release/v5.3 --recursive https://github.com/espressif/esp-
```

```
idf.git esp-idf-5.3
cd ~/esp/esp-idf-5.3
git checkout fb25eb02ebcf78a78b4c34a839238a4a56accec7
```

Step 3: Set up the Tools

For newer versions of Ubuntu, the `install.sh` script may not work as expected because `

ESP-IDF provides a script `install.sh` that installs the required tools such as the compiler, debugger, Python packages, etc.:

```
./install.sh all
```

Step 4: Create get_idf Alias

ESP-IDF provides a script `export.sh` that sets environment variables so the tools are usable from the command line. The script sets IDF_PATH, updates PATH with ESP-IDF tools, verifies Python compatibility, and enables idf.py auto-completion.

Create an alias for executing export.sh by adding the following line to `~/.bashrc` file:

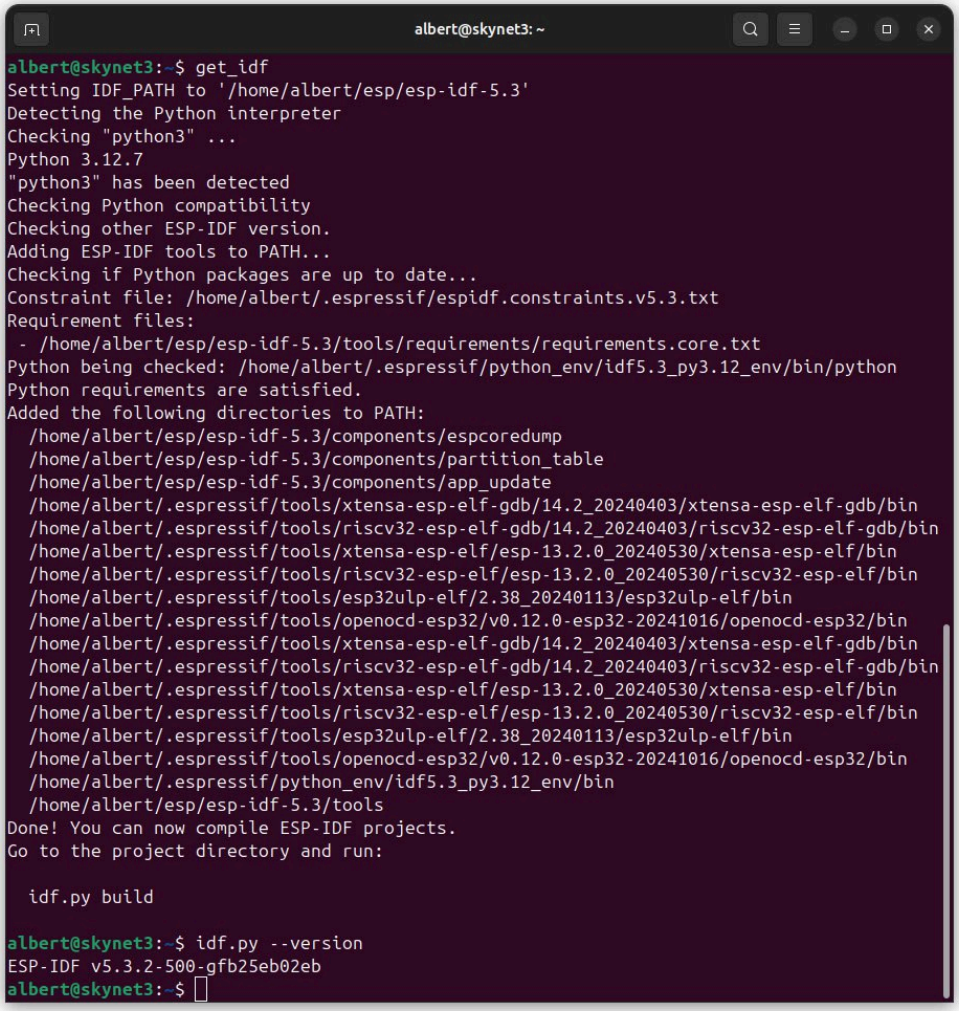
```
alias get_idf='. $HOME/esp/esp-idf-5.3/export.sh'
```

Step 5: Refresh Configuration

Restart your terminal session or run:

```
source ~/.bashrc
```

Now, running `get_idf` will set up or refresh the esp-idf environment in any terminal session.


albert@skynet3: ~
albert@skynet3:~$ get_idf
Setting IDF_PATH to '/home/albert/esp/esp-idf-5.3'
Detecting the Python interpreter
Checking "python3" ...
Python 3.12.7
"python3" has been detected
Checking Python compatibility
Checking other ESP-IDF version.
Adding ESP-IDF tools to PATH...
Checking if Python packages are up to date...
Constraint file: /home/albert/.espressif/espidf.constraints.v5.3.txt
Requirement files:
- /home/albert/esp/esp-idf-5.3/tools/requirements/requirements.core.txt
Python being checked: /home/albert/.espressif/python_env/idf5.3_py3.12_env/bin/python
Python requirements are satisfied.
Added the following directories to PATH:
/home/albert/esp/esp-idf-5.3/components/espcoredump
/home/albert/esp/esp-idf-5.3/components/partition_table
/home/albert/esp/esp-idf-5.3/components/app_update
/home/albert/.espressif/tools/xtensa-esp-elf-gdb/14.2_20240403/xtensa-esp-elf-gdb/bin
/home/albert/.espressif/tools/riscv32-esp-elf-gdb/14.2_20240403/riscv32-esp-elf-gdb/bin
/home/albert/.espressif/tools/xtensa-esp-elf/esp-13.2.0_20240530/xtensa-esp-elf/bin
/home/albert/.espressif/tools/riscv32-esp-elf/esp-13.2.0_20240530/riscv32-esp-elf/bin
/home/albert/.espressif/tools/esp32ulp-elf/2.38_20240113/esp32ulp-elf/bin
/home/albert/.espressif/tools/openocd-esp32/v0.12.0-esp32-20241016/openocd-esp32/bin
/home/albert/.espressif/tools/xtensa-esp-elf-gdb/14.2_20240403/xtensa-esp-elf-gdb/bin
/home/albert/.espressif/tools/riscv32-esp-elf-gdb/14.2_20240403/riscv32-esp-elf-gdb/bin
/home/albert/.espressif/tools/xtensa-esp-elf/esp-13.2.0_20240530/xtensa-esp-elf/bin
/home/albert/.espressif/tools/riscv32-esp-elf/esp-13.2.0_20240530/riscv32-esp-elf/bin
/home/albert/.espressif/tools/esp32ulp-elf/2.38_20240113/esp32ulp-elf/bin
/home/albert/.espressif/tools/openocd-esp32/v0.12.0-esp32-20241016/openocd-esp32/bin
/home/albert/.espressif/python_env/idf5.3_py3.12_env/bin
/home/albert/esp/esp-idf-5.3/tools
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:
idf.py build
albert@skynet3:~$ idf.py --version
ESP-IDF v5.3.2-500-gfb25eb02eb
albert@skynet3:~$

ESP-Matter Setup

This section demonstrates how to set up the Espressif's SDK for Matter ("ESP-IDF Framework" in "Espressif") for building Matter applications on ESP32 SoCs.

This project uses Espressif's SDK for Matter v1.4 (https://github.com/espressif/esp-matter/tree/release/v1.4), locked to commit 30af618 (https://github.com/espressif/esp-matter/commit/30af618a6e962623a0098ad6a33b468f33dc49c7).

Prerequisites:

- ESP-IDF (ESP-IDF Setup) development environment is set up.

Step 1: Install Prerequisites

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

Refer to the Matter Build Guide (https://github.com/espressif/connectedhomeip/blob/v1.3-branch/docs/guides/BUILDING.md) for more details.

Step 2: Clone ESP-Matter Repository

It includes esp-matter SDK and tools (e.g., CHIP-tool, CHIP-cert, ZAP).

```
cd ~/esp/
git clone -b release/v1.4 --recursive https://github.com/espressif/esp-
matter.git esp-matter-1.4
cd esp-matter-1.4
git checkout 30af618a6e962623a0098ad6a33b468f33dc49c7
```

Step 3: Bootstrap ESP-Matter

```
get_idf
~/esp/esp-matter-1.4/install.sh
```

Step 4: Create get_matter Alias

The script `~/esp/esp-matter/export.sh` configures the environment. Create an alias for executing it by adding the following line to `~/.bashrc` file:

```
alias get_matter='. $HOME/esp/esp-matter/export.sh'
```

Step 5: Refresh Configuration

Restart your terminal session or run:

```
source ~/.bashrc
```

Now, running `get_matter` will set up or refresh the ESP-Matter environment in any terminal session.

References

- Connected Home over IP v1.3 (https://github.com/espressif/connectedhomeip/tree/v1.3-branch)
- Connected Home IP Documentation (https://project-chip.github.io/connectedhomeip-doc/index.html)

ESP32 Project Workflow

This section describes the workflow for creating, building, and deploying a new project.

Prerequisites:

- ESP-IDF (ESP-IDF Setup) development environment is set up.
- ESP-Matter (ESP-Matter Setup) development environment is set up.
- ESP32 ("Hardware" in "Espressif") development board is available.
- CLion 2024.3.3 or later is installed.

Step 1: Set Up ESP-IDF and ESP-Matter Environment

Run the aliases, created during the development setup, to initialise the ESP-IDF and ESP-Matter environments in the current terminal session:

```
get_idf
get_matter
```

Step 2: Create a New Project

ESP-IDF provides the idf.py (https://docs.espressif.com/projects/esp-idf/en/v5.2.3/esp32/api-guides/tools/idf-py.html) command-line tool as a front-end for managing project builds, deployment, debugging, and other tasks, simplifying the workflow significantly. It integrates several essential tools, including CMake for project configuration, Ninja for building, and esptool.py for flashing the target device.

```
idf.py create-project <project name>
cd <project name>
```

Step 3: Set Target Device

The following command sets the target device:

```
idf.py set-target <chip_name>
```

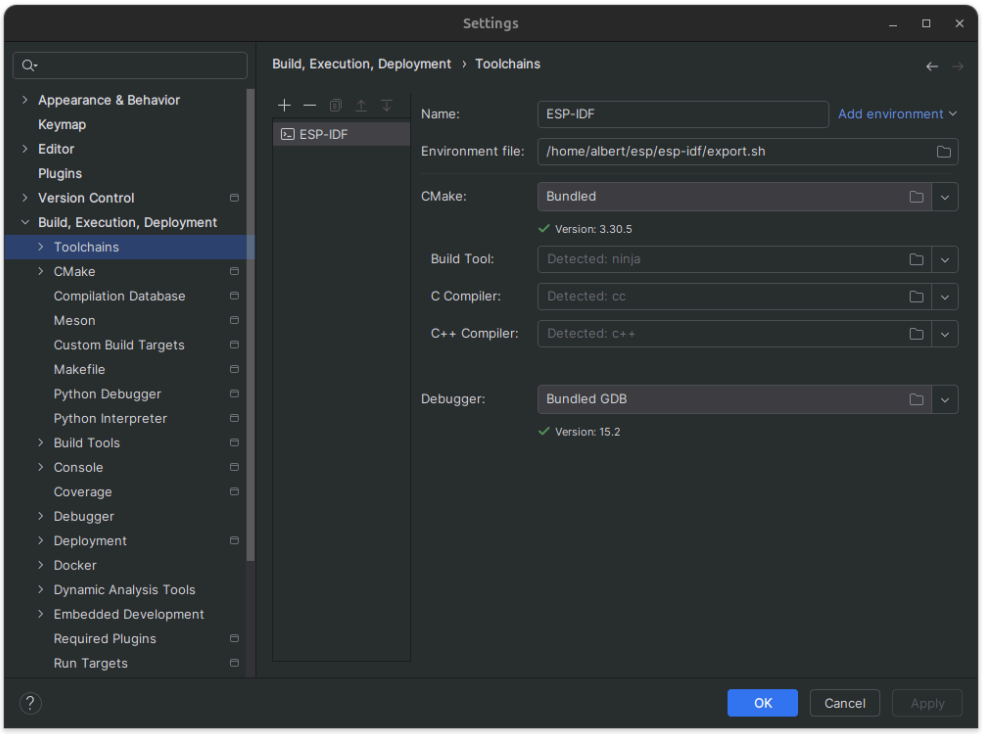
For example, to set the target device to ESP32-H2:

```
idf.py set-target esp32h2
```

It creates a new `sdkconfig` file in the root directory of the project. This configuration file can be modified via `idf.py menuconfig`.

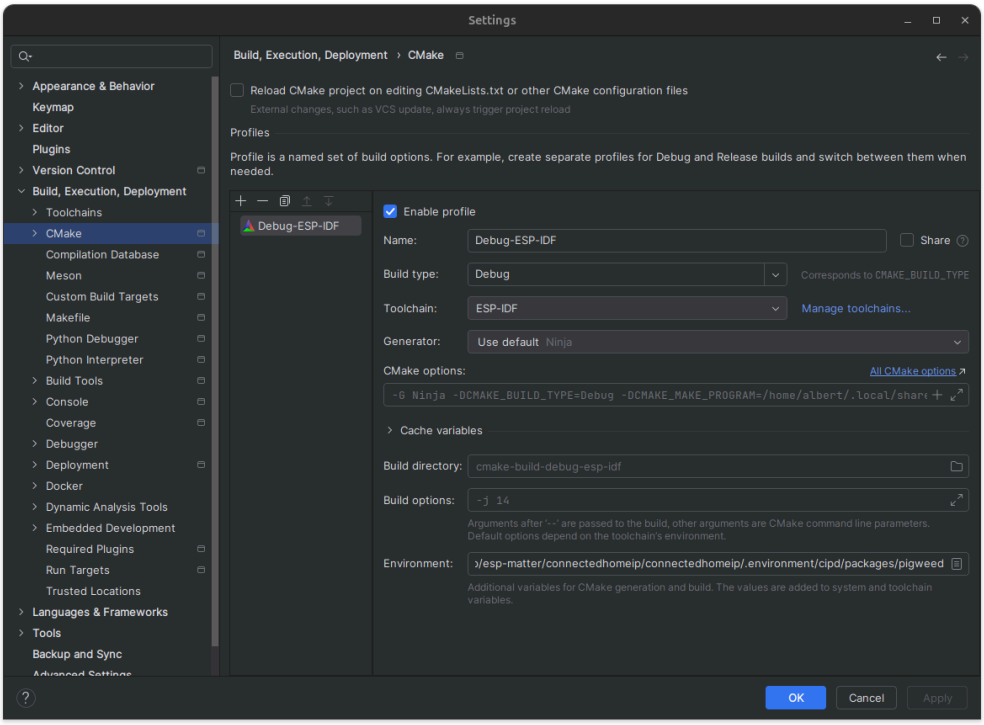
Step 4: Open the Project in CLion IDE

Create a toolchain named **ESP-IDF** with the environment file set to `esp-idf/script.sh`:

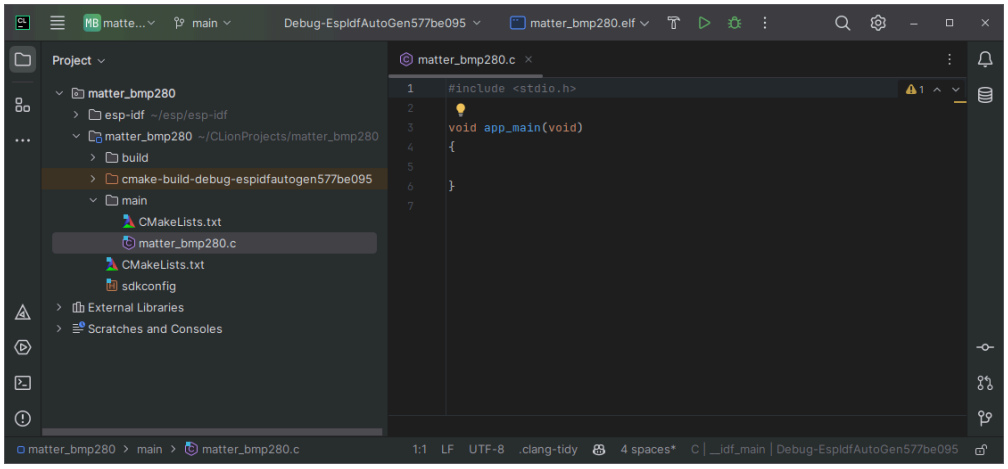


Set up a CMake profile with the following Matter environmental variables:

```
ESP_MATTER_PATH=~/esp/esp-matter-1.4;ZAP_INSTALL_PATH=~/esp/esp-matter-1.4/connectedhomeip/connectedhomeip/.environment/cipd/packages/zap
```



Structure of a new ESP-IDF project:



Step 5: Create Default Configuration

```
touch sdkconfig.defaults
```

Step 6: Update CMakeLists.txt

Update the `CMakeLists.txt` file to include the Matter SDK:

```
# For more information about build system see
# https://docs.espressif.com/projects/esp-idf/en/latest/api-
```

```
guides/build-system.html
# The following five lines of boilerplate have to be in your project's
# CMakeLists in this exact order for cmake to work correctly
cmake_minimum_required(VERSION 3.16)

# Set an error message if ESP_MATTER_PATH is not set
if(NOT DEFINED ENV{ESP_MATTER_PATH})
    message(FATAL_ERROR "Please set ESP_MATTER_PATH to the path of esp-
matter repo")
endif(NOT DEFINED ENV{ESP_MATTER_PATH})

# The set() commands should be placed after the cmake_minimum() line
but before the include() line.
set(PROJECT_VER "1.0")
set(PROJECT_VER_NUMBER 1)
set(ESP_MATTER_PATH $ENV{ESP_MATTER_PATH})
set(MATTER_SDK_PATH ${ESP_MATTER_PATH}/connectedhomeip/connectedhomeip)
set(ENV{PATH}
"$ENV{PATH}:${ESP_MATTER_PATH}/connectedhomeip/connectedhomeip/.environ
ment/cipd/packages/pigweed")

# Pulls in the rest of the CMake functionality to configure the
project, discover all the components, etc.
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
# Include common component dependencies and configurations from ESP-
Matter examples
include(${ESP_MATTER_PATH}/examples/common/cmake_common/components_incl
ude.cmake)

# Optional list of additional directories to search for components.
set(EXTRA_COMPONENT_DIRS
        "${MATTER_SDK_PATH}/config/esp32/components"
        "${ESP_MATTER_PATH}/components"
        ${extra_components_dirs_append})

# Declare the project
project(project-example)
```

```
# Set the C++ standard to C++17
idf_build_set_property(CXX_COMPILE_OPTIONS "-std=gnu++17;-Os;-
DCHIP_HAVE_CONFIG_H;-Wno-overloaded-virtual" APPEND)
idf_build_set_property(C_COMPILE_OPTIONS "-Os" APPEND)
# For RISCV chips, project_include.cmake sets -Wno-format, but does not
clear various
# flags that depend on -Wformat
idf_build_set_property(COMPILE_OPTIONS "-Wno-format-nonliteral;-Wno-
format-security" APPEND)
# Enable colored output in ninja builds
add_compile_options(-fdiagnostics-color=always -Wno-write-strings)
```

Step 7: Build the Project

```
idf.py build
```

Step 8: Determine Serial Port

Connect the ESP32 board to the computer and check under which serial port the board is visible. Serial ports have the following naming patterns: `/dev/tty`.

Step 9: Flash Project to Target

```
idf.py -p <PORT> flash
```

Step 10: Launch IDF Monitor

Use the monitor application and exit using `CTRL+]`:

```
idf.py -p <PORT> monitor
```

References

IDF Frontend - idf.py (https://docs.espressif.com/projects/esp-idf/en/stable/esp32h2/api-guides/tools/idf-py.html)

ESP Examples

ESP-IDF and ESP-Matter provide examples of applications that can be used as a reference. The examples are located in the following directories:

- `$IDF_PATH/examples`
- `$ESP_MATTER_PATH/examples`

The following examples were developed and can be used with the system:

- ESP OpenThread CLI (ESP OpenThread CLI)
- Thread Border Routers (Thread Border Routers)
  - ESP Basic Thread Border Router (ESP Basic Thread Border Router)
  - ESP Matter Thread Border Router (ESP Matter Thread Border Router)
- Matter Controller (Matter Controller)
  - CHIP-Tool (CHIP-Tool)
  - ESP Matter Controller (ESP Matter Controller)

ESP OpenThread CLI

This section describes how to build and run the OpenThread CLI ("OpenThread CLI" in "Thread") example project from the ESP-IDF ("ESP-IDF Framework" in "Espressif"). It can be used to test and debug the OpenThread stack on ESP32 series SoCs ("Hardware" in "Espressif").

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.
- ESP32 series development board is available. This example is tested on ESP32-H2-DevKitM-1 (https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32h2/esp32-h2-devkitm-1/index.html).

Build and Run

Step 1: Prepare the Environment

```
get_idf
cd $IDF_PATH/examples/openthread/ot_cli
```

Step 2: Set target

The following command sets the target device to ESP32-H2:

```
idf.py set-target esp32h2
```

Step 3 (Optional): Configure Device

The example can run with the default configuration. `idf.py menuconfig` can be used for customized settings:

- Enabling Joiner Role: `Component Config → OpenThread → Thread Core Features → Enable Joiner`

Step 4: Connect to Computer

Connect the ESP32-H2-DevKitM-1 to the computer using a USB cable.

Step 5: Build and Flash

```
idf.py build
idf.py -p <PORT_TO_ESP32_H2> flash monitor
```

Usage

Joining Thread Network

A Thread device can join the network using either a minimal dataset (providing only the Network Key) or a complete Active Operational Dataset.

Example 1: Joining with Minimal Provisioning (Network Key Only)

The device is initially provisioned with only the Network Key. After joining, the device synchronizes with the network and automatically receives the complete Active Operational Dataset.

```
dataset networkkey 00112233445566778899aabbccddeeff
dataset commit active
ifconfig up
thread start
```

Display the full Active Operational Dataset:

```
dataset active
```

Example 2: Joining with the Full Active Operational Dataset

In this method, all network parameters (such as PAN ID, channel, Mesh-Local Prefix, etc.) are provisioned upfront. The device is configured with the complete dataset before starting the Thread interface.

```
dataset set active
0e080000000000010000000300001835060004001fffe00208fe7bb701f5f1125d0708f
d75cbde7c6647bd0510b3914792d44f45b6c7d76eb9306eec94030f4f70656e54687265
61642d353833320102583204 10e35c581af5029b054fc904a24c2b27700c0402a0fff8
```

```
ifconfig up
thread start
```

Commissioning

Joiner device:

- Get the factory-assigned IEEE EUI-64 address (e.g., `744dbdfffe63f5c8`).

  ```
  eui64
  ```

- Bring the IPv6 interface up, enable the Thread Joiner role, and start the Thread protocol:

  ```
  ifconfig up
  joiner start J00MMM
  thread start
  ```

Commissioner device:

- Start the Commissioner role:

  ```
  commissioner start
  ```

- Add a joiner entry:

  ```
  commissioner joiner add <eui64|discerner> <pksd>
  ```

  Example:

  ```
  commissioner joiner add 744dbdfffe63f5c8 J00MMM or commissioner
  joiner add * J00MMM
  ```

- Display all Joiner entries in table format:

  ```
  commissioner joiner table
  ```

The Joiner device receives the Network Key when joining the network.

Retrieving Information

The command `router table` prints a list of routers in a table format. Each router is identified by its Extended MAC.

The command `extaddr` returns the Extended MAC address of a device.

The `joiner id` command returns the Joiner ID used for commissioning.

Sending UDP packets

Running UDP Server:

```
udpsockserver open
udpsockserver bind 12345
udpsockserver status
```

Sending a UDP packet and closing the server:

```
udpsockserver send fdf9:2548:ce39:efbb:9612:c4a0:477b:349a 12346 hello
udpsockserver close
```

Running UDP Client:

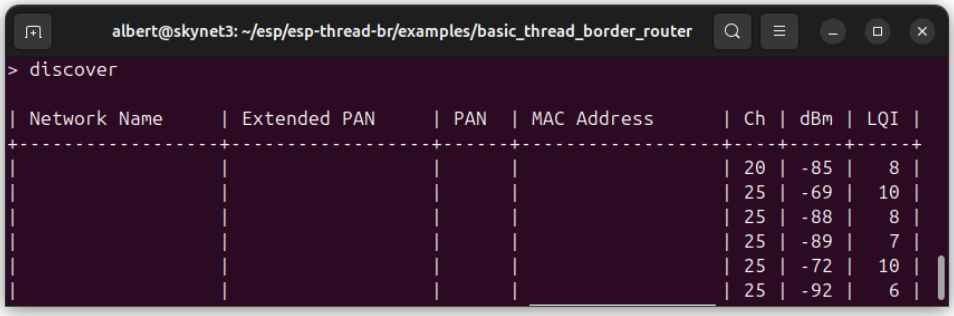
```
udpsockclient open # or udpsockclient open <port>, e.g., udpsockclient
open 12345
udpsockclient status
```

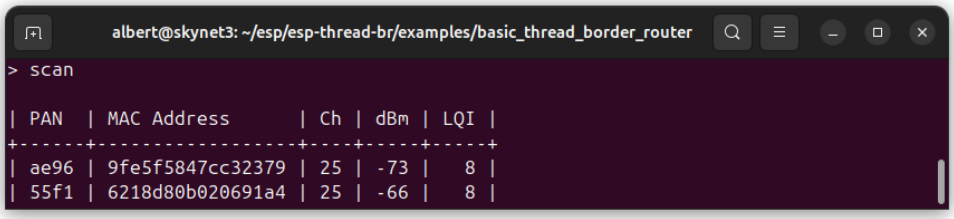
Sending a UDP packet and closing the client:

```
udpsockclient send fdf9:2548:ce39:efbb:79b9:4ac4:f686:8fc9 12346 hello
udpsockclient close
```

Scanning and Discovering

The `scan` command performs IEEE 802.15.4 scan to find nearby devices. The `discover` command performs an MLE Discovery operation to find Thread networks nearby.





References

- OpenThread CLI Command Reference (https://openthread.io/reference/cli/commands)
- Forming a Thread network on the Thread Border Router (https://openthread.io/codelabs/esp-openthread-hardware#3)
- How to set up a Command Line Interface on a thread device (https://mattercoder.com/codelabs/how-to-install-border-router-on-esp32/?index=..%2F..index#5)
- OpenThread CLI - Commissioning (https://github.com/openthread/ot-commissioner/tree/main/src/app/cli)
- OpenThread CLI - Operational Datasets (https://github.com/openthread/openthread/blob/main/src/cli/README_DATASET.md)
- ESP OT-CLI Example (https://github.com/espressif/esp-idf/tree/master/examples/openthread/ot_cli)
- Build and Run CLI device (https://docs.espressif.com/projects/esp-thread-br/en/latest/dev-guide/build_and_run.html#build-and-run-the-thread-cli-device)

- Set Up Service Registration Protocol (SRP) Server-Client Connectivity With OT CLI (https://openthread.io/reference/cli/concepts/srp)

ESP Thread Sniffer:

- https://docs.espressif.com/projects/esp-zigbee-sdk/en/latest/esp32h2/developing.html#sniffer-and-wireshark
- https://openthread.io/guides/pyspinel/sniffer
- https://github.com/espressif/esp-idf/blob/master/examples/openthread/ot_rcp/README.md

Thread Border Routers

Overview

Thread Border Routers ("Border Router" in "Thread") provide IP connectivity between the Thread Network of accessory devices (Accessory Devices) and adjacent external W-Fi network, such as a home LAN, building network, or the broader Internet.

This section walks through how to set up and run ESP Basic Thread Border Router (ESP Basic Thread Border Router) and ESP Matter Thread Border Router (ESP Matter Thread Border Router) examples on the ESP Thread Border Router ("ESP Thread Border Router Solution" in "Espressif") board.

ESP Basic Thread Border Router

Overview

The `ESP Basic Thread Border Router` example from ESP_IDF ("ESP-IDF Framework" in "Espressif") demonstrates how to build firmware for a basic Thread Border Router device running on Espressif's hardware.

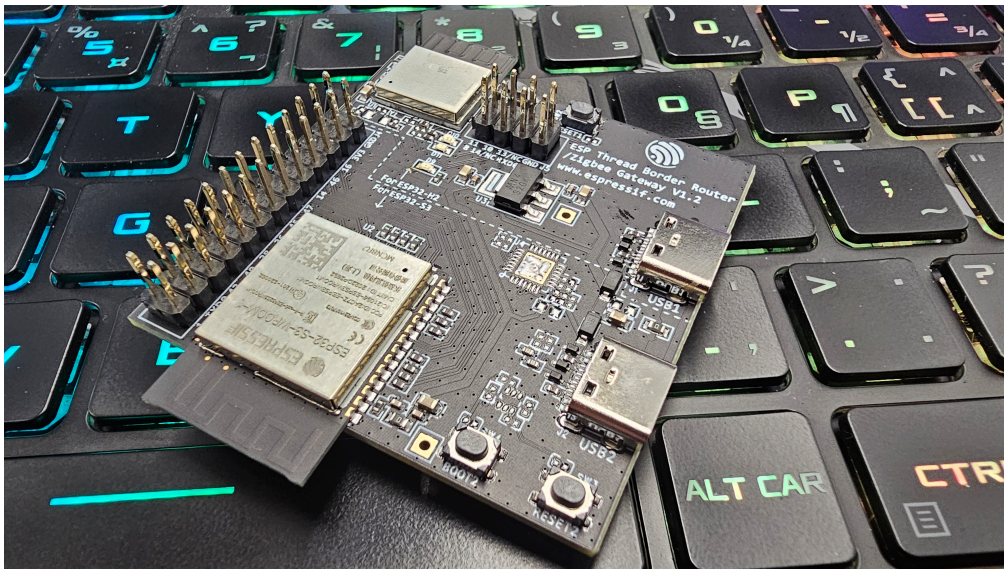
The firmware configures the OpenThread platform using default radio, host, and port settings. Optionally, it enables external coexistence and sets up mDNS (with the hostname "esp-ot-br"), along with OTA and web server support if enabled. Finally, it launches the Border Router.

Build and Run

This section demonstrates how to build and run the **ESP Basic Thread Border Router** example.

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.
- ESP THREAD BR-ZIGBEE GW ("Border Router" in "Thread") board



Step 1: Build the RCP Image

```
get_idf
cd $IDF_PATH/examples/openthread/ot_rcp/
idf.py set-target esp32h2   # Select the ESP32-H2. Skipping this step
would result in a build error.
idf.py build
```

The firmware does not need to be manually flashed onto the device. It will be integrated into the Border Router firmware and automatically installed onto the ESP32-H2 chip during the first boot-up. `idf.py menuconfig` can be used for customized settings.

Step 2: Clone Espressif Thread Border Router SDK

This project uses Espressif Thread Border Router SDK locked to commit cf3a09f (https://github.com/espressif/esp-thread-br/commit/cf3a09f5f44991a4e65b2d1c5113637e1d086b68).

```
cd ~/esp
git clone --recursive https://github.com/espressif/esp-thread-br.git
cd ~/esp/esp-thread-br/
git checkout cf3a09f5f44991a4e65b2d1c5113637e1d086b68
cd ~/esp/esp-thread-br/examples/basic_thread_border_router
```

Step 3: Configure the Device (Optional)

This section describes how to configure the device using `idf.py menuconfig`. This step is *optional* because the device can be configured using the OpenThread CLI ("OpenThread CLI" in "Thread") after flashing the firmware.

The default configuration in `sdkconfig.default` file is designed to work out of the box on the ESP Thread Border Router board with ESP32-S3 as the default SoC target. For other SoCs, the target must be configured using `idf.py set-target <chip_name>`.

The command below opens the configuration menu:

```
idf.py menuconfig
```

Enable **automatic start mode** and the **Web GUI**:

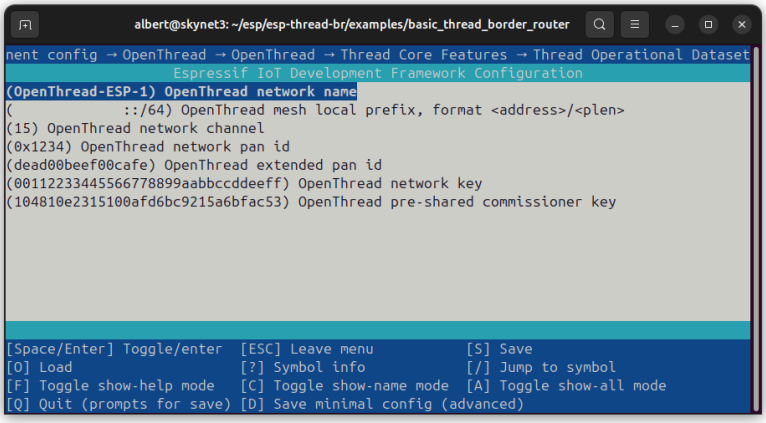
- ESP Thread Border Router Example → Enable the automatic start mode in Thread Border

In **automatic start mode**, the device first attempts to use the Wi-Fi SSID and password stored in **NVS** (Non-Volatile Storage). If no Wi-Fi credentials are found in NVS, it uses **EXAMPLE_WIFI_SSID** and **EXAMPLE_WIFI_PASSWORD**, retrieved from the following configuration options:

- Example Connection Configuration → WiFi SSID
- Example Connection Configuration → WiFi Password

Additionally, **Thread dataset** can be configured:

Component config → OpenThread → Thread Core Features → Thread Operational Dataset



Step 4: Connect the ESP Thread Border Router Board

Use **USB2 (ESP32-S3)** on the ESP Thread Border Router Board to connect the board to the computer. Only the ESP32-S3 (main SoC) port needs to be connected. The main SoC automatically programs the Thread co-processor.

Step 5: Build and Flash

```
idf.py build
idf.py -p <PORT> flash monitor
```

OpenThread CLI

This section demonstrates how to use the OpenThread CLI ("OpenThread CLI" in "Thread") on the ESP Thread Border Router.

OpenThread CLI is enabled in the ESP OpenThread component (https://github.com/espressif/esp-idf/blob/master/components/openthread/Kconfig) by default. It can also be enabled/disabled using the `idf.py menuconfig` command:

- `Component config → OpenThread → Thread Console → Enable OpenThread Command-Line Interface`

The `ESP Thread Border Router` project extends the standard `OpenThread CLI` by including the OpenThread Extension Commands (https://github.com/espressif/esp-thread-br/tree/main/components/esp_ot_cli_extension#wifi) component with additional commands demonstrated below, such as Wi-Fi management and IP address printing.

Wi-Fi Management

Connect the ESP Thread Border Router to a Wi-Fi network:

```
wifi connect -s <SSID> -p <PASSWORD>
```

Disconnect the ESP Thread Border Router from the Wi-Fi network:

```
wifi disconnect
```

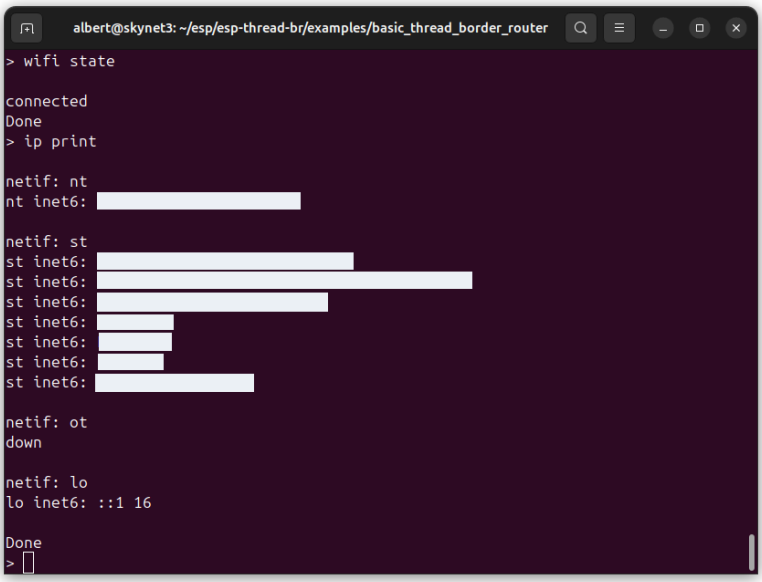
Print the current Wi-Fi network state:

```
wifi state
```

IP Addresses and Network Interfaces

Print all the IP address on each interface of lwIP of the Thread Border Router:

```
ip print
```

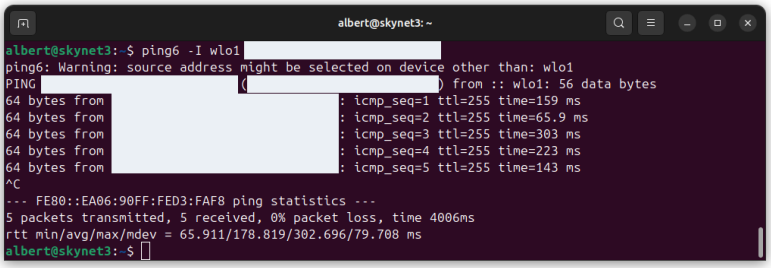


The output lists network interfaces (netif):

- **nt**
- **ot** (OpenThread Network Interface (https://github.com/espressif/esp-idf/blob/release/v5.3/components/openthread/src/esp_openthread_lwip_netif.c#L136)),
- **st** (Wi-Fi Station (STA) interface (https://github.com/espressif/esp-idf/blob/release/v5.3/components/esp_netif/lwip/netif/wlanif.c#L223)), this includes the **Global Unicast Address** (**GUA**), which falls within the 2000::/3 range (starts with 2xxx, 3xxx). This address is globally routable and can be used for external access if the Wi-Fi router's firewall allows incoming connections to this address.
- **lo** (Loopback interface (https://github.com/espressif/esp-lwip/blob/2.1.3-esp/src/core/netif.c#L151)).

If the Thread Border Router is connected to a Wi-Fi network, we can ping it from another device. The following commands should be run on a Linux machine to find the name of the Wi-Fi network interfaces and then use it to ping the Thread Border Router:

```
ip addr
ping6 -I <INTERFACE> <DEVICE_IP>
ping6 -I wlo1 fe80::aaaa:0000:0000:0000
ping6 -I wlo1 2604::aaaa:0000:0000:0000
```



Forming Thread network

The following commands form a Thread network ("Network" in "Thread"):

1. *Optional*. Delete any previous settings stored on non-volatile memory, and then trigger a platform reset:

```
factoryreset
```

2. Initialize a new dataset:

```
dataset init new
```

3. Commit the new Operational Dataset buffer to Active Operational Dataset:

```
dataset commit active
```

4. Enable IPv6 communication:

```
ifconfig up
```

5. Enable Thread interface

```
thread start
```

6. Show the Active Operational Dataset. The optional `-x` argument prints it as hex-encoded TLVs:

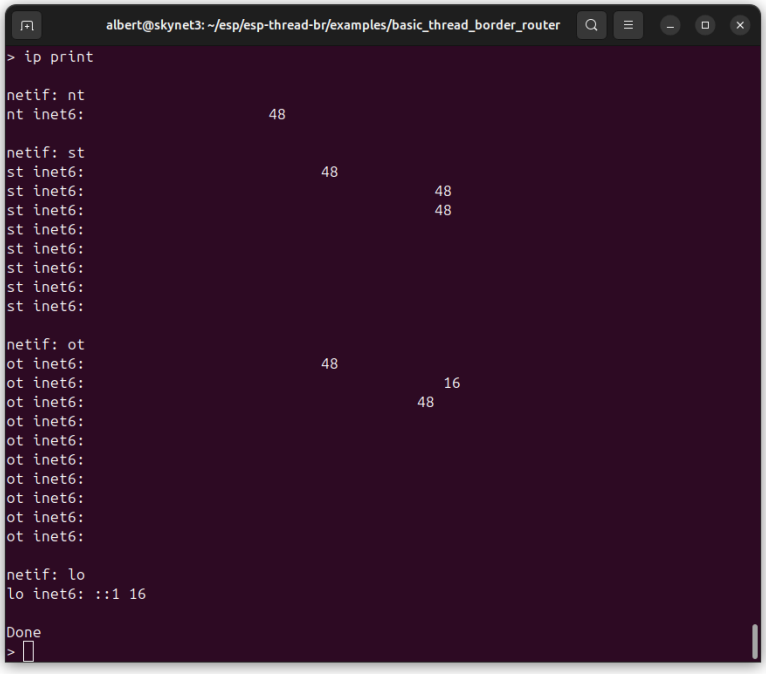
```
dataset active -x
```

```
> dataset active

Active Timestamp: 1
Channel: 23
Wake-up Channel: 21
Channel Mask: 0x07fff800
Ext PAN ID: cee2281bfad406a1
Mesh Local Prefix:
Network Key: aba4f103d160b985d67d746026503b96
Network Name: OpenThread-91dd
PAN ID: 0x91dd
PSKc: dadef003379811fafe86f995dcdfd127
Security Policy: 672 onrc 0
Done
> dataset active -x

0e080000000000010000000300000174a0300001535060004
11fafe86f995dcdfd1270c0402a0f7f8
Done
```

The ip print ("IP Addresses and Network Interfaces" in "ESP Basic Thread Border Router") command now shows the Thread interface with the IPv6 address:



Managing Border Routing Manager

The Border Routing Manager starts automatically after enabling the Thread interface. The `br` command can be used to manage the Border Routing Manager.

Print the current state of Border Routing Manager:

```
br state
```

Initializes the Border Routing Manager:

```
br init <infrastructure-network-index> <is-running>
```

Enables the Border Routing Manager:

```
br enable
```

Information Commands

The `platform` command prints the current platform. The `version` and `rcp version` print the OpenThread version and the radio version respectively.

RESTful API and Web GUI

The ESP Thread Border Router provides a RESTful API and Web GUI for managing the Thread Border Router. It is disabled by default and can be enabled using `idf.py menuconfig`:

- `ESP Thread Border Router Example → Enable the web server in Thread Border Router`

Web GUI Access

The Web GUI can be accessed using the IP address of the Thread Border Router:

```
http://<DEVICE_IP>/index.html
```

The Web GUI can be accessed using the IPv4 address assigned by the Wi-Fi router or the device's IPv6 Global Unicast Address (see `ip print` in extension-commands). IPv4 is only accessible within the local network unless port forwarding is configured on the router. IPv6 can be accessed externally if a firewall rule is added on the router.

RESTful API Client Library Generation

The API is documented using the OpenAPI specification in the openapi.yaml (https://github.com/espressif/esp-thread-br/blob/main/components/esp_ot_br_server/src/openapi.yaml) file. OpenAPI Generator (https://github.com/OpenAPITools/openapi-generator) can be used to generate a client library.

The following commands should be run on a Linux machine with installed Docker and Node.js to generate a TypeScript client library for the ESP-Thread Border Router server:

```
docker run --rm -v "${PWD}:/local" openapitools/openapi-generator-cli
generate -i /local/openapi.yaml -g typescript-fetch -o
/local/out/typescript-fetch
sudo chown -R $USER:$USER out/
npm install --safe-dev typescript
```

References

- Espressif Thread Border Router SDK (https://github.com/espressif/esp-thread-br)
- OpenThread - ESP Thread Border Router (https://mattercoder.com/codelabs/how-to-install-border-router-on-esp32/?index=..%2F..index#0)
- Espressif - Build and Run ESP Thread Border Router (https://docs.espressif.com/projects/esp-thread-br/en/latest/dev-guide/build_and_run.html)
- OpenThread - Build a Thread Network with the ESP32H2 and ESP Thread Border Router Board (https://openthread.io/codelabs/esp-openthread-hardware)
- How to Install Border Router on ESP32-DevKit and ESP32-H2 (https://mattercoder.com/codelabs/how-to-install-border-router-on-esp32/?index=..%2F..index#0)
- Web GUI (https://docs.espressif.com/projects/esp-thread-br/en/latest/codelab/web-gui.html)

- Global Unicast Address (https://docs.oracle.com/cd/E19120-01/open.solaris/819-3000/ipv6-overview-130/index.html)

ESP Matter Thread Border Router

Overview

The `ESP Matter Thread Border Router` example from Espressif's SDK for Matter ("ESP Matter Solution" in "Espressif") demonstrates how to build firmware for a Matter-compliant Thread Border Router device running on Espressif's hardware.

The firmware creates a Matter node and configures a Thread Border Router endpoint. If Thread support is enabled, it configures the OpenThread platform with default radio, host, and port settings. Finally, it starts the Matter stack with an event callback to handle network events and initializes the diagnostic console if enabled.

When the ESP32 connects to Wi-Fi and gets an IP, it sets up Wi-Fi as the backbone for OpenThread and starts the Thread Border Router to enable communication between Thread and Wi-Fi. A static variable stops it from running more than once.

Build and Run

This section demonstrates how to build and run the `ESP Matter Thread Border Router` example from Espressif's SDK for Matter ("ESP Matter Solution" in "Espressif").

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.
- ESP-Matter development environment (ESP-Matter Setup) is set up.
- ESP THREAD BR-ZIGBEE GW ("Border Router" in "Thread") board

Step 1: Build and Configure the RCP Image

The build process is similar to the Basic Thread Border Router example ("Build and Run" in "ESP Basic Thread Border Router"). The following steps from the Basic Thread Border Router example are required if not already completed:

1. Follow Step 1: Build the RCP Image ("Step 1: Build the RCP Image" in "ESP Basic Thread Border Router").

2. Follow optional Step 3: Configure the Device ("Step 3: Configure the Device (Optional)" in "ESP Basic Thread Border Router").

Step 2: Set Up Environment

Set up the environment and navigate to the `ESP Matter Thread Border Router` example directory:

```
get_idf
get_matter
cd $ESP_MATTER_PATH/examples/thread_border_router
```

Step 3: Set target to ESP32-S3

```
idf.py set-target esp32s3
```

Step 4: Connect the ESP Thread Border Router Board

Follow Step 4: Connect the ESP Thread Border Router Board ("Step 4: Connect the ESP Thread Border Router Board" in "ESP Basic Thread Border Router") from the Basic Thread Border Router example.

Step 5: Build and Flash

```
idf.py build
idf.py -p <PORT> flash monitor
```

Matter CLI

This section demonstrates how to use the Matter CLI on the ESP Matter Thread Border Router.

The device automatically starts BLE advertising when powered on. The following command can be used to check the state:

```
matter ble adv state
```

The following command prints the Matter configuration, including the PIN code and

Discriminator, needed for commissioning:

```
matter config
```

CHIP-Tool

The Thread Border Router management cluster allows provisioning of the Thread interface using a Matter commissioner.

Commissioning to Wi-Fi over BLE

Refer to Commissioning to Wi-Fi over BLE using CHIP Tool ("Commissioning to Wi-Fi over BLE" in "CHIP-Tool").

Joining a Thread Network

After commissioning, a fail-safe timer must be armed:

```
./chip-tool generalcommissioning arm-fail-safe <TIMEOUT_IN_SEC> 1
<NODE_ID> 0
./chip-tool generalcommissioning arm-fail-safe 120 1 0x1122 0
```

Provision the Border Router using an active dataset in HEX TLV format (the same format used for commissioning a Matter over Thread ("Commissioning to Thread over BLE" in "CHIP-Tool") device via the `ble-thread` command). The **Thread Border Router Management Cluster** should be used:

```
./chip-tool threadborderroutermanagement set-active-dataset-request
hex:<ACTIVE_DATASET> <NODE_ID> 1
```

If the active dataset command succeeds, complete the commissioning process by disarming the fail-safe timer and committing the configuration to non-volatile storage:

```
./chip-tool generalcommissioning commissioning-complete <NODE_ID> 1
```

Verify the configuration by retrieving the active dataset:

```
./chip-tool threadborderroutermanagement get-active-dataset-request
```

```
<NODE_ID> 1
```

The response includes a `DatasetResponse` containing the active dataset in hex-encoded format.

References

- ESP-Matter Thread Border Router (https://github.com/espressif/esp-matter/tree/main/examples/thread_border_router)

Matter Controller

This chapter describes the development of Matter Controllers.

CHIP-Tool (CHIP-Tool) and ESP Matter Controller examples (ESP Matter Controller) firmware are used to demonstrate the Matter Controller functionality.

The Orchestrator (Orchestrator) device is based on the ESP Matter Controller example.

CHIP-Tool

Overview

CHIP Tool is a Matter Controllers implementation that allows to commission a Matter (Matter) device into a network and to communicate with it. It is used for testing and debugging Matter devices.

CHIP Tool was built and installed on a **Raspberry Pi 4 Model B** (4GB RAM and 32GB SD card) running *Ubuntu Server for Raspberry Pi 24.04.1 LTS*. *Raspberry Pi Imager v1.8.5* was used to flash the Ubuntu image onto the SD card.

Matter v1.4.0.0 (https://github.com/project-chip/connectedhomeip/releases/tag/v1.4.0.0) SDK was used to build the CHIP-Tool application.

Raspberry Pi Setup

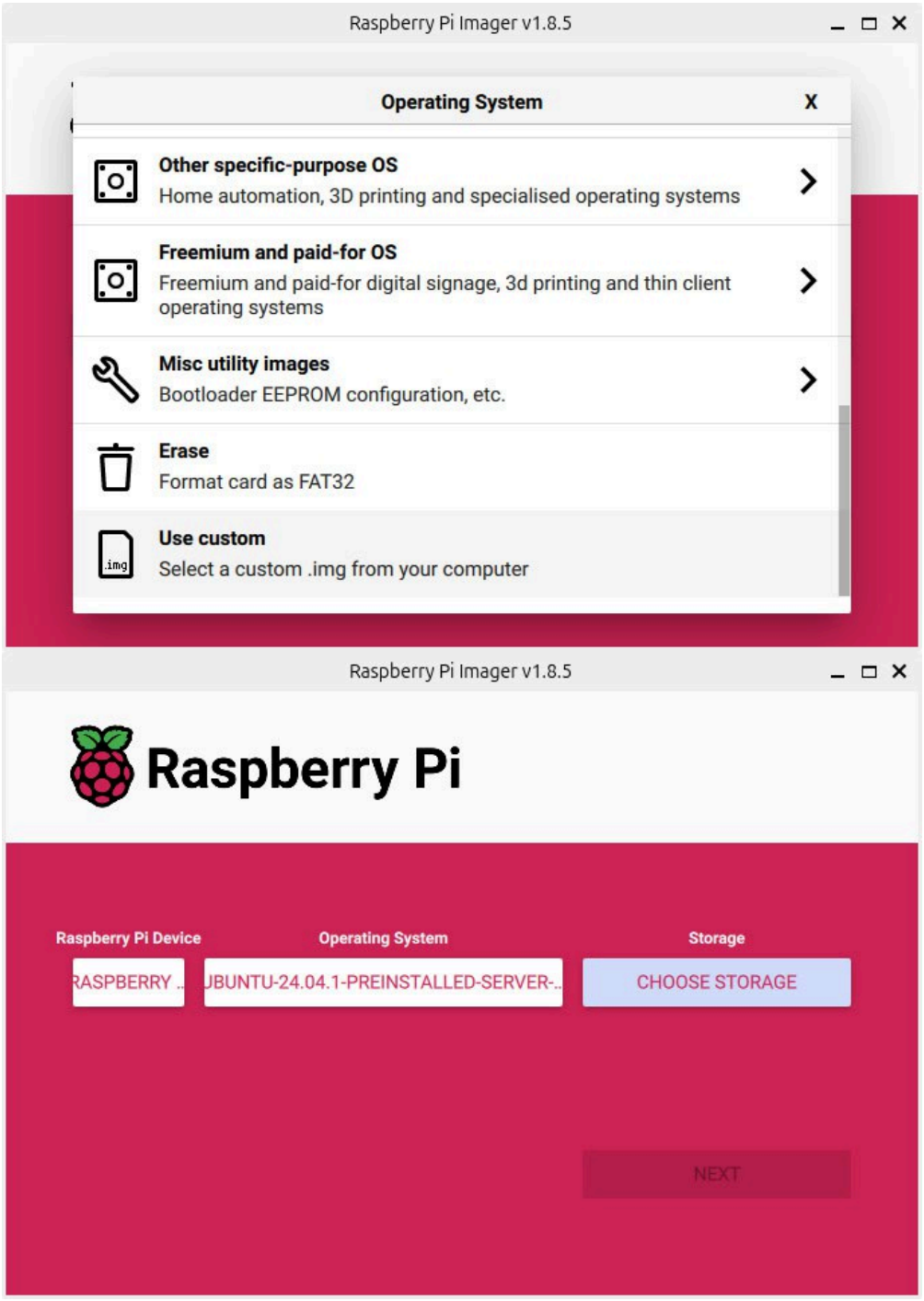
This section outlines the steps to set up a Raspberry Pi as a Matter controller using CHIP-Tool. The process is time-consuming.

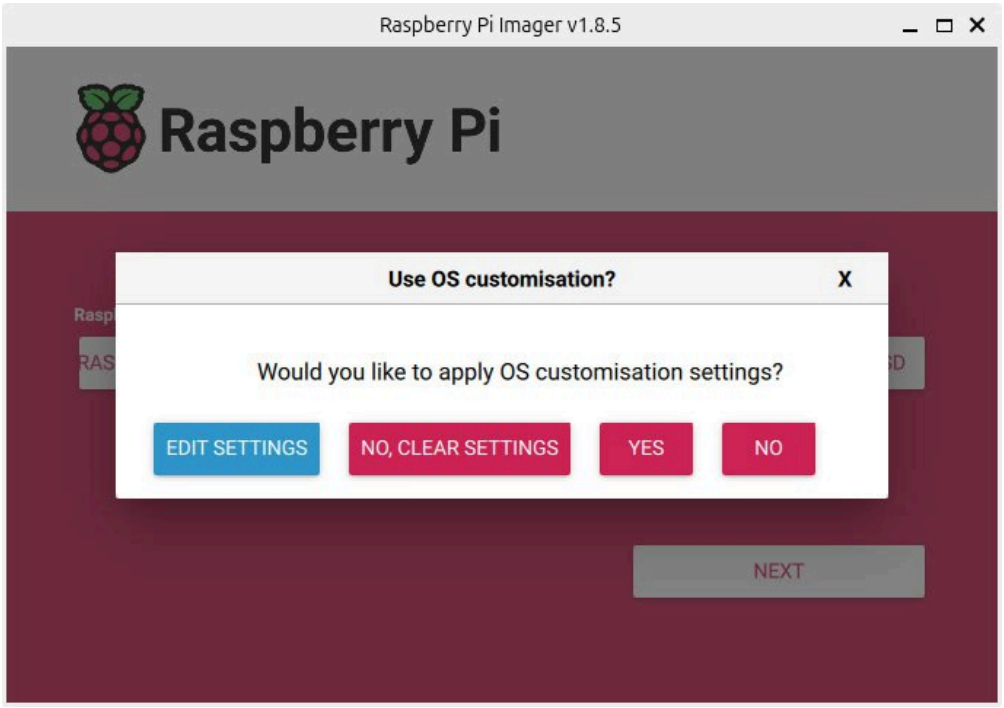
Step 1. Ubuntu installation

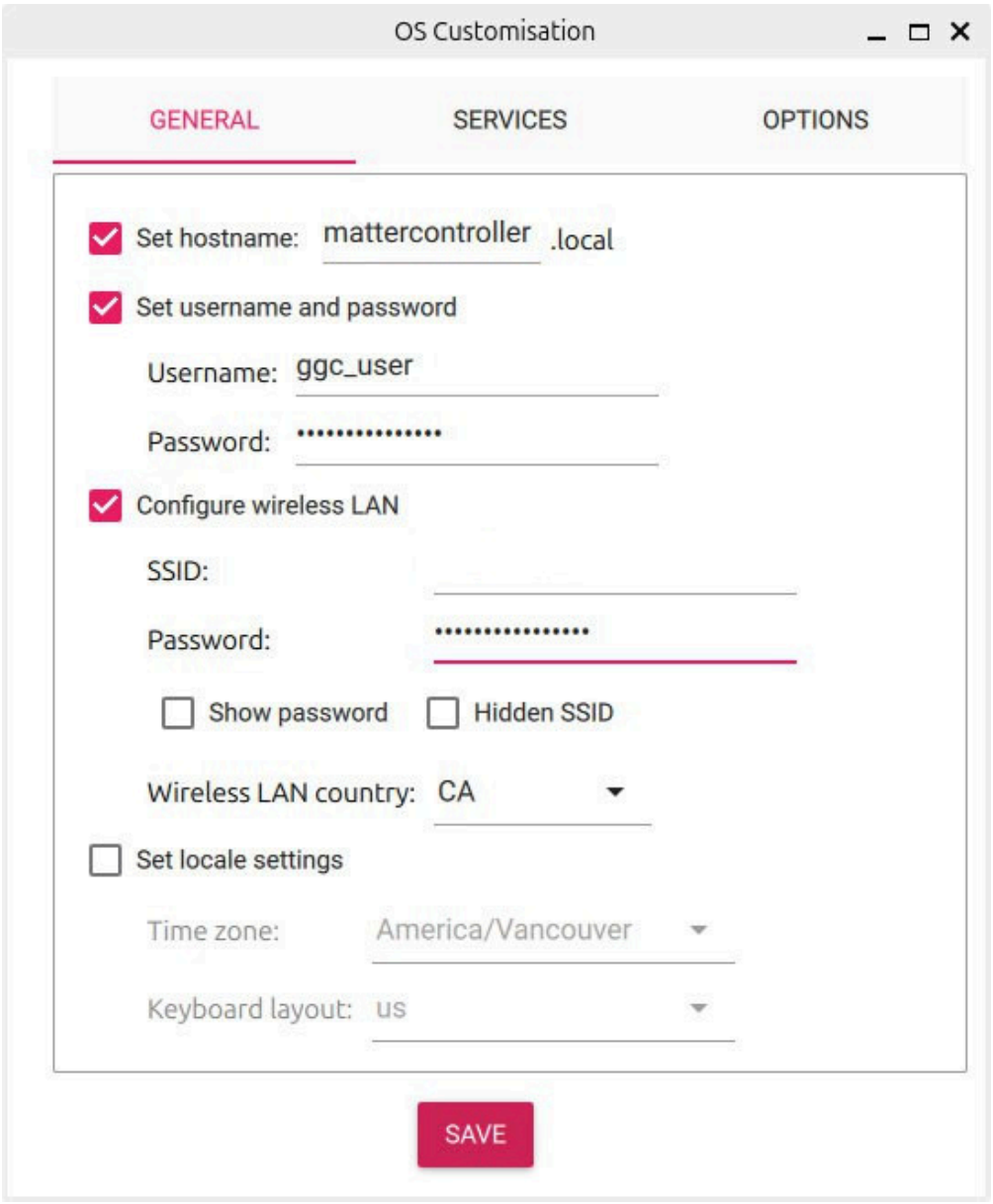
1. Install the OS
    1. Download the Ubuntu Server image from the Ubuntu website (https://ubuntu.com/download/raspberry-pi).
    2. Download and install Raspberry Pi Imager on Ubuntu Desktop:

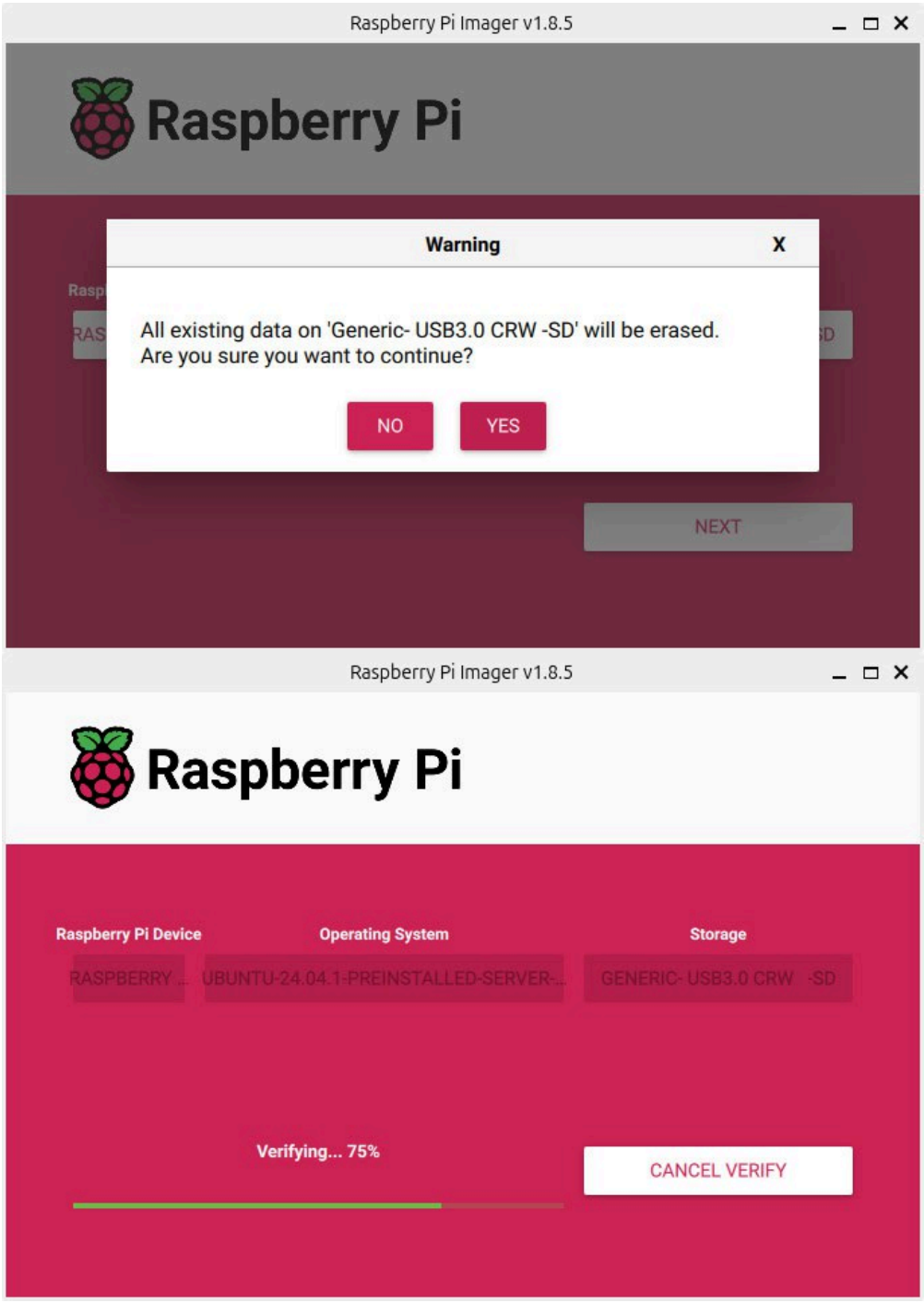
        ```
        sudo apt update
        sudo apt install rpi-imager
        ```

    3. Insert the SD card into card reader and connect it to the computer.
    4. Select the Ubuntu image and the SD card in Raspberry Pi Imager.
    5. Edit Settings (Wi-Fi and SSH credentials and enable SSH) and flash the image onto the SD card.


Raspberry Pi Imager v1.8.5
Operating System
X
Other specific-purpose OS
Home automation, 3D printing and specialised operating systems
Freemium and paid-for OS
Freemium and paid-for digital signage, 3d printing and thin client operating systems
Misc utility images
Bootloader EEPROM configuration, etc.
Erase
Format card as FAT32
Use custom
Select a custom .img from your computer
img
Raspberry Pi Imager v1.8.5
Raspberry Pi
Raspberry Pi Device
Operating System
Storage
RASPBERRY ..
UBUNTU-24.04.1-PREINSTALLED-SERVER-..
CHOOSE STORAGE
NEXT


Raspberry Pi Imager v1.8.5
Raspberry Pi
Use OS customisation?
X
Would you like to apply OS customisation settings?
EDIT SETTINGS
NO, CLEAR SETTINGS
YES
NO
NEXT

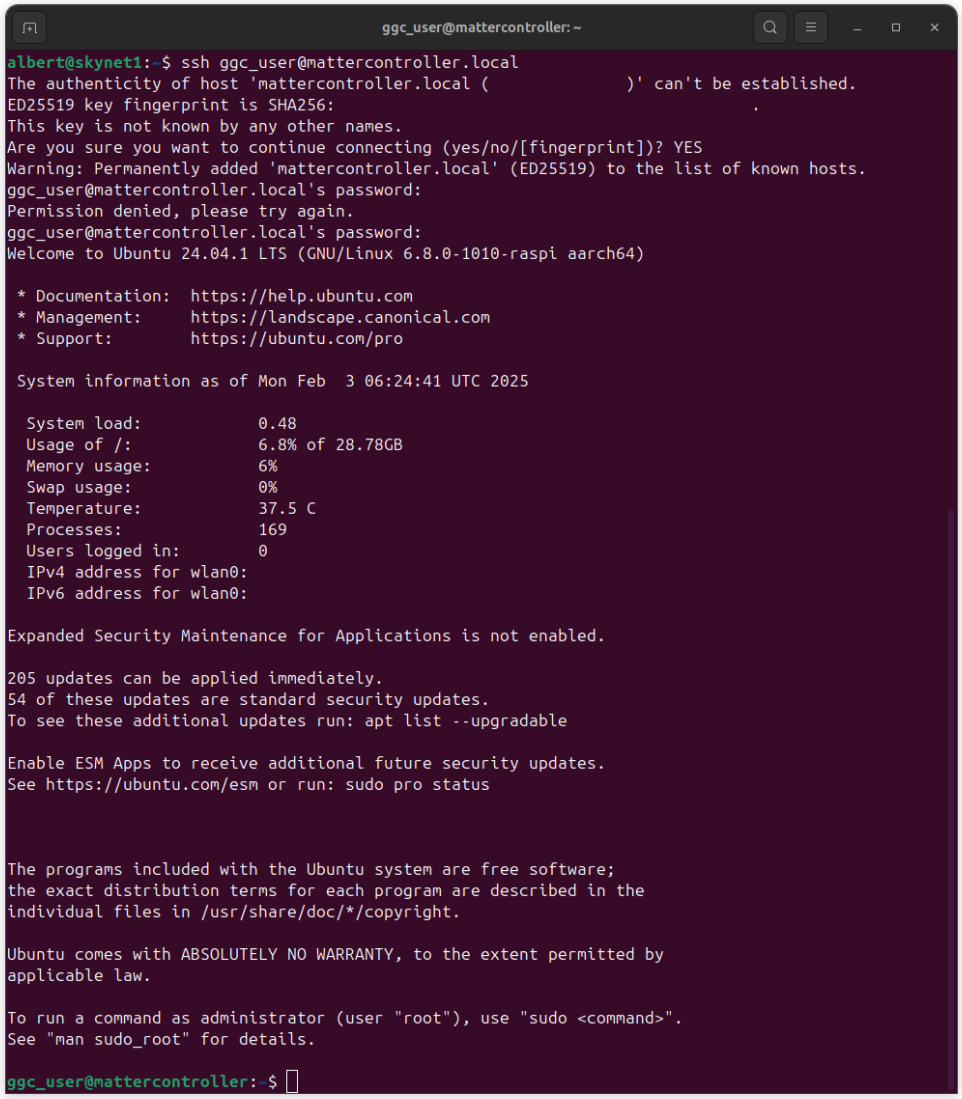

OS Customisation
GENERAL
SERVICES
OPTIONS
Set hostname:
mattercontroller
.local
Set username and password
Username:
ggc_user
Password:
Configure wireless LAN
SSID:
Password:
Show password
Hidden SSID
Wireless LAN country:
CA
Set locale settings
Time zone:
America/Vancouver
Keyboard layout:
us
SAVE



Step 2. Connecting to Raspberry Pi

Insert the SD card into the Raspberry Pi and power it on. It will automatically connect to Wi-Fi using the credentials provided during the installation process. It can take a few minutes for the Raspberry Pi to boot up. Once it is ready, connect to it using SSH with the credentials provided during the installation process.

```
ssh ggc_user@mattercontroller.local
```



Step 3. Installing Pre-requisites

1. Update the system and install Raspberry Pi-specific dependencies (https://github.com/project-chip/connectedhomeip/blob/master/docs/guides/BUILDING.md#installing-prerequisites-on-raspberry-pi-4):

```
sudo apt update
sudo apt upgrade
sudo apt-get install git gcc g++ pkg-config libssl-dev libdbus-1-dev
```

```
\
     libglib2.0-dev libavahi-client-dev ninja-build python3-venv
python3-dev \
     python3-pip unzip libgirepository1.0-dev libcairo2-dev
libreadline-dev \
     default-jre
sudo apt-get install bluez avahi-utils
sudo reboot
ssh ggc_user@mattercontroller.local
```

2. Edit the `bluetooth.service` unit:

```
sudo systemctl edit bluetooth.service
```

   Add the following content:

```
[Service]
ExecStart=
ExecStart=/usr/sbin/bluetoothd -E -P battery
Restart=always
RestartSec=3
```

3. Enable and Start the Bluetooth service:

```
sudo systemctl start bluetooth.service
```

4. Edit the dbus-fi.w1.wpa_supplicant1.service file:

```
sudo nano /etc/systemd/system/dbus-fi.w1.wpa_supplicant1.service
```

   Replace the line #11 with:

```
ExecStart=/usr/sbin/wpa_supplicant -u -s -O /run/wpa_supplicant
```

5. Edit the wpa_supplicant.conf file:

```
sudo nano /etc/wpa_supplicant/wpa_supplicant.conf
```

Add the following content:

```
ctrl_interface=DIR=/run/wpa_supplicant
update_config=1
```

6. Reboot the Raspberry Pi:

```
sudo reboot
ssh ggc_user@mattercontroller.local
```

Step 4. Building CHIP Tool

The application will require installation of Matter SDK on Raspberry Pi:

1. Clone the Matter SDK repository:

```
mkdir matter
cd ~/matter
git clone https://github.com/project-chip/connectedhomeip.git
cd connectedhomeip/
git checkout tags/v1.4.0.0
```

2. Prepare Matter SDK:

```
./scripts/checkout_submodules.py --shallow --platform  linux
source ~/matter/connectedhomeip/scripts/activate.sh
```

3. Build CHIP Tool using build_examples.py:

   The `targets` command retrieves supported build targets listed below:

   ```
   ./scripts/build/build_examples.py targets
   ```

   - **Ameba:**
     - `amebad-{all-clusters, all-clusters-minimal, light, light-switch, pigweed}`

- **ASR:** `asr-{asr582x, asr595x, asr550x}-{all-clusters, all-clusters-minimal, lighting, light-switch, lock, bridge, temperature-measurement, thermostat, ota-requestor, dishwasher, refrigerator}`
  - Optional flags: `[-ota] [-shell] [-no_logging] [-factory] [-rotating_id] [-rio]`
- **Android:** `android-{arm, arm64, x86, x64, androidstudio-arm, androidstudio-arm64, androidstudio-x86, androidstudio-x64}-{chip-tool, chip-test, tv-server, tv-casting-app, java-matter-controller, kotlin-matter-controller, virtual-device-app}`
  - Optional flags: `[-no-debug]`
- **Bouffalolab:** `bouffalolab-{bl602dk, bl704ldk, bl706dk, bl602-night-light, bl706-night-light, bl602-iot-matter-v1, xt-zb6-devkit}-light`
  - Optional flags: `[-ethernet] [-wifi] [-thread] [-easyflash] [-littlefs] [-shell] [-mfd] [-rotating_device_id] [-rpc] [-cdc] [-mot] [-resetcnt] [-memmonitor] [-115200] [-fp]`
- **Texas Instruments (TI):**
  - `cc32xx-{lock, air-purifier}`
  - `ti-cc13x4_26x4-{lighting, lock, pump, pump-controller}`
  - Optional flags: `[-mtd] [-ftd]`
- **Cypress (Infineon):** `cyw30739-{cyw30739b2_p5_evk_01, cyw30739b2_p5_evk_02, cyw30739b2_p5_evk_03, cyw930739m2evb_01, cyw930739m2evb_02}-{light, light-switch, lock, thermostat}`
- **Silicon Labs (EFR32):** `efr32-{brd2704b, brd4316a, brd4317a, brd4318a, brd4319a, brd4186a, brd4187a, brd2601b, brd4187c, brd4186c, brd2703a, brd4338a, brd2605a}-{window-covering, switch, unit-test, light, lock, thermostat, pump, air-quality-sensor-app}`
  - Optional flags: `[-rpc] [-with-ota-requestor] [-icd] [-low-power] [-shell] [-no-logging] [-openthread-mtd] [-heap-monitoring] [-no-openthread-cli] [-show-qr-code] [-wifi] [-rs9116] [-wf200] [-siwx917] [-ipv4] [-additional-data-advertising] [-use-ot-lib] [-use-ot-coap-lib] [-no-version] [-skip-rps-generation]`

- **Espressif (ESP32):** `esp32-{m5stack, c3devkit, devkitc, qemu}-{all-clusters, all-clusters-minimal, energy-management, ota-provider, ota-requestor, shell, light, lock, bridge, temperature-measurement, tests}`
  - Optional flags: `[-rpc] [-ipv6only] [-tracing] [-data-model-disabled] [-data-model-enabled]`
- **General:**
  - `genio-lighting-app`
- **Linux:**
  - `linux-fake-tests`
  - Optional flags: `[-mbedtls] [-boringssl] [-asan] [-tsan] [-ubsan] [-libfuzzer] [-ossfuzz] [-pw-fuzztest] [-coverage] [-dmalloc] [-clang] linux-arm64-{rpc-console, all-clusters, all-clusters-minimal, chip-tool, thermostat, java-matter-controller, kotlin-matter-controller, minmdns, light, light-data-model-no-unique-id, lock, shell, ota-provider, ota-requestor, simulated-app1, simulated-app2, python-bindings, tv-app, tv-casting-app, bridge, fabric-admin, fabric-bridge, tests, chip-cert, address-resolve-tool, contact-sensor, dishwasher, microwave-oven, refrigerator, rvc, air-purifier, lit-icd, air-quality-sensor, network-manager, energy-management}`
  - Optional flags: `[-nodeps] [-nlfaultinject] [-platform-mdns] [-minmdns-verbose] [-libnl] [-same-event-loop] [-no-interactive] [-ipv6only] [-no-ble] [-no-wifi] [-no-thread] [-no-shell] [-mbedtls] [-boringssl] [-asan] [-tsan] [-ubsan] [-libfuzzer] [-ossfuzz] [-pw-fuzztest] [-coverage] [-dmalloc] [-clang] [-test] [-rpc] [-with-ui] [-evse-test-event] [-enable-dnssd-tests] [-disable-dnssd-tests] [-chip-casting-simplified] [-data-model-check] [-data-model-disabled] [-data-model-enabled] [-check-failure-die]`
  - `linux-arm64-efr32-test-runner`
  - Optional flags: `[-clang]`

The `linux-arm64-chip-tool` and `linux-arm64-all-clusters` targets were selected from the Linux section, with the `-ipv6only` and `-platform-mdns` flags:

```
./scripts/build/build_examples.py --target linux-arm64-chip-tool-
ipv6only-platform-mdns gen
cd ~/matter/connectedhomeip/out/linux-arm64-chip-tool-ipv6only-
platform-mdns
ninja -j 1 & disown
```

Building the CHIP-Tool is a time-consuming process. To prevent SSH session timeouts, ninja was executed in the background. The job count was limited to 1 (`-j 1`) to reduce memory usage. `tail -f nohup.out` can be used to monitor the output.

4. Move the built CHIP Tool:

```
mv ~/matter/connectedhomeip/out/linux-arm64-chip-tool-ipv6only-
platform-mdns/chip-tool ~/matter/chip-tool
rm -rf ~/matter/connectedhomeip/out/linux-arm64-chip-tool-ipv6only-
platform-mdns
```

Step 5. Testing

Running CHIP Tool:

```
cd ~/matter
./chip-tool
```

Commissioning

This section describes the Matter commissioning (Matter Commissioning) using CHIP Tool.

The following command should be run on the Commissionee to print the static configuration that includes the **PIN code** and **Discriminator** (*0xf00* is *3840* in decimal):

```
matter config
```

Commissioning to Wi-Fi over BLE

A Commissionee joins an existing IP network over Bluetooth LE before being commissioned into a Matter network.

The following CHIP-Tool command starts commissioning onto a **Wi-Fi** network over BLE:

```
./chip-tool pairing ble-wifi 0x1122 SSID WIFIPASS 20202021 3840
./chip-tool pairing ble-wifi <NODE_ID_TO_ASSIGN> <SSID> <PASSWORD>
<PIN> <DISCRIMINATOR>
```

The parameters are defined as follows:

- `<NODE_ID_TO_ASSIGN>` is the node ID assigned to the device being commissioned, which can be a decimal or hexadecimal number prefixed with '0x'.
- `<SSID>` is the Wi-Fi SSID, provided as a plain string or in hexadecimal format as 'hex:XXXXXXXX', where each byte of the SSID is represented as two-digit hexadecimal numbers.
- `<PASSWORD>` is the Wi-Fi password, given as a plain string or hex data.
- `<PIN>` is the PIN code for authentication.
- `<DISCRIMINATOR>` is the discriminator value.

CHIP-Tool starts the commissioning process, which includes the following steps:

1. **Initialization**
    - Initializes storage and loads key-value store (KVS) configurations.
    - Detects network interfaces and identifies the WiFi interface.
    - Sets up UDP transport manager and BLE transport layers.
2. **Fabric and Node Setup**
    - Loads the fabric table, which contains information about existing secure networks.
    - Creates a new fabric with a unique Fabric ID and Node ID.
3. **BLE Scanning and Connection**
    - Scans for nearby BLE devices.
    - Identifies the correct device using a discriminator match.

- Establishes a BLE connection with the device.

4. **Secure Session Establishment**
    - Performs a PASE (Password Authenticated Session Establishment) handshake over BLE.
    - Exchanges secure messages to establish an encrypted session.
    - Marks the session as "Active" upon success.
5. **Commissioning Process**
    - Reads the device’s attributes and capabilities.
    - Arms a fail-safe mechanism to prevent accidental changes.
    - Configures the device based on regional regulatory requirements.
6. **Certificate Exchange**
    - The device provides PAI (Product Attestation Intermediate) and DAC (Device Attestation Certificate).
    - The system verifies the certificates and device authenticity.
7. **Attestation and Verification**
    - Validates the device’s attestation data to ensure a trusted identity.
    - Performs a revocation check to confirm the device’s legitimacy.
8. **Operational Certificate Signing**
    - The device generates a CSR (Certificate Signing Request).
    - The system issues a NOC (Node Operational Certificate) to authenticate the device in the network.
9. **Finalizing the Pairing**
    - Sends the root certificate to the device.
    - Marks the pairing process as "Success," making the device a trusted member of the

Matter network.

Commissioning to Thread over BLE

The following CHIP-Tool command initiates commissioning onto a **Thread** network over BLE:

```
./chip-tool pairing ble-thread 0x1122 20202021 3840
./chip-tool pairing ble-thread <NODE_ID_TO_ASSIGN> hex:
<OPERATIONAL_DATASET> <PIN> <DISCRIMINATOR>
```

The parameters are defined as follows:

- `<NODE_ID_TO_ASSIGN>` is the node ID assigned to the device being commissioned, which can be a decimal number or a hexadecimal number prefixed with '0x'.
- `<OPERATIONAL_DATASET>` is the Thread Operational Dataset (Thread), which contains the network credentials needed to join a Thread network. It is provided in hexadecimal format (`hex:XXXXXXXX`), where each byte of the dataset is represented as two-digit hexadecimal numbers.
- `<PIN>` is the PIN code for authentication.
- `<DISCRIMINATOR>` is the discriminator value.

Commissioning over Existing IP Network

A Commissionee already connected to a Wi-Fi, Ethernet, or Thread network can be commissioned without BLE discovery, using **mDNS service discovery** instead.

The following CHIP-Tool command commissions a device that is already connected to a **Wi-Fi** or **Ethernet** network:

```
./chip-tool pairing onnetwork 0x1122 20202021
./chip-tool pairing onnetwork <NODE_ID_TO_ASSIGN> <PIN>
```

The parameters are defined as follows:

- `<NODE_ID_TO_ASSIGN>` is the **node ID** assigned to the device being commissioned.

- `<PIN>` is the PIN code for authentication.

The following CHIP-Tool command commissions a device that is already connected to a **Thread** network:

```
./chip-tool pairing onnetwork-thread 0x1122 hex:<OPERATIONAL_DATASET>
20202021
./chip-tool pairing onnetwork-thread <NODE_ID_TO_ASSIGN> hex:
<OPERATIONAL_DATASET> <PIN>
```

The parameters are defined as follows:

- `<NODE_ID_TO_ASSIGN>` is the **node ID** assigned to the device being commissioned.
- `<OPERATIONAL_DATASET>` is the **Thread Operational Dataset**, which contains the network credentials for joining a Thread network.
- `<PIN>` is the **PIN code** for authentication.

Removing a Device from Fabric

The following CHIP-Tool command removes a device from the Matter Fabric:

```
./chip-tool pairing unpair 0x1122
./chip-tool pairing unpair <NODE_ID_TO_ASSIGN>
```

The parameters are defined as follows:

- `<NODE_ID_TO_ASSIGN>` is the **node ID** assigned to the device being removed.

Reading Device Attributes

The following command is used to read the current state of the "on/off" attribute from the target device:

```
./chip-tool onoff read on-off 0x1122 1
```

In this command:

- `onoff` is the cluster for controlling the power state.
- `read` is a request to retrieve the current state.
- `on-off` is the specific attribute being queried.
- `0x1122` is the hexadecimal node ID of the device.
- `1` is the endpoint number on the target device.

Similarly, the command below reads the measurement value from the temperature sensor:

```
./chip-tool temperaturemeasurement read measured-value 0x1122 1
```

Sending Commands to Device

The following command is used to turn on and toggle the device:

```
./chip-tool onoff on 0x1122 1
./chip-tool onoff toggle 0x1122 1
```

Subscribing to Attributes and Events

This section describes how to subscribe to attributes and events in a Matter device using CHIP-Tool.

Subscribing to Attributes

Display all the attributes available for subscription in a given cluster:

```
./chip-tool <cluster-name> subscribe
```

Subscribe to an attribute:

```
./chip-tool <cluster-name> subscribe <argument> <min-interval> <max-
interval> <node_id> <endpoint_id>
```

In this command:

- **cluster-name** is the name of the cluster.
- **argument** is the name of the chosen argument.
- **min-interval** specifies the minimum number of seconds that must elapse since the last report for the server to send a new report.
- **max-interval** specifies the number of seconds that must elapse since the last report for the server to send a new report.
- **node-id** is the user-defined ID of the commissioned node.
- **endpoint_id** is the ID of the endpoint where the chosen cluster is implemented.

For example:

```
./chip-tool doorlock subscribe lock-state 5 10 1 1
./chip-tool temperaturemeasurement subscribe measured-value 3 10 1 1
./chip-tool relativehumiditymeasurement subscribe measured-value 3 10 1
2
./chip-tool occupancysensing subscribe occupancy 3 10 1 3
```

Subscribing to Events

Display all the events available for subscription in a given cluster:

```
./chip-tool <cluster-name> subscribe-event
```

Subscribe to an event:

```
./chip-tool <cluster-name> subscribe-event <event-name> <min-interval>
<max-interval> <node_id> <endpoint_id>
```

In this command:

- **cluster-name** is the name of the cluster.
- **event-name** is the name of the chosen event.

- **min-interval** specifies the minimum number of seconds that must elapse since the last report for the server to send a new report.
- **max-interval** specifies the number of seconds that must elapse since the last report for the server to send a new report.
- **node_id** is the user-defined ID of the commissioned node.
- **endpoint_id** is the ID of the endpoint where the chosen cluster is implemented.

For example:

```
./chip-tool doorlock subscribe-event door-lock-alarm 5 10 1 1
```

References

- How to Install Matter on RPi (https://mattercoder.com/codelabs/how-to-install-matter-on-rpi/)
- Setting up the Matter Hub (Raspberry Pi) (https://docs.silabs.com/matter/latest/matter-thread/raspi-img)
- CHIP-Tool - Commissioning Device over BLE (https://github.com/project-chip/connectedhomeip/blob/master/examples/chip-tool/README.md#commission-a-device-over-ble)
- CHIP-tool Source Code (https://github.com/project-chip/connectedhomeip/tree/master/examples/chip-tool)
- Silicone Labs' Matter Commissioning Guide (https://docs.silabs.com/matter/2.2.1/matter-overview-guides/matter-commissioning)

ESP Matter Controller

Overview

This section demonstrates how to build and run the **ESP Matter Controller Example** from the **ESP-Matter SDK** ("ESP Matter Solution" in "Espressif") on **ESP Thread Border Router** ("Hardware" in "Espressif") board. See also Matter Controller implementation on **ESP32-C6** (https://github.com/albert-gee/esp_matter_controller).

This Matter Controllers implementation supports the following features:

- BLE-WiFi pairing
- BLE-Thread pairing
- On-network pairing
- Invoke cluster commands
- Read attributes commands
- Read events commands
- Write attributes commands
- Subscribe attributes commands
- Subscribe events commands
- Group settings command.

Build and Run

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.
- ESP-Matter development environment (ESP-Matter Setup) is set up.
- ESP Thread Border Router ("Hardware" in "Espressif") board is available.

Step 1: Set Up Environment

Set up the environment and navigate to the `ESP Matter Thread Border Router` example directory:

```
get_idf
get_matter
```

Step 2: Open Matter Controller Example

Navigate to the `Matter Controller` examples directory:

```
cd $ESP_MATTER_PATH/examples/controller
```

Step 3: Set the target device

Set the target device to **esp32s3** for Thread Border Router since it uses the ESP32-S3 chip:

```
idf.py -D SDKCONFIG_DEFAULTS="sdkconfig.defaults.otbr" set-target
esp32s3
```

Step 4: Edit Project's CMakeLists.txt

```
nano CMakeLists.txt
```

The following changes have to be made to the `CMakeLists.txt` file:

```
# Cut this part and paste it before the `if(NOT DEFINED
ENV{ESP_MATTER_DEVICE_PATH})` block:
set(ESP_MATTER_PATH $ENV{ESP_MATTER_PATH})
set(MATTER_SDK_PATH ${ESP_MATTER_PATH}/connectedhomeip/connectedhomeip)
set(ENV{PATH}
"$ENV{PATH}:${ESP_MATTER_PATH}/connectedhomeip/connectedhomeip/.environ
ment/cipd/packages/pigweed")
```

Step 5: Connect to Computer

Connect the board to the computer using a USB cable.

The **ESP Thread Border Router** board should be connected using USB-2.

Step 6: Build and Flash

```
idf.py build
idf.py -p <PORT> erase-flash flash monitor
```

Testing

Wi-Fi Management

Connect to Wi-Fi

```
matter esp wifi connect <SSID> <PASSWORD>

# Example: Connect to Wi-Fi with SSID "MyWiFi" and password "MyPass"
matter esp wifi connect MyWiFi MyPass
```

Thread CLI

See also ESP OpenThread CLI Usage ("Usage" in "ESP OpenThread CLI") for more Thread commands.

Initialize and Commit Thread Dataset

```
matter esp ot_cli dataset init new
matter esp ot_cli dataset commit active
```

Start Thread Network

```
matter esp ot_cli ifconfig up
matter esp ot_cli thread start
```

Commissioning Matter Devices

BLE-Wi-Fi Commissioning

Commission a Matter device using BLE and Wi-Fi:

```
matter esp controller pairing ble-wifi <NODE_ID> <WIFI_SSID>
<WIFI_PASSWORD> <PIN_CODE> <DISCRIMINATOR>

Example: Pair a device over BLE Wi-Fi with SSID "MyWiFi" and password
"MyPass"
matter esp controller pairing ble-wifi 0x1122 MyWiFi MyPass 20202021
3840
```

BLE-Thread Commissioning

Pair a Matter device using BLE and Thread:

```
matter esp controller pairing ble-thread <NODE_ID> <THREAD_DATASET>
<PIN_CODE> <DISCRIMINATOR>

# Example: Pair a device over BLE Thread with a dataset string
matter esp controller pairing ble-thread 0x1122 <DATASET> 20202021 3840
```

Reading Attributes

Read a specific attribute from a cluster on a Matter device:

```
matter esp controller read-attr <NODE_ID> <ENDPOINT> <CLUSTER_ID>
<ATTRIBUTE_ID>
```

Root Node Clusters

The root node (Endpoint 0) has the following clusters:

- Basic Information (0x0028)
- General Commissioning (0x0030)
- Network Commissioning (0x0031)
- General Diagnostics (0x0033)
- Administrator Commissioning (0x003C)
- Operational Credentials (0x003E)

- Group Key Management (0x003F)

```
# Example: BASIC INFORMATION CLUSTER (0x0028)
## VendorName
matter esp controller read-attr 0x1122 0 0x0028 1
## VendorID
matter esp controller read-attr 0x1122 0 0x0028 2
## ProductName
matter esp controller read-attr 0x1122 0 0x0028 3
## ProductID
matter esp controller read-attr 0x1122 0 0x0028 4

# Example: GENERAL DIAGNOSTICS CLUSTER (0x0033)
## NetworkInterfaces
matter esp controller read-attr 0x1122 0 0x0033 0
## RebootCount
matter esp controller read-attr 0x1122 0 0x0033 1
## UpTime
matter esp controller read-attr 0x1122 0 0x0033 2

# Example: NODE OPERATIONAL CREDENTIALS CLUSTER (0x003D)
## NOCs
matter esp controller read-attr 0x1122 0 0x003E 0
## Fabrics
matter esp controller read-attr 0x1122 0 0x003E 1
## SupportedFabrics
matter esp controller read-attr 0x1122 0 0x003E 2
## CommissionedFabrics
matter esp controller read-attr 0x1122 0 0x003E 3
## TrustedRootCertificates
matter esp controller read-attr 0x1122 0 0x003E 4
## CurrentFabricIndex
matter esp controller read-attr 0x1122 0 0x003E 5
```

On/Off Cluster

On/Off cluster (0x6).

```
# Example: OnOff attribute (0x0) from the On/Off cluster (0x6)
matter esp controller read-attr 0x1122 0x1 0x6 0x0
```

Temperature Measurement Cluster

Temperature Measurement cluster (0x0402).

```
## Example: Read Temperature Measurement's measured value
matter esp controller read-attr 0x1122 1 0x0402 0x0000
## Example: Read Temperature Measurement's min measured value
matter esp controller read-attr 0x1122 1 0x0402 0x0001
## Example: Read Temperature Measurement's max measured value
matter esp controller read-attr 0x1122 1 0x0402 0x0002
```

Pressure Measurement Cluster

Pressure Measurement cluster (0x0403).

```
## Example: Read Pressure Measurement's measured value
matter esp controller read-attr 0x1122 1 0x0403 0x0000
## Example: Read Pressure Measurement's min measured value
matter esp controller read-attr 0x1122 1 0x0403 0x0001
## Example: Read Pressure Measurement's max measured value
matter esp controller read-attr 0x1122 1 0x0403 0x0002
```

Thread Border Router Management Cluster

Thread Border Router Management cluster (0x0452).

```
## Example: Read Thread Border Router's name
matter esp controller read-attr 0x1122 1 0x0452 0x0000
## Example: Read Thread Border Router's vendor
matter esp controller read-attr 0x1122 1 0x0452 0x0001
```

Access Control Cluster

Access Control (Access Control) cluster (0x001F).

```
matter esp controller read-attr 0x1122 0 0x001F 0x0000
```

Invoking Cluster Commands

Invoke a command on a Matter device:

```
matter esp controller invoke-cmd <NODE_ID> <ENDPOINT> <CLUSTER_ID>
<COMMAND_ID> {}
```

On/Off Cluster Commands

```
# Example: Turn on a light (cluster 0x6, command 0x1) on node 0x1122 at
endpoint 0x1
matter esp controller invoke-cmd 0x1122 0x1 0x6 0x1 {}
```

References

- ESP-Matter Controller Example (https://github.com/espressif/esp-matter/tree/main/examples/controller)

ESP-Mesh-Lite Network

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.

Build and Run

Step 1: Prepare the Environment

```
get_idf
```

Step 2: Clone ESP-Mesh-Lite SDK

This project uses ESP-Mesh-Lite SDK ("ESP-Mesh-Lite" in "Espressif") locked to commit 1a27d19 (https://github.com/espressif/esp-mesh-lite/commit/1a27d1944819cf17ec76cd63ed90c20a1bc28ed9).

```
cd ~/esp
git clone https://github.com/espressif/esp-mesh-lite.git
cd ~/esp/esp-mesh-lite/
git checkout 1a27d1944819cf17ec76cd63ed90c20a1bc28ed9
```

Step 3: Choose No-Router Example

The **no_router** example in the ESP-Mesh-Lite repository demonstrates how to create a mesh network without a traditional Wi-Fi router. Devices communicate directly, forming a decentralized structure.

```
cd examples/no_router
```

Step 4: Set Target for No-Router Example

```
cd examples/no_router
idf.py set-target esp32c6
```

Step 5: Configure the Device

```
idf.py menuconfig
```

A device's role is set using `CONFIG_MESH_ROOT`, it can be configured in **menuconfig** `Example Configuration -> Root Device`:

- If **Enabled (`CONFIG_MESH_ROOT=y`)**, the device creates a **SoftAP** and becomes the **root node**.
- If **Disabled (`CONFIG_MESH_ROOT=n`)**, the device joins an existing network as a **STA** while also creating a **SoftAP** for other devices, making it an **intermediate node** that extends the mesh.

SoftAP settings **CONFIG_BRIDGE_SOFTAP_SSID** and **CONFIG_BRIDGE_SOFTAP_PASS** are configured in **menuconfig** `Component config -> Bridge Configuration -> The interface used to provide network data fowarding for other devices -> SoftAP Config`.

The following settings are also available:

- `Component config -> ESP Wi-Fi Mesh Lite`

Step 6: Build

```
idf.py build
```

Step 7: Flash

```
idf.py -p <PORT_TO_ESP32_C6> flash monitor
```

Provisioning

Provisioning example: https://github.com/espressif/esp-mesh-lite/tree/master/examples/mesh_wifi_provisioning

Provisioning Library: Provisioning library provides a mechanism to send network credentials and/or custom data to ESP32 (or its variants like S2, S3, C3, etc.) or ESP8266

devices.

This repository contains the source code for the companion Android app for this provisioning mechanism: https://github.com/espressif/esp-mesh-lite/tree/feature/zero_provisioning_android?tab=readme-ov-file#features

Accessory Devices

Overview

Accessory Devices are Matter (Matter)-compatible Thread End Devices ("End Device" in "Thread") that monitor and control environmental conditions within a Grow Chamber. They include **sensors** and **actuators**.

Sensors

A sensor is any device that generates some sort of output when exposed to a phenomenon.

The table below lists the sensors utilised in this project:

Sensor	Purpose
BMP280	Temperature and air pressure
BH1750 or TSL2561	Light intensity
DHT22	Temperature and humidity
MH-Z19B	Carbon dioxide (CO_2) levels

The sensor devices utilise the ESP32-H2 microcontrollers with Thread connectivity and low power consumption, enabling effective operation in various agricultural settings.

Actuators

Actuators are located on the output side of IoT solutions. They change their state based on an analog or digital signal from the microcontroller and produce an output that affects the environment.

The table below lists the actuators that will be utilised in this project:

Actuator	Purpose
Ultrasonic mist maker	Fog generation
Exhaust fan, 120mm DC	Ventilation
LED grow lights	Plant growth lighting
Air pump and air stone	Aeration for water inside Root Chamber

Actuator devices utilise the ESP32-DevKitM-1 microcontrollers. In contrast to ESP32-H2, the ESP32-DevKitM-1 does not support Thread connectivity and has higher power consumption. It is used to control actuators, which inherently consume a lot of power.

Matter Pressure and Temperature Sensor

Overview

The **Matter Pressure and Temperature Sensor** is a Matter (Matter)-compatible SED ("End Device" in "Thread") device that reads temperature and pressure data from a BMP280 sensor and exposes the readings as Matter attributes.

This section describes the development of a Matter-compatible temperature and pressure sensor.

Source Code: GitHub (https://github.com/albert-gee/matter_bmp280)

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.
- ESP-Matter development environment (ESP-Matter Setup) is set up.
- ESP32-H2-DevKitM-1 (https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32h2/esp32-h2-devkitm-1/index.html) is available.

Hardware Assembly

Wiring BMP280 to ESP32

The **ESP32-H2** development board reads data from the **BMP280**, a low-power sensor designed for battery-powered devices. They communicate using **I2C (Inter-Integrated Circuit)**, a serial, synchronous, half-duplex protocol. The ESP32 has two I2C ports, each capable of operating as a controller or target. In this project, the ESP32-H2 acts as the controller, while the BMP280 functions as the target.

The I2C bus has two lines: the Serial Data Line (SDA) and the Serial Clock Line (SCL). On the ESP32-H2, SDA and SCL can be assigned to any available GPIO pins.

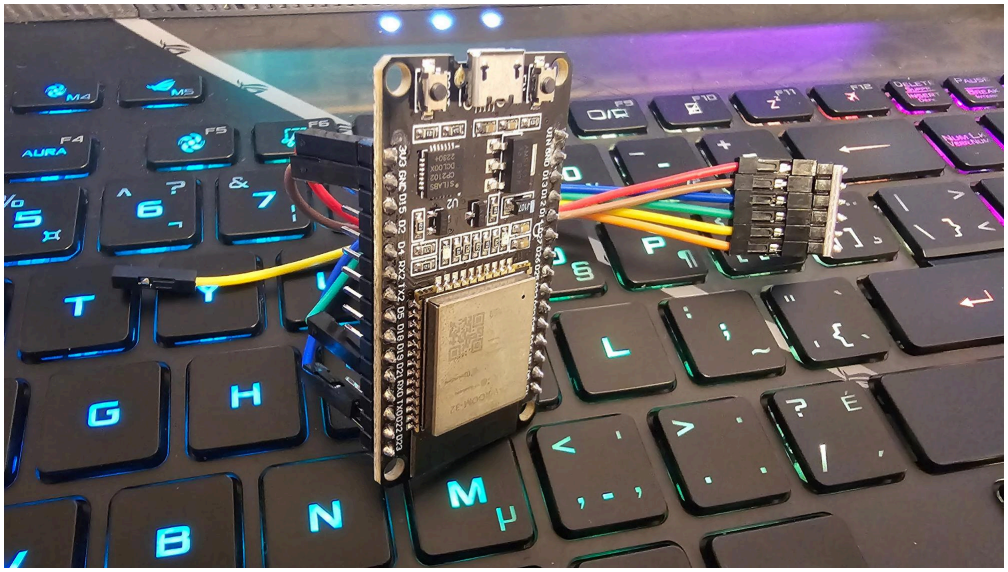
On the ESP32-DevKitM-1, GPIO21 is used for SDA, and GPIO22 is used for SCL.

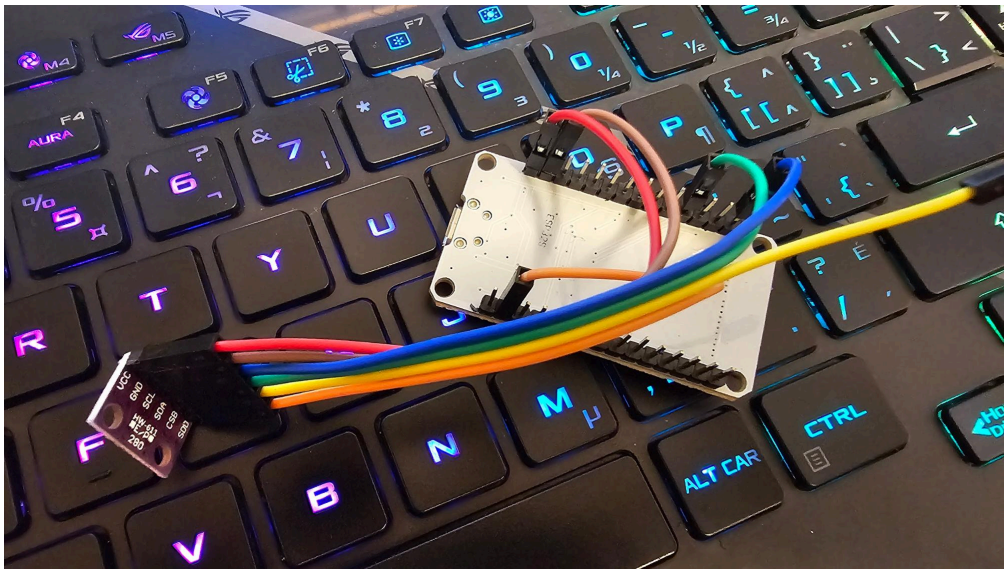
The table below shows the connections for the BMP280 sensor to the ESP32-DevKitM-1

microcontroller:

BMP280 Pin	ESP32 Pin
VCC	3.3V
GND	GND
SDA	GPIO21
SCL	GPIO22
SDO	GND

The pictures below show the connections for the BMP280 sensor to the ESP32-DevKitM-1 microcontroller:





Testing I2C Connectivity

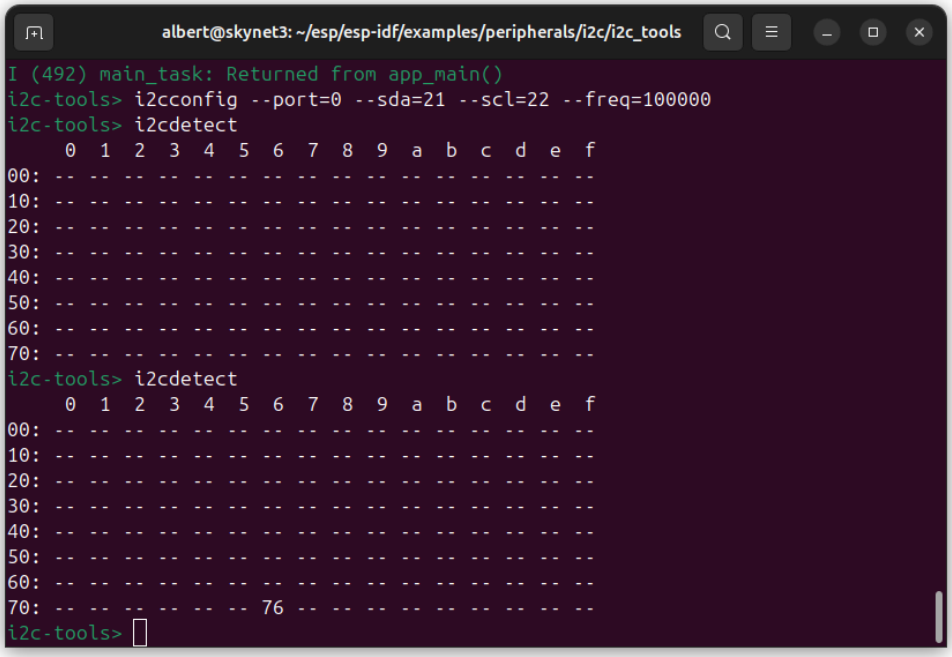
The I2C tools (https://github.com/espressif/esp-idf/tree/release/v5.4/examples/peripherals/i2c/i2c_tools) from ESP-IDF examples were used to test communication with the sensor device.

The following command configures the I2C bus with specific GPIO number, port number and frequency:

```
i2cconfig --port=0 --sda=21 --scl=22 --freq=100000
```

The following command scans an I2C bus for devices and output a table with the list of detected devices on the bus:

```
i2cdetect
```



It displays the address 0x76 since the SDO pin is connected to GND.

The following command get the value of the “ID” register which contains the chip identification number chip_id, which is 0x58:

```
i2cget -c 0x76 -r 0xD0 -l 1
```

- -c option to specify the address of I2C device (acquired from i2cdetect command).
- -r option to specify the register address you want to inspect.
- -l option to specify the length of the content.

Firmware Development

The firmware for the ESP32-H2 was developed using the **ESP-IDF framework**.

The BMP280 datasheet (*BST-BMP280-DS001-26*) is used as a reference during development.

Project Bootstrap

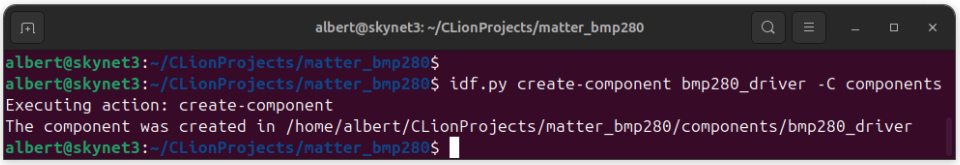
Follow the following steps from the ESP32 Project Workflow (ESP32 Project Workflow) guide:

- Step 1: Set Up ESP-IDF and ESP-Matter Environment ("Step 1: Set Up ESP-IDF and ESP-Matter Environment" in "ESP32 Project Workflow")
- Step 2: Create a New Project ("Step 2: Create a New Project" in "ESP32 Project Workflow")
- Step 3: Set Target Device ("Step 3: Set Target Device" in "ESP32 Project Workflow")
- Step 4: Open the Project in CLion IDE ("Step 4: Open the Project in CLion IDE" in "ESP32 Project Workflow")
- Step 5: Create Default Configuration ("Step 5: Create Default Configuration" in "ESP32 Project Workflow")
- Step 6: Update CMakeLists.txt ("Step 6: Update CMakeLists.txt" in "ESP32 Project Workflow")

Driver Component Development

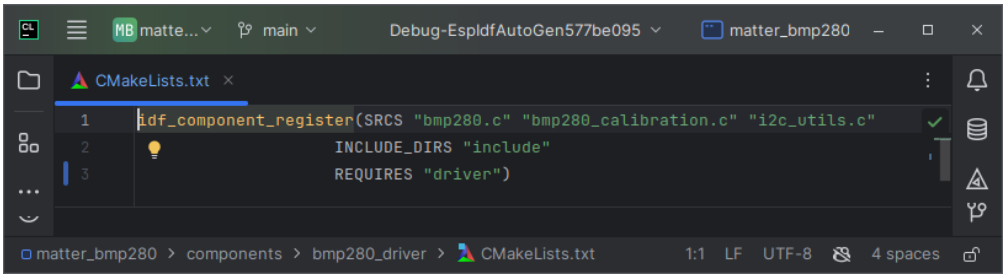
The command below creates a new component named `bmp280_driver` inside the `components` directory.

```
idf.py create-component bmp280_driver -C components
```



The ESP-IDF framework includes a driver for working with I2C devices. To ensure that this component can use the driver, the `REQUIRES` directive must be added to `components/bmp280_driver/CMakeLists.txt` as follows:

```
REQUIRES "driver"
```



This directive ensures that the build system includes the I2C driver as a dependency for the component during the build process.

The `i2c_utils.h` header file declares utility functions for initializing, managing, and performing operations on an I2C master bus and its connected devices.

The `i2c_utils.c` file implements these functions and includes `driver/i2c_master.h` ($IDF_PATH/components/driver/i2c/include/driver/i2c_master.h) to access the driver's API in controller mode.

The BMP280 code is designed according to the specifications outlined in the BST-BMP280-DS001-26 datasheet. Communication with the sensor is performed through read and write operations on its 8-bit registers.

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00
temp_lsb	0xFB	temp_lsb<7:0>								0x00
temp_msb	0xFA	temp_msb<7:0>								0x80
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00
press_lsb	0xF8	press_lsb<7:0>								0x00
press_msb	0xF7	press_msb<7:0>								0x80
config	0xF5	t_sb[2:0]			filter[2:0]				spi3w_en[0]	0x00
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]			mode[1:0]		0x00
status	0xF3					measuring[0]			im_update[0]	0x00
reset	0xE0	reset[7:0]								0x00
id	0xD0	chip_id[7:0]								0x58
calib25...calib00	0xA1…0x88	calibration data								individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Revision	Reset
Type:	do not write	read only	read / write	read only	read only	read only	write only

The `bmp280_driver.h` header file declares the configurations, constants, and functions needed to interface with and operate the sensor, based on the datasheet.

The `bmp280_driver.c` file implements these functions, providing support for sensor initialization, configuration, and data handling. It adheres to the datasheet to verify the sensor, apply compensation to raw measurements, and configure parameters like oversampling, operating modes, and filters.

The BMP280 sensor uses factory-programmed calibration parameters to adjust raw data for accurate measurements. These parameters are stored in the sensor's non-volatile memory (NVM) during production, are unique to each device, and cannot be modified by the user, as described in Section 3.11.2 of the datasheet. The `bmp280_calibration.h` header declares the constants, structures, and functions needed to handle this calibration data. It includes functionality to read the parameters and apply them to raw pressure and temperature readings. The `bmp280_calibration.c` file implements these functions, following the temperature and pressure compensation formulas specified in Section 3.11 of the datasheet.

Matter Component Development

The command below creates a new component named bmp280_driver inside the components directory.

```
idf.py create-component matter_interface -C components
```

The `matter_temperature_sensor.h` file defines functions for managing a Matter-compatible temperature sensor.

The `matter_temperature_sensor.c` file implements these functions using Espressif's Matter SDK, built on the Matter SDK.

In the Matter ecosystem, devices are represented as Nodes, which consist of Endpoints representing specific functions, such as temperature measurement. Endpoints are organized into Clusters that group related features. This implementation sets up a Node with an Endpoint for temperature measurement, configures its Clusters, and manages communication within the Matter network.

Integration of BMP280 Driver with ESP-Matter

The bmp280_driver component outputs temperature and pressure readings as 32-bit signed integers (int32_t), following the BMP280 datasheet specifications. These integers are used to process the sensor's raw 20-bit data for precise calculations, including compensation and scaling. The readings represent scaled real-world values to preserve decimal precision using integer arithmetic. For example, a temperature of 25.25°C is stored as 2525 (in units of 0.01°C), and a pressure of 1013.25 is stored as 101325 (in Pascals).

During integration with ESP-Matter, an issue arose because ESP-Matter defines attributes like measured_value, min_measured_value, and max_measured_value in its Temperature and Pressure Measurement Clusters using the nullable< int16_t> data type. This required the BMP280's outputs, represented as int32_t, to be reduced to fit within the int16_t range of -32,768 to 32,767. The BMP280 outputs temperature values within a range of -40°C to +85°C and pressure values within a range of 300 hPa to 1100 hPa. For temperature, no additional scaling was necessary because even the maximum value, when scaled (e.g., 85.00°C becomes 8500), fits comfortably within the int16_t range without any loss of precision.

For pressure, however, scaled values can exceed the limit of the int16_t range. Multiplying higher values by 100 to preserve decimal precision increases their magnitude

beyond the int16_t limit. For example, a pressure of 1013.25 becomes 101325 when scaled, which far exceeds the int16_t range.

To address this, the scaling approach was modified. Instead of multiplying by 100 to represent hundredths of Pascals, pressure values were scaled by 10. This reduced the magnitude of the scaled values, allowing them to fit within the int16_t range. For example, a pressure of 1013.25 is scaled to 10132, which fits comfortably within the range. Similarly, the maximum pressure of 1100.00 is scaled to 11000.

Reading the Temperature

To manage temperature readings, two approaches were considered:

- Regularly poll the temperature sensor, for example, using a timer. Each new reading is written directly to the attribute's stored value.
- Update the temperature reading only when a client requests it. In this approach, the sensor retrieves the latest temperature data and writes it to the attribute before responding to the client.

The second approach requires a mechanism to dynamically fetch the latest sensor data during a client read operation. ESP-Matter provides native support for callbacks during attribute write operations, but not for reads. In the ESP-Matter repository on GitHub, Issue #264 (https://github.com/espressif/esp-matter/issues/264) includes a discussion about using the `ATTRIBUTE_FLAG_OVERRIDE` flag to implement such functionality. The discussion describes a workaround involving modifications to the framework's internal `esp_matter_attribute.cpp` file to override the default attribute behavior and introduce custom read logic.

A more maintainable alternative would be to leverage the flag directly within application code. This approach avoids modifying the framework itself. However, it introduces additional complexity and can cause delays in client responses, as the sensor data must be fetched in real-time for each read request. Due to these challenges, the first option was chosen for its simplicity and faster response times.

Testing

Testing with CHIP-Tool

Refer to CHIP-Tool (CHIP-Tool) for more details.

```
matter config
matter onboardingcodes ble
```

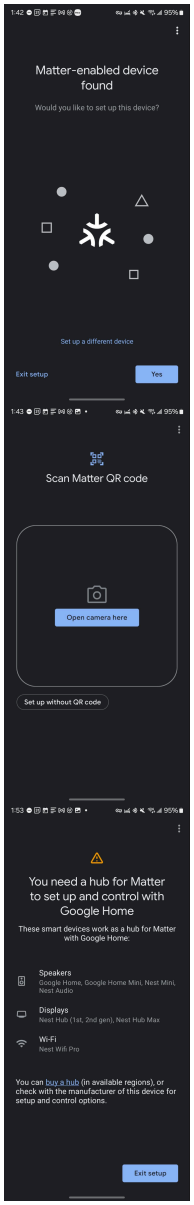
```
matter esp attribute get 0x0001 0x00000402 0x00000000
matter esp attribute get 0x0002 0x00000403 0x00000000
```

Connecting to Google Home

It is also possible to use third-party software, such as the Google Home app, as a controller. However, the device appears offline in Google Home despite being successfully commissioned. To address this issue, a structured troubleshooting process was undertaken:

- Flash Memory Reset: The device's flash memory was erased to remove any residual configurations that could interfere with its functionality.
- Network Configuration Check: It was confirmed that both the mobile device and the hardware were connected to a 2.4 GHz Wi-Fi network, as the system does not support 5 GHz networks.
- Device Reset: The hardware was reset to ensure proper initialization and resolve any temporary issues.

Vaishali Avhale, Associate QA Engineer at Espressif Systems, provided feedback on a related issue in the Espressif ESP-Matter GitHub repository (Issue #1125 (https://github.com/espressif/esp-matter/issues/1125)). She tested the system using a Google Nest Hub 2nd Generation as a hub and confirmed that it worked correctly in this configuration. She noted that a hub device is a required component for proper operation within Google's ecosystem.



References

- BMP280 – Data sheet (https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp280-ds001-26.pdf)
- Telink Matter Developers Guide (https://wiki.telink-semi.cn/doc/an/TelinkMatterDevelopersGuide_en.pdf)

- ESP-Matter Sensors Example (https://github.com/espressif/esp-matter/tree/main/examples/sensors)

Matter Relay Switch

Overview

The **Matter Relay Switch** is a Matter (Matter)-compatible REED ("End Device" in "Thread") device that can be controlled remotely to turn a relay on or off. It can be used to control various devices, such as lights or fans.

This section describes the development of the **Matter Relay Switch**.

Source Code: GitHub (https://github.com/albert-gee/matter_relay)

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.
- ESP-Matter development environment (ESP-Matter Setup) is set up.
- ESP32-H2-DevKitM-1 (https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32h2/esp32-h2-devkitm-1/index.html) is available.

Hardware Assembly

Wiring Relay to ESP32

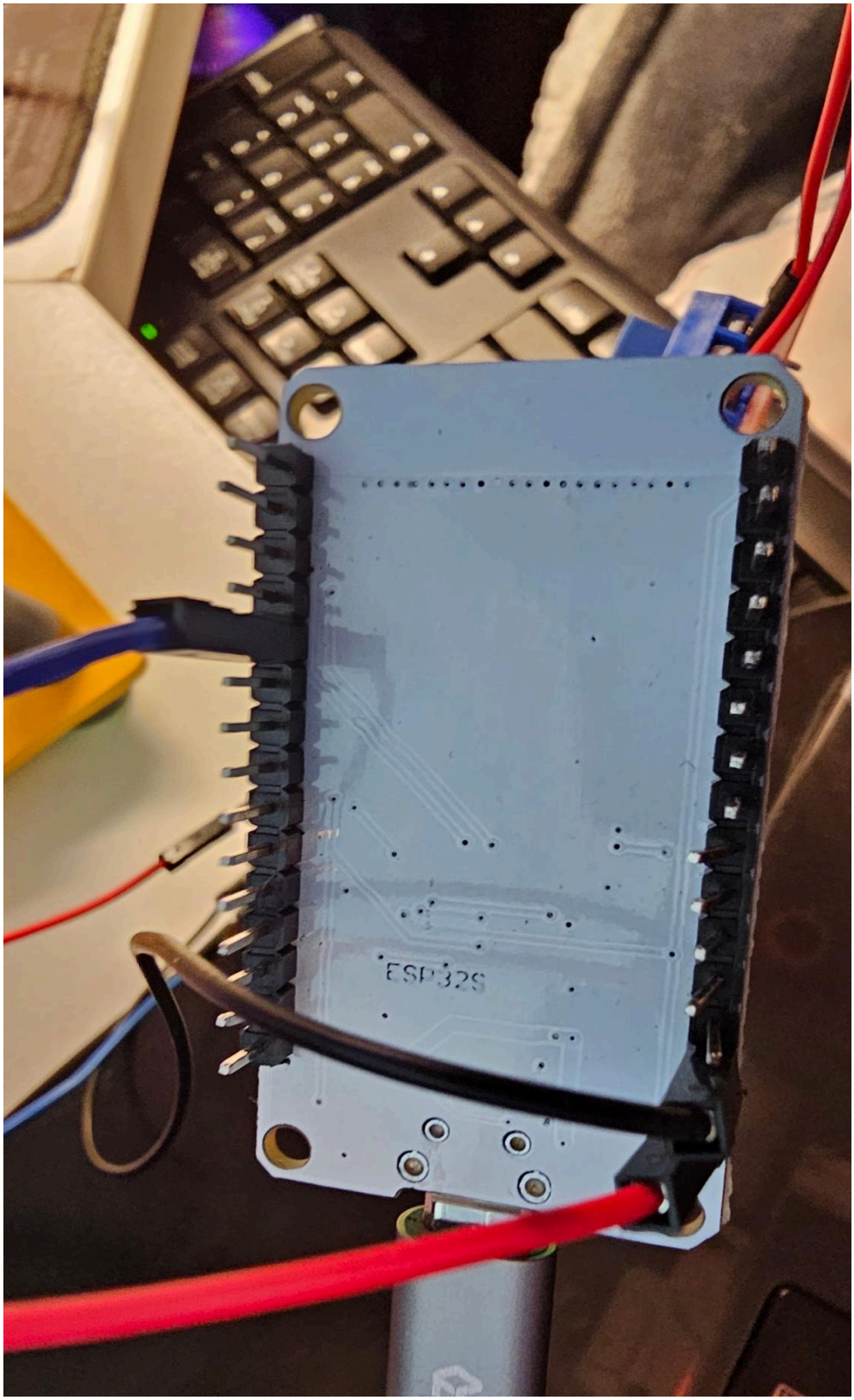
The **Valefod 5V 1-Channel Relay Module** is used to control loads up to 10A using a low-power control signal from a microcontroller such as an Arduino or ESP32. It features optocoupler isolation for improved signal integrity and supports both high and low trigger configurations.

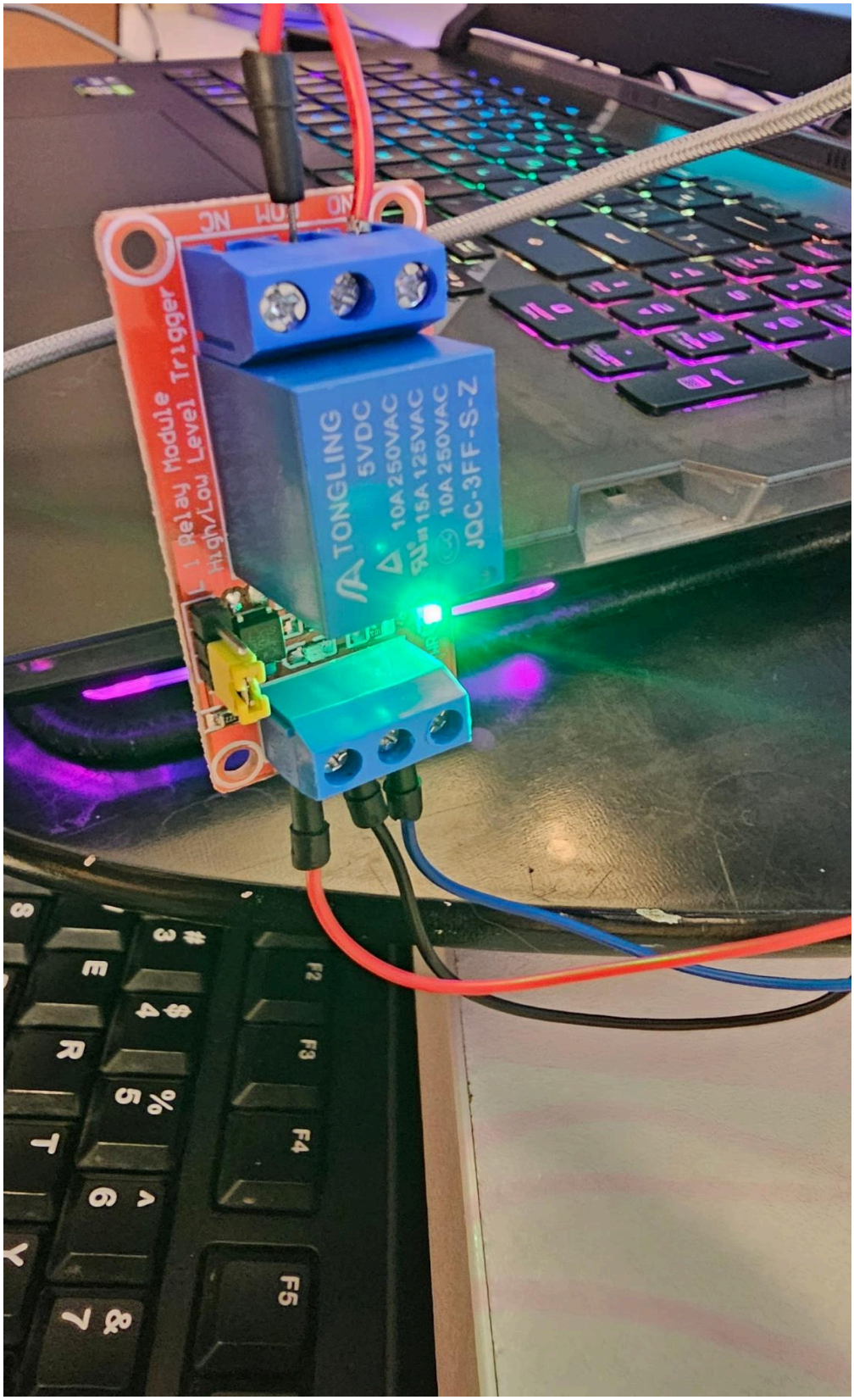
The relay has the following terminal layout:

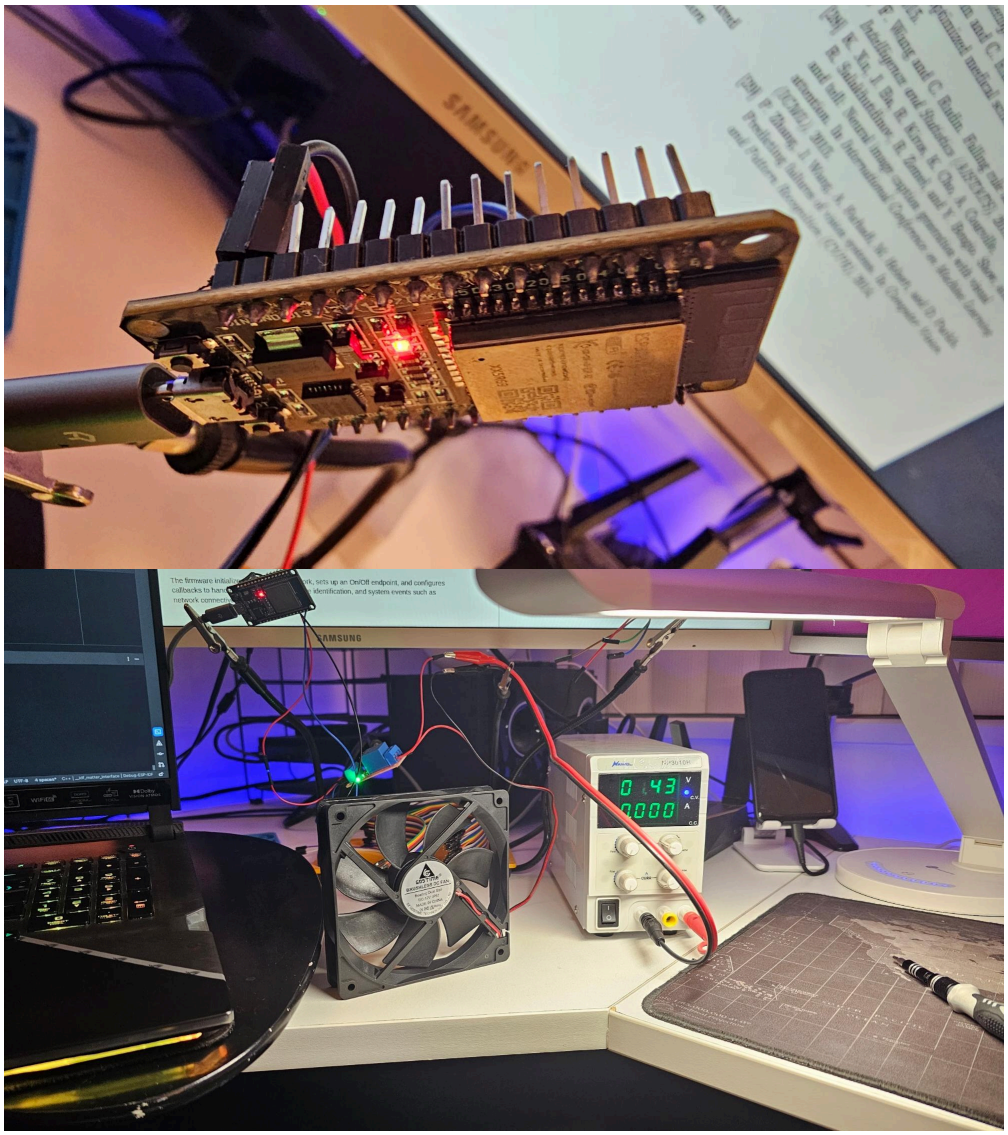
- **Side 1: Control Terminals**
  - DC+ (Power Supply Positive): Connect to a 5V power supply.
  - DC- (Power Supply Negative): Connect to the ground of the power supply.
  - IN (Input Signal): Connect to a GPIO pin on the microcontroller to control the relay.
- **Side 2: Switch Terminals**

- COM (Common Terminal): Shared terminal for the relay's switch.
- NO (Normally Open): Open circuit by default; closes when the relay is activated.
- NC (Normally Closed): Closed circuit by default; opens when the relay is activated.

The Normally Open (NO) terminal is used when the circuit should remain open (off) by default, requiring activation to close the circuit. The Normally Closed (NC) terminal is used when the circuit should remain closed (on) by default, opening only when the relay is activated.


ESP32S


1 Relay Module
High/Low Level Trigger
NC
COM
NO
TONGLING
5VDC
10A 250VAC
15A 125VAC
10A 250VAC
JQC-3FF-S-Z



Firmware Development

The firmware uses the ESP-IDF ("ESP-IDF Framework" in "Espressif") framework with the SDK for Matter. It sets up an On/Off endpoint and configures callbacks to manage Matter attributes, enabling GPIO-based relay control.

Project Bootstrap

Follow the following steps from the ESP32 Project Workflow (ESP32 Project Workflow) guide:

- Step 1: Set Up ESP-IDF and ESP-Matter Environment ("Step 1: Set Up ESP-IDF and ESP-Matter Environment" in "ESP32 Project Workflow")

- Step 2: Create a New Project ("Step 2: Create a New Project" in "ESP32 Project Workflow")
- Step 3: Set Target Device ("Step 3: Set Target Device" in "ESP32 Project Workflow")
- Step 4: Open the Project in CLion IDE ("Step 4: Open the Project in CLion IDE" in "ESP32 Project Workflow")
- Step 5: Create Default Configuration ("Step 5: Create Default Configuration" in "ESP32 Project Workflow")
- Step 6: Update CMakeLists.txt ("Step 6: Update CMakeLists.txt" in "ESP32 Project Workflow")

GPIO Configuration

The header file `$IDF_PATH/components/esp_driver_gpio/include/driver/gpio.h` can be included with:

```
#include "driver/gpio.h"
```

This header file is a part of the API provided by the `esp_driver_gpio` component. To declare that your component depends on esp_driver_gpio, the following line has to be added to CMakeLists.txt:

```
REQUIRES esp_driver_gpio
```

or

```
PRIV_REQUIRES esp_driver_gpio
```

Startup Process

The device boots up and loads the firmware, initializing flash memory, configuring system memory, setting up GPIOs, and loading stored data from NVS.

Once initialization is complete, it starts the Matter stack, creates the relay node, and runs the OpenThread stack. Finally, it begins BLE advertising (CHIPoBLE) to enable pairing.

Testing

Testing with CHIP-Tool

Refer to CHIP-Tool (CHIP-Tool) for more details.

```
matter config
matter onboardingcodes ble

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

Orchestrator

Overview

This chapter describes the development of the **Orchestrator** device which facilitates system automation through the following processes:

- Establishing network infrastructure by creating and managing Thread (Thread) networks and Matter (Matter) Fabrics.
- Commissioning devices to join Matter fabrics (Matter Commissioning) and Thread networks ("Commissioning" in "Thread")
- Monitoring sensor data from the Grow Chamber and adjusting actuators in real time.

Prerequisites:

- ESP-IDF development environment (ESP-IDF Setup) is set up.
- ESP-Matter development environment (ESP-Matter Setup) is set up.
- ESP Thread Border Router ("Hardware" in "Espressif") board is available.

Development

Project Bootstrap

The following steps from ESP32 Project Workflow (ESP32 Project Workflow) guide are required to bootstrap the project:

- **Step 1: Set Up the Environment** ("Step 1: Set Up ESP-IDF and ESP-Matter Environment" in "ESP32 Project Workflow")
- **Step 2: Create a New Project** ("Step 2: Create a New Project" in "ESP32 Project Workflow") named `orchestrator`
- **Step 3: Set Target Device** ("Step 3: Set Target Device" in "ESP32 Project Workflow") to `esp32s3` since ESP Thread Border Router board is used.

- **Step 4: Open the Project in CLion IDE** ("Step 4: Open the Project in CLion IDE" in "ESP32 Project Workflow")

Project Build Configuration

Configure the build process by updating the CMakeLists.txt (https://github.com/albert-gee/old_macdonald_orchestrator/blob/main/CMakeLists.txt). It should include the following:

- **ESP_MATTER_PATH** and **MATTER_SDK_PATH** environment variables integrate ESP-Matter and Matter SDK.
- **ESP_MATTER_DEVICE_PATH** points to the correct hardware HAL.
- **EXTRA_COMPONENT_DIRS** includes component directories for ESP-Matter, SDK, and device HAL.
- **Essential CMake files** from IDF and ESP Matter Device must be included before any IDF_TARGET references.
- **Compiler flags** that add C++17, optimizations, Matter defines, disable Thread driver, and suppress RISCV warnings.

Project Default Configuration

The `sdkconfig.defaults` file is used to specify default configuration settings:

- Set Flash size to 8MB
- Enable Wi-Fi station mode only.
- Enable OpenThread with a Border Router.
- Configure IPv6 support for OpenThread (LwIP).
- Enable ESP32 BLE Controller.
- Enable Bluetooth with NimBLE stack.
- Use a custom partition table from `partitions.csv`.
- Enable the CHIP shell for Matter.

- Enable Matter Controller, disable the server and commissioner features.
- Disable OTA Requestor and route hook.
- Enable WebSocket server support.

Custom partition table

`CONFIG_PARTITION_TABLE_CUSTOM=y` and `CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions.csv"` in `sdkconfig.defaults` enable a custom partition table defined in `partitions.csv`:

- `nvs` (24 KB) stores persistent key-value data
- `phy_init` (4 KB) holds PHY calibration settings
- `factory` (3 MB) contains the main application firmware.
- `rcp_fw` stores firmware updates for the Radio Co-Processor (RCP) (["Architecture" in "Thread"]).

System Initialization

The main function initializes the essential components:

- **ESP NVS** (https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/nvs_flash.html) library stores key-value pairs in the device's flash memory.
- **ESP Event Loop** (https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/esp_event.html) Library lets components publish events indicating important occurrences, such as a successful Wi-Fi connection to the event loop, while other components can register handlers to respond. This decouples event producers from consumers, allowing components to react to state changes without direct application involvement.
- **ESP-NETIF** (https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/network/esp_netif.html) library is an abstraction layer that provides thread-safe APIs for managing network interfaces on top of a TCP/IP stack.
- Components (Components)

Deployment

Follow the steps from the ESP32 Project Workflow (ESP32 Project Workflow) guide:

- **Step 1: Generate TLS Certificates** ("WebSocket Secure (WSS)" in "Websocket Server")
- **Step 2: Build the Project** ("Step 7: Build the Project" in "ESP32 Project Workflow")
- **Step 3: Connect to Computer** ("Step 4: Connect the ESP Thread Border Router Board" in "ESP Basic Thread Border Router")
- **Step 4: Determine Serial Port** ("Step 8: Determine Serial Port" in "ESP32 Project Workflow")
- **Step 5: Flash Project to Target** ("Step 9: Flash Project to Target" in "ESP32 Project Workflow")
- **Step 6: Launch IDF Monitor** ("Step 10: Launch IDF Monitor" in "ESP32 Project Workflow")

Components

The following custom components have been developed for the Orchestrator:

- Matter Interface (Matter Interface)
- Websocket Server (Websocket Server)
- Thread Interface (Thread Interface)

Matter Interface

Overview

The **Matter Interface** is a part of the **Orchestrator** (Orchestrator) firmware.

This component initializes the **Matter** (Matter) stack, including CHIP memory, the OTA requestor, default providers, and the event loop.

It also provides initialization for the Matter **controller** client with optional **commissioner** (Matter Commissioning) support.

Development

Pre-requisites:

- The steps in the Project Bootstrap ("Project Bootstrap" in "Orchestrator") section are completed.
- System Initialization ("System Initialization" in "Orchestrator") is completed.

Bootstrap Component

The command below creates a new component named `matter_controller_interface` inside the `components` directory.

```
idf.py create-component matter_controller_interface -C components
```

Change the file extension of the `matter_controller_interface.c` file to `matter_controller_interface.cpp` and move it to the `src` folder.

```
mv components/matter_controller_interface/matter_controller_interface.c
components/matter_controller_interface/src/matter_controller_interface.
cpp
```

Matter Initialization

The `esp_matter::start()` (https://github.com/espressif/esp-matter/blob/f9f62dd6851d93dd9c770f0ed0066e822f955c35/components/esp_matte

r/esp_matter_core.cpp#L849) function in `esp_matter_core.cpp` sets up the Matter stack and creates the **default event loop** by calling `esp_event_loop_create_default` (https://github.com/espressif/esp-idf/blob/a45d713b03fd96d8805d1cc116f02a4415b360c7/components/esp_event/default_event_loop.c#L92), which is shared across the system. As part of this initialization, **ESP-Netif** is also brought up automatically through the ESP32-specific `PlatformManagerImpl` (https://github.com/espressif/connectedhomeip/blob/9b8fffe0bb4e7ba7aac319f5905070a3476db7cb/src/platform/ESP32/PlatformManagerImpl.cpp#L73), which connects the Matter stack to platform features like networking and timers. Wi-Fi integration is handled by ConnectivityManagerImpl_WiFi (https://github.com/espressif/connectedhomeip/blob/9b8fffe0bb4e7ba7aac319f5905070a3476db7cb/src/platform/ESP32/ConnectivityManagerImpl_WiFi.cpp), which manages Wi-Fi provisioning, mode switching, and network status reporting.

This component does not use the `esp_matter::start()` function from esp_matter_core. Instead, it performs manual initialization of the Matter stack, including OTA setup, CHIP memory initialization, provider configuration, event loop startup, and event handler registration.

Unlike esp_matter::start(), it does not launch the Matter server, enable endpoints, initialize the data model, or run server-specific callbacks.

Matter Controller Initialization

The component initializes the **Matter controller client** with a given node ID, fabric ID, and port.

It locks the CHIP stack, sets up the controller, and optionally configures commissioner support if enabled.

Node ID—64-bit unsigned, valid range from *0x0000000000000001* to *0xFFFFFFFFFFFFEFFF*; *0x0000* and *0xFFFF_FFF0...* are reserved.

Fabric ID—64-bit unsigned non-zero; 0 is reserved.

Listen Port—16-bit integer; IANA-assigned port for Matter over UDP and TCP.

Default values are:

- *MATTER_CONTROLLER_NODE_ID_DEFAULT* 0x0000000000011234

- *MATTER_CONTROLLER_FABRIC_ID_DEFAULT* 0x0000000000000001
- *MATTER_CONTROLLER_LISTEN_PORT_DEFAULT* 5540

Utility Functions

This component provides helper functions to interact with Matter devices using the controller client. It wraps common commands like pairing, reading attributes, invoking cluster commands, and subscribing to attribute changes.

All functions internally manage CHIP stack locking and memory allocation for attribute/event paths.

References

- ESP-Matter Controller Example (https://github.com/espressif/esp-matter/tree/main/examples/controller)
- ESP-Matter Controller Docs (https://docs.espressif.com/projects/esp-matter/en/latest/esp32/developing.html#matter-controller)
- ESP-Matter Controller Component (https://github.com/espressif/esp-matter/tree/main/components/esp_matter_controller/commands)
- Espressif's SDK for Matter - Matter Controller (https://docs.espressif.com/projects/esp-matter/en/latest/esp32s3/developing.html#matter-controller)

Websocket Server

Overview

The Orchestrator device uses WebSocket to exchange messages with connected clients.

This section outlines the development of the **Websocket Server** component.

Development

Bootstrap Component

The command below creates a new component named `websocket_server` inside the `components` directory.

```
idf.py create-component websocket_server -C components
```

Change the file extension of the `websocket_server.c` file to `websocket_server.cpp` and move it to the `src` folder.

```
mv components/matter_controller_interface/matter_controller_interface.c
components/matter_controller_interface/src/matter_controller_interface.
cpp
```

Enable WebSocket Support by setting the `CONFIG_HTTPD_WS_SUPPORT` option to `y` is the `sdkconfig.defaults` file.

WebSocket Secure (WSS)

The server uses ESP-IDF's esp_https_server (https://github.com/espressif/esp-idf/tree/master/components/esp_https_server) to handle secure WebSocket (WSS) connections over HTTPS.

A private Root CA signs the Orchestrator's certificate, which includes its IP address in the SAN field. The client has the Root CA certificate to verify the Orchestrator during TLS handshake.

Clients connect to `/ws`. The server performs the WebSocket handshake and upgrades the connection.

The server handles incoming WebSocket frames:

- `TEXT` echoed back
- `PING` replies with PONG
- `CLOSE` closes the connection gracefully

Outgoing messages are sent asynchronously using ESP-IDF's work queue.

TLS Certificates

This project requires locally generated certificates to establish a WSS ("WebSocket Secure (WSS)" in "Websocket Server") connection between the Orchestrator and a client application.

First-time Setup (after cloning):

```
chmod +x ./components/websocket_server/certs/cert_generate.sh
```

Generating Certificates:

```
# Generate certificates using the default ESP32 IP (192.168.4.1):
./components/websocket_server/certs/cert_generate.sh

# Alternatively, specify a custom ESP32 IP address:
./components/websocket_server/certs/cert_generate.sh 192.168.1.50
```

Keep-Alive Mechanism .3

The keep-alive mechanism automatically monitors active WebSocket clients to detect and remove dead connections. It sends periodic `PING` messages to each client and waits for a `PONG` response. If a client fails to respond within the configured timeout, it is marked as inactive and disconnected.

The module tracks clients by their socket file descriptors and runs as a separate FreeRTOS task, using a queue to handle client add, remove, and update events.

Message Handling

Incoming messages are handled by a registered JSON parser (Messages).

References

- ESP Websocket Echo Server Example (https://github.com/espressif/esp-idf/tree/master/examples/protocols/http_server/ws_echo_server)
- ESP HTTPS Websocket Server Example (https://github.com/espressif/esp-idf/tree/master/examples/protocols/https_server/wss_server)
- ESP Restful Server Example (https://github.com/espressif/esp-idf/tree/master/examples/protocols/http_server/restful_server)
- ESP HTTPS Server Example (https://github.com/espressif/esp-idf/tree/master/examples/protocols/https_server/simple)
- cJSON (https://github.com/espressif/esp-idf/tree/master/components/json)

Thread Interface

Overview

This section outlines the development of the **Thread Interface** component, which provides the features and functions to create Thread (Thread) networks and commission ("Commissioning" in "Thread") devices to join them.

This component is part of the Orchestrator (Orchestrator) firmware.

Pre-requisites:

- The steps in Project Bootstrap ("Project Bootstrap" in "Orchestrator") section are completed.
- System Initialization ("System Initialization" in "Orchestrator") is completed.

Development

Step 1: Create Component

The command below creates a new component named `thread_interface` inside the `components` directory.

```
idf.py create-component thread_interface -C components
```

The `CONFIG_OPENTHREAD_ENABLE` configuration option has to be set to `y` in the `sdkconfig.defaults` file in order to enable the OpenThread component.

Step 2: Change Source File Extension

Change the file extension of the `thread_interface.c` file to `thread_interface.cpp`.

```
mv components/thread_interface/thread_interface.c
components/thread_interface/thread_interface.cpp
```

Step 3: Manage Thread Network

The `thread_interface.h` file declares functions for managing the Thread network:

- **Initializing the Thread Interface**

  The initialization process involves the following steps:

  1. Configure event handlers to manage Thread network events.
  2. Set up the Event VFS (https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/vfs.html#eventfd), a lightweight mechanism that represents events as file descriptors, similar to Linux's eventfd. This allows tasks to signal and wait for events. The Event VFS is part of the broader VFS (https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/vfs.html) component, which provides a unified interface for various file systems.
  3. Creating a Dedicated FreeRTOS Task, declared in `thread_task.h`, using the `xTaskCreate` function. This task:
     - Initializes a new OpenThread network interface using `esp_netif`.
     - Initializes the OpenThread stack and attaches the network interface.
     - Sets up the OpenThread CLI ("OpenThread CLI" in "Thread")
     - Starts OpenThread.

- **Shutting Down the Thread Interface**
  - Stops the OpenThread stack.
  - Deletes the OpenThread task using `vTaskDelete`.
  - Unregisters the Event VFS.
  - Cleans up allocated semaphores and network interfaces.

- **Retrieving** and **Setting the Operational Dataset ("Active Operational Dataset" in "Thread")**

- **Getting the Device's Thread Role ("Device Roles" in "Thread")**:
  - Disabled – The device is not participating in the Thread network.
  - Detached – The device is trying to connect to a Thread network but is not yet

attached.

- Child – The device is connected to a Thread network but does not route traffic.
- Router – The device is forwarding messages for other Thread nodes.
- Leader – The device is managing the Thread network and ensuring proper operation.

OpenThread Configuration

Create a file that defines default configurations for OpenThread on ESP32-C6:

```
touch esp_ot_config_esp32c6.h
```

The code below sets up the network interface for Thread communication and configures the radio to use native mode. It also manages storage and task queues through the port configuration.

```
#pragma once

#include "esp_openthread_types.h"
#include <driver/uart.h>
#include <driver/gpio.h>
#include <esp_netif.h>

#ifdef __cplusplus
extern "C" {
#endif

// ------------------------------------------------------------------
---------
// Network Interface Configuration for OpenThread
// ------------------------------------------------------------------
---------
#define OT_NETIF_INHERENT_CONFIG() {                          \
    .flags         = (esp_netif_flags_t)0,                    \
    .mac           = { 0 },                                   \
    .ip_info       = { 0 },                                   \
```

```
    .get_ip_event  = 0,                                      \
    .lost_ip_event = 0,                                      \
    .if_key        = "OT_DEF",                              \
    .if_desc       = "THREAD_NETIF",                         \
    .route_prio    = 15                                     \
}

/* Declare a static inherent configuration instance */
static const esp_netif_inherent_config_t
ot_netif_inherent_config_instance = OT_NETIF_INHERENT_CONFIG();

#define OT_NETIF_CONFIG() {                                  \
    .base  = &ot_netif_inherent_config_instance,             \
    .stack = &g_esp_netif_netstack_default_openthread         \
}

// ------------------------------------------------------------------
---------
// OpenThread Radio Default Configuration
// ------------------------------------------------------------------
---------
#define ESP_OPENTHREAD_DEFAULT_RADIO_CONFIG()              \
{                                                       \
.radio_mode = RADIO_MODE_NATIVE                    \
}

// ------------------------------------------------------------------
---------
// OpenThread Host Default Configuration
// ------------------------------------------------------------------
---------
#if CONFIG_OPENTHREAD_CONSOLE_TYPE_UART
#define ESP_OPENTHREAD_DEFAULT_HOST_CONFIG()                   \
    {                                                          \
        .host_connection_mode = HOST_CONNECTION_MODE_NONE,  \
    }
#elif CONFIG_OPENTHREAD_CONSOLE_TYPE_USB_SERIAL_JTAG
#define ESP_OPENTHREAD_DEFAULT_HOST_CONFIG()                       \
```

```
    {                                                                   \
        .host_connection_mode = HOST_CONNECTION_MODE_CLI_USB,           \
        .host_usb_config = USB_SERIAL_JTAG_DRIVER_CONFIG_DEFAULT()    \
    }
#endif

// ------------------------------------------------------------------
---------
// OpenThread Port Default Configuration
// ------------------------------------------------------------------
---------
#define ESP_OPENTHREAD_DEFAULT_PORT_CONFIG()    \
    {                                           \
        .storage_partition_name = "nvs",        \
        .netif_queue_size = 10,                 \
        .task_queue_size = 10                   \
    }

// ------------------------------------------------------------------
---------
// OpenThread Default Configuration
// ------------------------------------------------------------------
---------
#define ESP_OPENTHREAD_DEFAULT_CONFIG()    \
    {                                       \
    .radio_config = ESP_OPENTHREAD_DEFAULT_RADIO_CONFIG(),        \
    .host_config = ESP_OPENTHREAD_DEFAULT_HOST_CONFIG(),
\
    .port_config = ESP_OPENTHREAD_DEFAULT_PORT_CONFIG()
\
    }

#ifdef __cplusplus
}
#endif
```

References

OpenThread SDK:

- How to Write an OpenThread Application (https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/openthread.html#how-to-write-an-openthread-application)

Android SDK

- Android Thread Commissioner (https://github.com/openthread/ot-commissioner/tree/main/android)
- Thread Network SDK for Android (https://developers.home.google.com/thread)

Wi-Fi Interface

Overview

This section outlines the development of the **Wi-Fi-STA Interface** component, which provides the features and functions to connect to Wi-Fi networks and manage the device's Wi-Fi settings.

This component is part of the Orchestrator (Orchestrator) firmware.

Development

Bootstrapping the Component

1. Creates a new component named `wi_fi_sta_interface` inside the `components` directory.

   ```
   idf.py create-component wi_fi_sta_interface -C components
   ```

2. Rename and move the source file.

   1. Rename the `wi_fi_sta_interface.c` to `wi_fi_sta_interface.cpp`.
   2. Move it to src folder.

3. Update the `CMakeLists.txt` file.

   1. Change the location of the `wi_fi_sta_interface.cpp` file to the `src` folder.
   2. Add required dependencies `esp_event`, `esp_netif`, and `esp_wifi` to the REQUIRES section.

   The `CMakeLists.txt` file should look like this:

   ```
   idf_component_register(SRCS "src/wi_fi_sta_interface.cpp"
                          INCLUDE_DIRS "include"
                          REQUIRES esp_event esp_netif esp_wifi)
   ```

References

ESP Wi-Fi Docs (https://docs.espressif.com/projects/esp-idf/en/v5.2.5/esp32c6/api-reference/network/esp_wifi.html)

Event Handlers

The Orchestrator uses event handlers to respond to system-level events, categorized under specific **event bases**:

- **Wi-Fi Events** (`WIFI_EVENT`) are triggered when the device connects, disconnects, or starts in STA mode.
- **Thread Events** (`THREAD_EVENT`) from the OpenThread stack for events like Thread role changes, network state, or dataset updates.
- **CHIP Events** (`CHIP_EVENT`) from the Matter stack for events such as commissioning completion or attribute changes.

Each handler subscribes to its respective event base using the ESP-IDF event loop system. When an event occurs, the system calls the registered callback function automatically.

Messages

Overview

The inbound message handler (https://github.com/albert-gee/old_macdonald_orchestrator/blob/main/main/src/json/json_inbound_message.cpp#L170) is passed to the **Websocket Server** (Websocket Server) component when it is initialized. This function is automatically called whenever a message is received from a client. It parses the JSON message, verifies the `type` and `action` fields, and then forwards the request to the appropriate command handler (https://github.com/albert-gee/old_macdonald_orchestrator/blob/main/main/src/commands/commands.cpp).

Outbound messages (https://github.com/albert-gee/old_macdonald_orchestrator/blob/main/main/src/json/json_outbound_message.cpp) are generated by the Orchestrator in response to events or command results and are sent to all connected clients.

Format

Messages use JSON-encoded UTF-8 format and follow this structure:

Field	Type	Description
`type`	string	Type of message: – `"command"` is sent **from client** to request an action – `"info"` is sent **from device** as a response or update
`action`	string	Describes the specific command or event
`payload`	object	Contains the relevant data for the given action

Example of a command from Client:

```
{
  "type": "command",
  "action": "wifi.sta_connect",
  "payload": {
    "ssid": "MyWiFi",
    "password": "pass1234"
  }
}
```

Example of info from Orchestrator:

```
{
  "type": "info",
  "action": "wifi.status",
  "payload": {
    "status": "connected"
  }
}
```

JSON Data

The `cJSON` library is used to parse and generate JSON data.

It is part of the ESP-IDF components. `json` should be added to the `REQUIRES` section of the `CMakeLists.txt` file.

Usage:

```
# Include cJSON in source code:
#include "cJSON.h"

# Parsing JSON:
cJSON *root = cJSON_Parse(my_json_string);
cJSON *format = cJSON_GetObjectItem(root, "format");
int framerate = cJSON_GetObjectItem(format, "frame rate")->valueint;
cJSON_Delete(root); // Clean up after use

# Creating JSON:
```

```
cJSON *root = cJSON_CreateObject();
cJSON *fmt = cJSON_CreateObject();

cJSON_AddItemToObject(root, "name", cJSON_CreateString("Jack (\"Bee\")
Nimble"));
cJSON_AddItemToObject(root, "format", fmt);
cJSON_AddStringToObject(fmt, "type", "rect");
cJSON_AddNumberToObject(fmt, "width", 1920);
cJSON_AddNumberToObject(fmt, "height", 1080);
cJSON_AddFalseToObject(fmt, "interlace");
cJSON_AddNumberToObject(fmt, "frame rate", 24);

char *json_str = cJSON_Print(root);
printf("%s\n", json_str);

cJSON_Delete(root); // Clean up

# Manual Traversal:
void parse_object(cJSON *item) {
    cJSON *subitem = item->child;
    while (subitem) {
        printf("Key: %s\n", subitem->string);
        subitem = subitem->next;
    }
}
```

Dashboard

Overview

The **Dashboard** is a GUI application that farmers use to configure and manage system components, as well as monitor and adjust environmental conditions.

The environment can be configured for specific crops. When setting up a new plant, farmers can choose from predefined parameters suited to that crop. As the plant grows, they can manually adjust settings.

Source code: GitHub (https://github.com/albert-gee/old_macdonald_dashboard)