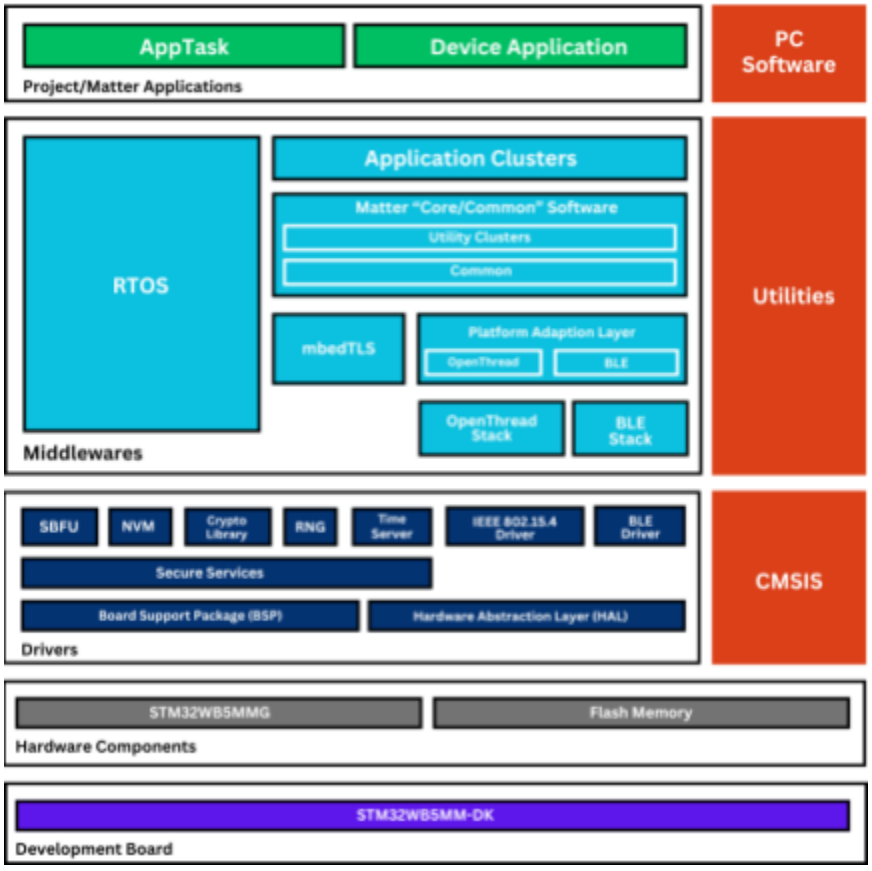
STM32 Matter over Thread

Developing an STM32 Matter Smart Home App with Environment Sensors: A Comprehensive Guide



Uche Nwogbo
Thomas More University
Elektronica Intern, Spring 2025
4th April 2025

Table of Contents

1. Introduction

As smart home technology evolves, seamless device interoperability, low-power communication, and secure networking have become essential. This guide provides a comprehensive walkthrough for developing an STM32 Matter over Thread smart home application, integrating environmental sensors for real-time data monitoring. By leveraging Matter for standardized smart home communication and Thread for reliable wireless connectivity, this project ensures effortless integration with Google Home and other Matter-compatible ecosystems.

This documentation takes a bottom-up approach, covering STM32 hardware configuration, Matter protocol stack setup, and Thread network integration, along with sensor interfacing and commissioning. By building the application from scratch, developers gain a deeper understanding of Matter device architecture, enabling better customization, debugging, and scalability. Whether for personal projects or commercial smart home solutions, this guide is a solid foundation for creating a robust and interoperable Matter over Thread device.

1.1. Matter Protocol

Matter is an open-source connectivity standard developed by the Connectivity Standards Alliance (CSA), formerly known as the Zigbee Alliance, to enhance interoperability among smart home devices from different manufacturers. The standard was created in 2019 by Amazon, Apple, and Google and was named Project Connected Home over IP or Project CHIP. It operates over IP-based networks, such as Ethernet, Wi-Fi, and Thread, ensuring a unified application layer for secure and reliable communication [1]. This standardization simplifies integration across various platforms and ecosystems.

The Matter stack consists of multiple layers that delineate responsibilities across the stack, promoting modularity and facilitating the development of interoperable smart home solutions[2].

- **Application Layer**: Encompasses application-specific functionalities and user interfaces built atop the Matter protocol.
- **Data Model Layer**: Defines the structure of data and actions supporting application functionality, including elements like endpoints, clusters, and attributes.
- **Interaction Model Layer**: Manages the interactions between nodes, dictating how data is transferred within the network.
- **Action Framing Layer**: Transforms interactions into message payloads for transmission.
- **Security Layer**: Encrypts payloads and appends a Message Authentication Code (MAC) to ensure data integrity and confidentiality.
- **Messaging Layer**: Handles data transmission over various network types, including Ethernet, Wi-Fi, and Thread.
- **Transport Layer**: Handles the transmission of the payload over IP networks using protocols like TCP or Matter's Message Reliability Protocol atop UDP.

1.2. Thread Protocol

Thread is an IPv6-based, low-power, secure, and reliable mesh networking protocol specifically designed for IoT applications. Developed by the Thread Group, it is built on the IEEE 802.15.4 standard, operating at 2.4 GHz (for link layer communication) with data rates up to 250 kbps [3]. OpenThread is the implementation of the Thread protocol used in this App, It is designed to run on resource-constrained embedded devices, making it an ideal choice for STM32-based Matter applications. STM32 integrates OpenThread to enable seamless wireless connectivity between smart home devices and the Border Router, ensuring interoperability and efficient communication in IoT ecosystems.

In a Thread network, devices assume specific roles[4]:

- **Router**: It transmits, receives, and forwards messages within the network. They also provide joining and security services for devices trying to join the network.
- **End Device**: It communicates with a single router without forwarding messages. There are several types based on their communication capacity, REED (Router-eligible End Device), FED (Full End Device), MED (Minimal End Device), SED (Sleepy End Device), SSED (Synchronized Sleepy End Devices), and BED (Bluetooth End Device).
- **Leader**: It is an elected role of one Router, which makes certain decisions in the Thread network such as allowing REEDs to upgrade to Routers.
- **Border Router**: It connects the Thread Network to adjacent networks on other physical layers like Wi-Fi or Ethernet.

Think of Thread as the ability to talk and hear in a conversation it allows devices to communicate with each other in a structured, efficient way, forming a reliable mesh network. Matter, on the other hand, is the language and rules of the conversation it defines how messages are structured, understood, and processed, ensuring that different devices, even from different manufacturers, can interact seamlessly. Just as a conversation needs both a way to transmit speech (Thread) and a common language to be meaningful (Matter), smart home devices need Thread for connectivity and Matter to standardize communication.

2. Step-by-Step Guide to Building the Matter Smart Home App

2.1. From Scratch Approach

This section covers the end-to-end process of developing a Matter-based smart home application from scratch. It outlines the key steps, including setting up the development environment, configuring the STM32 board, generating code with Matter tools, and integrating the BME280 sensor. The final steps involve building, flashing, and testing the application to ensure proper functionality within a Matter and Thread network.

2.1.1. Setting Up the Development Environment

1. **Install STM32CubeIDE**: Download and install the latest version of STM32CubeIDE from STMicroelectronics' official website

2. **Install CLI**: Download and install any command line interface of choice (Git Bash preferred) to facilitate command-line operations.
3. **Clone the Matter SDK**: Open Git Bash and execute the following commands to clone the Matter SDK repository:

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

   ☑ `cd connectedhomeip`

   ☑ `git submodule update --init`
4. **Install Python and Dependencies**: Ensure Python is installed, then install the required packages:

   ☑ `python3 -m pip install -r requirements.txt`
5. **Install ZAP (ZCL Advanced Platform)**: ZAP is used for generating Matter data models. Follow the instructions provided in the Matter SDK documentation to install ZAP.

2.1.2. Board Configurations

1. **Open STM32CubeIDE**: Launch STM32CubeIDE and create a new STM32 project for your specific microcontroller.
2. **Create a New STM32CubeIDE Project**

   ☑ Open STM32CubeIDE and select “File > New > STM32 Project”.

   ☑ Choose your STM32 microcontroller or development board (e.g., STM32WB5MM-DK).

   ☑ Click Next, give your project a name, and select Empty Project (with HAL) as the template.

   ☑ Click Finish to create the project.
3. **Configure System Core**: Select configurations based on your app features and specifics, for a simple Matter app you need IPCC, SYS, and RCC activated as seen below:

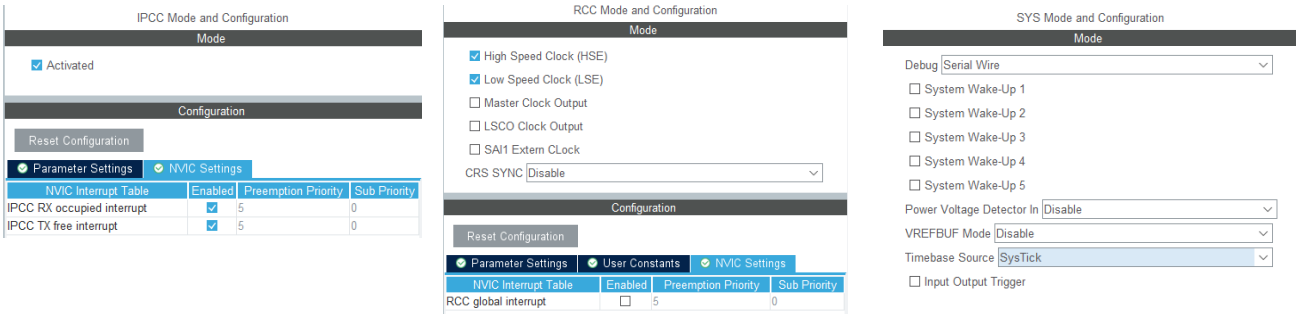


Figure 1: Screenshots from the IPCC, RCC, and SYS configurations in STM32CubeMX

4. **Configure Timers**: Choose peripherals based on your app features and specifics, for a simple Matter app you only need RTC activated as seen below:

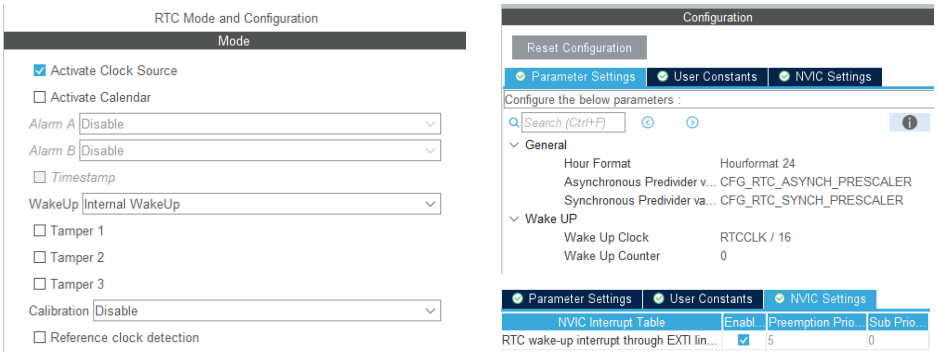


Figure 2: Screenshots from the RTC configuration in STM32CubeMX

5. **Configure Connectivity**: Choose peripherals based on your app features and specifics, for example, the BME280 sensor uses I2C protocol, and data (for debugging) is displayed in real-time via a serial wire terminal. RF controls the BLE and Thread radio, this is essential for commissioning Matter devices. LPUART can be used instead of UART if there are power constraints:

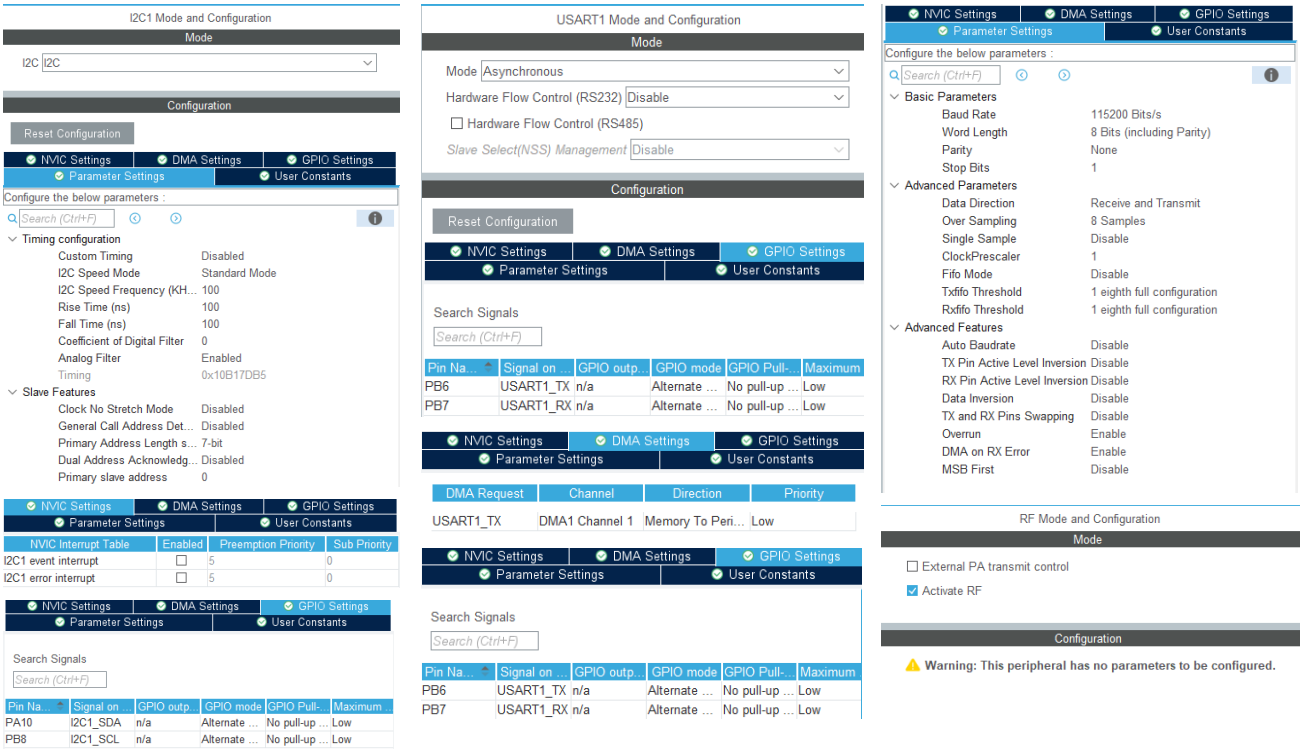


Figure 3: Screenshots from the I2C1 and USART1 configurations in STM32CubeMX

6. **Configure Security**: RNG (Random Number Generator) generates random keys for securely pairing and commissioning in Matter, this encryption prevents device cloning and unauthorized network access. Matter uses message authentication codes (MACs) and session tokens, which require random seeds to prevent replay attacks.

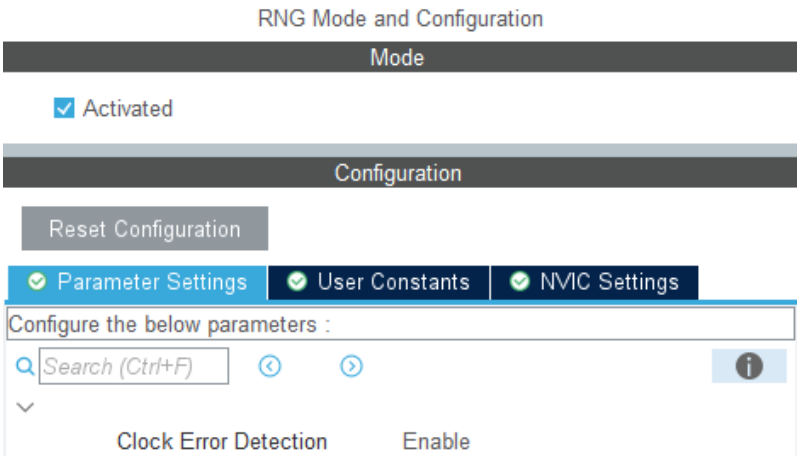


Figure 4: Screenshot from RNG configuration in STM32CubeMX

7. **Configure RTOS**: *FREERTOS* is recommended because Matter requires efficient multitasking, real-time responsiveness, and memory management to handle multiple concurrent operations. For a simple Matter app, you can have RTOS tasks for communication and sensor handling (receiving and transmitting data), highest priority should be given to whatever task handles Matter protocol communication (initialization and app tasks handling).

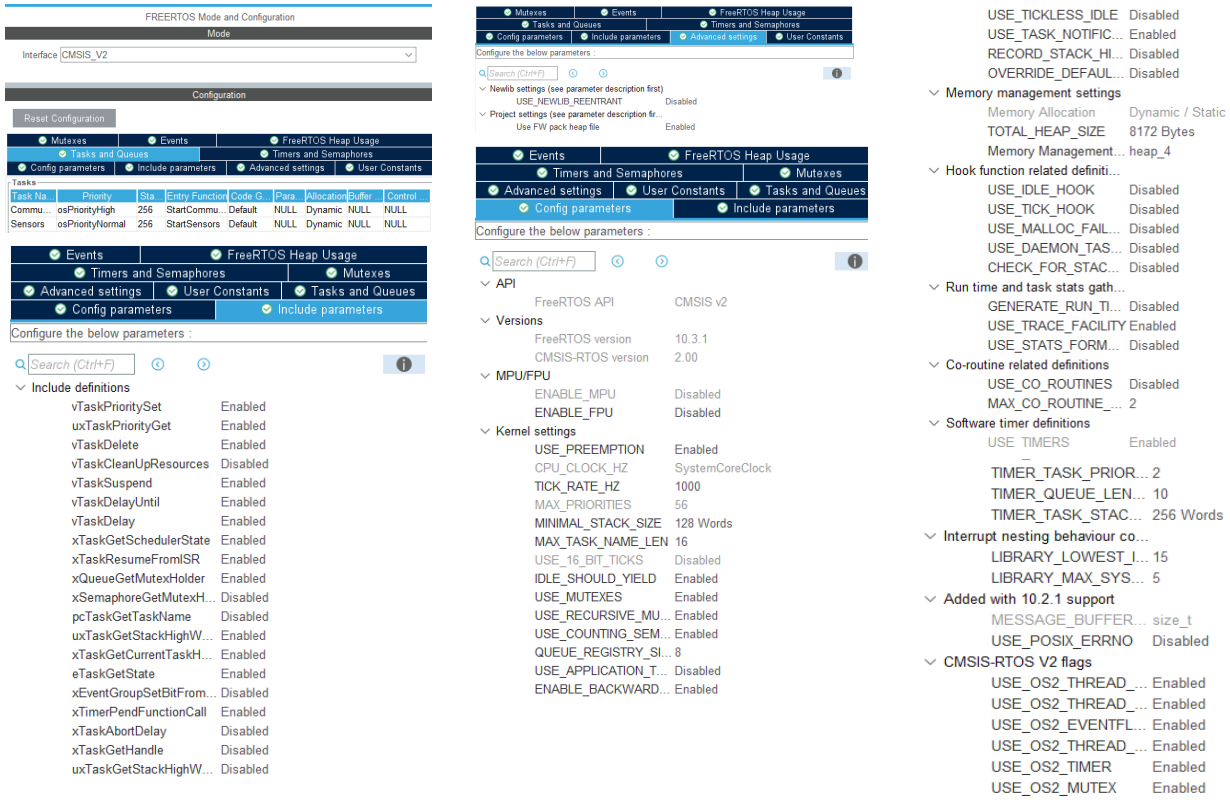


Figure 5: Screenshots from the FreeRTOS configuration in STM32CubeMX

8. **Configure Thread**: The *STM32_WPAN* (Wireless Personal Area Network) middleware provides integration for OpenThread, allowing devices to form a low-power, IPv6-based mesh network. This setup includes

enabling the Thread stack with the appropriate stack size, enabling certain parameters (in-built user constants), and choosing the correct type of Thread application (FTD with network or CLI and MTD).

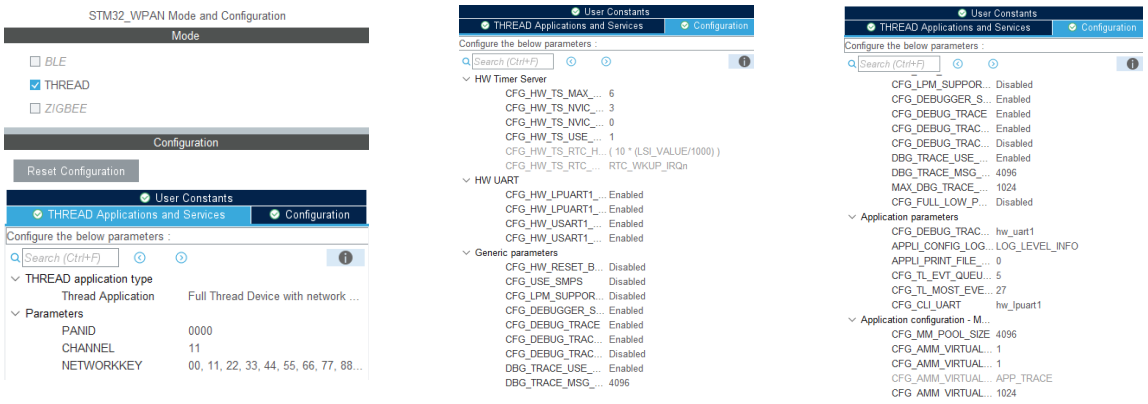


Figure 6: Screenshots from the STM32_WPAN Thread configuration in STM32CubeMX

Take note of the default parameters for PANID, channel number, and network key; they determine the radio frequency used for communication and security credentials that encrypt and authenticate messages within the Thread network.

9. **Configure Utilities**: *TINY_LPM* (Tiny Low-Power Manager) is recommended for low-power Matter apps, disable if this is not applicable. It manages low-power states in FreeRTOS without interfering with real-time tasks, puts the MCU in low-power modes (like standby), and allows the device to wake up efficiently when receiving Matter or Thread messages.
10. **Generate Code**: After all board peripherals are configured correctly, double-check that Thread is enabled under *STM32_WPAN* and that *I2C, UART, RNG*, and *GPIO*s are properly configured.

    ☑ Click “Generate Code” at the top right.

2.1.3. Integrating Matter SDK

1. **Download X-CUBE-MATTER Expansion Package**

    ☑ Visit the STMicroelectronics website and download the *X-CUBE-MATTER* expansion package, which includes the Matter SDK pre-integrated for STM32 microcontrollers.

    ☑ Extract the downloaded *X-CUBE-MATTER* package to a known directory on your system.

2. **Copy Required Folders into Your Project:** Navigate to the extracted *X-CUBE-MATTER* package and copy the following folders into your STM32CubeIDE project directory

    ☑ Middlewares → Copy *ST* and *Third_Party* folders (contain the Matter stack and dependencies) into your project.

    ☑ Utilities → Copy *lpm, Fonts, LCD,* and any other relevant utilities (based on your app features) into your project.

    ☑ Drivers → Copy *STM32WBxx_HAL_Driver, CMSIS,* and *BSP* into your project.

Some of these subfolders might have already been generated by STM32CubeIDE, use board-specific/automatically generated ones instead.

3. **Verify Project Structure:** Your project directory, paths, and symbols should now look like the figures below.

   ☑ To add include paths got to *Properties* > *C/C++ General* > *Paths and Symbols* > *Includes* > *GNU C/C++* > *Add*

   ☑ To add Symbols paths got to *Properties* > *C/C++ General* > *Paths and Symbols* > *Symbols* > *GNU C++* > *Add*

   ☑ Add the cloned connectedhomeip folder.

   ☑ Click "Apply and close"

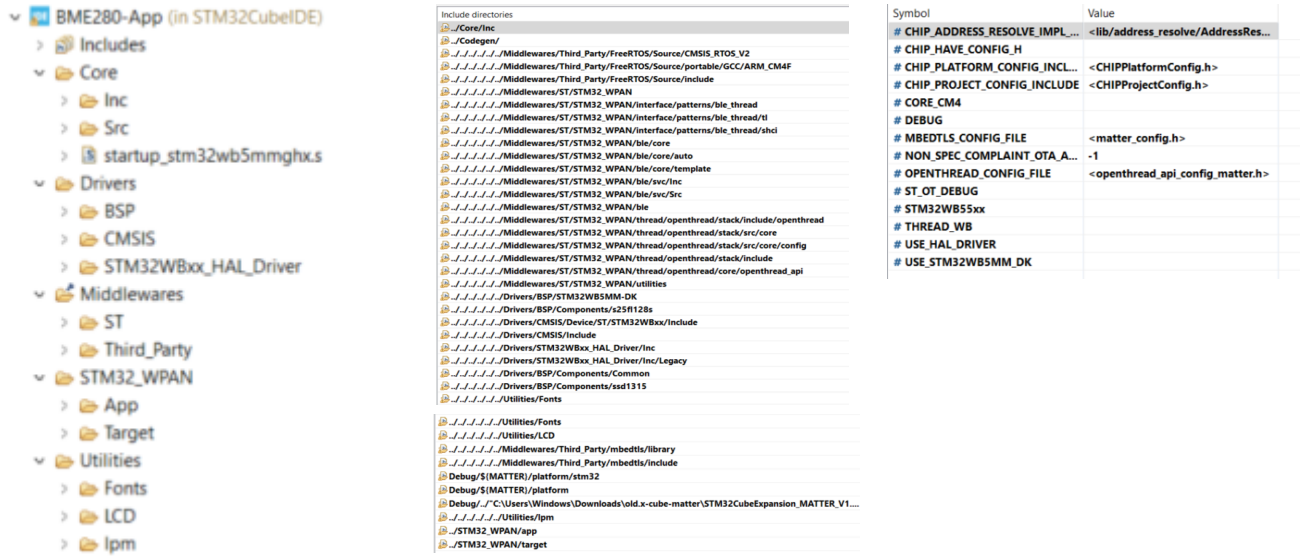


Figure 7: Screenshots from the paths and symbols under project properties in STM32CubeIDE

To ensure the Matter SDK and Thread stack work correctly within your STM32 project, you need to define specific preprocessor symbols. These symbols configure the compiler to include the right header files, enable debugging features, and ensure correct platform-specific configurations.

2.1.4. Generating Code with ZAP

1. **Setup the Environment**: Configure your environment to work with the ZAP tool. Open a CLI and run the following commands to upgrade `pip` and install necessary Python packages.

   ☑ `python -m pip install --upgrade pip`

   ☑ `pip install portalocker click lark jinja2 stringcase coloredlogs`

If you're using a Windows system, modify the *generate.py* file in the ZAP tool to replace the `fcntl` module with portalocker, as `fcntl` is not available on Windows.

2. **Generate ZAP Configuration**: Create a new ZAP configuration for your Matter device. In your CLI, export the ZAP installation path and change it to the matter directory.

   - ☑ `export ZAP_INSTALL_PATH=C:/path/to/zap-win-x64`
   - ☑ `cd ../../C:/connectedhomeip`

3. **Launch ZAP Tool**: Navigate to the Matter directory and launch the ZAP tool using an example project:

   - ☑ `cd /path/to/connectedhomeip`
   - ☑ `./scripts/tools/zap/run_zaptool.sh`

4. **Configure Device in ZAP GUI**

   - ☑ Select a device or endpoint from the device library.
   - ☑ Enable mandatory and optional clusters for the chosen endpoint.
   - ☑ Configure attributes and commands for each cluster as needed
   - ☑ Save your configuration as a new *.zap* file, e.g., *my_device.zap*
   - ☑ Click the "Generate" button in the ZAP interface to produce the initial source files.

5. **Save Files in Accessible Directory**: To keep everything organized well, manually move the generated files to a separate folder in the *zap_win_x64* folder.

   - ☑ `mv zap-generated/* ../zap_gen_files/<project_name>/`
   - ☑ `ls ../zap_gen_files/<project_name>/`

6. **Generate Matter-Specific Code**: Convert the *.zap* file into a *.matter* file, generate Matter-specific code, and generate callback definitions.

   - ☑ `python ./scripts/tools/zap/generate.py ../zap_gen_files/my_device/my_device.zap -o ../zap_gen_files/my_device/`
   - ☑ `python ./scripts/codegen.py --generator cpp-app --output-dir ../zap_gen_files/my_device/ ../zap_gen_files/my_device/my_device.matter`

7. After all these steps are completed, your *zap_gen_files* folder should contain these files.

   - ☑ **Matter and ZAP configuration files**: my_device.zap and my_device.matter
   - ☑ **Initial ZAP-generated files, specific to endpoint and cluster configurations**: access.h, endpoint_config.h, gen_config.h and IMClusterCommandHandler.cpp
   - ☑ **Matter callback files generated from *.matter* configuration**: callback-stub.cpp, cluster-init-callback.cpp and PluginApplicationCallbacks.h

2.2. From Existing Matter Project

Using existing Matter projects from *X-CUBE-MATTER* can significantly expedite the development process by providing a solid foundation to build. This approach allows developers to focus on customizing and extending functionalities rather than starting from scratch. The following sections outline the steps to import an existing STM32 Matter project, edit and generate code using the ZCL Advanced Platform (ZAP) tool, and modify the project to integrate additional features.

2.2.1. Importing and Understanding the STM32 Matter Project

1. **Download X-CUBE-MATTER Expansion Package**
   - ☑ Visit the STMicroelectronics website and download the *X-CUBE-MATTER* expansion package, which includes the Matter SDK pre-integrated for STM32 microcontrollers.
   - ☑ Extract the downloaded *X-CUBE-MATTER* package to a known directory on your system.
2. **Import the Existing Project**
   - ☑ Navigate to “File > Import > General > Existing Projects into Workspace” and click “Next”.
   - ☑ Click “Browse” and select the directory containing the extracted *X-CUBE-MATTER* projects: “X-CUBE-MATTER/Projects/STM32WB5MM-DK/Applications/Matter”.
   - ☑ Choose the specific application STM32CubeIDE project you wish to import (e.g., Lighting-App) and click “Finish”.
3. **Explore the Project Structure**
   - ☑ Familiarize yourself with the project's folders and files, noting key directories such as “Core”, “Drivers”, “Middlewares”, and “STM32CubeIDE”.
   - ☑ Verify that all the files in paths and symbols are present in the STM32CubeIDE project or *X-CUBE-MATTER* directory.
4. **Build and Run the Sample Project**
   - ☑ Follow the steps in sections 2.2 and 2.3 to build, run, and connect to the border router.
   - ☑ Take note of how the code runs, try out the user interface

2.2.2. Editing and Generating Code with ZAP

1. **Install ZAP Tool:** The ZAP (ZCL Advanced Platform) tool is used to generate code for Matter applications based on the Zigbee Cluster Library. Download the latest ZAP release from the GitHub repository.
   - ☑ `git clone https://github.com/project-chip/zap.git`
   - ☑ `cd zap`
   - ☑ `npm install`

- ☑ `npm run build`

2. **Launch ZAP:** Start the ZAP tool by running, this command opens the ZAP graphical interface in your default web browser.

   - ☑ `npm start`

3. **Download the Connected Home IP Repository and ZAP Tool:** Before running the ZAP tool, export the installation path in Git Bash.

   - ☑ Choose a location on your system to store the tools, e.g., "c:\chip-matter-tools".
   - ☑ Navigate to your chosen directory and download the repository:

     ```
     cd /c/chip-matter-tools
     ```

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

   - ☑ Navigate to the ZAP releases page and download the version that is compatible with your *X-CUBE-MATTER* package (latest version). Extract it to a subfolder within your tools directory, such as *zap-win-x64*
   - ☑ `export ZAP_INSTALL_PATH=C:/chip-matter-tools/zap-win-x64`

4. **Navigate to the Matter Project Directory:** Change to the Matter repository directory where the existing project is located and verify the *.zap* file exists.

   - ☑ `cd ../../C:/chip-matter-tools/connectedhomeip`
   - ☑ `ls examples/lighting-app/lighting-common/`

The expected output should contain a *.zap* file, such as *lighting-app.zap*.

5. **Launch the ZAP GUI:** Start the ZAP tool from an existing Matter project.

   - ☑ `./scripts/tools/zap/run_zaptool.sh examples/lighting-app/lighting-common/lighting-app.zap`

If there is an error with this script, try:

- ☑ Go to "connectedhomeip/examples/lighting-app/lighting-common", open *lighting-app.zap* with zap.exe

6. **Edit Clusters and Attributes in ZAP**

   - ☑ Add or remove clusters based on your application requirements.
   - ☑ Configure cluster attributes (e.g., measurement intervals, supported units).
   - ☑ Click "File > Save" to store the modified *.zap* file.

7. Generate Updated ZAP Code: Convert the *.zap* file to a *.matter* file and generate callback definitions for clusters.

   - ☑ `python ./scripts/tools/zap/generate.py ../zap_gen_files/my_light/light.zap -o ../zap_gen_files/my_light/`

- [x] `python ./scripts/codegen.py --generator cpp-app --output-dir ../zap_gen_files/my_light/ ../zap_gen_files/my_light/light.matter`

8. **Final Verifications**: Copy the generated files to the correct directory in your STM32 Matter project and verify all required files exist.

   - [x] `cp -r ../zap_gen_files/my_light/* Middlewares/Third-Party/connectedhomeip/devices/Thread/MyMatterDevice/`
   - [x] `ls Middlewares/Third-Party/connectedhomeip/devices/Thread/MyMatterDevice/`

2.3. Customizing Project-Specific Code

After generating the code in STM32CubeIDE, you need to edit and create key files to implement the features of your Matter-over-Thread Smart Home Application. Below is a general overview of the files that need to be modified or created, along with what needs to be added.

1. **Modify Entry Point of the Application**: The *main.c* file generated by STM32CubeIDE primarily initializes hardware peripherals, FreeRTOS, and starts Matter and Thread tasks. However, additional modifications are required to integrate specific application features. Add calls to initialize your chosen sensor, include UART print statements for debugging, and configure special GPIO instructions if needed. Since FreeRTOS handles task execution, any initialization required for sensor communication, Matter networking, or debugging should be placed before the RTOS scheduler is started. Notably, after calling `osKernelStart()`, the program no longer executes code inside the while loop in the *main.c*, making any instructions placed there ineffective.
2. **Create Main Application Logic**: The main application logic is implemented in *AppTask.cpp* and *AppTask.h*, which handle device state management, event processing, and Matter interactions. This file is responsible for polling sensor data, handling user interactions (such as button presses triggering Matter events), and updating the device's operational state. To ensure a structured approach, a state machine should be implemented to manage different device states, such as initializing, connecting, or handling errors. Event-driven execution should be prioritized to prevent blocking operations, ensuring that Matter and Thread tasks run efficiently. Debugging should be incorporated using `ChipLogProgress()` statements, which help track the device's state transitions and interactions with Matter controllers. Additionally, failure-handling mechanisms should be implemented—for example, if the sensor disconnects, the system should retry communication instead of failing outright, ensuring robustness in real-world deployments.
3. **Create Matter Cluster Event Handler**: The *ZclCallbacks.cpp* file is responsible for handling Matter cluster events. This file must be modified to define callback functions for each configured cluster in ZAP (Zigbee Cluster Application). The implementation should include responding to Matter commands, processing incoming requests, and updating sensor readings accordingly. Handling Matter commissioning events is crucial—this determines when and how the device joins the Thread network and interacts with a Matter controller. Additionally, sensor data should be formatted properly before transmission to ensure compatibility with Matter ecosystems. If the application includes actuators (such as LEDs or relays), their behavior must be linked to the appropriate GPIO control functions.
4. To verify that the required callbacks from the *PluginApplicationCallbacks.h* file exist, check the *X-CUBE-MATTER* middleware in the directory: "Middlewares\Third_Party\connectedhomeip\src\app\clusters".

If certain callback functions are missing, they should be manually created and added to the project directory to ensure full Matter cluster support.

5. **Modify Application-Specific Configuration**: The *app_config.h* file defines key device parameters, Matter settings, and Thread configurations. It is essential to set the PANID, Thread channel, and network key required for the device to join a Thread network. This configuration data is derived from the Thread settings in *STM32_WPAN* and should match the parameters set in your Matter ecosystem. Additionally, sensor update intervals should be defined, ensuring optimal polling frequency without excessive power consumption. For low-power applications, power-saving modes should be enabled or disabled based on device constraints. Conditional compilation (`#ifdef`) can be used to toggle debugging features and low-power settings, allowing flexibility in different development and deployment scenarios.
6. **Modify Network Commissioning Code**: Network commissioning is handled within the Matter and Thread network files copied from *X-CUBE-MATTER*. The key files to edit include *app_ble.c*, *app_matter.c*, and *app_thread.c*, which govern BLE pairing, Matter commissioning, and Thread network configuration. BLE is used for initial device setup, allowing a Matter controller (such as Google Home or Apple TV) to discover and onboard the device. The BLE advertisement and pairing logic should be properly implemented to ensure seamless commissioning. Once the device is successfully paired, *app_thread.c* must handle network communication, ensuring that the device remains connected to the Thread Border Router. Debugging statements should be included in these files to monitor network join events, status updates, and reconnection attempts. Proper handling of commissioning failures is crucial—if a pairing attempt fails, the device should retry or provide feedback (such as an LED blink pattern) to indicate the failure state.

Beyond modifying the core Matter and Thread files, you can add or edit additional files to optimize your application's performance, security, and scalability. For instance, if your device interacts with multiple sensors or actuators, you might want to create dedicated driver files (e.g. *bme280.c* and *bme280.h*) to encapsulate sensor-specific logic, keeping the main application files clean and modular. Implementing a custom event handler can improve responsiveness by allowing event-driven processing instead of relying solely on periodic polling. If your application requires Over-The-Air (OTA) updates, consider adding firmware update logic to handle secure downloads, verification, and seamless firmware flashing without disrupting normal operation. For debugging and monitoring, a diagnostics module can log real-time Matter, Thread, and sensor data, optionally storing logs in Non-Volatile Memory (NVM) for post-mortem analysis. If security is a concern, incorporating cryptographic utilities can help encrypt data before transmission and authenticate network communications. Lastly, if your device supports multiple communication protocols (e.g., BLE + Thread), a protocol abstraction layer can help manage different connectivity modes efficiently, ensuring smooth interoperability in diverse IoT environments.

2.4. Compiling and Flashing the Firmware

To successfully compile, flash, and debug your Matter-over-Thread STM32 application, follow these steps. This process ensures that your ST-Link is updated, your firmware is built correctly, and the device is programmed successfully. There are two ways of doing this; first using STM32CubeProgrammer (GUI or CLI) to flash the binaries (after building and debugging) and manually doing any firmware upgrades. The second method is more straightforward, using STM32CubeIDE to perform all these seamlessly from the GUI.

2.4.1. Using STM32CubeIDE

1. Check ST-Link Connection and Update

☑ Go to “Run > Debug Configurations”.

☑ Under Debugger, select ST-LINK (OpenOCD).

☑ Click “Show ST-Link Status” to verify the debugger is detected.

☑ If no ST-LINK is detected, click “Scan” and cross-check that the key corresponds to that of your device.

☑ Navigate to “Help > ST-Link Upgrade”.

☑ Click “Open in update mode”

☑ Click “Upgrade”

☑ Once done, disconnect and reconnect the **ST-Link USB cable** to apply changes.

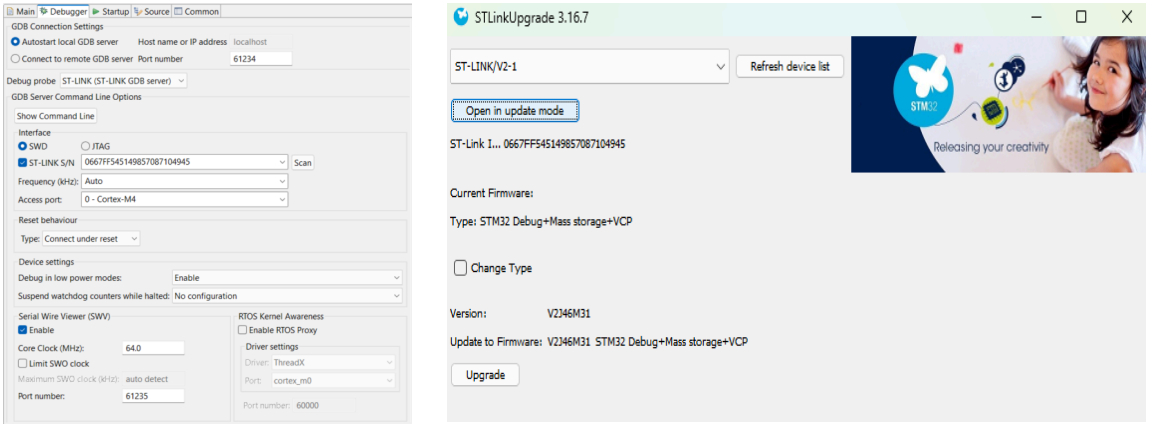


Figure 8: Screenshots from the debug configurations and ST-Link Upgrade in STM32CubeIDE

2. Compile the Matter Firmware in STM32CubeIDE

☑ Click “Project > Build Project” (or press `Ctrl + B`) and wait for the compilation to complete and verify that the firmware binary (*.bin*) is generated.

☑ After the build is complete, click “Run > Debug” (or press `F11`), STM32CubeIDE will ask to switch to the Debug Perspective—click Yes.

☑ Wait till debugging is complete and click “Run > Run” (or press `Ctrl + F11`).

3. **Opening UART Terminal**: For any Serial Wire output in your code, you need an external terminal viewer like PuTTy or TeraTerm (TeraTerm preferred).

☑ Open TeraTerm and select “Serial” in the connection window.

- ☑ Verify that the COM port number is correct for your ST-Link USB, if unsure, open Device Manager (Windows) and check under Ports (COM & LPT) for the correct port number. Choose from the dropdown menu and click “OK”
- ☑ Set the correct Baud Rate, Go to “Setup > Serial Port” and then Click “New setting”.
- ☑ Press the “reset” button on the STM board.

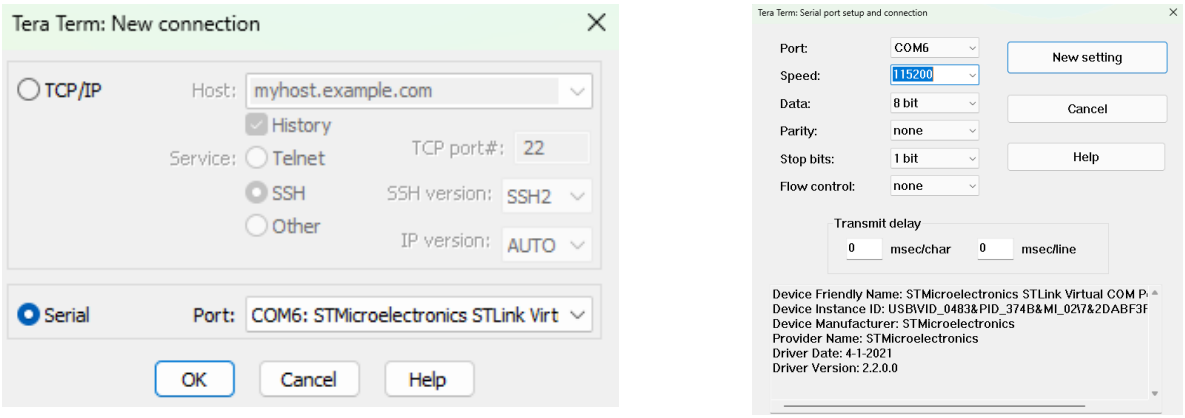


Figure 9: Screenshots from the TeraTerm setup to view serial wire output

2.4.2. Using STM32CubeProgrammer

1. **Verify ST-LINK Firmware Version**
   - ☑ Open STM32CubeProgrammer.
   - ☑ Under ST-LINK Configuration click "Firmware Upgrade" to check and update if necessary.
2. **Program the Cortex®-M0+ Core (Wireless Stack):** The Cortex®-M0+ core handles the wireless protocol stack essential for Matter over Thread applications.
   - ☑ Connect the STM32WB board to the PC via USB and click “Connect”
   - ☑ Click "Read FUS Version" to verify the current version.
   - ☑ If the FUS is outdated, update it by flashing the latest FUS firmware found in the *X-CUBE-MATTER* directory "...\Projects\STM32WB_Copro_Wireless_Binaries\stm32wb5x_FUS_fw.bin" and set the “Start address” as *0x080EC000*. Click "Start Programming" and wait for completion.
   - ☑ Flash the Thread Stack using the binary file *stm32wb5x_Thread_light_dynamic_fw.bin* and set the start address as *0x08096000.* Select “First install” No stack delete” and then click “Firmware Upgrade”

Always program the Cortex-M0+ core before the Cortex-M4 core. If the Cortex-M0+ is reprogrammed after the Cortex-M4, you must reprogram the Cortex-M4 core to maintain synchronization.

3. **Flashing the Matter Application (Cortex®-M4)**: Once the wireless stack is updated, the Matter firmware can be flashed onto the Cortex®-M4 core.

- ☑ In STM32CubeProgrammer, select the compiled Matter firmware binary from your project inside the “Debug” folder in STM32CubeIDE.
- ☑ Set the start address to 0x08000000.
- ☑ Click “Start Programming” and wait for completion.
- ☑ After flashing, reset the board and open the serial terminal as done above.

Once the firmware is successfully built and running on your STM32 Matter device, the UART terminal will display real-time logs showing the initialization and network connection process. Initially, the boot sequence logs will confirm that Matter and Thread stacks are successfully initialized, followed by FreeRTOS task scheduling. The device will then enter BLE advertising mode, printing messages such as "*BLE advertising started*" and "*Waiting for commissioning*" to indicate readiness for pairing. When the Google Home app initiates pairing, you should see logs like "*BLE connection established*" and "*Commissioning request received*" as the Matter controller attempts to provision the device. After commissioning over BLE is completed, the device will begin Thread network initialization, displaying messages like "*Thread stack initialized*" and "*Attempting to join Thread network*". Upon successful connection to the Google Nest Hub Border Router, you should see "Thread network connected successfully, Router assigned", confirming that the device has joined the Matter ecosystem. Finally, the device will exchange Matter messages with the network, printing logs related to cluster commands, attribute updates, and network events, indicating that the device is fully operational and controllable via the Google Home app.

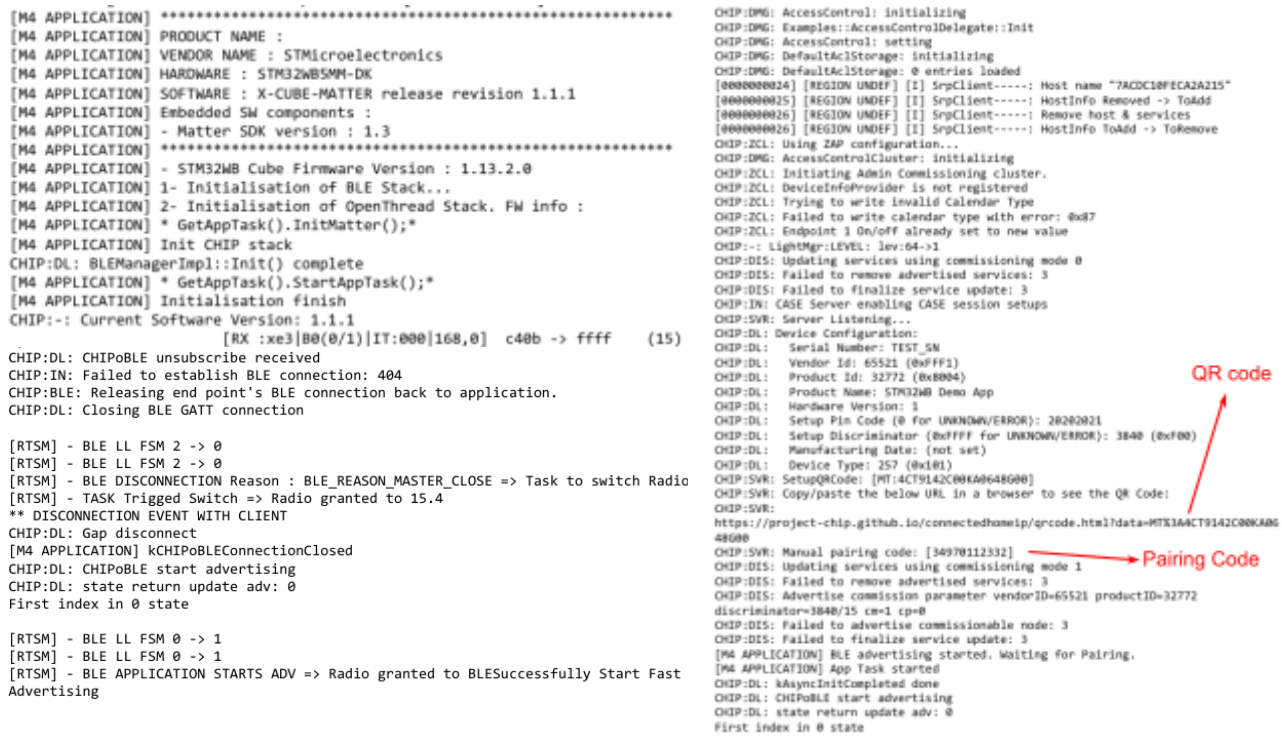


Figure 10: Screenshots from the TeraTerm terminal after hitting reset on the board

2.5. Connecting to Border Router

To connect your STM32 Matter device to a Google Nest Hub (2nd Gen) serving as a Thread Border Router, and to add the device using the Google Home app on your smartphone via a QR code, follow these steps:

1. **Ensure Prerequisites Are Met**

   ☑ Verify that your Nest Hub is updated to the latest firmware to function as a Thread Border Router.

   ☑ Ensure the Google Home app on your smartphone is updated to the latest version.

   ☑ Confirm that your STM32 device is flashed with Matter firmware and is in pairing mode.

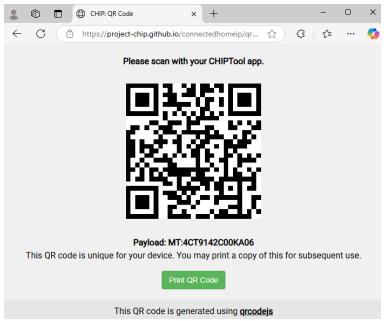
2. **Set Up the Google Nest Hub as a Thread Border Router:** Ensure your Nest Hub is connected to your Wi-Fi network (same as your smartphone) and linked to your Google Account. The Nest Hub (2nd Gen) should automatically function as a Thread Border Router. You can verify this in the Google Home app under the device settings.

3. **Add the Matter Device Using the Google Home App**

   ☑ Launch the Google Home app on your smartphone.

   ☑ Tap the "+" icon, then select "Set up device".

   ☑ Choose "New device", then tap "Matter-enabled device".



☑ Open the link in the terminal (in *Figure 10*) to view the pairing QR code. Use your smartphone's camera to scan the Matter pairing QR code.

☑ Alternatively, tap "Set up without QR code" and enter the numeric setup code manually. The pairing code is seen below the QR code link, a string of 11 numbers.

☑ Follow the on-screen instructions to complete the setup process.

Once the Matter device is successfully commissioned and connected to the Thread Border Router, the Google Home app provides an intuitive user interface for interacting with it. The app dynamically configures the controls based on the Matter clusters and attributes programmed in the firmware. For instance, if the device includes a temperature, humidity, or pressure sensor, the app will display real-time sensor data, updating periodically as new measurements are received.

For actuators like dimmable lights, the app provides an interactive sliding scale that allows users to adjust brightness smoothly, sending precise commands to the device. If the Matter device includes on/off functionality, a simple toggle switch will be available for control. Users can also integrate the device into Google Home automation routines, allowing it to respond to scheduled triggers or voice commands via Google Assistant. The seamless integration ensures that the Matter device operates reliably within the smart home ecosystem, providing a user-friendly experience with real-time status updates and responsive control.

3. Feasibility and Design Considerations

This section reflects on the practical insights gained throughout the development of the STM32 Matter-over-Thread smart home application. It covers hardware selection factors, technical challenges encountered, and possible future directions for improving and scaling the project. These reflections aim to guide others looking to build robust and interoperable Matter-compliant devices.

3.1. Choosing the Right Board

When selecting a development board for a Matter-over-Thread application, several critical properties must be considered regardless of the vendor (e.g., STM32, ESP32, or Nordic nRF series). Each platform (STM32, ESP32, nRF52/nRF53) has trade-offs in flexibility, ease of use, and level of Matter SDK integration but the properties below should guide any selection.

1. **Thread and Matter Compatibility**: It must support the 802.15.4 radio interface required by the Thread protocol and be certified or easily adaptable to the Matter specification.
2. **Dual-Core Architecture**: Boards like the STM32WB series feature a Cortex-M4 for the user application and a Cortex-M0+ dedicated to wireless stack execution, providing efficient separation of concerns. Single Core devices like NUCLEO-WB55RG can be equally used, but….
3. **Sufficient Flash and RAM**: Matter and Thread stacks are memory-intensive. A minimum of 1MB Flash and 256KB RAM is recommended for smooth performance.
4. **BLE Capability**: BLE is used for Matter commissioning, the board must support simultaneous BLE and Thread operation or quick switching between them.
5. **Developer Ecosystem and Toolchain**: Strong IDE support (e.g., STM32CubeIDE, Espressif IDF, Nordic SDK), robust documentation, and active community forums are vital for debugging and development.
6. **Peripheral Support**: For sensor integration, the board should support I2C/SPI/UART interfaces and offer real-time task handling via FreeRTOS or another RTOS.
7. **Established Online Resources**: Choose a board with a strong ecosystem of tutorials, application notes, community support, and vendor documentation. For example, STMicroelectronics provides extensive Matter development resources, including the X-CUBE-MATTER expansion package, platform-specific guides, and reference projects like the Lighting and Window Covering applications.

3.2. Challenges and Lessons Learned

Throughout the development of this project, a range of technical setbacks, configuration errors, and workflow inefficiencies revealed valuable insights into the complexities of working with Matter and Thread on embedded platforms. These challenges ranging from incorrect flashing sequences to integration issues with third-party libraries highlighted critical areas where deeper understanding, better planning, or improved tooling could have prevented time-consuming debugging. This section captures the key obstacles encountered and the lessons learned from overcoming them, serving as guidance for future development and helping to build a more resilient workflow for Matter-based applications.

1. **Underestimating Stack Requirements**: The division of responsibility between the Cortex-M0+ and Cortex-M4 cores was not initially well understood. Omitting the wireless stack firmware before flashing the Matter application caused broken Thread communication. This was resolved by always programming the Thread stack on the M0+ core first, then reflashing the application firmware on the M4.
2. **Wrong Start Addresses**: The firmware flashing process failed when incorrect start addresses were set in STM32CubeProgrammer—for example, using *0x08000000* for the Thread stack instead of *0x08096000*. Correcting this required referring to the ST-provided memory maps and aligning the addresses with each binary's purpose.
3. **ZAP Configuration Confusion**: Modifying *.zap* files without re-integrating the generated code back into the STM32 project led to missing callbacks and unexpected runtime behavior. The issue was addressed by carefully copying the regenerated ZAP output (e.g., *gen_config.h*, *IMClusterCommandHandler.cpp*) into the appropriate middleware directory and rebuilding the project.
4. **Skipping Thread Stack Flashing**: After updating the M0+ core, forgetting to reflash the Matter application on the M4 resulted in an unsynchronized system. This was resolved by adopting the practice of always reflashing the M4 core firmware after any wireless stack update.
5. **Serial Output Not Showing**: Initial UART output failed to display due to misconfigured pins and missing `ChipLogProgress()` calls. Proper UART setup in STM32CubeMX, alongside consistent logging macros, restored debugging output in the serial terminal.
6. **Lack of Documentation for Manual Setup**: Building the application from scratch exposed a lack of clear guidance for setting up the Matter SDK independently of example apps. This was overcome through extensive reference to vendor tutorials (e.g., ST Wiki), Matter GitHub issues, and testing various combinations of project structure and file inclusion.
7. **Wrong BLE Commissioning Flow**: BLE pairing failed when the Matter commissioning process was initiated without a valid QR or manual code. This was fixed by ensuring the ZAP configuration included the correct onboarding payload and using the chip tool or Google Home app to scan the QR code generated from the *.zap* configuration.
8. **Challenges with Third-Party Sensor Drivers**: Attempting to use drivers found online (like BME280 sensor drivers) led to repeated build errors and missing dependencies. Adapting external code to work with the STM32 HAL and FreeRTOS context proved more difficult than expected. The lesson learned was to write or adapt minimal, self-contained drivers tailored specifically to the project's architecture.
9. **Understanding Project Structure Before Editing**: When customizing example applications like *lighting-app*, blindly editing without first understanding the project flow caused confusion and unexpected

behavior. Knowing where Matter is initialized, where BLE advertising begins, and where sensor data is read or sent is essential before integrating new features. Careful tracing of entry points such as *main.c*, *AppTask.cpp*, and *Zcl_callbacks.cpp* were key to making meaningful modifications.

10. **Improper Project Setup and File Linking**: Several issues stemmed from not properly setting up the project structure at the start. Missing include paths, incorrect relative file references, and inconsistencies between the generated and actual project files led to building failures and undefined references. In some cases, outdated versions of generated directories were still being used after editing resulting in mismatched configurations and missing implementations. These problems highlighted the importance of keeping project directories synchronized after regeneration, correctly linking all required files in STM32CubeIDE (rather than copying), and regularly cleaning and rebuilding the project to avoid stale artifacts. Proper setup and consistent file management proved critical for ensuring a smooth build and integration process.

3.3. Future Improvements and Enhancements

As the project matured through trial-and-error and deep hardware-software integration, it became clear that several practices could improve the development flow and system reliability. While the application is functionally complete, future iterations can benefit from tighter structure, improved tooling, and development discipline. This section outlines enhancements to the development workflow, tool usage, and code organization that would make the process more maintainable, scalable, and less error-prone especially when working with vendor-specific Matter SDKs on constrained embedded platforms like STM32.

1. **Build Entirely from Scratch Using Board-Generated Configuration**: Rather than modifying existing examples from *X-CUBE-MATTER* like *Lighting-App*, create the application from scratch based on a CubeMX-generated configuration tailored to the board. This ensures only relevant peripherals are enabled, and avoids unnecessary complexity from pre-configured projects.
2. **Tailor Code to Match Board Configuration**: Customize initialization code (e.g., I2C, UART, GPIO) and sensor logic to align with the board-specific hardware setup generated by CubeMX. Avoid hardcoding peripherals from example projects that may not match the target hardware.
3. **Rebuild the C/C++ Indexer After Manual Edits**: After manually adding or modifying files (e.g., ZAP-generated headers or source files), rebuild the indexer in STM32CubeIDE to avoid false "unresolved" errors and to ensure the IDE reflects the latest project structure.
4. **Use Version Control from the Start**: Link the CubeIDE project to a GitHub repository early in development. Commit and push changes regularly, especially before building or regenerating code to preserve working states, enable rollbacks, and track configuration changes over time.
5. **Experiment with Different Toolchain Paths**: If one tool doesn't work (e.g., ZAP GUI fails to generate files), pivot to the CLI-based alternative like *zap_cli* or *generate.py* for more control. Having multiple ways to generate Matter files increases flexibility when working across platforms or debugging configuration issues.
6. **Handle Cortex-M0+ Flashing with Caution**: The M0+ core is critical for Thread operation, and flashing errors can potentially lock the core. Always verify the correct binary and start address before programming. If problems arise, recovery may not be possible even with a complete flash erase.

7. **Add Debugging Checkpoints**: Insert `ChipLogProgress()` or `printf()` messages after key operations such as BLE advertising, Thread initialization, commissioning start, or sensor polling. These checkpoints simplify runtime debugging and help trace where failures occur.
8. **Maintain Organized Application Structure**: Adopt clean architecture by isolating logic into functional files like drivers and "Managers". For instance, a file like *LightingManager.c* is used to manage device states and expose control functions to *AppTask.cpp*. This modularity improves readability, reusability, and debugging.

4. Conclusion

This report provided a comprehensive walkthrough of developing a Matter-over-Thread Smart Home Application using the STM32WB5MM-DK board, following the structure outlined in the introduction. Section 1 introduced the Matter and Thread protocols and laid out the motivation for building the project from scratch, while Section 2 offered two parallel development paths: one starting from a blank STM32 project and one adapting an existing reference implementation like *Lighting-App*. Detailed step-by-step instructions covered setting up the Matter SDK, configuring peripherals and Thread settings, generating Matter cluster files using ZAP, flashing both cores with the appropriate binaries, and commissioning the device to a Google Home Thread network. Section 3 discussed feasibility and design decisions emphasizing hardware selection criteria, technical challenges encountered (from flashing issues and configuration errors to BLE pairing difficulties), and proposed improvements for future work including debugging strategies, structured project layout, and source control practices.

While the project successfully researches the features of a functional Matter-compliant environmental sensor application, it also highlights a key constraint: STM32 Matter development resources are still relatively sparse, especially for developers starting from scratch. Vendor documentation often assumes prior knowledge, and troubleshooting unsupported configurations can be time-consuming. However, with enough persistence, detailed study of available resources, and structured debugging, it is feasible to build and scale reliable Matter devices on STM32 hardware. This project serves as both a practical implementation and a living guide for future Matter developers working in similarly under-documented environments.

References

[1] Connectivity Standards Alliance, “Matter: The Foundation for Connected Things,” 2023. [Online]. Available: https://csa-iot.org/all-solutions/matter/

[2] Silicon Labs, “Matter Fundamentals: Introduction,” 2023. [Online]. Available: https://docs.silabs.com/matter/latest/matter-fundamentals-introduction/

[3] Thread Group, “Thread: The Wireless Networking Protocol for IoT,” 2023. [Online]. Available: https://www.threadgroup.org/what-is-thread/overview

[4] Thread Group, “Thread Network Fundamentals,” Version 3, 2023. [Online]. Available: https://www.threadgroup.org/Portals/0/documents/support/Thread%20Network%20Fundamentals_v3.pdf

[5] STMicroelectronics, “STM32 and Thread Connectivity,” 2023. [Online]. Available: https://wiki.st.com/stm32mcu/wiki/Category:Thread

[6] STMicroelectronics, "Matter - Develop and Prototype," *STMicroelectronics Wiki*, 2024. [Online]. Available: https://wiki.st.com/stm32mcu/wiki/Connectivity:Matter_develop_and_prototype. [Accessed: 19-Mar-2025].

[7] Google, "Google Nest Hub and Matter Support," *Google Nest Help*, 2024. [Online]. Available: https://support.google.com/googlenest/answer/13127223 . [Accessed: 19-Mar-2025].

[8] Project CHIP, "ZAP - Zigbee Cluster Library Advanced Platform," *GitHub*, 2024. [Online]. Available: https://github.com/project-chip/zap. [Accessed: 19-Mar-2025].

[9] STMicroelectronics, "STM32WB Firmware Upgrade Services (FUS)," *STMicroelectronics Wiki*, 2024. [Online]. Available: https://wiki.st.com/stm32mcu/wiki/Connectivity:STM32WB_FUS . [Accessed: 19-Mar-2025].

[10] STMicroelectronics, "X-CUBE-MATTER Expansion Package," *STMicroelectronics*, 2024. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-matter.html . [Accessed: 19-Mar-2025].

[11] STMicroelectronics, "STM32CubeProgrammer User Guide," *STMicroelectronics Documentation*, 2024. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeprog.html . [Accessed: 19-Mar-2025].