Getting started with K32W0x1 in Matter

CONTENTS

1 INTRODUCTION

Matter (previously known as Project CHIP) is a new single, unified, application-layer connectivity standard designed to enable developers to connect and build reliable, secure IoT ecosystems and increase compatibility among Smart Home and Building devices.

For enabling Matter devices, NXP offers scalable, flexible and secure platforms to enable the variety of use cases Matter addresses – from end nodes to gateways – so device manufacturers can focus on product innovation and accelerating time to market.

This document focuses on NXP's standalone solution for Matter end nodes/routers with Thread using the K32W0x1 wireless microcontroller family.

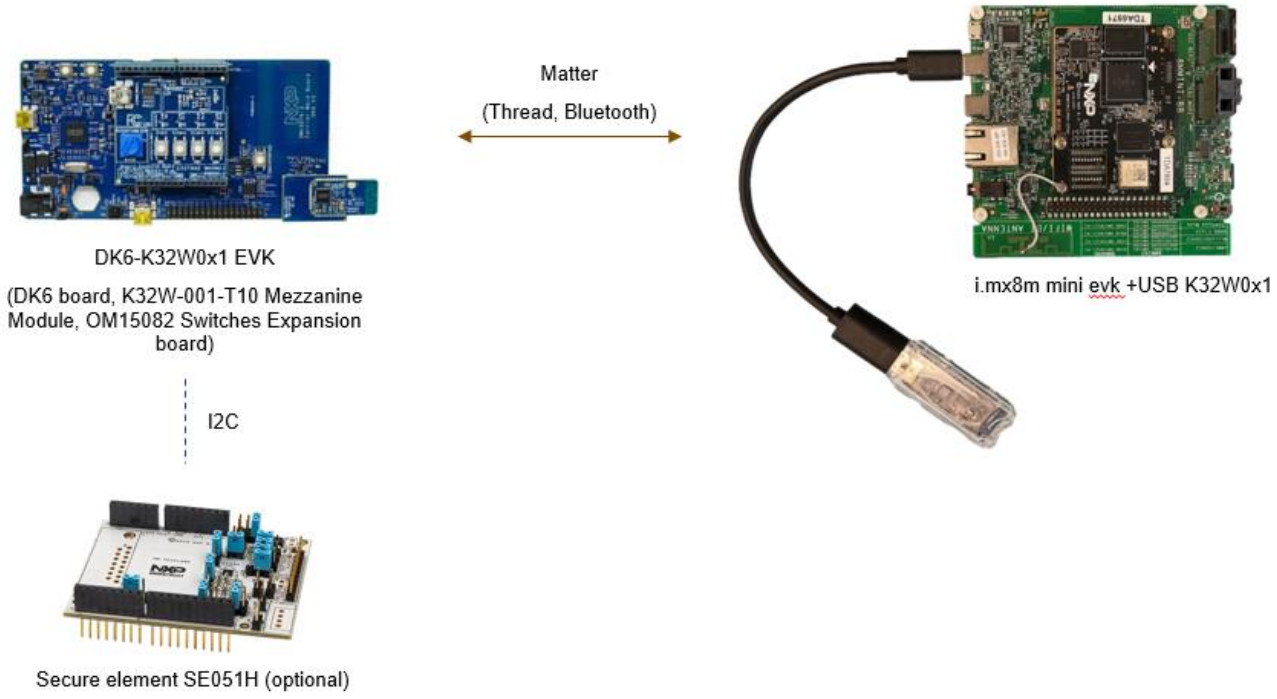
The K32W0x1 portfolio is designed for ultra-low-current multiprotocol wireless IoT devices with support for IEEE 802.15.4 mesh network protocols Zigbee® and Thread™ as well as Bluetooth® Low Energy 5.0. These wireless MCUs include multiple low-power modes and ultra-low radio Tx and Rx power consumption which enables IoT products powered by K32W0x1 to have extended battery life. With high Rx sensitivity and configurable Tx output power, the K32W0x1 MCUs offer reliable and robust connectivity performance.

https://www.nxp.com/part/K32W061#/
https://www.nxp.com/part/K32W041#/

2 HARDWARE SETUP

The minimum hardware required to create and run an end to end Matter setup with K32W0x1 is listed below:

- i.MX8M Mini EVK - acting as a Thread Border Router and Chip Controller: 8MMINILPD4-EVK
- USB-K32W0x1 – acting as RCP for Thread Network: USB-K32W0X1
- DK6-K32W061 + Expansion board OM15082- acting as Matter Accessory Device: IOTZTB-DK006
- SE051H - Secure Element- optional, to be used for cryptographic operations: SE051



3 MATTER ENVIRONMENT SETUP

Matter development relies on open-source resources, leveraging Linux based operating systems like Ubuntu and other tools like git, gcc and python. This also includes GN, a meta build system that generates makefiles and Ninja, a build system meant to replace Make tool.

Matter is available as an open-source SDK containing all the necessary components from scripts to install required tools to stack source code and vendor provided applications.

First step in developing a Matter application is to have Linux support for the build. The recommendation is to have a native Linux machine. If Windows is preferred operating system, support for the build can be set by using:

- Windows Subsystem for Linux (WSL);
- Linux Virtual Machine;

3.1 WSL Ubuntu 20.04 LTS

The Windows Subsystem for Linux (WSL) provides a GNU/Linux environment directly to Windows.

Use the following steps to install the WSL Ubunut 20.04 LTS:

1. On Windows 10, open PowerShell as administrator and run the following commands:

   Enable the Windows Subsystem for Linux:

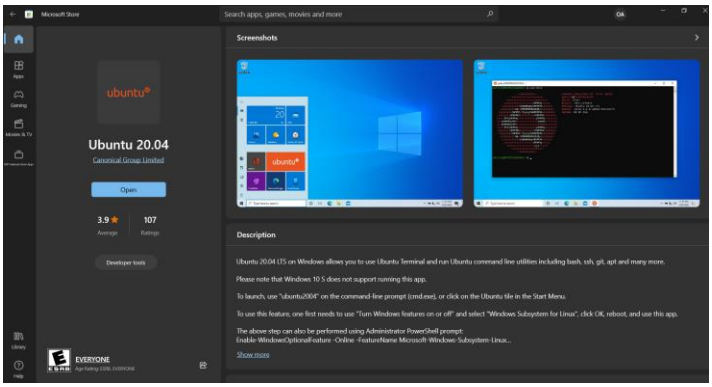
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

   Enable virtual machine feature:

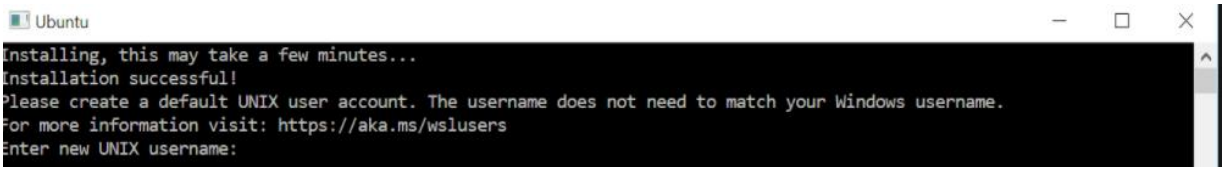
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart

   Restart your machine to complete the WSL installation

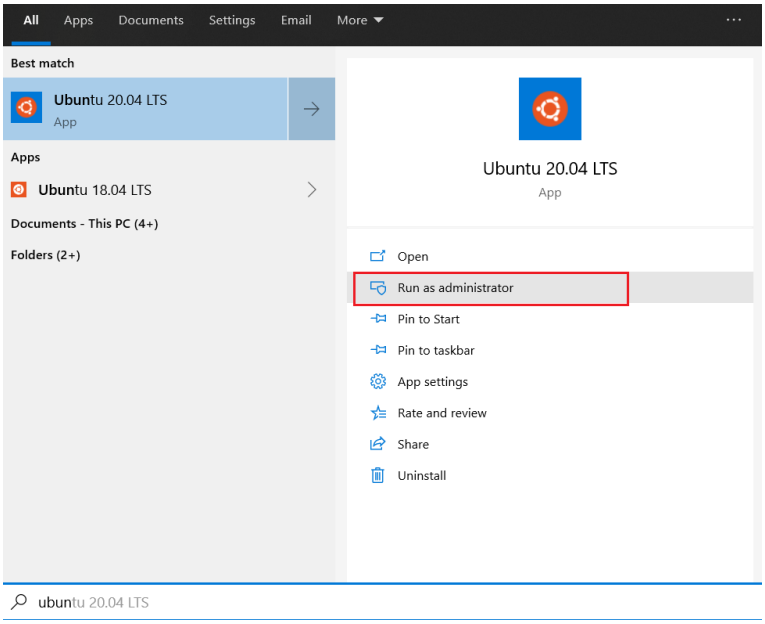
2. Install the Ubuntu 20.04 from Microsoft Store



   Create a user account and password for your new Linux distribution



After installation is complete, run Ubuntu 20.04 LTS as administrator. Administrator rights are required on the first run to install needed packages for Matter.

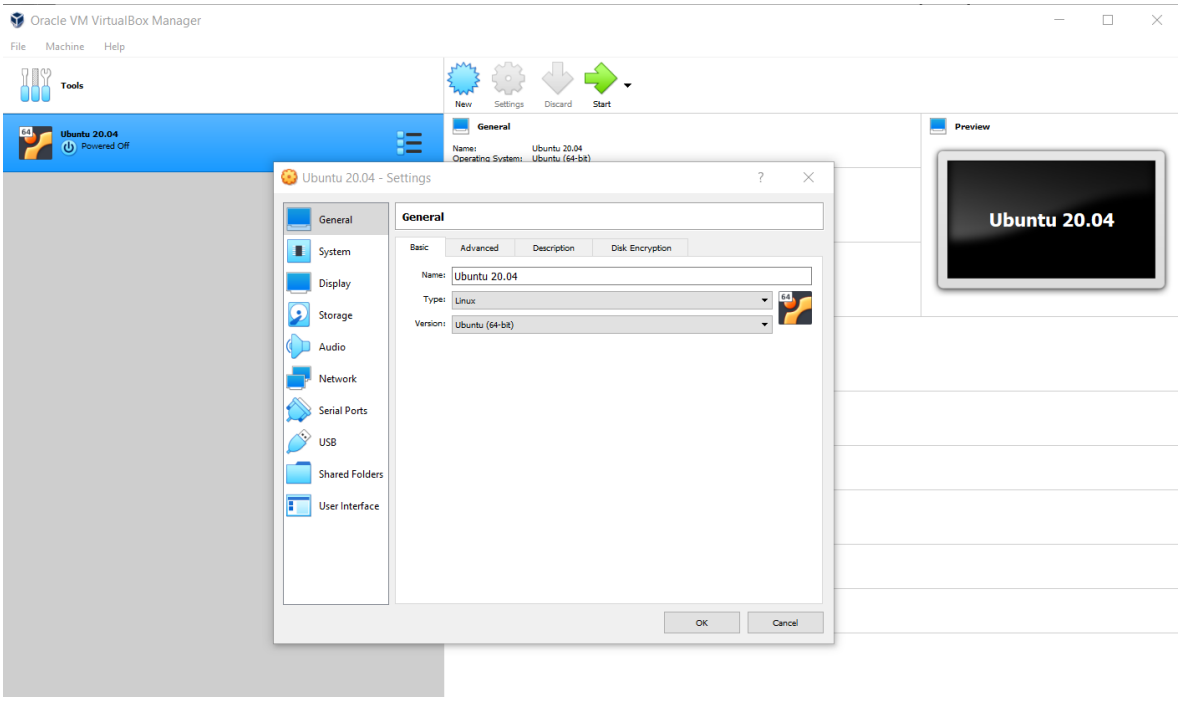


3.2 Linux Virtual Machine

The following steps guide you through virtualbox machine installation steps:

- Download and install virtual machine: https://www.virtualbox.org/wiki/Downloads
- Please consider that based on your operating system some extra steps will be required for enabling the Virtualization support. These extra steps consist in settings that need to be done in the machine's BIOS regarding virtualization support and they depend on the machine type and manufacturer. For further details, please consult your machine's user manual.
- Download the desktop image for Ubuntu 20.04 Focal and create virtual machine using the downloaded ISO.
- Configure the virtual machine:
    - Linux Ubuntu 20.04 (64 bit);
    - VM disk size – more than 20GB;

- Enable USB controller -> USB 1.1(OHCI) Controller (support for USB 2.0 is recommended, if possible, this being available via the Oracle VM VirtualBox Extension Pack);
- Enable network adapter-> Adapter 1 -> Attached to Bridged Adapter;
- Other recommended settings are to increase the number of cores and the size of RAM to be dedicated to the virtual machine. This depends on the user's actual machine. Minimum recommendation is to have 4 dedicated cores and 4 GB of RAM.



3.3 Setting up Matter Environment

The following steps guide you through creating Matter build environment for K32W0x1 MCU.

1. Matter Dependencies:

   Check for updates and install dos2unix (useful for WSL):

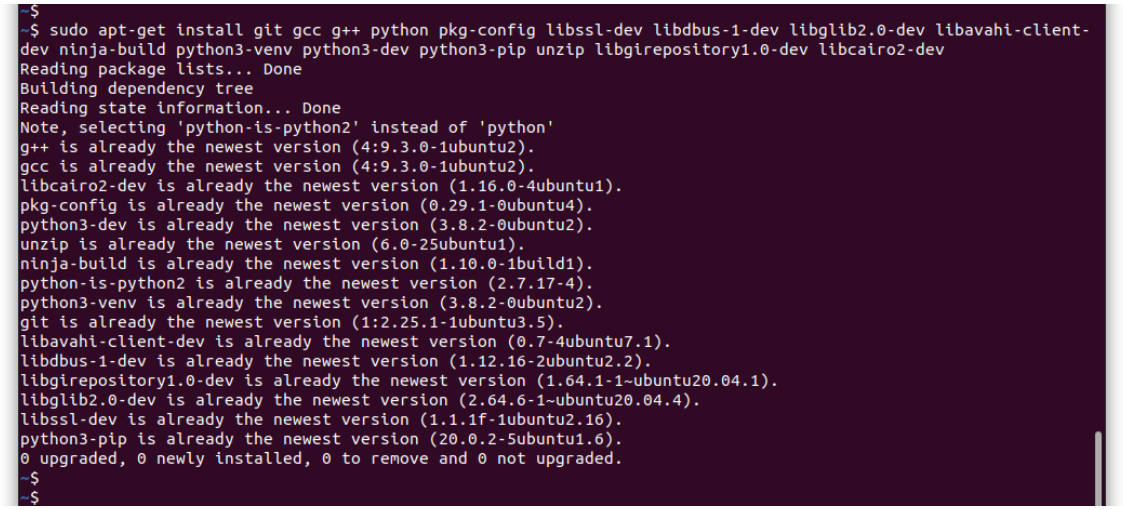
$ sudo apt update

$ sudo apt upgrade --y

$ sudo apt-get install dos2unix # optional, required if using files having DOS style line endings that need to be converted to Linux style line endings

   Install Matter dependencies:

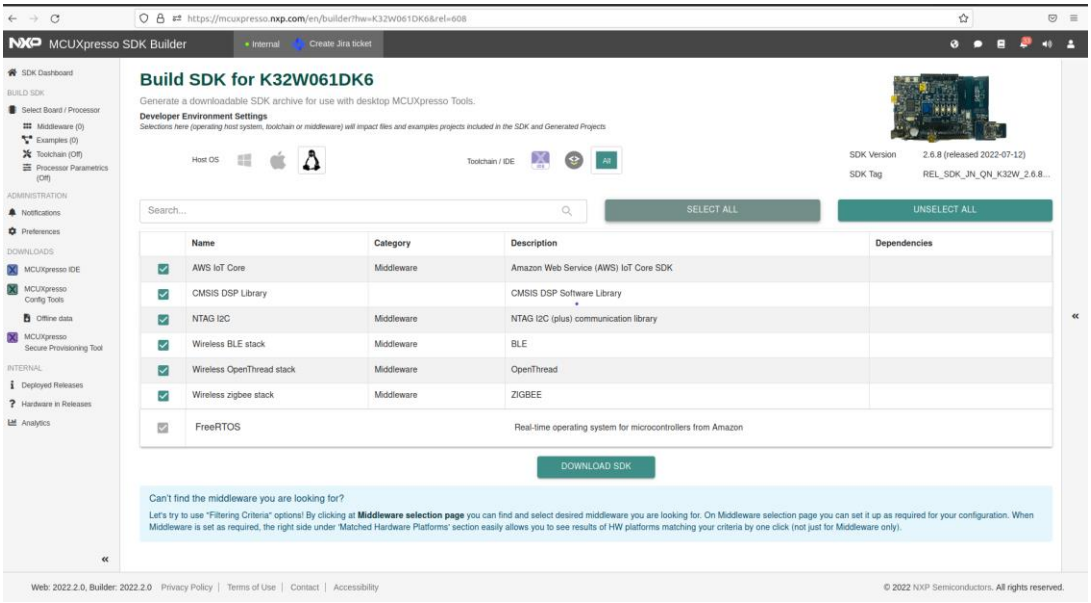
$ sudo apt-get install git gcc g++ python pkg-config libssl-dev libdbus-1-dev libglib2.0-dev libavahi-client-dev ninja-build python3-venv python3-dev python3-pip unzip libgirepository1.0-dev libcairo2-dev gcc-arm-none-eabi



   Restart the Linux machine/environment if required.

2. Getting the K32W0x1 SDK:

   The MCUXpresso SDK builder website allows the user to create SDKs for K32W0x1 by choosing the option for Host OS to be Linux and selecting all middleware components. For the following steps used as example, the SDK 2.6.8 for the K32W061 variant was used. Download and unzip into a predefined location on your PC (for example on Windows OS, C:/nxp or on Linux based OS, /home/user/Documents).



If the downloaded SDK is not the Linux host variant, then the following command must be run to resolve errors with the file formatting for the SDK intended for a non-Unix host system:

$ find . -type f -print0 | xargs -0 dos2unix

3. Matter Building Setup instructions:

   Clone the Matter SDK using the public repo:

$ git clone https://github.com/NXPmicro/matter/
$ cd matter
$ git checkout v1.0-branch-nxp
$ git pull
$ git submodule update --init –recursive

Start build environment by running the activate script:
$ source ./scripts/activate.sh

Note that the *activate.sh* script can take a long time to execute; this behavior is normal.


patching file fsl_os_abstraction_free_rtos.c
Hunk #3 succeeded at 473 with fuzz 1.
Hunk #4 succeeded at 524 with fuzz 2.
patching file OtaUtils.h
K32W SDK 2.6.6 was patched!
~/matter$
~/matter$ source scripts/activate.sh
WELCOME TO...
matter
BOOTSTRAP! Bootstrap may take a few minutes; please be patient.
Downloading and installing packages into local source directory:
Setting up CIPD package manager...done (1m4.6s)
Setting up Python environment.....done (2m0.8s)
Setting up pw packages.....skipped (0.1s)
Setting up Host tools.....done (0.1s)
Activating environment (setting environment variables):
Setting environment variables for CIPD package manager...done
Setting environment variables for Python environment.....done
Setting environment variables for pw packages.....skipped
Setting environment variables for Host tools.....done
Checking the environment:
20220926 18:08:13 INF Environment passes all checks!
Environment looks good, you are ready to go!
To reactivate this environment in the future, run this in your terminal:
source ./activate.sh
To deactivate this environment, run this:
deactivate
~/matter$

4 K32W0X1 MATTER EXAMPLES

In the current Matter SDK for K32W0x1 platform we are providing reference examples for end nodes, Thread End Device type applications, with support for low power, over the air update and cluster configuration and control.

The examples are listed in the matter-> examples:

```
~/matter/examples$ ls -l -R | grep nxp/k32w/k32w0:
./contact-sensor-app/nxp/k32w/k32w0:
./lighting-app/nxp/k32w/k32w0:
./lock-app/nxp/k32w/k32w0:
```

These applications provide support for persistence data storage (PDM) used for storing Matter and Thread configuration data. These are stored in the external flash memory available on the DK6 board as Macronix MX25R8035F connected through Quad SPIFI interface or the stacked flash of the same variant in the K32W041AM part connected through Dual SPIFI interface. The available space on this memory is 8 Mbit (1 MB). The SPIFI interface on the K32W0x1 part supports up to 16 MB of external flash space.

4.1 Matter application building instruction

The following build steps are based on the lighting app reference for NXP K32W0X1 - Building steps: https://github.com/NXPmicro/matter/tree/v1.0-branch-nxp/examples/lighting-app/nxp/k32w/k32w0

- Set the environment variable *NXP_K32W0_SDK_ROOT* to where you extract the SDK. An example is shown below, where the SDK is unzipped into the ~/Documents/SDK_2_6_8_K32W061DK6 directory:

$ export NXP_K32W0_SDK_ROOT=~/Documents/SDK_2_6_8_K32W061DK6

- Apply SDK patch from the Matter root directory using the following command:

$ ./third_party/nxp/k32w0_sdk/sdk_fixes/patch_k32w_sdk.sh

- Build K32W0x1 example application. Adapt the gn command with the device configuration by enabling/disabling the compile options:
  - Chip type (061/041/041A/041AM) for example, for K32W041AM, build_for_k32w041am
  - OTA requestor feature*: chip_enable_ota_requestor (true by default)*
  - Low power support*: chip_with_low_power (available for contact sensor and lock apps)*
  - Secure element SE015H usage*: chip_with_se05x*
  - OM15082 Expansion board attached to DK6 board*: chip_with_OM15082*
  - OpenThread command line interface*: chip_with_ot_cli*
  - Manufacturing data like DAC and PAI certificates and other security related data*: chip_with_factory_data*
  - Support for pigweed tokenizer applied on the constant strings*: chip_pw_tokenizer_logging*
  - Support for OpenThread CLI: *chip_with_ot_cli*
  - In case the K32W0x1 chip is used on a board without a 32KHz crystal, there is the option of using the internal 32KHz free running oscillator as a clock source. This can be enabled by setting *use_fro_32k=1*

$ cd examples/lighting-app/nxp/k32w/k32w0
$ gn gen out/debug --args="k32w0_sdk_root=\"${NXP_K32W0_SDK_ROOT}\" chip_with_OM15082=1 chip_with_ot_cli=0 is_debug=false chip_crypto=\"tinycrypt\" chip_with_se05x=0 chip_pw_tokenizer_logging=true mbedtls_repo=\"//third_party/connectedhomeip/third_party/nxp/libs/mbedtls\""

- Compile the demo application with the following command:

$ ninja -C out/debug

- Sign the image:

$ $NXP_K32W0_SDK_ROOT/tools/imagetool/sign_images.sh out/debug/

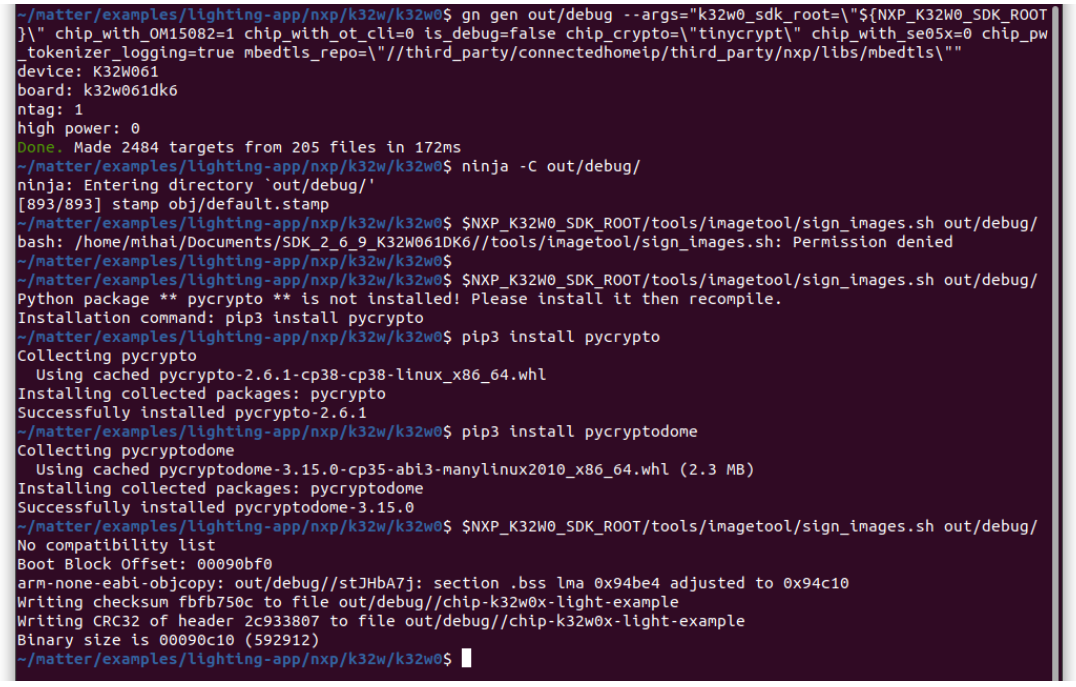
- If permission denied error is encountered, run the following command:

$ chmod +x $NXP_K32W0_SDK_ROOT/tools/imagetool/sign_images.sh

- There might be also a Python error related to pycrypto which can be fixed by issuing:

$ pip3 install pycrypto
$ pip3 install pycryptodome

Recommended versions for these Python packages are:
$ pip3 list | grep -i pycrypto
pycrypto 2.6.1
pycryptodome 3.9.8



- After build is complete, the results can be found in *examples/<application_name>/nxp/k32w/k32w0/out/debug* folder, which contains the elf and bin files for the application.
- For example, for the lighting app, in the examples/lighting-app/nxp/k32w/k32w0 folder, user can find chip-k32w0x-light-example as elf file and chip-k32w0x-light-example.bin as binary file.

4.2 Second Stage Bootloader application build

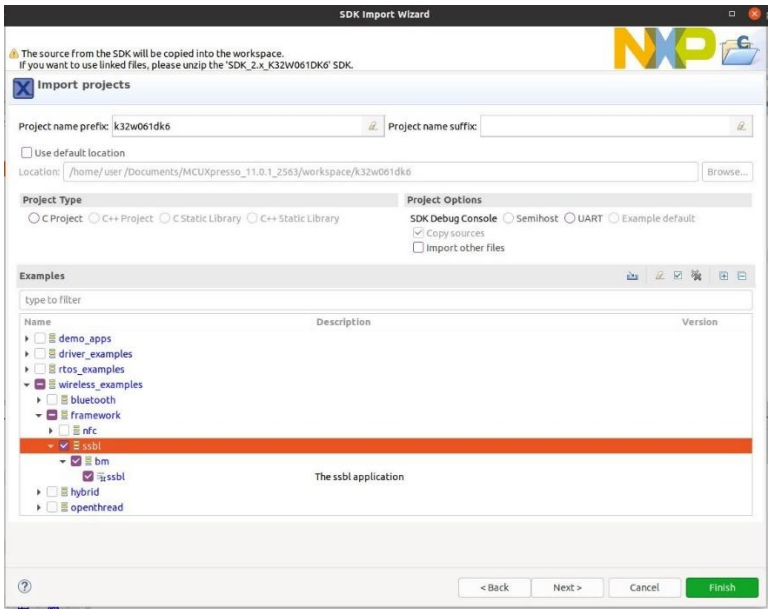
Using the *chip_enable_ota_requestor* build option in the Matter application build will automatically enable over-the-air update support. This means that the user must program two application binaries into the board, the first being the Second Stage BootLoader (SSBL) and the second being the binary obtained in the Matter application Building instructions step.

If *chip_enable_ota_requestor* is set to false, then only the Matter application binary is required; however, the device will be unable to receive over-the-air firmware updates.

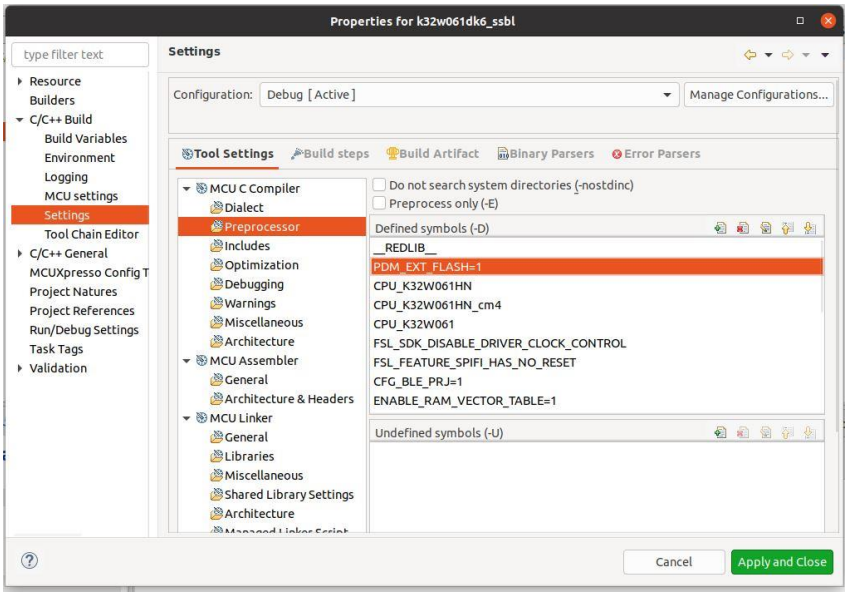
Below are are steps to build the SSBL application:

- As a prerequisite, the user must have the MCUXpresso IDE installed for this step, and the K32W0x1 SDK imported in the IDE.
- The SSBL can be built from the SDK demo examples:

  Import SDK example(s) -> select wireless->framework->ssbl application.



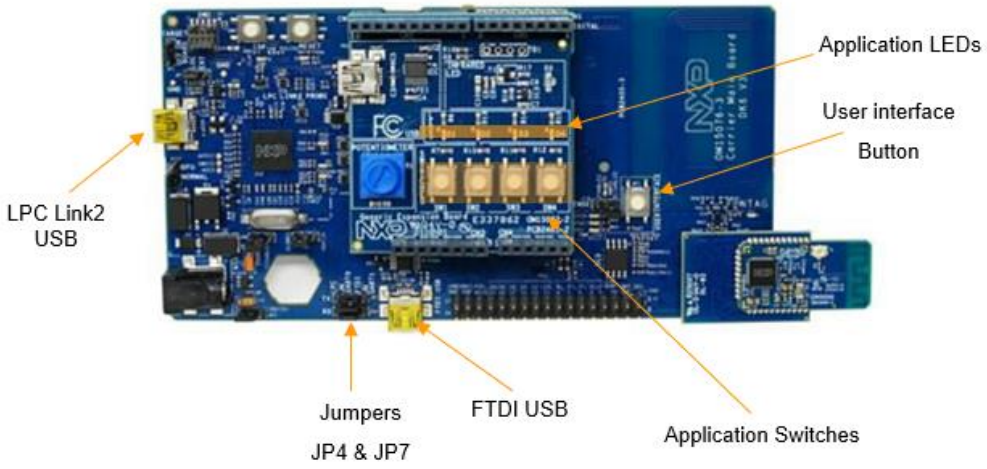
- The SSBL project must be compiled with the PDM_EXT_FLASH define:



- Next step is to build the project using MCUXpresso. The resulted binary file will be located in the corresponding debug/release folder: k32w061dk6_ssbl.bin

4.3 Writing Matter application to the DK6-K32W0x1 board

DK6 Hardware connections:

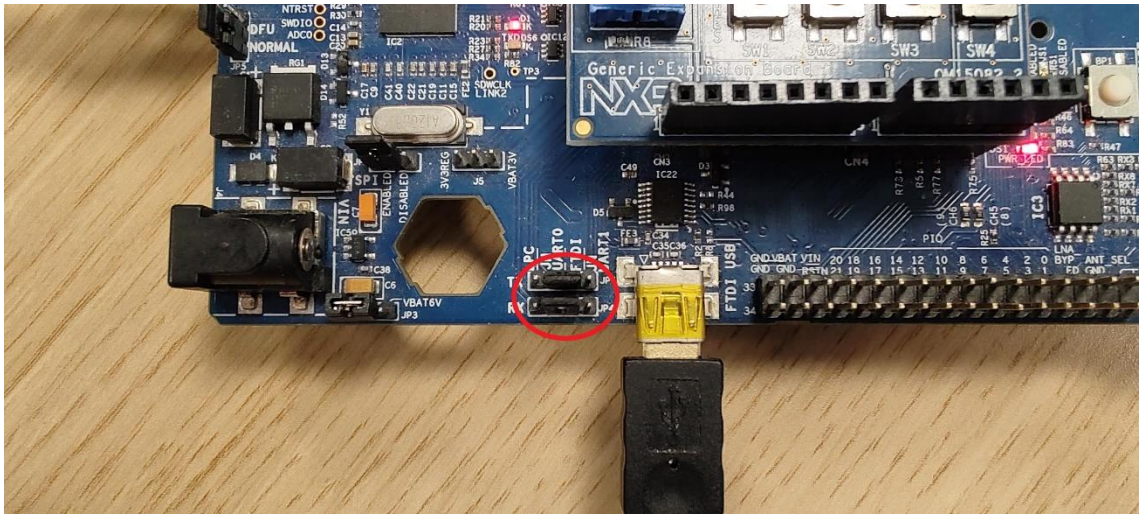


The Matter application can be downloaded into the DK6 K32W061 board either:

- via the LPC-LINK2 USB port using an IDE debugger (for example, MCUXpresso, Jlink);
- via UART0 using the DK6 Flash Programmer;

Flashing the application using DK6Programmer:

- Connect the DK6 board using a mini-USB cable to the connector marked with FTDI USB.
- Set the J4 and J7 jumpers to the middle position (JN UART0 - FTDI)



- The DK6 Programmer installer is included in the *{SDK path} -> tools folder*:

<SDK_root>\tools\JN-SW-4407-DK6-Flash-Programmer

- Once the application is installed, the COM port for K32W061DK6 can be identified by running the following command from a powershell terminal or command prompt:

DK6Programmer.exe --list

- Use the following commands to write to the board:
  - To erase the internal flash:

DK6Programmer.exe -V 5 -P 1000000 -s <COM_PORT> -e Flash

- To write the SSBL at address 0x0:

DK6Programmer.exe -V2 -s <COM_PORT> -P 1000000 -Y -p FLASH@0x00="k32w061dk6_ssbl.bin"

- To write the PSECT containing the image directory:
- First, image directory 0 must be written:

DK6Programmer.exe -V5 -s <COM_PORT> -P 1000000 -w image_dir_0=0000000010000000

Where the interpretation of the image directory 0 fields is:
00000000 -> start address 0x00000000
1000 -> size = 0x0010 pages of 512-bytes (= 8kB)
00 -> not bootable (only used by the SSBL to support SSBL update)
00 -> SSBL Image Type

- Second, image directory 1 must be written:

DK6Programmer.exe -V5 -s <COM_PORT> -P 1000000 -w image_dir_1=00400000C9040101

Where the interpretation of the image directory 1 fields is:
00400000 -> start address 0x00004000
CD04 -> 0x4C9 pages of 512-bytes (= 612,5kB)
01 -> bootable flag
01 -> image type for the application

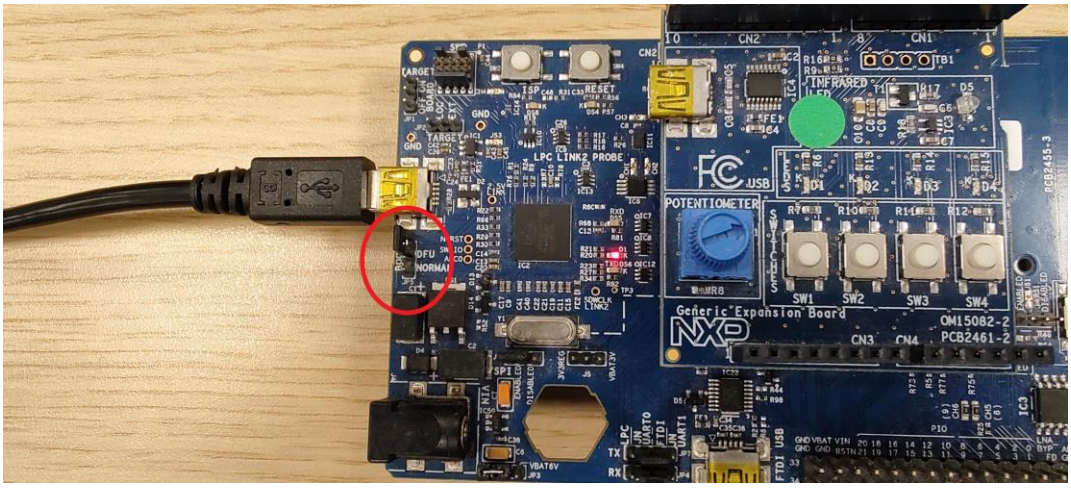
- To write the application at address 0x4000:

DK6Programmer.exe -V2 -s <COM_PORT> -P 1000000 -Y -p FLASH@0x4000="chip-k32w0x-light-example.bin"

Flashing the application using LPC-Link2 – J-Link Firmware:

By default, the LPC-Link2 is configured with CMSIS-DAP firmware. The J-Link firmware is also supported by this debugger and the following instructions can be used to change between the firmware versions.

- Put the board in DFU mode for debugger (JP5 – DFU mode) and connect the board to the PC via the LPC-LINK2 USB



- Download LPC Script according to the host OS (Windows, Linux, etc.) and install.

For Windows:

- Go to the installation folder (for example, by default, C:\nxp\LPCScrypt_2.1.2_57\)
- In the \scripts folder, open a Command Prompt and run the program_JLINK.cmd batch file.
- Make sure that the board is connected and press any key to continue.
- Unplug the board and switch the JP5 back to NORMAL.
- The firmware should be flashed to the LPC-Link2 LPC chip.

For Linux

- Go to the download folder and install the LPC script:
  $ chmod +x lpcscrypt-2.1.2_57.x86_64.deb.bin

$ ./lpcscrypt-2.1.2_57.x86_64.deb.bin

- Once installed, execute the script to reprogram your board with JLink firmware:
  $ <LPCScrypt_InstallDir>/scripts/program_JLINK
- Unplug the board and switch the JP5 back to NORMAL.
- The firmware should be flashed to the LPC-Link2 LPC chip.
- To revert to the CMSIS-DAP firmware, the program_CMSIS script needs to be run from the same folder using the same procedure

```
LPCScrypt - J-Link firmware programming script v2.1.2 Nov 2020.

Connect an LPC-Link2 or LPCXpresso V2/V3 Board via USB then press Space.

Press any key to continue . . .

Booting LPCScrypt target with "LPCScrypt_240.bin.hdr"
LPCScrypt target booted
.
Programming LPC-Link2 with "Firmware_JLink_LPC-Link2_20190404.bin"

 LPC-Link2 programmed successfully:
- To use: make link JP1 (nearest USB) and reboot.

Connect Next Board then press Space (or CTRL-C to Quit)

Press any key to continue . . .
```

Note: To revert to the CMSIS-DAP firmware, the program_CMSIS script needs to be run from the same folder using the same procedure.

- Download and install J-Link on your computer;
- Connect the board to the PC;
- Launch J-Link command to connect to the device from the installation folder:
- From Windows:
  - In C:\Program Files (x86)\SEGGER\Jlink folder, open a Command Prompt and run:

$ Jlink.exe -device K32W061 -if SWD -speed 4000 -autoconnect 1

- From Linux

$ JLinkExe -device K32W061 -if SWD -speed 4000 -autoconnect 1

- Erase the image using *erase* command

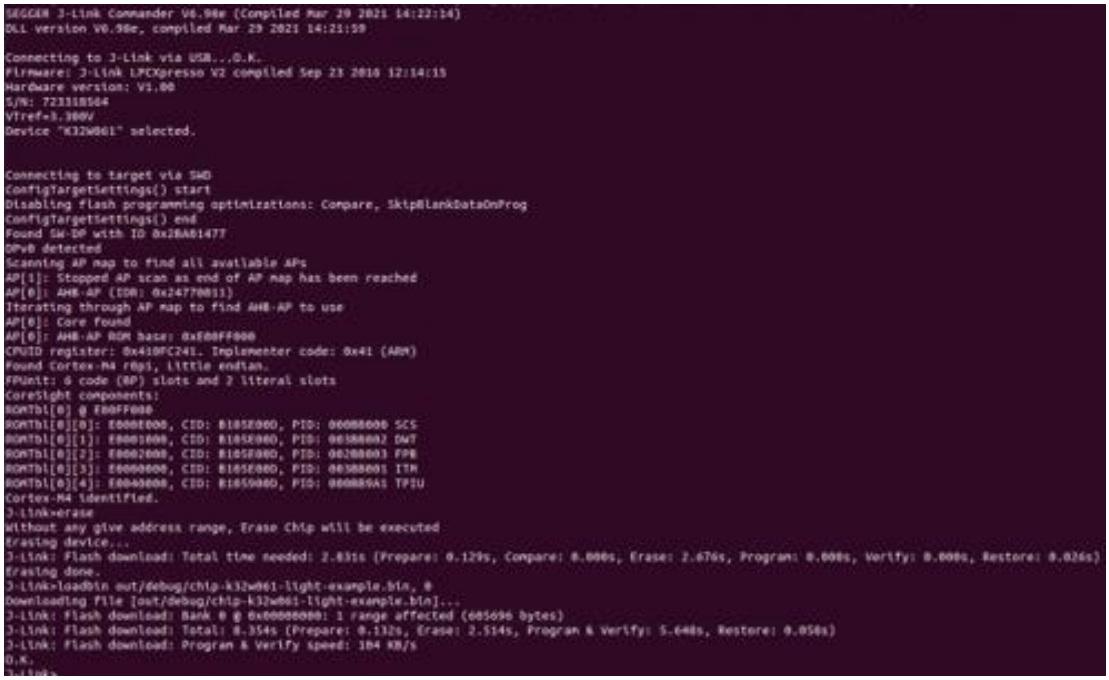
$ J-Link>erase

- Download new image using *loadbin* command (please consider that for example with SSBL, the start address will be 0x4000, otherwise it is 0)

$ J-Link>loadbin <path/to/binary/ k32w061dk6_ssbl.bin, 0x0000

$ J-Link>loadbin <path/to/binary/chip-k32w0x-light-example.bin, 0x4000

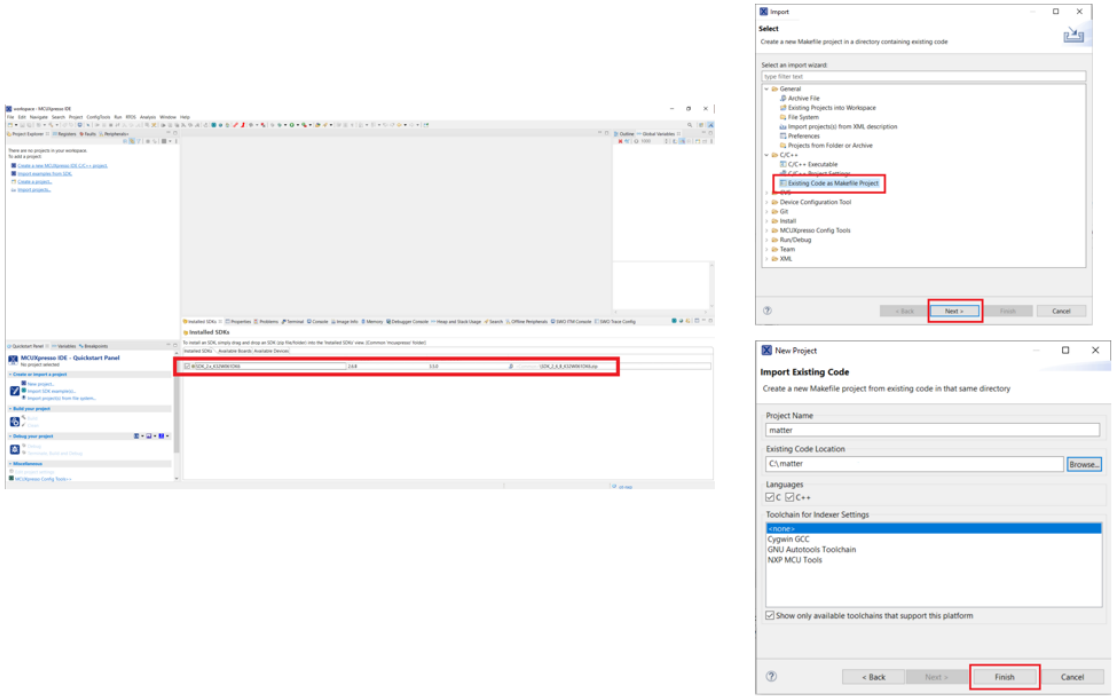
Note: Please consider that the PSECT configuration using DK6Programmer will be also required first time.



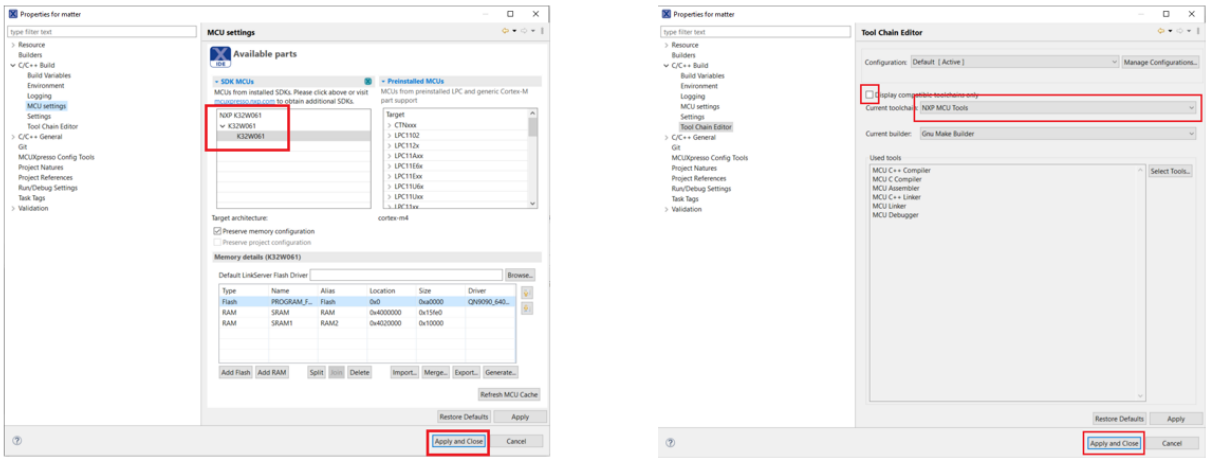
4.4 K32W0x1 Matter Application Debug

The following steps can be applied for Windows as well as for Ubuntu host OS:

- A debugger must be used here, whether it's the onboard LPC-Link2 debugger connected through the DK6 board's LPC-LINK2 USB port, or an external debugger connected to the JTAG on the same board.
- Download and install the MCUXpresso IDE (version >= 11.0.0) and create a workspace.
- Import K32W0x1 SDK into the IDE. This can be done by drag-and-drop the SDK archive into the *MCUXpresso IDEs Installed SDKs* tab.
- Import *connectedhomeip* repo in MCUXpresso IDE as Makefile Project. Use none as Toolchain for indexer settings: *File -> Import -> C/C++ -> Existing Code as Makefile Project*

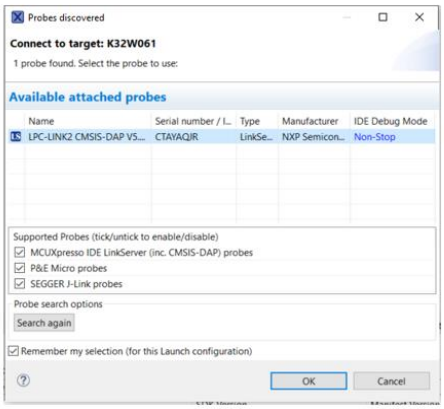


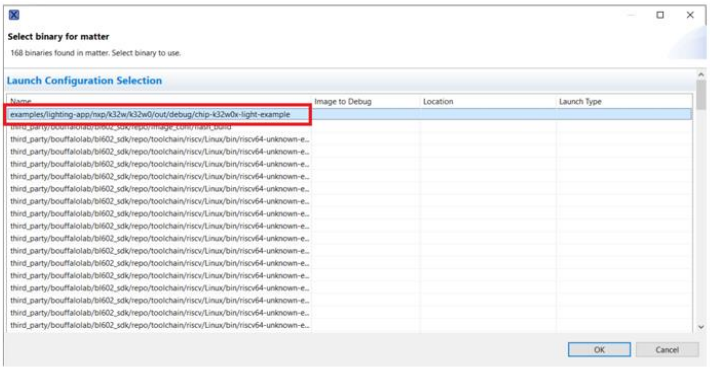
- Configure MCU Settings:
  Right click on the Project -> Properties -> C/C++ Build -> MCU Settings -> Select K32W061 -> Apply & Close
- Configure the toolchain editor:
  Right click on the Project -> C/C++ Build-> Tool Chain Editor
  Uncheck "Display compatible toolchains only" and select NXP MCU Tools from "Current toolchain" drop box.
  Apply & Close



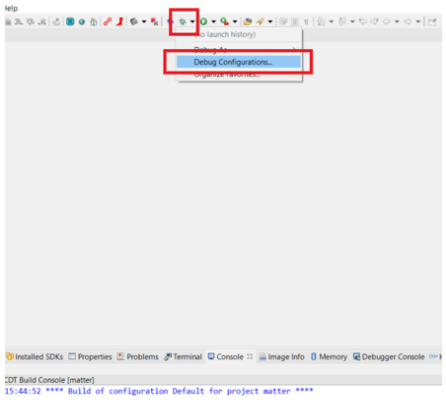
- Create a debug configuration:

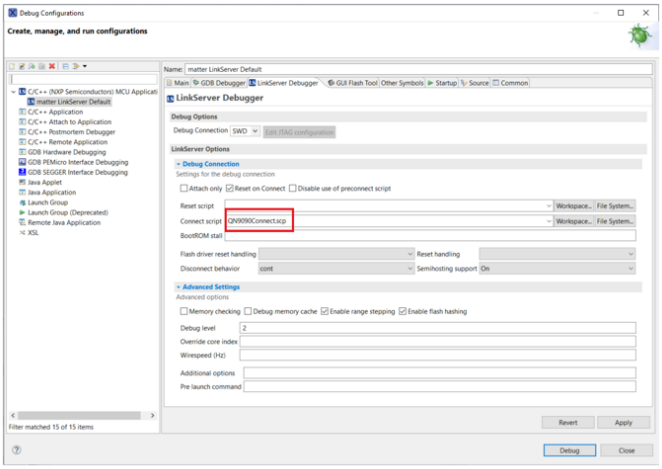
Right click on the Project -> Debug As->MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes -> OK -> Select elf file



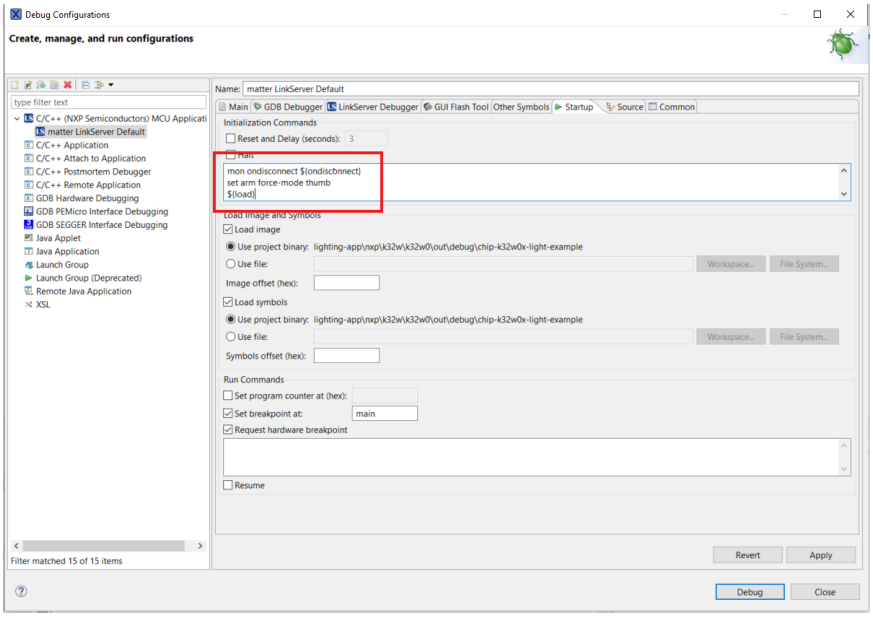


- Set the Connect script for the debug configuration to QN9090connect.scp from the dropdown list:
  Right click on the Project -> Debug As -> Debug configurations... -> LinkServer Debugger

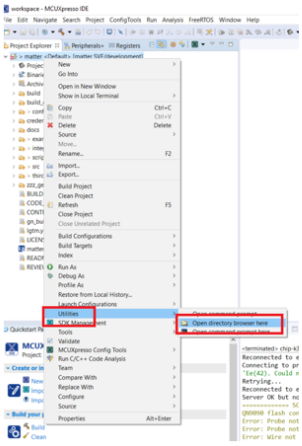


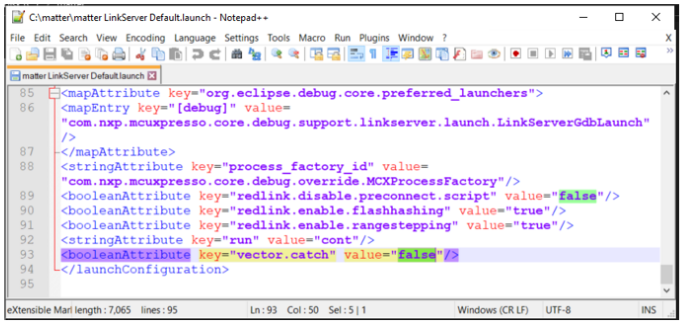


- Set the Initialization Commands:
  Right click on the Project -> Debug As -> Debug configurations... -> Startup
  set non-stop on
  set pagination off
  set mi-async
  set remotetimeout 60000
  ##target_extended_remote##
  set mem inaccessible-by-default ${mem.access}
  mon ondisconnect ${ondisconnect}
  set arm force-mode thumb
  ${load}

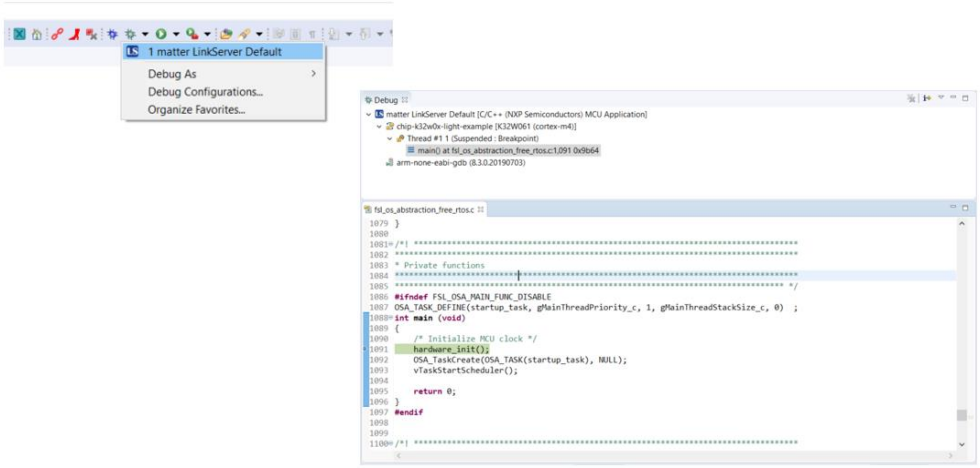


- Save the debug configuration by pressing “Apply”.
- Set the vector.catch value to false inside the .launch file:
  *Right click on the Project -> Utilities -> Open Directory Browser here -> edit *.launch file:*
  <booleanAttribute key="vector.catch" value="false"/>



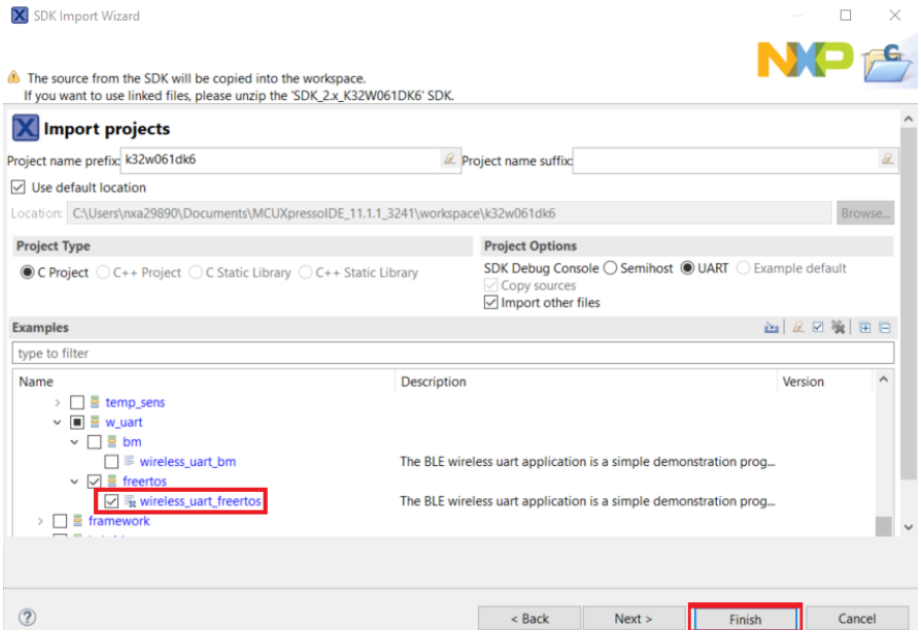


- Last step is to start debug the application:

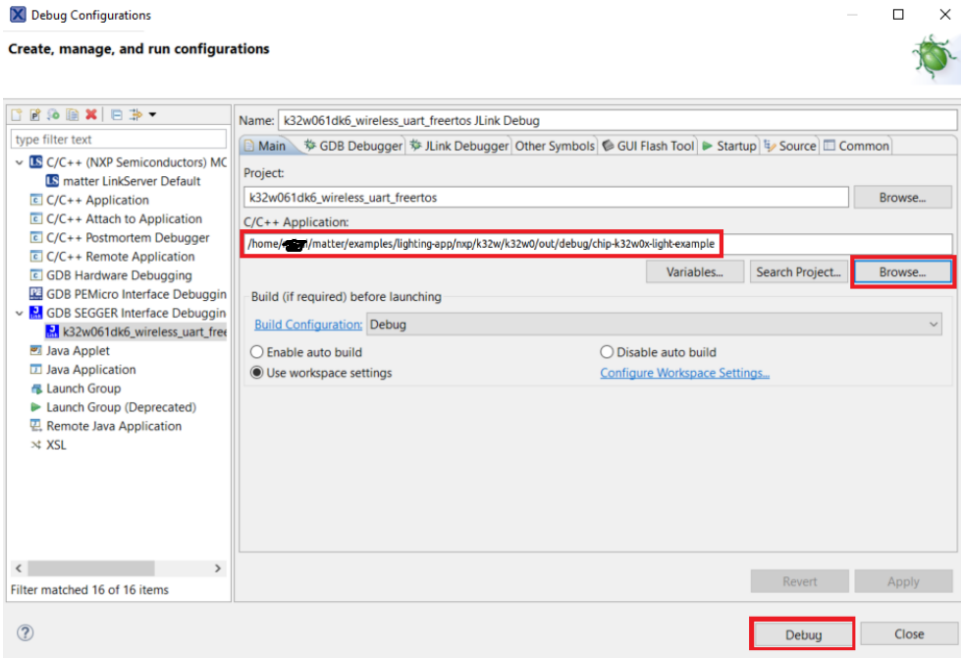


The alternative for this method is to build an SDK example and reuse the debug configuration by selecting the Matter example elf file:

- Import an SDK example:



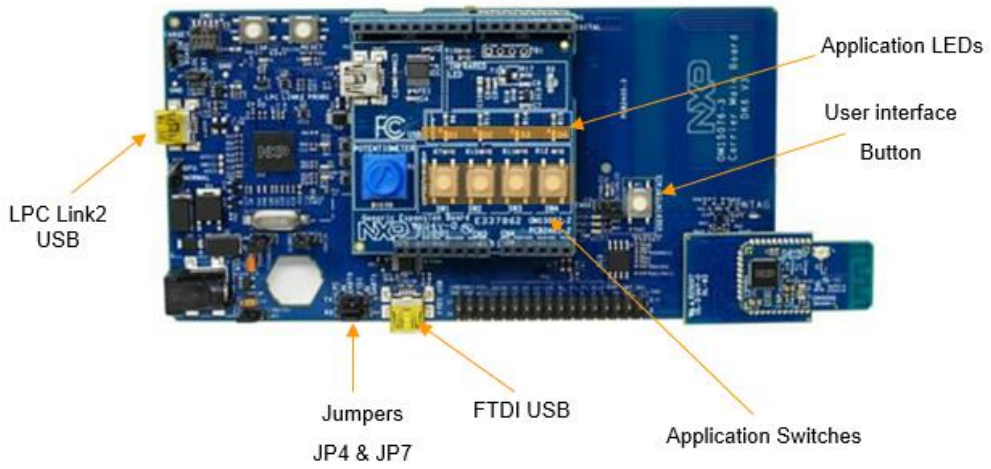
- Change the elf application to debug with the one from the Matter build:



4.5 K32W0x1 Matter Application User interface

The example application provides a simple UI that depicts the state of the device and offers basic user control.

This UI is implemented via the general-purpose LEDs and buttons built in to the OM15082 Expansion board attached to the DK6 board:



- Application LED D2 shows the overall state of the device and its connectivity. Four states are depicted:
  - Short Flash On (50ms on/950ms off) — The device is in an unprovisioned (unpaired) state and is waiting for a commissioning application to connect.

  - Rapid Even Flashing (100ms on/100ms off) — The device is in an unprovisioned state and a commissioning application is connected via BLE.
  - Short Flash Off (950ms on/50ms off) — The device is full provisioned, but does not yet have full network (Thread) or service connectivity.
  - Solid On — The device is fully provisioned and has full network and service connectivity.

- Application LED D3 shows the state of the simulated light bulb. When the LED is lit, the light bulb is on; when not lit, the light bulb is off.
- Application Button SW2 can be used to reset the device to a default state:
  - A short Press Button SW2 initiates a factory reset.
    After an initial period of 3 seconds, LED2 D2 and D3 will flash in unison to signal the pending reset. After 6 seconds will cause the device to reset its persistent configuration and initiate a reboot. The reset action can be cancelled by press SW2 button at any point before the 6 second limit.
- Application Button SW3 can be used to change the state of the simulated light bulb.
- DK6 development board – User Interface button
  - A short Press will start advertising for a predefined period;
  - Also, pushing this button starts the NFC emulation by writing the onboarding information in the NTAG.

In case the OM15082 Expansion board is not attached to the DK6 board, like the case in which low power support is enabled, the functionality of LED D2 and LED D3 is taken over by LED DS2, respectively LED DS3, which can be found on the DK6 board. Also, by long pressing the USERINTERFACE button, the factory reset action will be initiated.

5 MATTER NETWORK – CHIP TOOL COMISSIONGING AND CONTROL

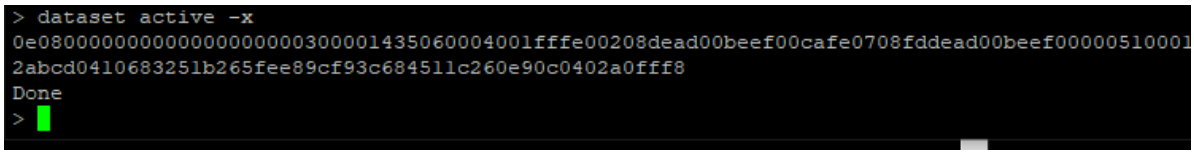
The prerequisite for this section is to have the i.MX8M Mini + 88W8987 + K32W0x1 setup as OTBR and Matter Controller in order to commission and control Matter devices. The steps for this setup can be found in "*Getting started with i.MX8 Mini in Matter*".

The following instructions will provide information on how to commission and run basic Matter Cluster commands for an application running K32W0x1 platform:

Commissioning K32W0x1 over Bluetooth LE:

- Get the operational active dataset from the Thread Network. On the i.mx8m mini run the following command:

$ ot-ctl dataset active –x



- Activate the BLE advertising on the K32W0x platform by pressing User Interface button;
- Run the following command to commission the device to the Thread Network:

$./chip-tool pairing ble-thread {NODE_ID_TO_ASSIGN}{ACTIVE_DATASET}{SETUP_PIN_CODE} {DISCRIMINATOR}

Where:

- {NODE_ID_TO_ASSIGN} - the matter node id;
- {ACTIVE_DATASET} - Thread operational active dataset. The format is hex:xxxxxx).
- {SETUP_PIN_CODE} – pin code, default value is 20202021;
- {DISCRIMINATOR} – discriminator, default value is 3840;

Example:

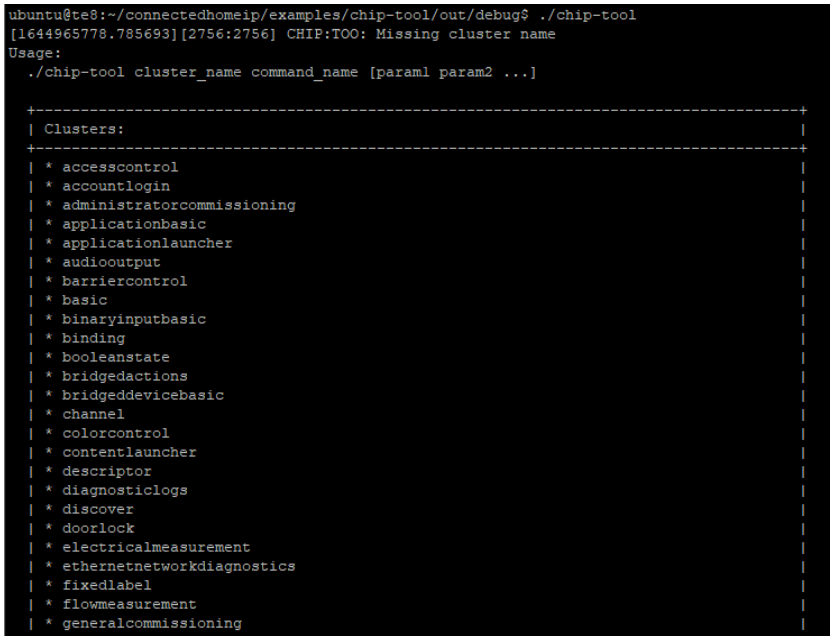
$./chip-tool pairing ble-thread 1 hex:0e080000000000010000000300000f35060004001fffe00208111111112222222220708fd8e93c50ace6eae051 00011223344556677889 9aabbccddeeff030e4726561644656d6f01021234041061e1206d2c2b46e079eb775f41f c72190c0402a0fff8 20202021 3840

Sending cluster commands to K32W0x1:

These commands are sent from the imx8m mini shell terminal.

The format of the cluster command is:

$./chip-tool <cluster_name> <command_name> <param1, param2 ..>



Example of usage:

- Send OnOff cluster -> Toggle command to node ID for cluster endpoint
    - *chip-tool onoff toggle <node_id> <cluster_endpoint>*

$./chip-tool onoff toggle 1 1

- Send OnOff cluster -> On command to node ID for cluster endpoint
    - *chip-tool onoff on <node_id> <cluster_endpoint>*

$./chip-tool onoff on 1 1

- Send OnOff cluster -> Off command to node ID for cluster endpoint
    - *chip-tool onoff off <node_id> <cluster_endpoint>*

$./chip-tool onoff off 1 1

- Send Attribute reporting configuration for on-off attribute using min/max interval to node ID for cluster endpoint
    - *chip-tool onoff report on-off <min_interval> <max_interval> <node_id> <cluster_endpoint>*

$./chip-tool onoff report on-off 300 301 1 1

- Read Channel Attribute from the Thread Diagnostic cluster from node ID
    - *chip-tool threadnetworkdiagnostics read channel <node_id> <cluster_endpoint>*

$./chip-tool threadnetworkdiagnostics read channel 1 0

- Read fabrics list based on the basic cluster from node ID
    - *chip-tool operationalcredentials read fabrics-list <node_id> <cluster_endpoint>*

$./chip-tool operationalcredentials read fabrics-list 1 0

6 K32W0X1 - EXPLORING WITH MATTER

6.1 Adding Manufacturing data to the application

By default, Matter repository is set to use test certificates found in the credentials folder. They are used in conjunction with default provisioning data and are stored in the embedded application code.

The following workflow describes how to change the certificates and provisioning data to the user's own manufacturing data.

Further details about this feature can be found at https://github.com/NXPmicro/matter/blob/v1.0-branch-nxp/examples/platform/nxp/doc/manufacturing_flow.md

Generate new certificates

This step allows the user to generate a custom Device Attestation Certificate (DAC), Product Attestation Intermediate (PAI) certificate and Product Attestation Authority (PAA) certificate.

This step can be skipped if these certificates are already available from a third-party entity.

Please note that for real production manufacturing data, the "production PAA" certificate authenticity is determined via the Distributed Compliance Ledger (DCL) rather than thorough a PAA certificate generated alongside a DAC and PAI certificate. The PAI certificate may also have a different lifecycle.

As prerequisite for this step, it is required to have the chip-cert compiled in the matter repository. This can be obtained by executing the following commands from the Matter root directory:

$ source scripts/activate.sh
$ gn gen out/host
$ ninja -C out/host

The script to be used for this step is found in the following folder as bash script *generate_cert.sh*:

<matter_root>*/scripts/tools/nxp/*

The output of this script will be a DAC, a PAI and a PAA. User must adjust the PAI_VID and PAI_PID values in the script according to their own vendor ID (VID) and product ID (PID).

$ ./ scripts/tools/nxp/generate_cert.sh ./src/tools/chip-cert/out/chip-cert

NXP provides demo certificates for testing with NXP vendor and product IDs that can be found at *<matter_root>/scripts/tools/nxp/demo_generated_certs.*

Generate new provisioning data:

This step is to generate new provisioning data and convert all the data to a binary that can be written in the internal flash of the K32W0x1 chip. This example is without any encryption on the data in the binary:

$ python3 ./scripts/tools/nxp/generate_nxp_chip_factory_bin.py -i 10000 -s UXKLzwHdN3DZZLBaL2iVGhQi/OoQwIwJRQV4rpEalbA= -p 14014 -d 1000 --dac_cert /path/to/certs/Chip-DAC-Generated-Cert.der --dac_key /path/to/certs/Chip-DAC-Generated-Key.der --pai_cert /path/to/certs/Chip-PAI-Generated-Cert.der --spake2p_path /out/host/spake2p --out out.bin

Same example as above, but with an already generated verifier passed as input:

$ python3 ./scripts/tools/nxp/generate_nxp_chip_factory_bin.py -i 10000 -s UXKLzwHdN3DZZLBaL2iVGhQi/OoQwIwJRQV4rpEalbA= -p 14014 -d 1000 --dac_cert /path/to/certs/Chip-DAC-Generated-Cert.der --dac_key /path/to/certs/Chip-DAC-Generated-Key.der --pai_cert /path/to/certs/Chip-PAI-Generated-Cert.der --spake2p_path ./src/tools/spake2p/out/spake2p --spake2p_verifier ivD5n3L2t5+zeFt6SjW7BhHRF30gFXWZVvvXgDxgCNcE+BGuTA5AUaVm3qDZBcMMKn1a6CakI4SxyPUnJr0CpJ4pwpr0DvpTlkQKqaRvkOQfAQ1XDyf55DuavM5KVGdDrg== --out out.bin

Here is the interpretation of the parameters:

-i	*-> SPAKE2+ iteration*
-s	*-> SPAKE2+ salt (passed as base64 encoded string)*
-p	*-> SPAKE2+ passcode*
-d	*-> discriminator*
--dac_cert	*-> path to the DAC (der format) location*
--dac_key	*-> path to the DAC key (der format) location*
--pai_cert	*-> path to the PAI (der format) location*
--spake2p_path	*-> path to the spake2p tool (compile it from ./src/tools/spake2p)*
--out	*-> name of the binary that will be used for storing all the generated data*
--aes128_key	*-> 128 bits AES key used to encrypt the whole dataset*
--spake2p_verifier	*-> SPAKE2+ verifier (passed as base64 encoded string). If this option is set, all SPAKE2+ inputs will be encoded in the final binary. The spake2p tool will not be used to generate a new verifier on the fly.*

Write the provision data to the internal flash:

Write out.bin to the internal flash of the K32W0x1 chip at location 0x9D200. This can be done using the FTDI USB connection on the DK6 board and the DK6Programmer software. Refer to the Hardware connection in chapter 4.3 for more information on installing the software.

Open a Windows Command Prompt and issue the following command.

C:\nxp\DK6ProductionFlashProgrammer\DK6Programmer.exe -Y -V2 -s <COM_PORT> -P 1000000 -Y -p FLASH@0x9D200="out.bin"

Generate a new Certificate Declaration (CD)

The script that generates the CD can be found at credentials/test/gen-test-cds.sh. In order to regenerate the CD with the correct data, user needs to update the vids, pid0 and device_type_id parameters with the appropriate values. The default Chip-Test-CD-Signing-* key and certificate can be found at credentials/test/certification-declaration. This CD can act as CSA certificate sand is hard-coded as Trust Anchor in the current chip-tool version. To use this certificate and avoid generating a new one, lines 69-70 must be commented in the gen-test-cds.sh script (the ones that are generating a new CD signing authority).

The command to run the actual CD generation is:

$ ./credentials/test/gen-test-cds.sh ./src/tools/chip-cert/out/chip-cert

SET the Vendor and Product IDs

Set the correct VID/PID and CD in the *examples/<example_app>/nxp/k32w/k32w0/ChipProjectConfig.h* file VID and PID values should correspond to the ones used in the DAC.

The following command can be used to obtain a byte array containing the Certificate Declaration:

$ hexdump -ve '1/1 "0x%.2x, "' path/to/generated/cd/Chip-Test-CD-Generated.der

Building the Application

As stated in the Matter application Building instructions from chapter 4.1, the gn command needs to be adjusted to support the factory data, by adding *chip_with_factory_data=1* to the command.

The command should look like the following:

*$gn gen out/debug --args="k32w0_sdk_root=\"${NXP_K32W0_SDK_ROOT}\" chip_with_OM15082=1 chip_with_ot_cli=0 is_debug=false chip_crypto=\"tinycrypt\" chip_with_se05x=0 chip_pw_tokenizer_logging=true mbedtls_repo=\"//third_party/connectedhomeip/third_party/nxp/libs/mbedtls\" **chip_with_factory_data=1**"*

Build the application using the command:

$ ninja -C out/debug

Running Commissioning using Manufacturer Data

The PAA certificate generated at step 1 in "*Generate new certificates*" needs to be copied to the device that acts as Matter controller. For example, using scp, user can copy with the command:

$ scp /path/to/generated/certificates/Chip-PAA-Generated-Cert.der root@<ip_of_matter_controller>:/path/to/paa/directory

Open a connection to the Matter controller and run chip-tool with a new PAA:

./chip-tool pairing ble-thread {NODE_ID_TO_ASSIGN} {ACTIVE_DATASET} {SETUP_PIN_CODE} {DISCRIMINATOR} --paa-trust-store-path /path/to/paa/directory

For example:

./chip-tool pairing ble-thread 1 hex:$hex_value 14014 1000 --paa-trust-store-path /path/to/paa/directory

Where:

- *paa-trust-store-path -> path to the generated PAA (der format). Avoid placing other certificates in the same location as this may confuse chip-tool.*
- *14014 is the discriminator*
- *1000 is the passcode. This is needed for testing self-generated DACs, but likely not required for "true production" with production PAI issued DACs*

Note that the descriminator entered above must match exactly the descriminator value that was selected when the DAC was generated.

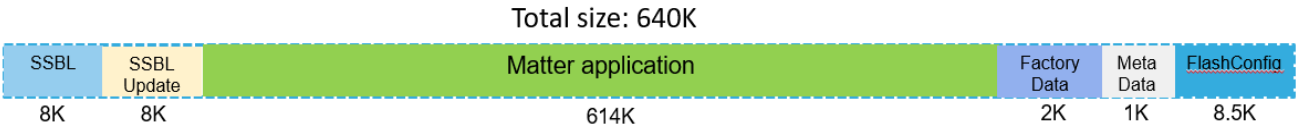
6.2 Over the air upgrade – Flash Configuration

The support for over-the-air upgrades is enabled by adding chip_enable_ota_requestor to the build command. By default this feature is enabled on the k32w0x1 Matter examples in the args.gni file, otherwise can be enabled like below:

$ gn gen out/debug --args="k32w0_sdk_root=\"${NXP_K32W0_SDK_ROOT}\" chip_with_OM15082=1 ***chip_enable_ota_requestor=1*** *chip_with_ot_cli=0 is_debug=false chip_crypto=\"tinycrypt\" chip_with_se05x=0 chip_pw_tokenizer_logging=true mbedtls_repo=\"//third_party/connectedhomeip/third_party/nxp/libs/mbedtls\""*

Flash configuration

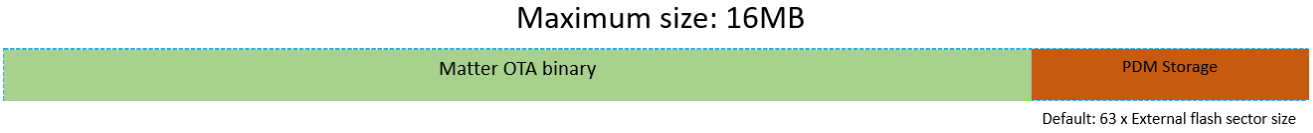
The internal flash must be prepared for the OTA process:



Total flash size available on K32W0x1 internal memory is 640KB

- SSBL and SSBL update region – needed for OTA (mandatory in Matter 1.0) – first 16k are reserved for the Bootloader;
- FlashConfig – Flash Configuration data that outlines that supported Flash regions, and their corresponding base addresses/sizes. User needs to take care handling this section and not overwrite it!
- Meta Data: e.g.: image signature
- Factory data, including certificates and other security related data
- Matter remains with 614K available for .text (code) and .rodata (constant strings)

The external flash memory is also used for over the air updates and has the following mapping:



6.3 Reading Tokenizer logs

The debug logging is controlled by the chip_pw_tokenizer_logging build parameter issued at build time. If this is not set or was set to false, then the user can open a serial connection from the host PC with a baudrate of 115200 bps without any changes.

The chip_pw_tokenizer_logging build option can be set to reduce the overall size of the firmware image while enabling debug logging statements by replacing ASCII character strings with encoded (or tokenized) strings.

If chip_pw_tokenizer_logging is set to true, then the detokenizer script must be used. This script decod es the tokenized logs either from a file or from a serial port. The detokenizer script is available in the following path, from the Matter root directory:

examples/platform/nxp/k32w/k32w0/scripts/detokenizer.py

The token database is created automatically after building the binary if the argument *chip_pw_tokenizer_logging*=true was used.

The script can be used in the following ways:

detokenizer.py serial [-h] -i INPUT -d DATABASE [-o OUTPUT]

detokenizer.py file [-h] -i INPUT -d DATABASE -o OUTPUT

Where:

- The first parameter is either serial or file and it selects between decoding from a file or from a serial port.
- *-i INPUT* - used to set the path of the file or the serial to decode from.
- *-d DATABASE* - represents the path to the token database to be used for decoding. The default path is out/debug/chip-k32w0x-light-example-database.bin after a successful build.
- *-o OUTPUT* - represents the path to the output file where the decoded logs will be stored. This parameter is required for file usage and optional for serial usage. If not provided when used with serial port, it will show the decoded log only at the stdout and not save it to file.

The detokenizer script must be run inside the example's folder after execution of the scripts/activate.sh script. The pw_tokenizer module used by the script is loaded by the activation script.

An example of running the detokenizer script to see logs of a lighting app:

$ python3 ../../../../../examples/platform/nxp/k32w/k32w0/scripts/detokenizer.py serial -i /dev/ttyACM0 -d out/debug/chip-k32w0x-light-example-database.bin -o device.txt

6.4 ZAP tool. Update Cluster/Endpoint functionality

ZAP - ZCL Advanced Platform is a generic generation engine and user interface for applications and libraries based on Zigbee Cluster Library.

It provides a user interfacefor developers to select specific application configuration (endpoints, clusters, attributes, commands) and perform SDK specific generation of artifacts based on the ZCL specifation and customer provided app configuration.

Building and running ZAP tool:

It is recomanded to always use the latest version, which at the time of publication of this document is (v16.17.1)

The ZAP tool requires nodejs to be installed, and this cam be done by executing the following commands:

- Download the latest version of nodejs
- Unzip the binary archive to any directory you wanna install Node:

$sudo mkdir -p /usr/local/lib/nodejs
$sudo tar -xJvf node-v16.15.1-linux-x64.tar.xz -C /usr/local/lib/nodejs

- Set the environment variable ~/.profile, add the text below to the end

$sudo nano ~/.profile
 # Nodejs
VERSION=v16.15.1
DISTRO=linux-x64
export PATH=/usr/local/lib/nodejs/node-v16.15.1-linux-x64/bin:$PATH

- Refresh the profile

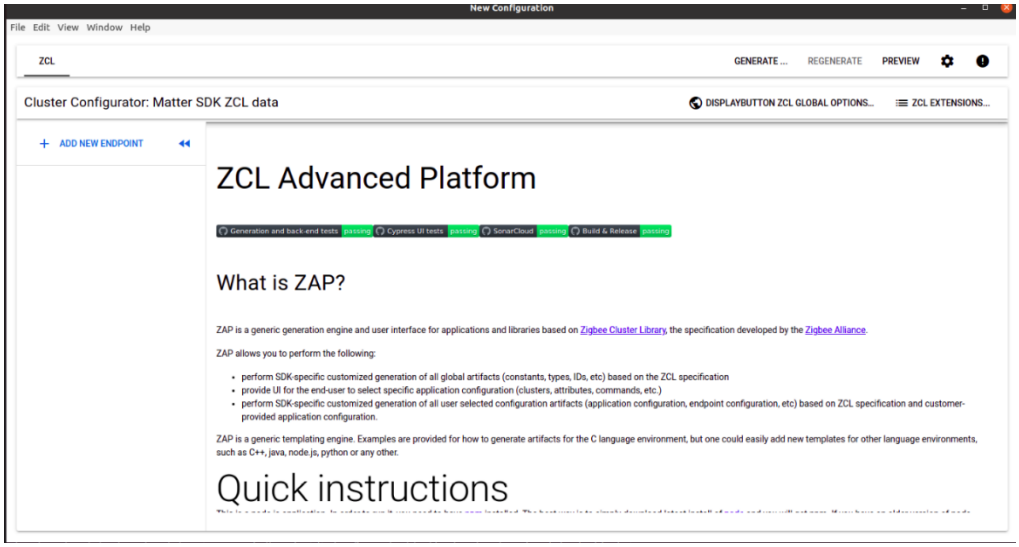
$. ~/.profile

Running the ZAP-Tool

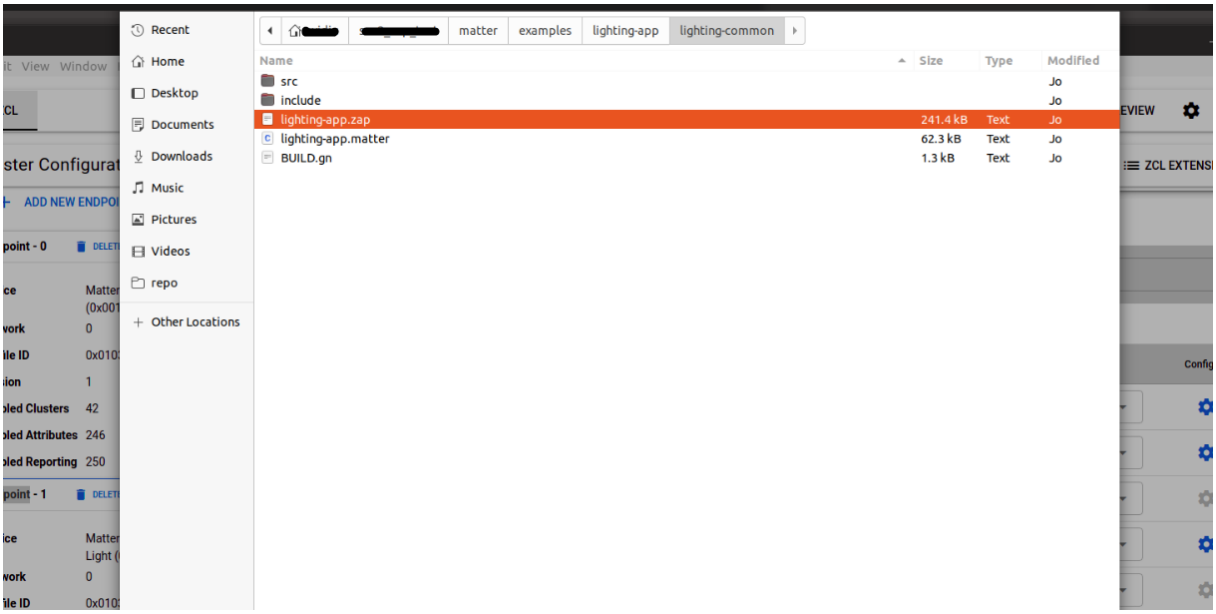
- Install the zap-tool from the Matter github repositroy:
- Open a terminal and navigate to the Matter root directory

*$cd *path_to_matter_sdk*/matter*
$source ./scripts/activate.sh
$./scripts/tools/zap/run_zaptool.sh

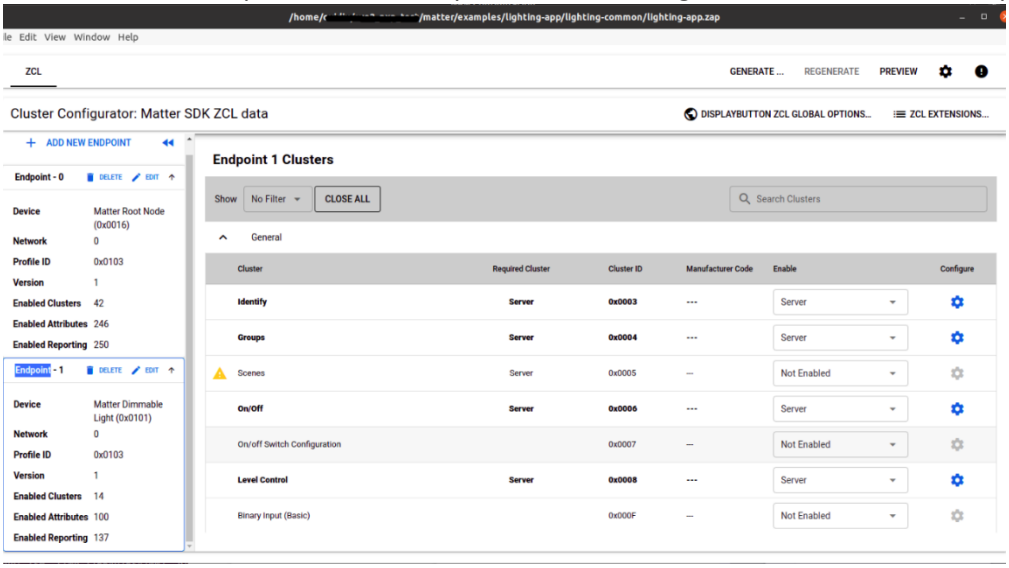
- The first time this script is executed will take longer, as zap-tool is being installed
- If the installation is succesfull you will see the below feedback on the terminal and ZAP-Tool will open:



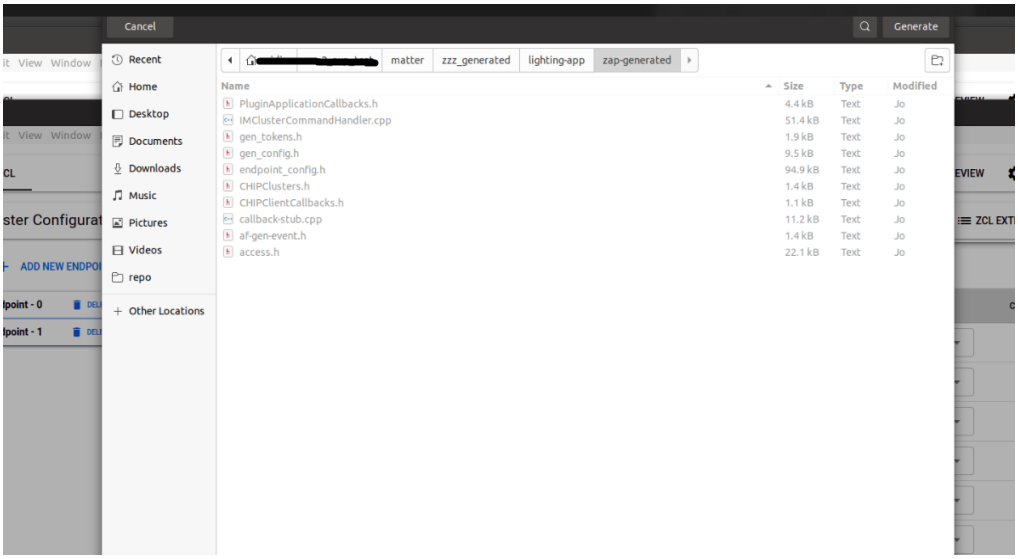
- Next, open the ZAP file for the corresponding project. For Lighting-app -> Go to *…/matter/examples/lighting-app/lighting-common* and select the lighting-app.zap file:



- A new ZAP-tool window will open with the endpoint/cluster configuration for the selected application:



- After performing all the changes, generate the new configuration -> by pressing the *Generate* button and save the new configuration in the …/matter/zzz_generated/lighting-app/zap-generated folder:



- Rebuild the application to apply the new configuration.