NXP Semiconductors	Document Number: SLN-VIZNAS-IOT-DG
Developer's Guide	Rev 1.2, 11/2020

MCU VIZNAS Solution Developer's Guide

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 Introduction

NXP's MCU-based SLN-VIZNAS-IOT development kit provides OEMs with a fully integrated, self-contained, software and hardware solution. This includes the i.MX RT106F run-time library and pre-integrated machine learning face recognition algorithms, as well as all required drivers for peripherals, such as camera and memories.

This cost-effective, easy-to-use face recognition implementation facilitates the demand for a face-based Friction Free Interface that can be embedded in a variety of products across home, commercial and industrial applications, thus eliminating the need to use hard to learn and time-consuming mechanisms to identify users.

TARGET APPLICATIONS

- **Safety/Security/Alarm devices:** E-locks, Alarm panels, remote sensors, and automated access
- **Smart appliances:** Washing machines, dryers, ovens, refrigerators, stoves, and dishwashers
- **Home comfort devices**: Thermostats, remote temperature sensors, and lighting
- **Counter-top appliances:** Microwaves, coffee machines, rice cookers, and blenders
- **Smart industrial devices:** Power tools, ergonomic stations, machine access and authorization

1.1 RT106F VISION CROSSOVER PROCESSOR OVERVIEW

The i.MX RT106F is an EdgeReady member of the i.MX RT1060 family of crossover processors, targeting low cost embedded face recognition applications. It features NXPs advanced implementation of the Arm® Cortex®-M7 core, which operates at speeds up to 600 MHz to provide high CPU performance and best real-time responses. This i.MX RT106F based solution enables system designers to easily and inexpensively add face recognition capabilities to a wide variety of smart appliances, smart homes, FRICTION FREE INTERFACE VISION HARDWARE and smart industrial devices. The i.MX RT106F processor is licensed to run NXPs i.MX RT run-time library for face recognition which may include:

- Camera drivers
- Image capture
- Image pre-processing
- Face alignment
- Face detection
- Face recognition
- Liveness Detection
- Emotion recognition

System Requirements and Prerequisites

Once you're ready to begin development, you will need to download MCUXpresso IDE. The current SDK is tested with version **11.2.1 of MCUXpresso IDE** and Segger J-Link **v6.6x**.

https://www.nxp.com/support/developer-resources/software-development-tools/mcuxpresso-software-and-tools/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE

Computer type	OS version	Terminal
Apple	Mac OS	PuTTY
PC	Windows 10	PuTTY/Tera Term
PC	Linux	PuTTY

Table 1: Supported Computer Configurations

Usage Condition

The following information is provided per Article 10.8 of the Radio Equipment Directive 2014/53/EU:

(a) Frequency bands in which the equipment operates.

(b) The maximum RF power transmitted.

PN	RF Technology	(a) Freq Range	(b) Max Transmitted Power
SLN-VIZNAS-IOT	Wi-Fi	2412MHz-2472MHz	17.9dBm

Table 2: Wi-Fi Frequency & Power

EUROPEAN DECLARATION OF CONFORMITY (Simplified DoC per Article 10.9 of the Radio Equipment Directive 2014/53/EU)

This apparatus, namely SLN-VIZNAS-IOT, conforms to the Radio Equipment Directive 2014/53/EU. The full EU Declaration of Conformity for this apparatus can be found at this location: https://www.nxp.com/

To maintain EMC Compliance, the CLOCK_DRIVE_STRENGTH_LOW preprocessor macro* must remain set to 1

***NOTE: See* **Preparing Bootstrap, Bootloader, and Main Application Images** *for an example of changing preprocessor macros*

2 Getting to Know the SLN-VIZNAS-IOT

2.1 Hardware Overview

The SLN-VIZNAS-IOT kit is intended to provide a reference for a real product design. The kit is designed using a small form factor which takes into account many of the design considerations a hardware engineer would make when creating a product. With that said, NXP has also fashioned the hardware to have some of the key hallmarks of a traditional development kit.

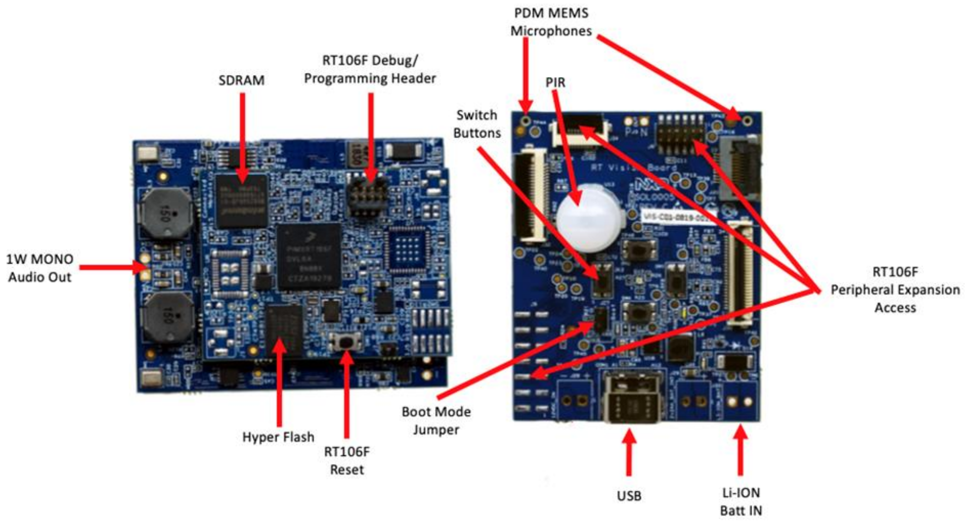


Figure 1: SLN-VIZNAS-IOT Base Board + Expansion Board Peripherals

Additionally, the SLN-VIZNAS-IOT kit comes with an IR+RGB **Dual Camera Adapter** like that shown below for use in secure applications.

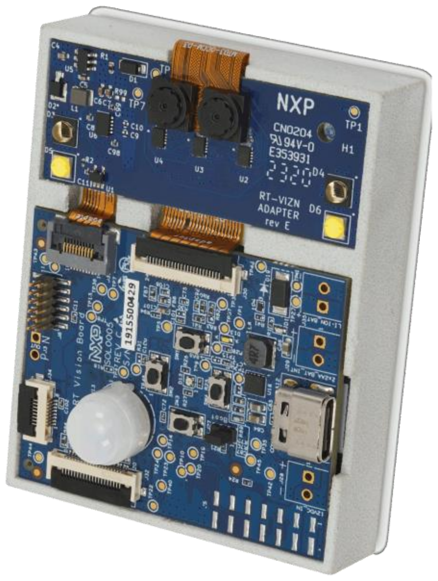


Figure 2: Dual Camera Adapter Kit Configuration

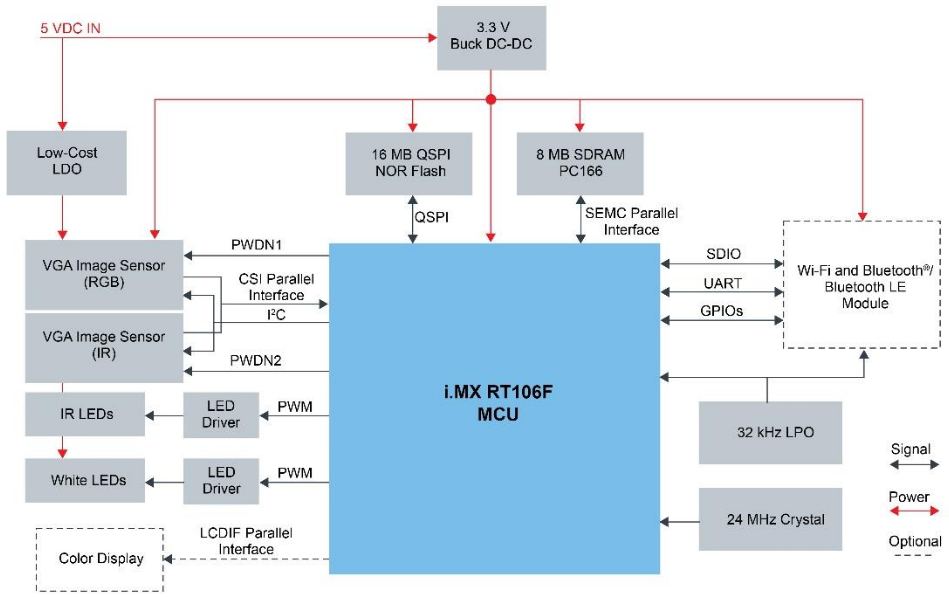


Figure 3: VIZNAS HW Block Diagram

2.2 Software Overview

The SLN-VIZNAS-IOT kit has been built and designed in such a way that enables best security practices while keeping a development kit feel. The main security mechanism that has been implemented is a series of image verification stages that are required for every image programmed onto the device. The sections below guide you through the overall software and security architectures and the implications they have during the development and production phases of your product development.

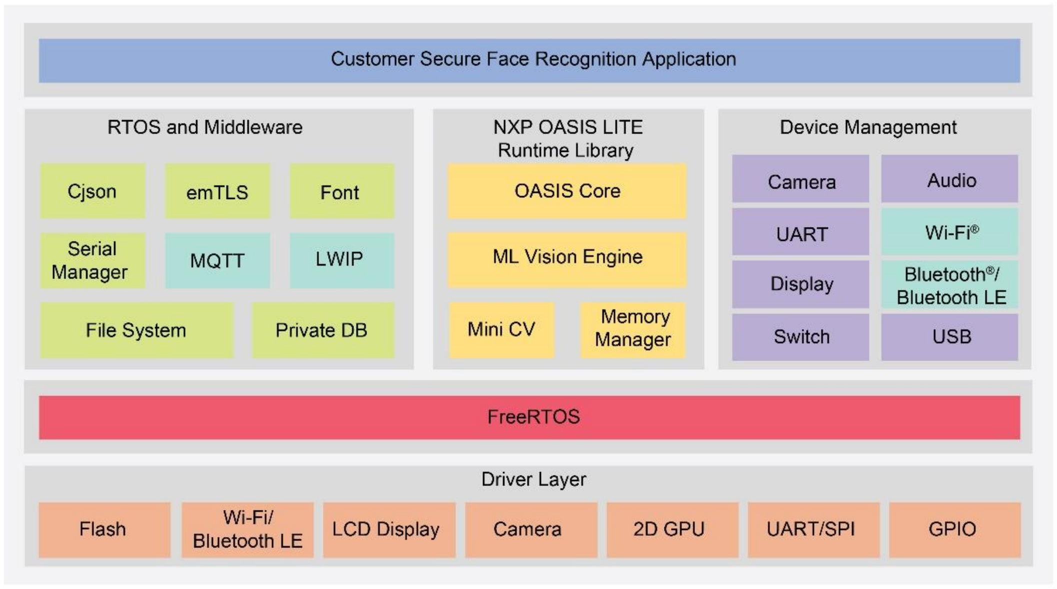


Figure 4: VIZNAS SW Block Diagram

2.3 Device Memory Map

To understand the various pieces of the system, it helps to see the memory map that NXP has developed for this application. There are many components required in the system to successfully boot and execute an application. A few of these sectors will be described in greater detail below.

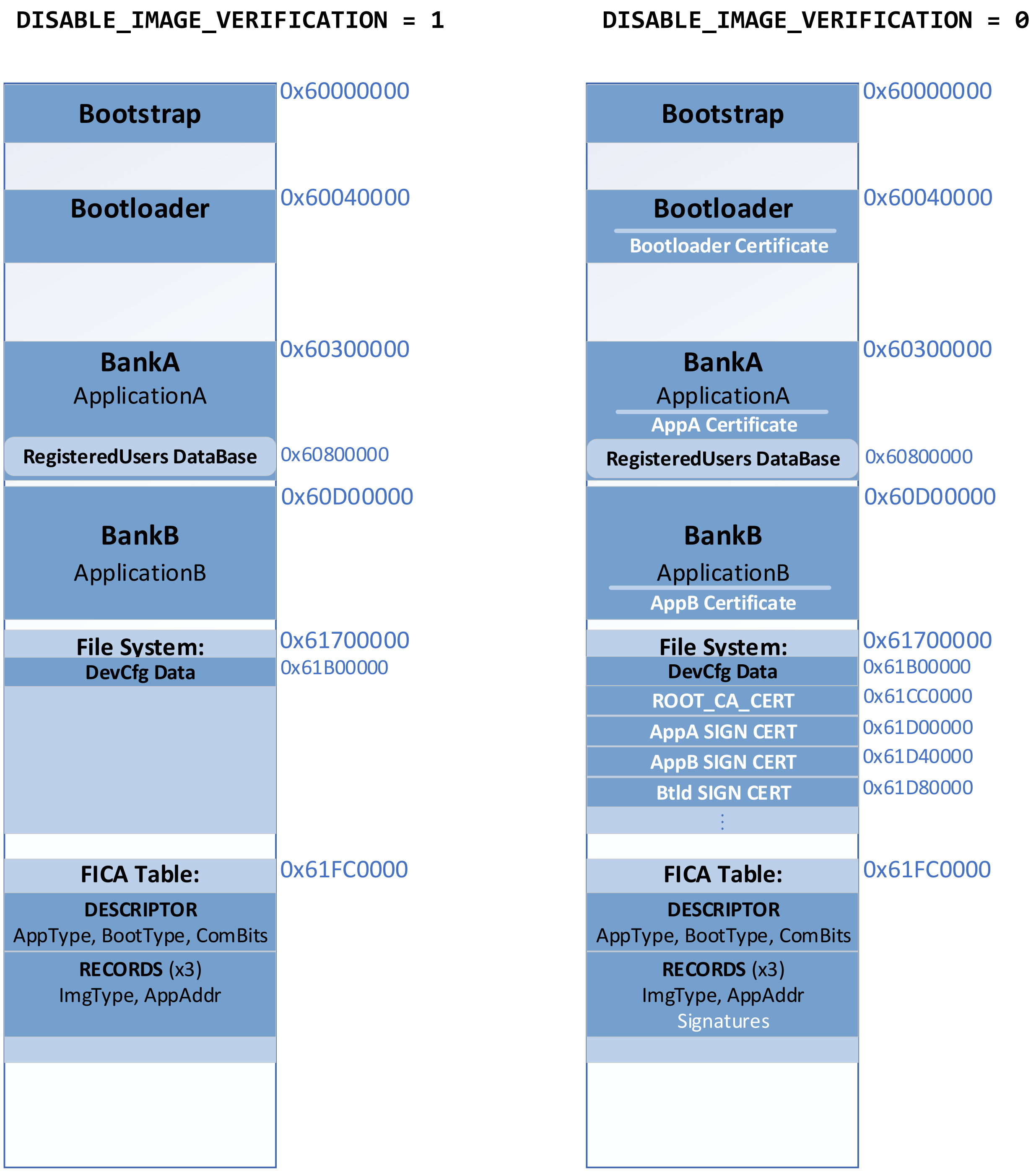


Figure 5: Device Memory Map

2.4 Security Architecture

The following figure shows the series of checks that occur at boot time. Configuration options in the various applications (ROM bootloader, bootstrap, bootloader) will determine which sequence is followed. **<u>The state of the board from factory is with all security checks disabled.</u>**

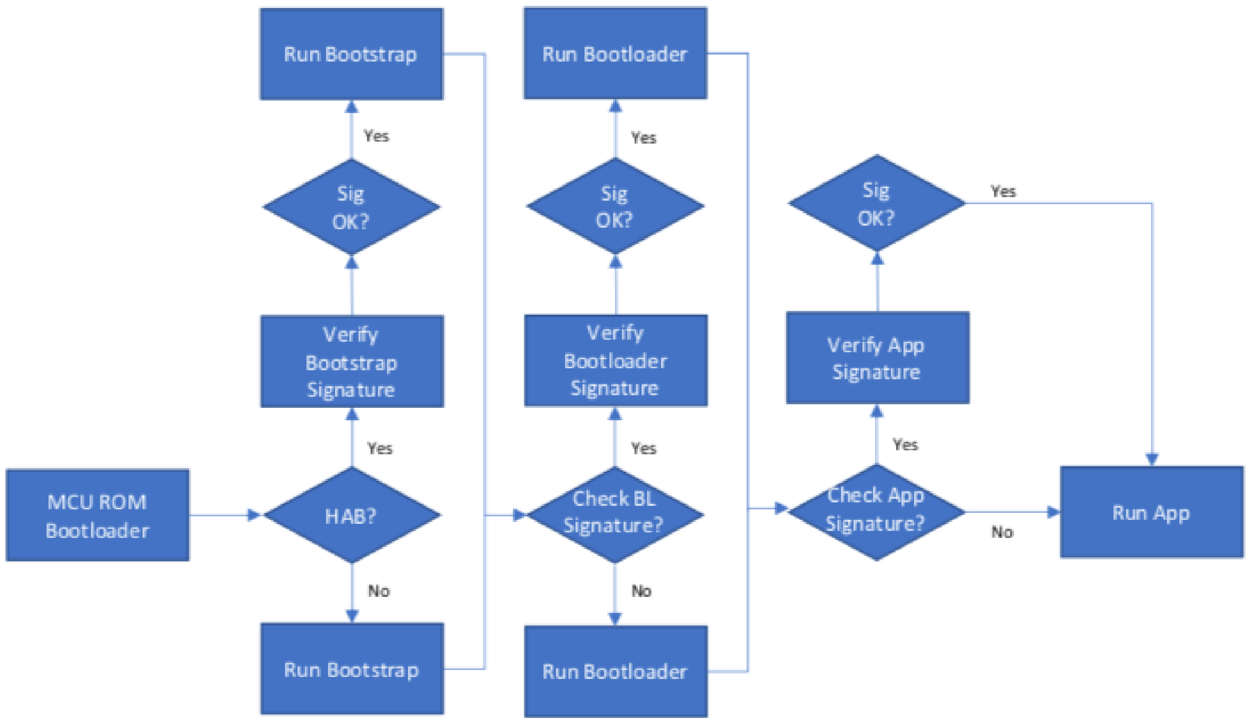


Figure 6: Boot Security Flow Chart

If at any point a signature check fails (in the case where HAB or image verification is enabled), the boot process stops.

2.4.1 Application Chain of Trust

The basis of the security architecture implemented in the SLN-VIZNAS-IOT is signed application images. Signing requires the use of a **Certificate Authority** (**CA**). NXP has its own CA for signing applications at the factory, but the CA is not something that is shared with customers.

The CA is used to create signing entities for the bootloader and application. A certificate from the CA is stored in the SLN-VIZNAS-IOT's filesystem and is used to verify the signatures of the signing entity certificates. In addition, locally stored certificates from the signing entities are used to verify the signature of firmware images coming in over the OTA/OTW bootloader interface.

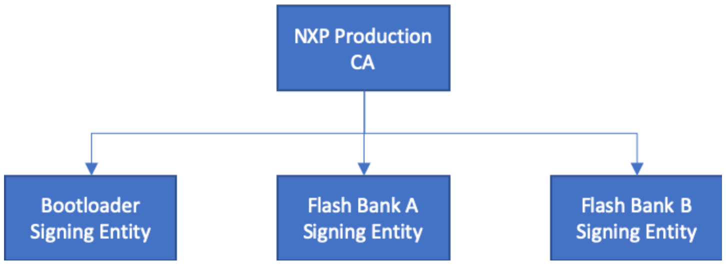

NXP Production CA
Bootloader Signing Entity
Flash Bank A Signing Entity
Flash Bank B Signing Entity

Figure 7: Signing Entities

2.4.2 Flash Image Configuration Area (FICA) and Image Verification

The FICA table is a section inside the filesystem that is responsible for describing the images that will be booted. It contains information about the image and signatures of the applications that will be used to ensure that only verified firmware is executed. This ensures malicious images cannot be executed without first being signed by the certificate authority and certificate that is programmed into the filesystem. Before any image is jumped to, it is first verified using the signature from its associated FICA entry.

For example, in the standard boot flow shown in Figure 6:

- The **bootstrap** will use the bootloader FICA entry to validate the bootloader
- The **bootloader** will use the AppA FICA entry to validate the AppA image
- The **bootloader** will use the AppB FICA entry to validate the AppB image

For final production, the solution provides programming scripts to enable i.MX RT High Assurance Boot (HAB) to verify and protect the bootstrap component. It is recommended that users enable HAB for their end product.

The downside of having this security protection enabled is that programming new images can be a little more complex as it requires signature generation. Taking in consideration that this flow may be time consuming and not required for basic development tasks, NXP introduced some bypasses to make the job easier for developers. **<u>These bypasses should not be deployed in production.</u>**

Again, the default configuration of the SLN-VIZNAS-IOT is to have HAB disabled and signature verifications bypassed. This is to ensure a smooth development experience.

2.4.3 Image Certificate Authority (CA) and Application Certificates

The SLN-VIZNAS-IOT kit comes pre-programmed with signed images (though signature verification is bypassed by default) as indicated in the **<u>Flash Image Configuration Area (FICA) and Image Verification</u>** section. The **bootloader_4343W** and **elock_oobe** application are signed using NXP's test CA and can be used to ensure the authenticity of all images which are intended to be booted.

The application signing certificates are located at the following locations in the filesystem:

- Address **0x61D00000** for Application Bank A
- Address **0x61D40000** for Application Bank B
- Address **0x61D80000** for the Bootloader

The certificate for the CA (used to verify the application signing certificates) is located at address **0x61CC0000** in the filesystem. For more detail, see Figure 5: Device Memory Map.

These certificates are used when **Image Verification** (see section **Enabling Image Verification**) is enabled to validate the signature provided when performing a secure update.

3 Get Started with MCUXpresso Tool Suite

The following section describes the steps required to setup the environment and prepare it for development.

3.1 MCUXpresso IDE

MCUXpresso IDE brings developers an easy-to-use Eclipse-based development environment for NXPs microcontrollers based on Arm® Cortex®-M cores. It offers advanced editing, compiling and debugging features with the addition of MCU-specific debugging views, code trace and profiling, multicore debugging, and integrated configuration tools. Its debug connections support every NXP evaluation boards with industry-leading open-source and commercial debug probes from ARM®, P&E Micro® and SEGGER®.

NOTE: The SLN-VIZNAS-IOT requires MCUXpresso IDE version 11.2.1 or greater.

1. To download **NXP MCUXpresso IDE** for free go online to: www.nxp.com/MCUXpresso

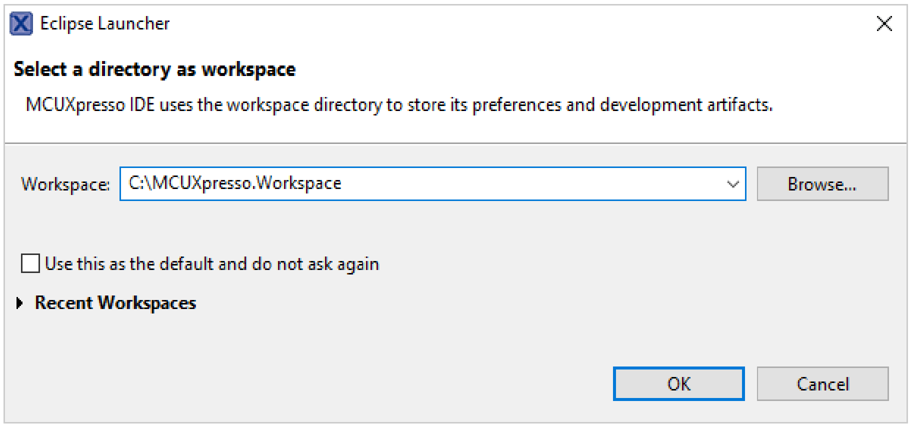
2. Select **MCUXpresso IDE** from the **PRODUCTS** tab.

3. Go to **DOWNLOADS** tab and select the **LATEST VERSION** of the tool.

If you do not already have one, you will be asked to sign-in/up with a free NXP user-account.

4. When MCUXpresso installer download completes, **double click on the executable**, follow the install instructions and **keep the default options**.

5. Launch MCUXpresso IDE and define the Workspace location where you will copy and store your projects (default **C:\MCUXpresso.Workspace**) and **press OK**.



MCUXpresso IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\MCUXpresso.Workspace

☐ Use this as the default and do not ask again

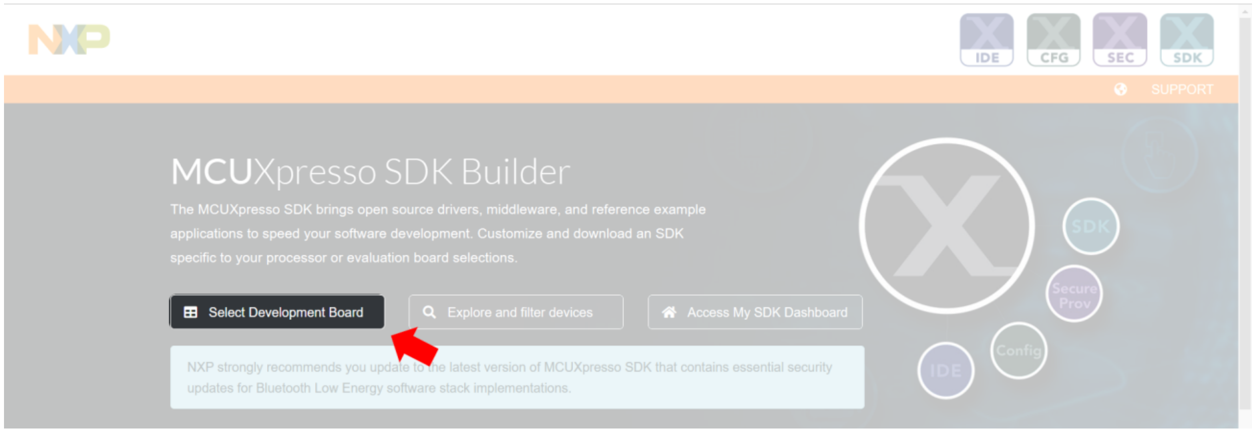
Figure 8: MCUXpresso IDE Workspace

3.2 Install the SDK

MCUXpresso SDK is a comprehensive software enablement package designed to simplify and accelerate application development with NXPs microcontrollers based on Arm® Cortex®-M cores. The MCUXpresso SDK includes production-grade software with integrated RTOS (optional), integrated stacks and middleware, reference software, and more. It is available in custom downloads based on user selections of MCU, evaluation board, and optional software components.

Before building the **SLN-VIZNAS-IOT SDK** example projects, the target SDK needs to be imported into MCUXpresso IDE.

The **MCUXpresso SDK** for the SLN-VIZNAS-IOT can be downloaded from NXP's SDK Builder by clicking **Select Development Board** and searching for "SLN-VIZNAS-IOT" in the search box.



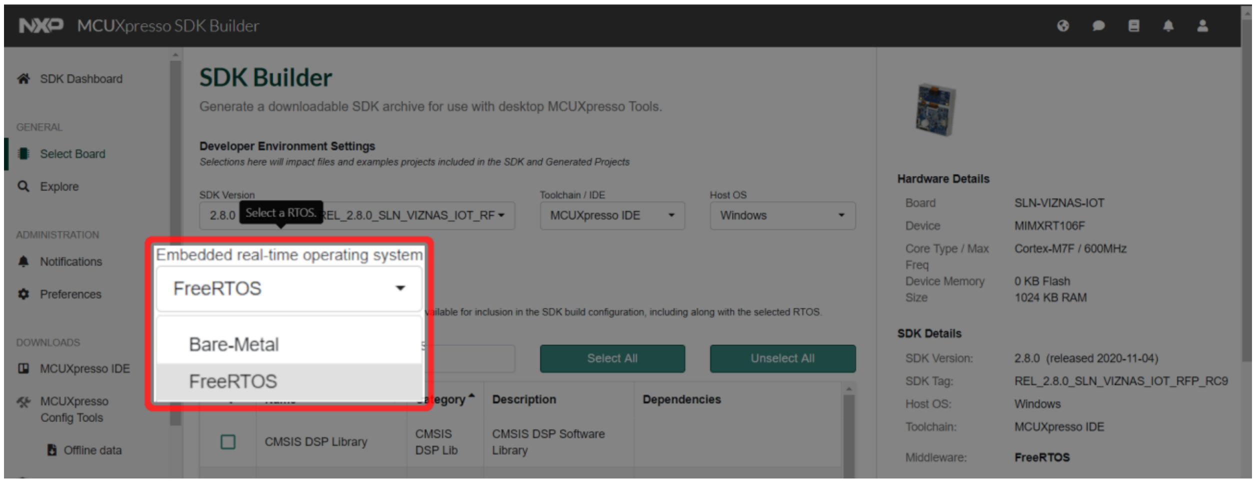
MCUXpresso SDK Builder

The MCUXpresso SDK brings open source drivers, middleware, and reference example applications to speed your software development. Customize and download an SDK specific to your processor or evaluation board selections.

NXP strongly recommends you update to the latest version of MCUXpresso SDK that contains essential security updates for Bluetooth Low Energy software stack implementations.

Figure 9: Select Development Board

When building the SDK, be sure to select **FreeRTOS** under **Embedded real-time operating system.**



SDK Builder

Generate a downloadable SDK archive for use with desktop MCUXpresso Tools.

Developer Environment Settings

Selections here will impact files and examples projects included in the SDK and Generated Projects

SDK Version: 2.8.0 (REL_2.8.0_SLN_VIZNAS_IOT_RF) | Toolchain / IDE: MCUXpresso IDE | Host OS: Windows

Embedded real-time operating system: FreeRTOS

...ailable for inclusion in the SDK build configuration, including along with the selected RTOS.

	Name	Category	Description	Dependencies
☐	CMSIS DSP Library	CMSIS DSP Lib	CMSIS DSP Software Library	

Hardware Details

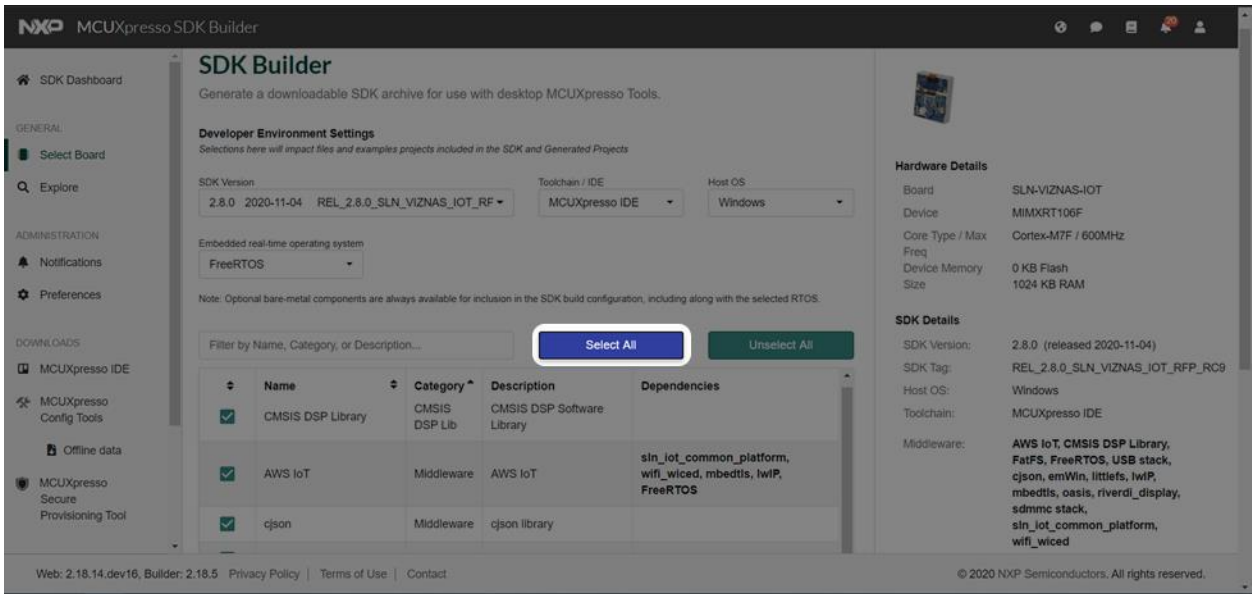
Board	SLN-VIZNAS-IOT
Device	MIMXRT106F
Core Type / Max Freq	Cortex-M7F / 600MHz
Device Memory Size	0 KB Flash 1024 KB RAM

SDK Details

SDK Version:	2.8.0 (released 2020-11-04)
SDK Tag:	REL_2.8.0_SLN_VIZNAS_IOT_RFP_RC9
Host OS:	Windows
Toolchain:	MCUXpresso IDE
Middleware:	FreeRTOS

Figure 10: Choose Embedded real-time operating system

After updating the operating system, be sure to click **Select All** to make sure all required components get added.



NXP MCUXpresso SDK Builder

SDK Dashboard
GENERAL
Select Board
Explore
ADMINISTRATION
Notifications
Preferences
DOWNLOADS
MCUXpresso IDE
MCUXpresso Config Tools
Offline data
MCUXpresso Secure Provisioning Tool

SDK Builder

Generate a downloadable SDK archive for use with desktop MCUXpresso Tools.

Developer Environment Settings
Selections here will impact files and examples projects included in the SDK and Generated Projects

SDK Version: 2.8.0 2020-11-04 REL_2.8.0_SLN_VIZNAS_IOT_RF
Toolchain / IDE: MCUXpresso IDE
Host OS: Windows
Embedded real-time operating system: FreeRTOS

Note: Optional bare-metal components are always available for inclusion in the SDK build configuration, including along with the selected RTOS.

Filter by Name, Category, or Description...
Select All
Unselect All

	Name	Category	Description	Dependencies
✓	CMSIS DSP Library	CMSIS DSP Lib	CMSIS DSP Software Library	
✓	AWS IoT	Middleware	AWS IoT	sln_iot_common_platform, wifi_wiced, mbedtls, lwIP, FreeRTOS
✓	cjson	Middleware	cjson library	

Hardware Details

Board	SLN-VIZNAS-IOT
Device	MIMXRT106F
Core Type / Max Freq	Cortex-M7F / 600MHz
Device Memory Size	0 KB Flash 1024 KB RAM

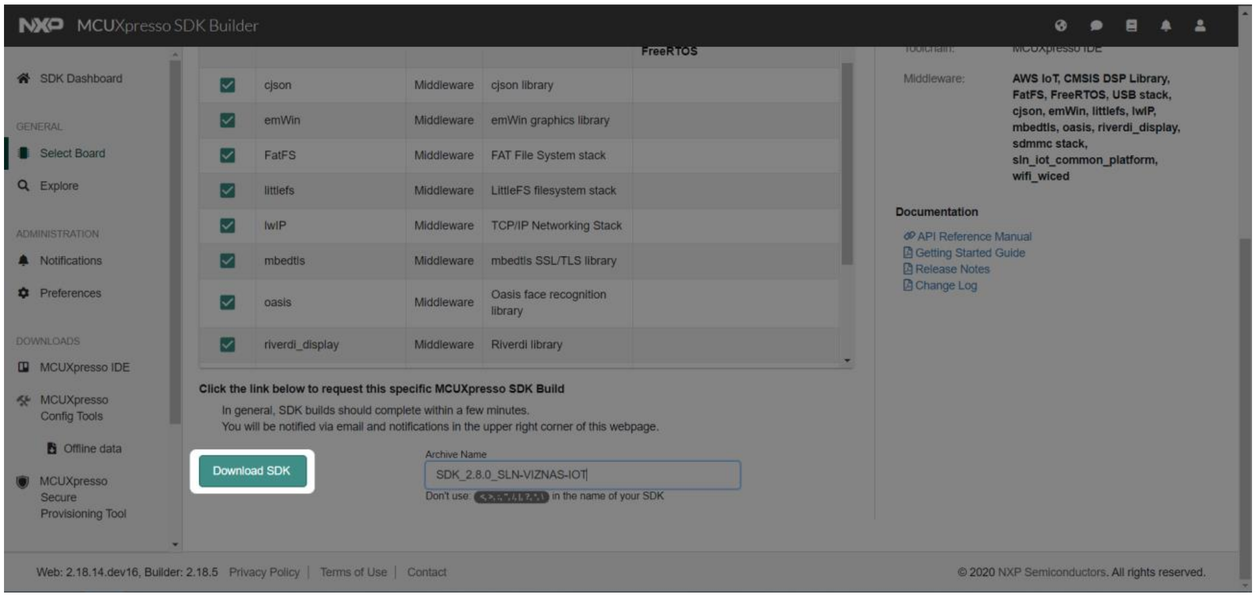
SDK Details

SDK Version:	2.8.0 (released 2020-11-04)
SDK Tag:	REL_2.8.0_SLN_VIZNAS_IOT_RFP_RC9
Host OS:	Windows
Toolchain:	MCUXpresso IDE
Middleware:	AWS IoT, CMSIS DSP Library, FatFS, FreeRTOS, USB stack, cjson, emWin, littlefs, lwIP, mbedtls, oasis, riverdi_display, sdmmc stack, sln_iot_common_platform, wifi_wiced

Web: 2.18.14.dev16, Builder: 2.18.5 Privacy Policy | Terms of Use | Contact © 2020 NXP Semiconductors. All rights reserved.

Figure 11: Select all SDK components

With these options selected, press the **Download SDK** button at the bottom of the page.



NXP MCUXpresso SDK Builder

	Name	Category	Description	Dependencies
✓	cjson	Middleware	cjson library	
✓	emWin	Middleware	emWin graphics library	
✓	FatFS	Middleware	FAT File System stack	
✓	littlefs	Middleware	LittleFS filesystem stack	
✓	lwIP	Middleware	TCP/IP Networking Stack	
✓	mbedtls	Middleware	mbedtls SSL/TLS library	
✓	oasis	Middleware	Oasis face recognition library	
✓	riverdi_display	Middleware	Riverdi library	

Click the link below to request this specific MCUXpresso SDK Build

In general, SDK builds should complete within a few minutes.
You will be notified via email and notifications in the upper right corner of this webpage.

Download SDK

Archive Name: SDK_2.8.0_SLN-VIZNAS-IOT
Don't use: [illegible] in the name of your SDK

Middleware: AWS IoT, CMSIS DSP Library, FatFS, FreeRTOS, USB stack, cjson, emWin, littlefs, lwIP, mbedtls, oasis, riverdi_display, sdmmc stack, sln_iot_common_platform, wifi_wiced

Documentation
API Reference Manual
Getting Started Guide
Release Notes
Change Log

Web: 2.18.14.dev16, Builder: 2.18.5 Privacy Policy | Terms of Use | Contact © 2020 NXP Semiconductors. All rights reserved.

Figure 12: Download SDK

Once the SDK is downloaded, import the SDK into MCUXpresso IDE by dragging the SDK zip folder into the **Installed SDKs** window in MCUXpresso IDE.

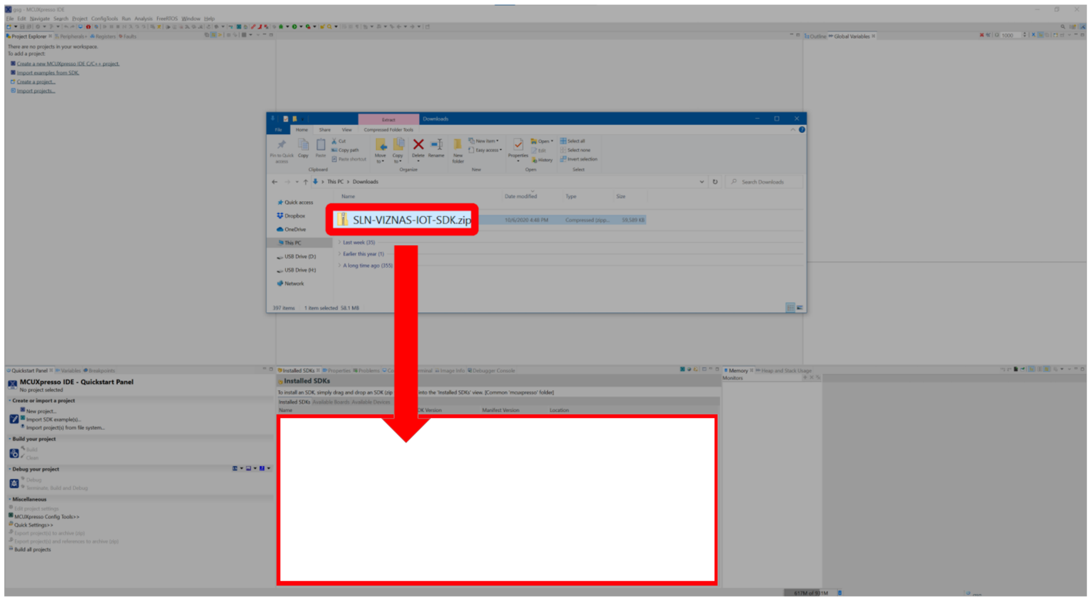
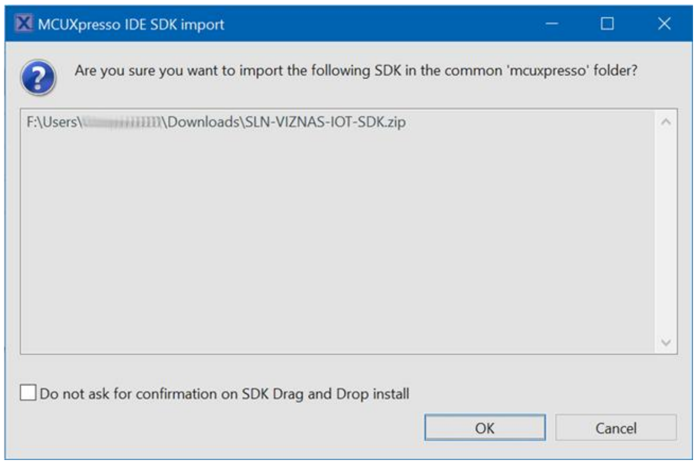


Figure 13: Drag and Drop SDK

For each package, a confirmation window will pop-up. Select **OK** to validate.



MCUXpresso IDE SDK import

Are you sure you want to import the following SDK in the common 'mcuxpresso' folder?

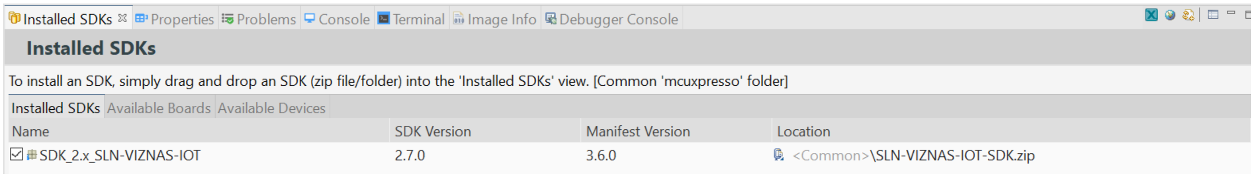
F:\Users\[redacted]\Downloads\SLN-VIZNAS-IOT-SDK.zip

☐ Do not ask for confirmation on SDK Drag and Drop install

OK Cancel

Figure 14: Import SDK Confirmation Window

Once the package has been imported, it will be displayed in the **Installed SDKs** window in MCUXpresso.



Installed SDKs

To install an SDK, simply drag and drop an SDK (zip file/folder) into the 'Installed SDKs' view. [Common 'mcuxpresso' folder]

Name	SDK Version	Manifest Version	Location
☑ SDK_2.x_SLN-VIZNAS-IOT	2.7.0	3.6.0	<Common>\SLN-VIZNAS-IOT-SDK.zip

Figure 15: SDK Import Successful

3.3 Import a SLN-VIZNAS-IOT Project

The SLN-VIZNAS-IOT SDK allows you to import existing application examples as a development starting point. The following steps will show you how to import one of these example projects into MCUXpresso IDE.

From the **Quickstart Panel**, select **Import SDK example(s).**

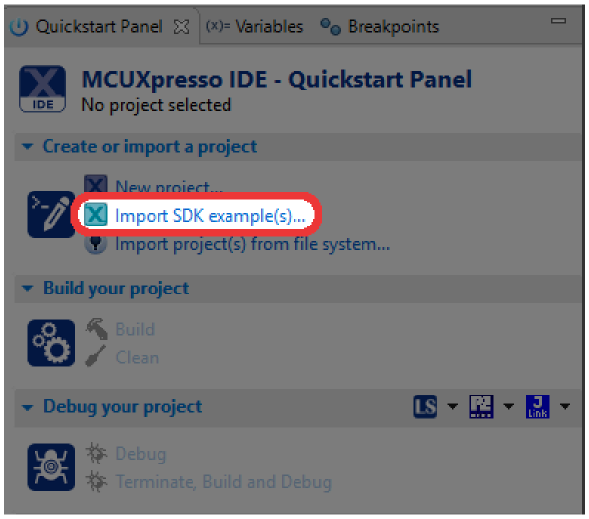
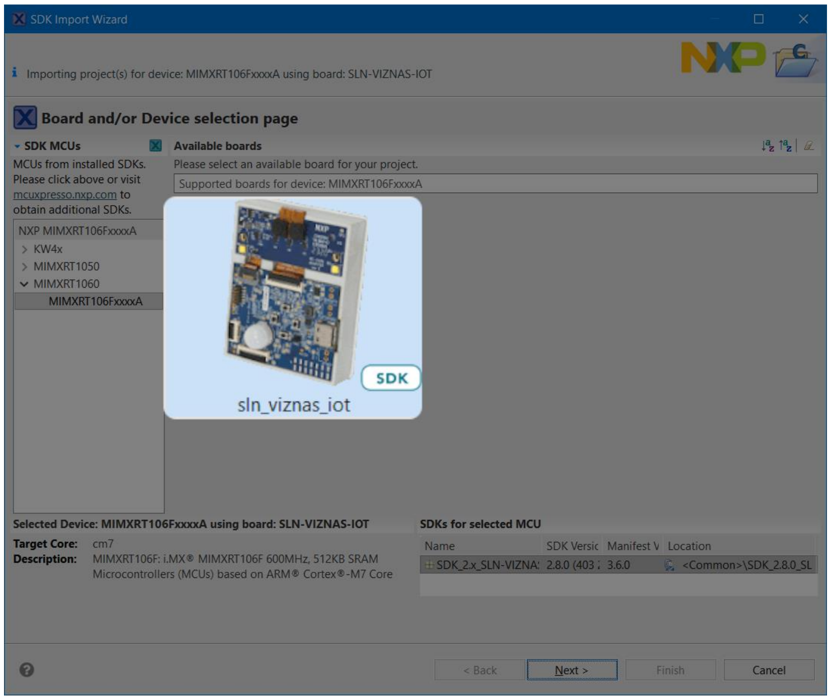


Figure 16: Import SDK Examples

For each SDK you have installed into MCUXpresso, a corresponding image will be shown. Select the **SLN-VIZNAS-IOT** image and proceed by selecting the **Next** button.



Importing project(s) for device: MIMXRT106FxxxxA using board: SLN-VIZNAS-IOT

MCUs from installed SDKs. Please click above or visit mcuxpresso.nxp.com to obtain additional SDKs.

Please select an available board for your project.

Supported boards for device: MIMXRT106FxxxxA

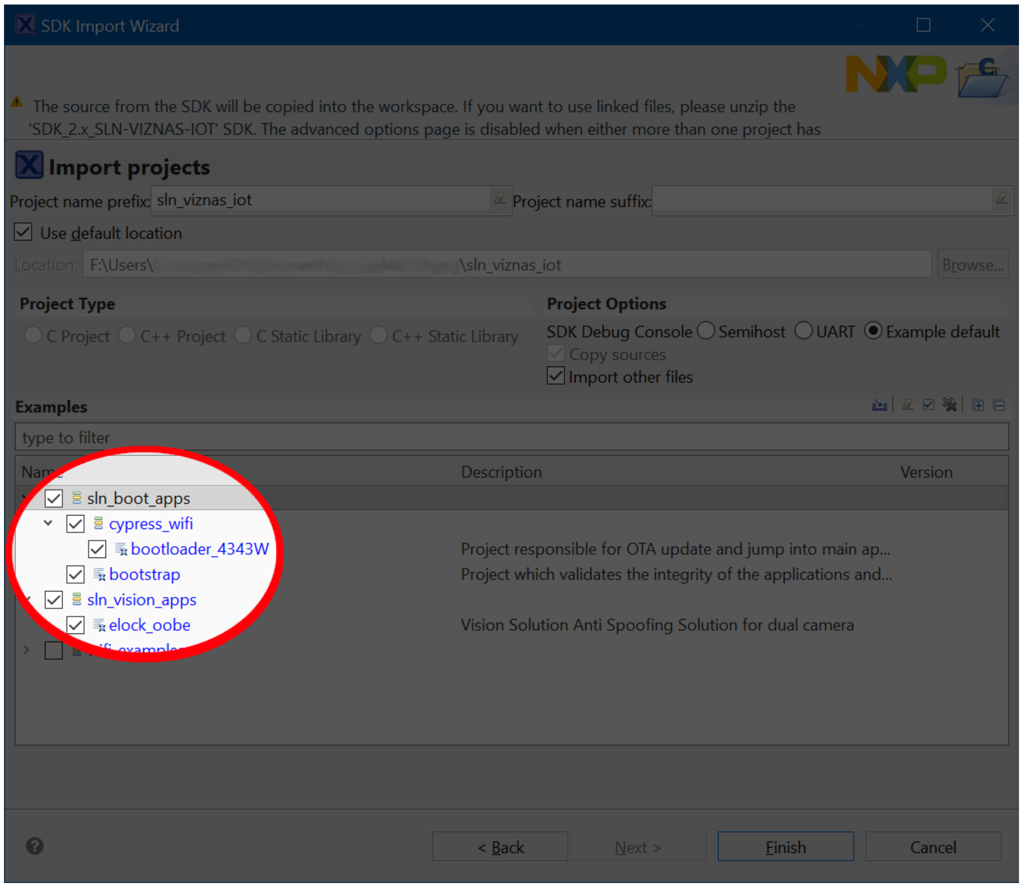
Selected Device: MIMXRT106FxxxxA using board: SLN-VIZNAS-IOT

Target Core: cm7
Description: MIMXRT106F: i.MX® MIMXRT106F 600MHz, 512KB SRAM Microcontrollers (MCUs) based on ARM® Cortex®-M7 Core

Name	SDK Versic	Manifest V	Location
SDK_2.x_SLN-VIZNA:	2.8.0 (403 ;	3.6.0	<Common>\SDK_2.8.0_SL

Figure 17: Import SLN-VIZNAS-IOT Examples Wizard

The import wizard will then display all the example applications that are available to import. For this guide we will be focused primarily on the **sln_viznas_iot_elock_oobe** application. This is the application that comes flashed by default on your **SLN-VIZNAS-IOT** kit.


SDK Import Wizard

The source from the SDK will be copied into the workspace. If you want to use linked files, please unzip the 'SDK_2.x_SLN-VIZNAS-IOT' SDK. The advanced options page is disabled when either more than one project has

Import projects

Project name prefix: sln_viznas_iot Project name suffix:

☑ Use default location

Location: F:\Users\[redacted]\sln_viznas_iot Browse...

Project Type

○ C Project ○ C++ Project ○ C Static Library ○ C++ Static Library

Project Options

SDK Debug Console ○ Semihost ○ UART ● Example default

☑ Copy sources

☑ Import other files

Examples

type to filter

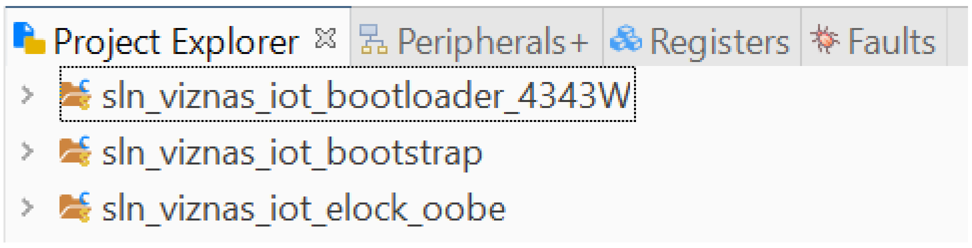
Name	Description	Version
☑ sln_boot_apps		
☑ cypress_wifi		
☑ bootloader_4343W	Project responsible for OTA update and jump into main ap...	
☑ bootstrap	Project which validates the integrity of the applications and...	
☑ sln_vision_apps		
☑ elock_oobe	Vision Solution Anti Spoofing Solution for dual camera	
☐ wifi_examples		

< Back Next > Finish Cancel

Figure 18: Import Examples

NOTE: If your kit's flash has been completely erased, you will also need the "sln_viznas_iot_bootloader_4343W" and "sln_viznas_iot_bootstrap" projects found under sln_boot_apps as well in order for the sln_viznas_iot_elock_oobe application to work.

Once the projects have successfully been imported, they will be listed in the project explorer ready to be built and run.


Project Explorer | Peripherals+ | Registers | Faults

- sln_viznas_iot_bootloader_4343W
- sln_viznas_iot_bootstrap
- sln_viznas_iot_elock_oobe

Figure 19: Projects Successfully Imported

3.4 (Optional) Increase Clock Drive Strength

By default, the sln_viznas_iot_elock_oobe application uses a lower clock rate and clock drive strength for the cameras than the 106F is capable of in order to maintain EMC compliance (see **RT106F VISION CROSSOVER PROCESSOR OVERVIEW**). This lower clock rate results in a lower FPS and overall recognition performance.

To make it easy for developers to utilize the full performance capabilities of the camera, the sln_viznas_iot_elock_oobe project provides a convenient preprocessor macro which can be configured to increase or decrease the camera clock's drive strength and clock rate. To configure this preprocessor macro, right click on the sln_viznas_iot_elock_oobe project in the **Project Explorer** pane as shown in the figure below and select **Properties**, found near the bottom of the context menu:

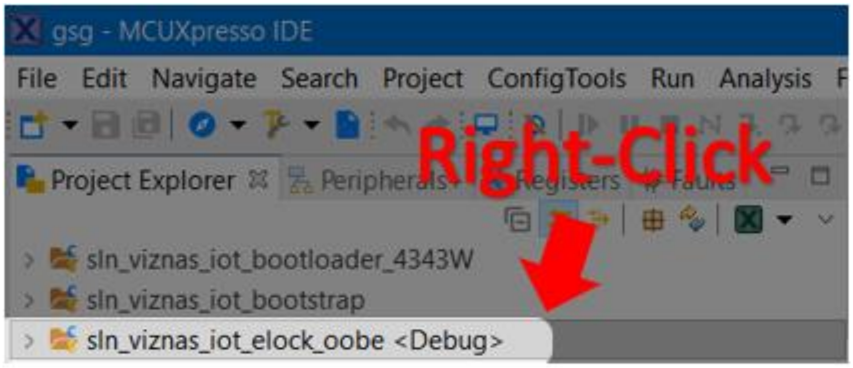

gsg - MCUXpresso IDE
File Edit Navigate Search Project ConfigTools Run Analysis
Project Explorer
Peripherals
Registers
Faults
Right-Click
sln_viznas_iot_bootloader_4343W
sln_viznas_iot_bootstrap
sln_viznas_iot_elock_oobe <Debug>

Figure 20: Right-Click Project

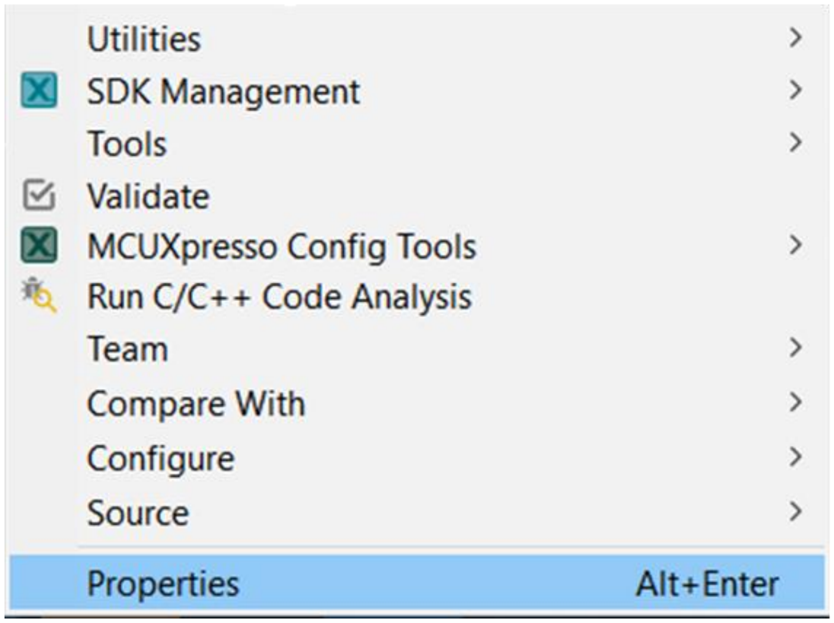


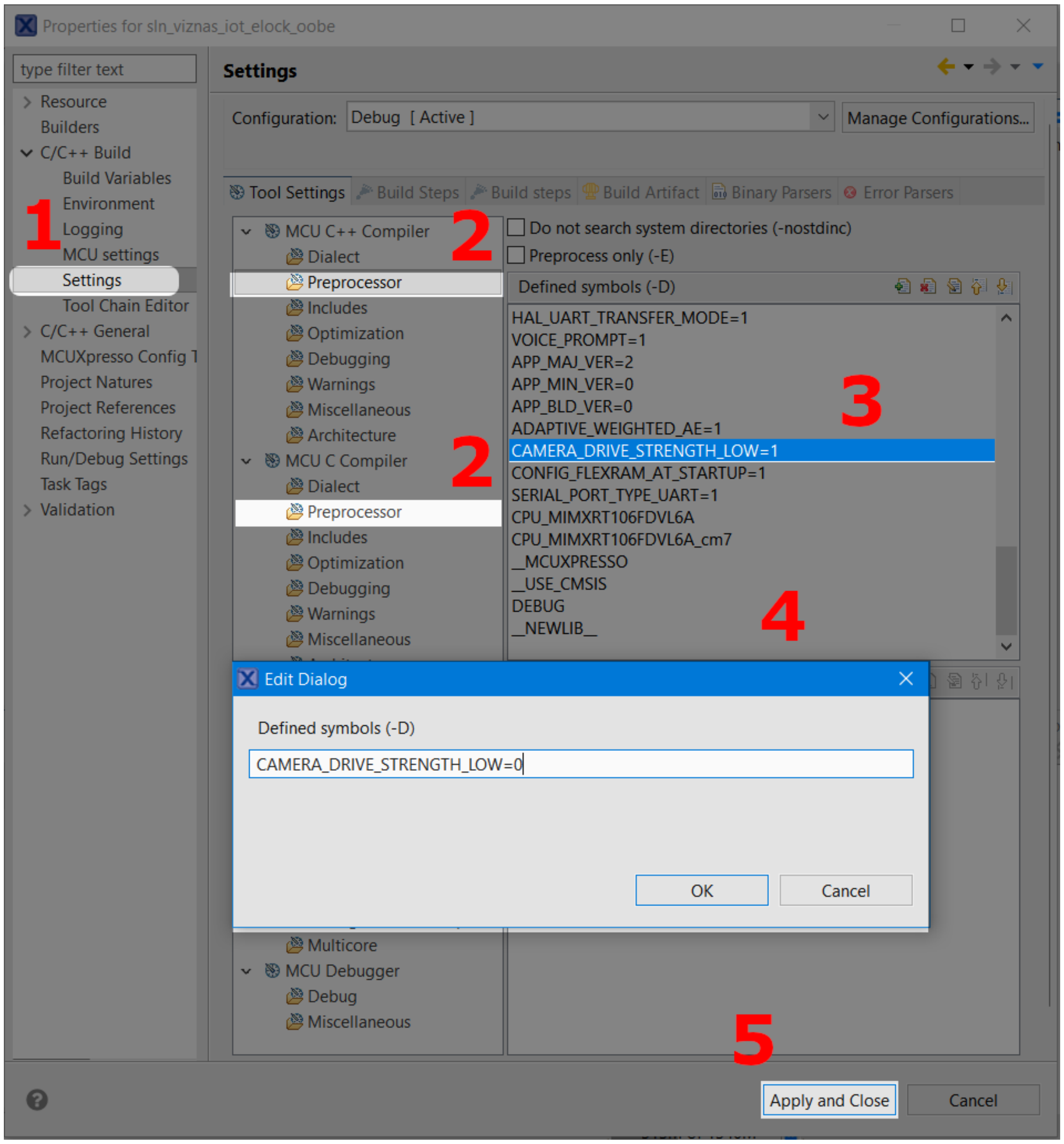
Figure 21: "Properties" Option

Clicking the **Properties** option will open up a new window which looks like the one shown in **Figure 22: Properties Window**. From there:

1. Click **Settings**
2. Click **MCU C++ Compiler -> Preprocessor**
3. Double click the **CAMERA_DRIVE_STRENGTH_LOW** macro
4. Change the value from 1 to 0 and click **OK.**

Repeat the previous steps for the **CAMERA_DRIVE_STRENGTH_LOW** preprocessor macro found under **MCU C COMPILER -> Preprocessor,** then003A

5. Click **Apply and Close**.



HAL_UART_TRANSFER_MODE=1
VOICE_PROMPT=1
APP_MAJ_VER=2
APP_MIN_VER=0
APP_BLD_VER=0
ADAPTIVE_WEIGHTED_AE=1
CAMERA_DRIVE_STRENGTH_LOW=1
CONFIG_FLEXRAM_AT_STARTUP=1
SERIAL_PORT_TYPE_UART=1
CPU_MIMXRT106FDVL6A
CPU_MIMXRT106FDVL6A_cm7
__MCUXPRESSO
__USE_CMSIS
DEBUG
__NEWLIB__

CAMERA_DRIVE_STRENGTH_LOW=0

Figure 22: Properties Window

4 Building and Programming

The **bootstrap** project is the first application that is booted. The bootstrap is a minimal FreeRTOS application that is responsible for image verification.

The **bootloader** project is a second stage bootloader that manages jumping into the **E-Lock OoBE** application. This application can be used for any additional bootloader functionality needed for the product. The bootloader is also responsible for Mass Storage Device drag-and-drop firmware updates via USB.

The **E-Lock OoBE** is the out-of-box application used to demonstrate the capabilities of the Oasis Lite machine learning engine for secure face recognition. This is the application (in addition to the bootloader and bootstrap) that is flashed on your SLN-VIZNAS-IOT kit by default.

4.1 Build a SLN-VIZNAS-IOT Project

In the **Project Explorer** window**,** select the project you intend to compile.

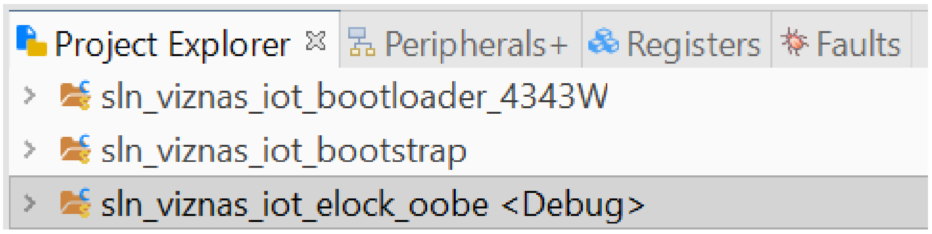


Figure 23: Project Explorer - Highlight Project

From the **Quickstart Panel**, select the option **Build** to start the compilation and linking of the application currently highlighted in the **Project Explorer** pane.

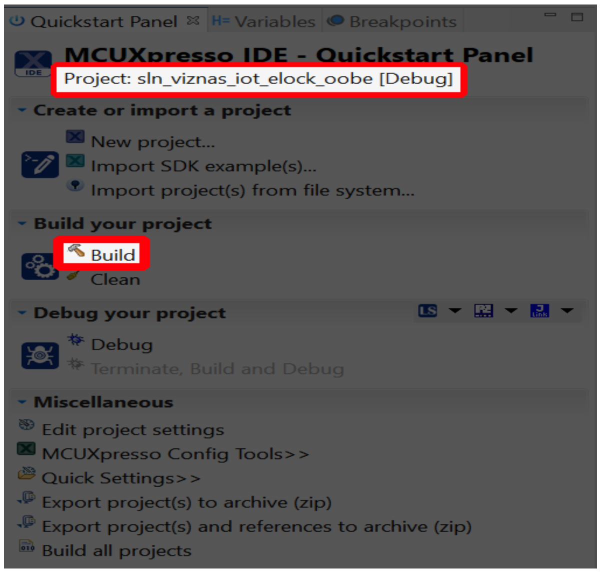
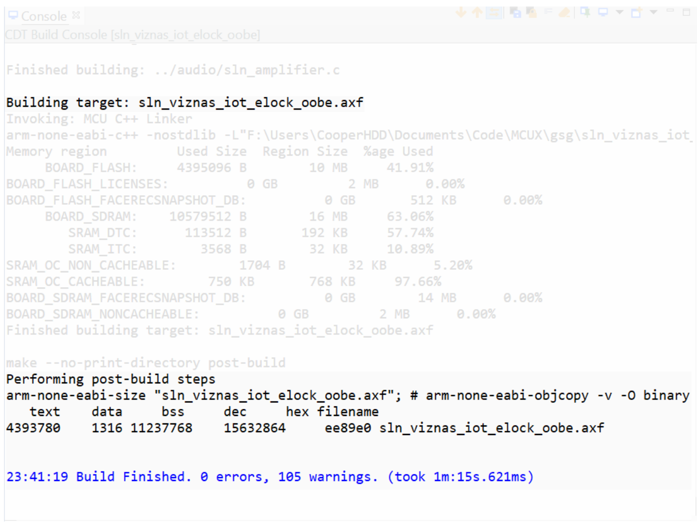

Quickstart Panel
Variables
Breakpoints
MCUXpresso IDE - Quickstart Panel
Project: sln_viznas_iot_elock_oobe [Debug]
Create or import a project
New project...
Import SDK example(s)...
Import project(s) from file system...
Build your project
Build
Clean
Debug your project
Debug
Terminate, Build and Debug
Miscellaneous
Edit project settings
MCUXpresso Config Tools>>
Quick Settings>>
Export project(s) to archive (zip)
Export project(s) and references to archive (zip)
Build all projects

Figure 24: Build Project

Wait for MCUXpresso to finish the build process.



```
Console
CDT Build Console [sln_viznas_iot_elock_oobe]

Finished building: ../audio/sln_amplifier.c

Building target: sln_viznas_iot_elock_oobe.axf
Invoking: MCU C++ Linker
arm-none-eabi-c++ -nostdlib -L"F:\Users\CooperHDD\Documents\Code\MCUX\gsg\sln_viznas_iot_
Memory region         Used Size  Region Size  %age Used
     BOARD_FLASH:     4395096 B        10 MB     41.91%
BOARD_FLASH_LICENSES:          0 GB         2 MB      0.00%
BOARD_FLASH_FACERECSNAPSHOT_DB:          0 GB       512 KB      0.00%
     BOARD_SDRAM:    10579512 B        16 MB     63.06%
        SRAM_DTC:      113512 B       192 KB     57.74%
        SRAM_ITC:        3568 B        32 KB     10.89%
SRAM_OC_NON_CACHEABLE:        1704 B        32 KB      5.20%
SRAM_OC_CACHEABLE:        750 KB       768 KB     97.66%
BOARD_SDRAM_FACERECSNAPSHOT_DB:          0 GB        14 MB      0.00%
BOARD_SDRAM_NONCACHEABLE:          0 GB         2 MB      0.00%
Finished building target: sln_viznas_iot_elock_oobe.axf

make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "sln_viznas_iot_elock_oobe.axf"; # arm-none-eabi-objcopy -v -O binary
   text    data     bss     dec     hex filename
4393780    1316 11237768    15632864     ee89e0 sln_viznas_iot_elock_oobe.axf


23:41:19 Build Finished. 0 errors, 105 warnings. (took 1m:15s.621ms)
```

Figure 25: Console 'Build' Output

If you received a message like the one shown above, your elock_oobe project has been successfully built.

Additionally, if you have use for a **binary** file instead of the **.axf** generated by default, simply right-click on the **.axf** you wish to convert and go to **Binary Utilities -> Create binary.** MCUXpresso stores generated **.axf** and **.bin** files under your project's **Debug** folder. Shown below is an example of how you can create a **binary** using a **.axf** file:

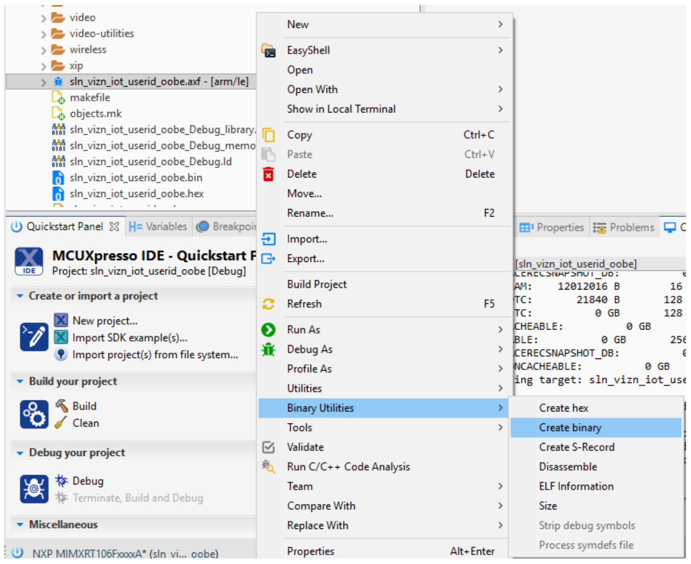


Figure 26: .axf to .bin

.bin files are useful for flashing with **OTW/OTA**, **MSD**, and the automated manufacturing tools. Each of these features are described in greater detail later in the guide.

4.2 Flash and Debug SLN-VIZNAS-IOT Project

With the **elock_oobe** project compiled, it is now time to program its associated binary into flash.

Flashing the SLN-VIZNAS-IOT kit will require a **Segger J-Link** with a **9-pin Cortex-M Adapter** and **V6.62a** or newer of the **J-Link Software and Documentation Pack** found on the Segger website at:

https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack.

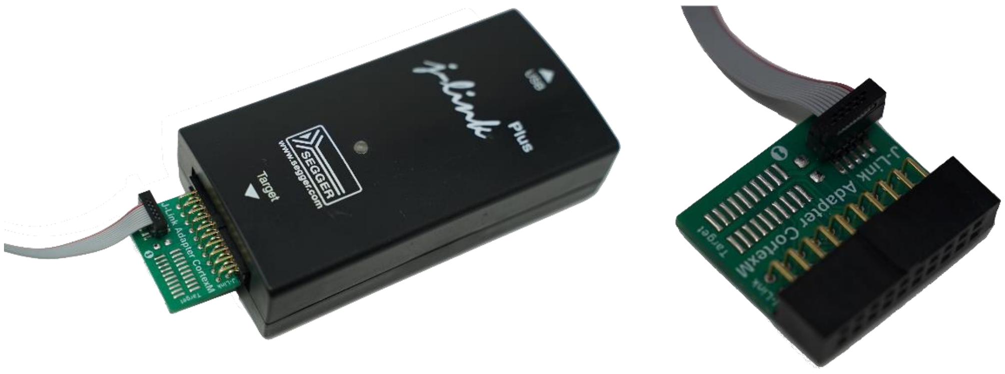


Figure 27: J-Link Plus and 9-Pin Cortex-M Adapter

NOTE: The MCUXpresso IDE installation process also installs the Segger J-Link Software and Documentation pack. MCUXpresso IDE version 11.2 or greater comes with a sufficient version of the Segger J-Link Software and Documentation Pack.

Older versions of J-Link Software and Documentation Pack will not have the proper configuration settings for the SLN-VIZNAS-IOT and will therefore be unable to flash the board.

To begin the process of flashing the kit, attach your **J-Link** debug probe into the header shown below.

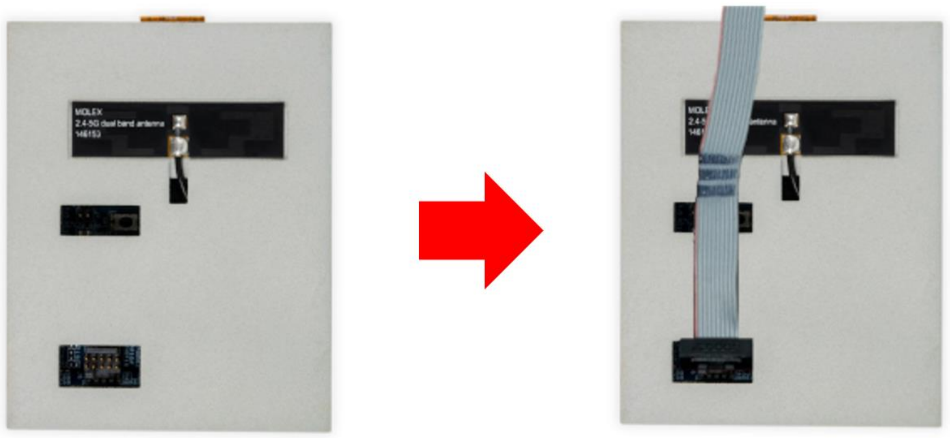


Figure 28: SLN-VIZNAS-IOT JTAG Header

Next, select the **Debug** option found under the **QuickStart** panel in MCUXpresso to start the process of loading the binary into the flash and begin debugging. Like the **Build** option, **Debug** will only flash and debug the project currently highlighted in the **Project Explorer** panel.

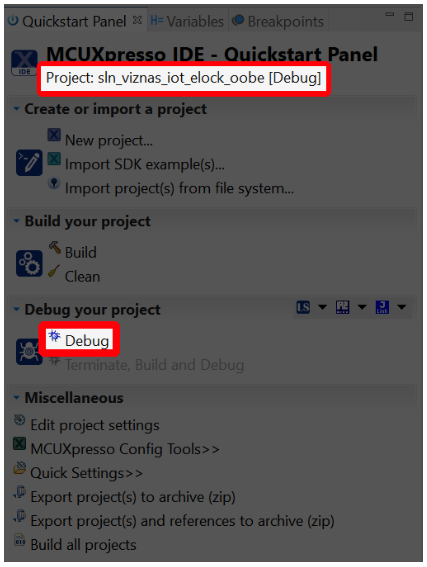
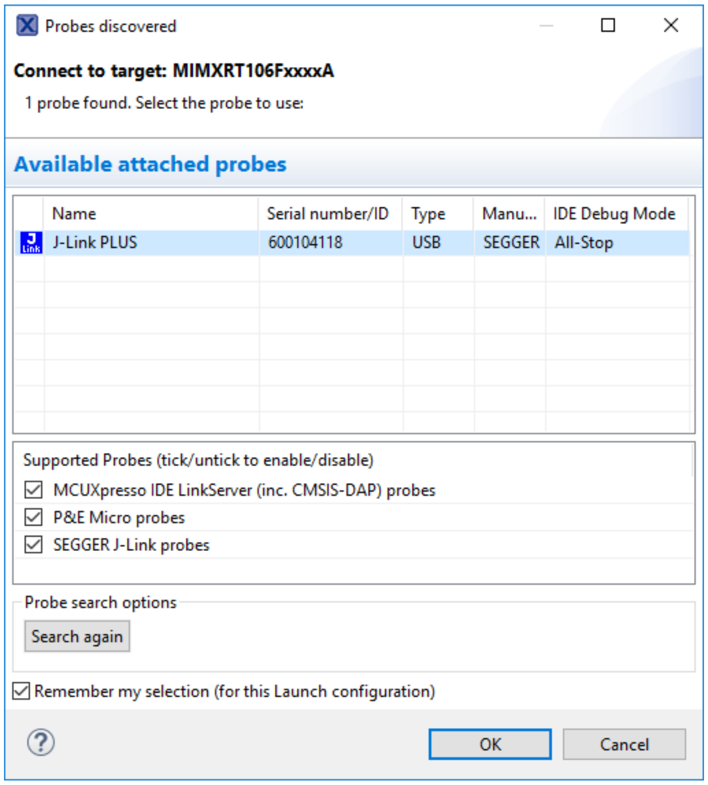

Quickstart Panel
Variables
Breakpoints
MCUXpresso IDE - Quickstart Panel
Project: sln_viznas_iot_elock_oobe [Debug]
Create or import a project
New project...
Import SDK example(s)...
Import project(s) from file system...
Build your project
Build
Clean
Debug your project
Debug
Terminate, Build and Debug
Miscellaneous
Edit project settings
MCUXpresso Config Tools>>
Quick Settings>>
Export project(s) to archive (zip)
Export project(s) and references to archive (zip)
Build all projects

Figure 29: Quickstart Panel - Debug

Select the J-Link probe that is connected to your kit and press OK.



Connect to target: MIMXRT106FxxxxA

1 probe found. Select the probe to use:

Available attached probes

Name	Serial number/ID	Type	Manu...	IDE Debug Mode
J-Link PLUS	600104118	USB	SEGGER	All-Stop

Supported Probes (tick/untick to enable/disable)

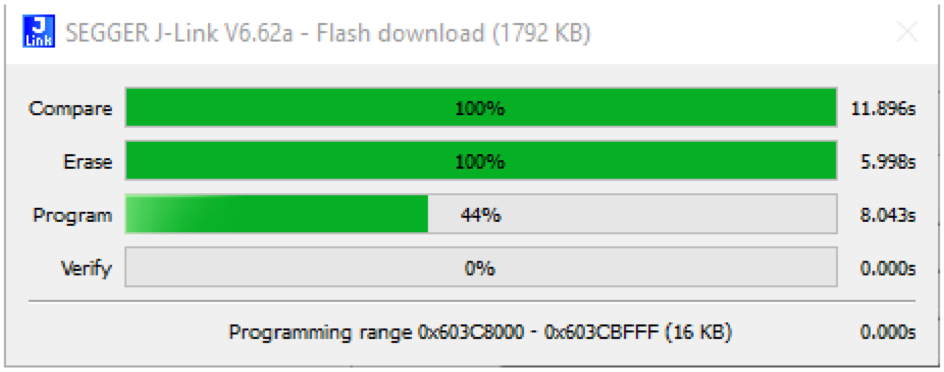
- [✓] MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes
- [✓] P&E Micro probes
- [✓] SEGGER J-Link probes

Probe search options

[✓] Remember my selection (for this Launch configuration)

Figure 30: Probe Discovery Window

This will launch the flashing tool and proceed to flash the binary associated with the currently selected project.

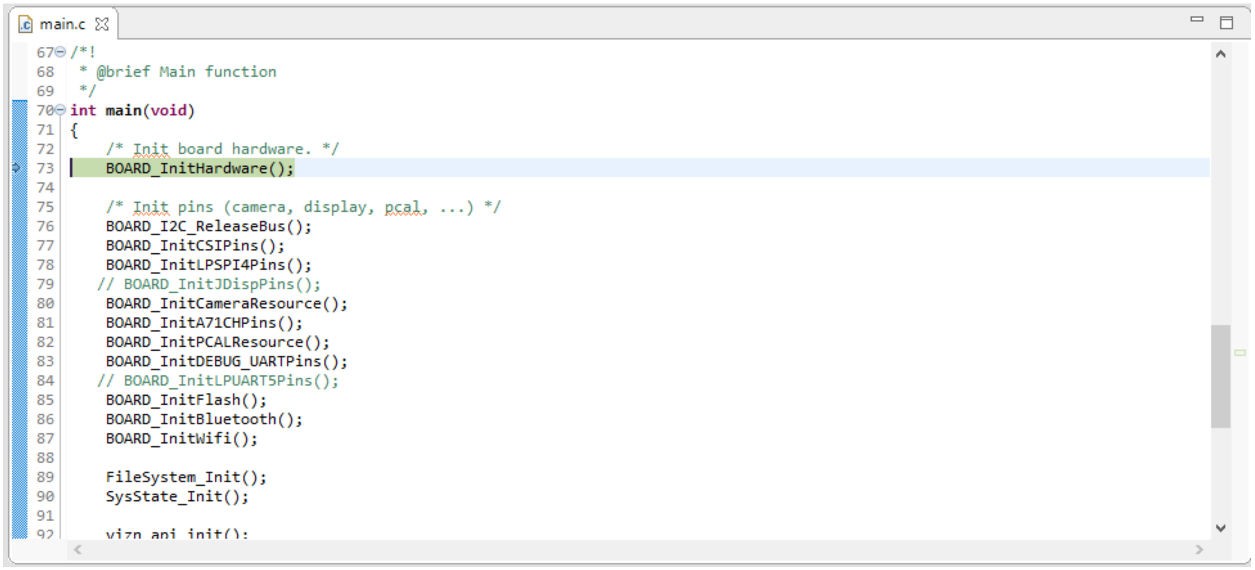


SEGGER J-Link V6.62a - Flash download (1792 KB)

```
Compare  100%   11.896s
Erase    100%    5.998s
Program   44%    8.043s
Verify     0%    0.000s

Programming range 0x603C8000 - 0x603CBFFF (16 KB)   0.000s
```

Figure 31: Flash Download in Progress

Once flashed, the program will automatically halt at main, indicated by the first instruction in main being highlighted and pointed to.



main.c

```
/*!
 * @brief Main function
 */
int main(void)
{
    /* Init board hardware. */
    BOARD_InitHardware();

    /* Init pins (camera, display, pcal, ...) */
    BOARD_I2C_ReleaseBus();
    BOARD_InitCSIPins();
    BOARD_InitLPSPI4Pins();
//  BOARD_InitJDispPins();
    BOARD_InitCameraResource();
    BOARD_InitA71CHPins();
    BOARD_InitPCALResource();
    BOARD_InitDEBUG_UARTPins();
//  BOARD_InitLPUART5Pins();
    BOARD_InitFlash();
    BOARD_InitBluetooth();
    BOARD_InitWifi();

    FileSystem_Init();
    SysState_Init();

    vizn_api_init();
```

Figure 32: Debug Begin

Finally, press the **Run** button found in the toolbar to begin running the application.

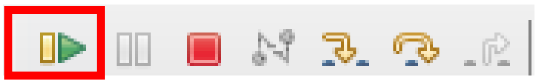


Figure 33: Debug Toolbar - Run Button

To learn more about debugging in MCUXpresso, check out the **MCUXpresso IDE User Guide.**

5 Bootloader

The SDK provided enables two forms of firmware update capability in addition to flashing via J-Link: An **Over-the-Air/Over-the-Wire** (**OTA/OTW**) interface that supports Wi-Fi/UART-based firmware flashing, and a **USB Mass Storage Device (MSD**) interface. Either option can be selected at boot time, but once one of them is running, the other is turned off.

5.1 Application Flow

The boot flow is described in detail in the **Security Architecture** section of this document. Once the boot flow reaches the bootloader, the bootloader must decide what to do. The below shows the three options available to the bootloader. The subsequent sections describe the **OTA/OTW** and **USB MSD** modes.

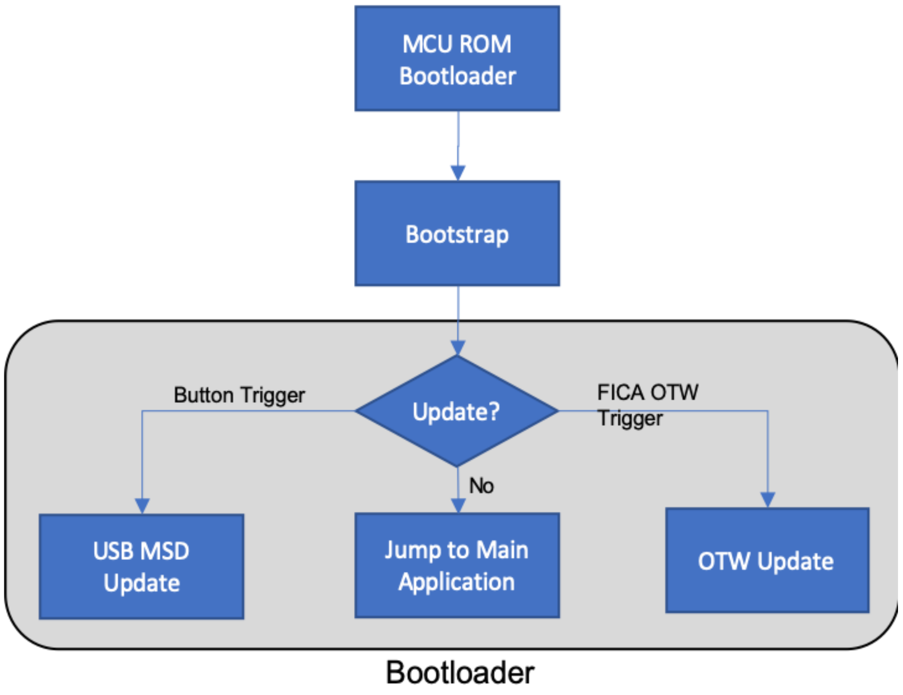


Figure 34: Bootloader Flow

5.2 Over-the-Air (OTA)/Over-the-Wire (OTW) Updates

The bootloader supports several mechanisms for updating the board's firmware, one of which is OTA/OTW updates. OTA and OTW updates are driven using a simple JSON interface, making it easy to implement host-side code. The mechanism for both OTA and OTW is the same, the only difference between the two being the fact that OTA is performed "Over the Air" via Wi-Fi, BLE, etc. and OTW is performed over UART, SPI, I2C, etc. The OTA update interface is performed by hosting a TCP server on the kit which receives the update-related JSON packets. OTW on the other hand, currently supports UART but can be extended to support any serial interface including SPI, TCP sockets, or even I2C. This OTA/OTW update method will be discussed in more depth in this section.

5.2.1 Transfers

An OTA/OTW update is made up of individual JSON transfers. Each transfer contains two pieces: a 4-byte size field and a JSON message. This allows the OTA/OTW data interface to be compatible across a wide range of interfaces.

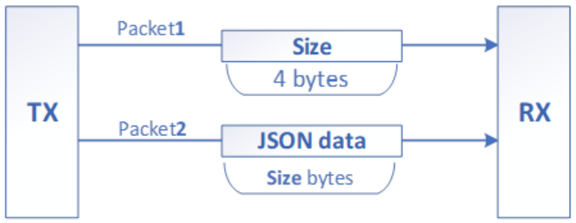


Figure 35: Transfer Format

Each transfer is followed by a transfer response.

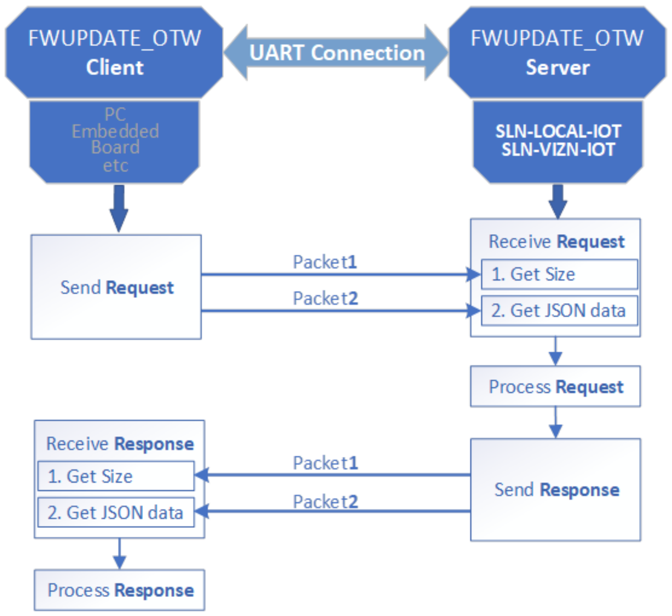


Figure 36: Request/Response Flow

5.2.2 Testing OTA/OTW Updates

In order to help demonstrate OTA and OTW updates, a test Python script can be found in **sln_viznas_iot_bootloader_4343W/unit_tests/fwupdate_client.py**. From a high level, this script will "JSONify" a specified binary and flash it either via the OTA or OTW mechanism, depending on the update method specified as an argument to the script. This method allows users to flash the main app binary without the need for a J-Link.

While the method for using OTA and OTW is nearly identical, the setup is slightly different between the two because OTA requires connection to a network and OTW requires a serial connection.

5.2.2.1 OTA Setup Procedure

This section will describe the setup steps necessary to perform an OTA update. To perform an OTA update using the test script, the SLN-VIZNAS-IOT will need to be connected to a Wi-Fi network and the proper bit in the FICA table will need to be set to indicate to the bootloader that an OTA update is being expected. The update bit is set by the test script, but the network connection must be established prior to running the script.

Connect to a Wi-Fi network:

a) Use the **"wifi credentials SSID PASSWORD"** command, specifying the credentials to the network which you would like to connect to.
b) With the credentials specified, reset the Wi-Fi driver using the **"wifi reset"** command so that the board will attempt to establish a connection.
c) Use the Wi-Fi IP command to ensure the board has successfully connected to the network.
d) If the IP address returned is "0.0.0.0", wait a few seconds and try running the command again until a valid IP address is returned.
e) If still encountering issues, verify the credentials specified in step 1 and power cycle the board.

5.2.2.2 OTW Setup Procedure

This section will describe the steps necessary to perform an OTW update. To perform an OTW update using the test script, a serial connection using an USB->UART connector is required, and the proper bit in the FICA table will need to be set to indicate to the bootloader that an OTW update is being expected. The update bit is set by the test script, but the serial connection must be established prior to running the script.

Connect to UART via USB to UART connector:

a) Plug in and power on board ***WITHOUT*** any connections made to the serial header.
b) Attach pins from the USB to UART connector to the serial header pins shown in the figure below:

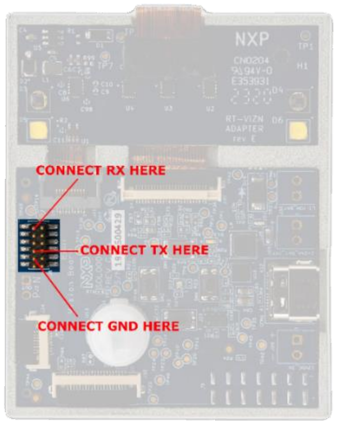
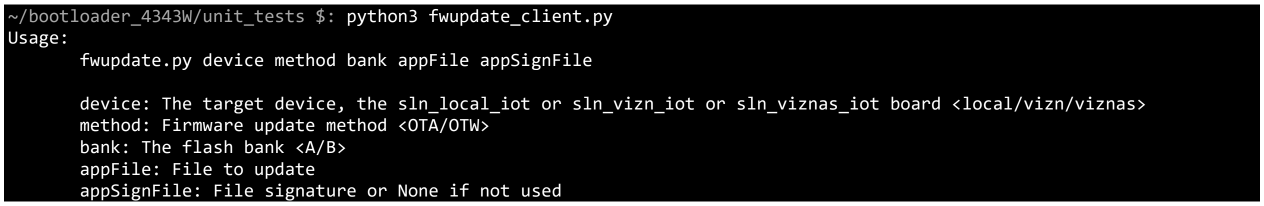


Figure 37: Serial Header Connections

NOTE: USB to UART pins MUST be connected AFTER the board has been powered on due to the serial header pins functioning as boot select mode pins during board startup.

5.2.2.3 Run fwupdate_client.py

With the prerequisite steps for OTA/OTW updates completed, the kit is ready to receive an update. To begin the update, open the **fwupdate_client.py** script found under **sln_viznas_iot_bootloader_4343W/scripts/unit_tests** in a terminal. Running the script without any arguments will show the arguments that the script takes.



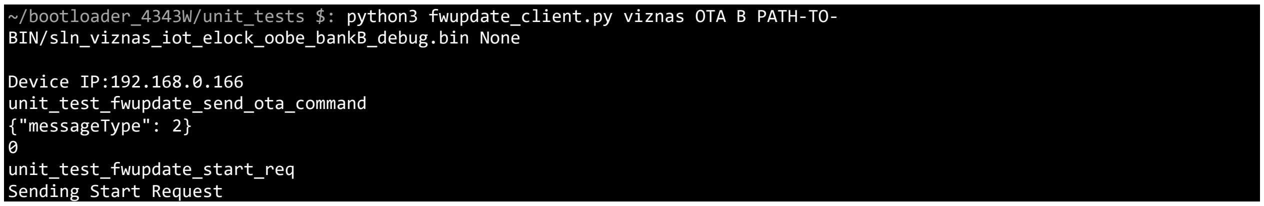
```
~/bootloader_4343W/unit_tests $: python3 fwupdate_client.py
Usage:
        fwupdate.py device method bank appFile appSignFile

        device: The target device, the sln_local_iot or sln_vizn_iot or sln_viznas_iot board <local/vizn/viznas>
        method: Firmware update method <OTA/OTW>
        bank: The flash bank <A/B>
        appFile: File to update
        appSignFile: File signature or None if not used
```

Figure 38: "fwupdate_client.py" w/o Args

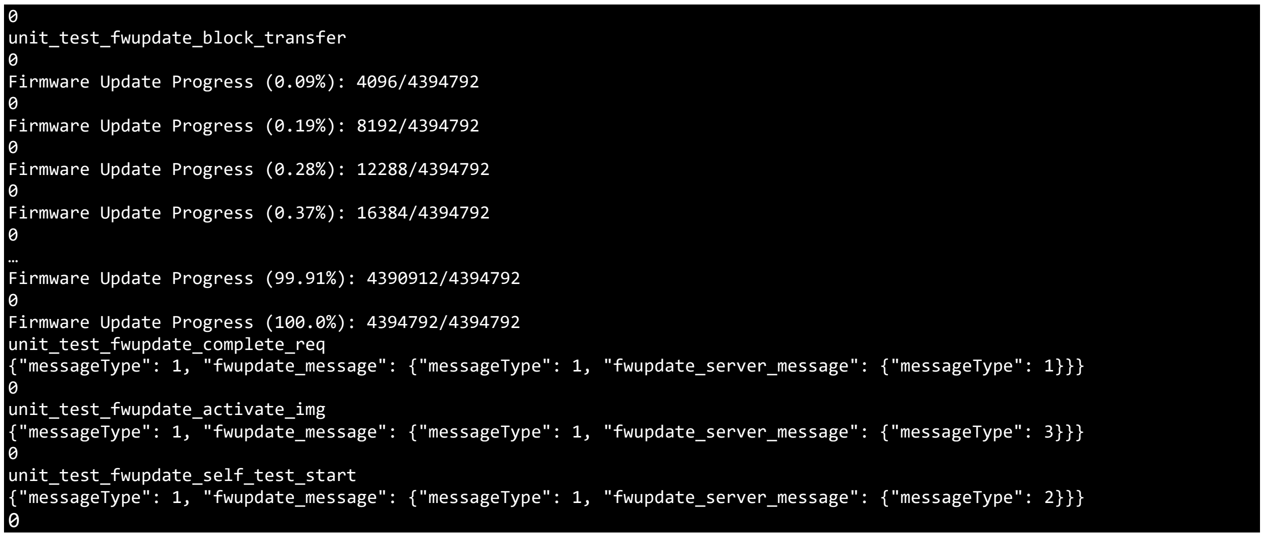
As shown in the figure above, the script requires that you specify:

a) Device: viznas
b) Method: OTA/OTW
c) Bank: A/B (see **Generate Bank B Binary**)
d) Appfile: Binary for the sln_viznas_iot_elock_oobe project (see **Build a SLN-VIZNAS-IOT Project**)
e) appsignFile: None/sln_viznas_iot_elock_oobe.sha256.txt (see **Using Ivaldi to Generate Signing Artifacts**)

Once all args are specified, the script will produce output like the following:



```
~/bootloader_4343W/unit_tests $: python3 fwupdate_client.py viznas OTA B PATH-TO-
BIN/sln_viznas_iot_elock_oobe_bankB_debug.bin None

Device IP:192.168.0.166
unit_test_fwupdate_send_ota_command
{"messageType": 2}
0
unit_test_fwupdate_start_req
Sending Start Request
```



```
0
unit_test_fwupdate_block_transfer
0
Firmware Update Progress (0.09%): 4096/4394792
0
Firmware Update Progress (0.19%): 8192/4394792
0
Firmware Update Progress (0.28%): 12288/4394792
0
Firmware Update Progress (0.37%): 16384/4394792
0
…
Firmware Update Progress (99.91%): 4390912/4394792
0
Firmware Update Progress (100.0%): 4394792/4394792
unit_test_fwupdate_complete_req
{"messageType": 1, "fwupdate_message": {"messageType": 1, "fwupdate_server_message": {"messageType": 1}}}
0
unit_test_fwupdate_activate_img
{"messageType": 1, "fwupdate_message": {"messageType": 1, "fwupdate_server_message": {"messageType": 3}}}
0
unit_test_fwupdate_self_test_start
{"messageType": 1, "fwupdate_message": {"messageType": 1, "fwupdate_server_message": {"messageType": 2}}}
0
```

Figure 39: "fwupdate_client.py" w/ Args

Upon completion, the SLN-VIZNAS-IOT will automatically restart itself and switch over to the new application bank, running the new application that was just flashed.

5.3 Mass Storage Device (MSD) Update

The bootloader application supports firmware update over **USB Mass Storage Device (MSD).** This allows the user to re-flash the main application binary (note, not the bootstrap and bootloader) without a J-Link probe. If the bootstrap and bootloader need to be updated, you must use the J-Link probe.

The MSD feature by default bypasses the signature verification described in **Security Architecture** to allow an easier development flow because signing images can be a process not suitable for quick debugging and validation.

To enable **MSD** mode, hold **SW1** while the board is powering on.

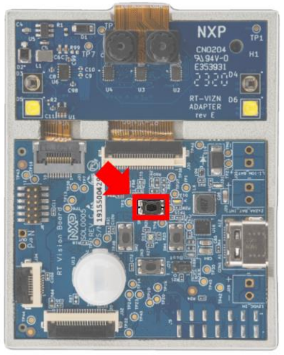


Figure 40: MSD Enablement Button

Upon success, the normal boot sequence with alternating red, blue and purple LEDs will take place, followed by the blinking of a purple LED on the front of the board every 2 seconds indicating the board is now in **MSD** mode. Once the LED has started blinking, you may let go of the button.

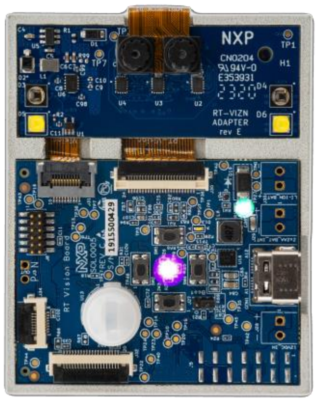


Figure 41: MSD Mode Lights

In addition to the flashing lights, the board will also enumerate as a USB Storage Device by your computer's OS.

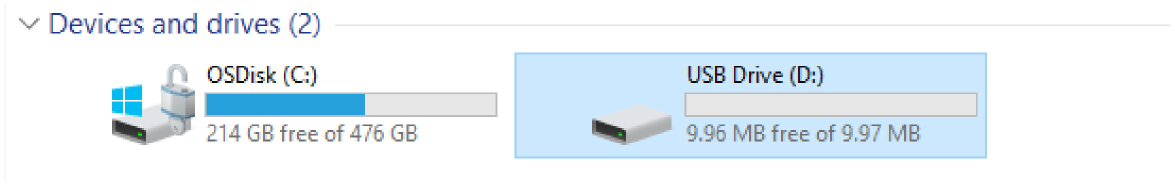


Figure 42: MSD USB Drive Enumeration

To flash a binary, drag and drop it onto the USB Drive associated with the kit. If your SLN-VIZNAS-IOT kit is running from Application Bank A (see **Device Memory Map**), you must provide a binary for Application Bank B and vice versa (see **Generate Bank B Binary**). This is a protection mechanism which helps ensure that there is always at least one "good" binary in case a corrupted image gets flashed due to a crash during the flashing process.

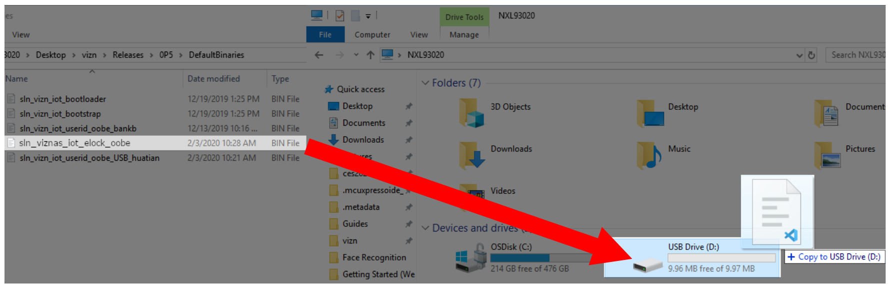


Figure 43: Dragging-and-Dropping New Binary

The new binary will be copied onto your SLN-VIZNAS-IOT, and the kit will automatically restart once flashing is complete.

5.4 Generate Bank B Binary

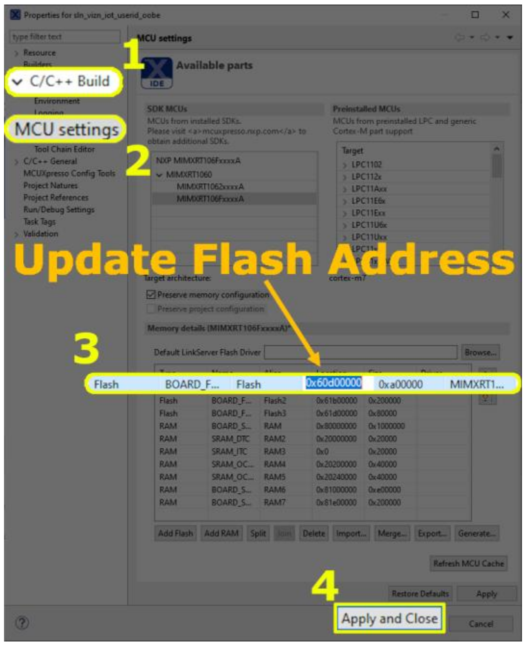
If the board is currently running from Application Bank A, MSD, OTA, and OTW flashing will require a binary created for Application Bank B and vice versa. This is a protection mechanism designed to help ensure that there is always at least one "good" binary flashed at any given time. To discover which application bank is currently in use, the **"version"** CLI command can be used.

To generate a binary for Application Bank B in MCUXpresso you must change the flash address for your kit in MCUXpresso. To do so, right-click on the **sln_viznas_iot_elock_oobe** application in in the **Project Explorer** panel and click on **Properties.**

```
SHELL>> version
SHELL>> Firmware Version:2.0.0 Bank:AppA
OASIS LITE:v4.7.1
```

Figure 44: "version" Command Output

Under the **Properties** dialog window that appears, click the drop-down arrow next to **C/C++ Build,** and select **MCU settings.** Change the **Flash** address from 0x60300000 to 0x60D00000, then click **Apply and Close.**



Flash	BOARD_F...	Flash	0x60d00000	0xa00000	MIMXRT1...
Flash	BOARD_F...	Flash2	0x61b00000	0x200000	
Flash	BOARD_F...	Flash3	0x61d00000	0x80000	
RAM	BOARD_S...	RAM	0x80000000	0x1000000	
RAM	SRAM_DTC	RAM2	0x20000000	0x20000	
RAM	SRAM_ITC	RAM3	0x0	0x20000	
RAM	SRAM_OC...	RAM4	0x20200000	0x40000	
RAM	SRAM_OC...	RAM5	0x20240000	0x40000	
RAM	BOARD_S...	RAM6	0x81000000	0xe00000	
RAM	BOARD_S...	RAM7	0x81e00000	0x200000	

Figure 45: Create Flash Bank B Binary

Rebuild your application using the steps found under **Building and Programming,** making sure to generate a binary from the .axf. The generated binary will be able to reflash the main application when the kit is running from Application Bank A.

6 Automated Manufacturing Tools

NXP provides a package of scripts that can be used for securely programming devices on the production line. This collection of scripts is called Ivaldi.

6.1 About Ivaldi

Ivaldi is a package of software scripts and tools that are responsible for manufacturing and re-programming without needing access to a J-Link.

The Ivaldi scripts make use of the serial downloader mode feature of the RT106F's boot ROM to communicate with an application called Flashloader that is programmed into the RT106F. The Flashloader which is programmed into RAM then communicates with a program called ***blhost*** which controls various parts of the chip and flash.

Ivaldi was created to focus on the build infrastructure of a customer's development and manufacturing cycle. Its primary focuses are:

- Factory programming and device set up
- Enabling HAB and eXIP
- Signing images for Application Banks A/B
- Writing and accessing OTP fuses

The rest of this chapter discusses general (i.e. unsecure) flashing of a device.

6.2 Requirements

The following requirements must be satisfied to run Ivaldi. It has been tested in Windows, Mac, and Linux environments.

- OpenSSL
- Python 3.6.x with virtualenv
- Linux/Ubuntu for Windows

Note: The package contains valuable README files. To set up the environment, follow the README.md file located in the root folder of the Ivaldi package. Without doing this, the tool will not work.

The Ivaldi tools can be downloaded under **Software and Tools** on the official webpage for the SLN-VIZNAS-IOT. Extract the ZIP file and open the README.md to start using the tool.

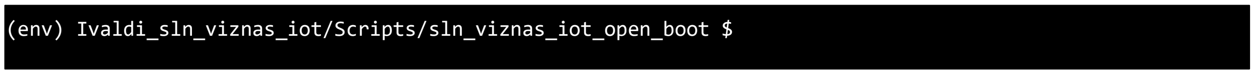
6.3 Creating a Signing Entity

The basis of the security architecture implemented in the SLN-VIZNAS-IOT is signed application images. Signing requires the use of a Certificate Authority (CA). NXP has its own CA for signing applications at the factory, but the root CA is not something that is shared with customers. The Ivaldi tools provided by NXP require signing, so the end user must create their own CA and signing artifacts.

6.3.1 Using Ivaldi to Generate Signing Artifacts

Ivaldi includes a script to generate all of the artifacts needed to properly sign application binaries and generate a FICA table. Prior to running the script, the Ivaldi environment must be set up completely as described in the README.md in the top-level directory.

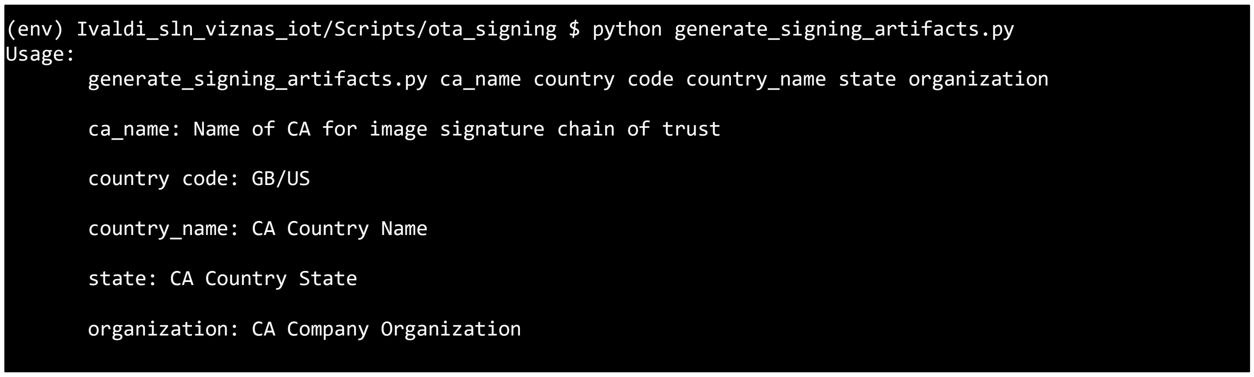
After following the README, the environment should look similar to that shown below. Take notice of the **(env)** at the beginning of the prompt.



```
(env) Ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_open_boot $
```

Figure 46: Virtual Env Prompt

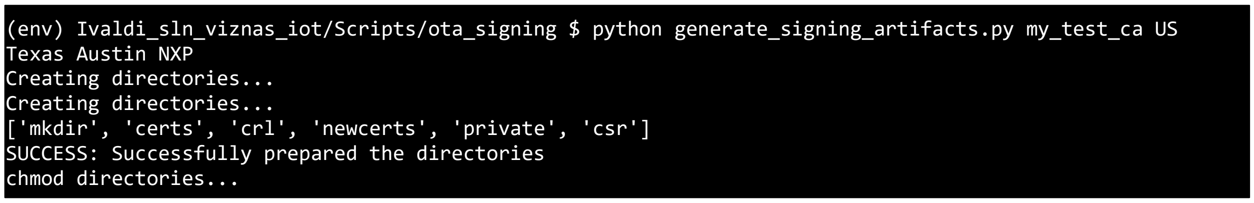
In the Python virtual environment, navigate to **Tools/Scripts/ota_signing.** Run the **generate_signing_arfifacts.py** script. When running without any arguments, the usage will be displayed.



```
(env) Ivaldi_sln_viznas_iot/Scripts/ota_signing $ python generate_signing_artifacts.py
Usage:
       generate_signing_artifacts.py ca_name country code country_name state organization

       ca_name: Name of CA for image signature chain of trust

       country code: GB/US

       country_name: CA Country Name

       state: CA Country State

       organization: CA Company Organization
```

Figure 47: generate_signing_artifacts.py Usage

Now, type in a name for your CA (**my_test_ca** is used as an example), along with the required location and organization information. **<u>When prompted for passwords for the PEM files, use the same password for all of them for this exercise</u>**. You can always re-generate a more secure CA when you're ready to prepare for production. The following figure shows an excerpt from the terminal output of the generation script.



```
(env) Ivaldi_sln_viznas_iot/Scripts/ota_signing $ python generate_signing_artifacts.py my_test_ca US
Texas Austin NXP
Creating directories...
Creating directories...
['mkdir', 'certs', 'crl', 'newcerts', 'private', 'csr']
SUCCESS: Successfully prepared the directories
chmod directories...
```

Figure 48: generate_signing_artifacts.py Example

You should now have a CA and signing certificates. Reference the README.md in the **Scripts/ota_signing** folder for more details about the directory structure and files that were generated by the script.

6.4 Move Boot Jumper

Running the flashing scripts, the SLN-VIZNAS-IOT boot jumper **(J27)** will need to be moved from position “1” to position “0”.

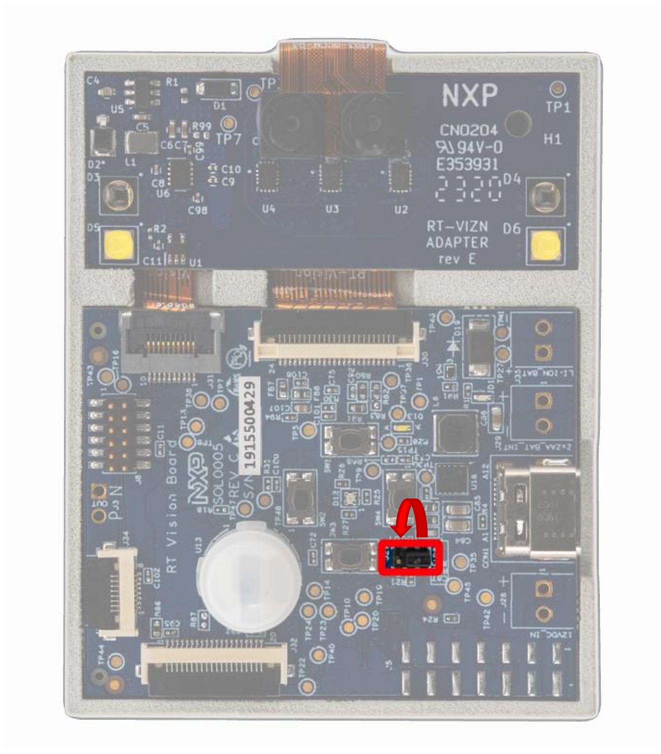


Figure 49: SLN-VIZNAS-IOT Boot Jumper

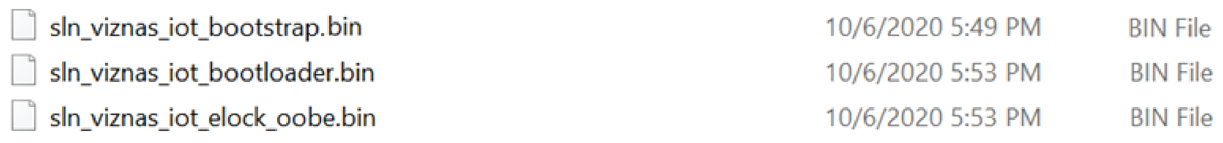
This jumper will need to be moved back to its original position after flashing the board using any of the scripts found in the following sections in order for the newly flashed firmware to run.

6.5 Open Boot Programming

The SLN-VIZNAS-IOT can be programmed with or without security features enabled. The steps to enable secure booting of the kit can be found in later sections. This section will describe the steps necessary to program the board without security features enabled.

To execute the following steps, ensure the boot jumper (J27), which is located on the top of the board, is in the "0" position.

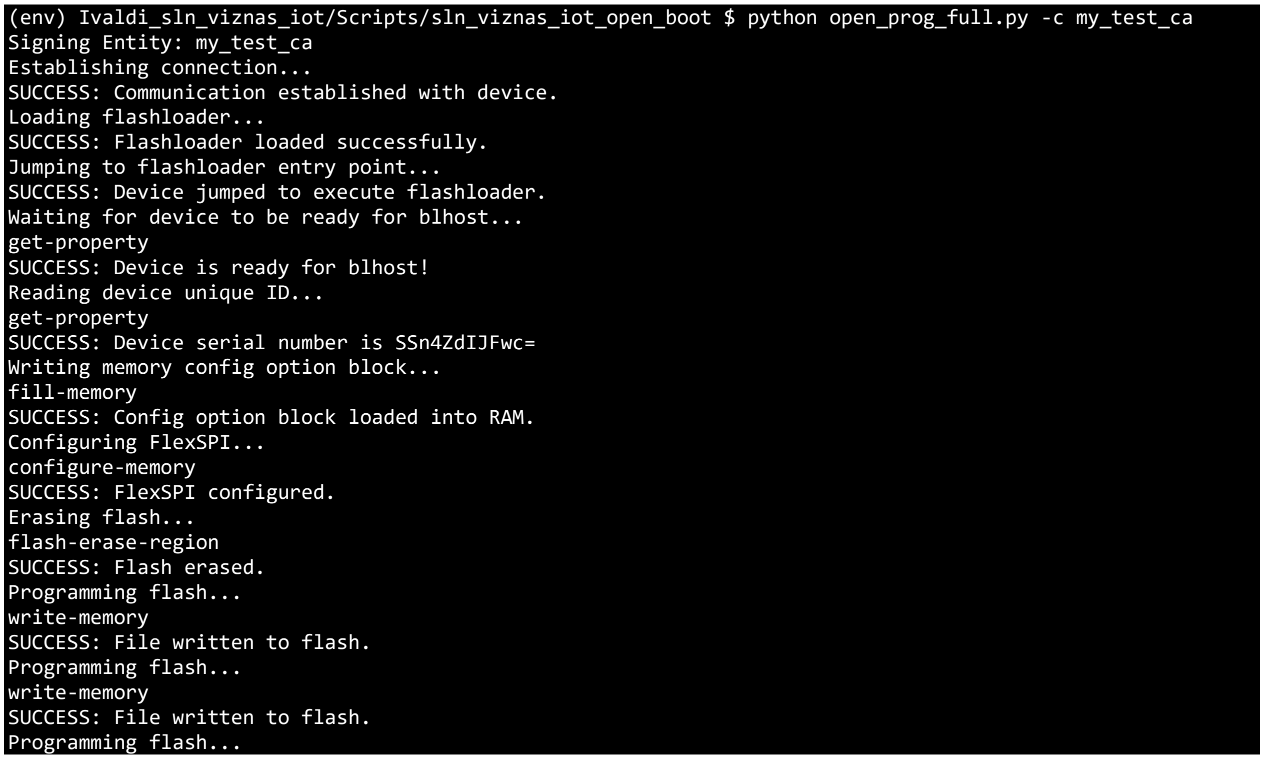
Assuming the steps to generate a signing entity, and the steps to update the boot jumper pin from the previous section have been followed, the SLN-VIZNAS-IOT can be programmed using the **open_prog_full.py** script found under the **Scripts/sln_viznas_iot_open_boot** folder. Before running the script, it's required that the **bootloader**, **bootstrap**, and **elock_oobe** binaries to be used for flashing are placed in the **Image_Binaries** folder.



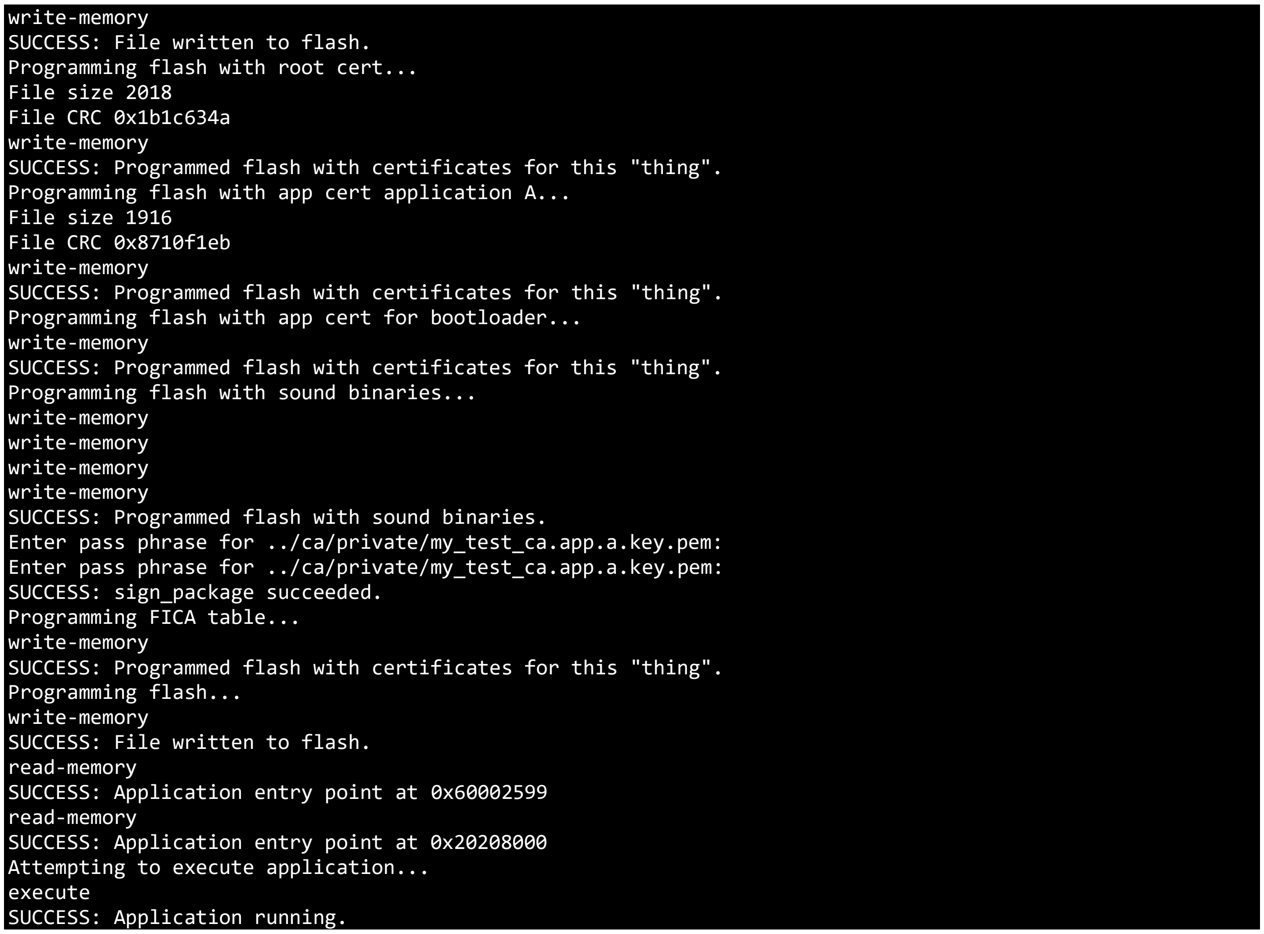
sln_viznas_iot_bootstrap.bin	10/6/2020 5:49 PM	BIN File
sln_viznas_iot_bootloader.bin	10/6/2020 5:53 PM	BIN File
sln_viznas_iot_elock_oobe.bin	10/6/2020 5:53 PM	BIN File

Figure 50: Required open_prog_full Binaries

Once the binaries are in place, navigate to the **Scripts/sln_viznas_iot_open_boot** folder and use the command "python open_prog_full.py -c CA_NAME" as shown in the figure below:



```
(env) Ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_open_boot $ python open_prog_full.py -c my_test_ca
Signing Entity: my_test_ca
Establishing connection...
SUCCESS: Communication established with device.
Loading flashloader...
SUCCESS: Flashloader loaded successfully.
Jumping to flashloader entry point...
SUCCESS: Device jumped to execute flashloader.
Waiting for device to be ready for blhost...
get-property
SUCCESS: Device is ready for blhost!
Reading device unique ID...
get-property
SUCCESS: Device serial number is SSn4ZdIJFwc=
Writing memory config option block...
fill-memory
SUCCESS: Config option block loaded into RAM.
Configuring FlexSPI...
configure-memory
SUCCESS: FlexSPI configured.
Erasing flash...
flash-erase-region
SUCCESS: Flash erased.
Programming flash...
write-memory
SUCCESS: File written to flash.
Programming flash...
write-memory
SUCCESS: File written to flash.
Programming flash...
```


```
write-memory
SUCCESS: File written to flash.
Programming flash with root cert...
File size 2018
File CRC 0x1b1c634a
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with app cert application A...
File size 1916
File CRC 0x8710f1eb
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with app cert for bootloader...
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with sound binaries...
write-memory
write-memory
write-memory
write-memory
SUCCESS: Programmed flash with sound binaries.
Enter pass phrase for ../ca/private/my_test_ca.app.a.key.pem:
Enter pass phrase for ../ca/private/my_test_ca.app.a.key.pem:
SUCCESS: sign_package succeeded.
Programming FICA table...
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash...
write-memory
SUCCESS: File written to flash.
read-memory
SUCCESS: Application entry point at 0x60002599
read-memory
SUCCESS: Application entry point at 0x20208000
Attempting to execute application...
execute
SUCCESS: Application running.
```
*Figure 51: **open_prog_full.py** Output*

The script will ask multiple times for the password used when generating your CA. Once the script completes, the board will reset itself and the board will need to be unplugged and the boot jumper placed back in its default, “1” position.

6.6 (Optional) Enabling Encrypted Execute-in-Place (eXIP) and High Assurance Boot (HAB)

The i.MX RT106F has some fundamental security enablement to protect against unsigned images and protect high-value software running on the device. These security features can be looked into at great detail by reading the RT1060 Reference manual in the RT106F Documentation area or the following whitepaper (https://www.nxp.com/docs/en/white-paper/IMXRTCROSSWP.pdf) however, the following documentation is to detail the steps to enable the **eXIP** and **HAB** features of the RT by using Python scripts which take the complication out of the process.

The Ivaldi tools, as well as containing automated OTW signing tools, also contain all the tools and scripts to enable **HAB** and **eXIP**. By the end of this section, the **bootstrap** will be signed to work with the **HAB** and the **bootloader** and the **elock_oobe** will be encrypted with individual encrypted context. The **bootloader** and **elock_oobe** have individual encrypted contexts to ensure that if any of the application banks are updated, the bootloader will not need updating.

The whole package contains the following features:

- Two individual encrypted Context for bootloader and app space.
- Potential support for bootloader update called the “bootloader loader” (coming in future releases).
- Encrypted context restoration for image failure or OTA failure.
- OTW update with eXIP support.
- Switching between eXIP and XIP for easy development.

For additional documentation, please build the docs in the Ivaldi/doc folder by following the contained README.md

6.6.1 Preparing the Environment

The following steps assumes that the section **Creating a Signing Entity** has been followed and completed, generating a CA and signing entity to create signed images. This is used to verify the signature of the application using an app certificate and CA certificate.

It also assumed that the reader is running the tools within the Ivaldi package and the paths are unchanged as delivered.

If not already done so, unzip the Ivaldi zip. In the top-level directory, open the README.md and follow all the steps to create and build the environment.

After following the README, the environment should look similar to that shown below. Take notice of the **“(env)”** at the beginning of the prompt.

```
(env) Ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_open_boot $
```

Figure 52: "Virtualenv” Prompt

Additionally, it is recommended to follow the section **Open Boot Programming** section, as this section will validate that all the image binaries are working correctly. There are several failure points that could occur while enabling HAB and eXIP so following the **Open Boot Programming** will help reduce potential failure points.

6.6.2 Generating the PKI and Signed Flashloader

The following instructions assume that the section **Creating a Signing Entity** has been completed, as it is needed to generate the CA and application certificate that will be loaded into the flash. It will also be used to generate the FICA table used to validate the application signature.

The first step is to create a signed flashloader which will be used to set everything up and communicate with blhost. The blhost tool in its simplest form is used to read and write registers, but it communicates with a flashloader. The flashloader is a RAM-based application that supports blhost communication. In normal circumstances, the flashloader can be executed without having been signed, but with HAB enabled, it needs to be signed with appropriate keys.

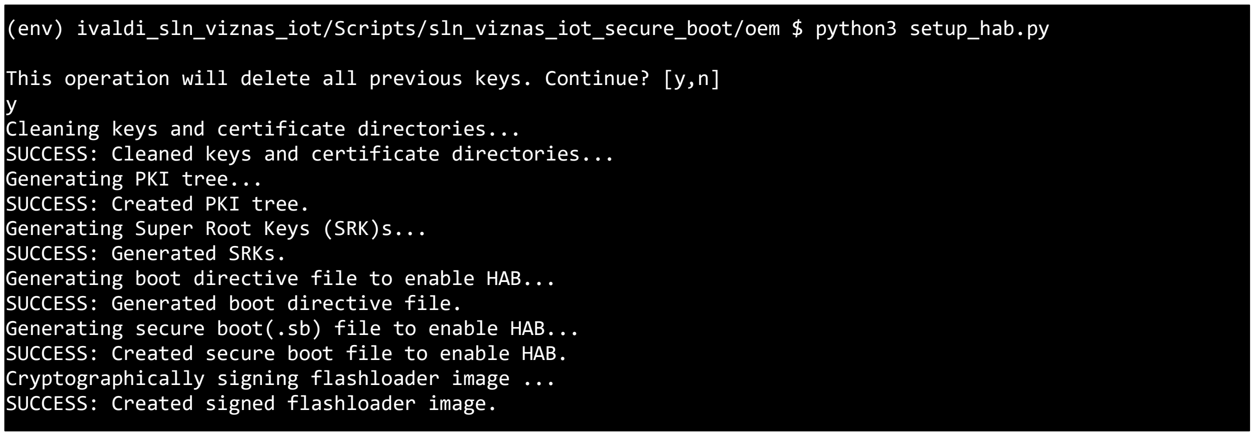
The secure boot scripts have been separated into two folders:

- **OEM –** These scripts should only be executed by the Product owner, and the output stored in a secure environment. This is because it contains important key information, which if lost, could brick boards or be open to copy cats/loss of image integrity.
- **MANF –** These scripts will be executed on the manufacturing line. They are used to execute the signed flashloader and communicate with the chip to encrypt the binaries. The scripts also contain the process of generating certificates, and the generation and programming of the FICA.

Within the Ivaldi package, navigate to the **Scripts/sln_viznas_iot_secure_boot/oem** folder and open the README within. The README starts by running the **setup_hab.py** script which is responsible for creating the PKI infrastructure and creating the signed flashloader.

PLEASE NOTE, DO NOT RUN THIS MORE THAN ONCE WITHOUT BACKING UP YOUR KEYS AND CRTS FOLDER IN THE IVALDI ROOT. THIS WILL RESULT IN BEING UNABLE TO USE FLASHLOADER AND PROGRAM NEW IMAGES VIA SERIAL DOWNLOAD MODE FOR EXISTING HAB-ENABLED DEVICES

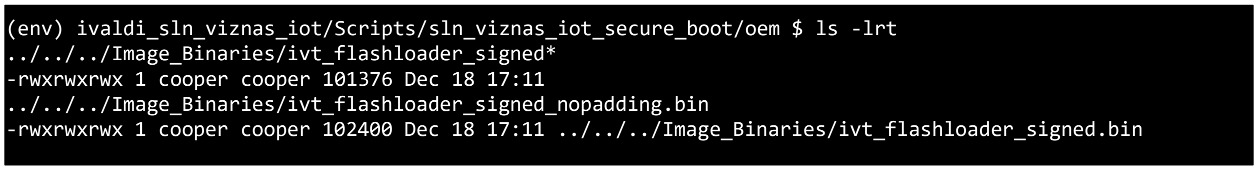
The following shows the output of the “setup_hab.py’ script that generates the PKI infrastructure and signed flashloader.



```
(env) ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_secure_boot/oem $ python3 setup_hab.py

This operation will delete all previous keys. Continue? [y,n]
y
Cleaning keys and certificate directories...
SUCCESS: Cleaned keys and certificate directories...
Generating PKI tree...
SUCCESS: Created PKI tree.
Generating Super Root Keys (SRK)s...
SUCCESS: Generated SRKs.
Generating boot directive file to enable HAB...
SUCCESS: Generated boot directive file.
Generating secure boot(.sb) file to enable HAB...
SUCCESS: Created secure boot file to enable HAB.
Cryptographically signing flashloader image ...
SUCCESS: Created signed flashloader image.
```

Figure 53: setup_hab.py Script

After this has run, you should see in the **Image_Binary** folder that the signed flashloader exists as shown below.



```
(env) ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_secure_boot/oem $ ls -lrt
../../../Image_Binaries/ivt_flashloader_signed*
-rwxrwxrwx 1 cooper cooper 101376 Dec 18 17:11
../../../Image_Binaries/ivt_flashloader_signed_nopadding.bin
-rwxrwxrwx 1 cooper cooper 102400 Dec 18 17:11 ../../../Image_Binaries/ivt_flashloader_signed.bin
```

Figure 54: Checking the Signed Flashloader

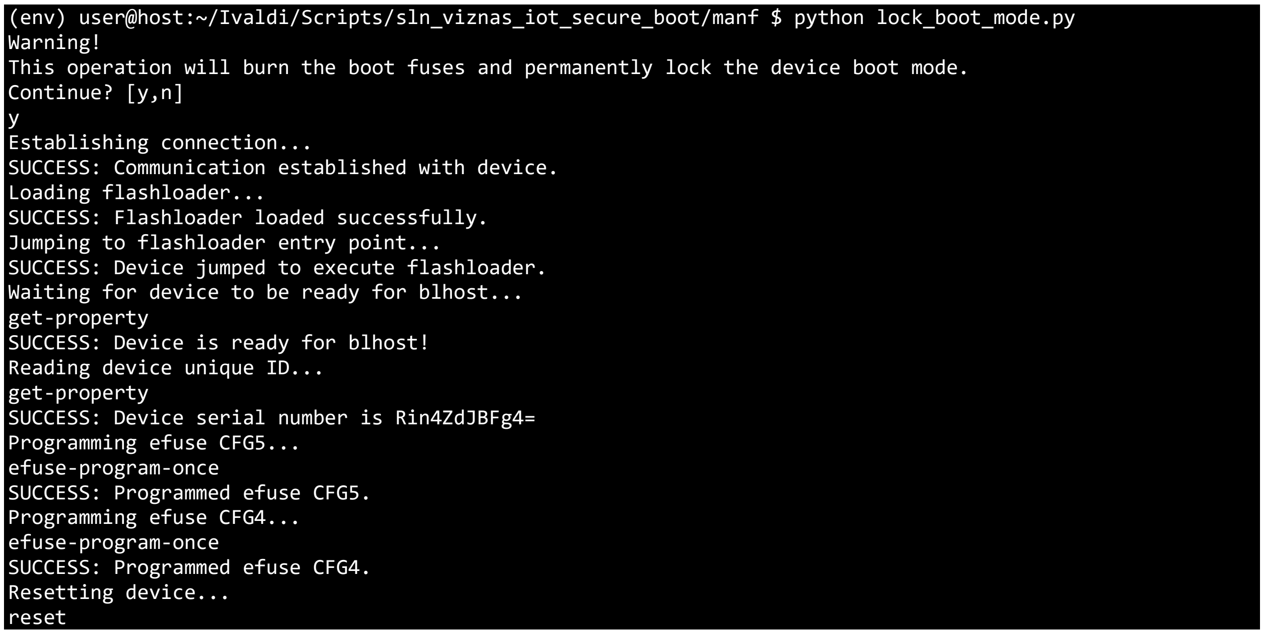
6.6.3 (Optional) Enabling Encrypted Execute in Place (eXIP)

Encrypted Execution in place (eXIP) is a feature of the i.MX RT106F that enables the chip to execute and decrypt on the fly allowing images to be store into external flash encrypted uniquely per part. This gives product makers comfort that their IP is protected, and physical attacks are not possible. It also means that the device cannot be flashed with malicious firmware that can be executed, as the device would fail with an encryption error. If the security of the device is compromised, it would also mean that any firmware bad actors are able to obtain could not be programmed into another device due to the unique nature of the encryption.

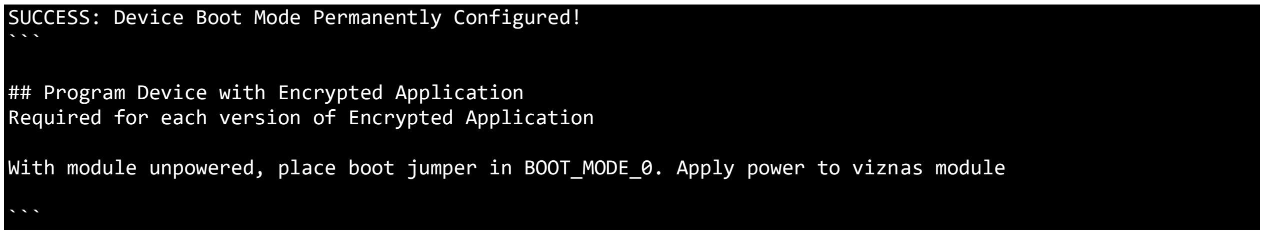
Enabling eXIP will cause execution of the elock_oobe to be slower than normal because of the on-the-fly encrypt and decryption of each instruction that is taking place. For this reason, a user may not want to enable eXIP and can skip this step if they choose.

To enable eXIP, navigate to **Scripts/sln_viznas_iot_secure_boot/manf** and run the **“lock_boot_mode.py”** script with no arguments.

The following figure shows the output of this script when run successfully:



```
(env) user@host:~/Ivaldi/Scripts/sln_viznas_iot_secure_boot/manf $ python lock_boot_mode.py
Warning!
This operation will burn the boot fuses and permanently lock the device boot mode.
Continue? [y,n]
y
Establishing connection...
SUCCESS: Communication established with device.
Loading flashloader...
SUCCESS: Flashloader loaded successfully.
Jumping to flashloader entry point...
SUCCESS: Device jumped to execute flashloader.
Waiting for device to be ready for blhost...
get-property
SUCCESS: Device is ready for blhost!
Reading device unique ID...
get-property
SUCCESS: Device serial number is Rin4ZdJBFg4=
Programming efuse CFG5...
efuse-program-once
SUCCESS: Programmed efuse CFG5.
Programming efuse CFG4...
efuse-program-once
SUCCESS: Programmed efuse CFG4.
Resetting device...
reset
```



```
SUCCESS: Device Boot Mode Permanently Configured!
```

Program Device with Encrypted Application
Required for each version of Encrypted Application

With module unpowered, place boot jumper in BOOT_MODE_0. Apply power to viznas module

```
```

Figure 55: Enabling eXIP and Locking Boot Mode

PLEASE NOTE, LOCKING THE BOOT MODE IS A PERMANENT, ONE-WAY ACTION THAT CANNOT BE UNDONE

6.6.4 Enabling High Assurance Boot (HAB)

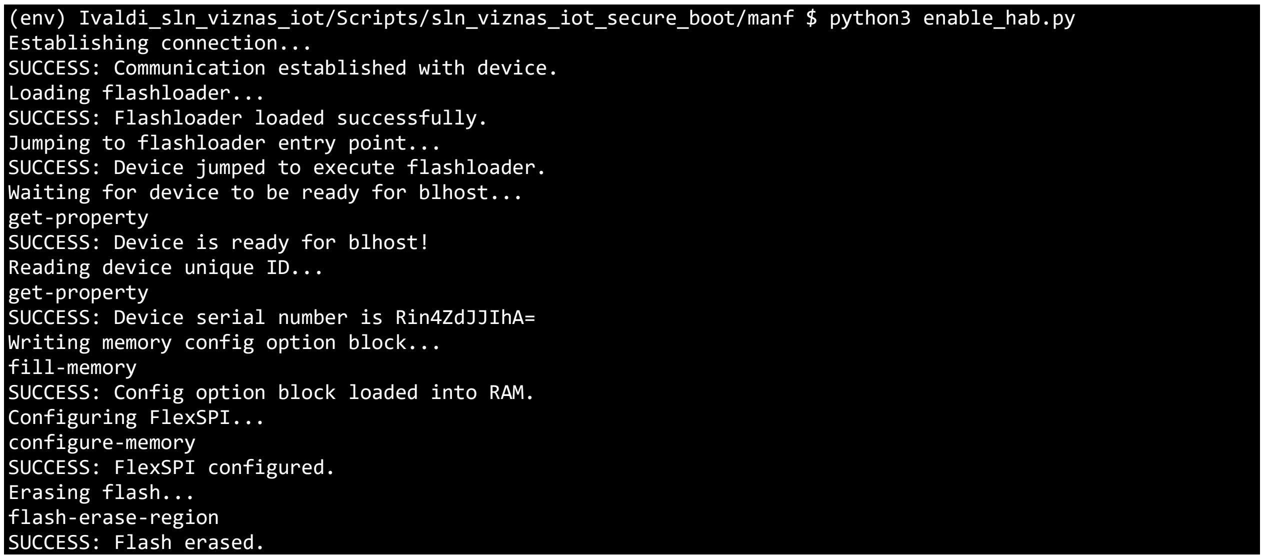
High Assurance Boot (HAB) is a feature of the RT106F that forces the ROM (Read Only Memory) to only boot into a signed image. This ensures image integrity and prevents physical and remote attacks from power on.

To execute the following steps, ensure the boot jumper (J27), which is located on the top of the board, is in the "0" position.

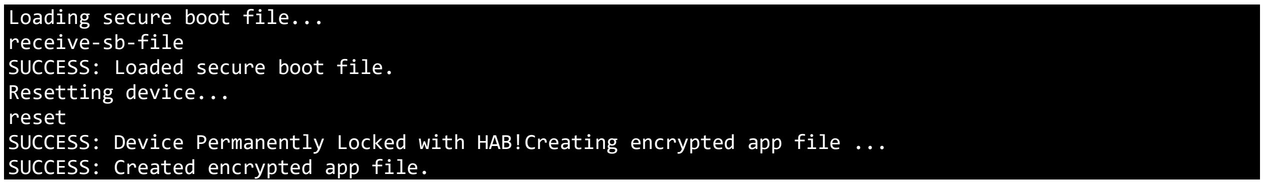
The following instructions will show how to enable the HAB on the RT106F using the PKI infrastructure created in the section **Generating the PKI and Signed Flashloader**.

Navigate to the "**Scripts/sln_viznas_iot_secure_boot/manf**" from the Ivaldi root and locate the **"enable_hab.py"** python script.

To enable HAB, run the **"enable_hab.py"** script as shown here. **NOTE: Once this script is run, it cannot be undone, thereby locking your device to only allow secure, signed images to run. This will cause you to be unable to debug the kit using MCUXpresso.**



```
(env) Ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_secure_boot/manf $ python3 enable_hab.py
Establishing connection...
SUCCESS: Communication established with device.
Loading flashloader...
SUCCESS: Flashloader loaded successfully.
Jumping to flashloader entry point...
SUCCESS: Device jumped to execute flashloader.
Waiting for device to be ready for blhost...
get-property
SUCCESS: Device is ready for blhost!
Reading device unique ID...
get-property
SUCCESS: Device serial number is Rin4ZdJJIhA=
Writing memory config option block...
fill-memory
SUCCESS: Config option block loaded into RAM.
Configuring FlexSPI...
configure-memory
SUCCESS: FlexSPI configured.
Erasing flash...
flash-erase-region
SUCCESS: Flash erased.
```



```
Loading secure boot file...
receive-sb-file
SUCCESS: Loaded secure boot file.
Resetting device...
reset
SUCCESS: Device Permanently Locked with HAB!Creating encrypted app file ...
SUCCESS: Created encrypted app file.
```

Figure 56: Enabling HAB using enable_hab.py

⚠	**PLEASE NOTE, IF YOU LOSE THE SIGNED FLASHLOADER AND CERT/KEYS, THE BOARD WILL NO LONGER BE FUNCTIONAL AS HAB ENSURES ONLY SIGNED IMAGES CAN BOOT.**

6.6.5 Preparing Bootstrap, Bootloader, and Main Application Images

The following section describes how to generate images for the **bootstrap**, **bootloader,** and main app (**elock_oobe** in this case) and put them into the correct folder in preparation for creating the signed binary which will be written into the flash of the device.

To generate the images, we will be signing and flashing, make sure you have the **bootstrap**, **bootloader_4343W**, and **elock_oobe** imported into the MCUXpresso workspace as shown in the following figure. If these projects are not in your MCUXpresso workspace, follow the steps in the **Import a SLN-VIZNAS-IOT Project** section.

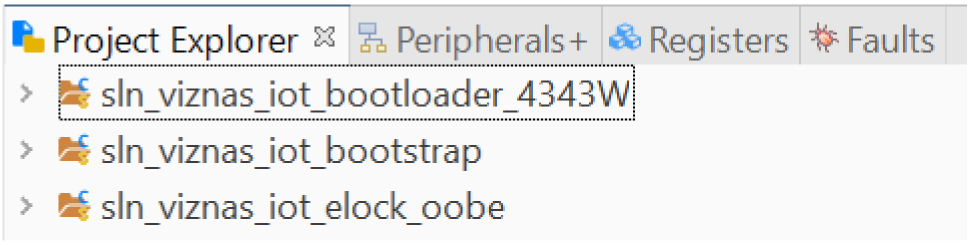
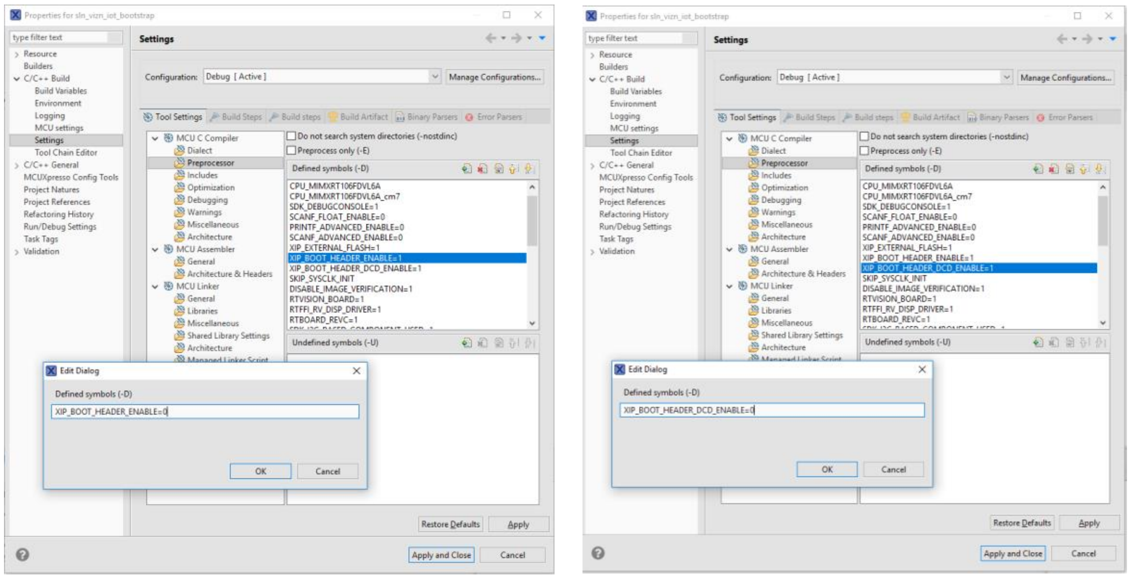


Figure 57: Importing the Applications for HAB and eXIP

Before creating the images, a modification needs to be made to the bootstrap. The IVT gets created by the Ivaldi scripts which means it needs to be removed from the default binary. To do this, right click on the bootstrap project and go to **Properties -> C/C++ Build -> Settings -> Preprocessors** and set the **XIP_BOOT_HEADER_ENABLE** and **XIP_BOOT_HEADER_DCD_ENABLE** to zero as shown here.



Properties for sln_vizn_iot_bootstrap

Settings

Configuration: Debug [Active]

Tool Settings → MCU C Compiler → Preprocessor

Defined symbols (-D)

```
CPU_MIMXRT106FDVL6A
CPU_MIMXRT106FDVL6A_cm7
SDK_DEBUGCONSOLE=1
SCANF_FLOAT_ENABLE=0
PRINTF_ADVANCED_ENABLE=0
SCANF_ADVANCED_ENABLE=0
XIP_EXTERNAL_FLASH=1
XIP_BOOT_HEADER_ENABLE=1
XIP_BOOT_HEADER_DCD_ENABLE=1
SKIP_SYSCLK_INIT
DISABLE_IMAGE_VERIFICATION=1
RTVISION_BOARD=1
RTFFI_RV_DISP_DRIVER=1
RTBOARD_REVC=1
```

Edit Dialog — Defined symbols (-D): `XIP_BOOT_HEADER_ENABLE=0`

Edit Dialog — Defined symbols (-D): `XIP_BOOT_HEADER_DCD_ENABLE=0`

Figure 58: Unsetting the XIP Boot Header

After this change, hit the **Build** button to generate a binary (.bin) file for the bootstrap.

As the Ivaldi script only accepts a .srec file, it is necessary to generate one. To do so, navigate to the **Debug** folder of the bootstrap application, right click on the .axf file and select **Binary Utilities -> Create S-Record.**

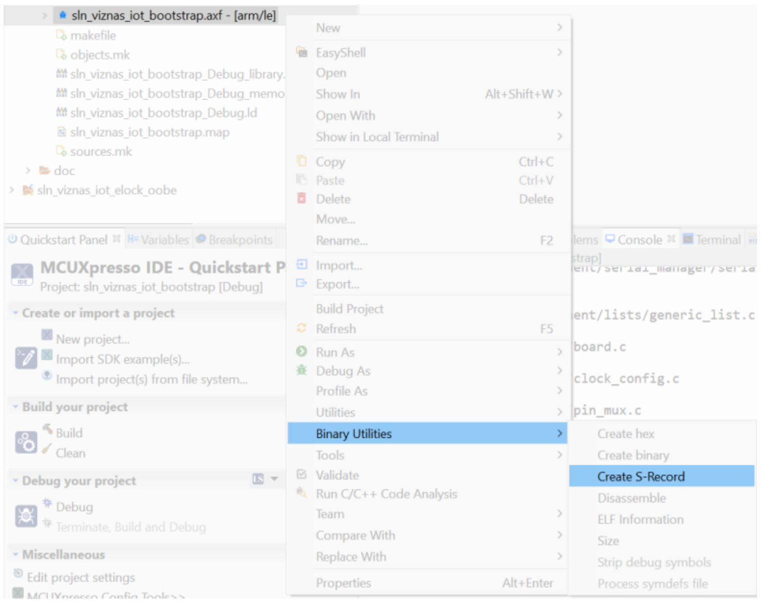
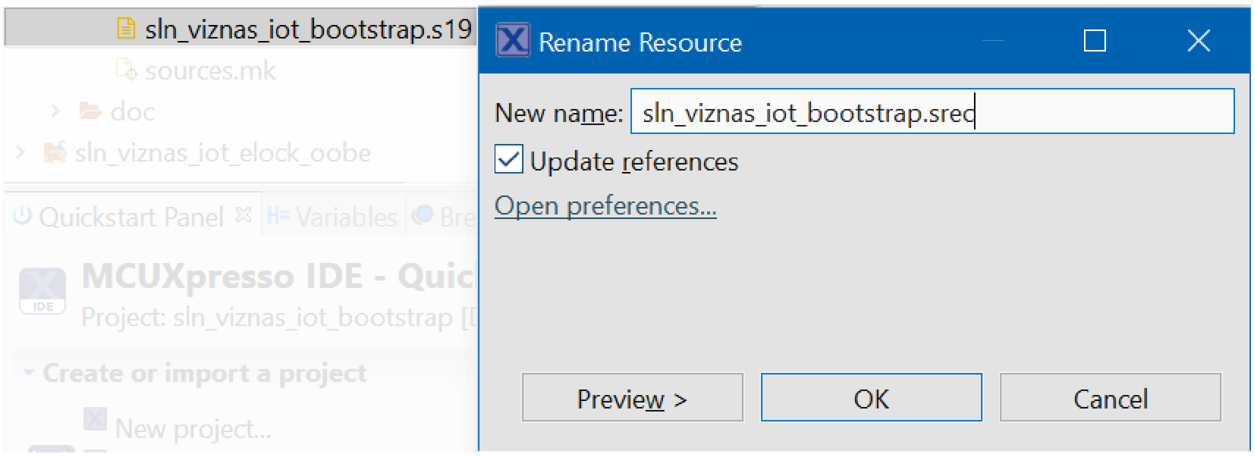


Figure 59: Generating the srec

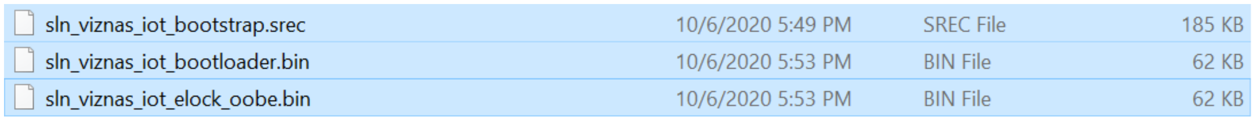
Create S-Record generates a “.s19” file, while our script requires “.srec” files. Simply right click on the s19 file generated in the previous step and rename it with the “.srec” file extension type.



New name: sln_viznas_iot_bootstrap.srec

Figure 60: Changing File Type to srec

Continue to build the **bootloader_4343W** and **elock_oobe** in the usual way. Once these applications are built, it is necessary to generate a binary for both projects using the ".axf" which is created by default. Like the srec generated in the previous step, a binary can be generated by navigating to the **Debug** folder in both the **bootloader_4343W** and **elock_oobe** applications, right-clicking on the .axf file and clicking on **Binary Utilities -> Create Binary.** Once the required images have been created, copy the two binaries and srec into the **Ivaldi/Image_Binaries** package as shown here.



sln_viznas_iot_bootstrap.srec	10/6/2020 5:49 PM	SREC File	185 KB
sln_viznas_iot_bootloader.bin	10/6/2020 5:53 PM	BIN File	62 KB
sln_viznas_iot_elock_oobe.bin	10/6/2020 5:53 PM	BIN File	62 KB

Figure 61: "Image_Binaries" Expected Folder Contents

6.6.6 Generating Secure Binary

This section will describe the instructions for creating a secure binary in preparation for programming it into the flash of the device.

Navigate to the **Scripts/sln_viznas_iot_secure_boot/oem** folder and open the **secure_app.py** python script. Inside this file contains the path and the file names of the binaries that will be used to create the secure binary. The following figure shows these path definitions inside the **secure_app.py** script.

```
DEF_HAB_PATH = IMG_DIR + “/sln_viznas_iot_bootstrap.srec”
DEF_HAB_PATH = IMG_DIR + “/sln_viznas_iot_bootloader.bin”
DEF_HAB_PATH = IMG_DIR + “/sln_viznas_iot_elock_oobe.bin
```

Figure 62: Secure App File Names

NOTE: The binary for the bootloader MUST be named "sln_viznas_iot_bootloader" ***NOT*** **"sln_viznas_iot_bootloader_4343W" or the above paths must be updated.**

It's important to know that the file names that are in this file are the names the script will look for. If the files in your **Image_Binaries** folder differ, please change the names of the files or modify the script to match. If the files do not match, unpredictable behavior will occur.

After aligning the files names under the **Image_Binaries** folder to those found in the script, run the **secure_app.py** script with the arguments corresponding to your desired setup:

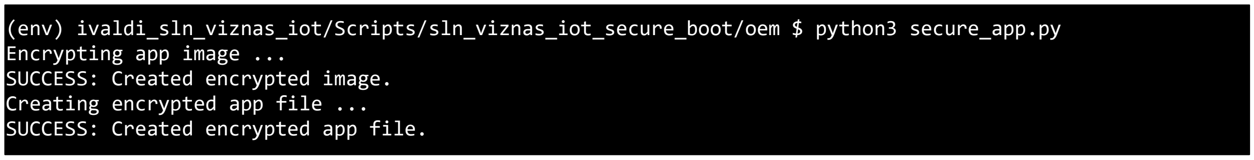
WITH eXIP ENABLED:

“python3 secure_app.py”

or *WITHOUT eXIP ENABLED:*

“python3 secure_app.py –s”

The output of the script when run with eXIP is shown below:



```
(env) ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_secure_boot/oem $ python3 secure_app.py
Encrypting app image ...
SUCCESS: Created encrypted image.
Creating encrypted app file ...
SUCCESS: Created encrypted app file.
```

Figure 63: secure.py output for securing images

The output shows the combining of the images into one consolidated, secure image. Navigate to the **Image_Binaries** directory and locate the “**boot_crypt_image_production1v0.sb**” or “**boot_sign_image_production1v0.sb**” in the case of signed only. This will be used in a later section when flashing the device.

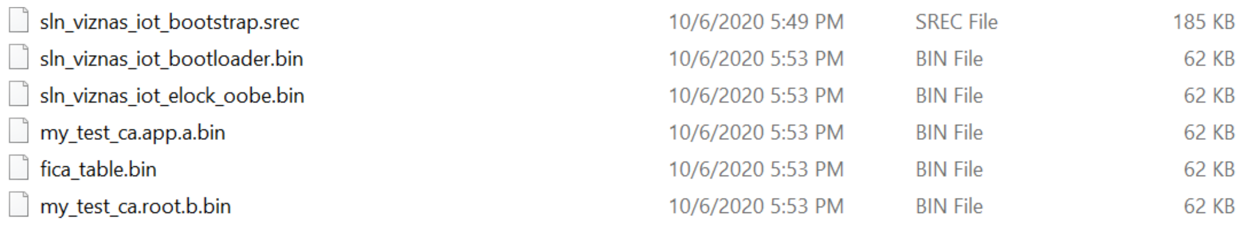
6.6.7 Preparing for Programming the Device

The following section describes the steps that need to be executed to ensure all the artifacts are available in preparation for programming the device. It is assumed that the section **Creating a Signing Entity** seen followed to generate a CA and Application certificate.

Copy all the file system generated files to the **Image_Binaries** folder within the Ivaldi root directory. The files that need to be copied are:

- **app_crt.bin** – This is the public image signing certificate
- **ca_crt.bin** – This is the public image CA certificate
- **fica_table.bin** – This is the Flash Image Configuration Area generated when creating a signed bootloader and elock_oobe.

At this point, your **Image_Binaries** folder should look similar to the following:



sln_viznas_iot_bootstrap.srec	10/6/2020 5:49 PM	SREC File	185 KB
sln_viznas_iot_bootloader.bin	10/6/2020 5:53 PM	BIN File	62 KB
sln_viznas_iot_elock_oobe.bin	10/6/2020 5:53 PM	BIN File	62 KB
my_test_ca.app.a.bin	10/6/2020 5:53 PM	BIN File	62 KB
fica_table.bin	10/6/2020 5:53 PM	BIN File	62 KB
my_test_ca.root.b.bin	10/6/2020 5:53 PM	BIN File	62 KB

Figure 64: "Image_Binaries" Content

6.6.8 Enabling and Programming the Signed and Encrypted Binaries

This section describes the usage of the **"customer_prog_sec_app.py"** script which flashes the signed image binary we created in the **Generating Secure Binary** section onto the kit.

The **"customer_prog_sec_app.py"** python script does several things.

- Runs the signed Flashloader for configuration
- Erase the current flash
- Programs the following:
  - Signed Bootstrap
  - Encrypted/Unencrypted bootloader and elock_oobe
  - Application image signing certificate
  - CA Image certificate
  - Device key and certificate

<u>To execute the following steps, move the jumper J27, which is located on the top of the board into the "0" position.</u>

Navigate to the **Scripts/sln_viznas_iot_secure_boot/manf** folder in the Ivaldi root and run the following command based on your desired security configuration:

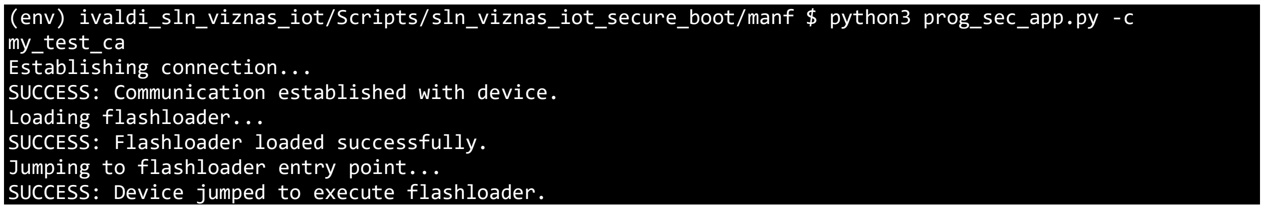
WITH eXIP ENABLED:

"python3 customer_prog_sec_app.py"

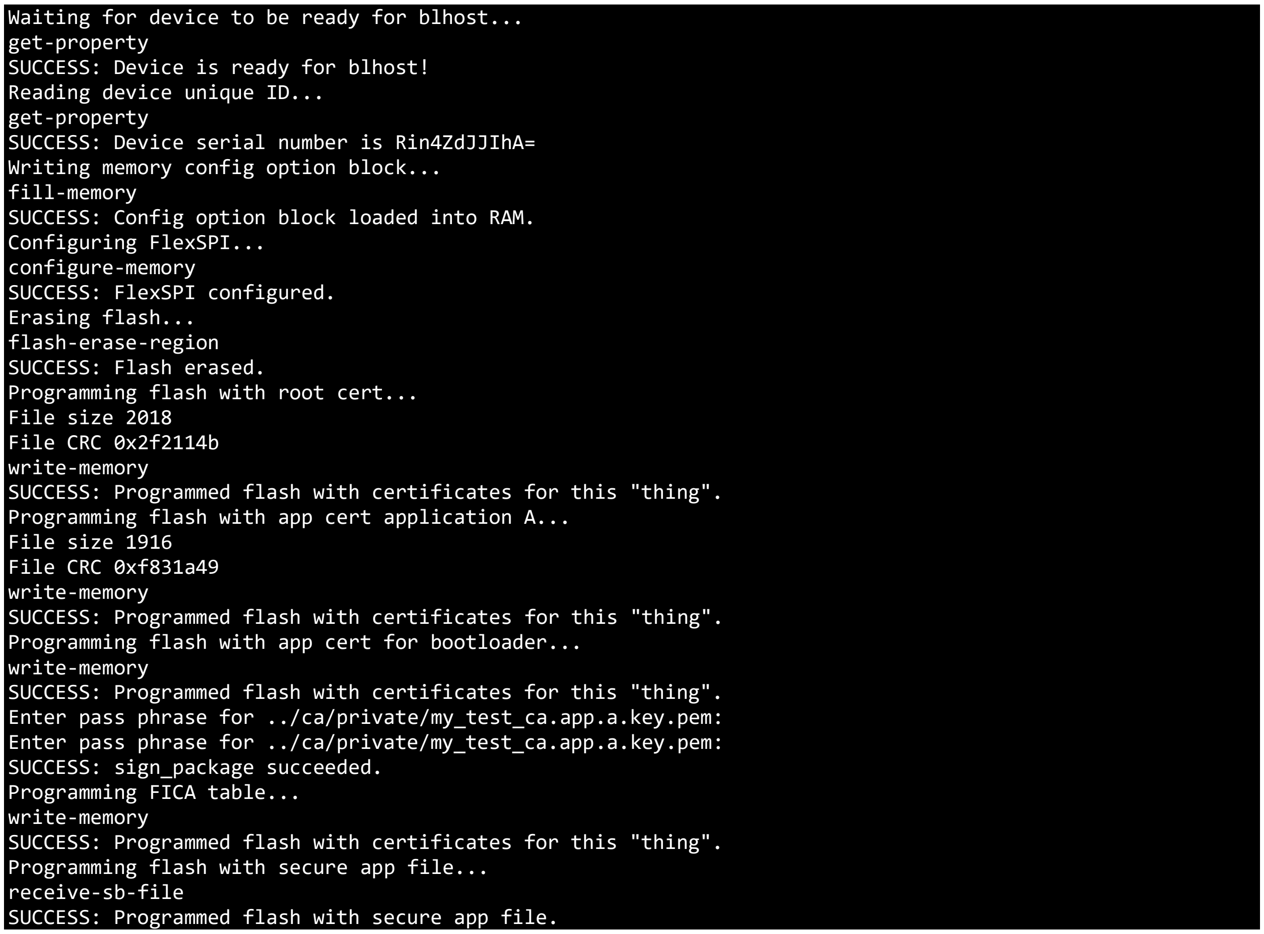
or WITHOUT eXIP ENABLED:

"python3 customer_prog_sec_app.py -s"

The following figure shows example output for the command if run with eXIP enabled:



```
(env) ivaldi_sln_viznas_iot/Scripts/sln_viznas_iot_secure_boot/manf $ python3 prog_sec_app.py -c
my_test_ca
Establishing connection...
SUCCESS: Communication established with device.
Loading flashloader...
SUCCESS: Flashloader loaded successfully.
Jumping to flashloader entry point...
SUCCESS: Device jumped to execute flashloader.
```



```
Waiting for device to be ready for blhost...
get-property
SUCCESS: Device is ready for blhost!
Reading device unique ID...
get-property
SUCCESS: Device serial number is Rin4ZdJJIhA=
Writing memory config option block...
fill-memory
SUCCESS: Config option block loaded into RAM.
Configuring FlexSPI...
configure-memory
SUCCESS: FlexSPI configured.
Erasing flash...
flash-erase-region
SUCCESS: Flash erased.
Programming flash with root cert...
File size 2018
File CRC 0x2f2114b
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with app cert application A...
File size 1916
File CRC 0xf831a49
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with app cert for bootloader...
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Enter pass phrase for ../ca/private/my_test_ca.app.a.key.pem:
Enter pass phrase for ../ca/private/my_test_ca.app.a.key.pem:
SUCCESS: sign_package succeeded.
Programming FICA table...
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with secure app file...
receive-sb-file
SUCCESS: Programmed flash with secure app file.
```

Figure 65: Using "cust_prog_sec_app.py"

This will start the manufacturing process to bring a factory new device (empty flash) to a device running a signed bootstrap (HAB enabled) and encrypted bootloader and elock_oobe stored into flash.

If done correctly, the device will reset itself. One complete, the board will need to be unplugged and the boot jumper (J27) will need to be returned to its default “1” position.

⚠	**PLEASE NOTE, THE LOCK_DEVICE.PY SHOULD ONLY BE USED IN PRODUCTION AS THIS DISABLES DEBUGGER ACCESS**

7 Filesystem

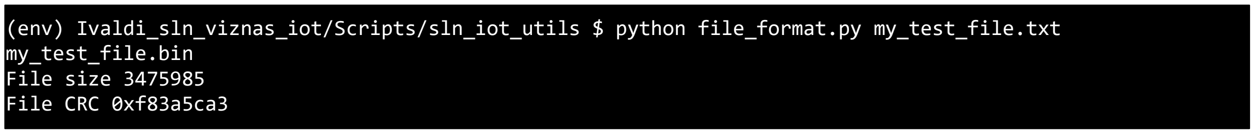
The SLN-VIZNAS-IOT has implemented a custom file system to manage files on-chip. A custom file system is used because:

The device executes code from flash (**XiP**) which means it needs to read flash from RAM functions.

HyperFlash has 256 KB sector sizes which do not allow for the granularity of files.

Update in-place features have been added to allow the updating of a big sector without a costly (in time) erase.

Within Ivaldi, there is a script that converts any file into a filesystem-compatible binary file. Any file that gets programmed to the filesystem must first pass through this script. This script is called **file_format.py** and is located in **Scripts/sln_viznas_iot_utils**.



```
(env) Ivaldi_sln_viznas_iot/Scripts/sln_iot_utils $ python file_format.py my_test_file.txt
my_test_file.bin
File size 3475985
File CRC 0xf83a5ca3
```

Figure 66: file_format.py Usage

8 Document Details

8.1 References

The following references are available to supplement this document:

Document/Link	Remark
http://www.nxp.com/MCUXpresso	MCUXpresso IDE Download
https://www.nxp.com/docs/en/user-guide/MCUXpresso_IDE_User_Guide.pdf	MCUXpresso IDE User Guide
https://www.nxp.com/docs/en/user-guide/SLN-VIZNAS-IOT-UG.pdf	SLN-VIZNAS-IOT User Guide
https://www.nxp.com/mcu-vision2	SLN-VIZNAS-IOT Home Page

Table 3: Reference Documents

8.2 Acronyms, Abbreviations, & Definitions

Acronym	Meaning	(Definition)
FTDI	Future Technology Devices International	
GUI	Graphic User Interface	
IOT	Internet of Things	
IVT	Instruction Vector Table	
JTAG	Joint Test Action Group	
MANF	Manufacturer	
MCU	Microcontroller Unit	
MEMS	Micro-Electro-Mechanical System	
MSD	Mass Storage Device	
OEM	Original Equipment Manufacturer	
OTW	Over the Wire	
OTP	One Time Programmable	
ROM	Read Only Memory	
RTOS	Real-Time Operating System	
SDK	Software Development Kit	
UART	Universal asynchronous receiver-transmitter	

Table 4: Abbreviations and Definitions

8.3 Revision History

Date	Version	Details of Change	Author	Reviewers
11/4/20	Production 1.2	Added Clock Drive Strength Section and fix to the Open Boot Programming section	Cooper Carnahan	NXP
10/28/20	Production 1.1	Minor Corrections	Cooper Carnahan	NXP
10/13/20	Production 1.0	Initial Version	Cooper Carnahan	NXP

Table 5: Revision History