MCU VIZN Solution Developer's Guide

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 Introduction

NXP's MCU-based SLN-VIZN-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
- 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 versions **11.1.0 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 7 / 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-VIZN-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-VIZN-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/

2 Getting to Know the SLN-VIZN-IOT

2.1 Hardware Overview

The SLN-VIZN-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.

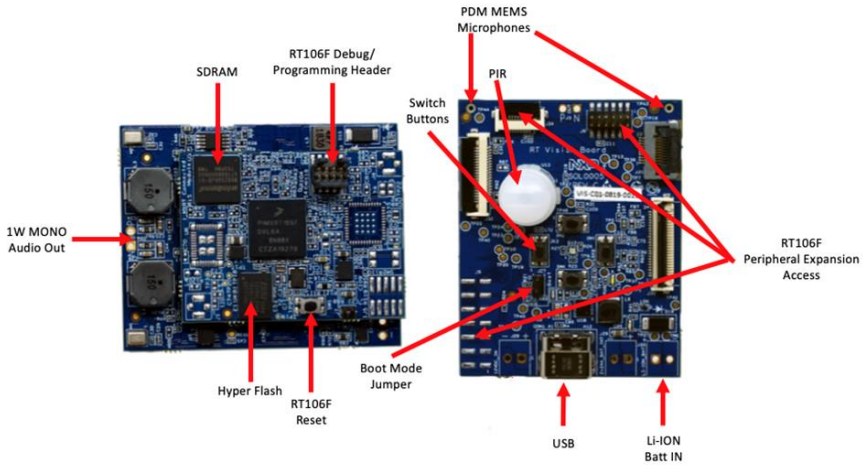


Figure 1: SLN-VIZN-IOT Base Board + Expansion Board Peripherals

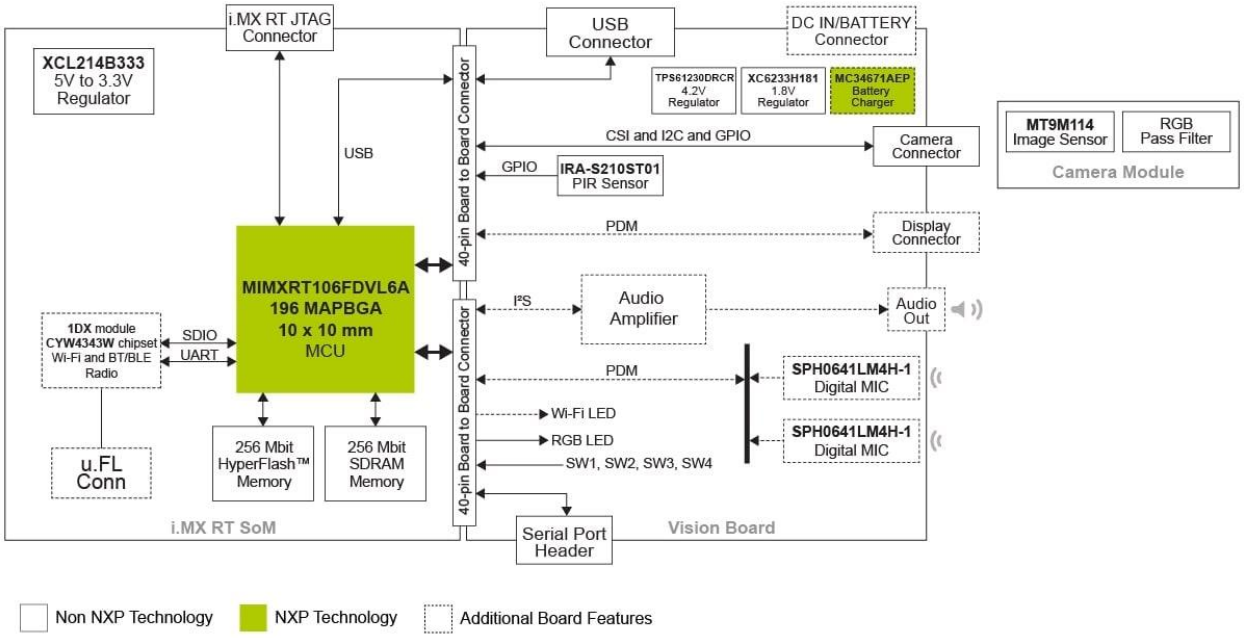


Figure 2: SLN-VIZN-IOT Hardware Block Diagram

2.2 Software Overview

The SLN-VIZN-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.

The below figure shows a high-level software architecture diagram. This figure shows everything that is included in the SDK for the SLN-VIZN-IOT package, though not all of these features are implemented in demo applications.

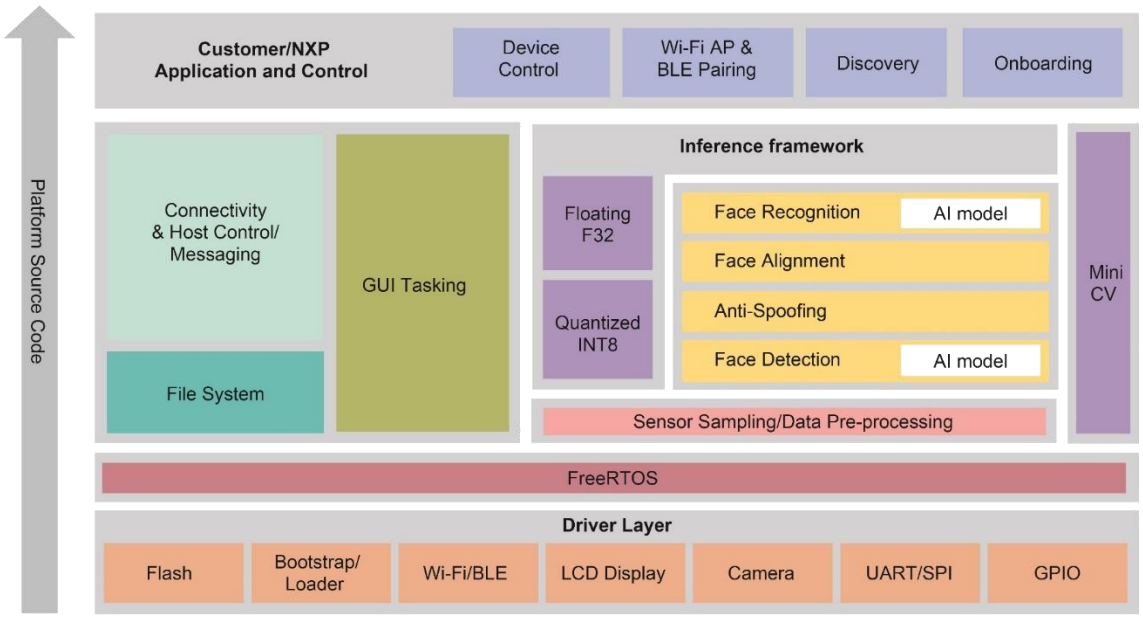


Figure 3: SLN-VIZN-IOT Software Block Diagram

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.

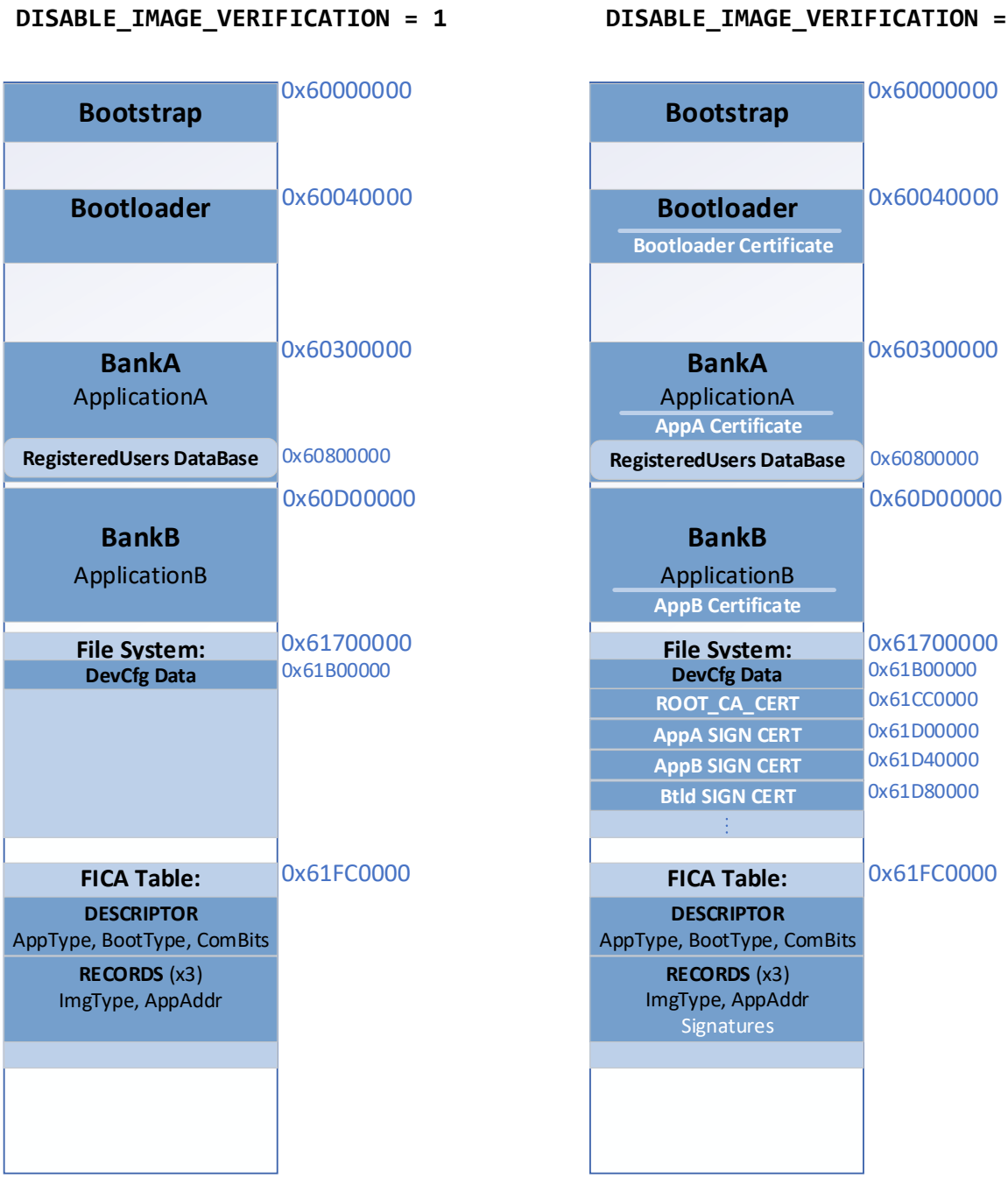


Figure 4: Device Memory Map

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. **The state of the board from factory is with all security checks disabled.**

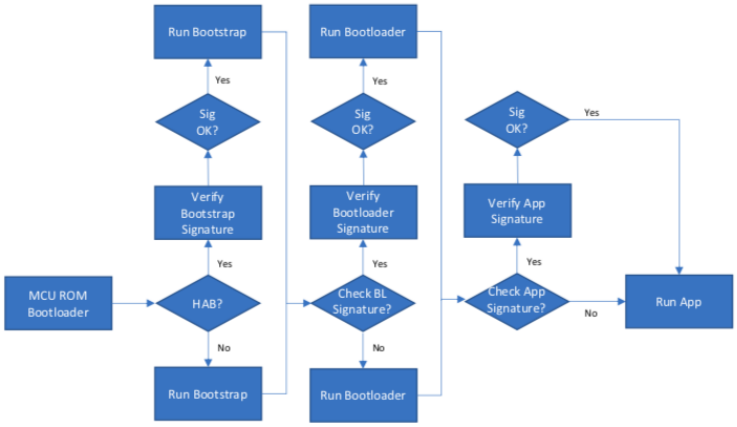


Figure 5: Boot Security Flow Chart

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-VIZN-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-VIZN-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 OTW bootloader interface.

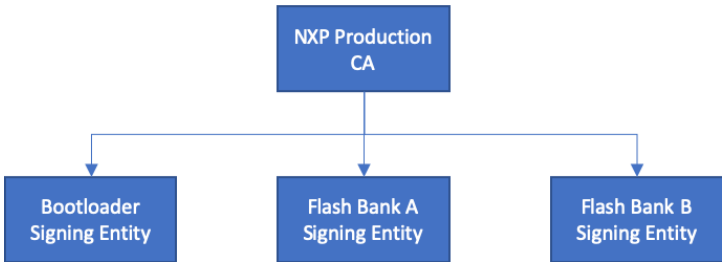


Figure 6: Signing Entities

2.4.2 Generate Application Bank B Binary

As previously mentioned, if the board is currently running from Application Bank A, MSD drag-and-drop flashing will require a binary created for Application Bank B and vice versa. Currently the active application flash bank information is not exposed to the user and can only be found through attempting to flash both a **Bank A** and **Bank B** application with MSD.

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_vizn_iot_userid_oobe application in in the **Project Explorer** panel and click on **Properties.**

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.**

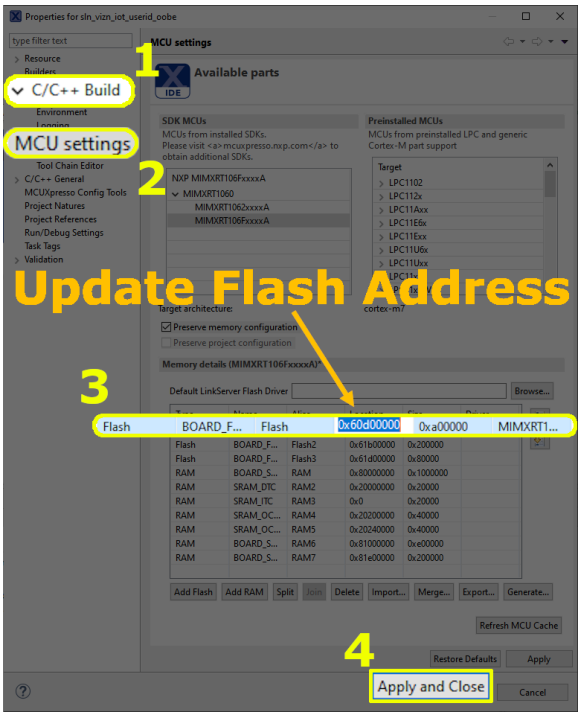


Figure 33: Create Flash Bank B Binary

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.

Be sure to change the flash address back to 0x60300000 if trying to run debugging using a J-Link.

Automated Manufacturing Tools can be used alongside your unique CA.

2.4.3 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 it being signed with 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 5:

- 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. **These bypasses should not be deployed in production.**

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

2.4.4 Image Certificate Authority (CA) and Application Certificates

The SLN-VIZN-IOT kit comes pre-programmed with signed images (though signature verification is bypassed by default) as indicated in the **Flash Image Configuration Area (FICA) and Image Verification** section. The **bootloader** and **userid_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 0x61D80000 for the bootloader

The certificate for the CA (used to verify the application signing certificates) is located at address 0x61CC0000 in the filesystem.

3 Get Started with MCUXpresso Tool Suite

The following section is going to describe the steps 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®

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**.

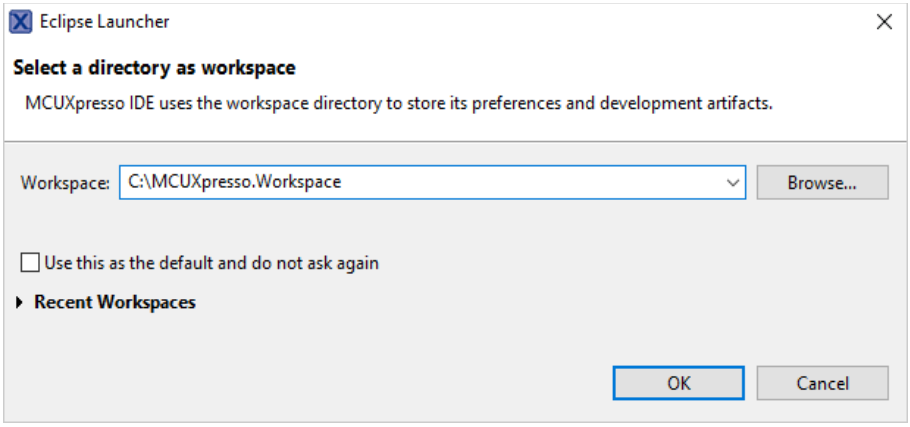


Figure 7: MCUXpresso IDE Workspace

3.2 Installing 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-VIZN-IOT SDK** example projects, the target SDK needs to be imported into MCUXpresso IDE.

The **MCUXpresso SDK** for the SLN-VIZN-IOT can be found in the **SLN-VIZN-IOT Software Collateral.zip** folder downloaded from the website using an SLN-VIZN-IOT collateral activation code obtained when purchasing a SLN-VIZN-IOT kit.

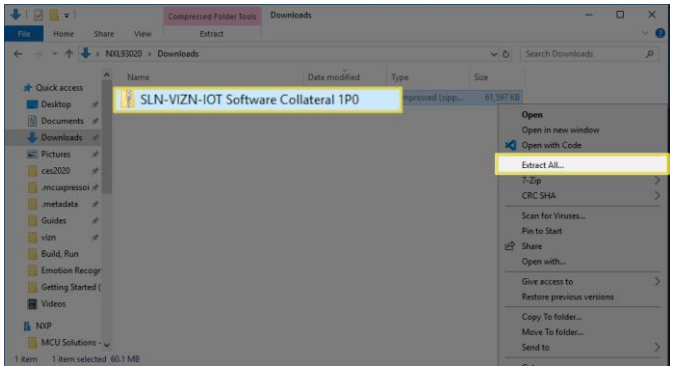


Figure 8: Extracting Software Collateral Zip

To import the SDK into MCUXpresso IDE, extract the zip folder and drag the **SLN-VIZN-IOT xPx SDK.zip** into the **Installed SDKs** window in MCUXpresso IDE.

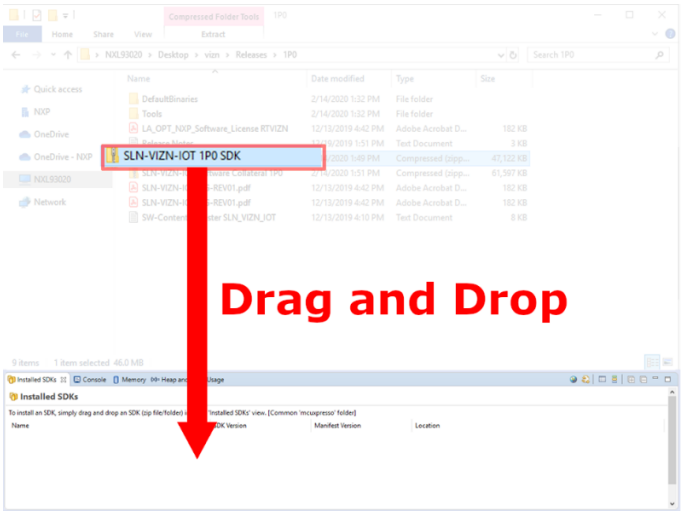


Figure 9: Drag and Drop SDK

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

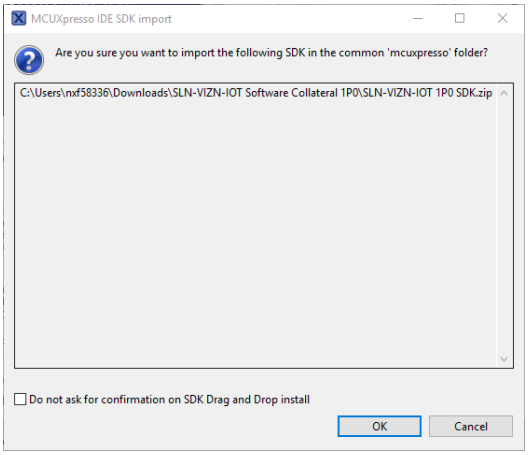


Figure 10: Import SDK Confirmation Window

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

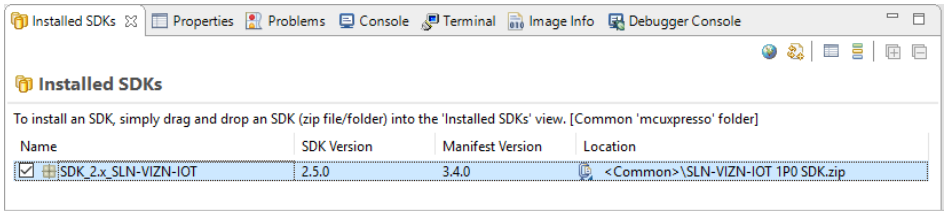


Figure 11: SDK Import Successful

3.3 Import a SLN-VIZN-IOT Project

The SLN-VIZN-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).**

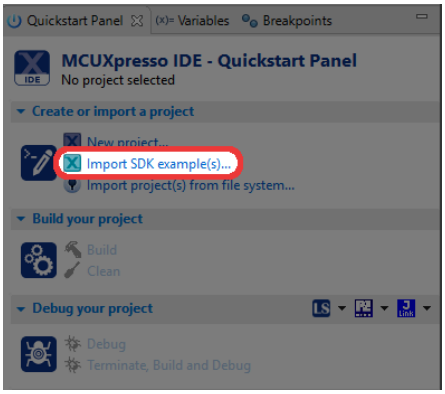


Figure 12: Import SDK Examples

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

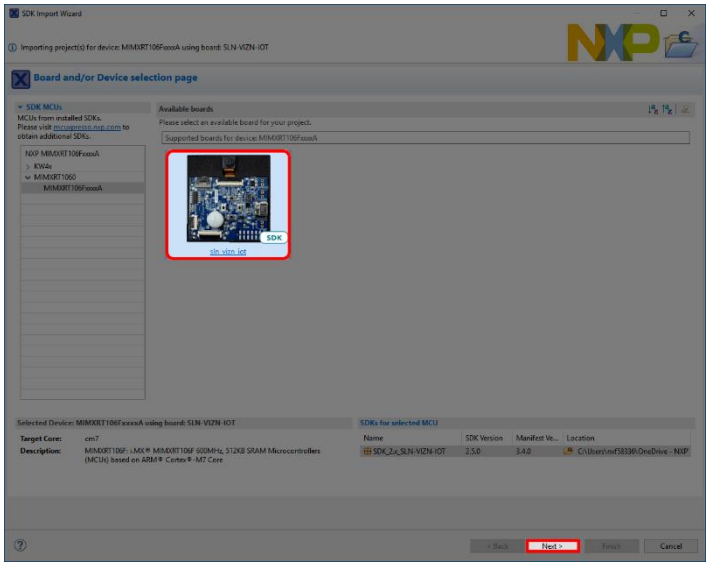


Figure 13: Import SLN_VIZN_IOT Examples

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_vizn_iot_userid_oobe** application. This is the application that comes flashed by default on your **SLN-VIZN-IOT** kit.

*If your kit's flash has been completely erased, you will also need the **sln_vizn_iot_bootloader** and **sln_vizn_iot_bootstrap** projects found under **sln_boot_apps** as well in order for the **sln_vizn_iot_userid_oobe** application to work.*

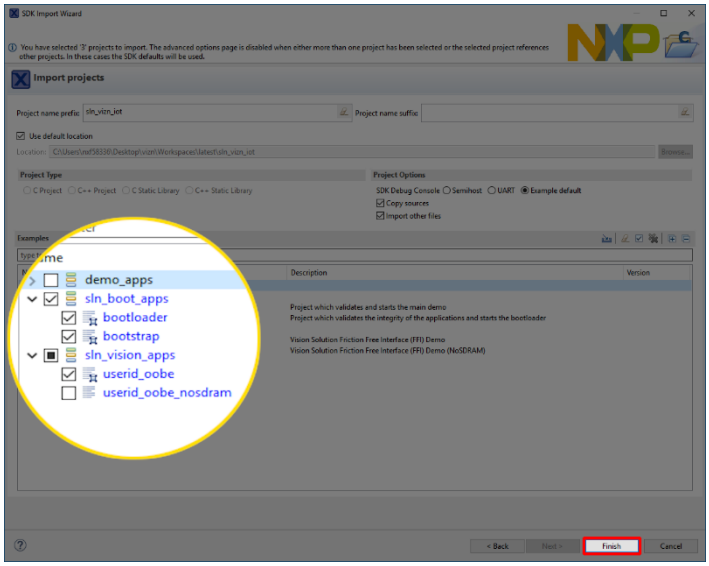
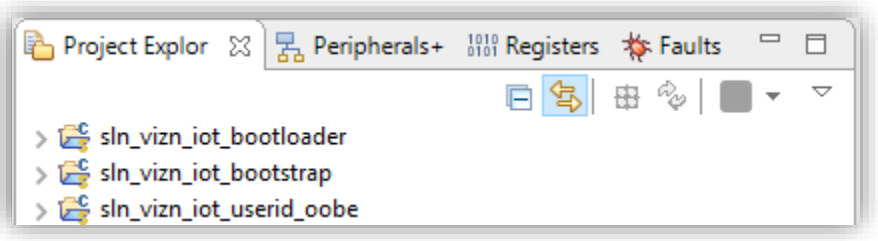


Figure 14: Import Examples

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



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

4.1 Build a SLN-VIZN-IOT Project

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

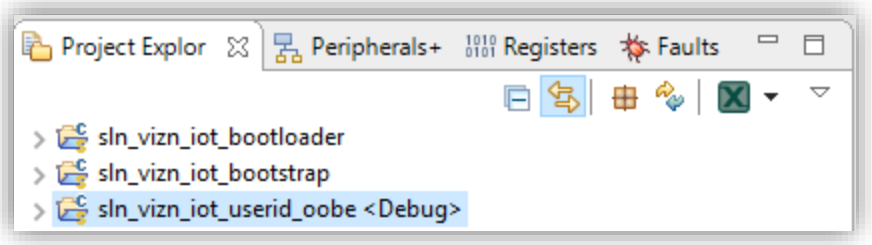


Figure 15: Project Explorer - Highlight Project

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

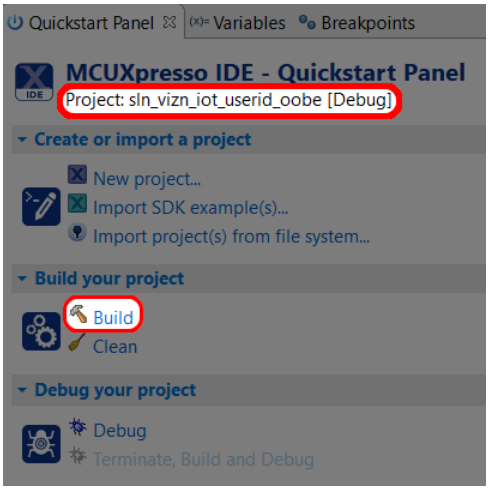


Figure 16: Build Project

Wait for MCUXpresso to finish the build process. This should take a relatively short time due to the small size of the application.

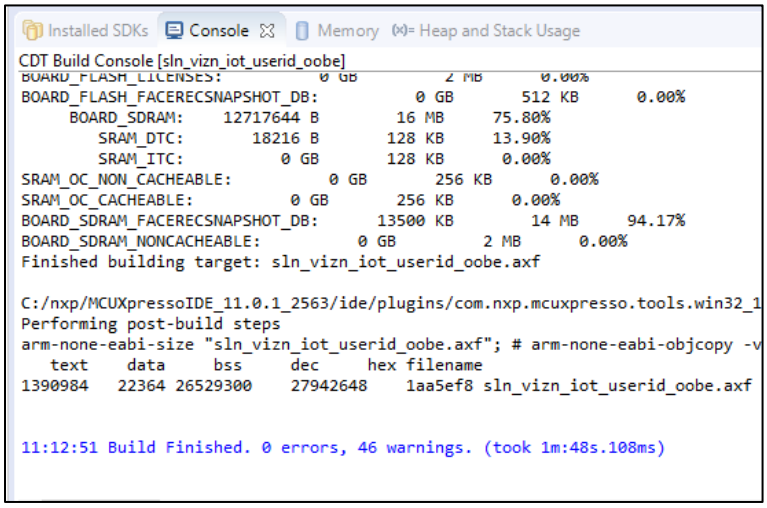


Figure 17: Console 'Build' Output

If you received a message like the one shown above, your SLN-VIZN-IOT 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:

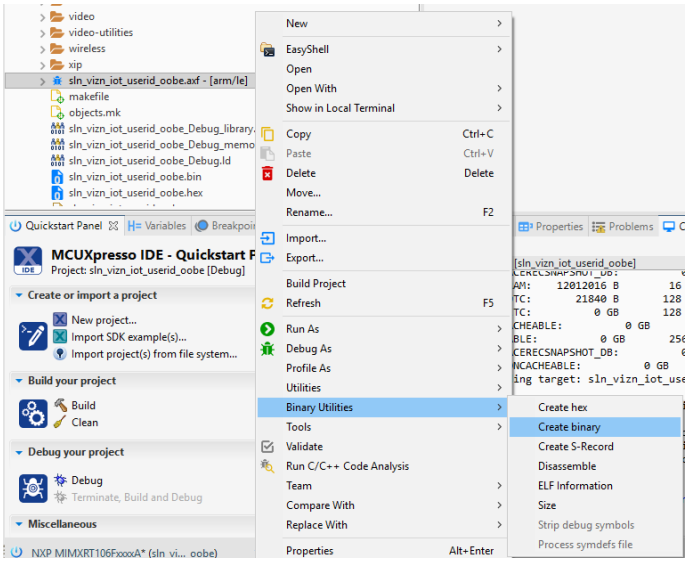


Figure 18: .axf to .bin

.bin files are useful for flashing with **OTW**, **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-VIZN-IOT Project

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

Flashing the SLN-VIZN-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 19: J-Link Plus and 9-Pin Cortex-M Adapter

Note: MCUXpresso IDE 11.1 currently comes with Segger J-Link V6.5x installed, however this WILL NOT work with the SLN-VIZN-IOT and must be upgraded to at least V6.62a. If you are unsure about which version of J-Link software you have, it is recommended to upgrade to the latest version just in case.

Older versions of J-Link Software and Documentation Pack will not have the proper configuration settings for the SLN-VIZN-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.

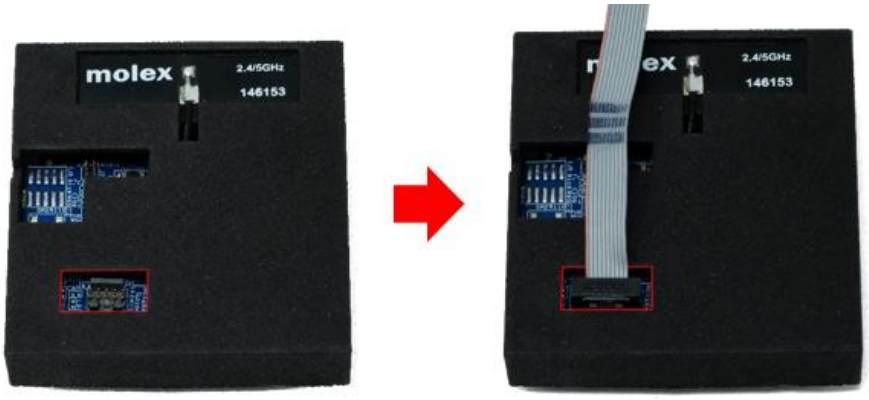


Figure 20: SLN-VIZN-IOT JTAG Header

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.

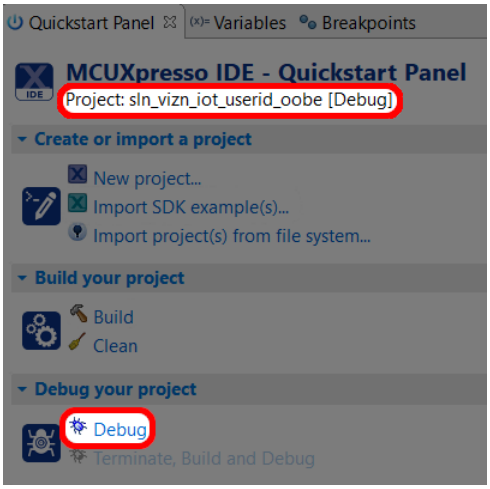


Figure 21: Quickstart Panel - Debug

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

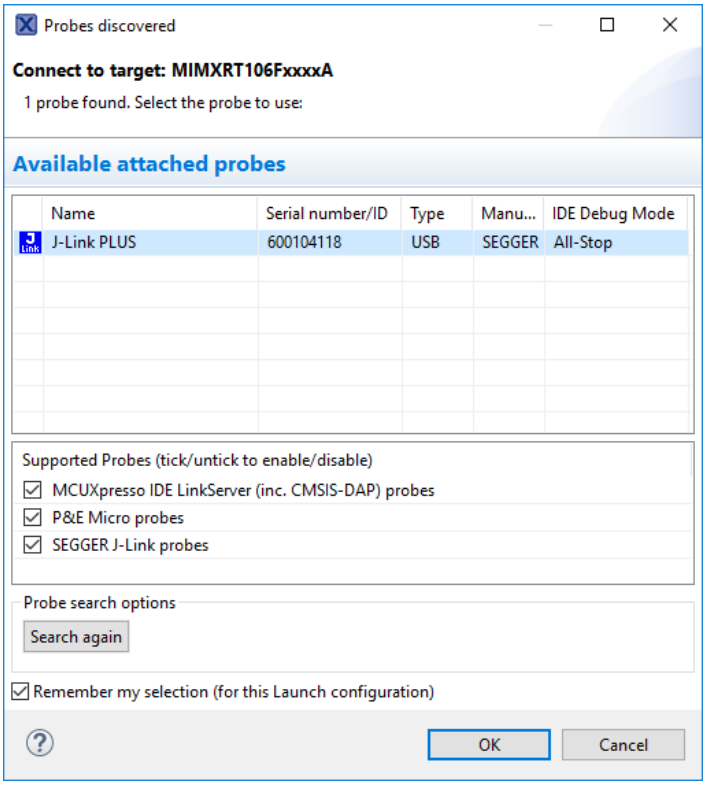


Figure 22: Probe Discovery Window

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

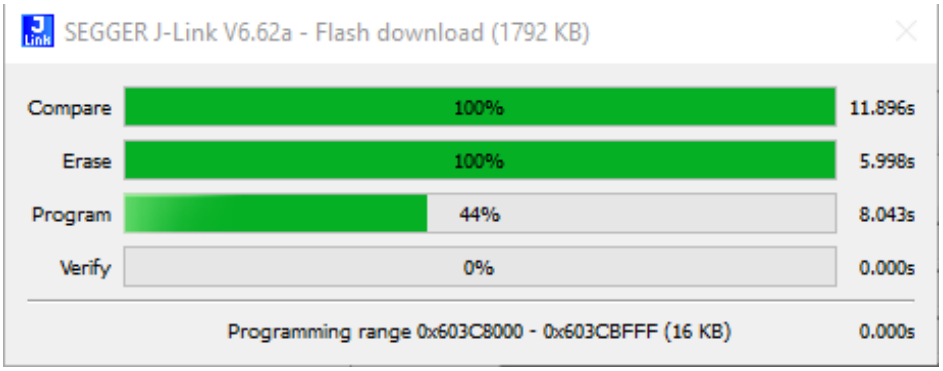


Figure 23: Flash Download in Progress

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

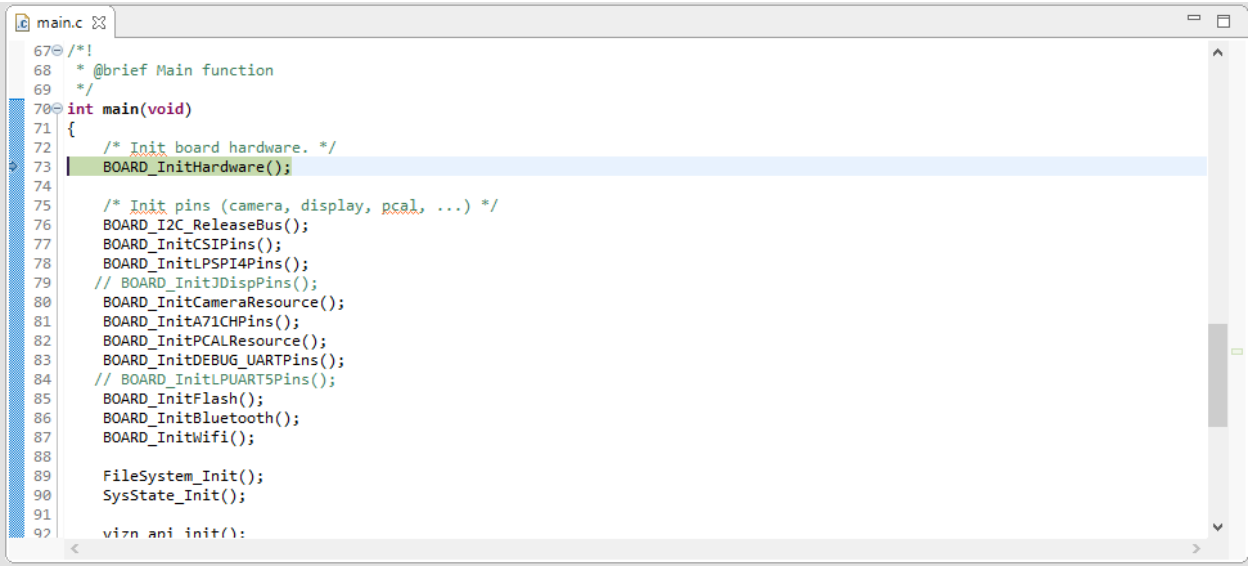


Figure 24: Debug Begin

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

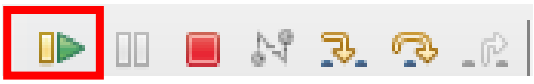


Figure 25: Debug Toolbar - Run Button

To learn more about debugging in MCUXpresso, check out the **MCUXpresso User Guide** found here:

https://www.nxp.com/docs/en/user-guide/MCUXpresso_IDE_User_Guide.pdf

5 Bootloader

The SDK provided enables two forms of firmware update capability in addition to flashing via J-Link: An **Over-the-Wire** (**OTW**) interface that supports UART 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 **OTW** and **USB MSD** modes.

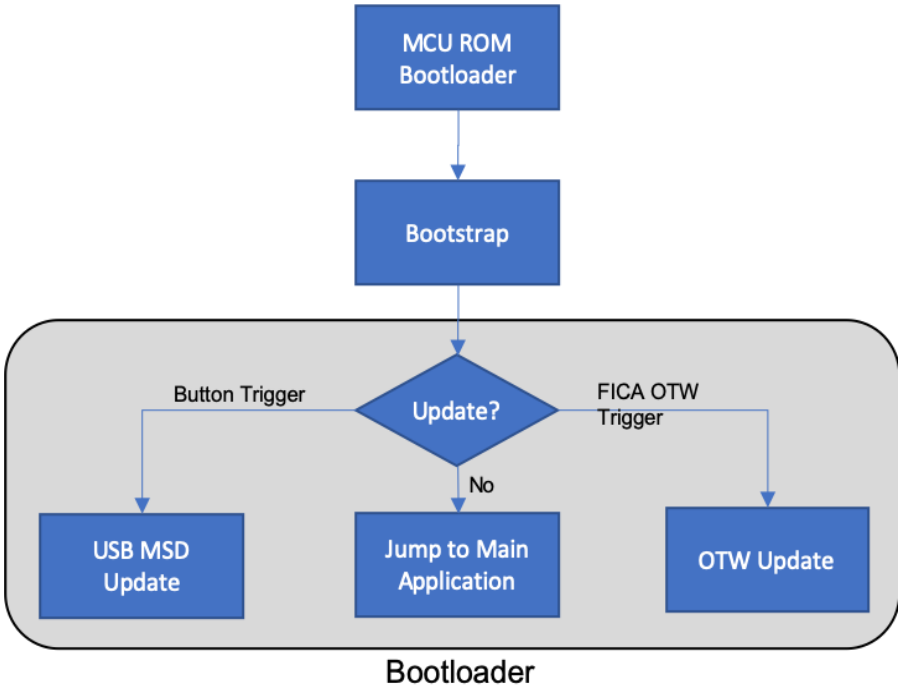


Figure 26: Bootloader Flow

5.2 Over-the-Wire (OTW) Updates

The OTW update interface currently supports UART but can be extended to support any serial interface including SPI, TCP sockets or even I2C. The OTW update is driven using a simple JSON interface, making it easy to implement host side code.

OTW must be triggered by setting a flag in the FICA area. In the SDK as delivered by NXP, this is accomplished in the demo application via the shell or eRPC host interface.

5.2.1 Transfers

Each transfer contains two pieces: a 4-byte size field and a JSON message. This allows the OTW data interface to be compatible across a wide range of interfaces.

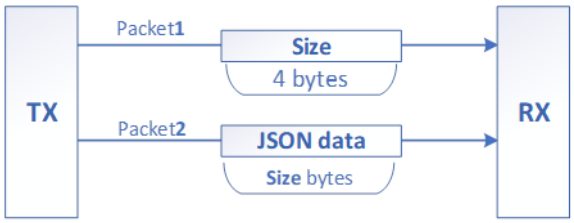


Figure 27: Transfer Format

Each transfer is followed by a transfer response.

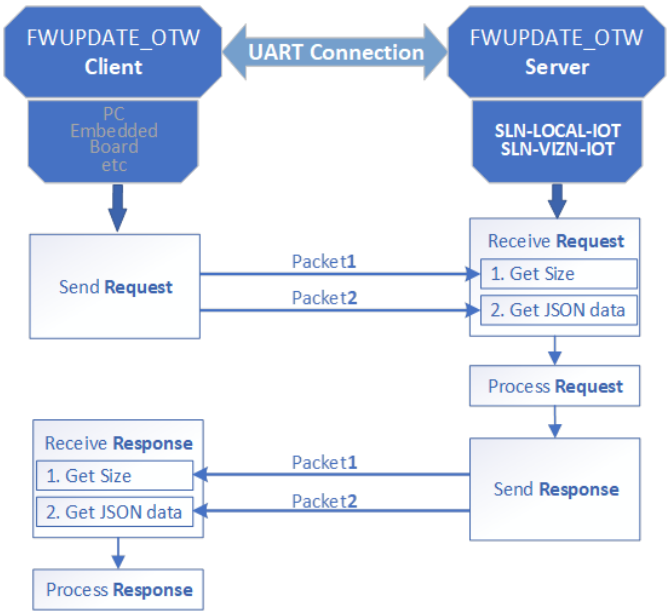


Figure 28: Request/Response Flow

5.2.2 JSON Messages

The OTW interface is driven entirely by JSON messages. This allows developers to easily create and debug client applications.

There are two types of messages passed: requests and responses. Requests must be made in the following order to successfully perform a firmware update:

1. Start
2. Block
3. Stop
4. Activate image
5. Start self-check

5.2.2.1 Start Request

This is the first request that must be sent to start a firmware update.

```
{
    "messageType":1,
    "fwupdate_message": {
        "messageType":0,
        "fwupdate_common_message": {
            "messageType":0,
            "job_id": <Job ID string>,
            "app_bank_type": <Flash Bank: ‘1’ for A ‘2’ for B>,
            "signature": <RSA Signature of image to be loaded>,
            "image_size": <Image Length>,
        }
    }
}
```

5.2.2.2 Block Request

Block requests are sent for each “chunk” of data to be programmed. Block sizes can be any size, though it’s suggested that they be as large as possible. The example script in the SDK sends 4800 bytes per block request.

```
{
    "messageType":1,
    "fwupdate_message": {
        "messageType":1,
        "fwupdate_server_message": {
            "messageType":0,
            "block": <Base64 encoded block of data>,
            "encoded_size": <Size of encoded block>,
            "block_size": <Size of block in bytes>,
            "offset": <Offset from base of flash>,
        }
    }
}
```

5.2.2.3 Stop Request

```
{
    "messageType":1,
    "fwupdate_message": {
        "messageType":1,
        "fwupdate_server_message": {
            "messageType":1
        }
    }
}
```

5.2.2.4 Activate Image Request

```
{
    "messageType":1,
    "fwupdate_message": {
        "messageType":1,
        "fwupdate_server_message": {
            "messageType":3
        }
    }
}
```

5.2.2.5 Start Self-Test Request

```
{
    "messageType":1,
    "fwupdate_message": {
        "messageType":1,
        "fwupdate_server_message": {
            "messageType":2
        }
    }
}
```

5.2.2.6 Response Format

```
{
    "error": <Operation return code>,
}
```

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.

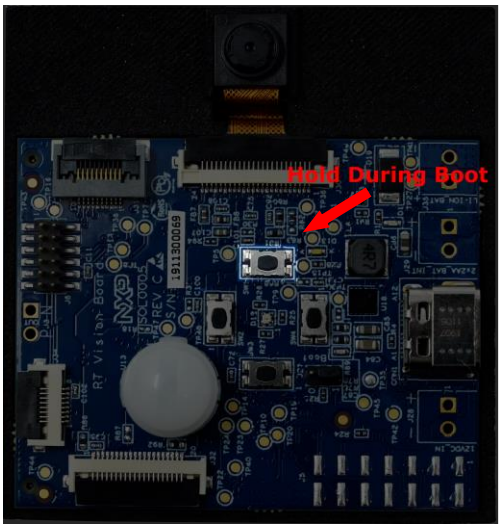


Figure 29: MSD Enablement Button

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. You may now let go of the button.

Figure 30: 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 31: MSD USB Drive Enumeration

To flash a new binary, drag and drop the new binary onto the USB Drive associated with the kit. If your SLN-VIZN-IOT kit is running from Application Bank A (see **Device Memory Map**), you must provide a binary for Application Bank B and vice versa. This is to prevent the overwrite of the application to be run in order to protect against a case where flashing is interrupted, and a corrupted image gets written and executed.

As there is currently no way to know which application bank is being run from, it is recommended to try dragging and dropping an application for each bank and seeing which one works.

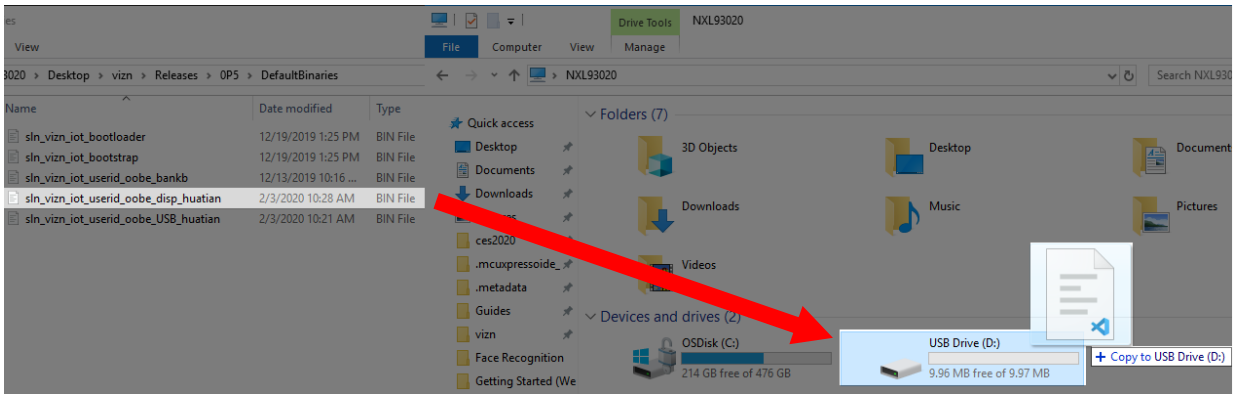


Figure 32: Dragging-and-Dropping New Binary

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

5.3.1 Generate Application Bank B Binary

As previously mentioned, if the board is currently running from Application Bank A, MSD drag-and-drop flashing will require a binary created for Application Bank B and vice versa. Currently the active application flash bank information is not exposed to the user and can only be found through attempting to flash both a **Bank A** and **Bank B** application with MSD.

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_vizn_iot_userid_oobe application in in the **Project Explorer** panel and click on **Properties.**

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.**

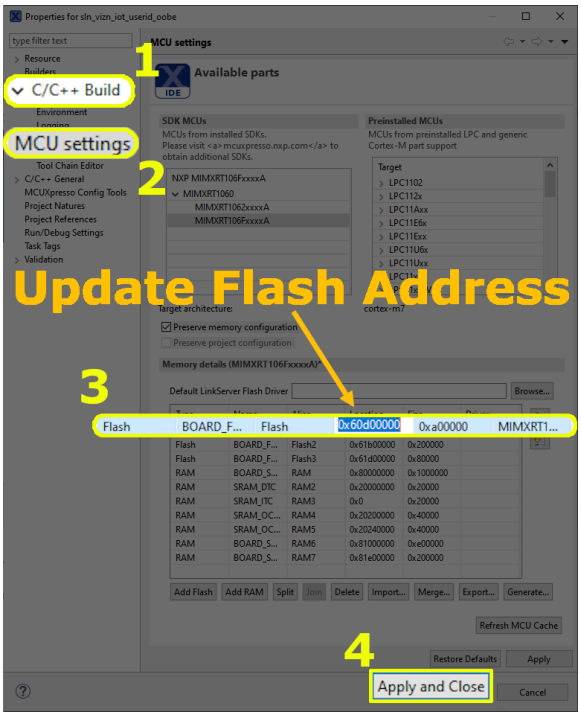


Figure 33: Create Flash Bank B Binary

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.

Be sure to change the flash address back to 0x60300000 if trying to run debugging using a J-Link.

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

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 are located in the "**Tools"** folder of the software package. 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-VIZN-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_vizn_iot/Scripts/sln_vizn_iot_open_boot $
```

Figure 34: 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_vizn_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 35: 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. **When prompted for passwords for the PEM files, use the same password for all of them for this exercise**. 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_vizn_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 36: 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 Open Boot Programming

```
(env) Ivaldi_sln_vizn_iot/Scripts/sln_vizn_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
Attemping to execute application...
execute
SUCCESS: Application running.
```

*Figure 37: **open_prog_full.py** Output*

6.5 (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 **userid_oobe** will be encrypted with individual encrypted context. The **bootloader** and **userid_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 containing README.md

6.5.1 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_vizn_iot/Scripts/sln_vizn_iot_open_boot $
```

Figure 38: "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.

6.5.2 Generating the PKI and Signed Flashloader

The following instructions assume that the section Error! Reference source not found. 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_vizn_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_vizn_iot/Scripts/sln_vizn_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 39: 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_vizn_iot/Scripts/sln_vizn_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 40: Checking the Signed Flashloader

6.5.3 Creating the Images

The following section describes how to generate the images and put them into the correct folder in preparation for creating the artifacts to load into the devices flash encrypted.

The next step is to ensure all the binaries are ready and located in the **DefaultBinaries** folder in the root directory of Ivaldi. To do this, ensure you have the bootstrap, bootloader and userid_oobe_demo imported into the MCUXpresso workspace as shown in the following figure.

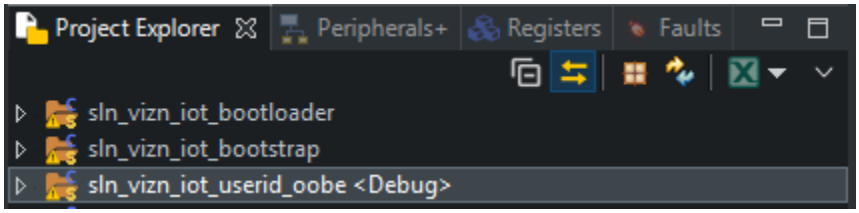


Figure 41: Importing the Applications for HAB and eXIP

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.

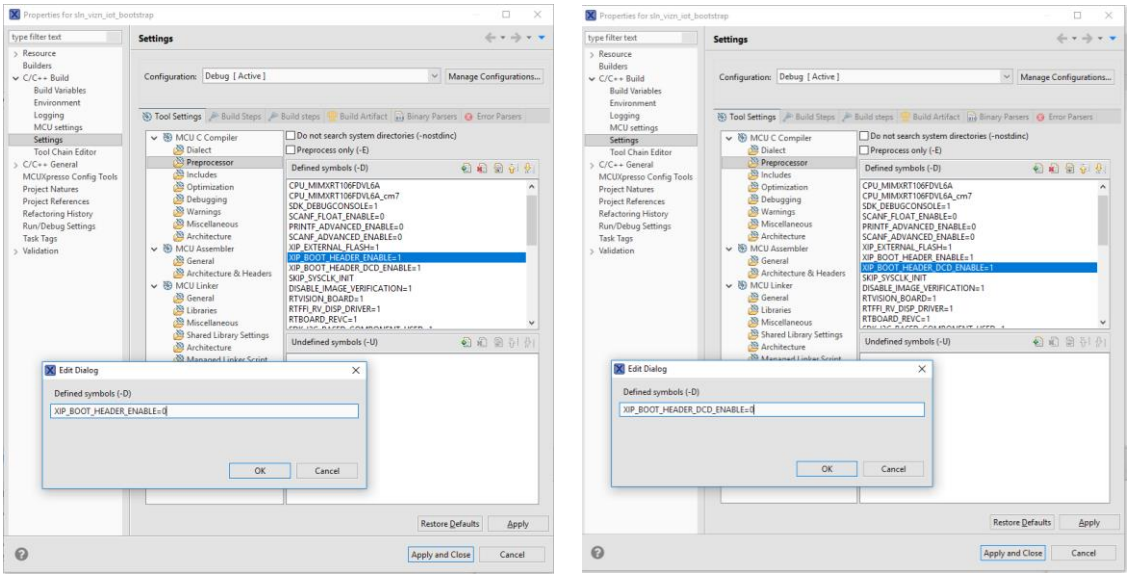


Figure 42: Unsetting the XIP Boot Header

After this change, hit the build button to generate an image for the bootstrap. As the Ivaldi scripts only accepts an srec file, it is necessary to generate one. Srec files can be generated using MCUXpresso's "binary utilities."

Navigate to the Debug folder of the bootstrap application, right click on the .axf file and navigate to **Binary Utilities -> Create S-Record.**

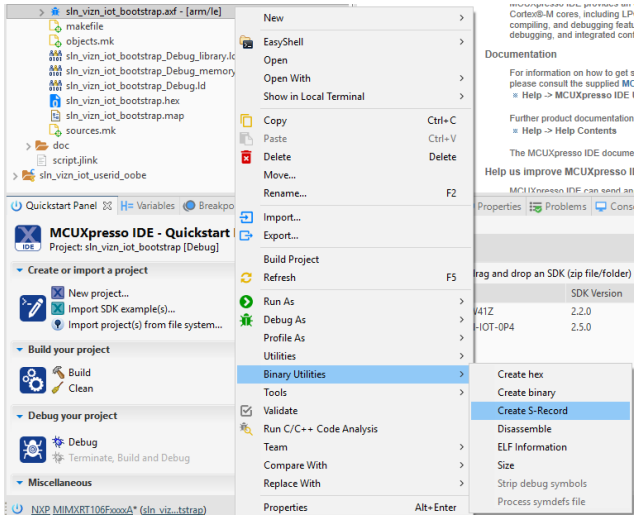


Figure 43: Generating the srec

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 like shown below.

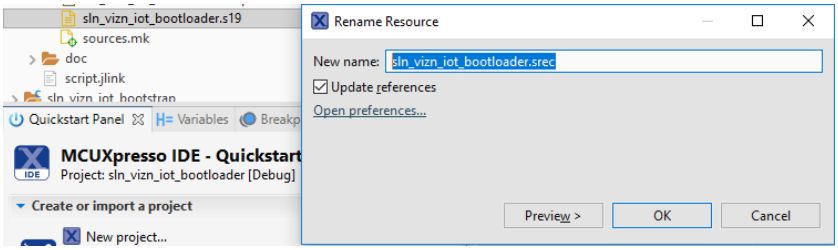


Figure 44: Changing File Type to srec

Continue to build the bootloader and userid_oobe_demo in the usual way. When these applications are built, it is required to generate binary files. Build these by navigating to the Debug folder in both the bootloader and userid_oobe_demo application, right click on the .axf file **Binary Utilities -> Create Binary**

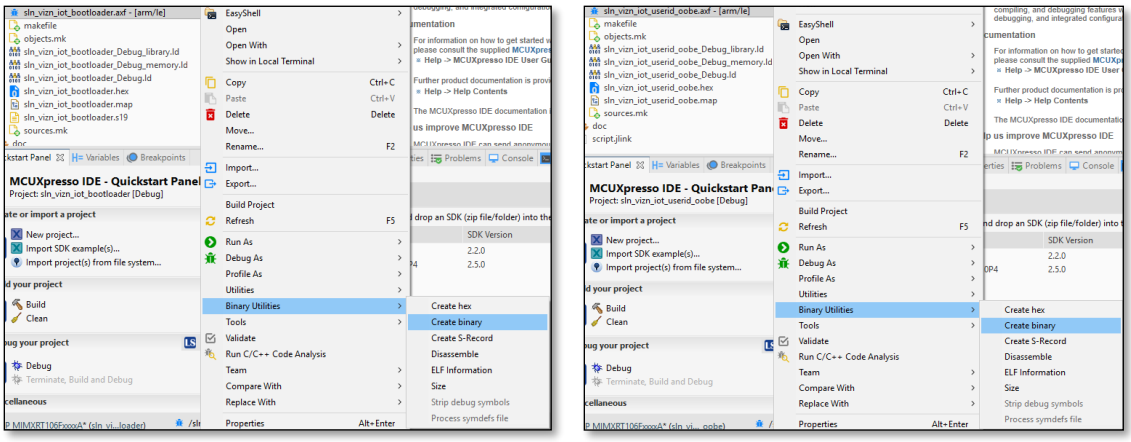


Figure 45: Generate Bootloader and userid_oobe Binary

Once the collateral has been created, copy the two binaries and srec into the **Ivaldi/Image_Binaries** package as shown here.

sln_vizn_iot_bootloader.bin	12/12/2019 1:54 PM	BIN File	350 KB
sln_vizn_iot_bootstrap.srec	12/19/2019 12:08 ...	SREC File	185 KB
sln_vizn_iot_userid_oobe.bin	12/18/2019 4:59 PM	BIN File	1,381 KB

Figure 46: "Image_Binaries" Expected Folder Contents

6.5.4 Generating Secure Binary

This section will describe the instructions on how to create a secure binary in preparation to programming it into the flash of the device.

Navigate to the **Scripts/sln_vizn_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_vizn_iot_bootstrap.srec"
DEF_BOOT_PATH  = IMG_DIR + "/sln_vizn_iot_bootloader.bin"
DEF_APP_PATH  = IMG_DIR + "/sln_vizn_iot_userid_oobe.bin"
```

Figure 47: Secure App File Names

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, run the script by executing: "**python3 secure_app.py**"

or to run without eXIP enabled: **"python3 secure_app.py --signed-only"**

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

```
(env) ivaldi_sln_vizn_iot/Scripts/sln_vizn_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 48: 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 the later section when programming the devices flash.

6.5.5 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, move the jumper J27, which is located on the top of the board into the "0" position.

The following instructions will show how to enable the HAB on the RT106F using the PKI infrastructure created in section Error! Reference source not found. as well as using the signed flashloader to implement it.

Navigate to the "**Scripts/sln_vizn_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.

```
(env) Ivaldi_sln_vizn_iot/Scripts/sln_vizn_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 49: 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.**

6.5.6 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 userid_oobe_demo.

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

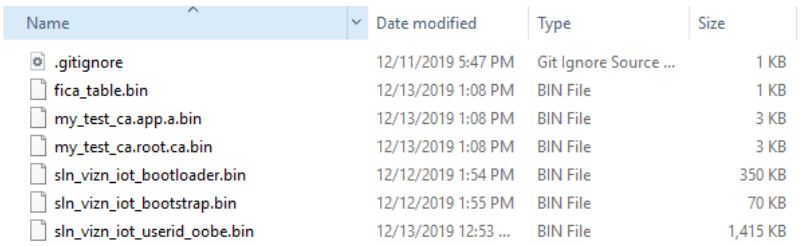


Figure 50: "Image_Binaries" Content

6.5.7 Enabling and Programming the Signed and Encrypted Binaries

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 safe comfort that their IP is protected, and physical attacks aren't 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.

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 userid_oobe_demo
    - Application image signing certificate
    - CA Image certificate
    - Device key and certificate

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

To enable this feature, navigate to the **Scripts/sln_vizn_iot_secure_boot/manf** folder in the Ivaldi root.

Run the following command "**python3 customer_prog_sec_app.py**" as shown below.

```
(env) ivaldi_sln_vizn_iot/Scripts/sln_vizn_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.

Unpower module, move the boot jumper in BOOT_MODE_1, and restore power
```

Figure 51: 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 userid_oobe_demo stored into flash.

If done correctly, the device will boot and run as described in the **UserID OoBE Demo** section.

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

7 Filesystem

The SLN-VIZN-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_vizn_iot_utils**.

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

Figure 52: 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
	SLN-VIZN-IOT User Guide
https://www.nxp.com/mcu-vision	SLN-VIZN-IOT Home Page
	SLN-VIZN-IOT Power Reference

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
2/11/20	Production 1.0	Complete revamp; split UG into UG + DG	Cooper Carnahan	NXP
19 December	Release 0.5	Added Manufacturing tools/security info	Cooper Carnahan	NXP
12 November	Release 0.4	Revision	Cooper Carnahan	NXP
30-September	Draft 0.1	Initial Draft	Cooper Carnahan	NXP

Table 5: Revision History

How to Reach Us:

Home Page:

www.nxp.com

Web Support:

www.nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells