NXP Semiconductors
User's Guide

Document identifier: SLN-LOCAL2-IOT-DG
Rev. 0, 19 April 2021

SLN-LOCAL2-IOT Developer’s Guide

Contents

Figures

Tables

Chapter 1
System Requirements and Prerequisites

The MCU Local Voice Control SDK requires an up-to-date computer which runs MCUXpresso IDE. It also requires a terminal program to communicate with the device via USB.

The MCUXpresso IDE is available here:

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

Table 1. Tested computer configurations

Computer type	OS version	Serial terminal application
PC	Windows 10	TeraTerm, PuTTY
Mac	macOS	Serial, CoolTerm, goSerial
PC	Linux	PuTTY

Below are listed development tools using MCU Local Voice Control SDK.

Table 2. Software tools and versions

Software tool	Version	Description
Segger	JLink_v6.98 or later	Tool to program the flash.
MCUXpresso IDE	Version 11.3.0	Eclipse base IDE for development environment

Chapter 2
Usage Conditions

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

- Frequency bands in which the equipment operates
- The maximum RF power transmitted

Table 3. Usage conditions

PN	RF technology	(a) Frequency range	(b) Max transmitted power
SLN-LOCAL2-IOT	WiFi	2412MHz – 2472MHz	17.9dBm

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

This apparatus, namely SLN-LOCAL2-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/

The product is expected to be used laying flat on a table, microphone output pointing up.

The data mode of the USB bus is not covered by the CE certification as this mode is used exceptionally to reprogram the device.

Chapter 3
Introduction

The NXP MCU Local Voice Control 2nd generation development kit (part number: **SLN-LOCAL2-IOT**) is a comprehensive, secure, and cost-optimized turnkey solution with a widely adopted development environment that enables customers to quickly get to market with a production-ready end-to-end software application.

SLN-LOCAL2-IOT embeds all the components required to produce a secure and *edge-computing* voice control product without a Wi-Fi or cloud connectivity. The architecture is built upon a single core:

- i.MX RT106S or RT105S for the main application, powered by an Arm® Cortex®-M7 core.

The SLN-LOCAL2-IOT hardware highlights are as follows:

- Up to 600 MHz (528 MHz by default) Cortex-M7 MCU core
- 1 MB on-chip RAM (512 KB TCM)
- 32 MB HyperFlash memory for Fast XiP (e**X**ecute **I**n **P**lace)
- Three PDM MEMS microphones
- TFA9894 Class-D amplifier
- Wi-Fi/Bluetooth combo chip
- Integrated speaker
- GPIO expansion headers

The SLN-LOCAL2-IOT software highlights are as follows:

- Two-stage bootstrap and bootloader, allowing for flexibility in customer implementation
- Secure boot flow with High Assurance Booting (HAB)
- Over-the-Air (OTA) update via WiFi
- Over-the-Wire (OTW) update via UART
- Automated manufacturing/reprogramming tools
- Speech recognition engine by deep learning
- Audio Front End (AFE) for far-field Automatic Speech Recognition (ASR)

SLN-LOCAL2-IOT is supported by a comprehensive and free-of-charge enablement suite from NXP and its partners, including the following:

- MCUXpresso development tools
- Hardware design files
- Local voice application software source code
- Software audio tuning tools
- Documentation
- Training material

3.1 Hardware overview

The SLN-LOCAL2-IOT kit is designed to provide a reference for a real product design. The board is designed using a small form factor and has many of the design considerations that hardware engineers evaluate. NXP also designed the hardware with some of the key hallmarks of a traditional development kit. Figure 1 and Figure 2 show the board components.

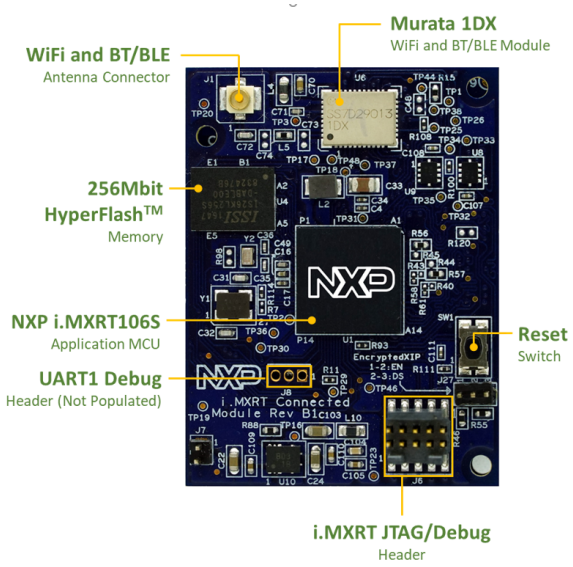


Figure 1. i.MX RT SOM (base board)

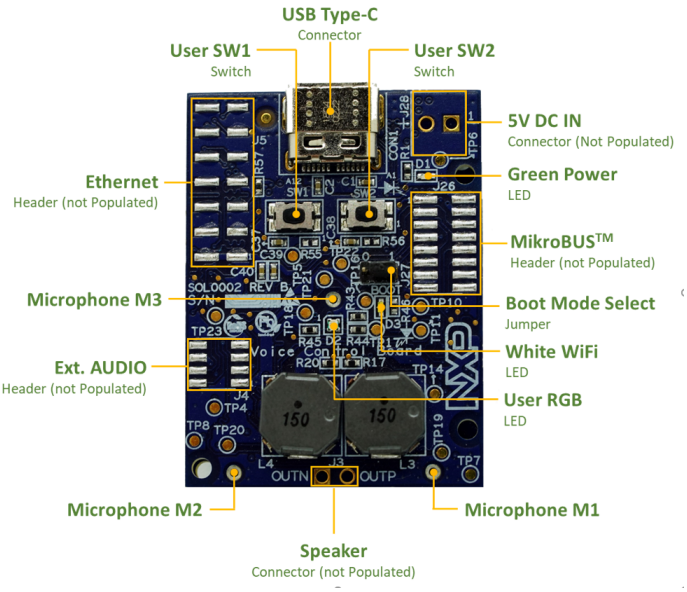


Figure 2. Voice shield (top board)

3.2 Software overview

Figure 3 shows a high-level software architecture diagram. This shows everything that is included in the SDK for the SLN-LOCAL2-IOT package, though not all of the features are implemented in demo applications.

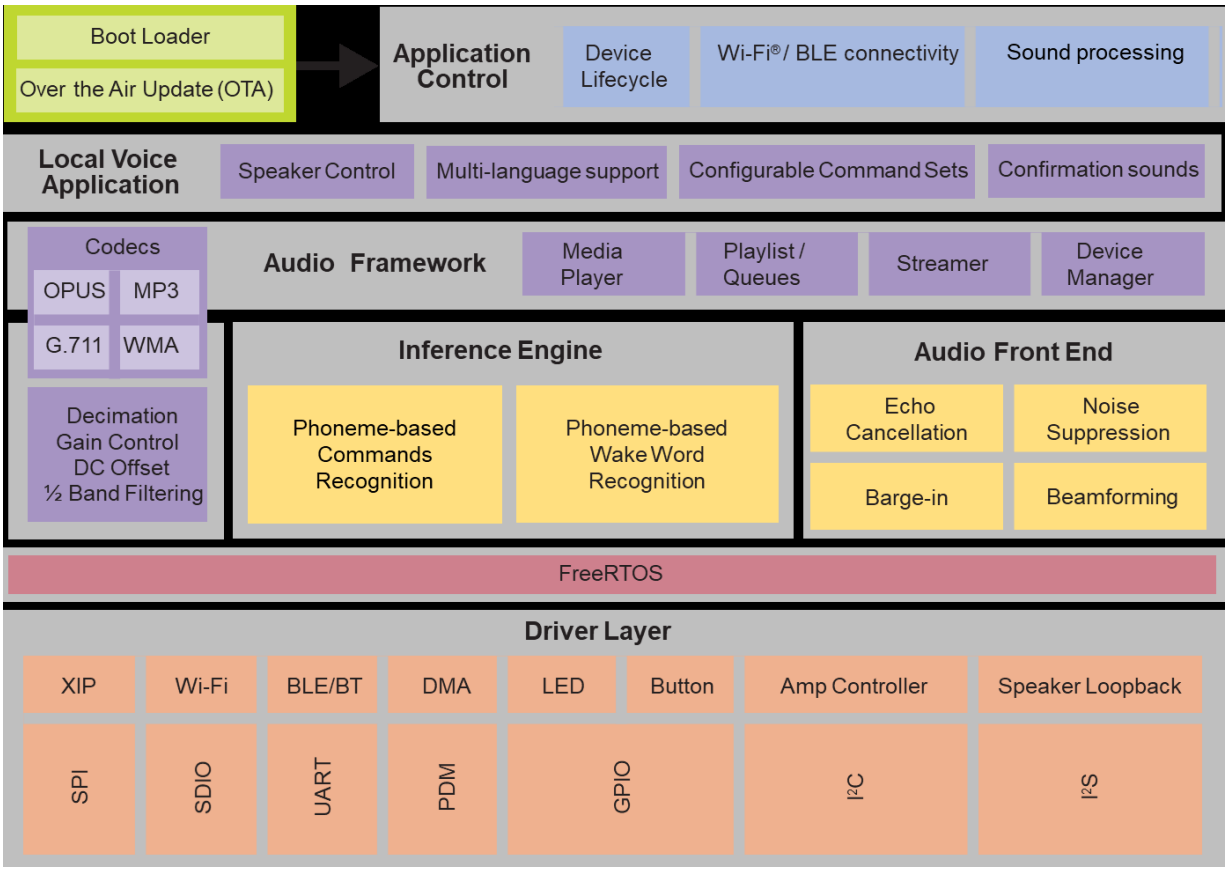


Figure 3. High-level software architecture

3.3 Device memory map

To understand the various pieces of the system, see the memory map (Table 4) that NXP developed for this application. There are many components required in the system to successfully boot and execute an application.

Table 4. Device memory map

Name	Start address	End address	Description
Boot Config Region	0x6000_0000	0x6000_0FFF	Used for XIP and setting up flash
IVT	0x6000_1000	0x6000_1FFF	Vector table
Bootstrap	0x6000_2000	0x6003_FFFF	
Bootloader	0x6004_0000	0x601F_FFFF	
Not Used	0x6020_0000	0x602F_FFFF	Not used
Application Bank A	0x6030_0000	0x60CF_FFFF	
Application Bank B	0x60D0_0000	0x616F_FFFF	

Table continues on the next page...

Table 4. Device memory map (continued)

Name	Start address	End address	Description
Filesystem	0x6170_0000	0x61F7_FFFF	Stores device settings and certificates listed in Table 5
Reserved	0x61F8_0000	0x61FB_FFFF	Reserved
Flash Image Config Area	0x61FC_0000	0x61FF_FFFF	FICA table is used for secure boot

3.4 Flash memory filesystem

The filesystem manages various entries of device settings and certificates stored in the flash memory. The sector size of the HyperFlash filesystem is 256 KB. Each file must be saved in one sector.

The contents are listed in Table 5. It shows all the files and their purposes in the SLN-LOCAL2-IOT kit. These files are programmed by default when receiving the kit.

Table 5. Full list of files in HyperFlash filesystem

Name	Start address	End address	Description
Audio Playback EN 01	0x6178_0000	0x617B_FFFF	“OK” sound in English
Audio Playback EN 02	0x617C_0000	0x617F_FFFF	“Can I help you?” sound in English
Audio Playback ZH 01	0x6180_0000	0x6183_FFFF	“OK” sound in Chinese
Audio Playback ZH 02	0x6184_0000	0x6187_FFFF	“Can I help you?” sound in Chinese
Audio Playback DE 01	0x6188_0000	0x618B_FFFF	“OK” sound in German
Audio Playback DE 02	0x618C_0000	0x618F_FFFF	“Can I help you?” sound in German
Audio Playback FR 01	0x6190_0000	0x6193_FFFF	“OK” sound in French
Audio Playback FR 02	0x6194_0000	0x6197_FFFF	“Can I help you?” sound in French
Audio Playback EN 03	0x6198_0000	0x619B_FFFF	“Say the temperature to be set” sound in English
Audio Playback EN 04	0x619C_0000	0x619F_FFFF	“Say the time to be set” sound in English
Audio Playback EN 05	0x61A0_0000	0x61A3_FFFF	“Temperature has been set” sound in English
Audio Playback EN 06	0x61A4_0000	0x61A7_FFFF	“Timer has been set” sound in English
ASR Control Configuration	0x61A8_0000	0x61AB_FFFF	Saved parameters of ASR control shell commands
WiFi Credential	0x61BC_0000	0x61BF_FFFF	WiFi SSID and password
Image CA Root Certificate	0x61CC_0000	0x61CF_FFFF	Can be used for connection to IoT Cloud

Table continues on the next page...

Table 5. Full list of files in HyperFlash filesystem (continued)

Name	Start address	End address	Description
Bank A Signing Certificate	0x61D0_0000	0x61D3_FFFF	Used for validating the signature of Bank A
Bank B Signing Certificate	0x61D4_0000	0x61D7_FFFF	Used for validating the signature of Bank B which is written during first OTA
Bootloader Signing Certificate	0x61D8_0000	0x61DB_FFFF	Use for validating the signature of bootloader

3.5 Audio application architecture

The audio capture application is implemented as a pipeline shown in Figure 4. The application can be configured to capture either two or three microphones (two by default). Because the i.MX RT106S device has enough processing power, the core is used to process the entire chain from the PDM decimation to the Automatic Speech Recognition (ASR) engine. Every 10 milliseconds, the DMA moves raw PDM data from each microphone. This data is fed into the NXP Solutions PDM decimation software IP to convert the audio into 16-bit, 16-kHz PCM data. When it comes out of the decimation block, it is fed to the Audio Front End (AFE) to perform beamforming and acoustic echo cancellation. At this point, it is a single 16-bit, 16-kHz mono audio signal.

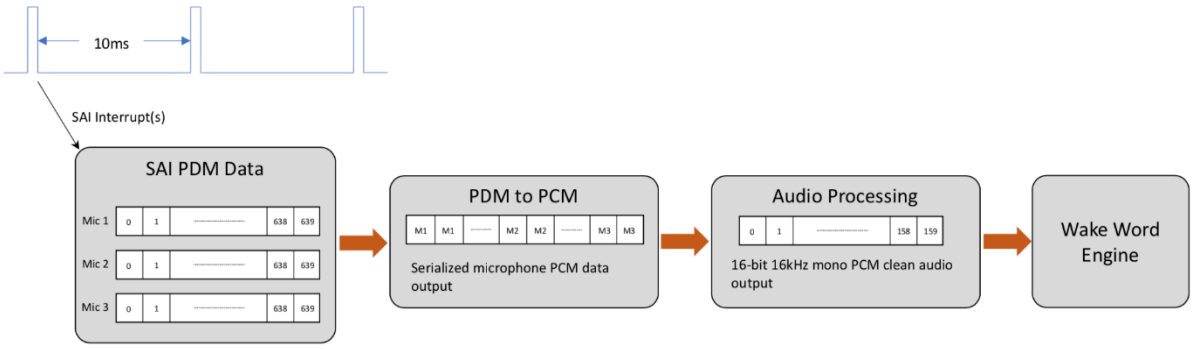


Figure 4. Audio application pipeline

Although the ASR works on multiples of 10 ms of audio data, a 30-ms data block is recommended for the input. Thus, the *audio_processing_task* accumulates 30 ms worth of processed audio before sending it to the ASR for processing.

3.6 ASR application

The AFE output signal is transferred to the ASR, where the wake word engine waits for a wake word. If a wake word is detected, the same language's command engine is loaded to process the voice control commands. Developers can also implement multiple groups of commands sequentially to create a dialog-style voice-control application.

NXP implemented the following three types of baseline demos:

- LED voice control demo
  - English
  - Two-stage (wake word and command) ASR
- IoT/elevator/audio/washing machine voice control demo
  - Selectable combinations of English, Chinese, German, and French
  - Two-stage ASR
- Oven voice control demo

— English

— Multiturn (4-way) dialog-style ASR

The ASR implemented with the selected languages can be easily replaced with other languages. NXP provides an application note to customize the local voice demos. Contact NXP (local-commands@nxp.com) for more information about the process of the phoneme-based speech recognition engine generation and custom wake words and commands.

3.7 User interfaces

The SLN-LOCAL2-IOT kit's functional features can be configured using a serial terminal interface. Figure 5 shows the shell prompt in the user's serial terminal window. The connection is made via the USB CDC.

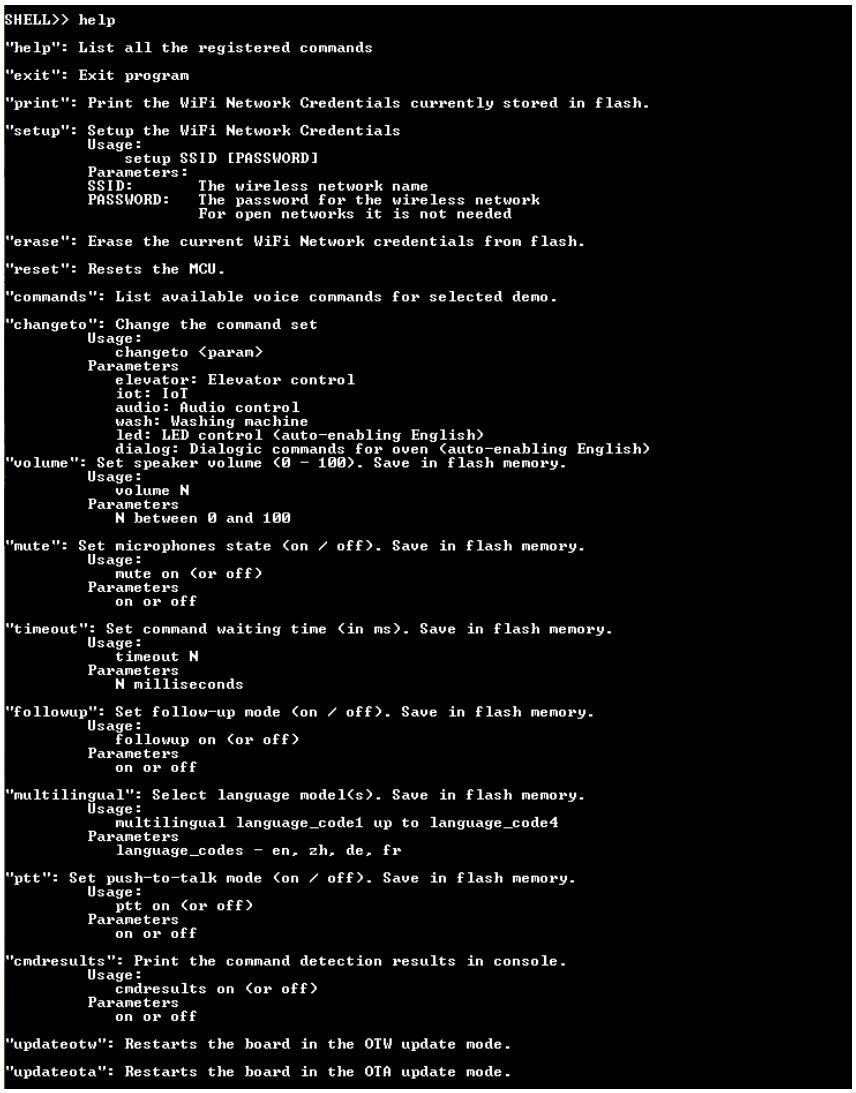


Figure 5. Shell prompt interface

The LED indicates various conditions of the SLN-LOCAL2-IOT kit. The LED is located on the kit, as shown in Figure 6. If the kit boots without any problems, the LED lits with green color while booting and then it turns off. When the kit detects a wake word, the LED lits with blue color while listening to a voice command. If a command is detected, the LED flickers with green color for 0.2 s. If no command is uttered or detected, the LED flickers with purple color for 0.2 s. Table 6 summarizes the LED colors and status description.

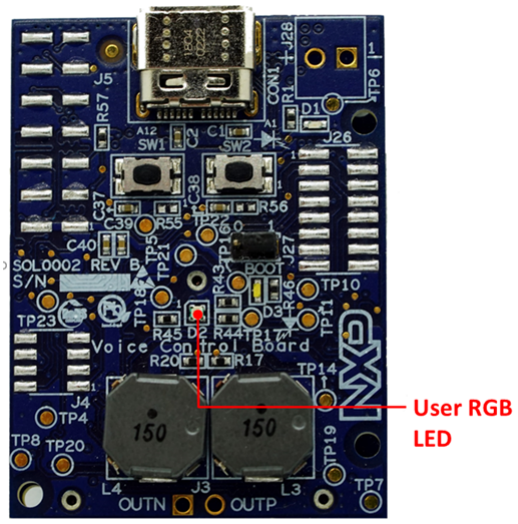


Figure 6. User LED on the SLN-LOCAL2-IOT kit

Table 6. Summary of LED color and behavior

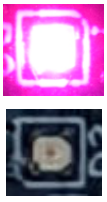
Function	LED State (D2)	Color	Description
Boot up	Solid Green 2 seconds		The device has powered on and is going through initialization
Wake word detected	Solid Blue		The device detected the wake word and listens to a command.
Command detected	Green blink 200ms		The device detected a command.
Timeout	Purple blink 200ms		If no command is detected within a certain time, the device stops listening to a command.
Microphone off	Solid Orange		Microphones are turned off.

Table continues on the next page...

Table 6. Summary of LED color and behavior (continued)

Function	LED State (D2)	Color	Description
Push-to-Talk (PTT) mode	Solid Cyan		The device is on PTT mode. By pressing SW1, wake word detection phase is bypassed and the device listens to a command.
Initialization Failed	Solid Red		The device failed to initialize AFE or ASR.
Audio stream error	Solid Purple		Audio stream after AFE is not transferred to ASR.
ASR memory error	Solid Orange		During initialization or language or demo change, an error occurred in verifying memory pool size.

There are two on-board buttons that can be used for input interfaces. SW1 is used for the PTT mode. SW2 is used for the MSD mode.

The following interfaces are available:

- Output interface
  - Serial terminal via USB CDC by default
  - UART
  - LED indicator
  - On-board speaker
- Input interface
  - Serial terminal via USB CDC – shell commands
  - Two on-board buttons

3.8 Security architecture

The SLN-LOCAL2-IOT kit is built and designed in a way that enables the best security practices, while maintaining the development kit feel. The main security mechanisms implemented are the **image verification** stages that are required for every image programmed into the device. By default, the image verification is enabled in the SLN-LOCAL2-IOT kit. For more details about the security architecture, see Security architecture.

3.9 Automated manufacturing tools

NXP provides a package of scripts that can be used for manufacturing programming and reprogramming of devices on the production line without the J-Link. This collection of scripts is called Ivaldi. The Ivaldi package allows developers to program all the required firmware binaries into a flash device using a single command. To learn more about the automated manufacturing tools, see Chapter 11.

Chapter 4
Getting started with MCUXpresso Tool Suite

4.1 MCUXpresso IDE

The MCUXpresso IDE has an easy-to-use Eclipse-based development environment for NXP MCUs based on the 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 all NXP evaluation boards with industry-leading open-source and commercial debug probes from Arm, P&E Micro®, and SEGGER®.

To download NXP MCUXpresso IDE, visit www.nxp.com/MCUXpresso (free of charge).

Launch the MCUXpresso IDE and define the workspace location to copy and store your projects and click **Launch**. Figure 7 shows an example of workspace configuration.

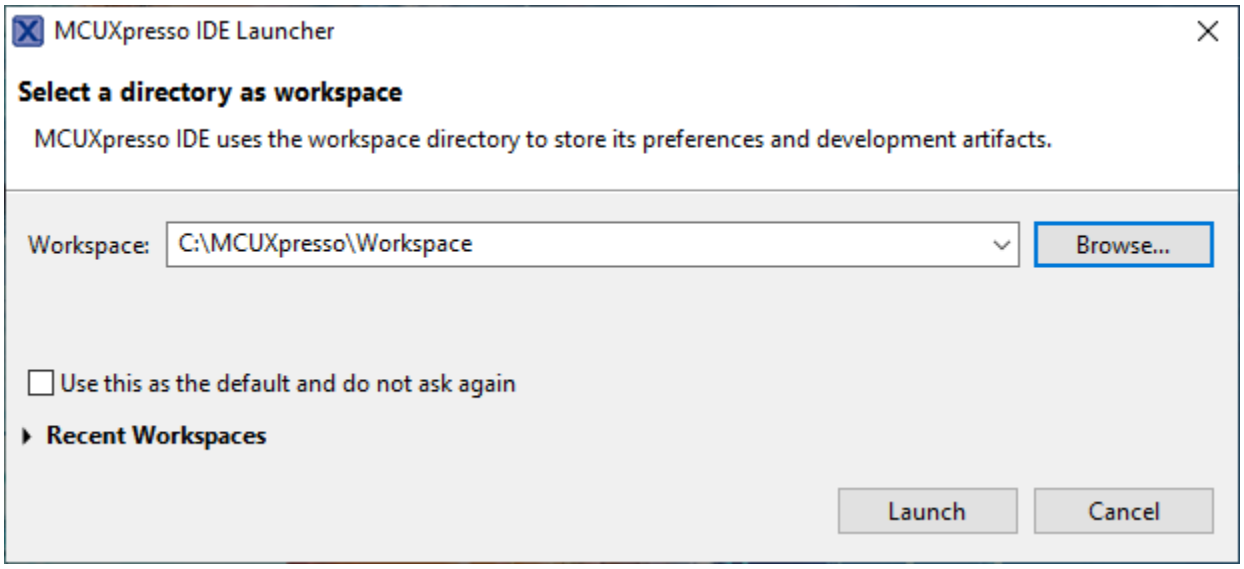


Figure 7. Creating MCUXpresso IDE workspace

4.2 Software Development Kit (SDK)

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

4.2.1 Downloading SDK

The SLN-LOCAL2-IOT SDK is distributed through the MCUXpresso SDK Builder, which is a web tool providing access to SDKs for NXP board platforms. This section describes where to locate, generate, and download the SDK before installing it.

Navigate to the MCUXpresso SDK Builder which should open the SLN-LOCAL2-IOT kit which allows you to build the SDK.

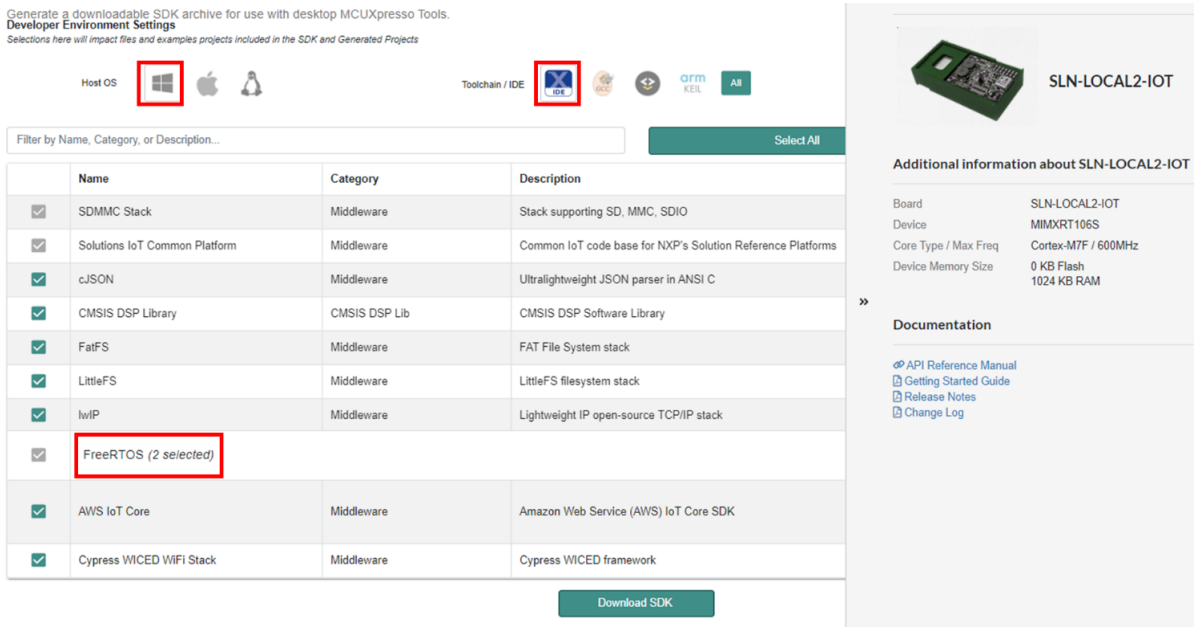


Figure 8. MCUXpresso SDK build for SLN-LOCAL2-IOT

NOTE

You may be asked to log into the NXP webpage to access the MCUXpresso SDK builder.

In Figure 8, there are three red boxes. Select the toolchain support, the host OS you are developing on, and the embedded real-time operating system. Click the "**Select All**" button to include all the relevant software packages and then click the "**Download SDK**" button.

4.2.2 Import SLN-LOCAL2-IOT SDK

Before building the SLN-LOCAL2-IOT SDK example projects, the target SDK must be imported into the MCUXpresso IDE by dragging and dropping the target SDK archive into the "Installed SDKs" window in the MCUXpresso IDE. Figure 9 shows the pop-up window which asks for confirmation (Click **"OK"**).

When the package is imported, it will be displayed in the list of installed SDKs. Figure 10 shows the installed SDKs in the MCUXpresso IDE.

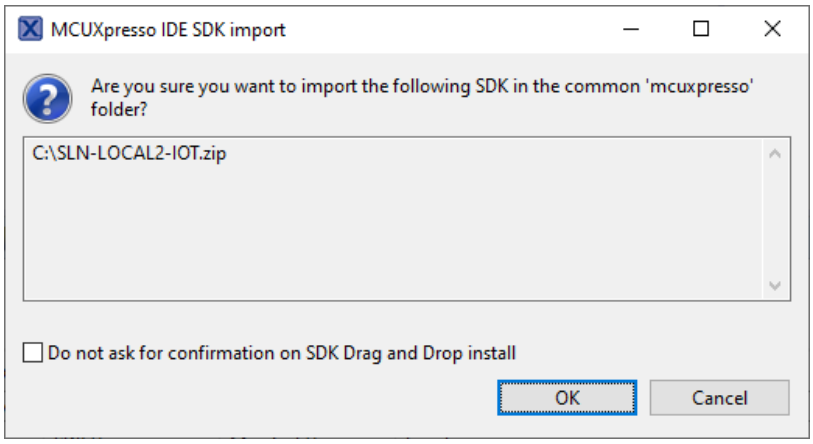


Figure 9. MCUXpresso SDK import confirmation window

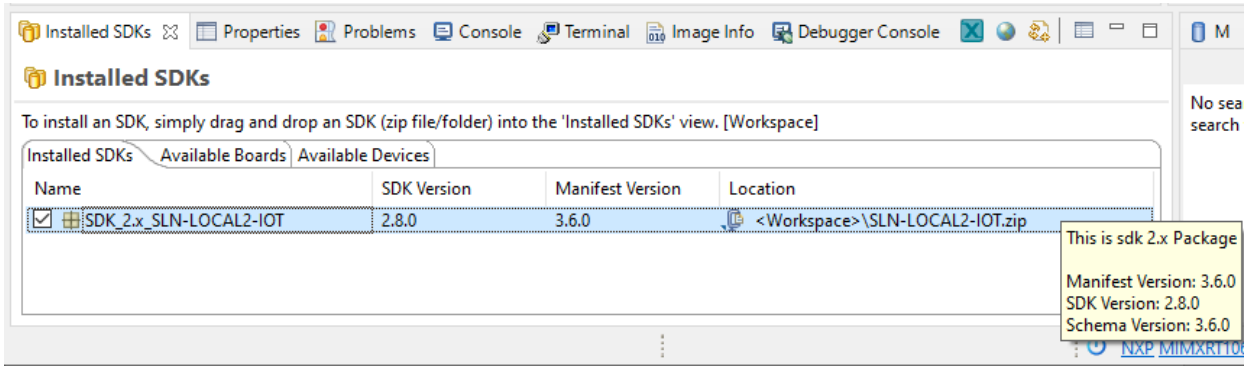


Figure 10. SLN-LOCAL2-IOT SDK installation in MCUXpresso IDE

4.2.3 Importing SLN-LOCAL2-IOT projects

The SLN-LOCAL2-IOT SDK allows you to import existing application examples as a development starting point. Some applications are intended to handle most of the voice aspects of the functionality, allowing developers to focus on the product innovation.

The following steps show how to import SLN-LOCAL2-IOT projects into MCUXpresso IDE.

From the **Quickstart Panel**, select **Import SDK examples(s)**, as shown in Figure 11.

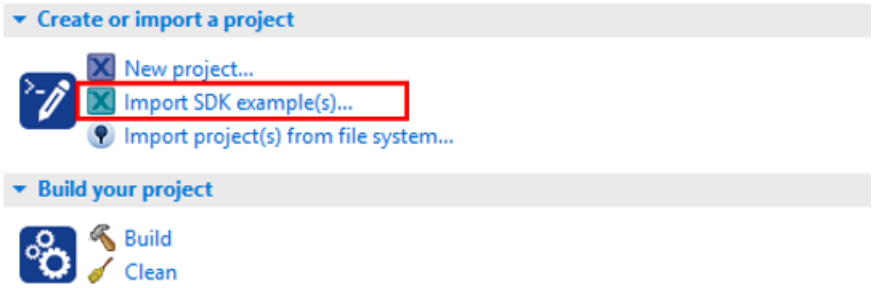


Figure 11. MCUXpresso Quickstart Panel Import SDK Example(s)

A list of all the installed board SDKs that have examples to import from appears. Select the “**sln_local2_iot**” image and then click the "**Next**" button, as shown in Figure 12.

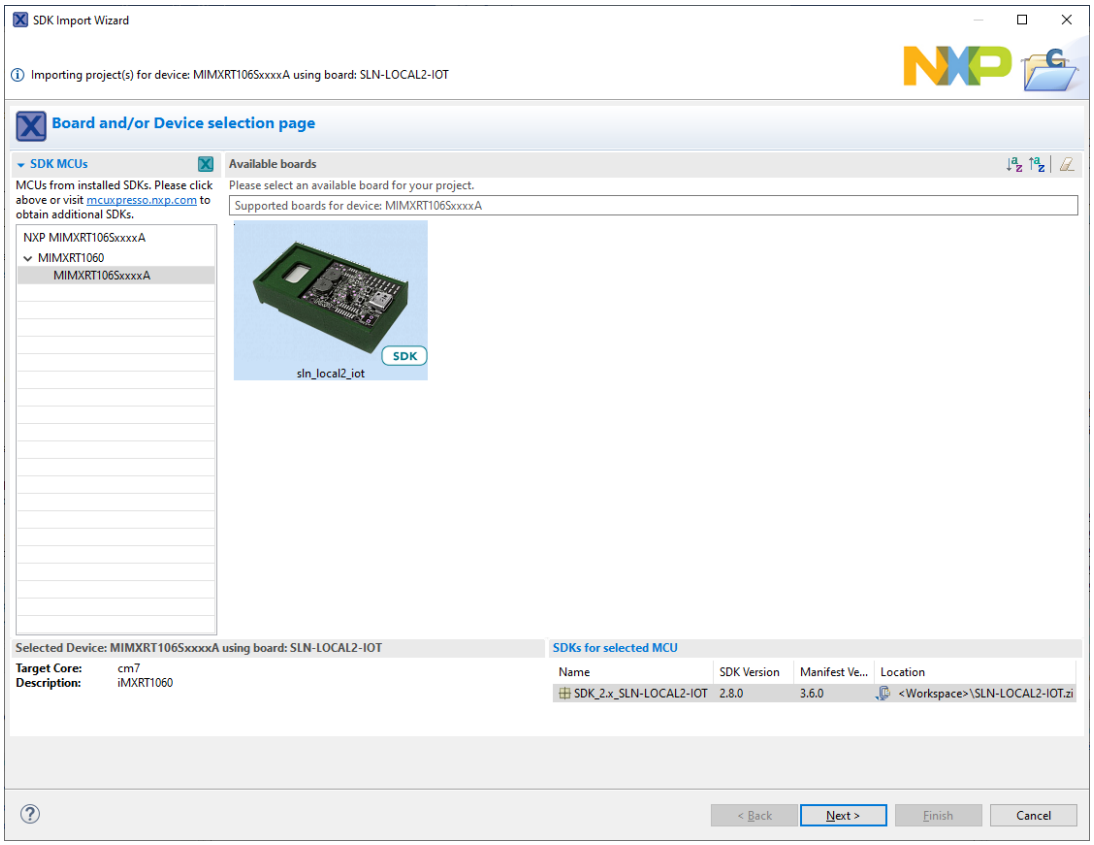


Figure 12. MCUXpresso SDK selection

The import wizard then displays all the applications that are available to import. Ensure that the SDK Debug Console is not moved from its default position. Figure 13 shows the import of all the projects to be used in this section.

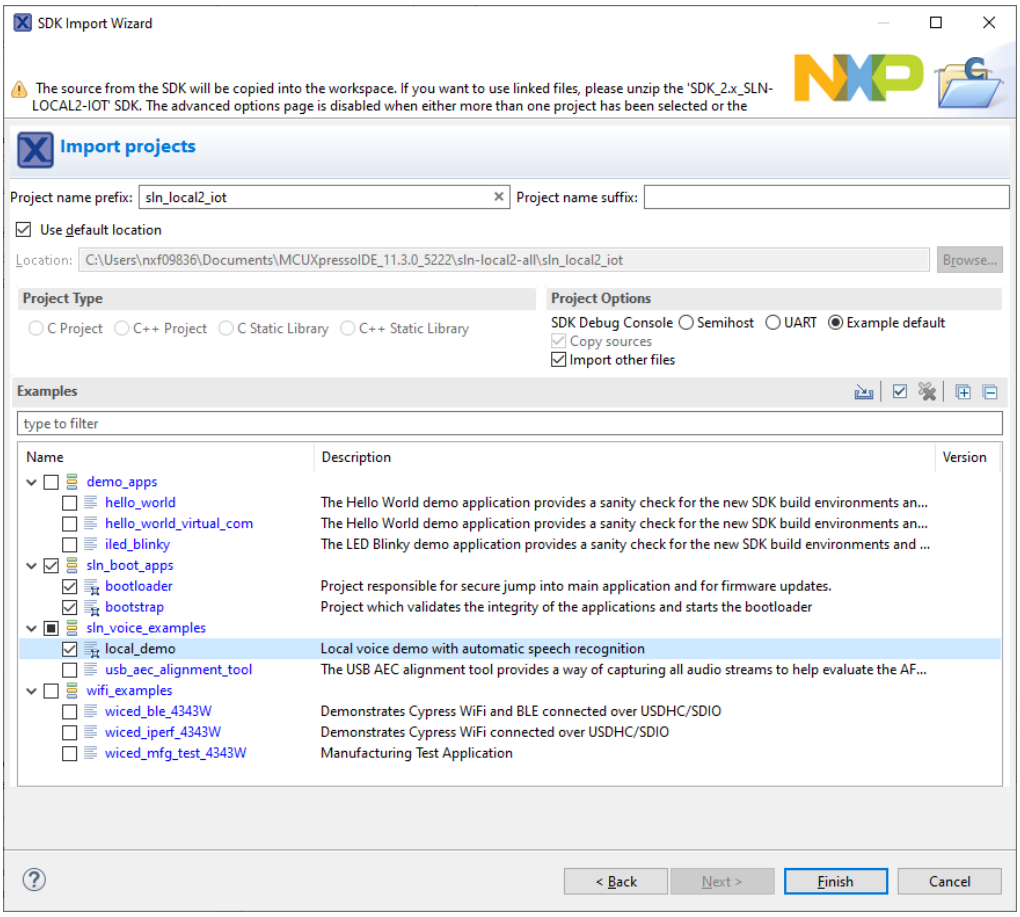


Figure 13. MCUXpresso project import selection

When the projects are successfully imported, they are listed in the project explorer, ready to be built and run. This document describes the **local_demo**, **bootloader**, and **bootstrap** for building and debugging. Figure 14 shows the **Project Explorer** window after the projects from the SLN-LOCAL2-IOT SDK are imported.

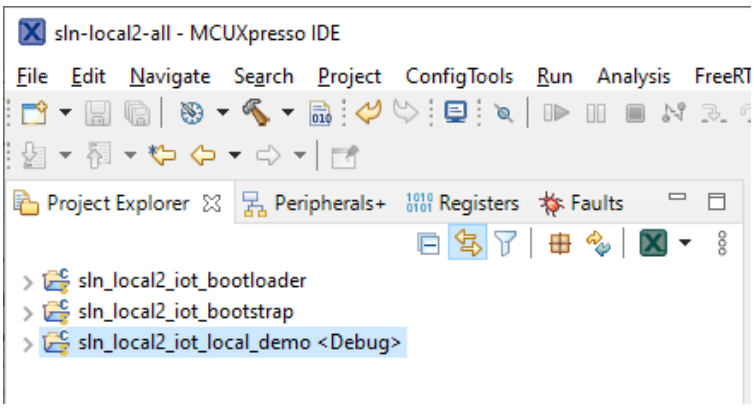


Figure 14. MCUXpresso Project Explorer

Chapter 5
Building and programming with MCUXpresso

5.1 Understanding the boot flow

Figure 15 shows the series of checks that occur during the boot. There are configuration options in various applications (ROM bootloader, bootstrap, bootloader) that determine which sequence is followed. If at any point a signature check fails and the High Assurance Booting (HAB) or image verification is enabled, the boot process stops.

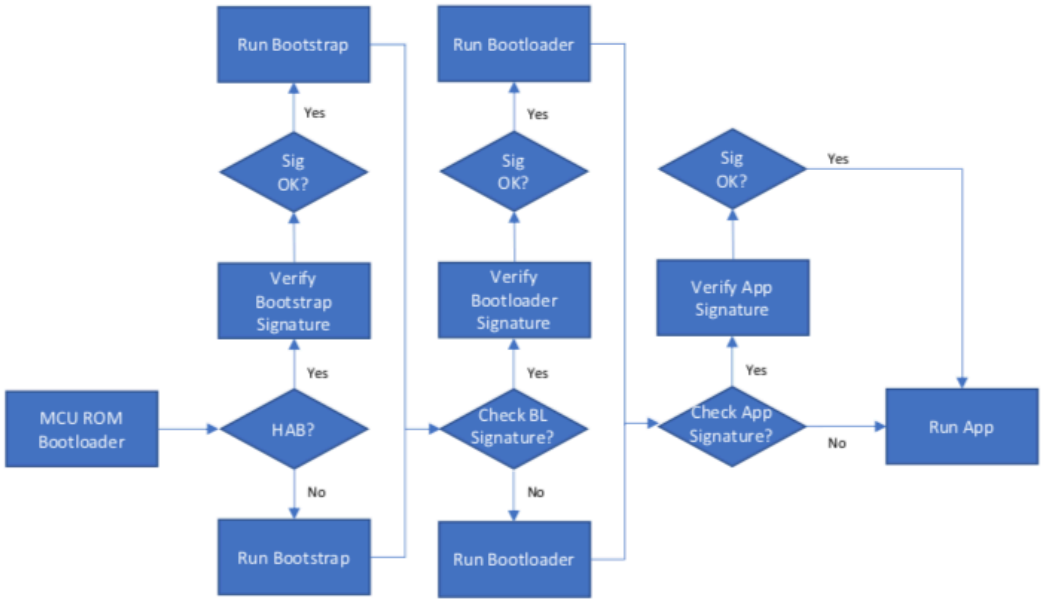


Figure 15. Boot security flowchart

By default, the SLN-LOCAL2-IOT kit has the image verification enabled and the HAB disabled in the bootstrap and bootloader.

The bootstrap project is the first application that boots. The architecture is described below. Bootstrap is a minimal FreeRTOS application that is responsible for image verification. If the i.MX RT HAB is enabled on the chip, bootstrap is the signed trusted firmware. This firmware is designed to avoid any updates, because the corruption of this image results in unbootable image and bricked device.

The **bootloader** project is a second-stage bootloader that manages jumping into the **local_demo** application. This application can be used for any additional bootloader functionality needed for the product. This bootloader is also responsible for the Mass Storage Device (MSD) dragging and dropping and updating the application image Over-the-Air (OTA) as well as Over-the-Wire (OTW). The bootloader also validates OTA / OTW images via signature verification.

The **local_demo** is the main application that runs the far-field local voice control.

5.2 Building the bootstrap, bootloader, and local voice control demo

From the **"Quickstart Panel"**, select **"Build"** to start the compilation and linking of the application for **sln_local2_iot_bootstrap**, **sln_local2_iot_bootloader**, and **sln_local2_iot_local_demo**. Figure 16 shows that the **sln_local2_iot_local_demo** project is selected and will start to compile after clicking the **"Build"** button.

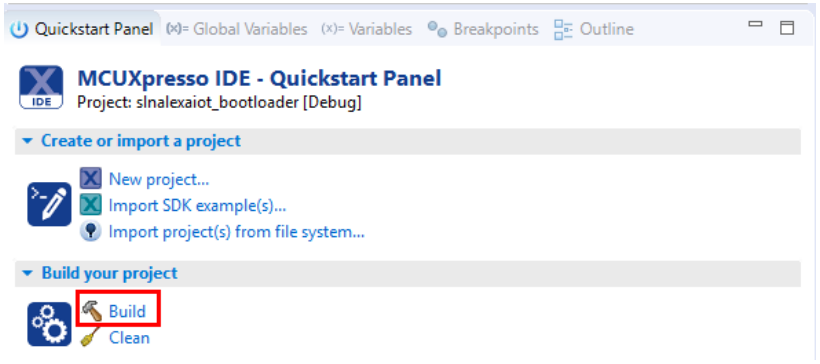


Figure 16. Quickstart panel

Wait for the console to finish the build. This may take a few minutes. Figure 17 shows the result of a successful compilation of the **sln_local2_iot_local_demo** project.

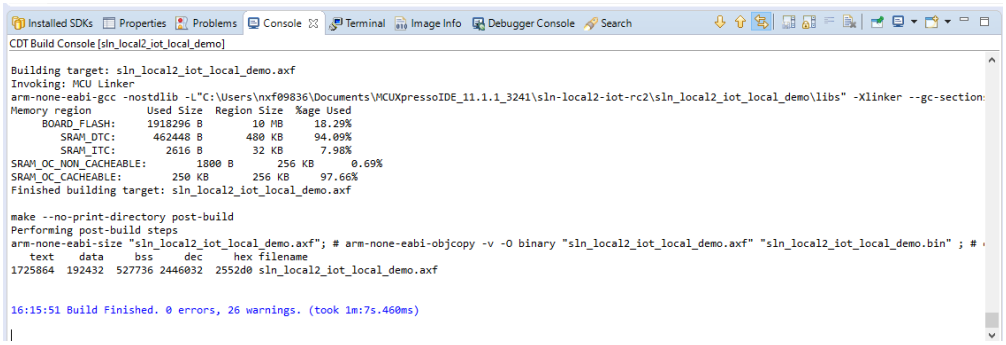


Figure 17. Console window showing successful compilation

5.3 Turning off image verification

The SLN-LOCAL2-IOT kit has the image verification turned on by default. This has the security feature of only booting images that are signed with the Certificate Authority that is associated with the application certificate and the Certificate Authority certificate programed in the flash. This disables programming an image into the flash and booting successfully.

For development purposes, consider turning this feature off to avoid signing images or when using NXP's security material, to avoid signing images. To do this, image verification must be turned off for both the bootstrap and the bootloader components.

When moving to production, it is suggested to turn the image verification on. To turn the image verification on, there is a single macro change required. The verification is application-specific, so if the entire security chain must be enabled, the setting must be updated in both the bootstrap and the bootloader applications.

5.3.1 Turning off bootstrap image verification

To turn off the image verification within the **bootstrap**, code modifications are required. Within the MCUXpresso IDE **bootstrap** project, right-click the root project and navigate to:

- **Properties > C/C++ Build > Settings ->Preprocessor**

Inside the **Preprocessor** section, change the MACRO "DISABLE_IMAGE_VERIFICATION" to "1" and click **Apply and Close**, as shown in Figure 18.

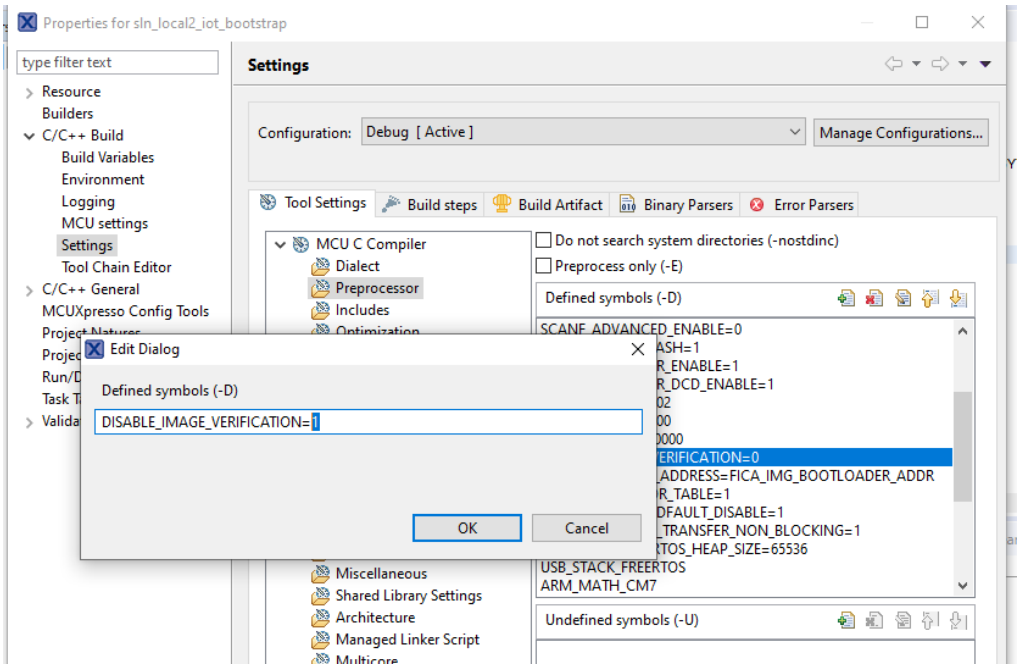


Figure 18. Disabling image verification in bootstrap

After that change, select the **Build** option from the quickstart panel (as shown in Figure 19) to start the compilation and linking of the **bootstrap**.

Figure 19. Build option in quickstart panel

5.3.2 Turning off bootloader image verification

To turn off the image verification within the **bootloader**, code modifications are required. Within the MCUXpresso bootloader project, right-click the root project and navigate to:

- **Properties > C/C++ Build > Settings > Preprocessor**

Inside the **Preprocessor** section, change the MACRO “DISABLE_IMAGE_VERIFICATION” to “1” and click “OK”, as shown in Figure 20.

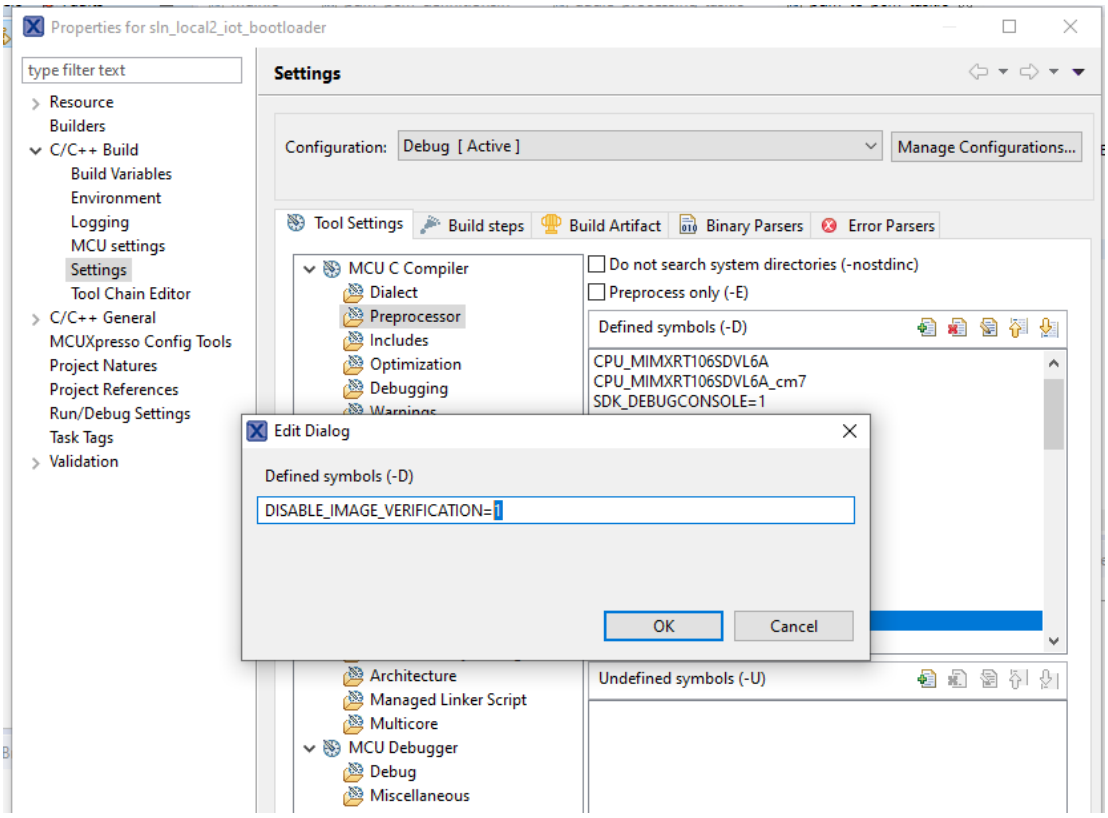


Figure 20. Disabling image verification in bootloader

After that change, select the **Build** option from the quickstart panel (as shown in Figure 21) to start the compilation and linking of the bootloader.

Figure 21. Build option in quickstart panel

5.4 Programming the firmware and artifacts

This section shows how to update the firmware. There are multiple ways to update the firmware, which also depends on whether the default NXP credentials are used. If the default NXP credentials are used, there are limitations on what can be updated without code changes.

By default, the image verification is on, which means that if the **bootloader** or **local_demo** are programmed without a valid signature in the Flash Image Configuration Area (FICA), the image verification fails and the code execution halts.

If the image verification is not disabled (it is enabled by default), then the only application that can be updated is the **local_demo** via the Mass Storage Device (MSD) update. To update the firmware without debugging, follow the steps in USB Mass Storage Device (MSD) update.

There are other ways to program firmware into the device with a section dedicated to the manufacturing package called “Ivaldi”, which is described in Automated manufacturing tools. These tools are available to manufacturers and developers for automated programing and taking a product from the assembly to the distribution autonomously.

The following section assumes that the **image verification** is disabled and all supporting artifacts are available to the developer and that the J-Link debug probe and MCUXpresso IDE are used.

5.4.1 Bootstrap, bootloader, and local voice control application images

With the **bootstrap**, **bootloader**, and **local_demo** all compiled, it is time to program them into the flash. This section assumes that you have either turned off the image verification or that the signing artifacts are already generated and ready to program, as described in NXP application image signing tool.

Perform the following steps for **bootstrap**, **bootloader**, and **local_demo**.

Select the debug option from those shown in Figure 22 and ensure that the debug probe is attached. This starts the process of loading the binary into the flash.

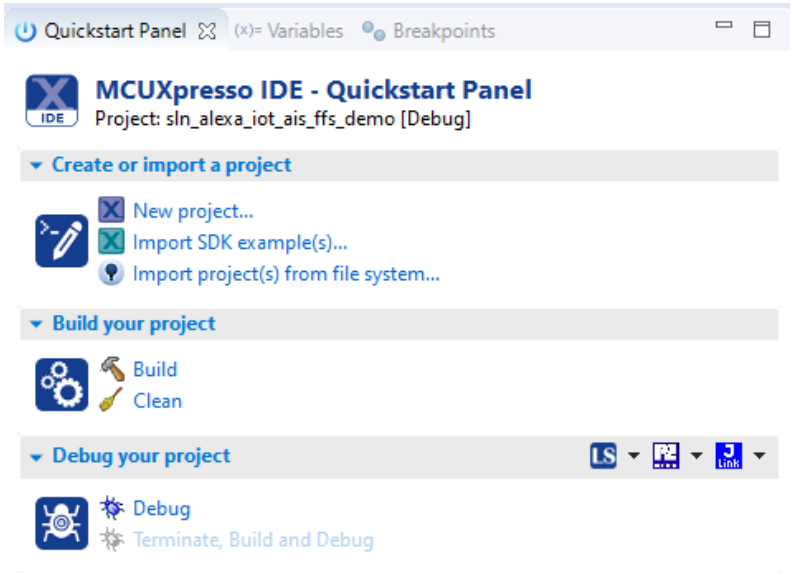


Figure 22. Debug window for applications

Select the J-Link probe that is connected to the board and click **"OK"**, as shown in Figure 23.

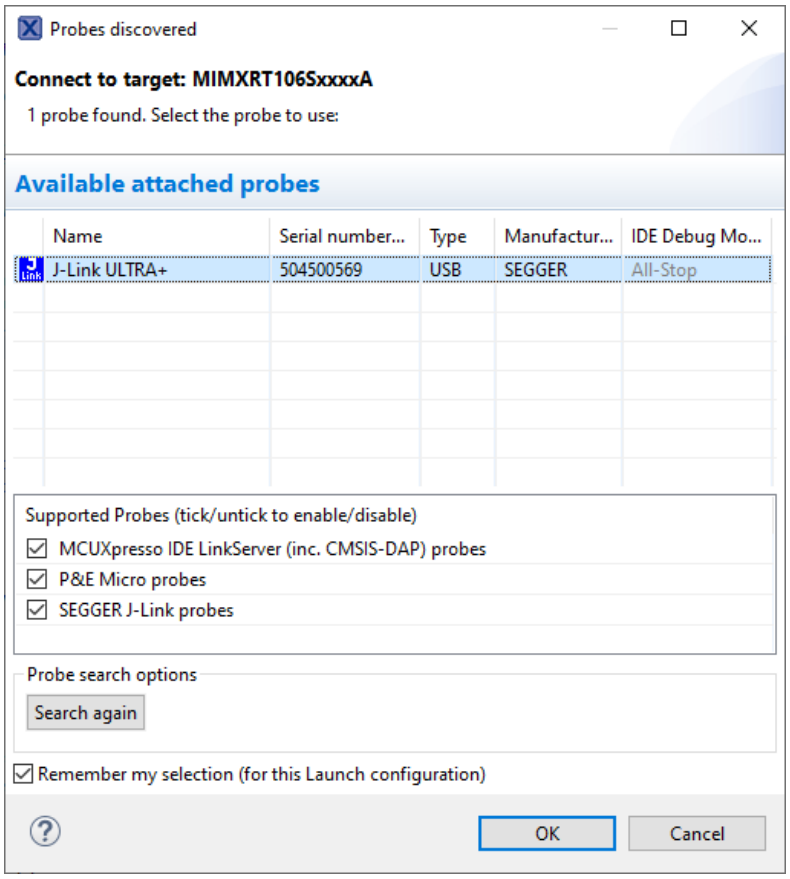


Figure 23. Probes discovered window

This launches the flashing tool and proceeds to load the image into the flash, as shown in Figure 24. When it is complete, you can proceed to the debug section.

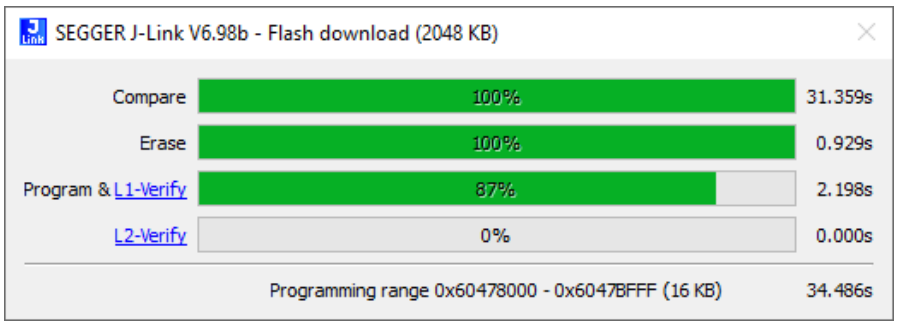


Figure 24. Downloading application image to flash

5.4.2 Audio playback files

The steps required to program the audio files to play from the SLN-LOCAL2-IOT kit are described here. It is assumed that the audio files are available to the developers using the pre-built RAW form of files inside the "**Image_Binaries/local_audio_files**" folder in the Ivaldi package, or they are generated, as described in Generating new audio playback files.

Ensure that the SLN-LOCAL2-IOT kit is USB-powered with the JTAG connected to the back of the board. Within the MCUXpresso IDE, ensure that you have selected a project to launch the debug configuration in and select the **GUI Flash Tool** icon, as shown in Figure 25.

Figure 25. Opening Flash GUI Tool for programming audio playback binaries

The **Probes discovered** window (shown in Figure 26) appears if the project has never been used to program the SLN-LOCAL2-IOT kit before.

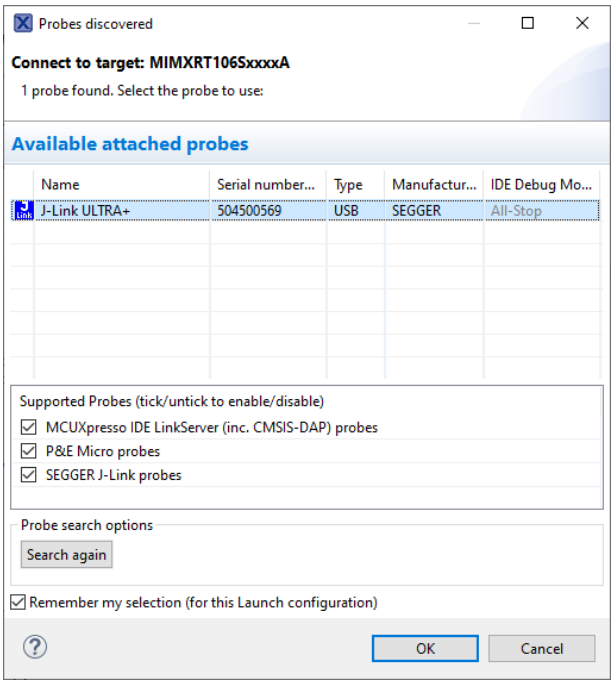


Figure 26. Probes discovered window for programming audio playback binaries

After clicking **"OK"**, the Flash GUI Tool shown in Figure 27 pops up.

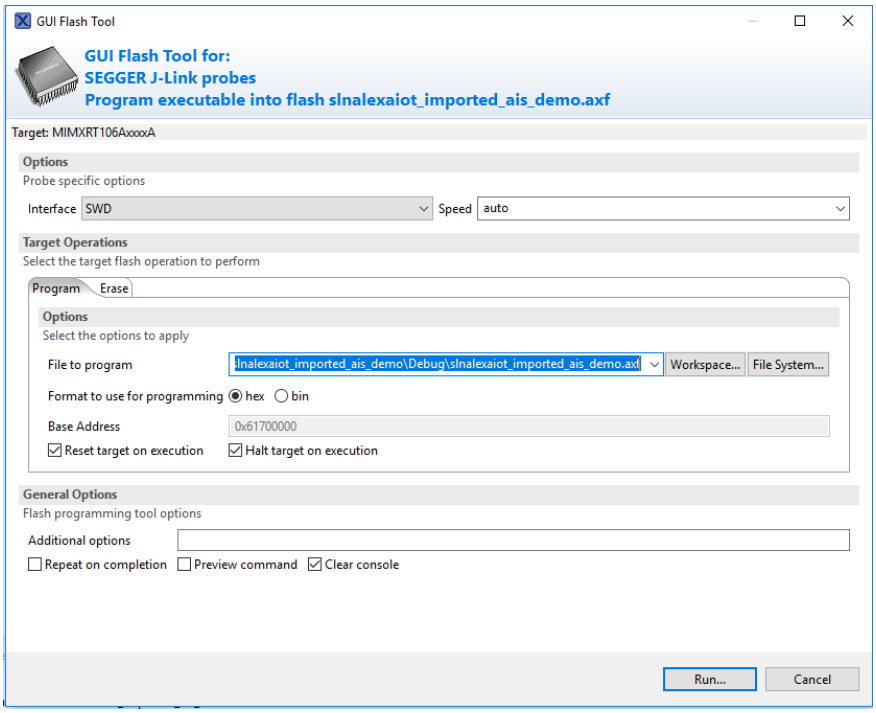


Figure 27. Opening GUI Flash Tool for audio playback binaries

The GUI Flash Tool automatically fills in the fields associated with the project that must be changed. Select the **"Filesystem"** button, which opens the window shown in Figure 28. Within that window, navigate to the audio playback binary files that were downloaded from the Ivaldi package or generated, as shown in Generating new audio playback files.

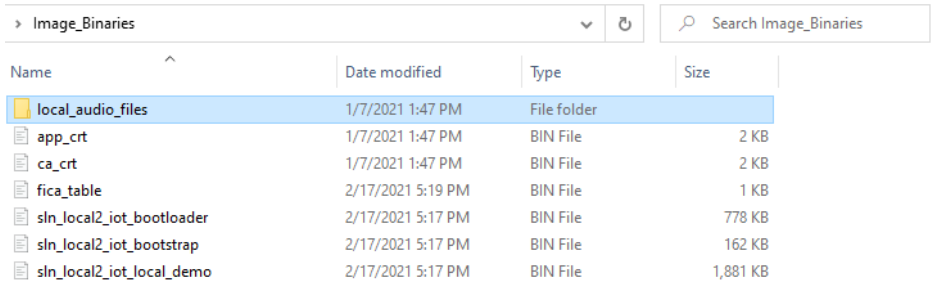


Figure 28. Selecting audio playback binary files

Within the **"Base Address"** textbox, enter the address where the audio file is located and click the **"Run"** button. In Figure 29, for demonstration purposes, we program the "OK" audio playback in English at the 0x6178_0000 address.

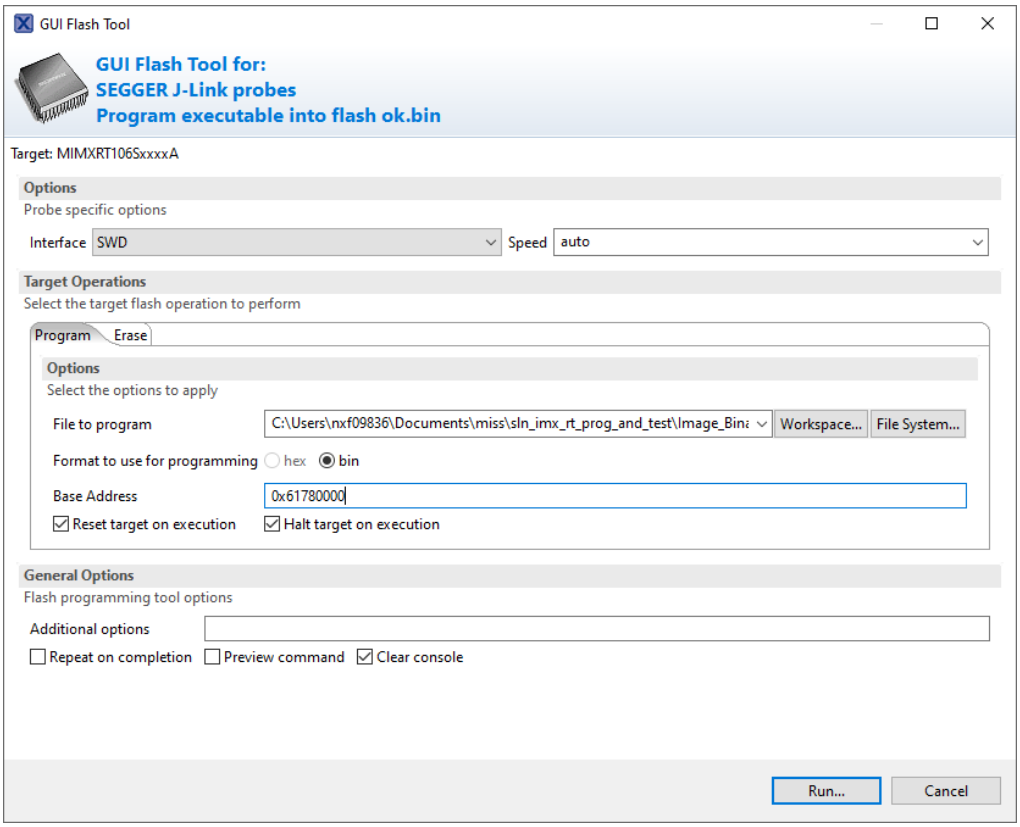


Figure 29. Updating the “OK” audio playback in English binary address

This programs the audio binary file into the designated flash section, as shown in Figure 30. After the flashing process is done, the **“Operation Completed”** window (shown in Figure 31) appears.

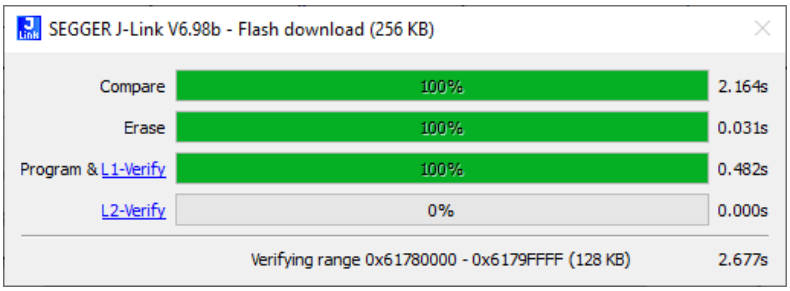


Figure 30. Programming the audio playback binaries

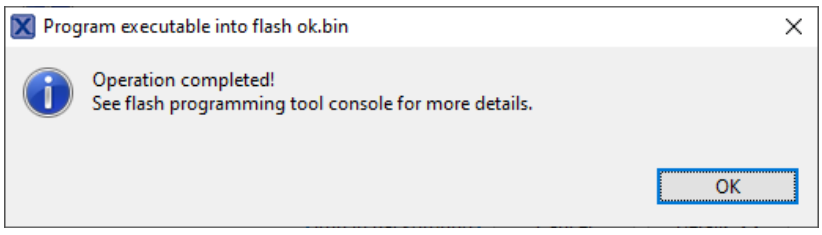


Figure 31. Audio “OK” in English binary programming completed

See Table 5 for the list of all audio file addresses saved in the HyperFlash memory.

5.4.3 Image verification certificate and keys

The following section describes the steps required to program the image Certificate Authority and Application public certificate to validate images. This section assumes that the artifacts are available to the developer using the pre-built binaries inside the "**Default Binaries**" folder in the release package or that they are generated and converted to files.

Ensure that the SLN-LOCAL2-IOT kit is USB-powered with the JTAG connected to the back of the kit. In the MCUXpresso IDE, ensure you have selected a project to launch the debug configuration in and select the GUI Flash Tool icon, as shown in Figure 32.

Figure 32. Opening Flash GUI Tool for Application/CA certificates

The **Probes discovered** window (Figure 33) is shown if the project has never been used to program the SLN-LOCAL2-IOT kit before.

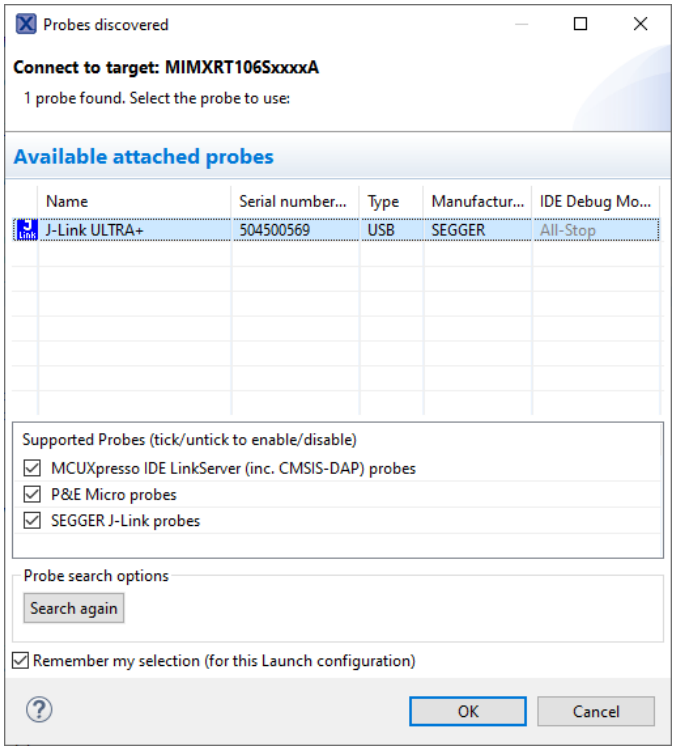


Figure 33. Probes discovered window for Signed Application/CA certificates

After clicking **"OK"**, the GUI Flash Tool pops up, as shown in Figure 34.

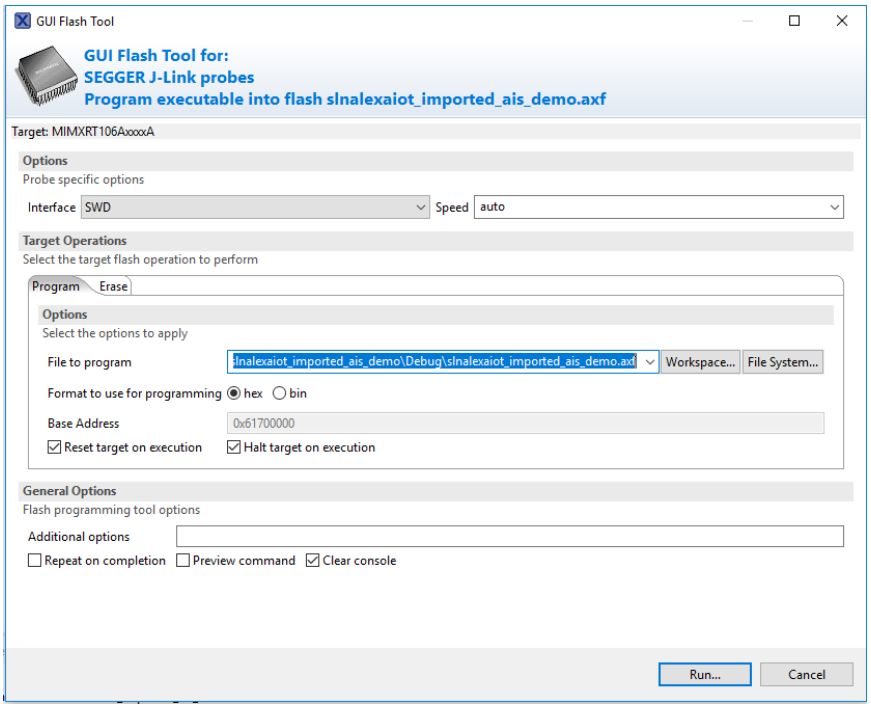


Figure 34. Opening Flash GUI Tool for Application/CA certificates

The GUI Flash Tool automatically fills in the fields associated with the project that must be changed. Select the **"File System..."** button, which opens the window, as shown in Figure 34. Within that window, navigate to the application certificate (**app_crt.bin**) and the CA (**ca_crt.bin**) that were recovered from the device or generated following the instructions in Automated manufacturing tools.

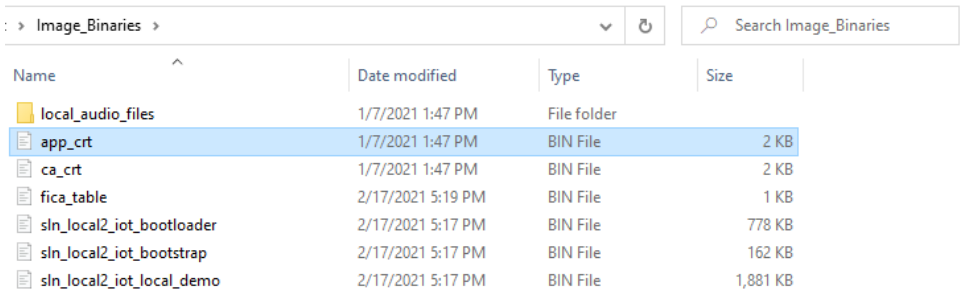


Figure 35. Selecting the Application/CA certificate binaries

Within the **"Base Address"** textbox, enter "0x61D00000" and "0x61D80000" (must be done for both) for the certificate (**app_crt.bin**) or "0x61CC0000" for the certificate authority public certificate (**ca_crt.bin**) and click the **"Run"** button. In Figure 36, for demonstration purposes, the same **app_crt.bin** certificate is used for both banks. For security reasons, using different certificates for each bank is recommended.

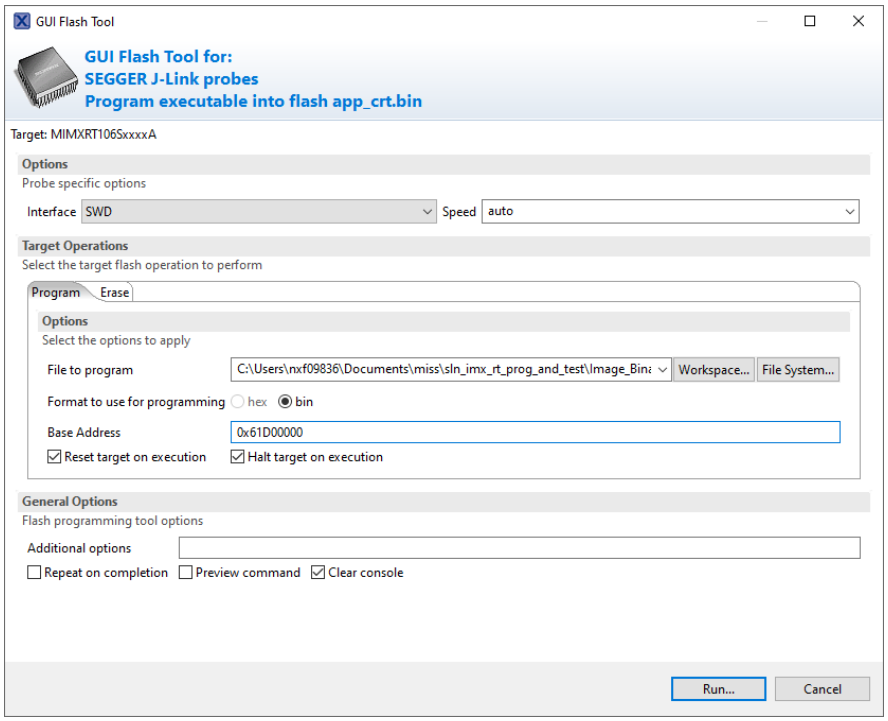


Figure 36. Updating the Application/CA certificate binaries address

This starts programming the certificate, which is in a file system format, into the designated flash section, as shown in Figure 37. After the flashing process is done, the **"Operation Completed"** window appears (Figure 38).

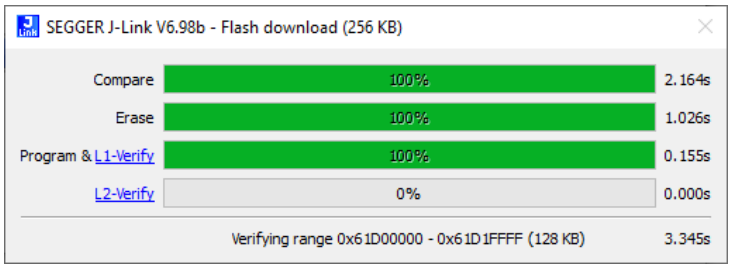


Figure 37. Programming the Application/CA certificate binaries

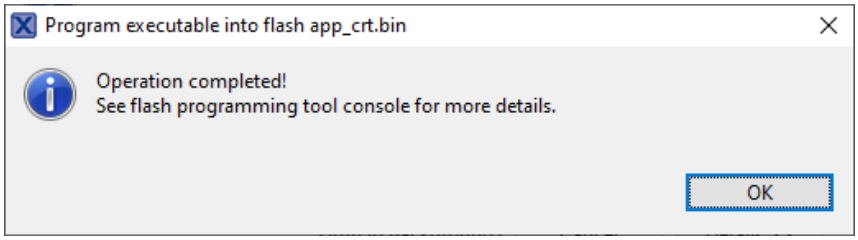


Figure 38. Application/CA Certificate programming complete

5.4.4 Flash Image Configuration Area (FICA)

The following section describes the steps required to program the Flash Image Configuration Area (FICA). The FICA is described in more detail in FICA and image verification. Regardless of whether the verification is turned on or off, the FICA must be programmed into the area, because it holds the boot information about which image should be booted.

Ensure that the SLN-LOCAL2-IOT kit is USB-powered with the JTAG connected to the back of the board. In the MCUXpresso IDE, ensure that you have selected a project to launch the debug configuration in and click the GUI Flash Tool icon, as shown in Figure 39.

Figure 39. Opening Flash GUI Tool for FICA

The **"Probes discovered"** window (Figure 40) is shown if the project has never been used to program the SLN-LOCAL2-IOT kit before.

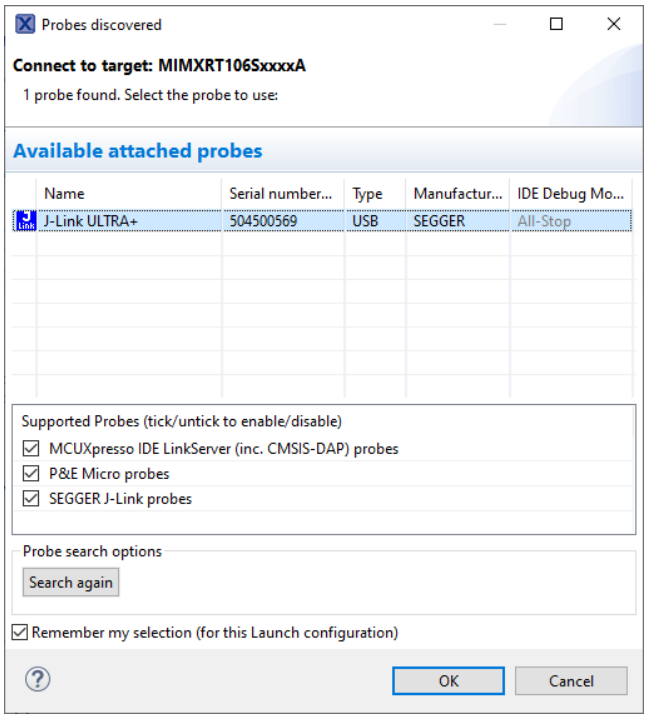


Figure 40. Probes discovered window for FICA table programming

After selecting **"OK"**, the Flash GUI Tool pops up (Figure 41).

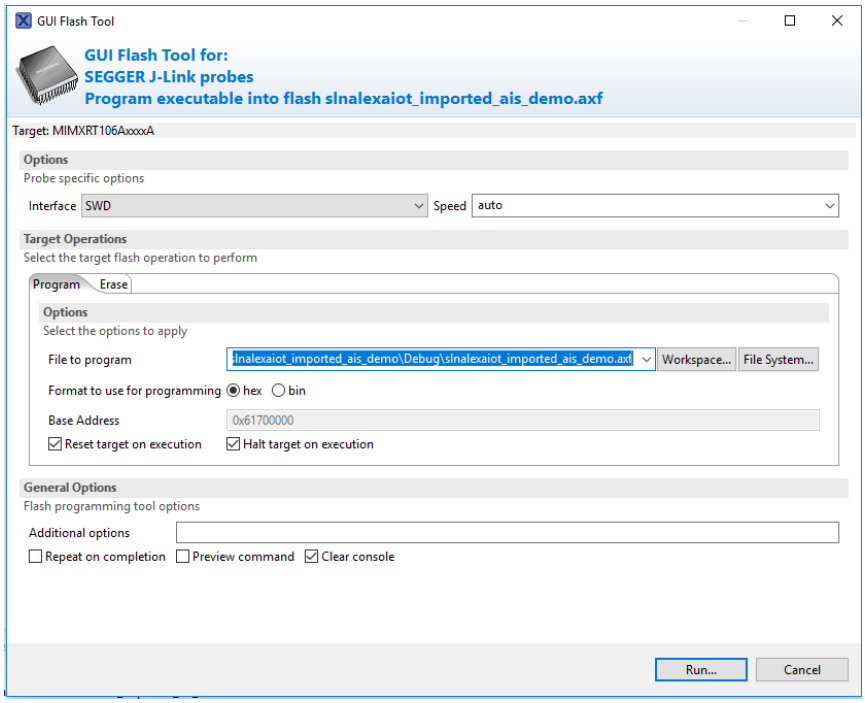


Figure 41. GUI Flash Tool

The Flash GUI Tool automatically fills in the fields associated with the project to be changed. Click the **“Filesystem”** button, which opens the window (Figure 42). Within that window, navigate to the directory that contains the generated FICA after running the Ivaldi package in Introduction, NXP application image signing tool, and Open Boot Programming tool. Select the “**fica_table.bin**” file to download.

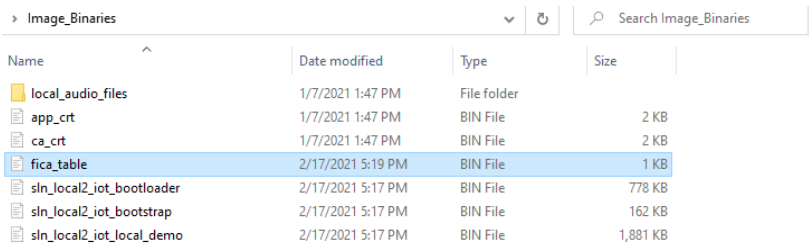


Figure 42. Selecting the FICA table binary

Within the **“Base Address”** textbox, enter **“0x61FC0000”** and hit the **"Run.."** button, as shown in Figure 43.

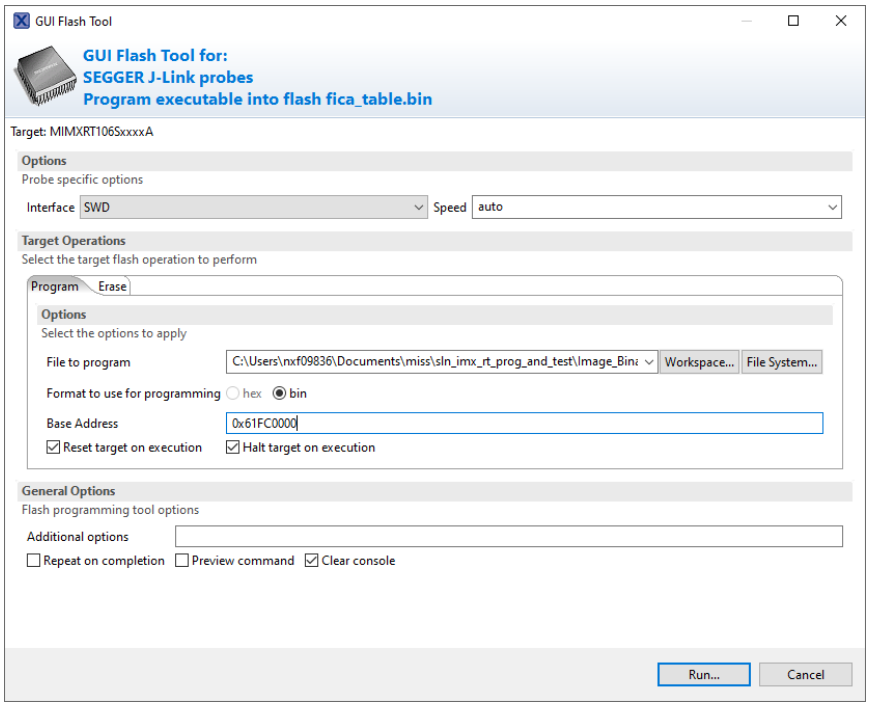


Figure 43. Updating the FICA table address

NOTE

If the verification is not turned off, the self-built images do not work with the NXP demo system.

This starts to program the FICA, which is in a file system format into the designated flash section, as shown in Figure 44. After the flashing process completes, the **"Operation Completed"** window appears (Figure 45).

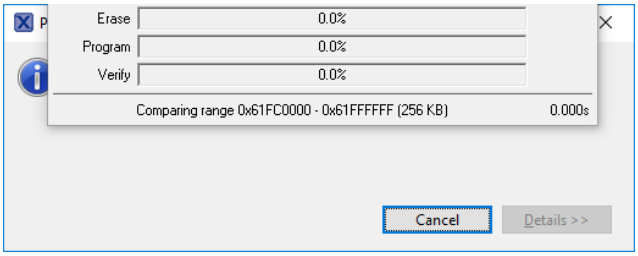


Figure 44. Programming the FICA table binary

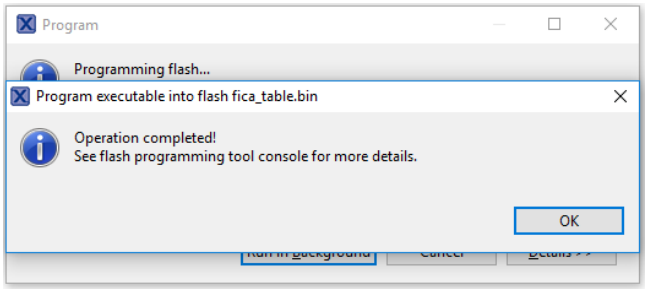


Figure 45. FICA table programming complete

Chapter 6
Hardware platform

The hardware platform of the SLN-LOCAL2-IOT development kit is described on the web page: www.nxp.com/mcu-local2.

- SLN-LOCAL2-IOT Schematics
- SLN-LOCAL2-IOT BOM
- SLN-LOCAL2-IOT Design Files

Chapter 7
Far-field local voice control framework

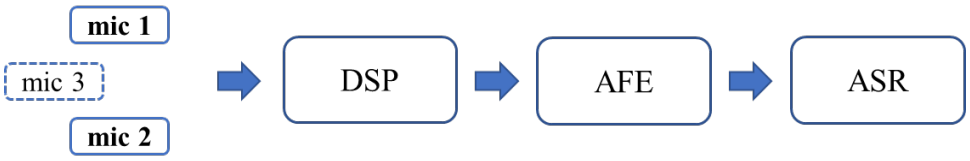


Figure 46. High-level overview of far-field local voice control framework

This section describes the software framework that supports the far-field local voice control. As shown in Figure 46, two (optionally three) microphones collect the acoustic signal, followed by the DSP, AFE, and ASR blocks.

The SLN-LOCAL2-IOT kit is acoustically qualified for far-field voice applications with three PDM microphones and has been internally tested with two-microphone configurations with a range of mainstream products also using the two-microphone configuration. When making modifications, ensure to re-test the application against standard acoustic test guidelines. The SLN-LOCAL2-IOT kit is based on the acoustic architecture of the SLN-ALEXA-IOT kit. It was tested based on the Amazon Voice Service self-test guidelines, which are available at https://developer.amazon.com.

NXP has pre-tuned and qualified the DSP and AFE libraries with the SLN-LOCAL2-IOT hardware platform. By default, modifications on the DSP and AFE are not needed. However, to create customized hardware or proof-of-concepts, see Acoustic modification and ensure that the modification is suitable for your product.

The ASR block in Figure 46 contains the speech recognition engine and the application software. NXP has implemented the following three types of baseline demos:

- LED voice control demo
  - English
  - Two-stage (wake word and command) ASR
- Smart Home (IoT) or elevator or audio device or washing machine voice control demo
  - Selectable combinations of English, Chinese, German, and French
  - Two-stage ASR
- Oven voice control demo
  - English
  - Multiturn (4-way) dialog-style ASR

The ASR implemented with the selected languages can be easily replaced with other languages. NXP provides an application note for customization of the local voice demos. Contact NXP (local-commands@nxp.com) for information about the process of phoneme-based speech recognition engine generation and custom wake words and commands.

Automatic speech recognition describes the baseline ASR demos that you can reuse for your product.

7.1 Automatic speech recognition

The flagship feature of SLN-LOCAL2-IOT is the bundled voice control engine, also called ASR. NXP offers a lightweight engine designed specifically for MCUs. It supports various use cases with flexible inference engine instances.

7.1.1 ASR application scenarios

Figure 47 shows the ASR with multiple languages, where each language consists of a wake word inference engine and *N* instances of command inference engines. Developers can implement various application scenarios that are described in the following subsections.

The SLN-LOCAL2-IOT with i.MX RT106S supports up to four languages of the ASR in runtime. With i.MX RT105S, it reduces to two languages because of the reduced RAM size.

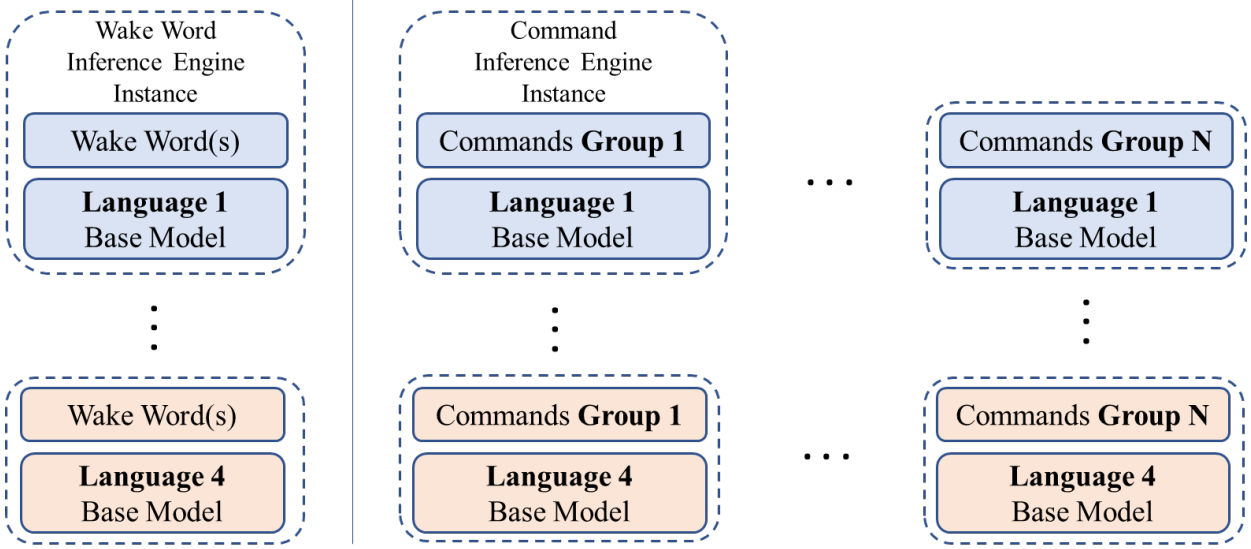


Figure 47. Inference engine instances matrix for flexible ASR applications

The simplest ASR application is a single-language two-stage ASR with only one wake word engine and one command engine. The ASR for multiple (up to four) languages can be created with the inference engine instances of *columns* in Figure 47. The ASR for the multiturn (e.g. dialog) application can be created with the inference engine instances of a *row* in Figure 47.

7.1.1.1 Scenario #1: Single-language two-stage voice control

Figure 48 shows the simplest ASR scenario with one wake word inference engine followed by a command engine instance in a selected language. By default, NXP has implemented the *LED voice control demo in English* language.

- Wake word
  - Hey, NXP
- Commands
  - L, E, D, red
  - L, E, D, green
  - L, E, D, blue
  - Cycle fast
  - Cycle slow

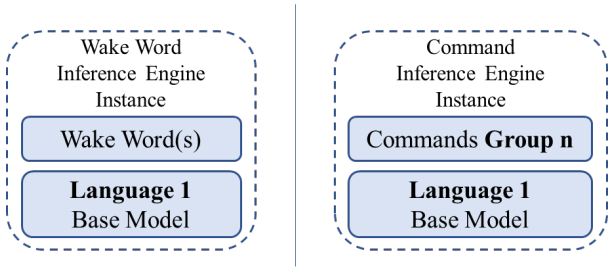


Figure 48. Inference engine instances of single-language two-stage scenario

The SLN-LOCAL2-IOT kit plays the “can I help you?” and “OK” audio responses respectively when the wake word and commands are detected. The audio playback files are saved in the filesystem. You can replace them with your own files. For details on the filesystem, see Filesystem. For the full list of audio file addresses, see Table 5.

7.1.1.2 Scenario #2: Multiple-language two-stage voice control

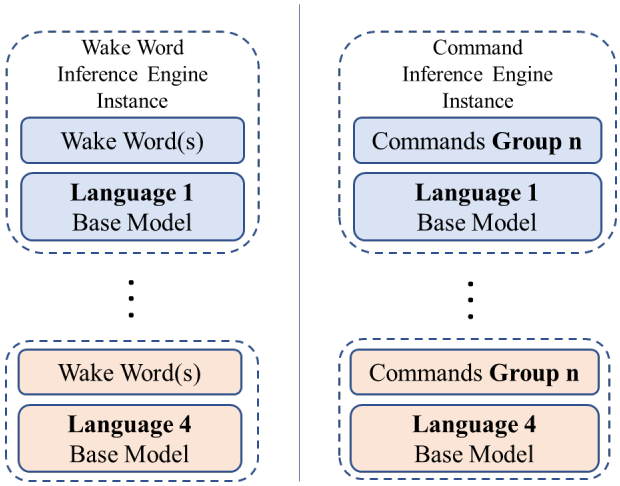


Figure 49. Multiple (up to four) languages of wake word and command inference engines

The SLN-LOCAL2-IOT kit with the i.MX RT106S MCU supports up to four languages of the wake word and command engine instances, as shown in Figure 49. This scenario is a two-stage voice control application. The four language instances for both the wake word and the command are saved in the flash memory. Users can select any combination of the four language instances. The selected languages’ wake word engines are loaded into their dedicated RAM memory pool and start receiving the voice data stream. When one of the wake words is detected, the same language’s command engine instance is loaded into its memory pool to start listening to the user’s voice command. For example, suppose that two languages (English and Mandarin) are enabled. The SLN-LOCAL2-IOT kit loads the wake word engines (that is English and Mandarin) into their RAM memory pools and starts listening to the user’s voice. If the user utters the English wake word “Hey, NXP”, the SLN-LOCAL2-IOT kit detects the wake word, loads the command engine for English into the RAM memory pool and starts listening to voice commands.

NOTE

For multiple wake engines listening to the voice stream simultaneously, the False Acceptance Rate (FAR) can increase. The wake word inference engines must be fine-tuned to mitigate the FAR.

It is also possible to load one wake word inference engine, followed by command engines of multiple languages. In this case, the FAR can be low. Developers must avoid similar pronounciation among different languages’ voice commands.

By default, NXP has implemented voice control demos for the Smart Home, Elevator, Audio Device Control, and Washing Machine applications in English, Chinese, German, and French. All the available wake words and commands are listed in Table 7.

Table 7. Wake words and commands for multi-language demos

Language	Wake word	Smart Home commands	Elevator commands	Audio Device Control commands	Washing Machine commands
English (EN)	Hey, NXP	Temperature Up Temperature Down Window Up Window Down Turn On Turn Off Brighter Darker	First Floor Second Floor Third Floor Fourth Floor Fifth Floor Main Lobby Going Up Going Down Open Door Close Door	Turn On Turn Off Play Pause Start Stop Next Track Previous Track Volume Up Volume Down	Wash Delicate Wash Normal Wash Heavy Duty Wash Whites Cancel
Chinese (ZH)	你好, 恩智浦	温度升高 温度降低 打开窗帘 关上窗帘 开灯 关灯 亮一点 暗一点	一楼 二楼 三楼 四楼 五楼 大堂 上行 下行 开门 关门	打开 关掉 播放 暂停 开始 停止 下一首 上一曲 提高音量 音量减小	精致洗 正常清洗 强力洗 洗白 取消
German (DE)	Hallo, NXP	Temperatur erhöhen Temperatur verringern Fenster hoch Fenster runter anschalten Ausschalten heller dunkler	Erste Etage Zweite Etage Dritte Etage Vierte Etage Fünfte Etage Hauptlobby Hochfahren Runterfahren Öffne die Tür Schließe die Tür	anschalten ausschalten abspielen Pause Anfang halt nächstes Lied vorheriges Lied Lautstärke erhöhen Lautstärke verringern	Feinwäsche Normalwäsche stark verschmutze Wäsche Weißwäsche abbrechen
French (FR)	Salut, NXP	Augmenter Température Diminuer Température	Premier Etage Deuxième Etage	Allumer Eteindre	Lavage Délicat Lavage Normal

Table continues on the next page...

Table 7. Wake words and commands for multi-language demos (continued)

Language	Wake word	Smart Home commands	Elevator commands	Audio Device Control commands	Washing Machine commands
		Monter Fenêtre Baisser Fenêtre Allumer Eteindre Augmenter Luminosité Diminuer Luminosité	Troisième Etage Quatrième Etage Cinquième Etage Entrée Principale Monter Descendre Ouvrir Porte Fermer Porte	Lecture Pause Démarrage Arrêt Piste Suivante Piste Précédente Augmenter Volume Baisser Volume	Lavage en Profondeur Lavage Blanc Annuler

Wake word engines of up to four languages can run simultaneously. Users must select a demo command group to run after a wake word is detected, because the scenario is a two-stage (wake word followed by commands) voice control. If a wake word is detected, the same language's demo command group is loaded to listen to voice commands.

If the SLN-LOCAL2-IOT kit is triggered by a wake word, it turns the LED blue playing the audio in the detected languages:

- Can I help you? (English)
- 我可以帮你吗? (Chinese)
- Kann ich Ihnen helfen? (German)
- Puis-je vous aider? (French)

Once a command is detected, it turns the **LED green** playing the audio **"OK!" in the accent of same language** for the wake word triggered.

The audio playback files are saved in the filesystem. You can replace them with your own files. For details on the filesystem, see Filesystem. For the full list of audio file addresses, see Table 5.

7.1.1.3 Scenario #3: Single-language N-stage voice control

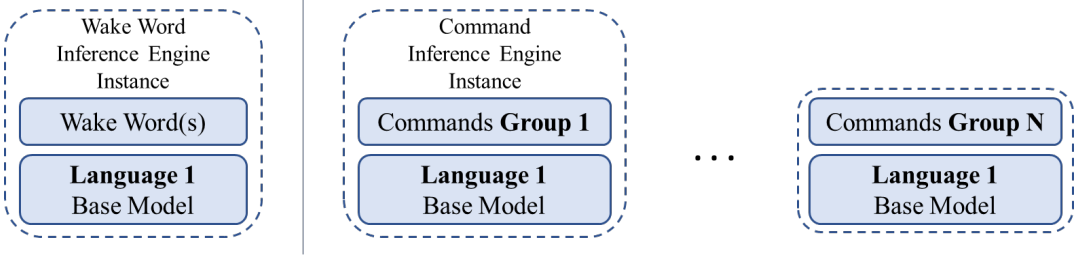


Figure 50. Wake word and command engine instances for single-language N-stage voice control

You can also create a multiturn application with the engines in the row instances in Figure 47. As shown in Figure 50, a language wake word engine is followed by a series up to N of command engines. NXP has implemented a *dialog-type voice control demo*. Figure 51 shows an example of the oven appliance use case.

The audio playback files are saved in the filesystem. You can replace them with your own files. For details on the filesystem, see Filesystem. For the full list of audio file addresses, see Table 5.

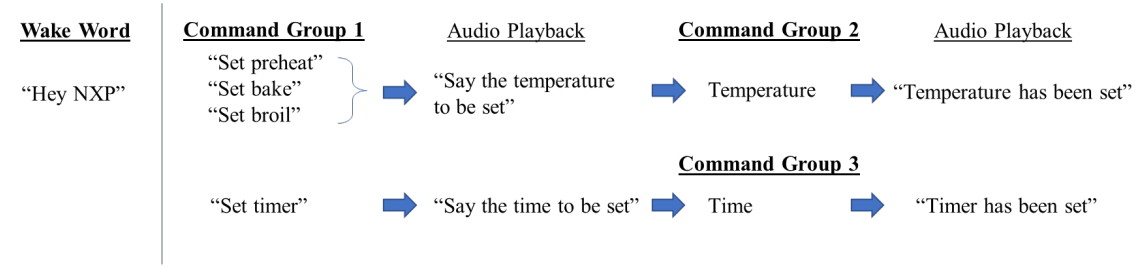


Figure 51. Dialog-type voice control with oven appliance use case

7.1.1.4 User interface

The three demo scenarios explained above are selected by the shell commands in the serial terminal window. Figure 52 shows the "**changeto**" and "**multilingual**" shell commands to select a demo and multiple languages. For more details about the usage, type "help" into the shell prompt in the serial terminal, as shown in Figure 53 and Figure 54 for the "**changeto**" and "**multilingual**" commands, respectively.

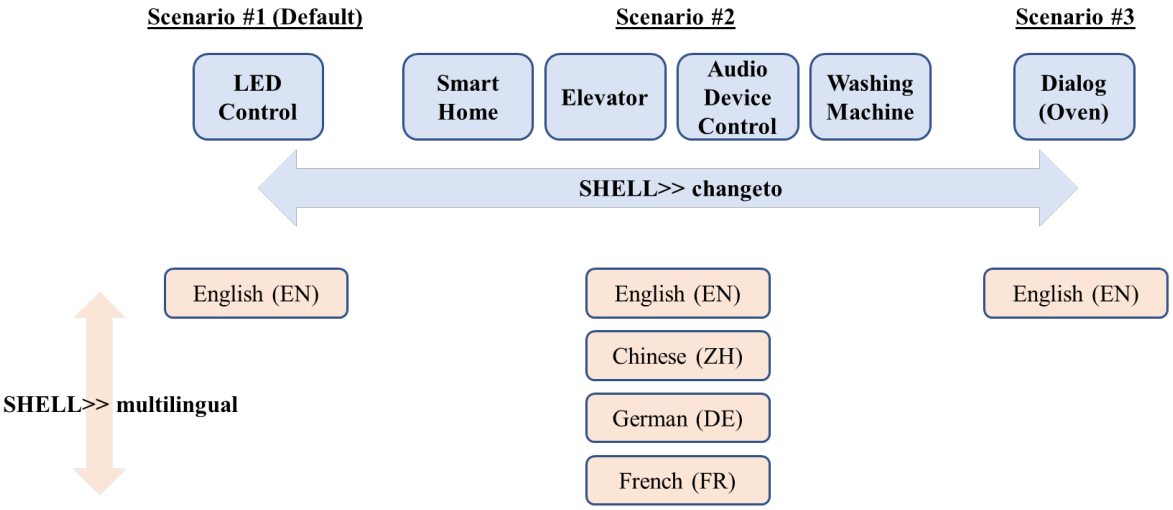


Figure 52. Demo selection by shell commands

```
"changeto": Change the command set
        Usage:
           changeto <param>
        Parameters
           elevator: Elevator control
           iot: IoT
           audio: Audio control
           wash: Washing machine
           led: LED control (auto-enabling English)
           dialog: Dialogic commands for oven (auto-enabling English)
```

Figure 53. Demo selection command

```
"multilingual": Select language model(s). Save in flash memory.
        Usage:
           multilingual language_code1 up to language_code4
        Parameters
           language_codes - en, zh, de, fr
```

Figure 54. Language selection command

7.1.2 Language-specific voice control engine

7.1.2.1 Specification

The speech recognition engine is based on the state-of-the-art deep neural network technique. Note that the engine is not intended for natural language understanding, but for the keyword spotting which is useful for various MCU-based applications. The computing resource consumption is based on fixed-point operations and almost constant. The specification of an inference engine instance is described in Table 8. Because the Chinese language requires tone recognition, its voice engine requires more resources than the other languages. The CPU consumption can increase with the number of commands. The rule-of-thumb is 0.08 MIPS per a 4-syllable command.

Table 8. Specification of an inference engine instance

	Chinese (with tone recognition)	Other languages
Code size	150 KB	30 KB
Data size	170 KB + 32 x *M* Bytes	155 KB + 32 x *M* Bytes
RAM	85 KB + 128 x *M* Bytes	45 KB + 128 x *M* Bytes
CPU	68 MIPS*	45 MIPS*

M: The number of wake words or commands.

*: Optimized for the SIMD instructions. The values of 68 and 45 represent typical voice control applications.

7.1.2.2 Architecture

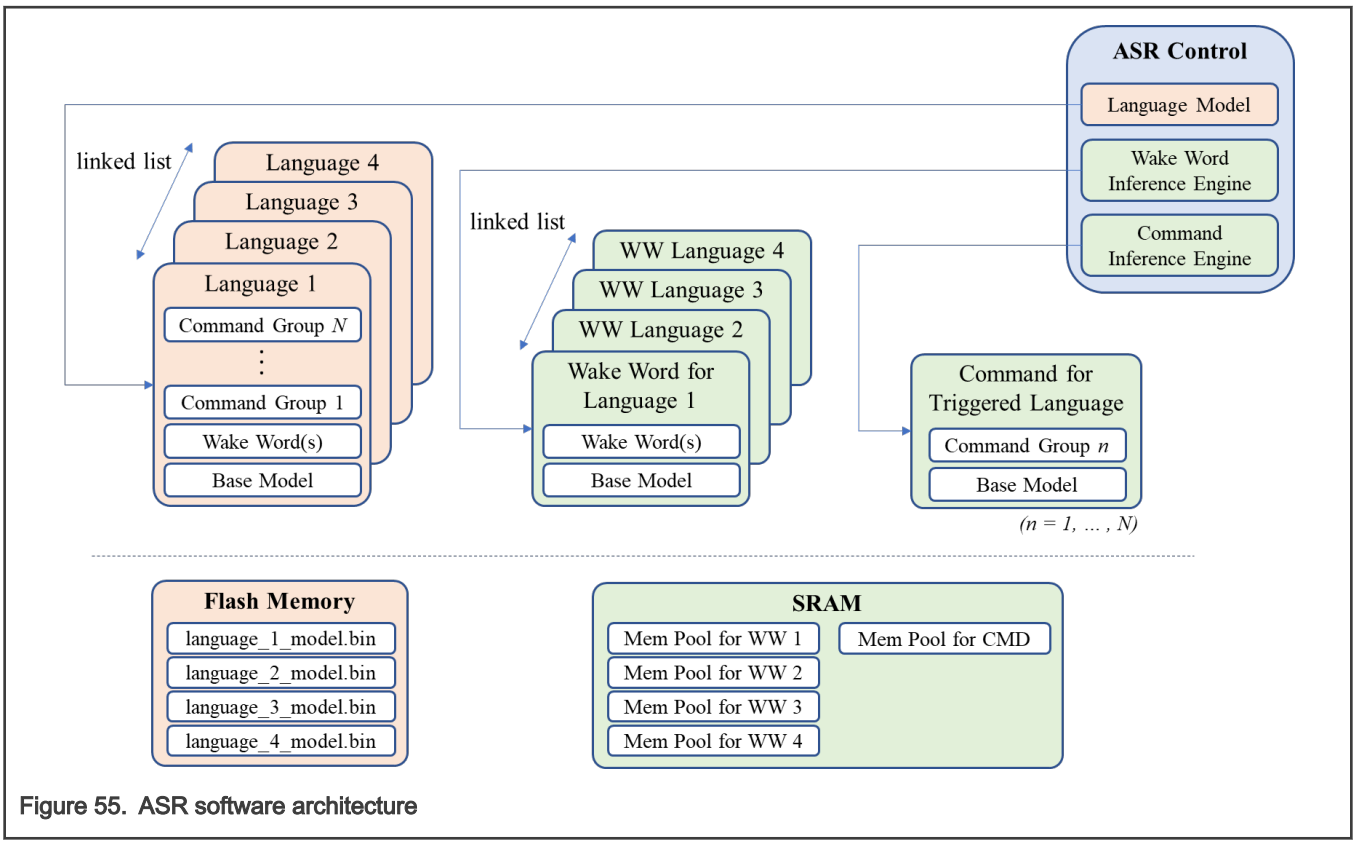


Figure 55. ASR software architecture

Figure 55 shows the software architecture of NXP's ASR for SLN-LOCAL2-IOT. The ASR control structure points to a command inference engine instance, the linked lists of language models, and the wake word engines. There are memory pools assigned for the command as well as the wake word inference engine instances in the SRAM. The **language_x_model.bin** files are located in the flash memory, where each BIN file is basically a pack of a language's base model and *N+1* groups of wake word(s) and commands.

The ASR control snippet is shown in Figure 56. It describes the three major structure members.

```
typedef struct _asr_control
{
    struct asr_language_model *langModel;      // linked list
    struct asr_inference_engine *infEngineWW;  // linked list
    struct asr_inference_engine *infEngineCMD; // not linked list
    asr_result_t result;                       // results of the command processing} asr_control_t;
};
```

Figure 56. ASR control snippet

7.1.2.3 Language model

```
struct asr_language_model
{
    asr_language_t iWhoAmI;
    uint8_t nGroups;
    unsigned char *addrBin;
    unsigned char *addrGroup[MAX_GROUPS];
    unsigned char *addrGroupMapID[MAX_GROUPS - 1];
    struct asr_language_model *next
};
```

Figure 57. ASR language model snippet

The language model structure in Figure 57 consists of the following members:

- **iWhoAmI**: the ASR model is language-specific. Each ASR voice engine must define a language. You can extend Table 9 with other languages.
- **nGroups**: the number of groups in a language model binary. By default, it contains a language-specific base model (counted as a group) and a wake word group. You can define *N* command groups.
- **addrBin**: the address of a language model binary. This address points to a base model.
- **addrGroup[MAX_GROUPS]**: the addresses of wake word and *N* command groups. MAX_GROUPS should be greater than or equal to *N+1*.
- **addrGroupMapID[MAX_GROUPS - 1]**: addresses of MapIDs. A MapID is an ID that can be assigned to a set of commands.
- **next**: the pointer to the next language model in a linked list.

Table 9. ASR language type

Language type	asr_language_t code	Encoding
Unknown	UNDEFINED_LANGUAGE	0x0000
English (EN)	ASR_ENGLISH	0x0001
Chinese (ZH)	ASR_CHINESE	0x0002

Table continues on the next page...

Table 9. ASR language type (continued)

Language type	asr_language_t code	Encoding
German (DE)	ASR_GERMAN	0x0004
French (FR)	ASR_FRENCH	0x0008

7.1.2.4 Inference engine

```
struct asr_inference_engine
{
    asr_inference_t iWhoAmI_inf;
    asr_language_t iWhoAmI_lang;
    void *handler;
    uint8_t nGroups;
    unsigned char *addrGroup[2];
    unsigned char *addrGroupMapID;
    char **idToKeyword;
    unsigned char *memPool;
    uint32_t memPoolSize;
    struct asr_inference_engine *next;
};
```

Figure 58. ASR inference engine snippet

The inference engine structure in Figure 58 consists of the following members:

- **iWhoAmI_inf**: inference engine type. It indicates either the wake word or the command inference engine. Developers can redefine or add from the list in Table 10.
- **iWhoAmI_lang**: language type information for an inference engine. The type definition is described in Table 9.
- **handler**: handler for an inference engine.
- **nGroups**: the number of groups for an inference engine. The default value is 2, as each OOB demo consists of base model (counted as a group) plus wake word group or command group n.
- **addrGroup[2]**: base + keyword group (either ww or cmd).
- **addrGroupMapID**: the address that contains the MapIDs.
- **idToKeyword**: the string list that indicates which command/wake word is detected. The string is printed out in a terminal window.
- **memPool**: memory pool address in SRAM for an inference engine.
- **memPoolSize**: size of an inference engine in a memory pool.
- **next**: pointer to the next inference engine in a linked list. If it is not in a linked list, the value should be NULL.

Table 10. Inference engine types

Inference engine type	asr_inference_t code	Encoding
Unknown	UNDEFINED_INFERENCE	0x0000
Wake Word	ASR_WW	0x0001
Commands for Smart Home (IoT)	ASR_CMD_IOT	0x0002

Table continues on the next page...

Table 10. Inference engine types (continued)

Inference engine type	asr_inference_t code	Encoding
Commands for Elevator	ASR_CMD_ELEVATOR	0x0004
Commands for Audio Device Control	ASR_CMD_AUDIO	0x0008
Commands for Washing Machine	ASR_CMD_WASH	0x0010
Commands for LED Control	ASR_CMD_LED	0x0020
Commands for Dialog Stage 1	ASR_CMD_DIALOGIC_1	0x0040
Commands for Dialog Stage 2 Temperature	ASR_CMD_DIALOGIC_2_TEMPERATURE	0x0080
Commands for Dialog Stage 2 Timer	ASR_CMD_DIALOGIC_2_TIMER	0x0100

7.1.3 ASR configuration

7.1.3.1 Languages

The SLN-LOCAL2-IOT with i.MX RT106S can support up to four languages in runtime. If customized hardware or proof-of-concepts are created, ensure the maximum number of languages to be enabled.

```
#define MULTILINGUAL            (1)
#define IMXRT105S               (0) // Not supported yet
#define MAX_INSTALLED_LANGUAGES (4)

#if MULTILINGUAL
#if defined(SLN_LOCAL2_RD)
#define MAX_CONCURRENT_LANGUAGES 3
#elif defined(SLN_LOCAL2_IOT)
#define MAX_CONCURRENT_LANGUAGES 4
#endif // defined(SLN_LOCAL2_RD)
#else
#define MAX_CONCURRENT_LANGUAGES (1)
#endif // MULTILINGUAL
```

Figure 59. Configuration for the maximum number of languages snippet

- If i.MX RT105S is considered, it can support up to two languages. Set IMXRT105S to 1.
- If only one language is sufficient, we always recommend to set MULTILINGUAL to 0. This allows the ASR application to save significant memory and CPU resources.
- Because MAX_CONCURRENT_LANGUAGES affects the resources, three microphones can be used when one or two languages are enabled. Only two microphones are enabled for three or four languages.

7.1.3.2 Installation of languages and inference engines

Developers must ensure that the language models and inference engines are properly installed when initializing the ASR. Figure 60 shows the language model installation function.

The **install_language()** function registers a language model in the ASR control structure. It unpacks the model binary from the flash memory and assigns addresses and parameters into the linked list of language models.

The reference example is implemented in **sln_local_voice.c**.

```
Int32_t install_language(asr_control_t *pAsrCtrl,
                         struct asr_language_model *pLangModel,
                         asr_language_t lang,
                         unsigned char *pAddrBin,
                         uint8 t nGroups)
```

Figure 60. Function install_language() snippet

- **pAsrCtrl**: ASR control structure, also shown in Figure 56.
- **pLangModel**: language model, also shown in Figure 57.
- **lang**: languages to be enabled. The types and encodings are listed in Table 9.
- **pAddrBin**: address of the language model binary to be installed.
- **nGroups**: total number of groups (base + wake word + *N* command groups) where *N* depends on applications.

The **install_inference_engine()** function registers an inference engine (either for a wake word or command group *n*) in the ASR control structure. It assigns a language's base model and a wake word (or command) group from the language model to an inference engine.

```
uint32_t install_inference_engine(asr_control_t *pAsrCtrl,
                                  struct asr_inference_engine *pInfEngine,
                                  asr_language_t lang,
                                  asr_inference_t infType,
                                  char **idToString,
                                  unsigned char *addrMemPool,
                                  uint32_t sizeMemPool)
```

Figure 61. Function install_inference_engine() snippet

- **pAsrCtrl**: ASR control structure, also shown in Figure 56.
- **pInfEngine**: inference engine, either for a wake word or a command group, also shown in Figure 58.
- **lang**: languages to be enabled. The types and encodings are in Table 9.
- **infType**: either a wake word or a command group, also shown in Table 10.
- **idToString**: the string list that indicates which command/wake word is detected. The string is printed out in a terminal window.
- **addrMemPool**: memory pool address in SRAM for an inference engine.
- **sizeMemPool**: size of an inference engine in a memory pool.

After the installation, the inference engines initialize their handlers. The handler within each instance should not be NULL, if the installation and initialization are successful.

7.1.4 ASR session control

7.1.4.1 Follow-up mode

The SLN-LOCAL2-IOT ASR session supports the follow-up mode where you can continue saying voice commands after the wake word is triggered once. For example, with the elevator voice control application, multiple passengers who go to different floors can say a voice command one by one after the first passenger triggers the device with a wake word. The mode is configured by the shell command in a serial terminal. You can see the command usage by typing "help", as shown in Figure 62.

```
"followup": Set follow-up mode (on / off). Save in flash memory.
        Usage:
           followup on (or off)
        Parameters
           on or off
```

Figure 62. ASR session control - follow-up mode

7.1.4.2 Timeout

After the device is triggered by a wake word, it waits for the user's voice command for a certain amount of time. Users can configure the response waiting time using a shell command in a serial terminal. Figure 63 shows the command usage.

```
"timeout": Set command waiting time (in ms). Save in flash memory.
        Usage:
           timeout N
        Parameters
           N milliseconds
```

Figure 63. ASR session control - timeout

7.1.4.3 Push-to-Talk (PTT) mode

In some applications, you may want to bypass the wake word detection stage. The SLN-LOCAL2-IOT kit offers the PTT feature. If it is enabled, you can directly say voice commands after pressing the SW1 button on the device. Figure 64 shows the command usage.

```
"ptt": Set push-to-talk mode (on / off). Save in flash memory.
        Usage:
           ptt on (or off)
        Parameters
           on or off
```

Figure 64. ASR session control - PTT mode

7.2 Acoustic modification

7.2.1 Changing microphone configuration

Open the **local _demo** variant of the project in the developer's environment. To change the number of microphones supported, open the **config_files** folder and the **pdm_pcm_definintions.h** file, as shown in Figure 65.

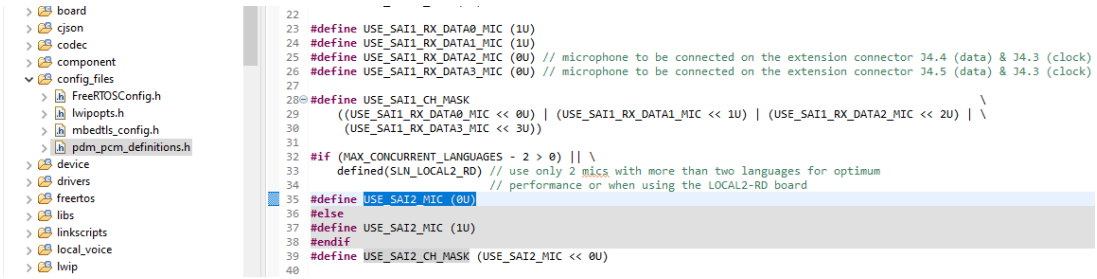


Figure 65. pdm_pcm_definitions.h file and USE_SAI2_MIC define

The **pdm_pcm_definitions.h** header file contains the whole configuration for the SAI data line to use. To switch between three or two microphones, set the **USE_SAI2_MIC** to **"0U"** (two microphones) or **"1U"** (three microphones). This propagates throughout the firmware and configures the audio front end.

To switch between two or three microphones, set the **USE_SAI2_MIC** as follows:

- "**#define USE_SAI2_MIC (1)**" – three microphones used.
- "**#define USE_SAI2_MIC (0)**" – two microphones used.

7.2.2 Changing the post gain

The post gain can affect how the board performs in low noise using a low-volume voice (far-field). In the default configuration, this setting is set to the maximum value. In real life, this can cause the device to wake up more often than other devices, which can generate problems if the devices in the consumer's home are less capable.

Table 11. u16PostProcessedGain description

Parameter	postProcessedGain		
Description	Acoustic signal digital gain in a linear scale up to x64. The signal level after applying this gain is still under the Dynamic Range Control (DRC) constraint, so it is not used as a traditional DRC make-up gain.		
Data type	**Data range**	**Unit**	**Default**
uint16_t	[0x0000,0x4000]	N/A	0x0600

Inside the **local_demo** or **usb_aec_alignment_tool** project, navigate to **audio_processing_task.c**, as shown in Figure 66.

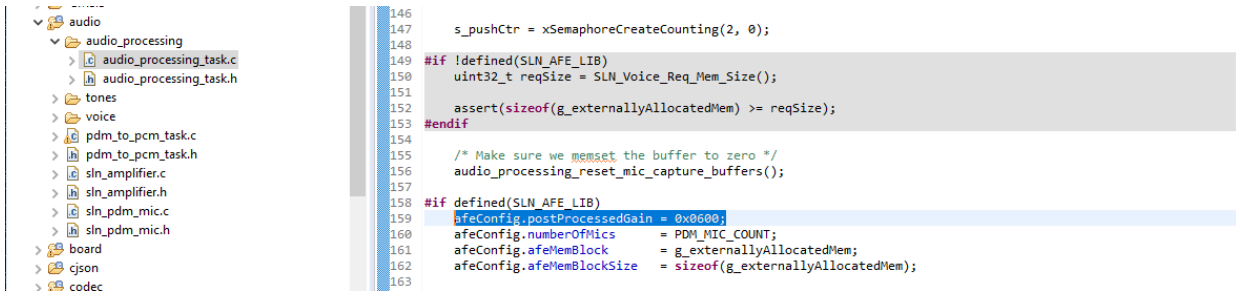


Figure 66. Gain variable in audio_process_task.c

Inside the **audio_processing_task.c** file, locate the code snippet from Figure 66 and adjust the post processing dynamic gain variable "**afeConfig.postProcessedGain**" corresponding to the signal level needed.

7.2.3 Changing the pre-processed microphone gain

The i.MX RT106S MCU does not have a PDM hardware block, which means that it is required to convert the PDM data to PCM within software. This allows the gain to be adjusted to fit the needs of the developer. The gain may be required to change because some product designs may have a higher/lower Echo Return Loss (ERL) which can affect the barge-in performance. See Audio Performance Requirements for Audio Front End of i.MX RT106A/L for more information. The microphone gain is also modified before the DC offset is applied, allowing for higher shift precision. To change the gain, open the file located in the **local_demo** or **usb_aec_alignment_tool** folders in the **audio** subfolder called **pdm_to_pcm_task.c**.

Table 12. SLN_DSP_SetGainFactor function description

Function	SLN_DSP_SetGainFactor
Description	Modifies the PCM gain before the DC Offset is applied. This increases the gain factor but it can also cause clipping.

Table continues on the next page...

Table 12. SLN_DSP_SetGainFactor function description (continued)

Parameter	Data range	Type	Description
memPool	N/A	uint8_t **	Pointer to the memory pool that the sln_intelligence_toolbox needs.
gainFactor	0x0000-0xffff	int16_t	The shift gain factor before the DC Offset is applied. A gain of zero means that there is no gain.

Locate the code snippet shown in Figure 67 and change the numerical value according to your needs, related to the table.

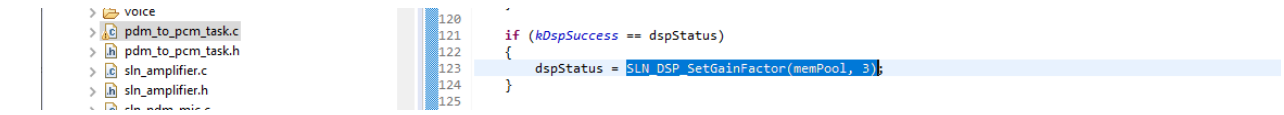


Figure 67. pdm_to_pcm_task.c set gain factor

Chapter 8
Security architecture

8.1 Application chain of trust

The basis of the security architecture implemented in the SLN-LOCAL2-IOT are the signed application images. The signing requires the use of a Certificate Authority (CA). NXP has its own CA to sign applications in the factory, but the CA is not shared with customers.

The CA is used to create signing entities for the bootloader and application, as shown in Figure 68. The certificate from the CA is stored in the SLN-LOCAL2-IOT's filesystem and used to verify the signatures of the signing entity certificates. In addition, the locally stored certificates from the signing entities are used to verify the signature of firmware images coming in the Over-the-Air (OTA) or Over-the-Wire (OTW) bootloader interfaces.

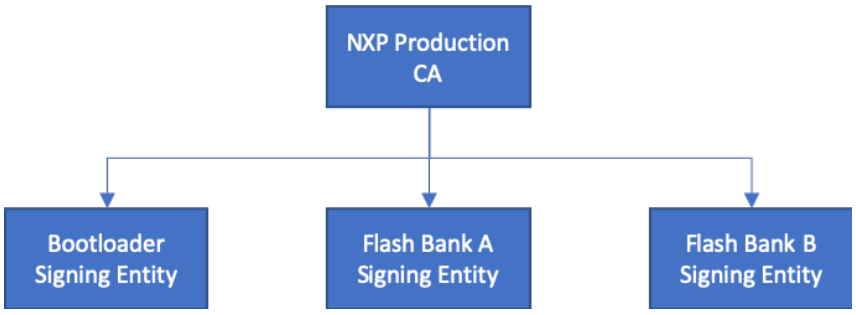


Figure 68. Application chain of trust

When creating new firmware images for a secure boot implementation, the **Automated Manufacturing Tool (Ivaldi)** can be used alongside your unique CA.

8.2 FICA and image verification

The FICA table is a section inside the filesystem that describes the images that will be booted. It contains information about the image and signatures of the applications used to ensure that only verified firmware is executed. This ensures that malicious images cannot be executed without 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, the standard boot flow (Figure 15) is as follows:

- The **bootstrap** uses the bootloader FICA entry to validate the bootloader.
- The **bootloader** uses the AppA FICA entry to validate the AppA image.
- The **bootloader** uses the AppB FICA entry to validate the AppB image.

For final production, the solution provides programming scripts to enable the i.MX RT High Assurance Boot (HAB) to verify and protect the bootstrap component. Enable the HAB for your end product. The downside of having this security protection enabled is that programming new images can be a little more complex, because it requires signature generation. Because this flow may be time consuming and not required for basic development tasks, NXP introduced some bypasses to make the job easier.

NOTE

These bypasses should not be deployed in production.

8.3 Image Certificate Authority (CA) and application certificates

The SLN-LOCAL2-IOT kit comes pre-programmed with signed images, as explained in FICA and image verification. The bootloader and demo applications are signed using NXP's test CA and they can be used to ensure that all images that are to be booted are authentic.

The application signing certificates are located at the following addresses 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.

Chapter 9
Bootloader

The SLN-LOCAL2-IOT SDK enables three forms of firmware update capability:

1. USB Mass Storage Device (MSD) interface
2. Over-The-Air (OTA) *via Wifi*
3. Over-The-Wire (OTW) *via UART*

The boot flow is described in detail in Understanding the boot flow. When the boot flow reaches the bootloader, it must decide whether to jump to the main application (i.e. local_demo) or to the firmware update mode. Figure 69 shows the four options available to the bootloader. The bootloader reroutes the boot flow to the main application, MSD, OTW, or OTA update. This section explains how to generate a BIN file to be updated. Then it describes MSD, OTW, and OTA.

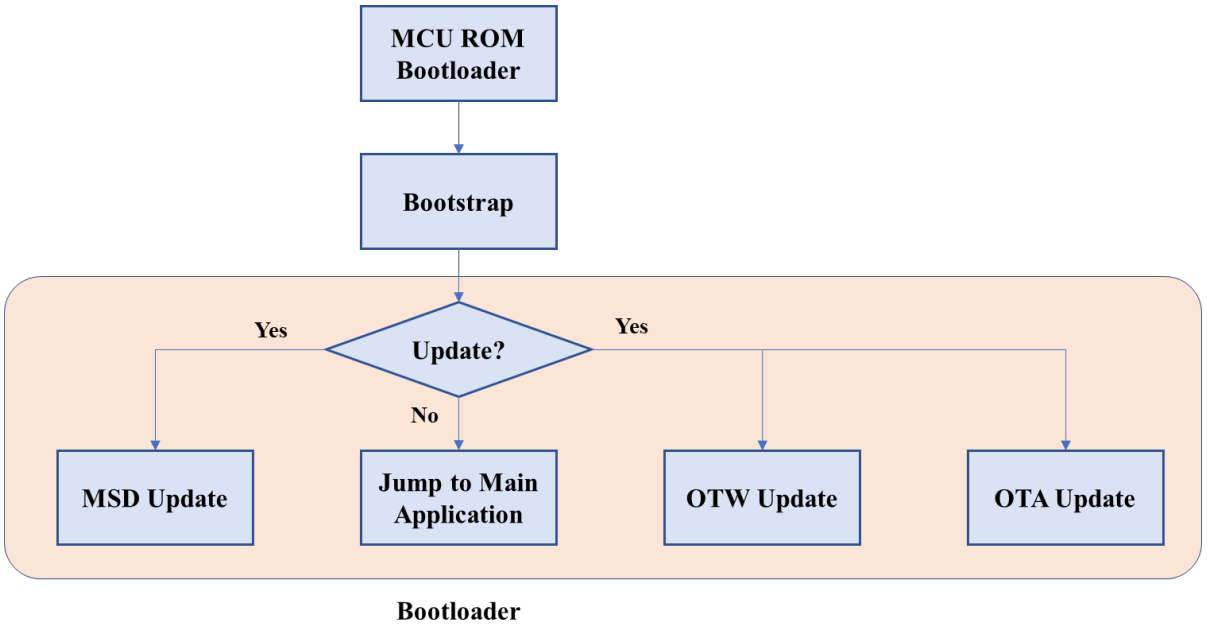


Figure 69. Bootloader flow

NOTE

The firmware update in the bootloader is only for the main application, not for the bootstrap and bootloader. If the bootstrap or bootloader must be updated, use the J-Link probe or the Ivaldi tool described in Automated manufacturing tools.

9.1 Application BIN file generation

There are two application banks in the flash memory, see Table 4, on the SLN-LOCAL2-IOT kit.

- Address for Application Bank A: 0x60300000
- Address for Application Bank B: 0x60D00000

Developers must configure the bank address properly when the main application is compiled. This ensures that the device is safe to jump into a new application image in one memory location without compromising the other one. If the application runs in Bank A, the new application image must be linked to Bank B.

To change the address from Bank A to Bank B, in the MCUXpresso IDE project explorer, right-click **sln_local2_iot_local_demo** (or the developer's application project name) > **Project Settings** > **Memory**, as shown in Figure 70.

Select **Edit Memory**, which opens the **Memory Configuration Editor**.

Change the address of the Flash type to **0x6030 0000** for **Application Bank A** and **0x60D0 0000** for **Application Bank B**.

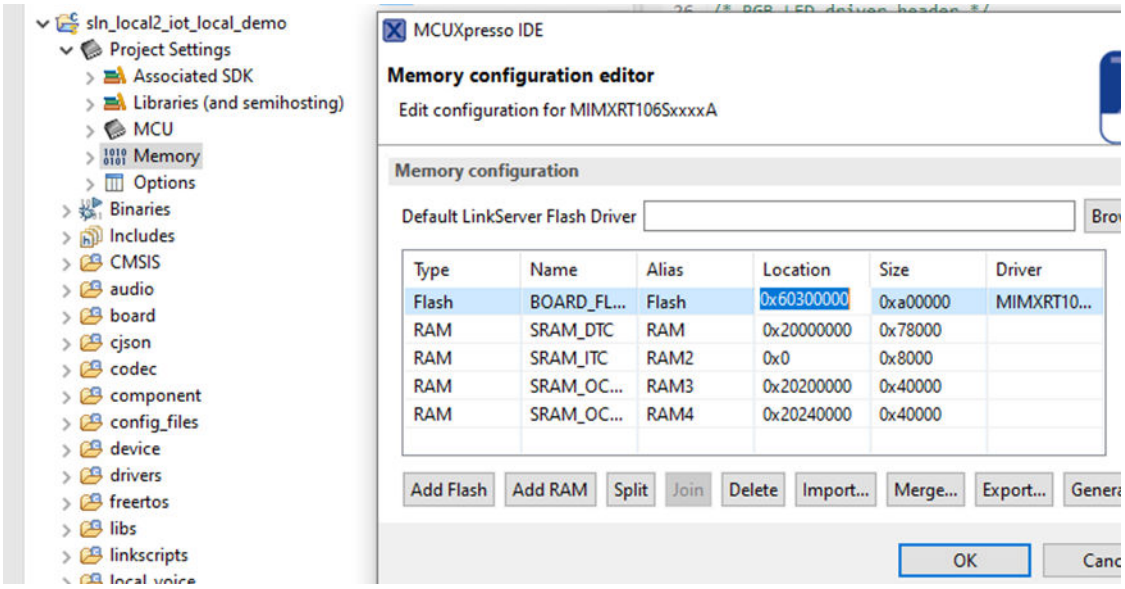


Figure 70. Editing memory configuration

Before building the application, make sure that the MCUXpresso project generates a **BIN** file as an outcome of the build process. Right-click the project name and open **Properties**, as shown in Figure 71.

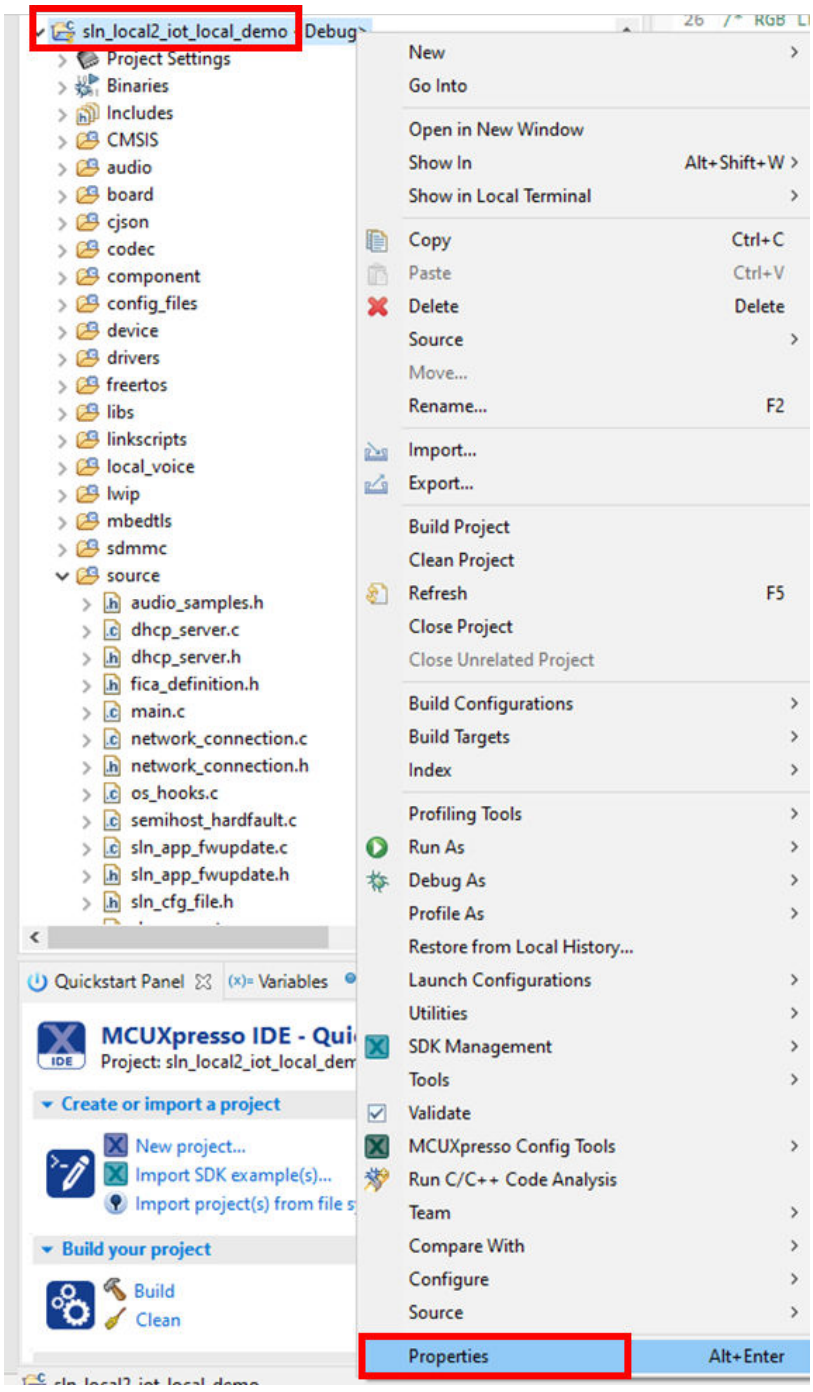


Figure 71. Project properties

Expand **C/C++ Build** in the menu and click **Settings**. Select the **Build steps** tab, where the **Post-build steps** can be edited. Click **Edit** and it shows the commands for the post-build steps. Figure 72 shows how to open the **"Post-build steps"** window.

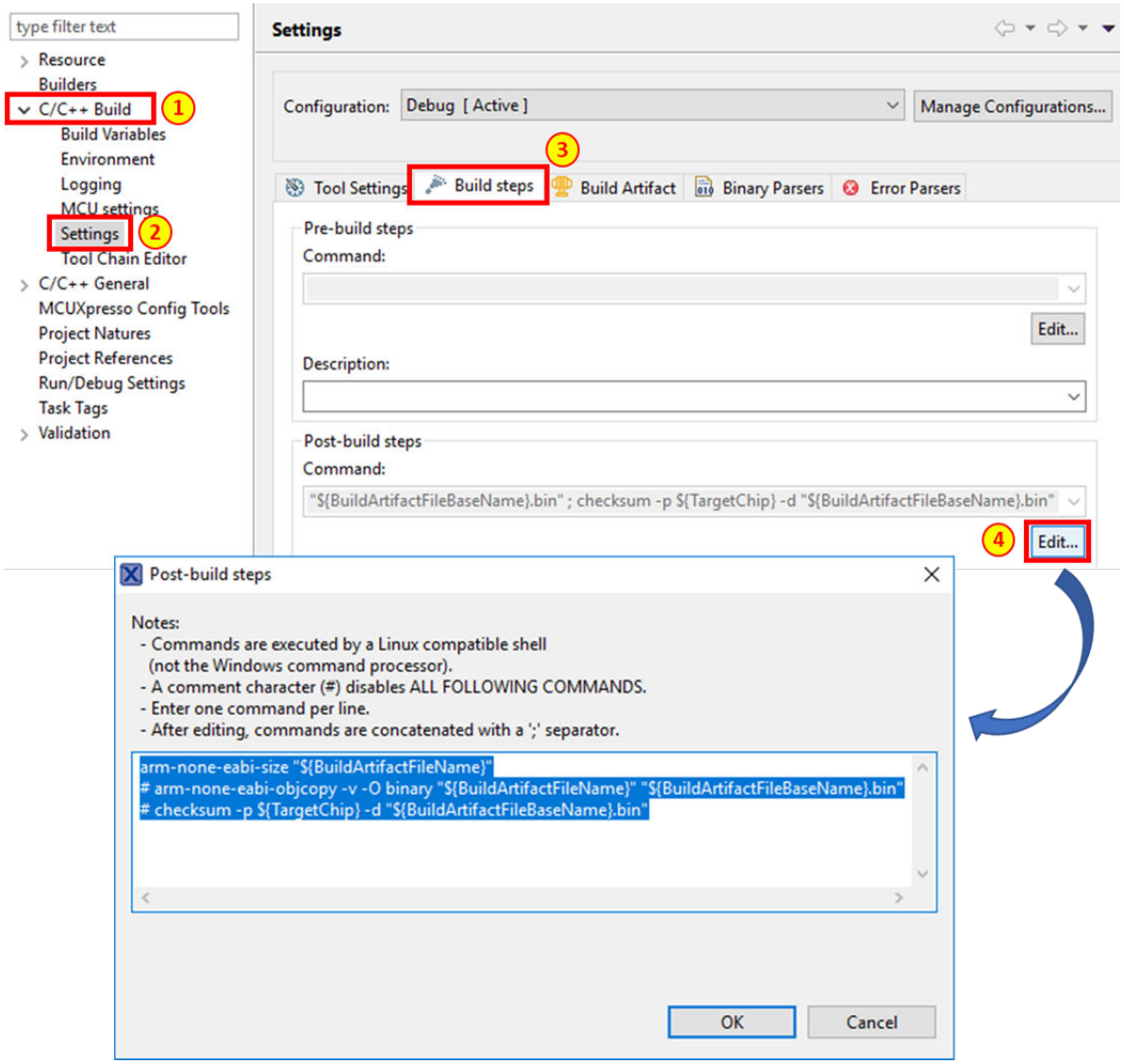


Figure 72. Editing post-build steps

The **"#"** command character disables all the following commands. To generate a **BIN** file in the post-build process, remove the **"#"** character on the second line and click **OK**. The resulting commands after removing **"#"** are shown in Figure 73:

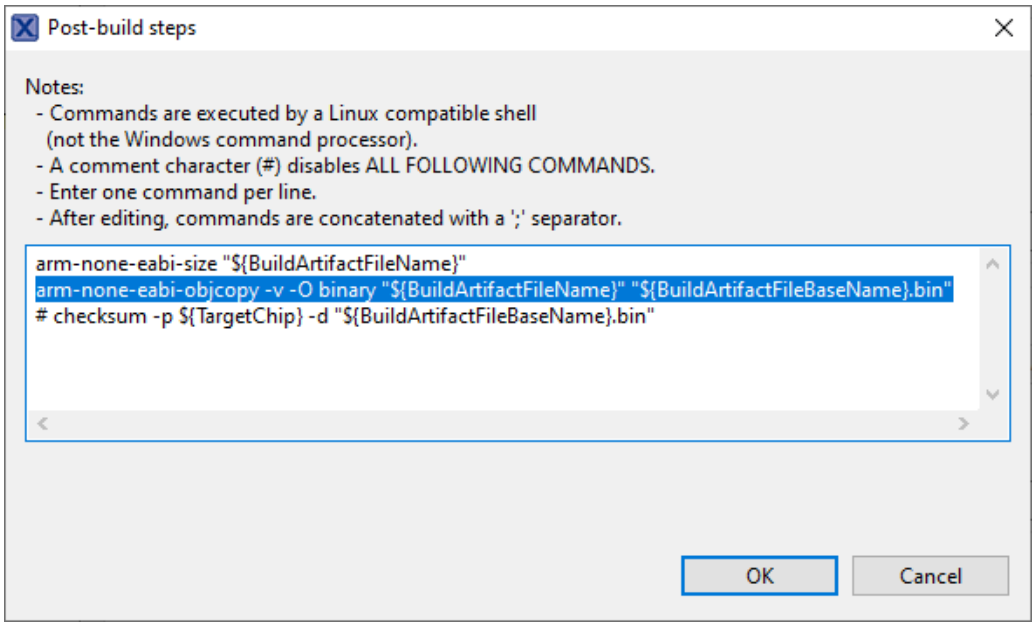


Figure 73. Post-build commands to generate BIN file

If the build process is done successfully, a **BIN** file is generated and placed in the **Debug** folder of the MCUXpresso project.

9.2 USB Mass Storage Device (MSD) update

The bootloader application supports firmware update over the USB MSD. This allows the user to re-flash the main application binary (not the bootstrap nor bootloader) without a J-Link probe. The bootstrap or bootloader must be updated using a J-Link probe.

By default, the MSD feature bypasses the signature verification to allow for easier development flow, because signing of images can be a process not suitable for quick debugging and validation.

NOTE

Bypassing the image verification is a security threat and it is the responsibility of the product designer to prevent the violation in production.

To put the device into the MSD mode, hold down switch 2 (SW2) and power cycle the board until the pink LED (D2) is lit. The pink LED turns on and off in 3-second intervals.

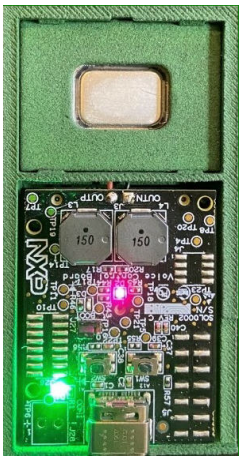


Figure 74. MSD update mode LED

Navigate to the PC's file explorer and ensure that the SLN-LOCAL2-IOT kit is mounted as a USB MSD. A mounted kit is displayed in the file explorer, as shown in Figure 75.

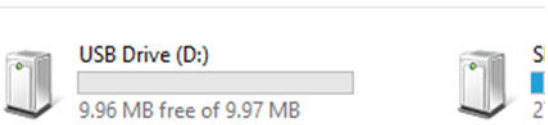


Figure 75. SLN-LOCAL2-IOT kit mounted as USB MSD

Drag and drop the generated **BIN** file for banks A or B into the MSD drive. This starts the download process and writes the **BIN** file to the flash. After the image is programmed into the flash, it starts to execute.

9.3 Over-the-Air (OTA) and Over-the-Wire (OTW) updates

The bootloader supports several mechanisms to update the board's firmware. The OTA/OTW updates are part of them. The OTA and OTW updates are driven using a simple JSON interface, making it easy to implement the host-side code. The mechanism for both OTA and OTW is the same. The only difference between the two is that the OTA update is performed over the air via Wi-Fi and the OTW update is performed over UART. The OTA update interface is performed by hosting a TCP server on the kit which receives the update-related JSON packets. The OTW update currently supports UART, but it can be extended to support any serial interfaces, including SPI, TCP sockets, or even I^2C. The OTA/OTW update method is described in more depth in this section.

9.4 Transfers

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

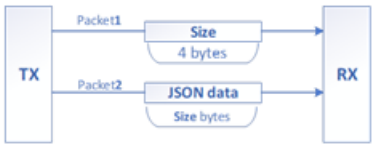


Figure 76. Transfer format

There are two types of messages passed: requests and responses. Figure 77 shows the request and response flow.

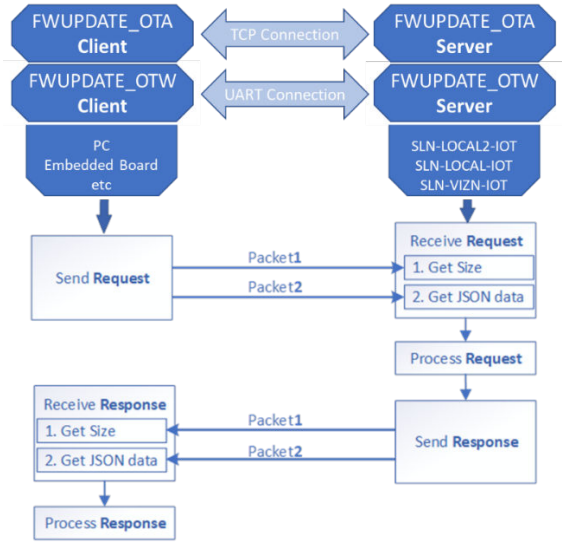


Figure 77. Request and response flow

9.4.1 JSON messages

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
6. Clean

Each transfer is followed by a transfer response.

9.4.1.1 Start request

The following 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>,
        }
    }
}
```

9.4.1.2 Block request

Block requests are sent for each “chunk” of data to be programmed. The block sizes can be of any size, though it is best when they are as large as possible. The example script in the SDK sends 4096 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>,
        }
    }
}
```

9.4.1.3 Stop request

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

9.4.1.4 Activate image request

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

9.4.1.5 Start self-test request

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

9.4.1.6 Clean request

```
{
    "messageType":1,
    "fwupdate_message": {
        "messageType":0,
        "fwupdate_common_message": {
            "messageType":2
        }
    }
}
```

9.4.1.7 Response format

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

9.5 Testing OTA/OTW updates

To demonstrate the OTA and OTW updates, use the Python test script in **sln_local2_iot_bootloader/unit_tests/fwupdate_client.py**. From a high-level perspective, this script “JSONifyes” a specified binary and flashes it either via the OTA or OTW mechanism, depending on the update method specified as an argument to the script. This method allows you to flash the main app binary without a J-Link. NXP provides OTA/OTW update tools which are currently intended as a unit test. These tools are not intended to be used for production environment in the current release.

While the use of OTA and OTW is nearly identical, the setup is slightly different between the two, because the OTA update requires a network connection and the OTW update requires a serial connection.

9.5.1 OTA setup

This section describes the steps necessary to perform an OTA update. To perform an OTA update using the test script, the SLN-LOCAL2-IOT kit must be connected to a Wi-Fi network and the proper bit in the FICA table must be set to indicate to the bootloader that an OTA update is being expected. The update bit is set by the test script. A TCP server is running in the main application and waits for a JSON sent by the script to set the FICA bit.

The SLN-LOCAL2-IOT kit and the client running the Python script must be on the same network for the OTA to work.

Connecting to a Wi-Fi network:

- Connect the board to the computer via the USB Type C port.
- Open your favorite serial monitor and connect to the board COM using the 115200 baud rate.
- In the serial monitor, type the following command: “**setup *SSID PASSWORD***”.
- The board will reset and then it will try to connect to the Wi-Fi network.
- If the connection is successful, the IP adress of the board is displayed in the serial monitor. Write it down.

9.5.2 OTW setup

To perform an OTW update using the test script, the SLN-LOCAL2-IOT must be connected via UART and the proper bit in the FICA table must be set to indicate to the bootloader that an OTW update is being expected. The update bit is set by the ***“updateotw”*** command in the shell.

Note that the USB CDC is already connected to the kit and supplying power as well as the serial communication. To run the OTW via UART, it requires an additional UART module connected to header J26 on the kit. Make sure that TX, RX, Vcc, and GND are connected properly. The schematic of header J26 is shown in Figure 78.

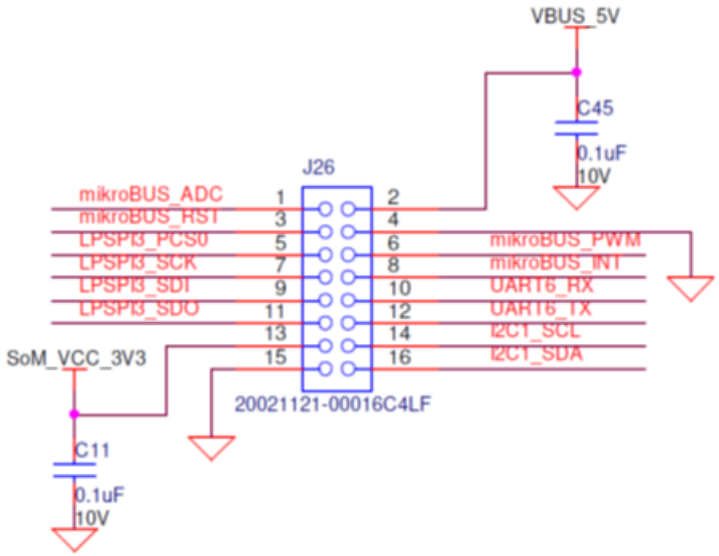


Figure 78. UART port header - J26

9.6 Running the test script

The test script is written in Python. It is recommended to set a virtual environment:

- python3 -m venv env
- source **env/bin/activate** (for Linux OS) or **.\env\Scripts\activate** (for PowerShell)

To run the test script, install the following modules:

- python3 -m pip install pyserial
- python3 -m pip install libscrc

This script is tested on Python 3.8.5 running in Windows OS and Linux OS. To start the update, open the **fwupdate_client.py** script in the **sln_local2_iot_bootloader/unit_tests** folder in a terminal. Running the script without any arguments shows the arguments that the script takes.

```
~/sln_local2_iot_bootloader/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
```

The script requires you to specify the following:

1. Device: local (equivalently applied to local2)
2. Method: OTA/OTW
3. Bank: A/B (see Application BIN file generation)
4. Appfile: binary for the **sln_local2_local_demo** project (see Building and programming with MCUXpresso)

5. appsignFile: none/**sln_local2_local_demo.sha256.txt** (see NXP application image signing tool)

When all args are specified, the script outputs the following:

```
~/ sln_local2_iot_bootloader/unit_tests $: python3 fwupdate_client.py local OTA B PATH-TO-BIN/
bundle.sln_local2_iot_local_demo_bankB.bin sln_local2_local_demo.sha256.txt

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

You must input the board IP address that was displayed in the serial monitor after the connection. The script connects to the board and sends a request to the TCP server so that it will set the FICA bit and reset the board into the OTA update mode.

When the board is in the OTA update mode, the transfer starts:

```
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
```

Upon completion, the SLN-LOCAL2-IOT kit restarts itself automatically and switches over to the new application bank, running the new application that was just flashed.

The OTA update sets the FICA bit and triggers the board by running the **fwupdate_client.py** script, whereas the OTW update must set the FICA bit using the **updateotw** shell command. For the OTW, make sure that the **COM_PORT** matches the target com port address.

```
if FWUPDATE_METHOD == "OTA":
    IP_ADDRESS=input('Device IP:')
    if(check_ip(IP_ADDRESS)):
        print("Invalid IP Address")
        sys.exit(1)
    PORT=8889
elif FWUPDATE_METHOD == "OTW":
    # If working on Windows, change this to the appropriate COM
    COM_PORT='/dev/ttyUSB0'
```

Chapter 10
Filesystem

The SLN-LOCAL2-IOT implements a custom filesystem to manage files with HyperFlash on the kit. The reasons why a custom filesystem is chosen are as follows:

1. The device executes code from the flash (XiP), which means that it must read the flash from RAM functions.
2. HyperFlash has a 256-KB sector size, which does not allow for file granularity.
3. The update-in-place features are added to allow the update of big sectors without a time-consuming erase.

10.1 Generating filesystem-compatible files

Within the Ivaldi package, there is a script that converts any file into a filesystem supported file. Any file that gets programmed to the filesystem must first pass through this script. This is required of all certificates and keys as well as any other files that the reader needs.

Within the "**Scripts/sln_iot_utils**" folder of the release package, there is a Python script called **file_format.py** which is responsible for creating a binary file formatted for the firmware's filesystem. This script accepts the following parameters:

- "-if" parameter - passes the input file to be converted for the embedded filesystem
- "-of" parameter - passes the output file name
- "-ft" parameter - passes the flash type of the board; the acceptable values are as follows:
  - "-ft H" for HyperFlash (used for SLN-LOCAL2-IOT)
  - "-ft Q" for QSPI Flash (used for future platforms based on QSPI)

For SLN-LOCAL2-IOT, the **file_format.py** script should be called with "-ft H" parameter, because the platform has HyperFlash. For example, you can run the command below. This will generate the binary file to be flashed into the device.

An example of running the **file_format.py** to convert the **"../ota_signing/ca/certs/<cert_name>.ca.crt.pem"** file to the **"<cert_name>.ca.crt.pem.bin"** file suitable for the HyperFlash filesystem is in Figure 79.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/sln_iot_utils$
python3 file_format.py -h
usage: file_format.py [-h] -if IN_FILE -of OUT_FILE -ft FLASH_TYPE

optional arguments:
  -h, --help            show this help message and exit
  -if IN_FILE, --in-file IN_FILE
                        Specify input file name
  -of OUT_FILE, --out-file OUT_FILE
                        Specify output file name
  -ft FLASH_TYPE, --flash-type FLASH_TYPE
                        Specify board`s flash type. Add [-ft H] for HyperFlash
                        and [-ft Q] for QSPI
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/sln_iot_utils$
python3 file_format.py -if ../ota_signing/ca/certs/my_prod.root.ca.crt.pem -of my_prod.root.ca.crt.p
em.bin -ft H
Converting for HyperFlash
File size 2010
File CRC 0x2525ad9a
```

Figure 79. file_format.py script description, usage, and logs

10.2 Generating new audio playback files

The custom filesystem for HyperFlash limits its size to 256 KB per file, which includes the file header as well as the sound data. Therefore, make sure that the audio file is smaller than 256 KB.

Before generating the binary files, simply create a 16-bit, 48-kHz audio file. The current configuration of the amplifier only supports 48-kHz playback.

Chapter 11
Automated manufacturing tools

NXP provides a package of scripts that can be used for manufacturing programming and reprogramming of devices on the production line without the J-Link. This collection of scripts is called Ivaldi. The Ivaldi package allows developers to program all the required firmware binaries into a flash device using a single command.

11.1 Introduction

11.1.1 About Ivaldi

Ivaldi is a package of scripts responsible for manufacturing and reprogramming devices without J-Link. The Ivaldi package uses the serial downloader mode of the i.MX RT106S's boot ROM to communicate with an application called ***Flashloader*** that is programmed in the i.MX RT106S MCU. This 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 setting up a new device/product with certificate/key pairs
- Enabling High Assurance Boot (HAB)
- Signing images for OTA, OTW, and HAB
- Writing and accessing One Time Programmable (OTP) fuses

11.1.2 Download the package

The Ivaldi package can be downloaded from www.nxp.com/mcu-local2.

Extract the **ZIP** file and open the **README.md** file located in the root folder of the package to set up the environment.

11.1.3 Requirements

The following requirements must be met to run Ivaldi. It is tested with Windows, Mac, and Linux operating systems.

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

11.1.4 Platform configuration

The Ivaldi package uses the **Scripts/sln_local2_iot_config/board_config.py** configuration file, which includes the following:

- Names of the binaries to be flashed (from the **Image_Binaries** folder):
  - BOOTSTRAP_NAME
  - BOOTLOADER_NAME
  - MAIN_APP_NAME
  - Names of audio playback binaries
- Flash configurations:
  - FLASH_TYPE
  - FLASH_START_ADDR

- FLASH_SIZE

- Flash map:
  - Binaries' images addresses
  - Certificates' addresses

If specific image binaries (such as the main application or audio playback) must be updated in the Ivaldi package's **Image_Binaries** folder, make sure that the **Scripts/sln_local2_iot_config/board_config.py** configuration file has correct file names and addresses.

NOTE

Any changes in **Scripts/sln_local2_iot_config/board_config.py** (except for binaries' names) also require updating the embedded code and configurations.

11.1.5 Boot programming modes and security features

The Ivaldi package supports multiple boot settings with various security features. In the **open boot programming**, the HAB is disabled. In the **secure boot programming**, the HAB is enabled. There are various security feature options with (or without) signing certificates. Table 13 summarizes the security features for the open and secure boot modes. By default, the SLN-LOCAL2-IOT kit is enabled with image verification in the open boot mode.

Table 13. Summary of boot mode and security features

Boot mode/ security features	HAB	Signing certificates (image verification)	
Open boot	No	No	Recommended only for development
	No	Yes	Default
Secure boot	Yes	Yes	Most secure

11.2 NXP application image signing tool

The signing tool is a Python application that is responsible for using a signed Certificate Signing Request (CSR) to sign the binaries and append the certificate to the binary ready to be deployed to OTA/OTW services.

The following instructions assume that the README file in the Ivaldi root directory is followed to set up the Python virtual environment. If this is not done, the scripts fail.

To start, navigate to the "**Scripts/ota_signing**" directory inside Ivaldi.

11.2.1 Generating signing entity

The Ivaldi tools provided by NXP include the CA, but the end users must create their own CA and signing artifacts. For information about the chain of trust used by NXP from the factory, see Application chain of trust.

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

In the Python virtual environment, navigate to **Scripts/ota_signing**. Run the **generate_signing_artifacts.py** script. When running without any arguments, the usage is displayed.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing$
python3 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
```

Figure 80. Signing artifact generation usage

Type in the **ca_name** (e.g., **my_prod**) and the rest of the information for the CA. The CA name is the name given to the CA chain that will be used to sign the images. Developers can always re-generate a more secure CA when preparing for production. When prompted for passwords for the PEM files, use the same password for all of them for this exercise. Figure 81 shows an excerpt from the terminal output of the generation script.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing$
python3 generate_signing_artifacts.py my_prod US Texas Austin NXP
Creating directories...
Creating directories...
['mkdir', 'certs', 'crl', 'newcerts', 'private', 'csr']
SUCCESS: Successfully prepared the directories
chmod directories...
['chmod', '700', 'private']
SUCCESS: Successfully prepared the directories
creating index file...
['touch', 'index.txt', 'serial', 'crlnumber', 'index.txt.attr']
SUCCESS: Successfully prepared the directories
Creating Serial File...
Modifying contents for local path...
SUCCESS: openssl.cnf copied.
Creating Root Key...
Enter pass phrase for private/my_prod.root.ca.key.pem:
Verifying - Enter pass phrase for private/my_prod.root.ca.key.pem:
SUCCESS: Created Root Key
Changing Root Key Permissions...
SUCCESS: Changed Root Key Permissions
Creating Root Certificate...
Enter pass phrase for private/my_prod.root.ca.key.pem:
SUCCESS: Created Root Certificate
Changing certificate permissions...
SUCCESS: Changed certificate permissions
Creating Private Key...
Enter pass phrase for private/my_prod.app.a.key.pem:
Verifying - Enter pass phrase for private/my_prod.app.a.key.pem:
SUCCESS: Created private key
Changing Key Permissions..
SUCCESS: Changing Key Permissions
Creating Certificate..
Enter pass phrase for private/my_prod.app.a.key.pem:
SUCCESS: Creating Certificate
Sign the CSR..
Enter pass phrase for /mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing/ca
/private/my_prod.root.ca.key.pem:
SUCCESS: Signed the CSR
Modifying certificate permissions...
SUCCESS: Modifed the certificate permissions
Creating Private Key...
Enter pass phrase for private/my_prod.app.b.key.pem:
Verifying - Enter pass phrase for private/my_prod.app.b.key.pem:
SUCCESS: Created private key
Changing Key Permissions..
SUCCESS: Changing Key Permissions
Creating Certificate..
Enter pass phrase for private/my_prod.app.b.key.pem:
SUCCESS: Creating Certificate
Sign the CSR..
Enter pass phrase for /mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing/ca
/private/my_prod.root.ca.key.pem:
SUCCESS: Signed the CSR
Modifying certificate permissions...
SUCCESS: Modifed the certificate permissions
```

Figure 81. Signing artifact generation excerpt

When **generate_signing_artifacts.py** succeeds, the **ca** folder is generated. This folder contains the certificates and private keys with the user-defined <ca_name>.

- ./ca/certs/
  - **<ca_name>**.app.a.crt.pem
  - **<ca_name>**.app.b.crt.pem

  - **<ca_name>**.root.ca.crt.pem

- ./ca/private/
  - **<ca_name>**.app.a.key.pem
  - **<ca_name>**.app.b.key.pem
  - **<ca_name>**.root.ca.key.pem

11.2.2 Installing the CA and application certificates

For the device to verify the image signature, the device must have the root CA and application certificates. Before programming it into the device, it must be converted into a binary format for the filesystem to use it. To do this, run the "**generate_image_crt_files.py**" script. Pass in the name of the generated CA in the command line.

Execute the command with the "-ft H" attribute for the HyperFlash and "-ft Q" for the QSPI, as shown in Figure 82.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing$
python3 generate_image_crt_files.py -h
usage: generate_image_crt_files.py [-h] -cn CA_NAME -ft FLASH_TYPE

optional arguments:
  -h, --help            show this help message and exit
  -cn CA_NAME, --ca-name CA_NAME
                        Specify CA name
  -ft FLASH_TYPE, --flash-type FLASH_TYPE
                        Specify board`s flash type. Add [-ft H] for HyperFlash
                        and [-ft Q] for QSPI
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing$
python3 generate_image_crt_files.py -cn my_prod -ft H
Generating bins for HyperFlash
File size 2010
File CRC 0x2525ad9a
File size 1904
File CRC 0x174de84b
```

Figure 82. Signing artifacts binary files generation for HyperFlash

The output of this script are two binary files – **ca_crt.bin** and **app_crt.bin**. Move these files to the **Image_Binaries** folder of the Ivaldi package, as shown in Figure 83. They will be programmed into the SLN-LOCAL2-IOT kit by a boot programming tool.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing$
mv app_crt.bin ../../Image_Binaries/
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/ota_signing$
mv ca_crt.bin ../../Image_Binaries/
```

Figure 83. Moving ca_crt.bin and app_crt.bin to Image_Binaries folder

11.3 Open Boot Programming tool

The Open Boot Programming tool is responsible for connecting to the device and programming it with the correct images and certificates. This method is a quick and easy way to take a device/product from the assembly line and prepare it for shipping. It is a good practice to run the Open Boot Programming script before enabling the HAB to ensure all images and artifacts are in working order. Before running the script, complete the following items:

- The files and folders shown in Figure 84 should be in the "**Image_Binaries**" folder in the Ivaldi root.

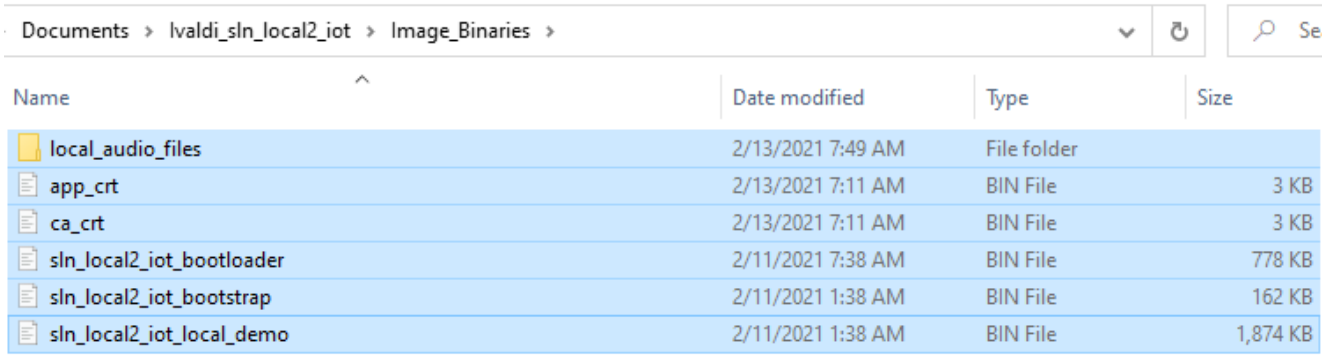


Figure 84. Files and folder for Open Boot Programming tool

- The file names should be properly configured, as in the "**Scripts/sln_local2_iot_config/board_config.py**" script.
- The "**Scripts/sln_local2_iot_open_boot/open_prog_full.py**" script should have the correct ca_name (e.g., **my_prod**), as below. The default name is "**prod**".

NOTE

End user will need to update the device signing entity used below (by default prod.app.a used).

```
# python sign_package.py -p PLATFORM_PREFIX -a prod.app.a
cmd = ['python', 'sign_package.py', '-p', board_config.PLATFORM_PREFIX, '-m', board_config.MAIN_APP_NAME, '-
bl', board_config.BOOTLOADER_NAME, '-a', 'my_prod.app.a', '-bc', board_config.ROOT_FOLDER]
```

- The SLN-LOCAL2-IOT kit must be put into the serial download mode. Make sure that jumper J27, which is located on the top of the kit, is in position "0".

To program the firmware and artifacts, execute the "**open_prog_full.py**" script which performs the following actions, as shown in Figure 85:

- Communicating with the BootROM and loading/executing the Flashloader
- Erasing the flash
- Programming the images (bootstrap, bootloader, local_demo, audio playback files, and FICA table) with certificates
- Jumping into the bootstrap and executing

After the script is executed, do not forget to exit the serial download mode, even though the board will boot into the application after the programming.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot/Scripts/sln_local2_iot_open_boot$
python3 open_prog_full.py
Importing board_config.py from ../sln_local2_iot_config/ folder
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 +m6metepMEk=
SUCCESS: Device thing name is _m6metepMEk
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 bootstrap...
write-memory
SUCCESS: Bootstrap written to flash.
Programming bootloader...
write-memory
SUCCESS: Bootloader written to flash.
Programming flash with root cert...
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with app cert application A...
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_prod.app.a.key.pem:
Enter pass phrase for ../ca/private/my_prod.app.a.key.pem:
Programming flash with AUDIO_EN_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_01 sound file for this "thing".
Programming flash with AUDIO_EN_02 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_02 sound file for this "thing".
Programming flash with AUDIO_ZH_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_ZH_01 sound file for this "thing".
Programming flash with AUDIO_ZH_02 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_ZH_02 sound file for this "thing".
Programming flash with AUDIO_DE_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_DE_01 sound file for this "thing".
Programming flash with AUDIO_DE_02 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_DE_02 sound file for this "thing".
Programming flash with AUDIO_FR_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_FR_01 sound file for this "thing".
Programming flash with AUDIO_FR_02 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_FR_02 sound file for this "thing".
Programming flash with AUDIO_EN_03 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_03 sound file for this "thing".
Programming flash with AUDIO_EN_04 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_04 sound file for this "thing".
Programming flash with AUDIO_EN_05 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_05 sound file for this "thing".
Programming flash with AUDIO_EN_06 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_06 sound file for this "thing".
Programming FICA table...
write-memory
SUCCESS: Programmed flash with FICA for this "thing".
Programming Application Bank A...
write-memory
SUCCESS: Application Bank A written to flash.
read-memory
SUCCESS: Application entry point at 0x60002451
read-memory
SUCCESS: Application entry point at 0x20208000
Attemping to execute application...
execute
SUCCESS: Application running.
```

Figure 85. Output of Ivaldi Open Boot Programming

11.4 Secure boot programming with High Assurance Boot (HAB)

The i.MX RT106S MCU has some fundamental security enablement to protect itself against unsigned images and ensure the integrity of high-value software running on the device. The HAB forces the Read Only Memory (ROM) to only boot into a signed image. This ensures image integrity and prevents from physical and remote attacks since the device is powered on. The HAB is described in detail in the *i.MX RT1060 Processor Reference Manual* (document IMXRT1060RM). See the white paper related to the security aspect of i.MX RT processor.

The implementation steps to enable the HAB of the i.MX RT processor for the SLN-LOCAL2-IOT kit is assured by the Python scripts. With the Ivaldi package, the bootstrap is signed to work with the HAB.

For additional information about the Ivaldi tool's HAB enablement, build the documents in the **Ivaldi/doc** folder according to the **README.md** file.

11.4.1 HAB setup

This section assumes that NXP application image signing tool is completed as 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 the blhost tool. The blhost tool in its simplest form is used to read and write registers, but it communicates with the Flashloader. The Flashloader is a RAM-based application that supports the blhost communication. In normal circumstances, the Flashloader can be executed without being signed. When the HAB is enabled, the Flashloader must be signed by the generated keys.

The secure boot Python scripts are separated into two folders:

- **OEM** – The scripts should only be executed by the Product Owner and the output must be stored in a secure environment. This is because it contains important key information, which if lost, could brick the SLN-LOCAL2-IOT kits or result in a loss of image integrity. The example scripts demonstrate how to configure the Public Key Infrastructure (PKI) and generate a secure binary.
- **MANF** – The scripts are executed on the manufacturing line. They are used to execute the signed Flashloader and communicate with the chip to encrypt the binaries. The scripts contain the generation and programming of FICA. The scripts also serve as examples for the production line programming. Note that the script to enable the HAB should only be performed once per device with a known PKI (i.e., certificates and keys).

> **NOTE**
>
> This process has several failure points, if you have insufficient knowledge of the device. Some of these features are one-way and they permanently impact the behavior of the i.MX RT106S MCU.

The "**./oem/setup_hab.py**" script creates the PKI infrastructure, secure-boot (SB) file, and a signed Flashloader.

- The generated PKI files are located in the **crts** and **keys** folders.
- The following secure-boot (SB) file is located in the **Image_Binaries** folder:
  - **enable_hab.sb**
- The following signed Flashloader files are located in the **Image_Binaries** folder:
  - **ivt_flashloader_signed.bin**
  - **ivt_flashloader_signed_nopadding.bin**

Figure 86 shows the output of the **setup_hab.py** script.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_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 86. Running setup_hab.py

NOTE

Do not run the **setup_hab.py** file more than once without backing up the generated keys and certificates. This results in inability to use the Flashloader and program new images via the serial download mode for the existing HAB-enabled devices.

To enable the HAB with the generated secure boot image file (**enable_hab.sb**), set the i.MX RT106S to the serial download mode by moving jumper J27, which is located on the top of the SLN-LOCAL2-IOT kit, into the “0” position. Then, execute the “**Script/sln_local2_iot_secure_boot/manf/enable_hab.py**” script. Figure 87 shows the usage and output of the “**enable_hab.py**” script.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_iot_secure_boot/manf$ python3 enable_hab.py -h
usage: enable_hab.py [-h] [-cf CONFIG_FOLDER]

optional arguments:
  -h, --help            show this help message and exit
  -cf CONFIG_FOLDER, --config-folder CONFIG_FOLDER
                        Specify the folder that contains board_config.py file
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_iot_secure_boot/manf$ python3 enable_hab.py
Importing board_config.py from ../../sln_local2_iot_config/ folder
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 Rin4ZdJBCy8=
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!
```

Figure 87. Usage of enable_hab.py and its output

NOTE

If you lose the signed Flashloader and certificate/keys, the board will no longer be functional, because this ensures that only signed images can boot.

11.4.2 Creating the images

How to generate the images is described before creating the artifacts and loading them into the encrypted devices. Because the Instruction Vector Table (IVT) is created by the Ivaldi scripts, configure the bootstrap project before creating the image. To do this, right-click on the bootstrap project and navigate to **Properties > C/C++ Build > Settings > Preprocessors**. As shown in Figure 88, set **XIP_BOOT_HEADER_ENABLE** and **XIP_BOOT_HEADER_DCD_ENABLE** to zero.

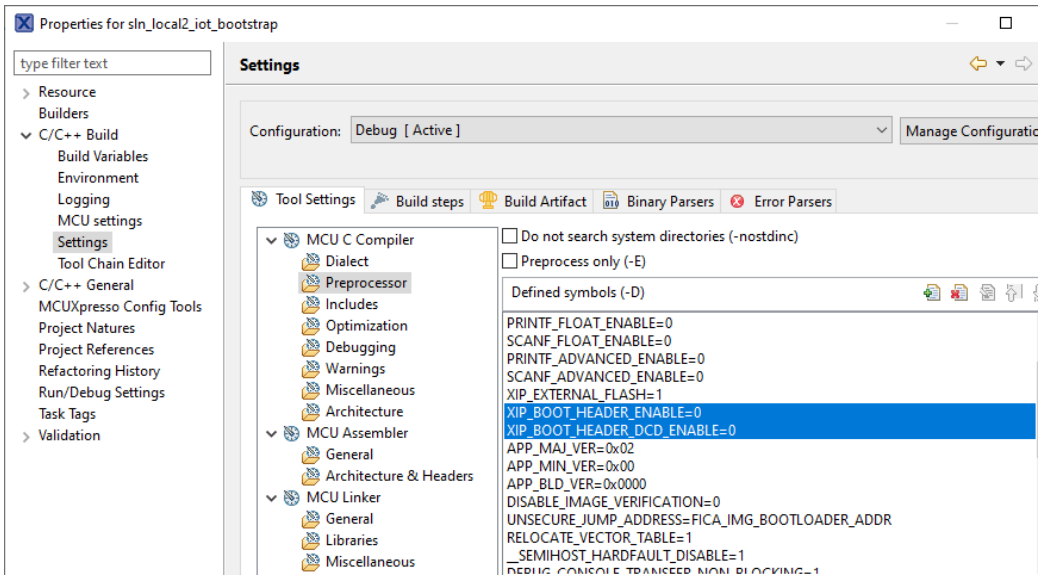


Figure 88. Unsetting of the XIP boot header

After these definitions are updated, build the bootstrap project to generate an image. Because the Ivaldi script only accepts the SREC file format, use the MCUXpresso Binary Utilities to convert the **AXF** file to the **s-record** file in the **bootstrap project**. Right-click the **AXF** file and navigate to **Binary Utilities > Create S-Record**.

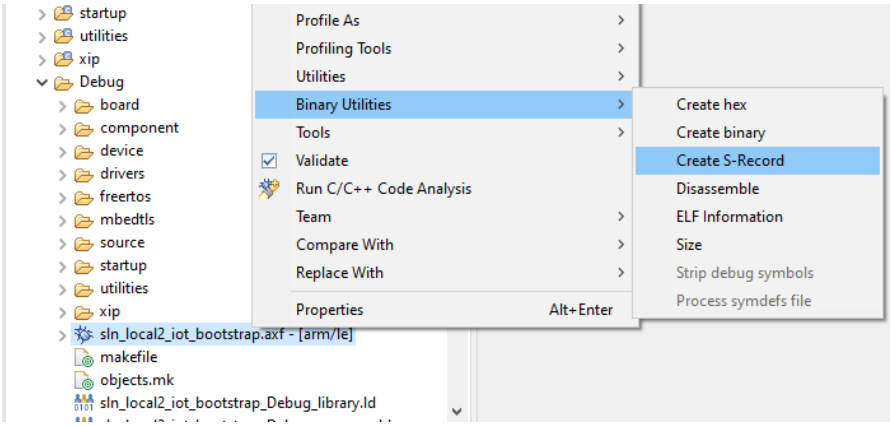


Figure 89. Converting to s-record file

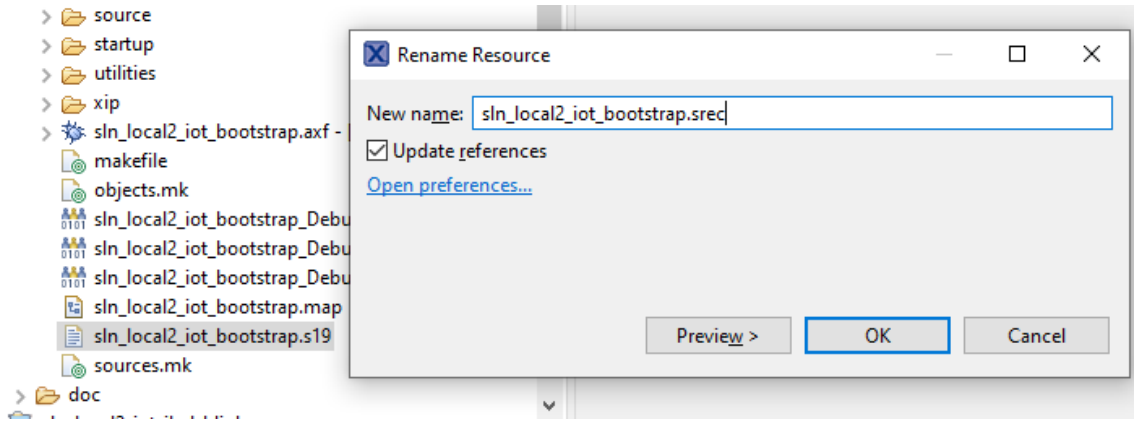


Figure 90. Changing from S19 to SREC

Because this creates an **S19** file, rename the file to **SREC**, as shown in Figure 90.

Continue to build the **bootloader** and **local_demo** projects in the usual way. When these applications are built, it is required to generate the **BIN** files. Build these by navigating to the **Debug** folder in both the **bootloader** and **local_demo** application projects. Right-click the **AXF** file and navigate to **Binary Utilities > Create Binary**.

When the collateral is created, copy the **BIN** and **SREC** files into the **Image_Binaries** folder.

Figure 91 shows all the required files before executing the HAB.

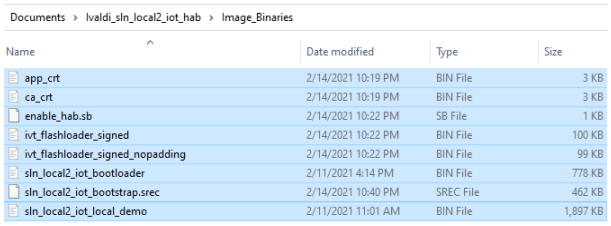


Figure 91. Image binaries before executing HAB

11.4.3 Programming the images

Create a secure image and program the created image into the SLN-LOCAL2-IOT kit.

Execute the **Scripts/sln_local2_iot_secure_boot/oem/secure_app.py** script with the **--signed-only** option. It generates the **boot_sign_image.sb** file with the images created in Creating the images and saves the generated SB file into the **Image_Binaries** folder. Figure 92 shows the usage and output of the **secure_app.py –signed-only** script.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_iot_secure_boot/oem$ python3 secure_app.py -h
usage: secure_app.py [-h] [-s] [--bin] [-cf CONFIG_FOLDER]
                     [uvoice_hab] [bootloader] [uvoice_app]

positional arguments:
  uvoice_hab            path to Srec file
  bootloader            path to bin file
  uvoice_app            path to bin file

optional arguments:
  -h, --help            show this help message and exit
  -s, --signed-only     Only sign rather than encrypt app
  --bin                 App hab file in 'bin' format rather than 'srec'
  -cf CONFIG_FOLDER, --config-folder CONFIG_FOLDER
                        Specify the folder that contains board_config.py file
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_iot_secure_boot/oem$ python3 secure_app.py --signed-only
Importing board_config.py from ../../sln_local2_iot_config/ folder
Cryptographicly signing app image ...
SUCCESS: Created signed image.
Creating signed app file ...
SUCCESS: Created signed app file.
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_iot_secure_boot/oem$ ls ../../../Image_Binaries/boot*
../../../Image_Binaries/boot_sign_image.sb
```

Figure 92. Usage of secure_app.py and its output with --signed-only option

Program the created **boot_sign_image.sb** file into the SLN-LOCAL2-IOT kit in the HAB enabled by executing the **prog_sec_app.py** script with the **--signed-only** option. The script performs the following actions and its output is shown in Figure 93:

- It runs the signed flashloader for the configuration.
- It erases the current flash.
- It programs the signed bootstrap, the signed bootloader and **local_demo**, the application image-signing certificate, the CA image certificate, and the device key and certificate.
- It jumps into the bootstrap and executes.
- It waits until the flow gets to **local_demo**.

NOTE

The spcripts use the file names specified in the **Scripts/ sln_local2_iot_config/board_config.py** folder. For different file names, update the **board_config.py** file.

NOTE

The lock_device.py file should be used only in production, because it disables debugger access.

```
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_iot_secure_boot/manf$ python3 prog_sec_app.py -h
usage: prog_sec_app.py [-h] [-s] [-cf CONFIG_FOLDER]

optional arguments:
  -h, --help            show this help message and exit
  -s, --signed-only     Run signed rather than encrypted app
  -cf CONFIG_FOLDER, --config-folder CONFIG_FOLDER
                        Specify the folder that contains board_config.py file
(env) jongmin@NXL92849:/mnt/c/Users/nxf09836/Documents/Ivaldi_sln_local2_iot_hab/S
cripts/sln_local2_iot_secure_boot/manf$ python3 prog_sec_app.py --signed-only
Importing board_config.py from ../../sln_local2_iot_config/ folder

NOTE: Move jumper J27 from position 1 (Serial mode off) to position 0 (Serial mode
 on)

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 Rin4ZdJBCy8=
SUCCESS: Device thing name is Rin4ZdJBCy8
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...
write-memory
SUCCESS: Programmed flash with certificates for this "thing".
Programming flash with app cert application A...
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_secure_prod.app.a.key.pem:
Enter pass phrase for ../ca/private/my_secure_prod.app.a.key.pem:
Programming flash with AUDIO_EN_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_01 sound file for this "thing".
Programming flash with AUDIO_EN_02 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_02 sound file for this "thing".
Programming flash with AUDIO_ZH_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_ZH_01 sound file for this "thing".
Programming flash with AUDIO_ZH_02 sound file...
00write-memory
SUCCESS: Programmed flash with AUDIO_ZH_02 sound file for this "thing".
Programming flash with AUDIO_DE_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_DE_01 sound file for this "thing".
Programming flash with AUDIO_DE_02 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_DE_02 sound file for this "thing".
Programming flash with AUDIO_FR_01 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_FR_01 sound file for this "thing".
Programming flash with AUDIO_FR_02 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_FR_02 sound file for this "thing".
Programming flash with AUDIO_EN_03 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_03 sound file for this "thing".
Programming flash with AUDIO_EN_04 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_04 sound file for this "thing".
Programming flash with AUDIO_EN_05 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_05 sound file for this "thing".
Programming flash with AUDIO_EN_06 sound file...
write-memory
SUCCESS: Programmed flash with AUDIO_EN_06 sound file for this "thing".
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.
read-memory
SUCCESS: Application entry point at 0x60002451
read-memory
SUCCESS: Application entry point at 0x20208000
Attemping to execute application...
execute
SUCCESS: Application running.

NOTE:Unpower module, move jumper J27 from position 0 (Serial mode on) to position
1 (Serial mode off)
```

Figure 93. Usage of prog_sec_app.py and its output with --signed-only option

Chapter 12
References

The following references supplement this document:

- *MCU Local Voice Control SLN-LOCAL2-IOT-UG Solution User's Guide* (document SLN-LOCAL-IOT-UG)
- Hardware files (gerbers, schematics, BOM)

Chapter 13
Acronyms

Table 14. Acronyms

Acronym	Meaning	(Definition)
AFE	Audio Front End	
ASR	Automatic Speech Recognition	
CA	Certificate Authority	
FICA	Flash Image Configuration Area	Memory space of the external flash that contains information about the binary images of the application and bootloader stages.
GUI	Graphic User Interface	
HAB	High-Assurance Bootloader	
IOT	Internet Of Thing	
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	
OTA	Over The Air	
OTW	Over The Wire	
OTP	One Time Programmable	
PCM	Pulse-code modulation	
PDM	Pulse-density modulation	
PKI	Public Key Infrastructure	
ROM	Read Only Memory	
RTOS	Real-Time Operating System	

Table continues on the next page...

Table 14. Acronyms (continued)

Acronym	Meaning	(Definition)
SDK	Software Development Kit	
UART	Universal Asynchronous Receiver-Transmitter	

Chapter 14
Revision history

Table 15. Revision history

Revision	Date	Substantive changes
0	19 April 2021	Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 19 April 2021

Document identifier: SLN-LOCAL2-IOT-DG

arm