

IMXWGU

i.MX Windows IoT User's Guide

Rev. 1.5.1 — 24 May 2024

User guide

Document information

Information	Content
Keywords	i.MX, Windows IoT
Abstract	i.MX Windows IoT User's Guide describes the process of building and installing Windows IoT OS BSP (Board Support Package) for the i.MX platform. It also covers special i.MX features and how to use them.



1 Overview

The User's Guide describes the process of building and installing Windows IoT OS BSP (Board Support Package) for the i.MX platform. It also covers special i.MX features and how to use them. The guide lists the steps to run the i.MX platform, including board DIP switch settings (see i.MX Windows IoT Quick Start Guide, IMXWQSG) and instructions on the usage and configuration of the U-Boot bootloader. Features covered in this guide may be specific to particular boards or SoCs. For the capabilities of a particular board or SoC, see i.MX Windows IoT Release Notes (IMXWNR).

1.1 Audience

This chapter is intended for software, hardware, and system engineers planning to use the product and anyone who wants to know more about the product.

1.2 Conventions

This chapter uses the following conventions:

- Courier New font: This font is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

1.3 How to start

The i.MX Windows IoT BSP is a collection of binary files, source code, and support files you can use to create a bootable Windows IoT image for i.MX development systems.

1.4 Using Prebuilt Binaries to create an image

The Prebuilt Binary package contains prebuilt release-signed binaries of the drivers and firmware required for Windows IoT Enterprise to run on the NXP i.MX development boards. It is the fastest way to get started running on physical hardware.

If you have downloaded the BSP with the Prebuilt Binaries, see i.MX Windows IoT Quick Start Guide. It will guide you through creating a Windows IoT image that includes the BSP binaries and deploying it to an i.MX development board.

1.5 Using Source Files to create an image

The BSP Source Files package contains the source files of the drivers and firmware required for Windows IoT Enterprise to run on NXP i.MX development boards. It is intended to be used as a reference for partners that have created their own hardware designs that use i.MX 8/9 families of SoCs and must customize the drivers and firmware for their own design.

If you have downloaded an archive with BSP sources, first build Windows drivers and boot firmware from the source before you can create a Windows IoT image and deploy it to your device. Start from [Building Windows IoT for NXP i.MX Processors](#) that will guide you through the process of building Windows drivers and boot firmware from the source. Once you have successfully built the driver and firmware binaries, you can go back to the chapter in i.MX Windows IoT Quick Start Guide that describes how to deploy the Windows IoT image to a development board.

1.6 References

For more information about Windows IoT Enterprise, see [Microsoft online documentation](#).

The following quick start guides available on the [NXP website](#) contain basic information on the board and setting it up:

- [i.MX 8M Quad Evaluation Kit Quick Start Guide](#)
- [i.MX 8M Mini Evaluation Kit Quick Start Guide](#)
- [i.MX 8M Nano Evaluation Kit Quick Start Guide](#)
- [i.MX 8M Plus Evaluation Kit Quick Start Guide](#)
- [i.MX 8QuadXPlus Multisensory Enablement Kit](#)
- [i.MX 93 Evaluation Kit](#)

2 Building Windows IoT for NXP i.MX Processors

2.1 Building the drivers in the BSP

2.1.1 Required tools

The following tools are required to build the drivers:

- git
- git-lfs
- software to unpack zip, gzip, and tar archives
- Visual Studio 2019
- Windows Kits (ADK/SDK/WDK)

2.1.1.1 Visual Studio 2019

- Make sure that you **install Visual Studio 2019 before the WDK** so that the WDK can install a required plugin.
- Download [Visual Studio 2019](#).
- During installation, select **Desktop development with C++**.
- During installation, select the following in the Individual components tab. If these options are not available, try updating VS2019 to the *Latest* release:
 - **MSVC v142 - VS 2019 C++ ARM64 Spectre-mitigated libs (16.11)**
 - **MSVC v142 - VS 2019 C++ ARM64 build tools (16.11)**
 - **Windows 10 SDK (10.0.19041.0)**

2.1.1.2 Windows Kits from Windows 10, version 2004 (10.0.19041.685)

Warning: Make sure that any previous versions of the ADK and WDK have been uninstalled!

- Install [ADK 2004](#)
 - You can also install a Windows PE add-on for ADK as it is needed for preparing installation of an SD card later.
- Install [WDK 2004](#)
 - Scroll down and select Windows 10, version 2004.
 - Make sure that you allow the Visual Studio Extension to install after the WDK install is completed.
- If the WDK installer says it could not find the correct SDK version, install [SDK 2004](#)
 - Scroll down and select Windows 10 SDK, version 2004 (10.0.19041.0).
- After installing all Windows Kits, restart the computer and check if you have the correct versions installed in the Control panel.

2.1.2 Obtaining sources for building the drivers

For building the drivers, use the NXP i.MX BSP sources package provided as `W<os_version>-imx-windows-bsp-<build_date>.zip`. The package contains sources for both the boot firmware and Windows drivers.

2.1.2.1 Preparing source for building the drivers

To prepare sources for building drivers, follow these steps:

1. Create an empty directory, further referred as `<BSP_DIR>`, and extract the downloaded archive there. The path to this directory must be as short as possible, containing only letters and underscores. Braces and other special characters can cause build errors.
2. Populate the directory by running `Init.bat`.

2.1.3 Structure of Windows driver sources

The *imx-windows-iot*- sources of Windows drivers have the following structure:

BSP	Contains boot firmware, driver binaries (generated at build time), and scripts needed to deploy BSP to the development board.
build	Contains build scripts and the VS2019 solution file.
components	Contains third-party binaries and utility projects.
driver	Contains driver sources.
hals	Contains hal extension sources.

2.1.4 One-time environment setup

To generate driver packages on a development machine, install test certificates.

1. Open an Administrator Command Prompt.
2. Navigate to your BSP, the folder `imx-windows-iot\build\tools`.
3. Launch `StartBuildEnv.bat`.
4. Run `SetupCertificate.bat` to install the test certificates.
5. The HAL Extensions must be signed by certificates provided by Microsoft. The required certificates that are included in WDK have expired. Download the [Windows 11, version 22H2 EWDK](#) and use the "Windows OEM HAL Extension Test Cert 2017 \(\TEST ONLY\)" and "Windows OEM Intermediate 2017 \(\TEST ONLY\)" found in the `EWDK.iso` file or contact Microsoft for help.

Some tools may not work correctly if LongPath is not enabled, therefore run the following command in the console:

Execute `reg add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem /v LongPathsEnabled /t REG_DWORD /d 1` command.

2.1.5 Building the drivers

1. Open the solution `imx-windows-iot\build\solution\iMXPlatform\iMXPlatform.sln` located in the path where you have extracted the BSP archive.
2. Choose the `Debug` or `Release` build type.
3. If the secure boot feature is enabled, it is required to use signed drivers.
4. To build, press `Ctrl-Shift-B` or choose `Build -> Build Solution` from the menu. It compiles all driver packages; then `imx-windows-iot\BSP\IoTEntOnNXP\drivers` for deployment.
5. The updated drivers could now be injected into the installation image or manually installed to the running development board.
 - To manually install drivers, copy them to the development board via USB drive, network share, scp, remote desktop. The drivers can be installed either by clicking `install` in the right-click menu of the 'inf' file or by the devcon command-line utility.

- To debug, use the `.kdfiles` of WinDBG.
- To initiate the driver reload, use `devcon` or reset the board.
- To create an installation SD card, see [i.MX Windows IoT Quick Start Guide](#).

2.2 Building ARM64 Firmware

This chapter describes the process of setting up a build environment to build the latest firmware and update the firmware on the development board.

2.2.1 Required tools

- git
- git-lfs
- software to unpack zip, gzip, and tar archives

2.2.2 Obtaining sources for building ARM64 Firmware

For building the ARM64 Firmware, you need:

1. The NXP i.MX BSP sources package is available at www.nxp.com. The package contains sources for both the boot firmware and Windows drivers.
2. The i.MX firmware and NXP Code Signing Tool (CST). Obtaining is described in [Preparing sources for building firmware](#).

2.2.2.1 Preparing sources for building firmware

1. Create an empty directory, further referred as `<BSP_DIR>`, and extract the downloaded archive there.

```
unzip W<os_version>-imx-windows-bsp-<build_date>.zip -d win10-iot-bsp
```

The command creates the `win10-iot-bsp` directory containing the `.git` repository with the BSP release.

Note: The path to this directory must be as short as possible, containing only letters and underscores. Braces and other special characters can cause build errors.

2. Populate the directory by running `Init.sh`.

Note: Script checks out sources from the repository by `git reset --hard`. The `Init.sh` shall check out the submodules that are required to build the i.MX boot firmware by `git submodule update --init --recursive`. During prerelease testing, the `Init.sh` executed inside the Ubuntu environment has run into “server certificate verification failed. CAfile: `/etc/ssl/certs/ca-certificates.crt` CRLfile: none” error. The problem could be solved by installing `apt-transport-https` `ca-certificates` and `certificate update`.

```
sudo apt update ; sudo apt-get install apt-transport-https
ca-certificates -y ; sudo
\update-ca-certificates
```

3. Extract the [Code Signing Tool](#) inside the bsp repository and rename the newly created folder to `cst` to get the `<BSP_DIR>/cst` folder:

```
tar xf cst-3.1.0.tgz
mv release cst
rm cst-3.1.0.tgz
```

4. Extract the [i.MX firmware](#) from the NXP website and place it in `firmware-imx`.

```
chmod +x firmware-imx-8.18.bin
./firmware-imx-8.18.bin
mv firmware-imx-8.18 firmware-imx
```

```
rm firmware-imx-8.18.bin
```

Note: It extracts the tool inside the bsp repository and renames the newly created folder to `firmware-imx` to get `<BSP_DIR>/firmware-imx/firmware/ ddr/` in the directory tree.

5. Your directory structure must contain the following folders.

```
- <BSP_DIR>
|- cst (manually downloaded)
|- firmware-imx (manually downloaded)
|- Documentation
|- MSRSec
|- RIoT
|- imx-atf
|- imx-mkimage
|- imx-optee-os
|- imx-windows-iot
|- mu_platform_nxp
|- patches
|- uboot-imx
```

2.2.3 Setting up your build environment

1. Start a Linux environment such as:

- Dedicated Linux system
- Linux Virtual Machine
- Windows Subsystem for Linux ([WSL setup instructions](#))

Note: W-imx-windows-bsp-1.5.1.zip and above requires python 3.10 version and higher. The build process was validated with Ubuntu 22.04 (jammy) in WSL and also in standalone Ubuntu.

2. Obtain and prepare the BSP sources by following all steps described in Obtaining BSP sources. Use `Init.sh`, not `Init.bat` to populate the repository and all submodules.
3. Install or update build tools. The shell commands below can be used to do this process on Ubuntu 22.04:

```
sudo apt-get update
sudo apt-get upgrade
```

4. Install the required software. The `mu_project` currently requires python 3.10 and higher.

```
sudo apt-get install attr build-essential python3 python3-dev python3-venv
device-tree-compiler bison flex swig iasl uuid-dev wget git bc libssl-dev
zlib1g-dev python3-pip mono-devel gawk libgnutls28-dev
```

5. Download the Arm64 cross-compiler.

```
pushd ~
wget https://releases.linaro.org/components/toolchain/binaries/7.4-2019.02/
aarch64-linux-gnu/gcc-linaro-7.4.1-2019.02-x86_64_aarch64-linux-gnu.tar.xz
tar xf gcc-linaro-7.4.1-2019.02-x86_64_aarch64-linux-gnu.tar.xz
rm gcc-linaro-7.4.1-2019.02-x86_64_aarch64-linux-gnu.tar.xz
* \# The cross compiler prefix is required to be exported later
\# into AARCH64\ TOOLCHAIN\ PATH variable.
\# export AARCH64\ TOOLCHAIN\ PATH=~\gcc-linaro-7.4.1-2019.02\
\# -x86\_64\_aarch64-linux-gnu\bin\aarch64-linux-gnu-*
popd
```

6. Change the directory to the `BSP_DIR`. The following commands reference the files inside the BSP directory. That `BSP_DIR` contains extracted W-imx-windows-bsp.zip.

```
cd BSP_DIR
```

7. [Project MU](#) strongly suggests the use of a Python Virtual Environment for each workspace. In this case, BSP revision-separated environments allow workspaces to keep specific Pip module versions without modifying the global system state when the firmware is compiled.

```
python3 -m venv <path to new environment>
source <path to new environment>/bin/activate
For example:
python3 -m venv ~/venv/win_fw_build
source ~/venv/win_fw_build/bin/activate
```

The virtual environment does not use system packages. Therefore, do not use `sudo` when installing packages using `pip`.

8. Install the required Python packages.
 - a. Install or update `mu_platform` Python dependencies using `pip`.

```
pushd mu_platform_nxp
pip3 install -r requirements.txt --upgrade
```

- b. Install the `pycryptodome` package (successor of `pycrypto`).

```
pip3 install pycryptodome
```

- c. Install the `pyelftools` package.

```
pip3 install pyelftools
```

- d. Install the `cryptography` package.

```
pip3 install cryptography
```

9. Initialize and update MU platform dependencies. (This step is optional because `buildme64.sh` does these steps automatically.)

```
python3 NXP/MX8M_EVK/PlatformBuild.py --update
```

10. Return to BSP root.

```
popd
```

2.2.4 Building the firmware

To build the boot firmware:

1. Open the cmd prompt inside `BSP_DIR`.

```
cd <BSP_DIR>
```

2. Activate your python virtual environment (Use the path specified when creating the environment.)

```
source ~/venv/win_fw_build/bin/activate
```

3. Export `AARCH64_TOOLCHAIN_PATH` cross compiler prefix. In this guide, the toolchain has been placed inside the home (`~/`) directory.

```
export AARCH64_TOOLCHAIN_PATH=~/gcc-linaro-7.4.1-2019.02-x86_64_aarch64-
linux-gnu/bin/aarch64-linux-gnu-
```

4. Optionally, if there is a major update, you may need to step into `mu_platform_nxp` and run `python3 NXP/MX8M_EVK/PlatformBuild.py` with `--setup --force` and then `--update` manually. To get a

clean and up-to-date MU build environment, stash or commit your changes. The command performs `git reset --hard`.

5. Build the firmware and create `firmware.bin`. To build the boot firmware, execute the `buildme64.sh -b <BOARD_NAME> -t all [-clean]` script provided in `BSP_DIR` (the root of extracted BSP sources).

```
./buildme64.sh -b MX8M_EVK -t all -c
```

The `buildme64.sh` script bundled in BSP also copies `flash.bin` and `uefi.fit` into `<BSP_DIR> / imx-windows-iot/components/Arm64BootFirmware/<board_name>`. It allows rebuilding only UEFI or U-Boot.

- Use `-b MX8M_EVK` or `-b 8M` to select i.MX 8M EVK
- Use `-b MX8M_MINI_EVK` or `-b 8Mm` to select i.MX 8M Mini EVK
- Use `-b MX8M_NANO_EVK` or `-b 8Mn` to select i.MX 8M Nano EVK
- Use `-b MX8M_PLUS_EVK` or `-b 8Mp` to select i.MX 8M Plus EVK
- Use `-b MX8QXP_MEK` or `-b 8X` to select i.MX 8QXP_MEK
- Use `-b MX93_11X11_EVK` or `-b 93` to select i.MX 93 EVK
- Use `-t secured_efi` to build `signed_firmware_uuu.bin`

Options for builds:

- `-b|-board` specifies the board for which binaries will be built
 - `all` = build all devices,
 - `8M, MX8M_EVK`
 - `8Mm, MX8M_MINI_EVK`
 - `8Mn, MX8M_NANO_EVK`
 - `8Mp, MX8M_PLUS_EVK`
 - `8X, MX8QXP_MEK`
 - `93, MX93_11X11_EVK`
- `-t|-target_app` specifies a target to build
 - `all` = build all components
 - `u|uboot` = build U-Boot (by default with UUU tool)
 - `optee` = build optee core
 - `apps|tee_apps` = build optee trusted applications
 - `uimg|uboot_image` = create bootable image
 - `tools|uefi_tools` = build UEFI tools
 - `uefi` = build UEFI
 - `profile_dev` = build UEFI with development profile (set by default)
 - `profile_secure` = build UEFI with secure profile
 - `profile_frontpage` = build UEFI with frontpage profile
 - `secured_efi|secured_uefi` = build UEFI in secure mode + sign image (the name of the resulting firmware is prefixed with `signed_`)
- `[-cap|-capsule]` creates capsule
- `[-c|-clean]` cleans build files before build
- `[-fw|-fw_bin]` requests build of firmware from existing binaries
- `[-nu|-no_uuu]` builds U-Boot without UUU tool (the name of the resulting firmware does not contain `_uuu` suffix)
- `[-h|-help]` prints manual for script usage
- `[-bc|build_configuration]` specifies build configuration of UEFI (`RELEASE` is selected as default)
 - `release` or `RELEASE` = for Release version of UEFI

- `debug` or `DEBUG` = for Debug version of UEFI
 - `[-ao|-advance_option]` Advanced options for **experienced users**
 - `rpmb_reset_fat` = clears RPMB FAT
 - `rpmb_write_key` = writes RPMB KEY into RPMB
 - `no_rpmb_test_key` = uses the Hardware-Unique key (HUK) instead of `TEST KEY` (`TEST KEY` is used as default)
 - `optee_core_v` = turns on verbose mode of OpTEE core
 - `optee_core_vv` = turns on the highest verbose mode of OpTEE core
 - `optee_ta_v` = turns on verbose mode of OpTEE trusted applications
 - `optee_ta_vv` = turns on the highest verbose mode of OpTEE trusted applications
 - `[TARGET_WINDOWS_BSP_DIR]` Specifies path to `imx-windows-iot`, in which the firmware shall be updated.
 - `[KEY_ROOT]` specifies the path to the custom the PKI root.
6. To deploy `firmware_uuu.bin` to the i.MX development board, follow the process described in [i.MX Windows IoT Quick Start Guide](#).

2.2.5 Common causes of build errors

1. ImportError: No module named Crypto.PublicKey.
 - This error is encountered when the `pycryptodome` module is missing or if the obsolete `pycrypto` is removed.
2. Unable to enter a directory. The directory does not exist.
 - We have run into this problem in case `gitmodules` were not downloaded completely (for example, `MSRSec` is empty) or `cst` or `firmware` directories were missing. Try repeating the Obtaining BSP sources step by step.
3. The build fails in WSL while the BSP is located somewhere in `/mnt/c` of the WSL.
 - Try `setfatattr -n system.wsl_case_sensitive -v 1 <BSP_DIR>`. OP-Tee also requires symbolic links. We have been able to build boot firmware in `/mnt/c/` on Windows OS version 1909. Workaround is to copy the BSP to the WSL filesystem, for example, to HOME.

3 Display/GPU driver

This chapter contains several notes related to the Windows i.MX GPU driver. The kernel graphic driver consists among others of the GPU driver `galcore.sys` and the display controller driver `dispctrl.dll`. The setup information file `galcore.inf` contains several parameters that are written into the Windows registry database and later used for the driver configuration. To change these parameters, one of the following options can be used:

- Update the registry database directly under the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{4d36e968-e325-11ce-bfc1-08002be10318}\0000` key. Then you can either reboot the board, or restart the driver using `devcon` (`devcon.exe restart ACPI\VERI700x`), or Disable device and Enable device using Device Manager for Display adapters\i.MX GPU device.
- Update the INF file and uninstall/re-install the GPU driver, and then reboot.

3.1 Display interface selection

The following Registry (INF) parameter is used to select the display interface for a particular display. Applicability is platform-dependent, see [Quick Start Guide](#), the features list for a particular platform, and limitations in [Release Notes](#). If a platform supports only one possibility, the parameter is ignored. For specific hardware configuration, see the platform reference manual.

The `Display<n>Interface` (where `<n>` display id = 0,1,2,...) parameter is of the `REG_DWORD` type.

Possible values:

`DISP_INTERFACE_HDMI = 0x1`

`DISP_INTERFACE_MIPI_DSI0 = 0x2`

`DISP_INTERFACE_MIPI_DSI1 = 0x3`

`DISP_INTERFACE_LVDS0 = 0x4`

`DISP_INTERFACE_LVDS1 = 0x5`

`DISP_INTERFACE_LVDS_DUAL0 = 0x6`

`DISP_INTERFACE_PARALLEL_LCD = 0x7`

Some platforms support multiple display interfaces. See [Release Notes](#). To enable multiple displays, set the following registry parameter to one:

The `EnableMultiMon` parameter is of the `REG_DWORD` type. Values: 0 = single display determined by `Display0Interface`. 1 = multiple displays enabled, the first display is determined by the `Display0Interface` parameter, the second display is determined by the `Display1Interface` parameter, and so on.

3.2 Display resolution and timing parameters

The display resolution is configured differently for individual display interfaces.

3.2.1 HDMI display interface

The display is configured with native display resolution read from the EDID of the connected display. Typically, this is 1920x1080@60 Hz for most standard HDMI monitors. It is not possible to change/override the native resolution.

Currently, the maximum resolution supported by the display driver is 1080p (1920x1080@60 Hz), which is also set if the native display resolution exceeds this maximum.

3.2.2 LVDS, MIPI-DSI and Parallel display interfaces

The display resolution and timing parameters are obtained from the following registry parameter:

The `Display<n>EDID` (where `<n>` display id = 0,1,2,...), parameter is of the `REG_BINARY` type.

The parameter contains EDID data encoded according to the EDID structure v1.4 data format (standard published by VESA). The first data block 128 bytes long is used, that is, only a basic EDID structure without any extensions. Resolution and timing parameters are parsed from Standard timing information - Descriptor 1 (offsets 54 - 71), specifically pixel clock, horizontal active pixels resp. vertical active lines, blanking pixels, synchronization pulse width, front porch, and VSYNC, HSYNC signals polarity. During EDID loading from the registry, the EDID header is checked (offsets 0 - 7) and the checksum must match (offset 127). Other EDID data are irrelevant to the GPU driver. Default EDID data in the INF file sets 1280x720@60 Hz mode.

The BSP package contains preprepared testing EDID data for several standard modes: see `<BSP>\imx-windows-iot\driver\display\dispdll\util\include\edidtst.h`

3.3 Display specific parameters

3.3.1 LVDS display interface

Registry (INF) parameters related to the LVDS interface:

- The `Display<n>BusDataWidth` (where `<n>` is display id = 0,1,2,...) parameter of the `REG_DWORD` type determines the number of pixels mapped to the output signal. 24 bpp or 18 bpp are supported. The default value is 24.
- The `Display<n>BusMapping` (where `<n>` is display id = 0,1,2,...) parameter of the `REG_DWORD` type determines the pixel-mapping type in the output signal. `DISP_BUS_MAPPING_SPWG = 0x1` or `DISP_BUS_MAPPING_JEIDA = 0x2` are supported. The default value is 0x1.

3.3.2 MIPI-DSI display interface

Registry (INF) parameters related to the MIPI-DSI interface:

- The `Display<n>NumLanes` (where `<n>` is display id = 0,1,2,...) parameter of the `REG_DWORD` type determines the number of DSI lanes. Possible values are 1-4. The default value is 4.
- The `Display<n>ChannelId` (where `<n>` is display id = 0,1,2,...) parameter of the `REG_DWORD` type determines the virtual channel ID of the display. The default value is 0.

3.4 Display support in firmware

Display-related peripherals are configured in U-Boot for i.MX 8M and the following paragraphs are not valid for them. The following description is related to the firmware driver for i.MX 8M Plus, i.MX 8M Nano, i.MX 8M Mini, i.MX 8QXP, and i.MX 93.

3.4.1 Firmware display interface selection

The firmware display interface can be selected in the `giMX8TokenSpaceGuid.PcdDisplayInterface` parameter in the platform description file (dsc) located in `<BSP>/mu_platform_nxp/NXP/<Board>/<Board>.dsc`. Possible values are `HDMI = 0`, `MIPI_DSI0 = 1`, `LVDS0 = 2`, `LVDS1 = 3`, `LVDS dual = 4`, `MIPI_DSI1 = 6`. The available display interfaces depend on the specific board (see Release Notes or SoC reference manual for more information). The parameter is used for interfaces that do not allow automatic detection. If there is only one instance of display interface (for example, LVDS) on the SoC, it is assumed that it has index 0 (that is, LVDS0)

Automatic detection is implemented for HDMI-based display interfaces that include IMX-MIPI-HDMI (MIPI-DSI to HDMI converter), IMX-LVDS-HDMI (LVDS to HDMI converter), and the native HDMI interface. These interfaces are probed in the same order of priority as stated above (that is, MIPI-HDMI, LVDS-HDMI, native HDMI) and if successfully detected, `giMX8TokenSpaceGuid.PcdDisplayInterface` is overridden with the detected display interface.

After changing any of the parameters, the firmware must be recompiled.

3.4.2 Firmware display resolution

The firmware display resolution is stored in the `PreferredTiming` variable. This variable is initialized in the `LcdDisplayDetect` function in the `iMX8LcdHwLib.c` (for i.MX 8M Plus, i.MX 8M Nano, i.MX 8M Mini) or `iMX93DisplayHwLib.c` (for i.MX 93) or `iMXDpuHwLib.c` (for i.MX 8QXP) file respectively. These source files contain several predefined resolutions and timing parameters. For example to select 1024x768@60 resolution, initialize the `PreferredTiming` variable: `LcdInitPreferredTiming(&PreferredTiming_1024x768_60, &PreferredTiming);`

For HDMI-based display interfaces (see previous paragraph), the `giMX8TokenSpaceGuid.PcdDisplayReadEDID` parameter (TRUE/FALSE) allows enabling/disabling EDID reading. The resolution and display parameters are then extracted from the Detailed Timing descriptor of EDID data (native resolution). The `giMX8TokenSpaceGuid.PcdDisplayForceConverterMaxResolution` parameter (TRUE/FALSE) allows clamping display resolution to the supported maximum, that is, if the EDID Detailed Timing descriptor contains a resolution higher than the supported maximum, EDID data are discarded, and the supported maximum resolution is used instead. Both these parameters are located in the platform description file (dsc) - see the previous paragraph.

After changing any of the parameters, the firmware must be recompiled.

Note: Only a limited set of pixel clocks is supported, so for a new resolution with a pixel clock different from the predefined in the above source files, the corresponding clock driver must be updated.

4 Power management

Power management consists of the **Processor Power Management (PPM)** that includes low-power state transition of processor cores and of the **Device Power Management (DPM)** that includes power gating and clock gating of individual devices. An important part of customization of power management is the **Power Engine Plugin (PEP) driver** that defines the processor and platform low-power states and can handle power and clock gating for individual devices. This chapter contains information on the current support of power management for i.MX 8/9 platforms, relevant tools, and utilities.

4.1 Power management user scenarios

We consider 2 power scenarios that could be of interest for vendors using i.MX 8/9 platforms:

- At runtime: reduce runtime power consumption by putting unused resources to temporary possibly short sleep states:
devices - to clock gating/power gating or other low-power states, for example, D3/F1, CPUs to CPU-suspend in Standby or Power Down mode.
- When an IoT device is idle: platform entering low-power idle states (wait state, power off state) with minimal power consumption and wake-up capability via selected devices.

4.2 Device power management DPM on i.MX 8/9 platforms

There are working samples of **power management framework (PoFx)** callbacks in I2C and PWM drivers. The functionality must be enabled by `I2C_POWER_MANAGEMENT` and `PWM_POWER_MANAGEMENT` macros.

The **Dx states** are Devices states (D0=Running, D3=low power)

The **Fx states** are Components states (F0=Running, F1=low power)

The i.MX 8/9 implementation is based on the Single Component [KMDF Power Framework \(PoFx\) Sample](#) provided by Microsoft.

The power state transitions from D0/F0 to D3/F1 and back are based on the device activity (for example, running some test traffic). The device power state can be checked in WinDbg using the `!fxdevice` command. The state transition happens based on OS decision (made incl. the S0 Idle Timeout), and the driver is notified using the [PO_FX_COMPONENT_IDLE_STATE_CALLBACK](#). It must change HW status to low power (if State > 0) or to running (if State = 0). The functionality is located in the files `imxi2cpofx.h/cpp` and `imxpwm_pofx.h/cpp`. The PoFx functionality can be copied to other drivers based on specific vendor requirements.

The Device driver interacts with the Windows PoFx framework using the [WdfDeviceAssignS0IdleSettings](#) and [WdfDeviceWdmAssignPowerFrameworkSettings](#) methods.

When the power management support is implemented in the device driver, the `Power Management` tab becomes visible in the device properties in the Device Manager:

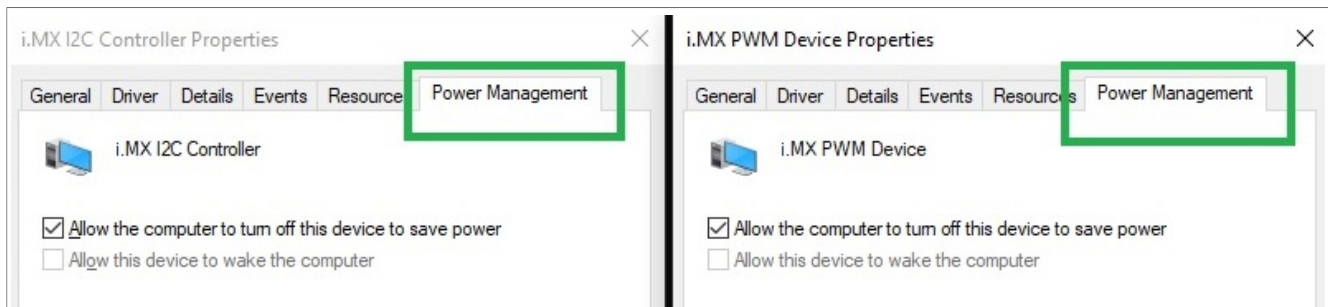


Figure 1. Power-managed devices I2C PWM

4.3 Processor power management PPM on i.MX 8/9 platforms

The **Power Engine Plugin (PEP)** driver is visible in the Device Manager -> System devices for all the supported i.MX 8/9 boards:

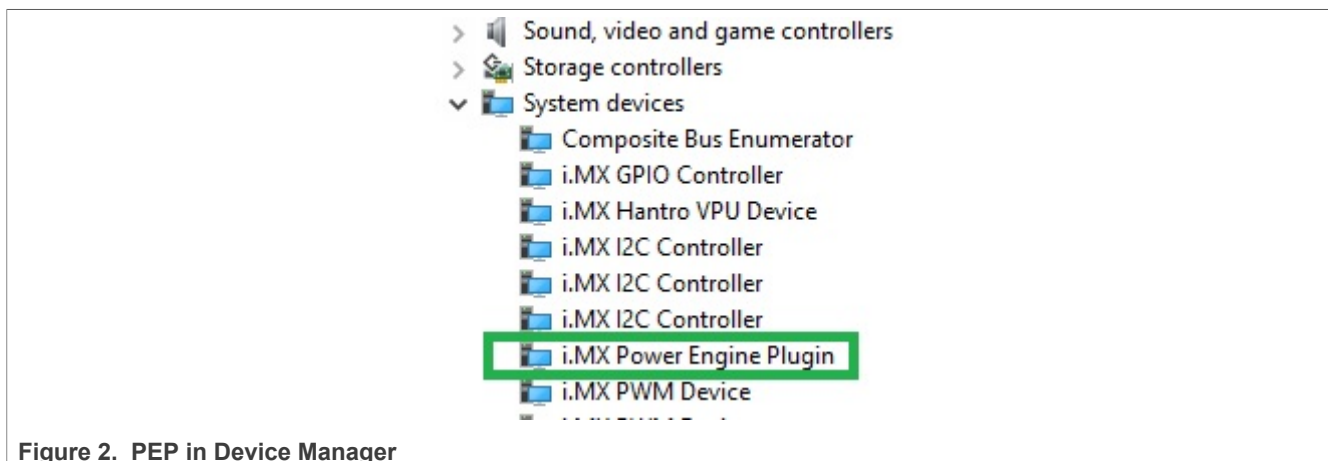


Figure 2. PEP in Device Manager

The PEP handles putting of CPU cores to coordinated low-power (sleep) states as the requested by the operating system. The call sequence that puts the CPU core to sleep looks as follows:

```
WindowsOS -> PEP::AcceptProcessorNotification -> PEP:: PpmIdleExecute -> WFI
(Wait for
    Interrupt instruction)
```

```
WindowsOS -> PEP::AcceptProcessorNotification -> PEP:: PpmIdleExecute -> SCM
call
->Imx-Atf PSCI CPU_SUSPEND -> HW instructions to CPU sleep
```

The sleeping CPU core is woken from the sleep state by interrupt, either a Processor to Processor Interrupt (PPI) for example, IRQ27 or by device interrupt IRQ > 32, for example, a USB device like a mouse or keyboard. See section related to IRQs.

PEP and Imx-Atf: ATF is the Arm Trusted Firmware, integrated with Uefi and U-Boot in firmware.bin. The ATF implements the PowerStateCoordinatedInterface (PSCI industry standard, DEN0022E_Power_State_Coordination_Interface.pdf) from the ARM specification, incl. the CPU_SUSPEND method used to put CPU cores into low-power sleep states. The CPU_SUSPEND can specify either standby or Power-down mode. When the last CPU core goes into the low-power Power-down mode, the whole platform must enter the platform power down, which includes DDR self-refresh, and set up for wake-up using selected interrupts. In Release

Milestone 6, the platform power down is not yet fully integrated with Win10 IoT OS so it needs more effort to have this functional.

The PEP also ensures that before entering the Coordinated low-power state (defined in PEP) all devices are in the required low-power state. This is defined in PEP: DpmDeviceIdleConstraints, the constraints are expected to be extended in future releases.

PEP and WinDbg: the PEP driver is loaded in Windows OS as one of the first drivers during startup. It can be replaced and debugged with WinDbg as usual. Enable the `#define DBG_MESSAGE_PRINTING` in `imxpep_dbg.h` file to get traces in the WinDbg command window.

4.4 Power management tools and debugging

The following tools can be used to analyze the current Power management functionality:

Utility	Description
powercfg /a	Available sleep states
powercfg /sleepstudy	Sleep study HTML report
powercfg /energy	Energy efficiency analysis and issues
WinDbg !fxdevice	Device power management status

4.4.1 powercfg /a

This command displays the available sleep states.

In i.MX that uses the Modern Standby the only supported state is **S0 Low Power Idle - Network Connected**.

```
C:\> powercfg /a
The following sleep states are available on this system:
    Standby (S0 Low Power Idle) Network Connected
```

4.4.2 powercfg /sleepstudy

This command generates a detailed HTML report with analysis of Sleep states during the last 3 days.

It includes how many % of time was spent in the Deepest Runtime Idle Platform State (DRIPS) during each Sleep period.

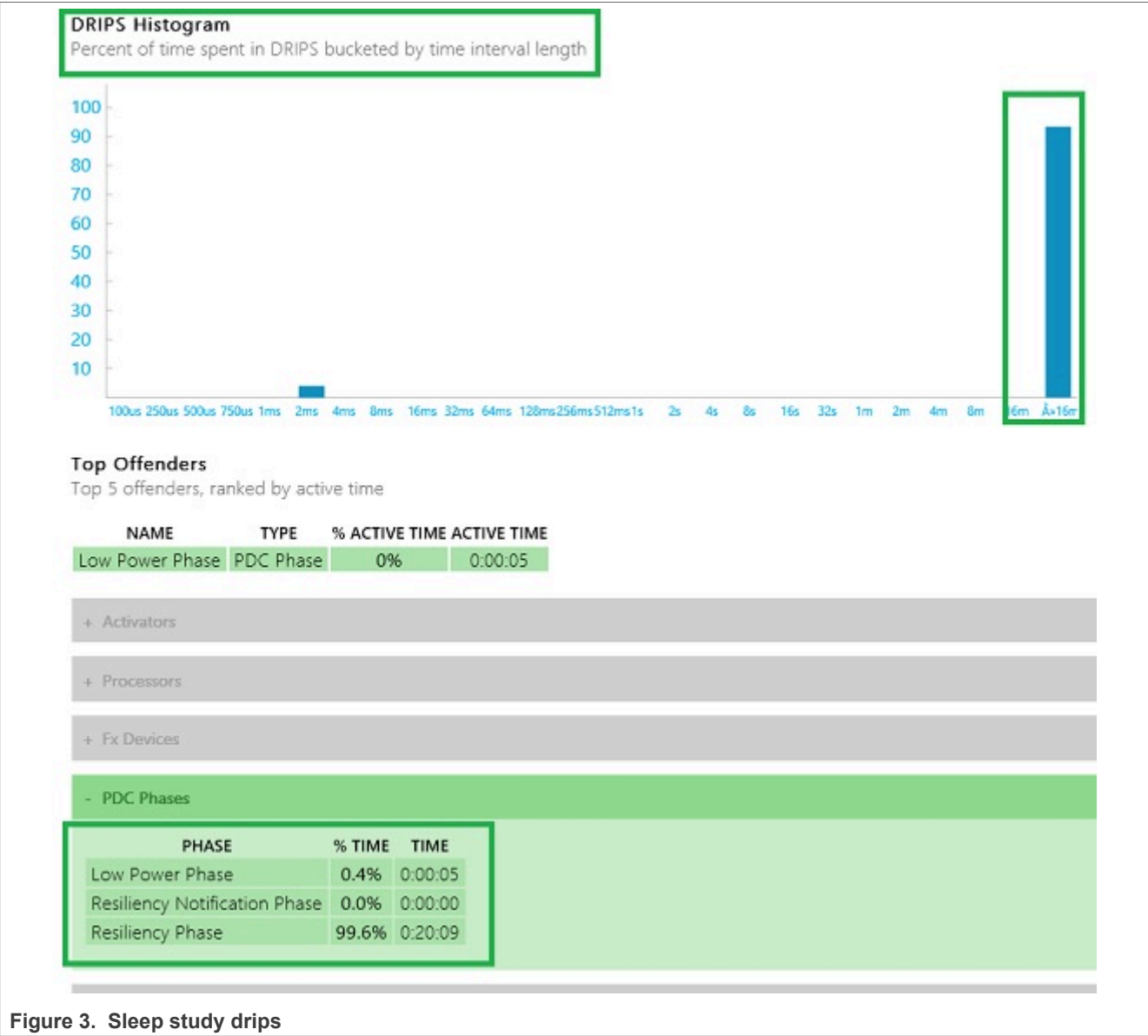


Figure 3. Sleep study drips

4.4.3 powercfg /energy

This command generates the energy consumption analysis and issues a report.

4.4.4 WinDbg !fxdevice

The fxdevice command gives detailed status and history of the power state transition of each power-managed device.

For example I2C2 in D0 state when active, in D3 state (power down) when idle, and the PoFx IRP log:

```
!fxdevice ... find the relevant device

Within 30 sec after I2C test run => active D0/F0 state:
!fxdevice 0xfffffaf8256372010
DevNode: 0xfffffaf8251115aa0
```

```
UniqueId: "\_SB.I2C2"
InstancePath: "ACPI\NXP0104\2"
Device Power State: PowerDeviceD0
Component Count: 1
  Component 0: Current:F0/Deepest:F1 - IDLE      (RefCount = 0)

After 30 sec after I2C test run => low power D3/F1 state:
!fxdevice 0xfffffaf8256372010
DevNode: 0xfffffaf8251115aa0
UniqueId: "\_SB.I2C2"
InstancePath: "ACPI\NXP0104\2"
Device Power State: PowerDeviceD3
Component Count: 1
  Component 0: Current:F1/Deepest:F1 - IDLE      (RefCount = 0)

nt!DbgBreakPointWithStatus:
fffff803`43c08330 d43e0000 brk                #0xF000
0: kd> !fxdevice fffffaf82570e9aa0
!fxdevice 0xfffffaf82570e9aa0
DevNode: 0xfffffaf8251115aa0
UniqueId: "\_SB.I2C2"
InstancePath: "ACPI\NXP0104\2"
Device Power State: PowerDeviceD3
PEP Owner: Default PEP
Acpi Plugin: 0
Acpi Handle: 0
Device Status Flags: DevicePowerNotRequired_DeviceNotified
DevicePowerNotRequired_ReceivedFromPEP
Device Idle Timeout: 0000000000
Device Power On: No Activity
Device Power Off: No Activity
Device Unregister: No Activity
Component Count: 1
  Component 0: Current:F1/Deepest:F1 - IDLE      (RefCount = 0)
  Pep Component: 0xfffffaf8256df24d0
    Active: 0 Latency: 0 Residency: 0 Wake: 0 Dx IRP: 0 WW IRP: 0
    Component Idle State Change: No Activity
    Component Activation: No Activity
    Component Active: No Activity
Log has 25 entries starting at 0:
#      IntTime      CPU    Cid    Tid
---      -
0 000000076660627f    3      4     f0 Device registered with 1 component(s)
1 000000076660627f    3      4     f0 Start power management
2 000000076660627f    3      4     f0 Component 0 latency set to 8000001
3 000000076660627f    3      4     f0 Component 0 residency set to 120000001
4 0000000766609e64    1      4    5c0 Component 0 changed to idle state F1
5 0000000766609e64    1      4    5c0 Power not required from default PEP
6 0000000766609e64    1      4    5c0 Power not required to device
7 0000000766609e64    2      4     ec Power IRP requested with status 0
8 0000000766609e64    2      4     ec Power IRP type D3 dispatched to device
stack
9 0000000766609e64    3      4     e0 Device power state changed to D3
10 0000000766633b5a    1      4     ec Power required from default PEP
11 0000000766633b5a    1      4     ec Power required to device
12 0000000766633b5a    1      4     ec Driver device power required callback
pending
13 0000000766633b5a    1      4     ec Power IRP requested with status 0
```

14	0000000766633b5a	1	4	ec	Power IRP type D0 dispatched to device stack
15	0000000766638961	2	4	b78	Device power state changed to D0
16	0000000766638961	2	4	b78	Device powered
17	0000000766638961	2	4	b78	Driver device power required callback completed
18	0000000766638961	3	4	18	Component 0 changed to idle state F0
19	000000076ddba246	0	4	18	Component 0 changed to idle state F1
20	000000076ddba246	0	4	18	Power not required from default PEP
21	000000076ddba246	0	4	18	Power not required to device
22	000000076ddba246	0	4	b78	Power IRP requested with status 0
23	000000076ddba246	0	4	b78	Power IRP type D3 dispatched to device stack
24	000000076ddba246	0	4	e0	Device power state changed to D3

5 Secure boot

5.1 Basic concepts

Secure Boot is a feature that prevents loading malicious pieces of software (rootkits) during the system boot. To perform a secure boot, the feature has to be supported by the whole boot chain, starting at the device ROM code and ending in Windows. For more information on how to prepare the board for Secure Boot, see [Secure Provisioning](#).

For more detailed information on each platform, see:

- [Secure boot on i.MX 8M](#)
- [Secure boot on i.MX 8QXP](#)
- [Secure boot on i.MX 93](#)

This section uses the CST toolset. It can be obtained from [NXP](#).

5.2 Secure boot on i.MX 8M

5.2.1 System boot on i.MX 8M

The boot process starts after the device's power-on reset. The hardware logic forces the processor to start executing internal ROM code. Based on the state of the register `BOOT_MODE[13:0]` together with eFUSEs and GPIO pins (depends on configuration), the ROM code selects a boot device (Serial NOR flash via FlexSPI, NAND flash, SD/MMC, Serial (SPI) NOR). The boot process then continues executing the code from the boot device. ROM searches the Image Vector Table (IVT) on the address, which is based on the selected boot device. For example, `0x8400` for i.MX 8M Mini SD/eMMC boot. There it finds an entry point for the code jump. For more details, see [i.MX 8M Applications Processor Reference Manual](#).

5.2.2 System boot components

There are many software components involved in the boot process to run some complex operating systems, including Windows. This project uses U-Boot SPL as the first stage bootloader (also called Secondary Program Loader, SPL). On i.MX 8M, the processor has limited access to peripherals when exiting the ROM code area, since most of them are not initialized. Therefore, the first stage bootloader must fit the system's on-chip RAM (OCRAM). Its main purpose is to initialize DDR to get access to full system memory and to load a proper second stage bootloader. The first stage boot loader and second stage bootloader are considered SoC/firmware bootloaders, whereas UEFI provides an environment for Microsoft and OEMs.

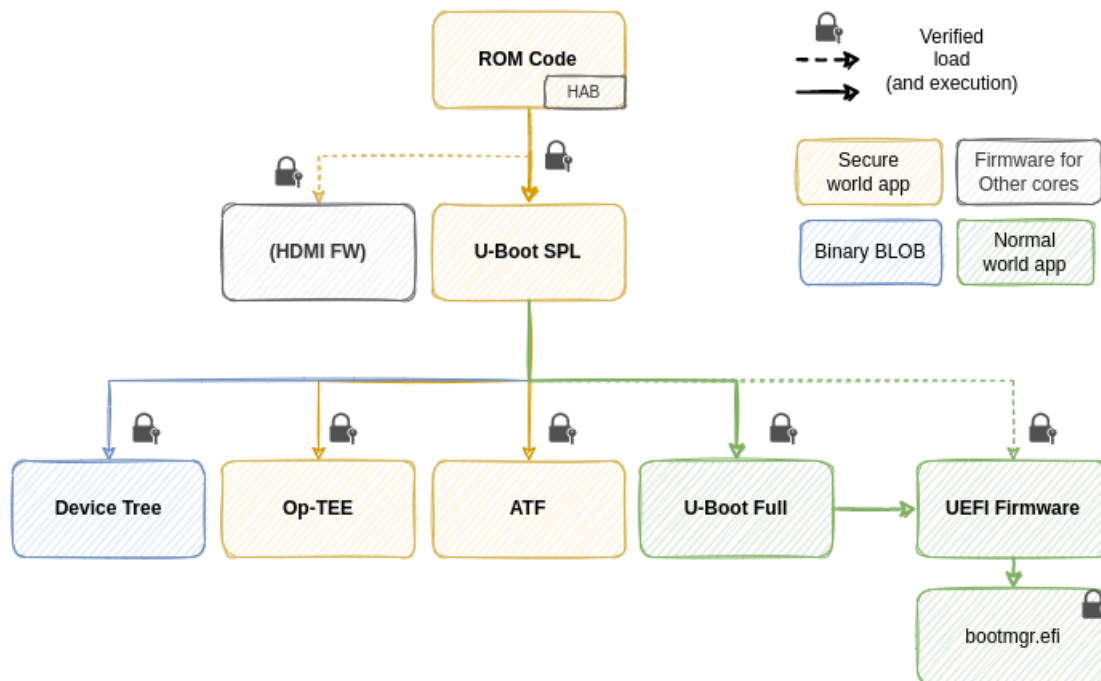


Figure 4. Boot flow

5.2.2.1 U-Boot SPL

This project uses [U-Boot SPL](#) as the first-stage bootloader. The main purpose of U-Boot SPL is to initialize the external memory that is needed to run proper U-Boot. The U-Boot SPL loads a few more components that participate in the configuration and security of the device - Device Tree blob, OP-TEE, and ARM Trusted firmware.

5.2.2.2 Device Tree Blob

Device Tree Blob (DTB) is a binary representation of [Device Tree](#). Device Tree is a data format for the description of the system hardware in a format of the tree of device nodes. The format is understood (and required) for example, by U-Boot proper and the Linux kernel. Thanks to Device Tree, a single program binary can support multiple platforms, just by changing the DTB that is used.

5.2.2.3 OP-TEE

Open Portable Trusted Execution Environment ([OP-TEE](#)) is an opensource implementation of the Trusted Execution Environment using ARM TrustZone technology. It provides a way of running applications within the secure world. This project uses OP-TEE as a runtime environment for [fTPM](#) and Authenticated Variables.

5.2.2.4 ATF

ARM Trusted Firmware is an implementation of firmware running with elevated privileges (EL3) and is used mostly as a proxy between the OS running in the non-secure world and OP-TEE running in the secure world.

5.2.2.5 U-Boot proper

The [U-Boot proper](#) is used in this project to perform early display initialization and load the UEFI bootloader. When enabled (disabled by default), the U-Boot provides a powerful CLI interface and can serve as a tool for device provisioning and/or debugging.

5.2.2.6 UEFI

The Unified Extensible Firmware Interface (UEFI) is a specification defining a unified interface between the firmware and the OS. UEFI firmware does the rest of the initialization and hands off the control to the Windows Boot Manager.

For more details, see [Boot and UEFI](#).

5.2.3 Ensuring firmware security

To ensure integrity and to prove the genuinity of all boot components, they must be signed, and the validity of the digital signature must be verified before passing the control to the next stage of the boot.

5.2.3.1 Security configuration

The reaction of the chip on various security events is massively dependent on its security configuration that may be affected by several fuses and [HAB](#).

5.2.3.1.1 Open/Closed

The open/closed state determines whether SECO allows execution of unauthenticated program images. Open chip allows execution of any program image - unauthenticated images and authenticated images with bad signature. Closed chip allows only execution of authenticated images. The state is defined by the `SEC_CONFIG[1:0]` eFUSE:

Fuse value	Effect
00	Reserved
01	Open
1x	Closed

5.2.3.1.2 SRKH

The Super Root Key Hash (SRKH) is a set of 8 eFUSES that contain a combined hash of hashes of particular Super Root Keys. They are one of the main components of the [HAB chain of trust](#).

5.2.3.2 Bootloader verification chain

All firmware signatures are generated at build time using private keys from the HAB chain of trust.

1. ROM Code verifies U-Boot SPL
2. U-Boot SPL checks [Device Tree Blob](#), [ATF](#), [OP-TEE](#), [U-Boot proper](#), and [UEFI](#)
3. UEFI checks efi modules and Windows Boot Manager

ROM code cannot be changed and is considered trusted. To verify the signature of SPL, the ROM contains a module called High Assurance Boot.

HAB is a software component responsible for verifying digital signatures. Its API is available to external applications via the ROM vector table (RVT). Before jumping to SPL, ROM verifies the signature of SPL. Only a valid SPL signature allows the booting flow to proceed (see [Chip lifecycle](#)).

Once loaded and verified, **U-Boot SPL** is also considered secure and trusted. U-Boot SPL loads the container image containing [Device Tree Blob](#), [ATF](#), [OP-TEE](#), [U-Boot proper](#), and [UEFI](#). When building with `-t secured_efi`, the U-Boot SPL verifies the signature of each component of the FIT image. The U-Boot SPL will proceed to the proper U-Boot only with the matching signature. The verification is realized by the HAB module.

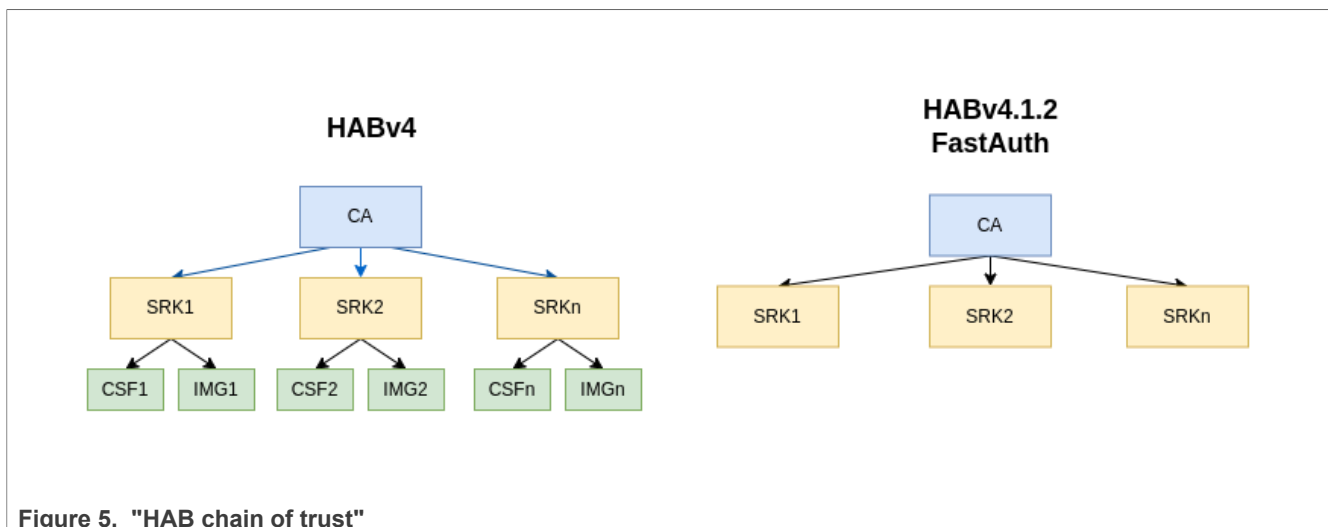
5.2.3.3 HAB chain of trust

HAB chain of trust is a set of certificates and keys, forming the Public Key Infrastructure (PKI) used for signing and verification of Secure Boot components handled by HAB. This repository contains a pregenerated PKI. To use your own PKI, point the environment variable `KEY_ROOT` to your key root folder.

Important: Building BSP with the default `KEY_ROOT` produces signed, but not secure binaries since they are signed with well-known keys!

NXP provides a set of tools, called [CST](#) that helps with generating custom PKI and signing.

The HAB chain of trust consists of single Certification Authority (CA), four Super Root Keys (SRK) and (optionally) four image (IMG) keys and command sequence file (CSF) keys. Depending on the HAB version, firmware images and CSFs can be signed directly by SRK (HABv4) or by IMG and CSF keys that are signed by an appropriate SRK (HABv4.1.2 FastAuth).



5.2.3.4 i.MX firmware image verification

Even though the SECO (AHAB) is responsible for signature verification, the verification key itself cannot be burned to eFUSES since there are not enough of them. To circumvent that, only a footprint of the key is written to the device. The verification key itself is then packed along the signature to the firmware binary. HAB then verifies the key against the footprint and uses the key to verify the signature. This information is stored in the CSF block, see [Figure 5](#)

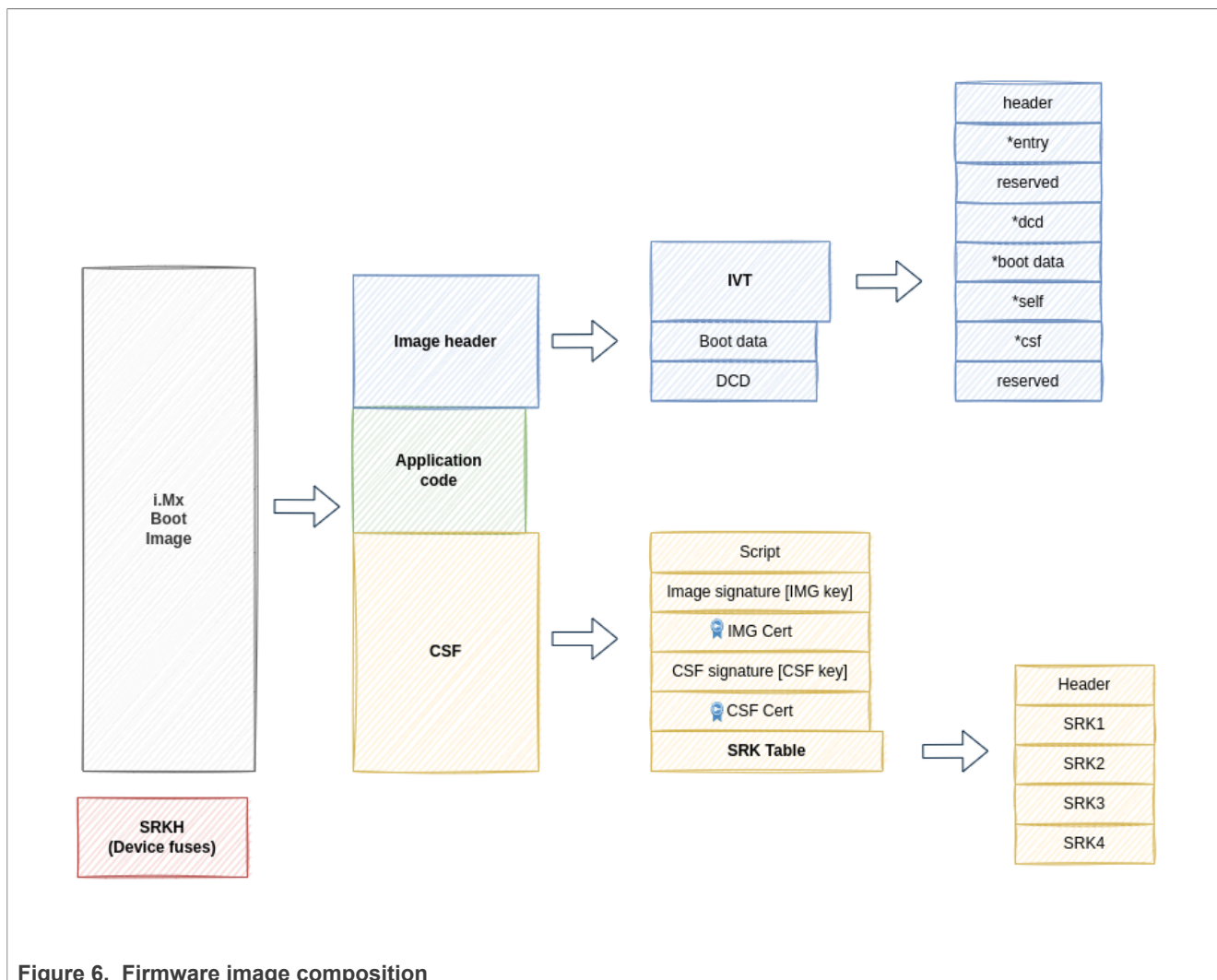


Figure 6. Firmware image composition

When HAB verifies the signature of the i.MX firmware image, the steps are as follows:

1. Get the CSF location from IVT.
2. Extract the SRK table from CSF.
3. Compute SRKH and verify against fuses. Break if invalid.
4. Verify the CSF and IMG certificates by appropriate SRK from the SRK table. Break if invalid.
5. Verify the CSF signature and the image signature.

5.3 Secure boot on i.MX 8QXP

5.3.1 System boot on i.MX 8QXP

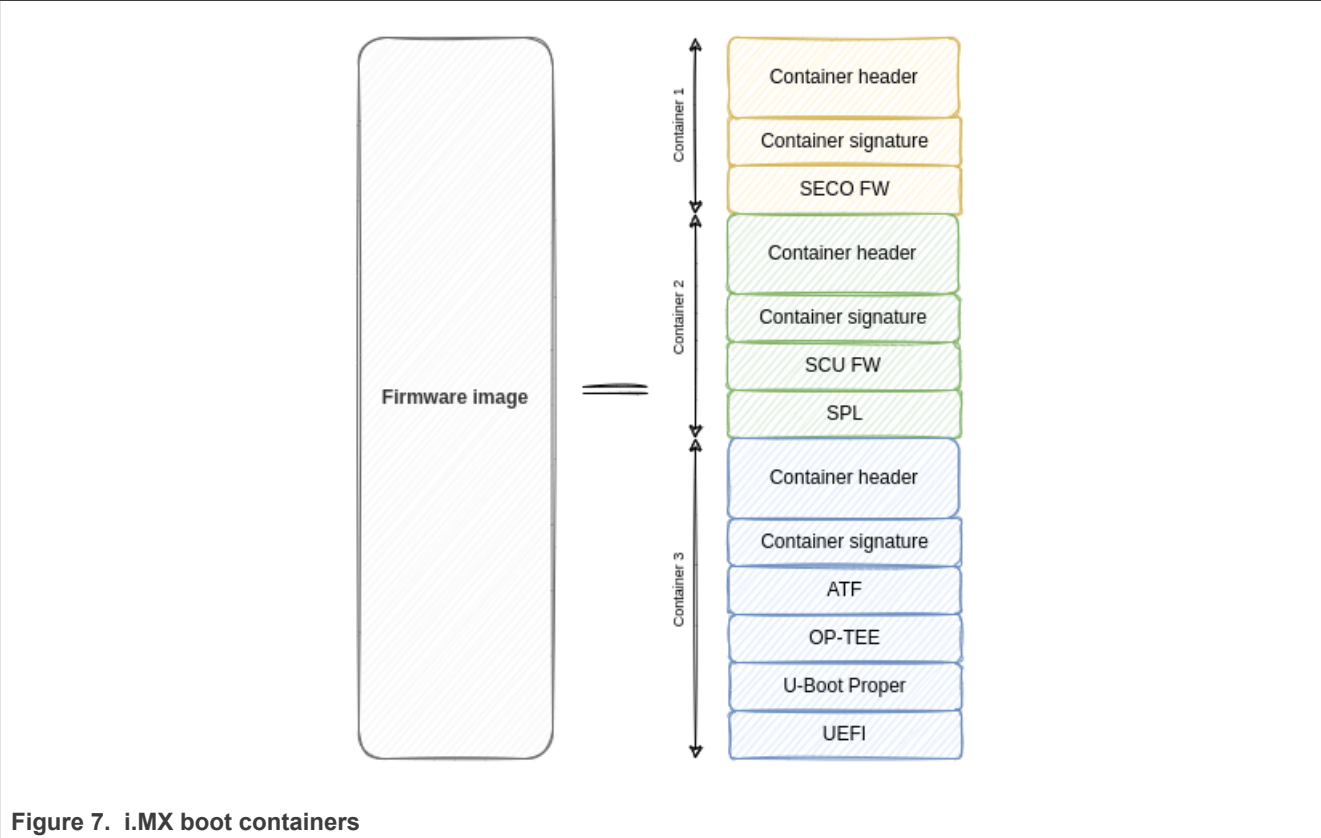
Compared to i.MX 8M, where the system boots from ARM Cortex-A cores from ROM memory, i.MX 8QXP starts its boot in a dedicated security subsystem (SECO) and a system control unit (SCU). They are separate ARM Cortex-M cores that run their own code, starting in their respective ROMs. During the boot process, there is a firmware downloaded for each of these cores, where their program flow continues. Then, firmware for other system cores is loaded. SCU ROM code selects a boot device (SD/MMC, NAND flash, FlexSPI NOR flash, Serial downloader on USB) based on `SCU_BOOT_MODE` pins and `Force Boot From Fuse` efuse. The first

stage bootloader for application cores may then be loaded directly to RAM (compared to i.MX 8M, which needs an SPL that will fit into OCRM and set up the DDR first).

For more details, see chapter 5.5 Secure Boot Flow with SCU and SECO in [i.MX DualX/8DualXPlus/8QuadXPlus Applications Processor Reference Manual](#)

5.3.2 i.MX boot containers

Application images that participate in the i.MX 8QXP system boot are packed into so-called containers and **at least** two of them are needed to boot the board. The first one is provided and signed by NXP and contains SECO FW. The second one contains SCU FW and application code for other cores. Each container consists of the container header, the container signature block (may be empty) and one or more images. Each image has its own header, which defines the load address and entry point. Containers are composed using the imx-mkimage tool.



5.3.3 System boot components

There are many software components involved in the boot process to run some complex operating systems, including Windows. This project uses U-Boot SPL as the first-stage bootloader (also called Secondary Program Loader, SPL). The first-stage bootloader and second-stage bootloader are considered SoC/firmware bootloaders, whereas UEFI provides an environment for Microsoft and OEMs.

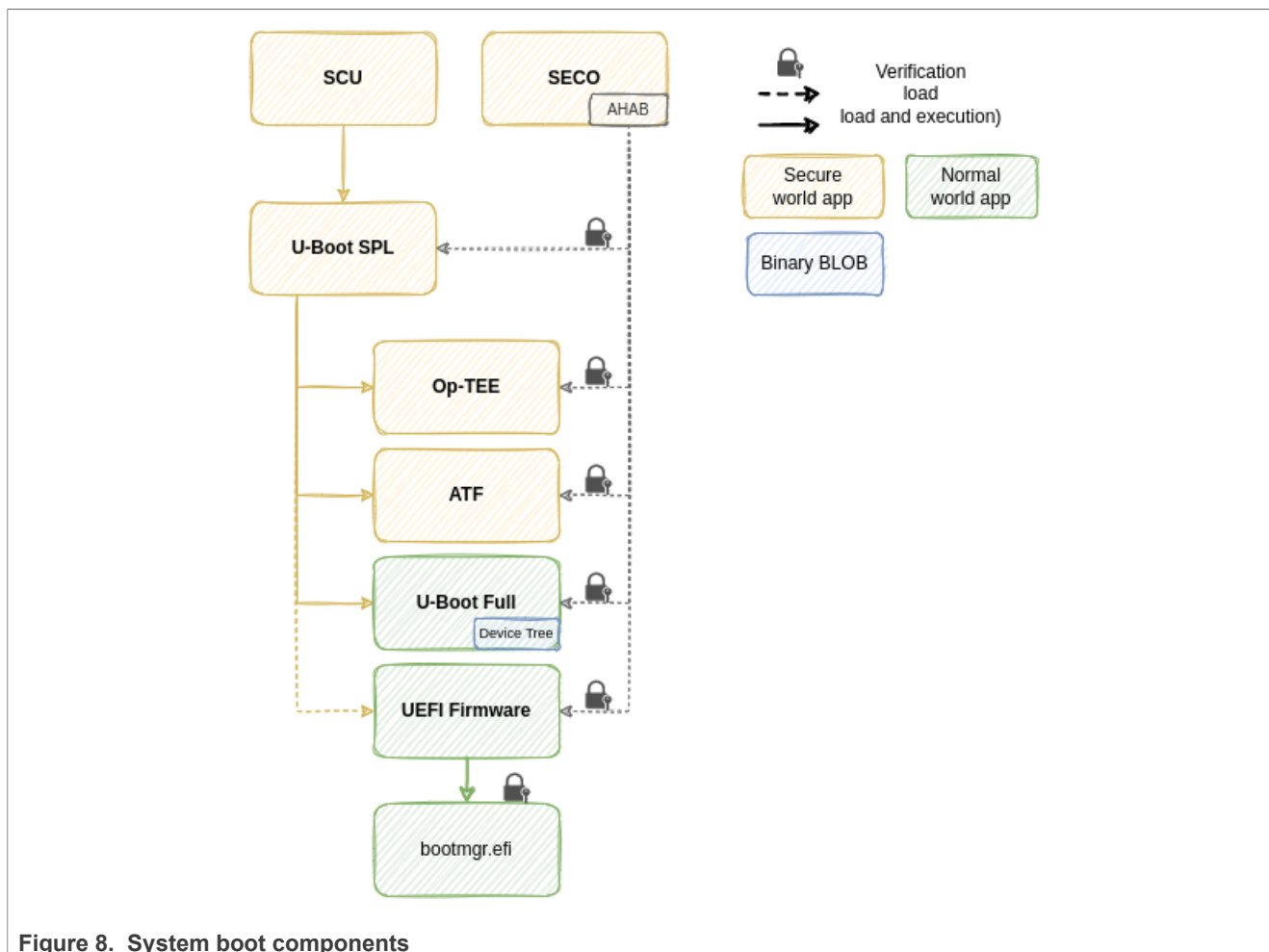


Figure 8. System boot components

5.3.3.1 U-Boot SPL

This project uses [U-Boot SPL](#) as the first-stage bootloader. The main purpose of U-Boot SPL is to initialize the external memory that is needed to run proper U-Boot. The U-Boot SPL actually loads a few more components that participate in the configuration and security of the device - Device Tree blob, OP-TEE, and ARM Trusted firmware.

5.3.3.2 Device Tree Blob

Device Tree Blob (DTB) is a binary representation of [Device Tree](#). Device Tree is a data format for the description of system hardware in a format of the tree of device nodes. The format is understood (and required) for example, by U-Boot proper and the Linux kernel. Thanks to Device Tree, a single program binary can support multiple platforms, just by changing the DTB that is used.

5.3.3.3 OP-TEE

Open Portable Trusted Execution Environment ([OP-TEE](#)) is an opensource implementation of the Trusted Execution Environment using ARM TrustZone technology. It provides a way of running applications within the secure world. This project uses OP-TEE as a runtime environment for [TPM](#) and Authenticated Variables.

5.3.3.4 ATF

ARM Trusted Firmware is an implementation of firmware running with elevated privileges (EL3) and is used mostly as a proxy between the OS running in the non-secure world and OP-TEE running in the secure world.

5.3.3.5 U-Boot proper

The [U-Boot](#) proper is used in this project to perform early display initialization and load the UEFI bootloader. When enabled (disabled by default), the U-Boot provides a powerful CLI interface and can serve as a tool for device provisioning and/or debugging.

5.3.3.6 UEFI

The Unified Extensible Firmware Interface (UEFI) is a specification defining a unified interface between the firmware and the OS. UEFI firmware does the rest of the initialization and hands off the control to the Windows Boot Manager.

For more details, see [Boot and UEFI](#).

5.3.4 Ensuring firmware security

To ensure integrity and to prove the genuinity of all boot components, they must be signed, and the validity of the digital signature must be verified before passing the control to the next stage of the boot.

5.3.4.1 Security configuration

The reaction of the chip on various security events is massively dependent on its security configuration that may be affected by several fuses and SECO.

5.3.4.1.1 Open/Closed

The open/closed state determines whether SECO allows execution of unauthenticated program images. Open chip allows execution of any program image - unauthenticated images and authenticated images with bad signature. Closed chip allows only execution of authenticated images. The state can be controlled, for example, from U-Boot cli via the ``ahab_status`` command. The status can be either ``NXP closed`` (open) or ``OEM closed`` (closed).

Example:

```
`=> ahab_status`  
  
`Lifecycle: 0x0020, NXP closed`
```

5.3.4.1.2 SRKH

The Super Root Key Hash (SRKH) is a set of 16 eFUSES (on i.MX 8QXP) that contain a combined hash of hashes of particular Super Root Keys. They are one of the main components of the [Advanced High Assurance Boot \(AHAB\) chain of trust](#).

5.3.4.2 Bootloader verification chain

All firmware signatures are generated at build time using private keys from the AHAB chain of trust.

1. SECO verifies Container 1 (SECO FW) and Container 2 (SCU FW+SPL)

2. SPL checks Container 3 ([ATF](#), [OP-TEE](#), [U-Boot proper](#), and [UEFI](#))
3. UEFI checks efi modules and Windows Boot Manager

SCU and SECO **ROM code** cannot be changed and is considered trusted. To verify the signature of SPL, SCU relies on SECO FW that does the signature check via its AHAB module. When the chip is closed, only a valid SPL signature allows booting flow to proceed (see Chip lifecycle). Once loaded and verified, **U-Boot SPL** is also considered secure and trusted. U-Boot SPL loads the container image containing [Device Tree Blob](#), [ATF](#), [OP-TEE](#), [U-Boot proper](#), and [UEFI](#). When building with `-t secured_efi`, the U-Boot SPL verifies the signature of each component of the FIT image. The U-Boot SPL proceeds to the proper U-Boot only when a matching signature is present. The SPL requests signature verification from SECO AHAB.

U-Boot proper asks SECO for signature verification of UEFI firmware. The binary was already checked by SPL since it is a part of the container that is loaded by SPL; however, U-Boot currently does not support partial signature checking (enabled in SPL, but disabled in U-Boot proper). The U-Boot proper transfers the control to UEFI.

5.3.4.3 AHAB chain of trust

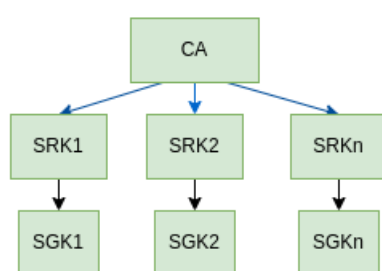
AHAB chain of trust is a set of certificates and keys, forming the Public Key Infrastructure (PKI) used for signing and verification of Secure Boot components handled by AHAB. This repository contains a pregenerated PKI. To use your own PKI, point the environment variable `KEY_ROOT` to your key root folder.

Important: Building BSP with the default `KEY_ROOT` will produce signed, but not secure binaries since they are signed with well-known keys!

NXP provides a set of tools, called [CST](#) that helps with generating custom PKI and signing.

The AHAB chain of trust consists of a single Certification Authority (CA), four Super Root Keys (SRK) and (optionally) four subordinate (SGK) keys. When using SGK keys (SRK generated with CA flag set), the firmware container is signed by the SGK key. Otherwise, the container is signed directly by the SRK key.

AHAB with subordinate SGK keys
Containers are signed by SGK
SRKs have CA flag enabled



AHAB
Containers are signed by SRK
SRKs have CA flag disabled

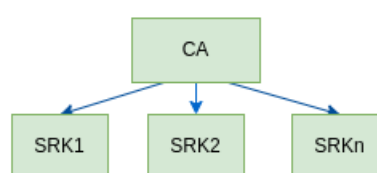


Figure 9. AHAB chain of trust

5.3.4.4 i.MX firmware image verification

Even though the SECO (AHAB) is responsible for signature verification, the verification key itself cannot be burned to eFUSES since there are not enough of them. To circumvent that, only a footprint of the key is written to the device. The verification key itself is then packed along with the signature to the firmware binary. SECO then verifies the key against the footprint and then uses the key to verify the signature.

When SECO verifies the signature of the i.MX firmware image, it does the following:

1. Get the SRK table location from the container signature header.

2. Extract the SRK table.
3. Compute SRKH and verify against fuses. Break if invalid.
4. (optional) Verify the SGK certificate by an appropriate SRK from the SRK table. Break if invalid.
5. Verify the container signature.

5.4 Secure boot on i.MX 93

5.4.1 System boot on i.MX 93

i.MX 93 boots from on-chip ROM code. Based on various fuse values and boot switches, the ROM selects the proper boot medium and flow. Secure aspects of the platform boot are handled by the EdgeLock secure enclave ROM (ELE). The boot ROM contains the Advanced High Assurance Boot (AHAB) library that enables secure boot functionality, with ELE as a backend. For more details, see chapter 8.1 Single Boot Flow (Cortex-A55) in i.MX 93 Applications Processor Reference Manual.

5.4.2 i.MX Boot Containers

Application images that participate in the i.MX 93 system boot are packed into so-called images and containers. A boot container may contain one or more boot images (A55 image, M33 image, and ELE FW). Each container consists of the container header, the container signature block (may be empty), and one or more images. Each image has its own load address and entry point. Containers are composed using the `imx-mkimage` tool.

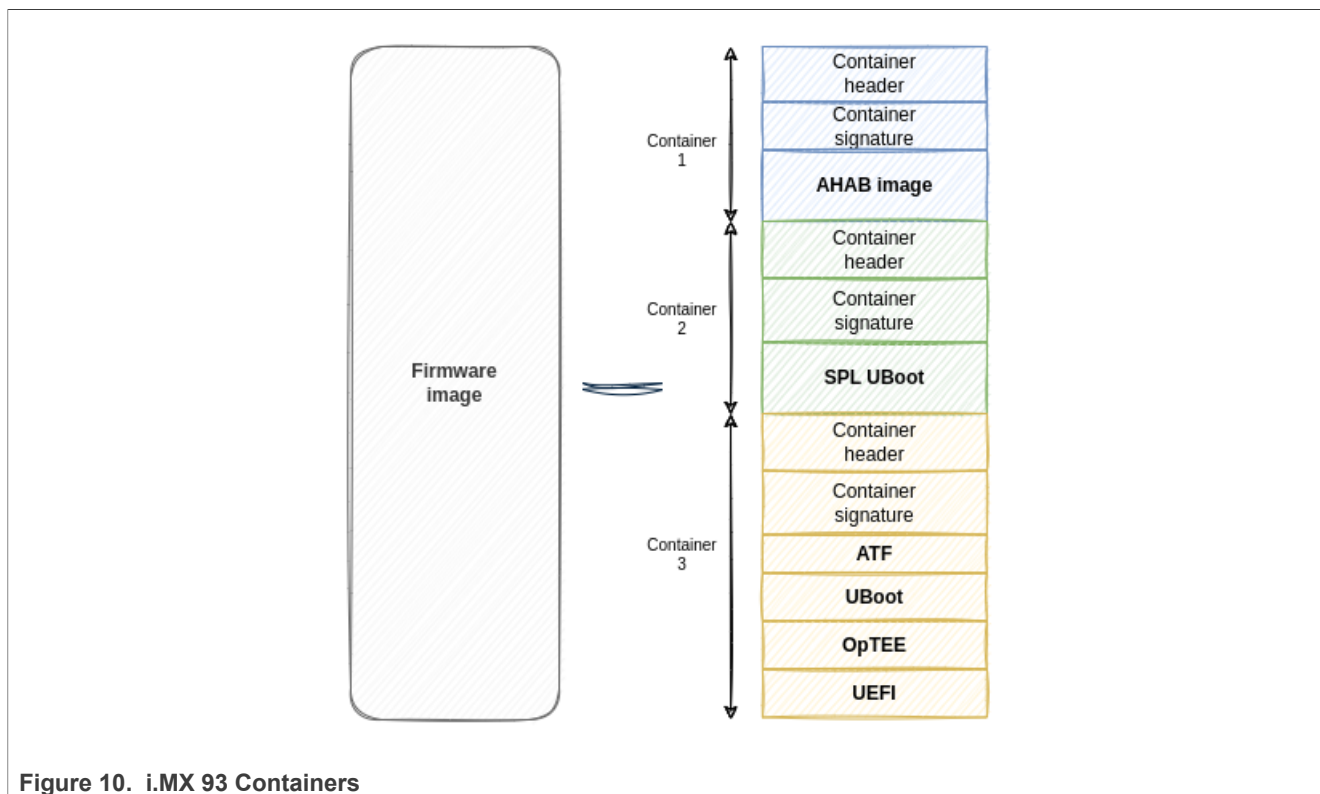
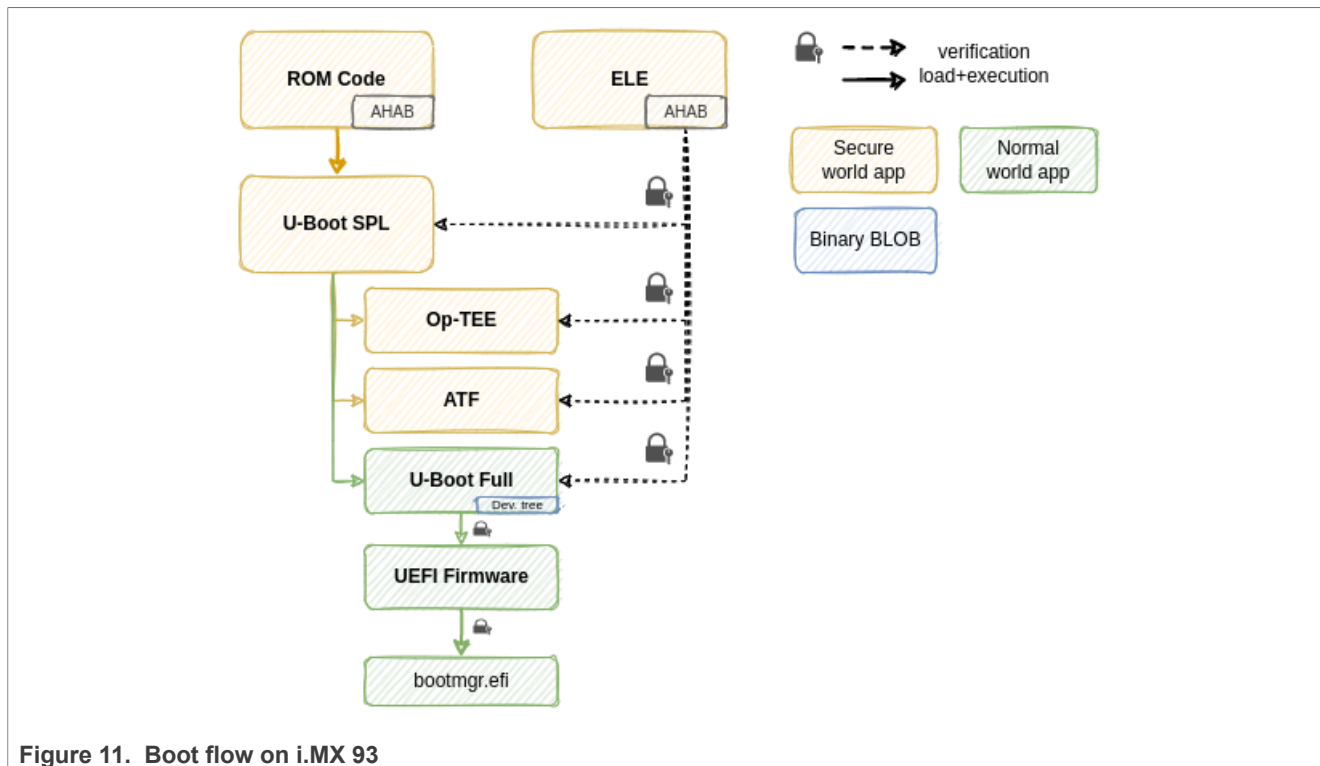


Figure 10. i.MX 93 Containers

5.4.3 System boot components

There are many software components involved in the boot process to run some complex operating systems, including Windows. This project uses U-Boot SPL as the first stage bootloader (also called Secondary Program Loader, SPL).

The first-stage bootloader and second-stage bootloader are considered SoC/firmware bootloaders, whereas UEFI provides an environment for Microsoft and OEMs.



5.4.3.1 U-Boot SPL

This project uses U-Boot SPL as the first stage bootloader. Compared to i.MX 8M, the SPL has access to full system memory. The purpose of SPL on i.MX 93 is to load other firmware components to non-continuous memory. The SPL understands the i.MX container format and loads the following components that participate in the configuration and security of the device to their respective load addresses: OP-TEE, ARM Trusted Firmware, U-Boot proper, and UEFI firmware.

5.4.3.2 Device Tree Blob

Device Tree Blob (DTB) is a binary representation of [Device Tree](#). Device Tree is a data format for the description of the system hardware in a format of the tree of device nodes. The format is understood (and required) for example, by U-Boot proper and the Linux kernel. Thanks to Device Tree, a single program binary can support multiple platforms, just by changing the DTB that is used.

5.4.3.3 OP-TEE

Open Portable Trusted Execution Environment ([OP-TEE](#)) is an opensource implementation of the Trusted Execution Environment using ARM TrustZone technology. It provides a way of running applications within the secure world. This project uses OP-TEE as a runtime environment for [fTPM](#) and Authenticated Variables.

5.4.3.4 ATF

ARM Trusted Firmware is an implementation of firmware running with elevated privileges (EL3) and is used mostly as a proxy between the OS running in the non-secure world and OP-TEE running in the secure world.

5.4.3.5 U-Boot proper

The [U-Boot](#) proper is used in this project to perform early display initialization and load the UEFI bootloader. When enabled (disabled by default), the U-Boot provides a powerful CLI interface and can serve as a tool for device provisioning and/or debugging.

5.4.3.6 UEFI

The Unified Extensible Firmware Interface (UEFI) is a specification defining a unified interface between the firmware and the OS. UEFI firmware does the rest of the initialization and hands off the control to the Windows Boot Manager.

For more details, see [Boot and UEFI](#).

5.4.4 Ensuring firmware security

To ensure integrity and to prove the genuinity of all boot components, they must be signed, and the validity of the digital signature must be verified before passing the control to the next stage of the boot.

5.4.4.1 Security configuration

The reaction of the chip on various security events is massively dependent on its security configuration that may be affected by several fuses and ELE.

5.4.4.1.1 Open/Closed

The open/closed state determines whether AHAB allows execution of unauthenticated program images. Open chip allows execution of any program image - unauthenticated images and authenticated images with bad signature. Closed chip allows only execution of authenticated images. The state can be controlled, for example, from U-Boot cli via the ``ahab_status`` command. The status can be either ``NXP closed`` (open) or ``OEM closed`` (closed).

Example:

```
`=> ahab_status`  
  
`Lifecycle: 0x0020, NXP closed`
```

5.4.4.1.2 SRKH

The Super Root Key Hash (SRKH) on i.MX 93 is the SHA256 hash of the SRK table. The SRKH is stored in a set of 8 32-bit eFUSES that contain the hash of the SRK table, containing the Super Root Keys. They are one of the main components of the [Advanced High Assurance Boot \(AHAB\) chain of trust](#).

5.4.4.2 Bootloader verification chain

All firmware signatures are generated at build time using private keys from the AHAB chain of trust.

1. ELE verifies Container 1 (AHAB) and Container 2 (SPL)
2. SPL checks Container 3 ([ATF](#), [OP-TEE](#), [U-Boot proper](#), and [UEFI](#))
3. UEFI checks efi modules and Windows Boot Manager

ROM code cannot be changed and is considered trusted. To verify the signature of SPL, ROM relies on ELE that does the signature check via its AHAB module. When the chip is closed, only a valid SPL signature allows

booting flow to proceed (see Chip lifecycle). Once loaded and verified, **U-Boot SPL** is also considered secure and trusted. U-Boot SPL loads the container image containing [Device Tree Blob](#), [ATF](#), [OP-TEE](#), [U-Boot proper](#), and [UEFI](#). When building with `-t secured_efi`, the U-Boot SPL verifies the signature of each component of the FIT image. The U-Boot SPL proceeds to the proper U-Boot only when a matching signature is present. The SPL requests signature verification from ELE/AHAB.

5.4.4.3 AHAB chain of trust

AHAB chain of trust is a set of certificates and keys, forming the Public Key Infrastructure (PKI) used for signing and verification of Secure Boot components handled by AHAB. This repository contains a pregenerated PKI. To use your own PKI, point the environment variable `KEY_ROOT` to your key root folder.

Important: Building BSP with the default `KEY_ROOT` will produce signed, but not secure binaries since they are signed with well-known keys!

NXP provides a set of tools, called [CST](#) that helps with generating custom PKI and signing.

The AHAB chain of trust consists of the single Certification Authority (CA), four Super Root Keys (SRK) and (optionally) four subordinate (SGK) keys. When using SGK keys (SRK generated with CA flag set), the firmware container is signed by the SGK key. Otherwise, the container is signed directly by the SRK key.

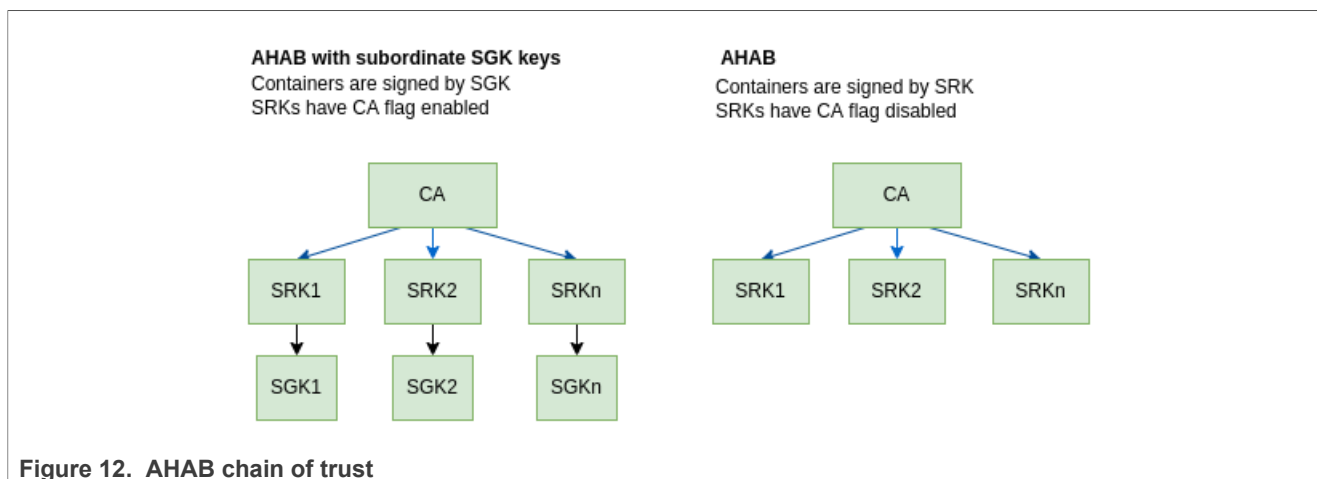


Figure 12. AHAB chain of trust

5.4.4.4 i.MX firmware image verification

Even though the ELE (AHAB) is responsible for signature verification, the verification key itself cannot be burned to eFUSES since there are not enough of them. To circumvent that, only a footprint of the key is written to the device. The verification key itself is then packed along with the signature to the firmware binary. ELE then verifies the key against the footprint and then uses the key to verify the signature.

When ELE verifies the signature of the i.MX firmware image, it does the following:

1. Get the SRK table location from the container signature header.
2. Extract the SRK table.
3. Compute SRKH and verify against fuses. Break if invalid.
4. (optional) Verify the SGK certificate by an appropriate SRK from the SRK table. Break if invalid.
5. Verify container signature.

5.5 Secure storage

There is numerous sensitive information in the system that must be stored securely - credentials, cryptographic keys, and so on. They may be both volatile and non-volatile and must be hidden not only from other applications

running under Windows, but also from other operating systems and peripherals. One example is Authenticated Variables (AuthVars) functionality, described in the UEFI specification. This mechanism is used for storing sensitive system data. Only authenticated issuers may read and modify these data. AuthVars are also used for storing UEFI provisioning data (PK, KEK, db, dbx).

5.5.1 RPMB

This repository uses the Replay Protected Memory Block (as defined in JEDEC eMMC specification JESD84-B51) as a secure storage backend. RPMB is a special partition on eMMC memory where every read or write operation must be authenticated. RPMB access is replay protected in a way, that every operation contains a signature (MAC), that contains an incremental write counter. The signature is generated using a symmetric key that is burned in the eMMC controller and must be known also by the issuer of the command. The process of writing the key to the eMMC controller is a one-way process and the key is written in plaintext. Therefore, it must be done in a secure environment.

RPMB is mandatory for this system to work since it is used as a secure storage backend for OP-TEE (and OP-TEE is used by UEFI for storing AuthVars). For more information on how OP-TEE uses RPMB, see the [following link](#)

5.5.2 Secure vs. non-secure build

The firmware binary can be built in two setups based on flags passed to the `buildme64.sh` script:

- Secure build - when building **with** `-t secured_efi` or `-t secured_uefi`
- Non-Secure build - when building **without** `-t secured_efi` or `-t secured_uefi`

Note: The effect of `-t secured_efi` is identical to `-t secured_uefi`, both parameters are interchangeable.

5.5.2.1 Secure build

Secure build provides secure binaries with all Secure Boot dependencies enabled. UEFI firmware is built with the support of Secure Boot and AuthVars and Measured Boot is enabled. All firmware binaries are signed during the build (U-Boot SPL, DT, OP-TEE, ATF, U-Boot, UEFI) and signature checks in U-Boot SPL are enforced.

Note: Secure firmware binary will not boot on a clean device. To boot the secure firmware binary, the RPMB key must be present in the eMMC controller. Otherwise, the initialization of OP-TEE and all dependencies will fail. For more information, see [Secure provisioning](#).

5.5.2.2 Non-Secure build

Non-secure build provides an easy way for testing and prototyping. In this setup, firmware binaries are not signed and SPL signature checks are disabled. The Secure Boot, AuthVars, and Measured Boot are disabled. This setup boots even without the RPMB key provisioned (for example, a new device).

5.6 Secure Boot in UEFI and Windows

UEFI and Windows use their own chain of trust, which is composed of Platform Key (PK), Key Exchange Key (KEK), forbidden signature database (dbx) and valid signature database (db). Those credentials are stored as UEFI Secure variables. Those variables must be programmed at the OEM site.

Important: Even when building with `-t secured_efi`, the boot chain is not fully secured until PK is written. Until then, the UEFI and Windows are in setup mode where signatures are not checked.

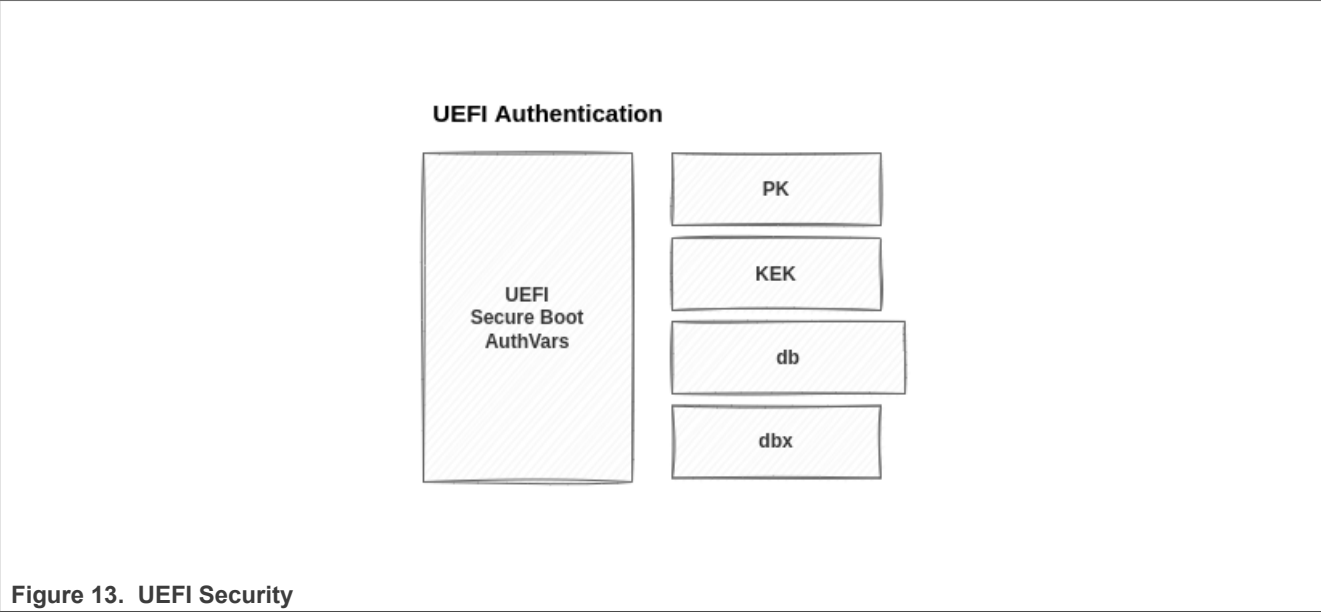


Figure 13. UEFI Security

6 Secure provisioning

To achieve full system security with Secure Boot, perform the following steps in the correct order:

1. Prepare keys for HAB/AHAB.
2. Lock the device (burn SRKH and SEC_CONFIG fuses).
3. Write the RPMB key.
4. Boot the device and load the UEFI keys.

There are many ways to generate HAB/AHAB keys. This guide presents a simple way to generate HAB/AHAB keys using the [CST](#) toolset. After download, see User Guide in `<cst_directory>/docs/CST_UG.pdf`.

For detailed steps, follow device-specific guides:

[Secure provisioning i.MX 8M](#)

[Secure provisioning i.MX 8QXP](#)

[Secure provisioning i.MX 93](#)

6.1 Secure provisioning i.MX 8M

These steps are only applicable for the i.MX 8M family. Use an appropriate [guide](#) for your platform, otherwise you risk bricking your device.

6.1.1 Generate HAB keys

Generate the PKI using the `keys/hab4_pki_tree.sh` script. Use the following options to generate four p256 ECC SRK and four IMG and CSF keys.

```
cd <cst_directory>/keys
./hab4_pki_tree.sh
Do you want to use an existing CA key (y/n)? : n
Do you want to use Elliptic Curve Cryptography (y/n)? : n
Enter key length in bits for PKI tree: 4096
Enter PKI tree duration (years): 10
How many Super Root Keys should be generated? 4
Do you want the SRK certificates to have the CA flag set? (y/n)? : y
```

The script populates the `keys` and `certs` folders within the CST root folder with private keys and appropriate certificates. Set the `KEY_ROOT` environment variable to absolute path to the CST root folder (the folder containing `keys` and `certs` subfolders).

```
export KEY_ROOT=<cst_directory>
```

Build will automatically fetch keys and certificates from this path to sign firmware binaries.

6.1.1.1 Prepare SRK table

CST provides `srktool` to prepare the SRK table. SRKH values must be written to fuses.

```
cd <cst_directory>/certs
../linux64/bin/srktool -h 4 -t SRK_1_2_3_4_table.bin -e SRK_fuse.bin -d sha256 -
c
./SRK1_sha256_4096_65537_v3_ca.crt.pem, ./SRK2_sha256_4096_65537_v3_ca.crt.pem, ./
SRK3_sha256_4096_65537_v3_ca.crt.pem, ./SRK4_sha256_4096_65537_v3_ca.crt.pem
```

```
-f 1
```

The program prints a summary with the following results:

```
Number of certificates      = 4
SRK table binary filename = SRK_1_2_3_4_table.bin
SRK Fuse binary filename  = SRK_fuse.bin
SRK Fuse binary dump:
SRK HASH[0] = 0x17B73726
SRK HASH[1] = 0x8E5CCC0E
SRK HASH[2] = 0xBC30A7BE
SRK HASH[3] = 0xE9B59C78
SRK HASH[4] = 0x2C682DAE
SRK HASH[5] = 0xDE5FE6C0
SRK HASH[6] = 0x3FF3DC81
SRK HASH[7] = 0x44B5B6FE
```

The SRK HASH[] array contains the SRKH value divided by four bytes. These are the values that are written to SRK_HASH eFUSE in the next step.

For more information on how to use `srktool`, see chapter 3.1.3 Generating HAB4 SRK tables and Efuse Hash in `<cst_directory>/docs/CST_UG.pdf`

6.1.2 Building secured binary

With HAB/AHAB keys prepared, you are able to build a signed secure binary. Build the firmware with `-t secured_efi` or `-t secured_uefi` flag enabled, for example:

```
./buildme64.sh -b 8Mm -t all -t secured_efi -nu
```

The command above produces `win10-iot-bsp/imx-windows-iot/BSP/firmware/MX8M_MINI_EVK/signed_flash.bin`, which is a signed binary image containing all boot components.

6.1.3 Locking the device for i.MX 8M

Warning: The steps described in this section are irreversible. Always make sure you know what you are doing. Any misconfiguration may lead to a bricked device.

This guide uses U-Boot's command-line interface as a tool for burning eFUSES.

6.1.4 Burning SRK_HASH

1. Load a stock image to the SD card (non-secure build).
2. Enter U-Boot command-line interface (press escape on prompt during boot).
3. To find proper fuse indexes, see the fuse map for your device.
4. Burn SRK_HASH fuses with values from `srktool`- see `SRK_fuse.bin`. Use the output values of `srktool` or use the following command: `hexdump -e '/4 "0x"' -e '/4 "%X""\n"' < SRK_fuse.bin`

Example for i.MX 8M Mini:

```
fuse prog -y 6 0 0x17B73726
fuse prog -y 6 1 0x8E5CCC0E
fuse prog -y 6 2 0xBC30A7BE
fuse prog -y 6 3 0xE9B59C78
fuse prog -y 7 0 0x2C682DAE
fuse prog -y 7 1 0xDE5FE6C0
fuse prog -y 7 2 0x3FF3DC81
fuse prog -y 7 3 0x44B5B6FE
```

```
reset
```

The device now contains an SRK Hash composed of your PKI keys and is able to verify firmware binary signatures. Until locked, the device accepts unsigned binaries and binaries with bad signature.

Tip: Before locking the chip, boot a signed image from the step [Building secured binary](#) and check HAB events:

1. Prepare an SD card with secured binary.
2. Enter the U-Boot command line.
3. Enter the `hab_status` command.

The command must output the following text, saying that all signatures are valid:

```
Secure boot enabled
HAB Configuration: 0xcc, HAB State: 0x99
No HAB Events Found!
```

6.1.5 Burning SEC_CONFIG

1. Load a stock image to the SD card (non-secure build).
2. Enter U-Boot command-line interface (press escape on prompt during boot).
3. To find proper fuse indexes, see the fuse map for your device.
4. Burn the `SEC_CONFIG` fuse to achieve the "closed" state.

Example for i.MX 8M Mini:

```
fuse prog 1 3 0x02000000
reset
```

The chip is now locked and accepts only firmware signed with appropriate keys.

6.2 Secure provisioning i.MX 8QXP

These steps are only applicable for the i.MX 8QXP family. Use an appropriate [guide](#) for your platform otherwise you risk bricking your device.

6.2.1 Generate AHAB keys

This section presents a way of generating AHAB keys. They are only applicable for i.MX 8QXP and i.MX 93 SoC.

Start by running the `keys/ahab_pki_tree.sh` script, use the following options to generate four p384 ECC SRK with CA flag disabled (SRK used for container signing).

```
cd <cst_directory>/keys
./ahab_pki_tree.sh
Do you want to use an existing CA key (y/n)? : n
Do you want to use Elliptic Curve Cryptography (y/n)? : y
Enter length for elliptic curve to be used for PKI tree:
Possible values p256, p384, p521: p384
Enter the digest algorithm to use: sha384
Enter PKI tree duration (years): 5
Do you want the SRK certificates to have the CA flag set? (y/n)? : n
```

The script populates the `keys` and `certs` folders within the CST root folder with private keys and appropriate certificates. Set the `KEY_ROOT` environment variable to the absolute path to the CST root folder (the folder containing `keys` and `certs` subfolders).

```
export KEY_ROOT=<cst_directory>
```

Build automatically fetches keys and certificates from this path to sign firmware binaries.

6.2.1.1 Prepare SRK table

CST provides `srktool` to prepare the SRK table. SRKH values that must be written to fuses.

```
cd <cst_directory>/certs
../linux64/bin/srktool -a -s sha384 -t SRKtable.bin -e SRKfuse.bin -f 1 -c
SRK1_sha384_secp384r1_v3_usr_cert.pem,SRK2_sha384_secp384r1_v3_usr_cert.pem,SRK3_sha384_secp384r1_v3_usr_cert.pem
```

The program prints a summary with the following results:

```
Number of certificates      = 4
SRK table binary filename  = SRKtable.bin
SRK Fuse binary filename   = SRKfuse.bin
SRK Fuse binary dump:
SRK HASH[0] = 0x336D1608
SRK HASH[1] = 0xDFCC2D5E
SRK HASH[2] = 0xB582FA14
SRK HASH[3] = 0xDA325A05
SRK HASH[4] = 0xEAB66EDE
SRK HASH[5] = 0xB64F7A87
SRK HASH[6] = 0xC9CAD3BF
SRK HASH[7] = 0x479DC210
SRK HASH[8] = 0x79DA681C
SRK HASH[9] = 0x8C55E093
SRK HASH[10] = 0x3CF9CF19
SRK HASH[11] = 0xC7B6DDF0
SRK HASH[12] = 0xE0C3363E
SRK HASH[13] = 0x73D8A971
SRK HASH[14] = 0x240A0EEE
SRK HASH[15] = 0xE46CE431
```

The `SRK_HASH[]` array contains the SRKH value divided by four bytes. These are the values that will be written to `SRK_HASH` eFUSE in the next step (applicable only for i.MX 8QXP).

For more information on how to use `srktool`, see chapter 3.2.3 Generating AHAB SRK tables and Efuse Hash in `<cst_directory>/docs/CST_UG.pdf`

6.2.2 Building secured binary

With HAB/AHAB keys prepared, you are able to build a signed secure binary. Build the firmware with `-t secured_efi` or `-t secured_uefi` flag enabled, for example:

```
./buildme64.sh -b 8Mm -t all -t secured_efi -nu
```

The command above produces `win10-iot-bsp/imx-windows-iot/BSP/firmware/MX8M_MINI_EVK/signed_flash.bin`, which is a signed binary image containing all boot components.

6.2.3 Locking the device (i.MX 8QXP)

Warning: CAUTION: The steps described in this section are irreversible. Always make sure you know what you are doing. Any misconfiguration may lead to a bricked device.

The following steps are only applicable for i.MX 8QXP and i.MX 93 SoC. For i.MX 8M, see section [Locking the device \(i.MX 8M\)](#) above.

This guide uses U-Boot's command-line interface as a tool for burning eFUSES.

6.2.3.1 Burning SRK_HASH

1. Load a stock image to the SD card.
2. Enter U-Boot command-line interface (press escape on prompt during boot).
3. To find proper fuse indexes, see the fuse map for your device.
4. Burn SRK_HASH fuses with values from `srktool` - see `SRKfuse.bin`. Use the output values of `srktool` or use the following command: `hexdump -e '/4 "0x"' -e '/4 "%X""\n"' < SRKfuse.bin`

```
#### For i.MX 8QXP only
#### Dump SRKH to console
hexdump -e '/4 "0x"' -e '/4 "%X""\n"' < SRKfuse.bin
0x336D1608
0xDFCC2D5E
0xB582FA14
0xDA325A05
0xEAB66EDE
0xB64F7A87
0xC9CAD3BF
0x479DC210
0x79DA681C
0x8C55E093
0x3CF9CF19
0xC7B6DDF0
0xE0C3363E
0x73D8A971
0x240A0EEE
0xE46CE431
```

```
#### For i.MX 8QXP only
#### Write values to fuses via UBoot CLI
fuse prog 0 730 0x336d1608
fuse prog 0 731 0xdfcc2d5e
fuse prog 0 732 0xb582fa14
fuse prog 0 733 0xda325a05
fuse prog 0 734 0xeab66ede
fuse prog 0 735 0xb64f7a87
fuse prog 0 736 0xc9cad3bf
fuse prog 0 737 0x479dc210
fuse prog 0 738 0x79da681c
fuse prog 0 739 0x8c55e093
fuse prog 0 740 0x3cf9cf19
fuse prog 0 741 0xc7b6ddf0
fuse prog 0 742 0xe0c3363e
fuse prog 0 743 0x73d8a971
fuse prog 0 744 0x240a0eee
fuse prog 0 745 0xe46ce431
reset
```

The device now contains an SRK Hash composed of your PKI keys and is able to verify firmware binary signatures. Until locked, the device accepts unsigned binaries and binaries with bad signature.

Tip: Before locking the chip, boot a signed image from the step [Building secured binary](#) and check AHAB events:

1. Prepare the SD card with secured binary.
2. Enter the U-Boot command line.
3. Enter the `ahab_status` command.

The command must output the following text, indicating that all signatures are valid:

```
=> ahab_status
Lifecycle: 0x0020, NXP closed
No SECO Events Found!
```

In case of any error, U-Boot prints and parse SECO events. Example for a missing signature:

```
=> ahab_status
Lifecycle: 0x0020, NXP closed
SECO Event[0] = 0x0087EE00
  CMD = AHAB_AUTH_CONTAINER_REQ (0x87)
  IND = AHAB_NO_AUTHENTICATION_IND (0xEE)
```

6.2.3.2 Closing the chip

1. Load a stock image to the SD card (non-secure build).
2. Enter U-Boot command-line interface (press escape on prompt during boot).
3. Close the chip and reboot.

Example:

```
=> ahab_close
=> reset
```

The chip is now locked and accepts only firmware signed with appropriate keys. You can check that via the `ahab_status` command, the lifecycle must be `0x80 OEM closed`.

```
=> ahab_status
Lifecycle: `0x80, OEM closed`
```

6.3 Secure provisioning i.MX 93

These steps are only applicable for i.MX 93 family. Use an appropriate [guide](#) for your platform otherwise you risk bricking your device.

6.3.1 Generate AHAB keys

This section presents a way of generating AHAB keys. They are only applicable for i.MX 8QXP and i.MX 93 SoC.

Start by running the `keys/ahab_pki_tree.sh` script, use the following options to generate four p384 ECC SRK with CA flag disabled (SRK used for container signing).

```
cd <cst_directory>/keys
./ahab_pki_tree.sh
Do you want to use an existing CA key (y/n)? : n
Do you want to use Elliptic Curve Cryptography (y/n)? : y
Enter length for elliptic curve to be used for PKI tree:
Possible values p256, p384, p521: p384
Enter the digest algorithm to use: sha384
Enter PKI tree duration (years): 5
Do you want the SRK certificates to have the CA flag set? (y/n)? : n
```

The script populates the `keys` and `crts` folders within the CST root folder with private keys and appropriate certificates. Set the `KEY_ROOT` environment variable to the absolute path to the CST root folder (the folder containing `keys` and `crts` subfolders).

```
export KEY_ROOT=<cst_directory>
```

Build automatically fetches keys and certificates from this path to sign firmware binaries.

6.3.1.1 Prepare SRK table

CST provides `srktool` to prepare the SRK table from which the SRKH value will be created.

```
cd <cst_directory>/crts
../linux64/bin/srktool -a -s sha384 -t SRKtable.bin -e SRKfuse.bin -f 1 -c
SRK1_sha384_secp384r1_v3_usr crt.pem,SRK2_sha384_secp384r1_v3_usr crt.pem,
SRK3_sha384_secp384r1_v3_usr crt.pem,SRK4_sha384_secp384r1_v3_usr crt.pem
```

The program prints a summary with the following results:

```
Number of certificates      = 4
SRK table binary filename  = SRKtable.bin
SRK Fuse binary filename   = SRKfuse.bin
SRK Fuse binary dump:
SRK HASH[0] = 0x336D1608
SRK HASH[1] = 0xDFCC2D5E
SRK HASH[2] = 0xB582FA14
SRK HASH[3] = 0xDA325A05
SRK HASH[4] = 0xEAB66EDE
SRK HASH[5] = 0xB64F7A87
SRK HASH[6] = 0xC9CAD3BF
SRK HASH[7] = 0x479DC210
SRK HASH[8] = 0x79DA681C
SRK HASH[9] = 0x8C55E093
SRK HASH[10] = 0x3CF9CF19
SRK HASH[11] = 0xC7B6DDF0
SRK HASH[12] = 0xE0C3363E
SRK HASH[13] = 0x73D8A971
SRK HASH[14] = 0x240A0EEE
SRK HASH[15] = 0xE46CE431
```

The `SRK HASH[]` is SHA-512 hash of the SRK table and is valid only for the i.MX 8QXP family (i.MX 93 needs SHA-256 format). SRKH for i.MX 93 will be prepared later.

For more information on how to use `srktool`, see chapter 3.2.3 *Generating AHAB SRK tables and Efuse Hash* in `<cst_directory>/docs/CST_UG.pdf`

6.3.2 Building secured binary

With HAB/AHAB keys prepared, you are able to build a signed secure binary. Build the firmware with `-t secured_efi` or `-t secured_uefi` flag enabled, for example:

```
./buildme64.sh -b 8Mm -t all -t secured_efi -nu
```

The command above produces `win10-iot-bsp/imx-windows-iot/BSP/firmware/MX8M_MINI_EVK/signed_flash.bin`, which is a signed binary image containing all boot components.

6.3.3 Locking the device

Warning: CAUTION: The steps described in this section are irreversible. Always make sure you know what you are doing. Any misconfiguration may lead to a bricked device.

The following steps are only applicable for i.MX 8QXP and i.MX 93 SoC. For i.MX 8M, see section [Locking the device \(i.MX 8M\)](#) above.

Note: The CST tool currently does not support the i.MX 93 SRKH format. It is therefore necessary to create the hash manually, follow [Preparing SRKH \(i.MX 93\)](#).

This guide uses U-Boot's command-line interface as a tool for burning eFUSES.

6.3.3.1 Preparing SRKH

1. Enter the folder containing your `SRKtable.bin`
2. Generate SRKH using the following command: `openssl dgst -sha256 -binary SRKtable.bin > SRKfuse93.bin`
3. Print the contents of SRKH in the format used for writing to fuses: `hexdump -e '/4 "0x"' -e '/4 "%X""\n"' < SRKfuse93.bin`

6.3.3.2 Burning SRK_HASH

1. Load a stock image to the SD card.
2. Enter U-Boot command-line interface (press escape on prompt during boot).
3. To find proper fuse indexes, see the fuse map for your device.
4. **i.MX 8QXP:** Burn `SRK_HASH` fuses with values from `srktool` - see `SRK_fuse.bin`. Use the output values of `srktool` or use the following command: `hexdump -e '/4 "0x"' -e '/4 "%X""\n"' < SRKfuse.bin`
5. **i.MX 93:** Burn `SRK_HASH` fuses with values from step 4 of "Preparing SRKH (imx93 only)" above.

```
#### For i.MX93 only
#### Dump SRKH to console
hexdump -e '/4 "0x"' -e '/4 "%X""\n"' < SRKfuse93.bin
0xA3B1A4B0
0x2AAEEEC5
0xCFC0D333
0xCC440EFC
0x73F4D517
0xC8D3F8A0
0xF8893889
```

```
0x42CF6504
```

```
#### For i.MX93 only
#### Write values to fuses via UBoot CLI
fuse prog -y 16 0 0xA3B1A4B0
fuse prog -y 16 1 0x2AAEEEC5
fuse prog -y 16 2 0xCFC0D333
fuse prog -y 16 3 0xCC440EFC
fuse prog -y 16 4 0x73F4D517
fuse prog -y 16 5 0xC8D3F8A0
fuse prog -y 16 6 0xF8893889
fuse prog -y 16 7 0x42CF6504
reset
```

The device now contains an SRK Hash composed of your PKI keys and is able to verify firmware binary signatures. Until locked, the device accepts unsigned binaries and binaries with bad signature.

Tip: Before locking the chip, boot a signed image from the step [Building secured binary](#) and check AHAB events:

1. Prepare the SD card with a secured binary.
2. Enter the U-Boot command line.
3. Enter the `ahab_status` command.

The command must output the following text, indicating that all signatures are valid:

```
=> ahab_status
Lifecycle: 0x0020, NXP closed

No SECO Events Found!
```

In case of any error, U-Boot prints and parses SECO events. Example for a missing signature:

```
=> ahab_status
Lifecycle: 0x0020, NXP closed

SECO Event[0] = 0x0087EE00
    CMD = AHAB_AUTH_CONTAINER_REQ (0x87)
    IND = AHAB_NO_AUTHENTICATION_IND (0xEE)
```

6.3.3.3 Closing the chip

1. Load a stock image to the SD card (non-secure build).
2. Enter U-Boot command-line interface (press escape on prompt during boot).
3. Close the chip and reboot.

Example:

```
=> ahab_close
=> reset
```

The chip is now locked and accepts only firmware signed with appropriate keys. You can check that via the `ahab_status` command, the lifecycle must be `0x80 OEM closed``.

```
=> ahab_status
Lifecycle: `0x80, OEM closed`
```

6.4 RPMB, UEFI

6.4.1 RPMB

The following steps for loading the RPMB key are only applicable with a device in the "closed" state.

The used OP-TEE implementation allows the use of the Hardware-Unique key (HUK) that is accessible only from software running in the secure world and therefore unreachable from a normal OS. This principle provides enhanced security since the key does not need to be stored in memory, it is generated on demand.

OP-TEE itself is able to burn the key, when built with `CFG_RPMB_WRITE_KEY=y`. The following steps guide you on how to prepare a "provisioning" build that contains OP-TEE with RPMB key provisioning enabled. OP-TEE uses HUK as RPMB key by default.

1. Rebuild the firmware using `./buildme64.sh -b <board-type> -t all -t secured_efi -ao rpmb_write_key -ao no_rpmb_test_key` and store the `signed_firmware.bin` separately. This firmware must be used only for RPMB provisioning (at a secured place).
2. Burn the provisioning `signed_firmware.bin` to the SD card and boot it.

OP-TEE automatically burns the RPMB key to the eMMC controller during first boot. The RPMB is now fully provisioned and the boot process should now be unblocked and proceed to UEFI and Windows. You can now use your production `signed_firmware.bin`. The boot chain is now secured up to UEFI firmware.

6.4.2 UEFI

Even with Secure Boot settings enabled, the UEFI firmware and Windows still reside in **setup mode**, where signatures are not checked. The UEFI automatically transfers to **user mode** with Secure Boot enabled when PK is written and the OS is restarted. For more details, see [Windows Secure Boot Key Creation and Management Guidance](#).

6.5 Troubleshooting

6.5.1 Firmware built as `secure` fails to boot or hangs in UEFI

There may be a problem with RPMB, either the RPMB key was not written yet, or a different key is used.

```
AUTH-VAR[0]:TEEC OpenSession() failed. (TeeResult=0xFFFF0007) (TaOpenSession: /bsp/mu_platform_nxp/Microsoft/OpTEEClientPkg/Drivers/AuthVarOpTEERuntimeDxe/AuthVarsDxe.c, 1027)
AUTH-VAR[0]:TaOpenSession() failed. (Status=Protocol Error) (VariableAuthOpTEERuntimeInitialize: /bsp/mu_platform_nxp/Microsoft/OpTEEClientPkg/Drivers/AuthVarOpTEERuntimeDxe/AuthVarsDxe.c, 1554)
Error: Image at 000E1EF0000 start failed: Protocol Error
Remove-symbol-file /bsp/mu_platform_nxp/Build/MEK_IMX8QXP_3GB/DEBUG_GCC5/AARCH64/Microsoft/OpTEEClientPkg/Drivers/AuthVarOpTEERuntimeDxe/AuthVarsDxe/DEBUG/VariableAuthOpTEERuntimeDxe.dll 0xE1F00000
AcpiPlatform.efi
Error: Image at 000E6F44000 start failed: 00000001
Remove-symbol-file /bsp/mu_platform_nxp/Build/MEK_IMX8QXP_3GB/DEBUG_GCC5/AARCH64/MdeModulePkg/Universal/ACPI/ACPIPlatformDxe/ACPIPlatformDxe/DEBUG/ACPIPlatform.dll 0xE6F45000
mbiosPlatformDxe.efi
```

Figure 14. RPMB key missing

6.5.2 Resolution

Rebuild OP-TEE with debug prints enabled:

```
make PLATFORM=imx PLATFORM_FLAVOR=$optee_plat \
    CFG_TEE_CORE_DEBUG=y CFG_TEE_CORE_LOG_LEVEL=4 \
    CFG_RPMB_FS=y CFG_REE_FS=n \
    CFG_CORE_HEAP_SIZE=131072
```

Boot the device with new OP-TEE, review boot messages. The following messages are signaling that there is a missing RPMB key:

```
D/TC:? 0 tee_rpmb_init:1122 RPMB: Syncing device information
D/TC:? 0 tee_rpmb_init:1130 RPMB: RPMB size is 32*128 KB
D/TC:? 0 tee_rpmb_init:1132 RPMB: Reliable Write Sector Count is 1
D/TC:? 0 tee_rpmb_init:1158 RPMB INIT: Deriving key
D/TC:? 0 tee_rpmb_key_gen:310 RPMB: Using test key
D/TC:? 0 tee_rpmb_init:1173 RPMB INIT: Verifying Key
E/TC:? 0 tee_rpmb_verify_key_sync_counter:1021 Verify key returning 0xffff0008
D/TC:? 0 tee_rpmb_init:1181 RPMB INIT: Auth key not yet written
D/TC:? 0 tee_rpmb_write_and_verify_key:1096 RPMB INIT: CFG_RPMB_WRITE_KEY is not set
54B: copy section headers:990 sys copy from to bin
```

Figure 15. RMPB no key log

Follow the RPMB [secure provisioning](#) chapter.

7 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

8 Revision history

The table below summarizes the revisions to this document.

Revision number	Release date	Description
IMXWGU_Rev.1.5.1	24 May 2024	Updated for version 1.5.1
IMXWGU_Rev.1.5.0	12 August 2023	Updated for version 1.5.0
IMXWGU_Rev.1.4.1	31 July 2023	Minor technical changes.
IMXWGU_Rev.1.4.0	20 March 2023	Updated for version 1.4.0
IMXWGU_Rev.1.3.0	20 December 2022	Updated for version 1.3.0
IMXWGU_Rev.1.2.1	26 October 2022	Updated for version 1.2.1
IMXWGU_Rev.1.2.0	23 September 2022	Section 1.7 is removed.
IMXWGU_Rev.1.1.0	27 June 2022	Public release for the i.MX 8M Nano and i.MX 8M Plus platforms.
IMXWGU_Rev.1.0.0	14 April 2022	Public release for the i.MX 8M and i.MX 8M Mini platforms.
IMXWGU_Rev.0.9.1	14 March 2022	Public preview release for the i.MX 8M platform.
IMXWGU_Rev.0.9.0	10 January 2022	Private preview release for the i.MX 8M platform.

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at <https://www.nxp.com/profile/terms>, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μ Vision, Versatile — are trademarks and/or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

Microsoft, Azure, and ThreadX — are trademarks of the Microsoft group of companies.

Contents

1	Overview	2		
1.1	Audience	2	5.2.2.2	Device Tree Blob
1.2	Conventions	2	5.2.2.3	OP-TEE
1.3	How to start	2	5.2.2.4	ATF
1.4	Using Prebuilt Binaries to create an image	2	5.2.2.5	U-Boot proper
1.5	Using Source Files to create an image	2	5.2.2.6	UEFI
1.6	References	2	5.2.3	Ensuring firmware security
2	Building Windows IoT for NXP i.MX Processors	4	5.2.3.1	Security configuration
2.1	Building the drivers in the BSP	4	5.2.3.2	Bootloader verification chain
2.1.1	Required tools	4	5.2.3.3	HAB chain of trust
2.1.1.1	Visual Studio 2019	4	5.2.3.4	i.MX firmware image verification
2.1.1.2	Windows Kits from Windows 10, version 2004 (10.0.19041.685)	4	5.3	Secure boot on i.MX 8QXP
2.1.2	Obtaining sources for building the drivers	4	5.3.1	System boot on i.MX 8QXP
2.1.2.1	Preparing source for building the drivers	4	5.3.2	i.MX boot containers
2.1.3	Structure of Windows driver sources	5	5.3.3	System boot components
2.1.4	One-time environment setup	5	5.3.3.1	U-Boot SPL
2.1.5	Building the drivers	5	5.3.3.2	Device Tree Blob
2.2	Building ARM64 Firmware	6	5.3.3.3	OP-TEE
2.2.1	Required tools	6	5.3.3.4	ATF
2.2.2	Obtaining sources for building ARM64 Firmware	6	5.3.3.5	U-Boot proper
2.2.2.1	Preparing sources for building firmware	6	5.3.3.6	UEFI
2.2.3	Setting up your build environment	7	5.3.4	Ensuring firmware security
2.2.4	Building the firmware	8	5.3.4.1	Security configuration
2.2.5	Common causes of build errors	10	5.3.4.2	Bootloader verification chain
3	Display/GPU driver	11	5.3.4.3	AHAB chain of trust
3.1	Display interface selection	11	5.3.4.4	i.MX firmware image verification
3.2	Display resolution and timing parameters	11	5.4	Secure boot on i.MX 93
3.2.1	HDMI display interface	11	5.4.1	System boot on i.MX 93
3.2.2	LVDS, MIPI-DSI and Parallel display interfaces	12	5.4.2	i.MX Boot Containers
3.3	Display specific parameters	12	5.4.3	System boot components
3.3.1	LVDS display interface	12	5.4.3.1	U-Boot SPL
3.3.2	MIPI-DSI display interface	12	5.4.3.2	Device Tree Blob
3.4	Display support in firmware	12	5.4.3.3	OP-TEE
3.4.1	Firmware display interface selection	12	5.4.3.4	ATF
3.4.2	Firmware display resolution	13	5.4.3.5	U-Boot proper
4	Power management	14	5.4.3.6	UEFI
4.1	Power management user scenarios	14	5.4.4	Ensuring firmware security
4.2	Device power management DPM on i.MX 8/9 platforms	14	5.4.4.1	Security configuration
4.3	Processor power management PPM on i.MX 8/9 platforms	15	5.4.4.2	Bootloader verification chain
4.4	Power management tools and debugging	16	5.4.4.3	AHAB chain of trust
4.4.1	powercfg /a	16	5.4.4.4	i.MX firmware image verification
4.4.2	powercfg /sleepstudy	16	5.5	Secure storage
4.4.3	powercfg /energy	17	5.5.1	RPMB
4.4.4	WinDbg !fxdevice	17	5.5.2	Secure vs. non-secure build
5	Secure boot	20	5.5.2.1	Secure build
5.1	Basic concepts	20	5.5.2.2	Non-Secure build
5.2	Secure boot on i.MX 8M	20	5.6	Secure Boot in UEFI and Windows
5.2.1	System boot on i.MX 8M	20	6	Secure provisioning
5.2.2	System boot components	20	6.1	Secure provisioning i.MX 8M
5.2.2.1	U-Boot SPL	21	6.1.1	Generate HAB keys
			6.1.1.1	Prepare SRK table
			6.1.2	Building secured binary
			6.1.3	Locking the device for i.MX 8M
			6.1.4	Burning SRK_HASH
			6.1.5	Burning SEC_CONFIG
			6.2	Secure provisioning i.MX 8QXP
			6.2.1	Generate AHAB keys

6.2.1.1 Prepare SRK table 38

6.2.2 Building secured binary 38

6.2.3 Locking the device (i.MX 8QXP)39

6.2.3.1 Burning SRK_HASH 39

6.2.3.2 Closing the chip 40

6.3 Secure provisioning i.MX 93 40

6.3.1 Generate AHAB keys 40

6.3.1.1 Prepare SRK table 41

6.3.2 Building secured binary 42

6.3.3 Locking the device 42

6.3.3.1 Preparing SRKH 42

6.3.3.2 Burning SRK_HASH 42

6.3.3.3 Closing the chip 43

6.4 RPMB, UEFI 44

6.4.1 RPMB 44

6.4.2 UEFI 44

6.5 Troubleshooting 44

6.5.1 Firmware built as secure fails to boot or
hangs in UEFI 44

6.5.2 Resolution 44

7 **Note about the source code in the
document 46**

8 **Revision history 47**

Legal information 48

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.