MCU-SMHMI-SDUG

Smart HMI Software Development User Guide

Rev. 0 — 25 October 2022 **User guide**

Document information

Information	Content
Keywords	SLN-TLHMI-IOT, Human Machine Interface (HMI), IoT
Abstract	The purpose of this guide is to help developers better understand the software design and architecture of the applications in order to more easily and efficiently implement applications using the SLN-TLHMI-IOT

1 Introduction

Welcome to the Developer Guide for the SLN-TLHMI-IOT!

The purpose of this guide is to help developers better understand the software design and architecture of the applications in order to more easily and efficiently implement applications using the SLN-TLHMI-IOT.

This guide covers such topics as the bootloader and the framework + HAL architecture design, as well as other features that may be relevant to application development using SLN-TLHMI-IOT.

Check out the Smart HMI Getting Started Guide for an overview of the out of box features available in the SLN-TLHMI-IOT applications.

2 Setup and installation

This section is focused on the setup and installation of the tools necessary to begin developing applications using NXP's framework architecture.

This guide focuses on MCUXpresso IDE for development.

2.1 MCUXpresso IDE

MCUXpresso IDE brings developers an easy-to-use Eclipse-based development environment for NXP MCUs based on Arm Cortex-M cores, including its general-purpose crossover and Bluetooth-enabled MCUs. MCUXpresso IDE offers advanced editing, compiling, and debugging features with the addition of MCU-specific debugging views, code trace and profiling, multicore debugging, and integrated configuration tools. MCUXpresso IDE debug connections support Freedom, Tower system, LPCXpresso, i.MX RT-based EVKs, and your custom development boards with industry-leading open-source and commercial debug probes from NXP, P&E Micro, and SEGGER.

For more information, see the NXP website

2.2 Install the toolchain

MCUXpresso IDE can be downloaded from the NXP website by using the below link:

Get MCUXpresso IDE

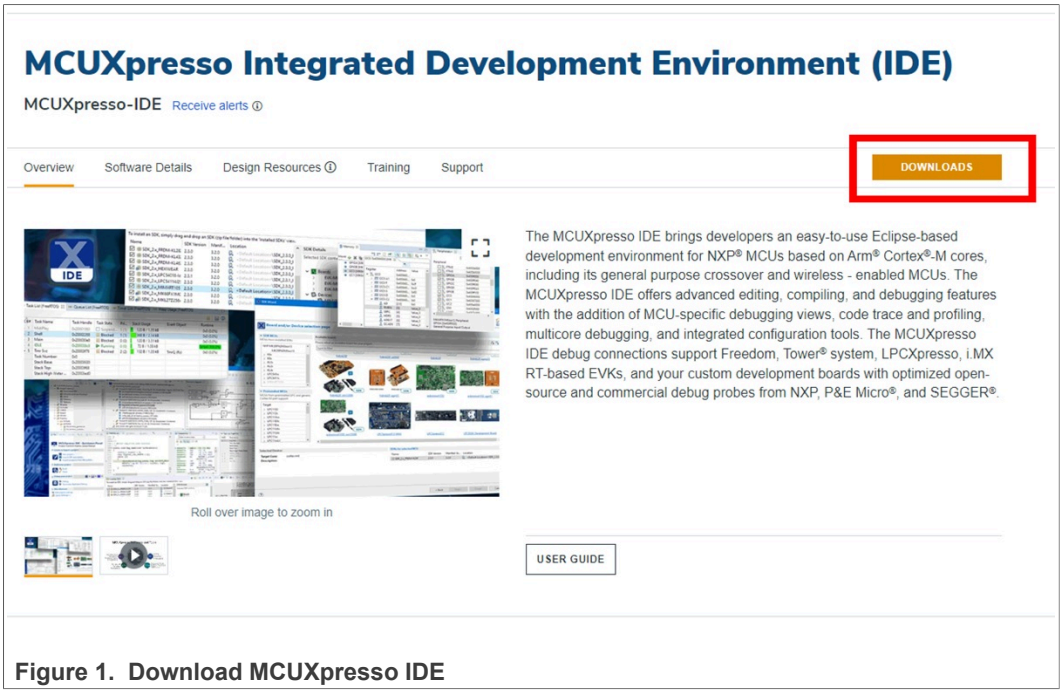


Figure 1. Download MCUXpresso IDE

To download the correct version of IDE, check out the Smart HMI Getting Started Guide. Once the download has been completed, follow the instructions in the installer to get started.

Note: *There is a bug in version 11.5.1 of MCUXpresso IDE that prevents building projects for SLN-TLHMI-IOT, so version 11.5.0, 11.6.0, or greater is required.*

Figure 2. Check MCUXpresso IDE version with v11.5.0

2.3 Install the SDK

To build projects using MCUXpresso IDE, install an SDK for the platform you intend to use. A compatible SDK has the required dependencies and platform-specific drivers needed to compile projects.

A compatible SDK can be downloaded from the official NXP SDK builder

1. To build the SDK for your preferred setup, use MCUXpresso IDE to install the SDK.
2. To do this, open the application and click**Download and Install SDKs** on the MCUXpresso IDE welcome screen as shown below:

Figure 3. Download and Install SDKs

3. A catalog of all the SDKs that can be downloaded through MCUXpresso is available. These SDKs provide device knowledge, drivers, middleware, and reference example applications for your development board or MCU. Type ***evkmimxrt1170*** in the filter section and download evkmimxrt1170 SDK. The current applications were developed and tested on SDK 2.11.1.

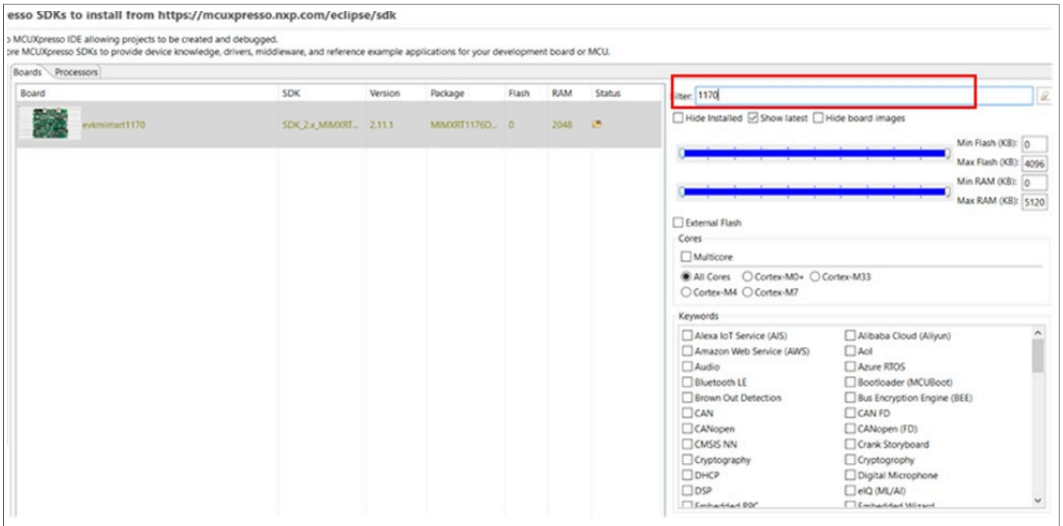


Figure 4. Download RT1170 SDK

4. A prompt displays the license agreement associated with the 1170 SDK.
5. Read and accept the license to automatically start the SDK installation.
6. MCUXpresso proceeds to download the SDK.

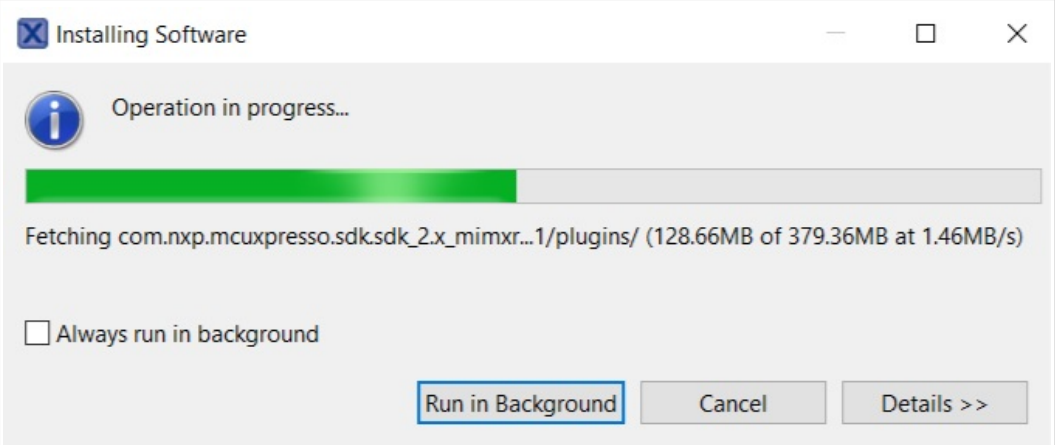


Figure 5. Install RT1170 SDK

2.4 Import example projects

Note: *To build example projects that you import regardless of how they are imported, you **must** have a compatible MCUXpresso SDK package for SLN-TLHMI-IOT installed.*

MCUXpresso IDE allows you open example projects from the source folder.

2.4.1 Import from Github

Note: *Before you begin, make sure you have Git downloaded and installed on the machine you intend to use.*

The latest software updates for the SLN-TLHMI-IOT application can be downloaded from our official Github repository. Here, you will find the most up-to-date version of the code that contains the newest features available for the Smart TLHMI project.

To import the SLN-TLHMI-IOT Smart TLHMI application into MCUXpresso IDE using Github, perform the following steps:

1. Clone the `sln_tlhmi_iot` repository.
   - Cloning directly to your MCUXpresso workspace location is recommended, but not required.
2. In MCUXpresso, navigate to the **File** from Toolbar.
3. Click **Open Projects from File System...**.
4. Select **Directory...**.
5. Navigate to the file path of the project cloned in the first step and click **Select Folder**.
6. Check the box next to each project (`bootloader`, `coffee_machine\cm4`, `coffee_machine\cm7`, `coffee_machine\lvgl_vglite_lib` and `elevator\cm4`, `elevator\cm7`, `elevator\lvgl_vglite_lib`) you wish to import.
7. Click **Finish**

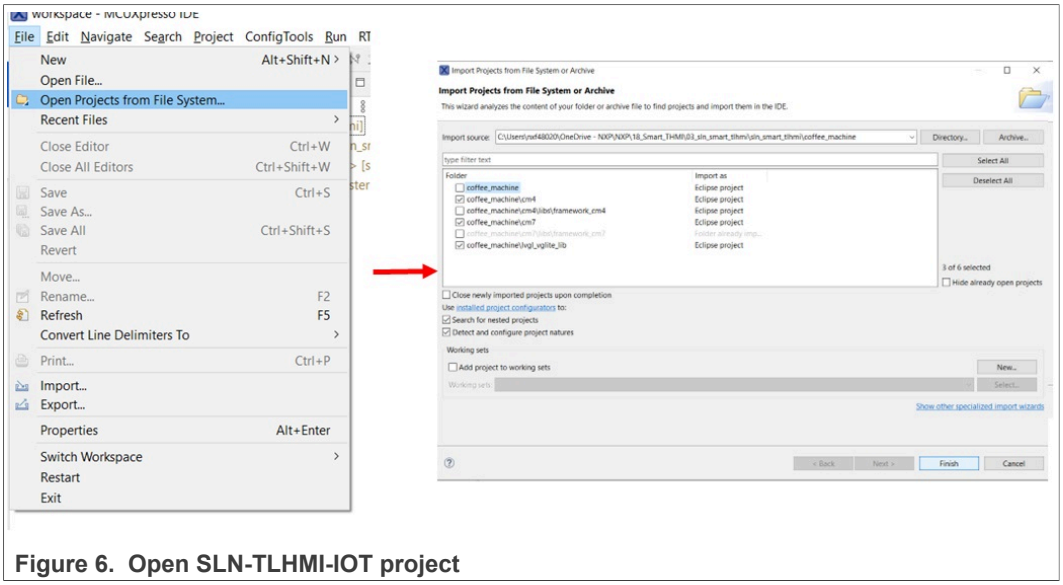


Figure 6. Open SLN-TLHMI-IOT project

After following the above steps, confirm that the projects can be found in the **Project Explorer** panel to ensure they were successfully imported.

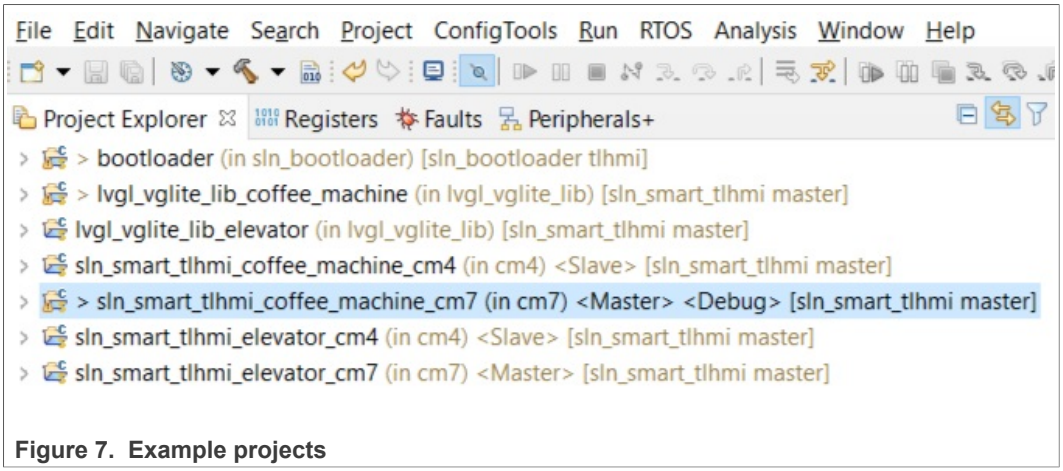


Figure 7. Example projects

3 Ivaldi

3.1 Automated manufacturing tools

This section provides an overview of Ivaldi, prerequisites, platform configuration, and open boot programming.

3.1.1 About Ivaldi

Ivaldi is a package that is responsible for manufacturing and reprogramming without J-Link. It uses the serial downloader mode within the RT117H boot ROM to communicate with an application called Flashloader that is programmed into RT117H. It then communicates with a program called blhost that controls various parts of the chip and flash. Ivaldi was created to focus on the build infrastructure of a customer's development and manufacturing cycle. Its primary focuses are:

- Factory programming and setting up a new device/product
- Generating AWS IoT Devices
- Creating certificate/key pairs for devices
- Associating policies with devices
- Signing images for OTA and HAB
- Writing and Accessing OTP fuses

The following section gives information about the general flashing of a device without debugging tools.

Note: *ITo use Ivaldi, put the board in Serial Download Mode. For doing that, move jumper J203 on the top of the board into position "0". For more information, see Smart HMI Hardware Development User Guide*

3.1.2 Requirements

- Section 5.1.1 must be followed
- OpenSSL
- AWS CLI installed
  - https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html
  - https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html#cli-quick-configuration
- Python 3.6.x
- Linux / Windows CMD / Ubuntu for Windows
- README.md from ivaldi root folder must be followed

3.1.3 Platform configuration

Ivaldi uses a platform configuration file `Scripts/sln_platforms_config/sln_tlhmi_iot_config/board_config.py`. This file describes:

- The names of the binaries (from the `Image_Binaries` folder) which will be flashed:
  - BOOTLOADER_NAME
  - DEMO1_NAME
  - DEMO1_NAME_RESOURCES
  - DEMO2_NAME
  - DEMO2_NAME_RESOURCES
- Flash configurations:
  - FLASH_TYPE
  - FLASH_START_ADDR
  - FLASH_SIZE
- Flash Map
  - Binaries' images addresses
  - Filesystem starting address and size
  - FICA table addresses

To configure Ivaldi to use specific image binaries from Image_Binaries folder, update `Scripts/sln_platforms_config/sln_tlhmi_iot_config/board_config.py` file.

Note: *Any changes in `scripts/sln_platforms_config/sln_tlhmi_iot_config/board_config.py` (except binaries' names) require updating the embedded code and configurations.*

3.1.4 Open Boot Programming

The Open Boot Programming tool is responsible for creating a device and programming it with the correct images, certificates, and artifacts. This method is a quick and easy way of taking a device/product from the assembly line and getting it ready to ship. It is also good practice to run the Open Boot Programming script before enabling the security features to ensure that all images and artifacts are in the working order. The Open Boot Programming script must only be run when all the images and artifacts are obtained. Before running the script, ensure that the following files and folders exist in the “Image_Binaries” directory of Ivaldi root and that all the files mentioned in the `board_config.py` exist. After the script was executed, do not forget to exit the serial downloader mode by moving back the J203 jumper.

A directory "Scripts/sln_tlhmi_iot_open_boot" within the Ivaldi package contains the “open_prog_full.py” script and a README.

The README file contains build requirements for each image before running the script. If the requirements are not fulfilled, it could cause the boot failure.

To program the firmware and artifacts, execute the`open_prog_full.py` script that performs the following actions:

- Communicate with the BootROM to program Flashloader
- Create a device with
  - Certificate
  - Private Key
  - Policy Attached in the cloud
- Erase the flash
- Generate littlefs format filesystem, that contains files specified in the `littlefs_file_list.py`
- Programming the images
  - Bootloader
  - demo1
  - demo1_resources
  - demo2
  - demo2_resources
  - Program the FICA
  - Program the littlefs

In the current `open_prog_full.py` python script, the littlefs is being generated to contain all the files mentioned in `littlefs_file_list.py`. Four files are expected:

- Root CA certificate
- AppA sign certificate - validated by the CA certificate and used to sign all the images that are being written or send for update
- AWS certificate - used to validate connection with AWS server
- AWK public key - used to communicate with AWS MQTT server

One drawback of the current littlefs implementation is that it does not support the attributes. It is used in the `SLN_TLHMI_IOT` project to generate encrypted files.

Note: *Open programming script assumes that the policy is called `tlhmi_deployment`. Update the script to use the correct policy name in the customers aws account..*

4 Bootloader

4.1 Introduction

The Smart Lock project uses a "bootloader + main application" architecture to provide additional security and update-related functionality to the main application. The bootloader handles all boot-related tasks including, but not limited to:

- Launching the main application and, if necessary, initializing the peripherals
- Firmware updates using either the Mass Storage Device (MSD), Over-the-Air, or Over-the-Wire update method
  - Protects against update failures by using a primary and backup application "flash bank"
- Image certification/verification

4.1.1 Why use a bootloader?

By separating the boot process from the main application, the main application can be safely updated and verified without the risk of creating an irrecoverable state due to a failed update, or running a malicious, unauthorized, and unsigned firmware binary flashed by a bad actor. It is essential in any production application to take precautions to ensure the integrity and stability of the firmware before, during, and after an update, and the bootloader application is simply one measure to help provide this assurance.

The following sections describe how to use many of the bootloader's primary features to assist developer interested in understanding, utilizing, and expanding them.

4.1.2 Application Banks

The bootloader file system uses dual application "banks" referred to as "Bank A" and "Bank B" to provide a backup/redundancy "known good" application to prevent bricking when flashing an update via either the MSD, OTA, or OTW update method. For example, if an application update is being flashed via MSD to the Bank A application bank, even if that update fails midway, Bank B still contains a fully operational backup.

In the `SLN-TLHMI-IOT`, Bank A is at `0x30100000` while Bank B is at `0x31500000`.

Providing an application binary built for the proper application bank address is crucial during MSD, OTA, and OTW updates, and the failure will result in a failure to flash the binary.

Note: *The bootloader does not automatically recover from a botched flashing procedure but reverts to the alternate working application flash bank instead.*

4.1.3 Logging

The bootloader supports debug logging over UART to help diagnose and debug issues that may arise while using or modifying the bootloader. For example, the debug logger can be helpful when trying to understand why an application update might have failed.

Logging is enabled by default in the **Debug** build mode configuration. The logging functionality, however, comes with an increase in bootloader performance and can slow down the boot process by as much as 200 ms. As a result, it may be desirable to disable debug logging in production applications. To disable logging to the bootloader, simply build and run the bootloader in the `Release` build mode configuration. It can be done by

right-clicking on the bootloader project in the **Project Explorer** view and navigating to **Build Configurations -> Set Active -> Release** as shown in the figure below:

To make use of the debug logging feature, use a UART->USB converter to:

- Connect `GND` pin of converter to `J202: Pin 8`
- Connect `TX` pin of converter to `J202: Pin 3`
- Connect `RX` pin of converter to `J202: Pin 4`

Once the converter has been properly attached, connect to the board using a serial terminal emulator, for example, *PuTTY* or *Tera Term* configured with the following serial settings:

- Speed: `115200`
- Data: `8 Bit`
- Parity: `None`
- Stop Bits: `1 bit`
- Flow Control: `None`

4.2 Overview

The bootloader employs several different boot-up methods to augment the boot-up behavior. Currently, the bootloader supports two primary boot modes:

- **Normal Mode**
- **Mass Storage Device (MSD) Update Mode**

Normal mode, as the name would imply, is the default boot mode in which the bootloader simply loads the main application.

Mass Storage Device Update mode is a special boot mode in which the board enters an update state where the board appears as a mass storage device to a host PC device. In this mode, the bootloader is capable of receiving and flashing a new binary by copying that binary to the board as one would for a regular USB storage device.

More information on each of these modes can be found in the subsequent sections of this document.

4.2.1 How is boot mode determined?

To determine the boot mode, the bootloader checks several different boot flags, which are set based on various conditions.

For each different boot mode (excluding Normal boot, which is taken by default), there is a different corresponding boot flag. Boot flag gets set depending on the boot mode in question and the platform being used. On the SLN-TLHMI-IOT, for example, the MSD boot flag is set when the `SW0` button is held during bootup.

4.3 Normal boot

By default, if no other boot flags are set during the boot phase, the Normal boot mode is used. During Normal boot, the Bootloader boots to the "main" application, which is flashed at the current application bank flash address (for more information, see Application Banks). For example, if the current flash bank is set to **Bank A**, then the Bootloader jumps to the flash address associated with **Bank A** and begins running the application at that address.

The OOBE has a set of three applications that can be booted into at startup. By default, the application always boots in the **Bank A**, which corresponds to the `coffee_machine` application. To change the boot application, use buttons labeled `SW1-SW3` when powering the board.

The following list shows the associations of boot application to switch.

- `SW1` - Bank A - coffee_machine
- `SW2` - Bank B - elevator
- `SW3` - Bank C - TBD

The decision to what application to jump is handled inside the bootloader. To reach the bootloader, a soft or hard reset is needed.

For example, to boot in elevator application:

1. Unplug the board
2. Press and hold the **SW2** button
3. Plug the board in.

From the Bootloader's perspective, there is no information what application it is jumping into, because it uses addresses and not names. After an update procedure, the application that was written in an inactive bank is overwritten, so the links between banks and demos are not valid anymore.

4.3.1 Turn on Image Verification

In the OOBE Bootloader demo, Image Verification is disabled to encourage developers to play with the code. If Image Verification is enabled, Normal boot checks that the image certificate for the firmware image to run has been signed by a trusted certificate authority to ensure that the application comes from a trusted source. Should the signature check fail, the Bootloader does not run the application to avoid executing untrusted, potentially malicious firmware.

For more details regarding image verification, see Image Verification.

To enable the image verification, **DISABLE_IMAGE_VERIFICATION** must be set to 0 inside the **Preprocessors** sections:

1. Within the MCUXpresso Bootloader project, right-click the root project and navigate to **Properties > C/C++ Build > Settings > Preprocessor** .
2. Inside the **Preprocessors** section, change the MACRO **DISABLE_IMAGE_VERIFICATION** to “0” and click the **Apply and Close** button as described in the figure below.

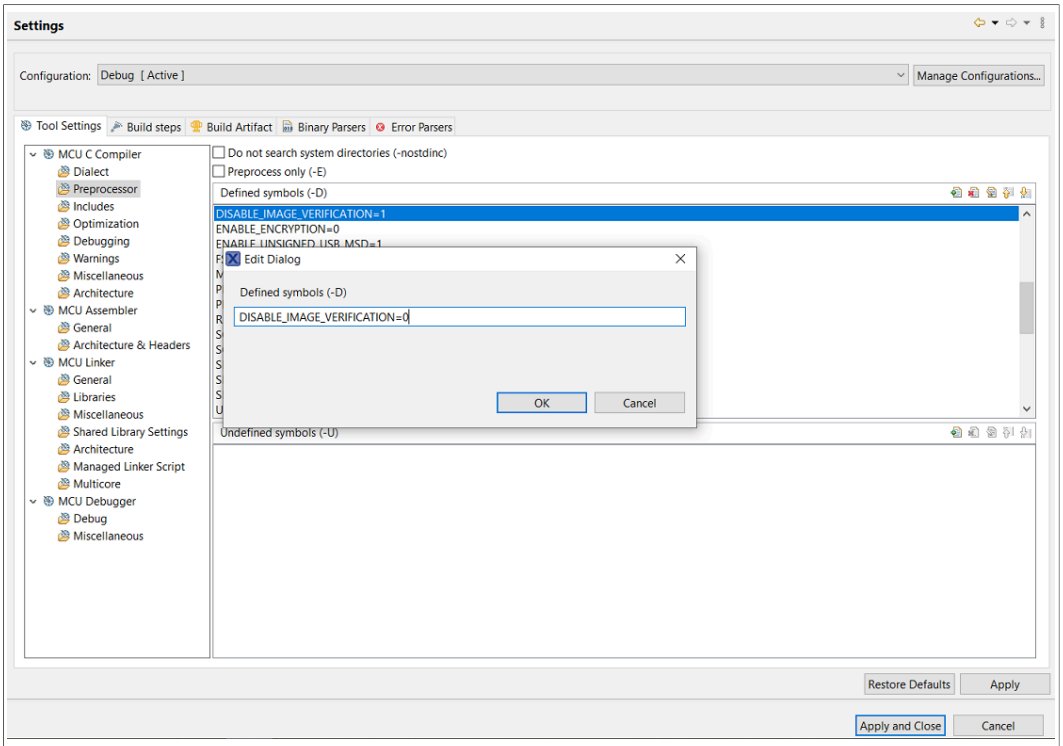


Figure 8. Enable image verification

3. After that change, rebuild the Bootloader.
4. To flash the device with proper FICA and certificates, use Automated manufacturing tools (Ivaldi).

4.3.2 Disable Debug Console

In the OOBE Bootloader demo, Debug Console is enabled to help developers test and debug their code. This feature introduces unwanted message being displayed and increases the boot-up time. To disable this, set **ENABLE_LOGGING** to 0 in `FreeRTOSConfig.h`

Note: *The current implementation of the debug console adds about 150 ms to the boot time.*

4.4 Mass Storage Device updates (MSD)

The MSD feature allows the device to be updated using USB instead of the Segger tool. Only the main application or its resources (coffee_machine/elevator) can be flashed in this manner. If the bootloader must be updated, the Segger tool or the Factory Programming flow is necessary. The MSD feature, by default, bypasses the signature verification to simplify the development flow, since signing images can be unsuitable for quick debugging and validation.

4.4.1 Enabling MSD mode

To enable MSD mode on the SLN-TLHMI-IOT, press and hold the **SW0** button while powering on the board. If done correctly, the board's onboard LED changes to purple and begins blinking at an interval of roughly 1 second.

Note: *As mentioned in the Smart HMI Getting Started Guide, to properly use `SW0` as a general-purpose switch the `SW8` dip switch must be set as 0001.*

Additionally, if connected to a Windows PC, your computer must make a sound indicating a new USB device has been connected. After observing the LED blinking behavior, navigate to “My Computer”, and confirm that the SLN-TLHMI-IOT kit has mounted as a Mass Storage Device as shown in the picture below.

Figure 9. Bootloader MSD file explorer

The size of the new storage device is equal to the Bank Size of the device from which you subtract the file system metadata.

4.4.2 Flashing a new binary

The binary size increases exponentially when adding the GUI resources. Almost 70 % of the total size is occupied by these sounds and images. To speed up the development and to decrease the load on the updating mechanism, the large images have been split into **code and resources**, both with fixed addresses in the flash. Update operations can be done on individual components, or all together into a bundle.

Right now the MSD can be used to update:

- Main Application
- Resources
- Bundle update (Main Application + Resources)

4.4.2.1 Main application

To update the main application, a binary must be built for the address `0x30100000`. Because of the remap functionality enabled in the bootloader, this binary can be placed in each of the three banks, and still work as it is running from the base address. The bootloader checks for the current unused bank and tries to write the image in that specific bank. When dragging and dropping a binary for the main application, the bootloader checks if the reset handler of the new image is a flash address. No other verification is done; the functionality's correctness must be handled by the developer. After the new image has been written, a resource copy is done. This means that during the update procedure, the resources will stay the same.

4.4.2.2 Resources

When updating the resources, the binary needs to be renamed into `RES.bin`. The Bootloader contains a list of known files, `res.bin` is one of those files. No verification is done on the resources binary.

In the same way as updating the main application, the bootloader checks for active bank and writes the binary in the unused one. After the write is completed, the older firmware is copied, and the new bank is activated.

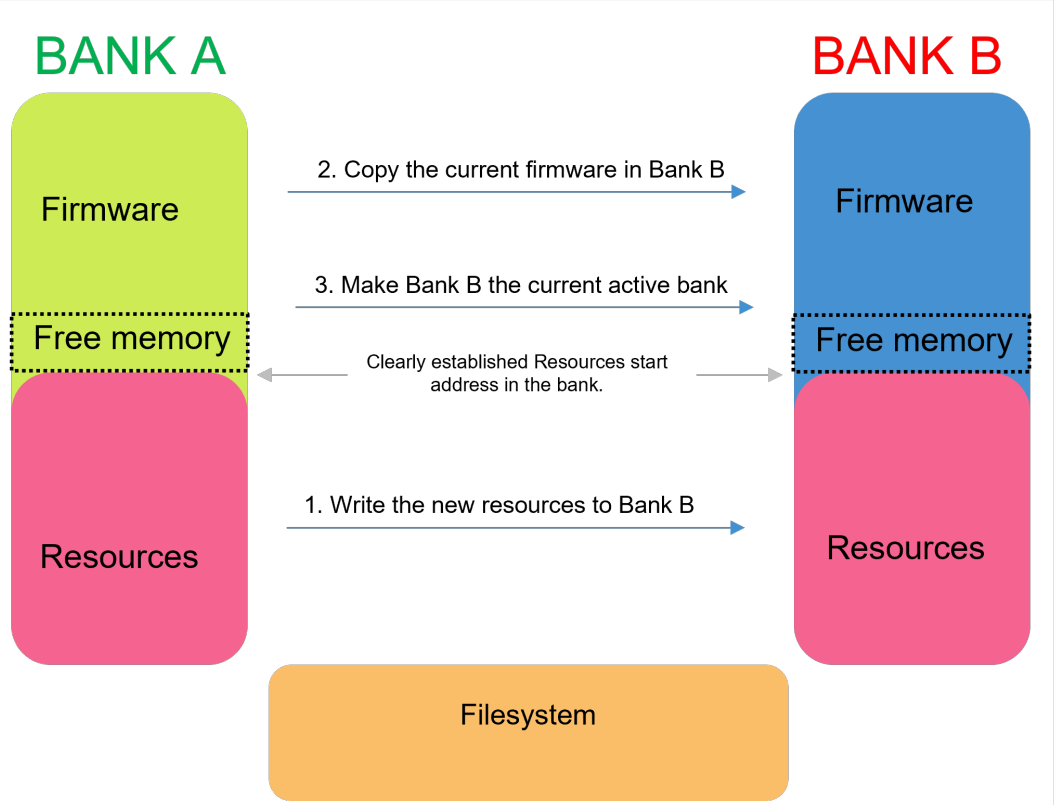


Figure 10. Update resources

4.4.2.3 Bundle

To update using the bundle method, a python script is used to generate the bundle. The script is part of the ivaldi suite of scripts that are delivered to the customer. The script is called `bundle_generate_tlhmi.py`. When calling it, two parameters must be set, both being the locations for two important files:

- bundle configuration file (-bf) - contains a list of files that are going to be fused to generate the bundle.
- board configuration file (-cf) - position of the files in flash to build the metadata.

In the released version of ivaldi, both bundle config and board config are placed under the platform config folder. A full linux bash command to call this script looks like:

```
python bundle_generate_tlhmi.py -bf ../../../Scripts/
sln_platforms_config/sln_tlhmi_iot_config/ -cf ../../../
Scripts/sln_platforms_config/sln_tlhmi_iot_config/
```

After this, in the `Scripts\ota_signing\sign\output` folder, four files are present.

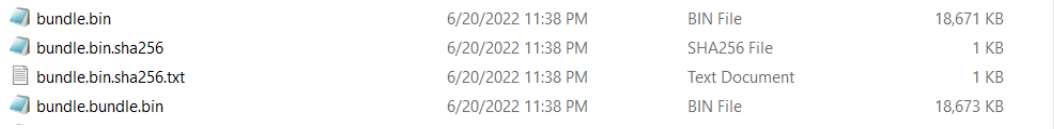


Figure 11. Update bundle_generate script

For MSD only `bundle.bin` is of interest, the other three are relevant for Over-The-Air (OTA) updates, where validation is an important feature. To update with the

`bundle.bin`, drag and drop the binary. The name must not be modified, as this name is part of a hardcoded list of known files.

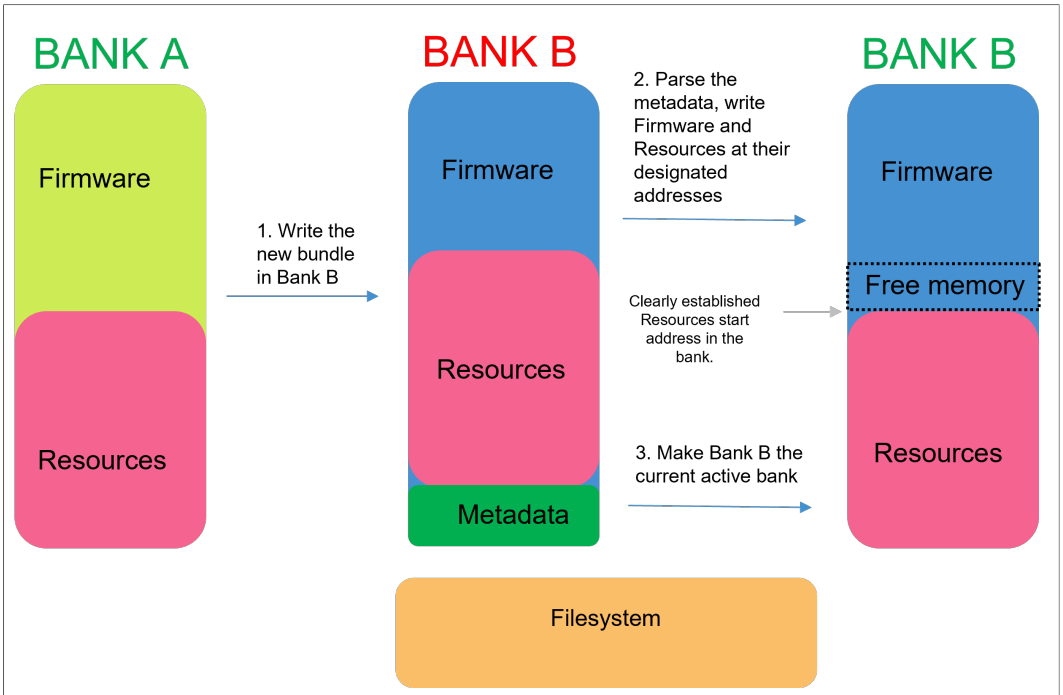


Figure 12. Update bundle

For the bootloader to parse and write all the modules to their designated addresses, metadata must be added to the package. Two types of metadata exist:

- Bundle metadata is placed at the end of the bundle and contains:
  - Bundle size
  - Number of modules
  - Signature of the whole bundle
- Module metadata is placed after every module and contains:
  - Module type (Application or Resources)
  - Module starting address
  - Module length
  - Module signature

Upon completion, the board automatically reboots itself into the new firmware, which was flashed. . To verify this, open the serial CLI, type typing the `version` command, and check that the application is running from the alternate flash bank.

4.5 Image Verification

Image Verification is a mechanism in which we validate that the image running has not been altered either by internal or external factors.

4.5.1 Application chain of trust

The basis of the security architecture implemented in the SLN-TLHMI-IOT has signed application images. Signing requires the use of a Certificate Authority (CA). NXP has its

own CA for signing applications at the factory, but the CA is not something that is shared with customers.

The CA is used to create signing entities for applications as shown in the figure below. A certificate from the CA is stored in the SLN-TLHMI-IOT's filesystem and is used to verify the signatures of the signing entity certificates. In addition, locally stored certificates from the signing entities are used to verify the signature of firmware images coming in Over-the-Air (OTA) updates.

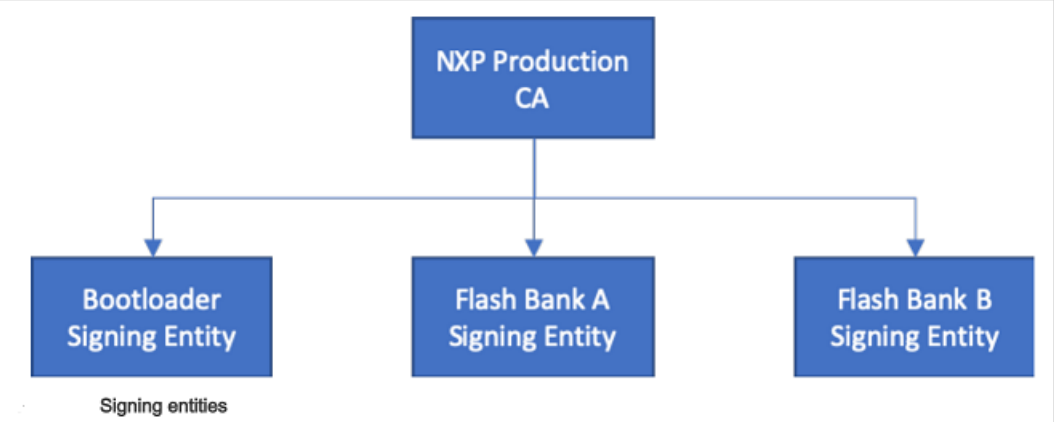


Figure 13. Chain of trust

4.5.2 Flash Image Configuration Area (FICA) and Image Verification

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

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

Note: *As mentioned when describing the application banks, `Bank C` is not used for redundancy in the update mechanism, as such, it has no entry into the FICA table. The purpose of the bank is only to showcase all 3 applications without the need of reflashing the board.*

Developers can turn on the image verification and reprogram the bootloader as shown in the Turning on image verification section. To decrease the risks of an attack, have Image Verification on.

4.6 Application banks

For this project, we enabled three application flash banks, **Bank A**, **Bank B**, and **Bank C**. It is done to showcase in our OOBE all projects (`coffee_machine, elevator`) simultaneously.

In a real-life scenario, only 2 banks are needed. In the updating mechanism that has been implemented, we use 2 banks by doing a ping-pong between **Bank A** and **Bank B**.

The SLN-TLHMI-IOT utilizes a series of dual "application flash banks" used as a redundancy mechanism when updating the firmware via one of the bootloader's update mechanisms (see Section 4.4) or via the AWS OTA mechanism.

4.6.1 Banks

The application we developed for SLN-TLHMI has 2 inter-dependent parts:

- Application (code)
- Resources (icons, sounds, pictures)

So a bank is a reserved space in the flash that stores both of these components. The application running tries to read resources from the same bank.

In the OOBE, the size of a bank is 20 MB (`0x1400000`), 6 MB (`0x600000`) for the code area and 14 MB (`0xE00000`) for resources. If there is a need to increase or decrease this value, check `fica_definitions.h`

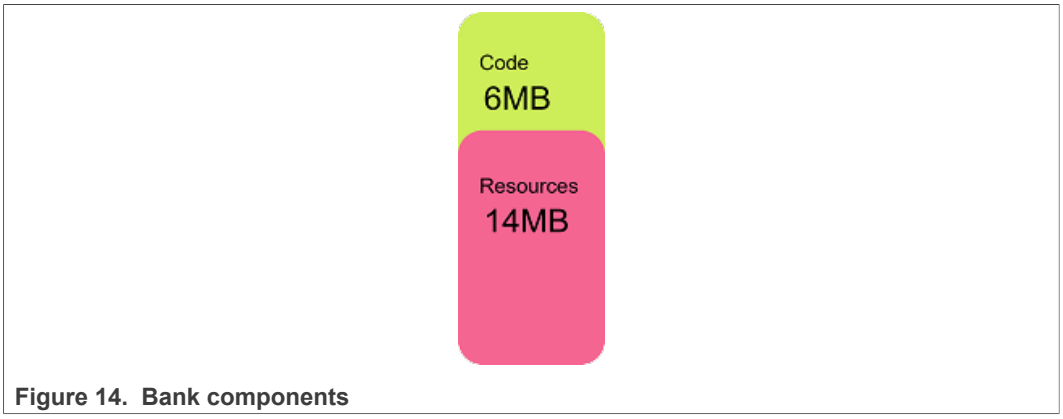


Figure 14. Bank components

4.6.2 Addresses

The flash address for each of the application flash banks is as follows:

- Bank A - `0x30100000`
  - Bank A App - `0x30100000`
  - Bank A resources - `0x30700000`
- Bank B - `0x31500000`
  - Bank B App - `0x31500000`
  - Bank B resources - `0x31B00000`
- Bank C - `0x32900000`
  - Bank C App - `0x32900000`
  - Bank C resources - `0x32F00000`

4.6.3 Remapping

The i.MXRT117H chip supports the flash remapping function, which allows users to remap flash address to the FlexSPI interface. The flash remapping function is beneficial in the following use-cases:

- To flash multiple firmware.
- To switch one of the firmware to run when the condition is met.

- To update the firmware in the wireless application (the usual process is to download the firmware to flash, perform the validity check, and then switch to new firmware to run. The flash remapping function helps to directly run the firmware wherever it locates to XIP flash.)

For more information, check: How to Use Flash Remapping Function

In older Solution's projects like SLN-VIZN3D-IOT and SLN-VIZNAS-IOT, the images were built for a specific bank. With the enablement of the remapping functionality, all applications must be built having the Flash Starting Address set to `0x30100000`.

The updating mechanisms implemented in the bootloader or the main application leverage this feature. Because of this, the updating procedure does not have to keep track of what bank the application is running from. The binary that is going to be used for an update, is always going to be built with the **Bank A** memory settings and is going to be placed in the non-active slot.

Note: *The OOBE is meant to showcase all 3 applications. After an update procedure, the application that was written in a non-active bank is going to be overwritten.*

4.6.3.1 Convert .axf to .bin

When building a project in MCUXpresso IDE, the default behavior is to create an `.axf` file. However, some of the bootloader update mechanisms including MSD updates require the use of a `.bin` file.

Converting an `.axf` file to `.bin` can be done in MCUXpresso without any additional setup.

To perform this conversion, navigate to the project directory that contains your compiled project binary and right-click the `.axf` file in that directory.

Note: *The binary for your project is located in either the **Debug** or **Release** folder depending on your current build config.*

In the context menu, select **Binary Utilities->Create binary**.

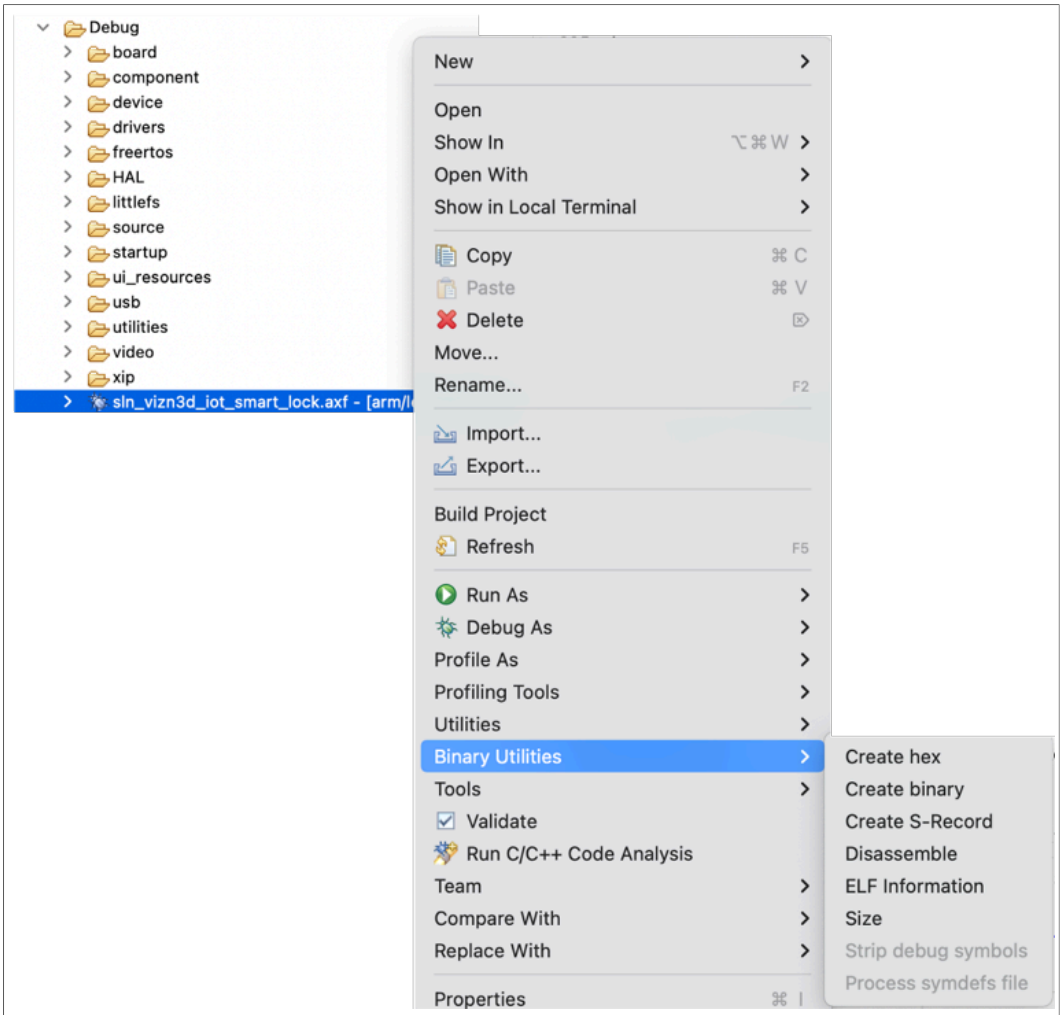


Figure 15. Convert to binary

Verify that the binary has been successfully created.

5 Over the air update

5.1 OTA (Over-the-Air) updates

The following section gives instructions on how to generate, sign, deploy, and update the firmware. It also describes all the tools provided with this solution to give context to what is happening. This section assumes that the `SLN-TLHMI-IOT` kit has been migrated to communicate with a non-NXP AWS IoT Cloud server and the reader has access with the correct permissions. OTA (Over-the-Air) updates are the process of pushing new firmware from a remote service down to a connected device. When it happens, the device programs the new image into the flash and reboots into that image assuming all necessary checks have passed. As shown in the architecture section of this document, there are two application partitions. The application is always going to run into one of these sections. It means that the second section is free to write into without affecting the existing image. It also ensures that the device is safe to jump into the new image without worrying about being compromised assuming the relevant checks have been made. The `SLN-TLHMI-IOT` kit leverages the Amazon OTA service within AWS IoT. This also

leverages the Amazon FreeRTOS OTA client to check for updates and download the image.

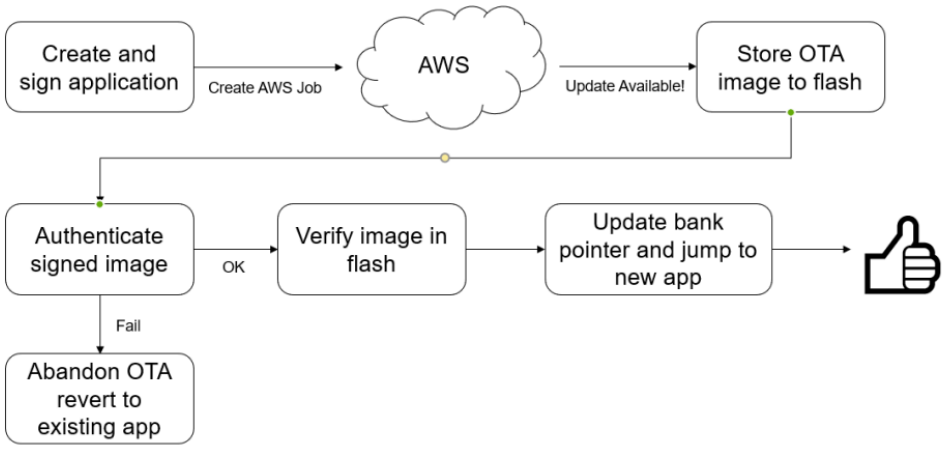


Figure 16. OTA high-level architecture

5.1.1 Migration guide

This section provides the steps to migrate the `SLN-TLHMI-IOT` kit to a developer's/ organization's own fully controlled AWS account. If the `SLN-TLHMI-IOT` kit is left connected to the default server, it is managed by NXP and restricts the developer's access and control of certain features. The unavailable features are described in the `SLN-TLHMI-IOT-DG`.

The advantages of doing migrating are:

- Full control of OTA jobs and deployment
- Customization of firmware/cloud control

To fully use the aws environment, create an AWS Account.

To communicate with AWS, the device must provide certain artifacts and securely connect to AWS IoT. If the artifacts are provided on the cloud, the device cannot connect successfully. For steps to create an Amazon "Thing", see https://docs.aws.amazon.com/iot/latest/developerguide/create-iot-resources.html The communication between the device and the AWS IoT cloud is secured based on the private key and on the device certificates created together with the Amazon "Thing".

Note: *These steps are not required, as our manufacturing tool scripts (Ivaldi) do all the necessary setups, including "Thing" creation. For more details on Ivaldi, see Automated manufacturing tools.*

5.1.1.1 RT117H firmware changes

This section provides an overview of steps to make the necessary source code changes to ensure that the firmware communicates with the correct AWS Account.

As prerequisites:

- an AWS Account is created.
- the Get Started with MCUXpresso Tool suite and Building and Programming sections in the MCU-SMHMI-SDUG guide are read.

- the projects are in your workspace and you are ready to make code changes

The change is required only in the coffee machine application. The changes are a must to ensure that the device connects to the correct AWS Endpoint for OTA.

To get started:

1. Follow the IoT Console Sign-in online resource to log in to the desired account.
2. Navigate to the AWS IoT Core service which opens the console.
3. Within the AWS IoT Console, select the **Settings** button down toward the bottom left section of the page as shown in Figure 17 below.

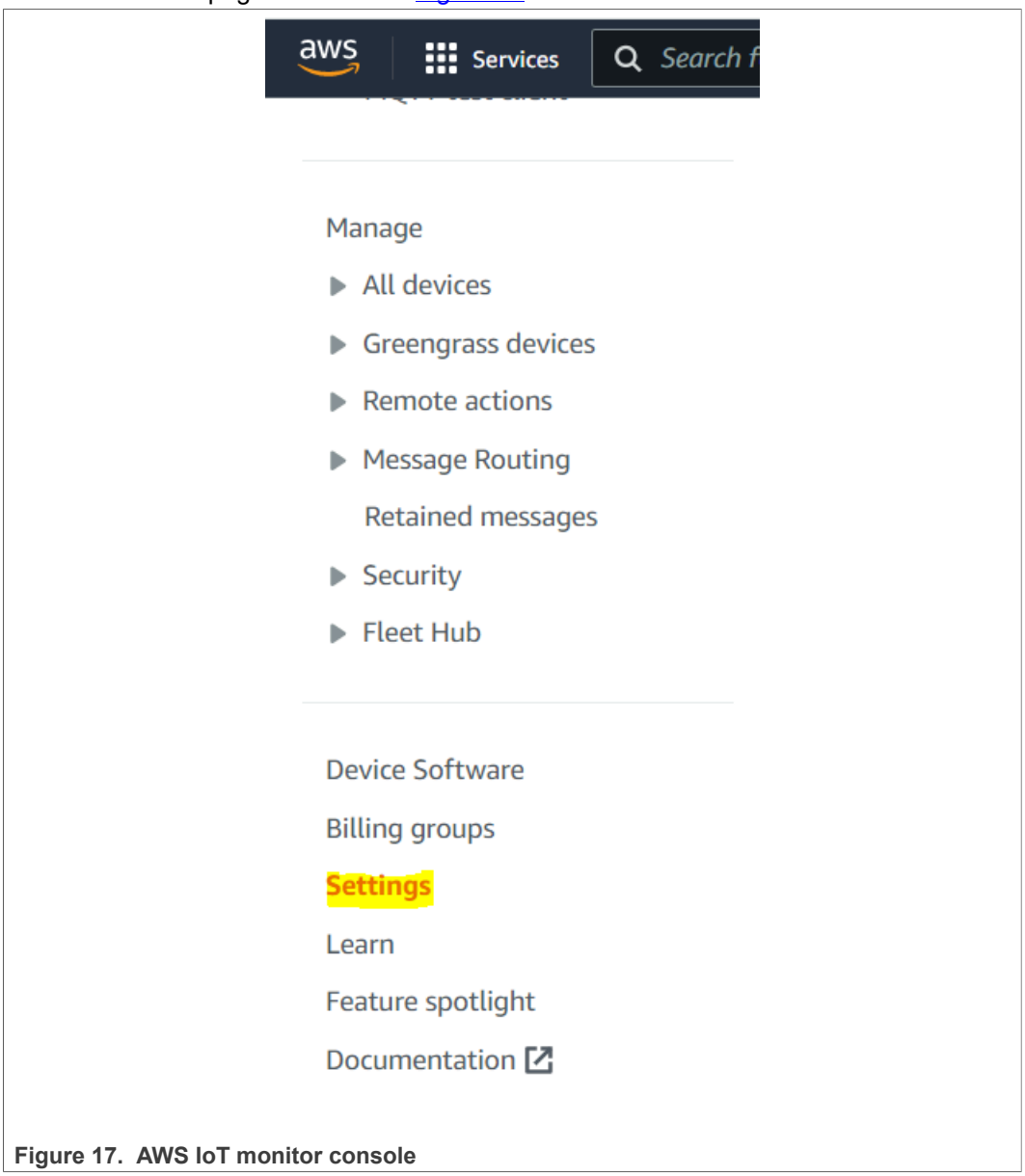


Figure 17. AWS IoT monitor console

Warning:
Ensure that the correct server location for the device that was created is used. If the wrong server is used, it causes a connection issue.

4. It opens the Settings page that has controls for logging and events. At the top of the page, there are **Endpoint Settings**. Copy the endpoint string, which has the following structure `"id".iot."server".amazon.com`.

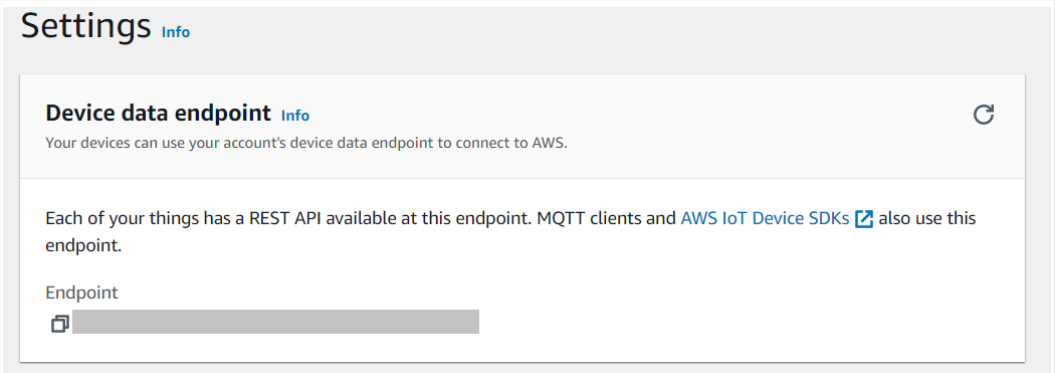


Figure 18. AWS Custom endpoint URL

5. The endpoint is obtained and must be inserted into the firmware. Within the bootloader application, navigate to the **source > aws_clientcredential.h** file. Within the `aws_clientcredential.h` file, locate the array called `clientcredentialMQTT_BROKER_ENDPOINT` and change the existing contents to the endpoint obtained from AWS IoT Endpoint Settings.

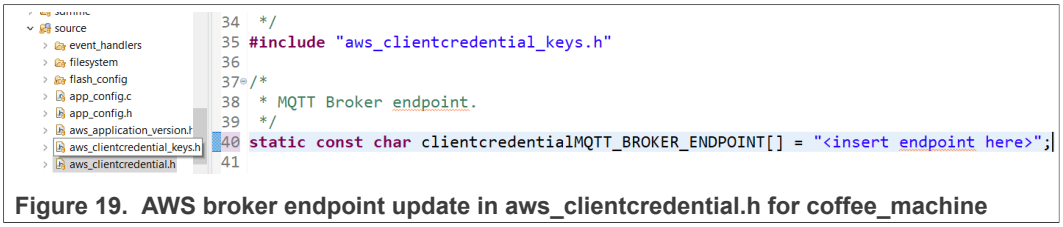


Figure 19. AWS broker endpoint update in aws_clientcredential.h for coffee_machine

5.1.1.2 Ivaldi guide

The following section describes the steps to set up the Ivaldi environment. This chapter assumes that the client has already downloaded and unzipped the `Ivaldi_sln_tlhmi_iot.zip` package. For additional details, check Section 3.1.

Perform the following steps to configure the Ivaldi environment.

Note: *These steps must be executed only once. Ensure that none of the commands return errors. For additional details, check the* `Ivaldi_sln_tlhmi_iot/README.md` *and* `Ivaldi_sln_tlhmi_iot/Scripts/ota_signing/README.md` *files. The Ivaldi tool was tested on the below Operating Systems and the corresponding Command-Line Interfaces:*

- *Linux – Bash CLI*
- *Windows – WSL (Windows Subsystem for Linux)*
- *CLI*
- *Windows – CMD (Command Prompt) CLI*

1. Install the following tools.
   - OpenSSL # to check if installed: openssl version
   - AWS CLI # to check if installed: aws --version
     - Must be configured according to your account # to configure: `aws configure`
     - https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html
     - https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html

- Python 3.6.x

2. Set up the environment and install the requirements. Open a CLI (from the list mentioned above) and run the below commands.
   - `cd Ivaldi_sln_tlhmi_iot/`
   - `pip install virtualenv #` installs the virtual environment tool
   - `virtualenv env #` generates a new virtual environment
   - `source env/bin/activate #` activates the virtual environment (on Linux or WSL)
   - `env\Scripts\activate #` activates the virtual environment (on CMD)
   - `(env) pip install -r requirements.txt #` installs the python dependencies
   - `(env) python setup.py install #` setups the environment.
3. Generate the certificates. Adjust the below command's parameters according to your needs (replace: [code], [country], [state], [org]) and run it within the same terminal opened in the previous step. The script below asks for the password several times, each time provide the same password. As a result, the `Ivaldi_sln_tlhmi_iot/Scripts/ ota_signing/ca/` folder containing all the required certificates is created.
   - `cd Scripts/ota_signing/`
   - `(env) python generate_signing_artifacts.py prod [code] [country] [state] [org]`

   Example: (env) python `generate_signing_artifacts.py` prod FR France Normandy NXP
4. Add the previously generated certificates in the filesystem that is going to be deployed on the board. To do that, add the path for the file in `Scripts/sln_platforms_config/sln_tlhmi_iot_config/littlefs_file_list.py`
5. Add the password provided in Step 3 to the ivaldi scripts. This approach of providing the password is not recommended due to security reasons, but may be used for a quick test of the setup.
   - Open the `Scripts/ota_signing/sign/sign_me.py` file and add the password on line 49 (example: PKEY_PASS = 'my_password').
   - Open the `Scripts/ota_signing/sign/bundle_generate_tlhmi.py` file and add the password on line 139 (PKEY_PASS = 'my_password').
6. Test the environment by flashing an open boot device. Connect the device to the PC via USB. Make sure you have all the required demos inside the Image Binaries folder and that the serial mode jumper is properly set. Within the same terminal as before, run the below commands.
   - `(env) cd ../sln_tlhmi_iot_open_boot/`
   - `(env) python open_prog_full.py`

5.1.2 Preparing an OTA image

This section describes the steps to create a binary to update the demo app. When building an OTA image, make sure to properly sign the image that will be sent. Image authentication is a key factor in the AWS high-level architecture. As the SLN-TLHMI-IOT kit is built to communicate with an NXP demonstration AWS IoT account, OTA is managed by NXP. For OTA to be managed by the developer, the Migration Guide must be executed to provide access to an AWS IoT Core implementation for OTA management. Without this process, OTA is not manageable for the developer. Before starting, check the Ivaldi tool

5.1.3 Building image

As mentioned before in Section 4.4 , the current bootloader enables the remapping feature that helps customers easily deploy new images, without keeping track of the currently active bank. All bootable images must be built with Flash address at `0x30100000`. The current implementation supports update with the same image version or an older version. Best practices dictate that the version must be always higher. To re-enable this functionality set `otaconfigAllowDowngrade` to 0 inside the `ota_config.h` file.

5.1.4 Sign Image

The following section describes what the NXP Application Image Signing Tool (Signing Tool) is and how to use it. The Signing Tool is a python application that is responsible for using a signed Certificate Signing Request (CSR) to sign the binaries and append the certificate to the binary ready to be deployed to the AWS IoT OTA service. The Signing Tool requires Python3 to run. The following instructions assume that the README file in the Ivaldi root directory has been followed to set up the Python virtual environment. If this is not done, the scripts fail. Navigate to the `Scripts/ota_signing` directory inside Ivaldi. For more details, check the “QUICK SETUP” section from the `Scripts/ota_signing/README.md` file.

5.1.4.1 Creating a root, intermediate pair with sign server, and certificates

A tool was created to generate all the artifacts needed for OTA signing. This tool is called `generate_signing_artifacts.py` and was derived from publicly available information for generating CA certificate artifacts. The `generate_signing_artifacts.py` takes 5 parameters that are all used to create the artifacts. The `ca_name` is the entity where all the file names are labeled and used as the common name. It asks you to enter a “pass phrase” and enter the same each time. Once `generate_signing_artifacts.py` succeeds, a “ca” folder inside `Scripts/ota_signing` appears. Inside the “ca” folder you can find: “certs” and “private” folders.

Inside the “certs” folder there are 3 files:

- “`<ca_name>.app.a.crt.pem`”
- “`<ca_name>.app.b.crt.pem`”
- “`<ca_name>.root.ca.crt.pem`”.

Inside the “private” folder there are 3 files:

- “`<ca_name>.app.a.key.pem`”
- “`<ca_name>.app.b.key.pem`”
- “`<ca_name>.root.ca.key.pem`”

```
(env) User@TLHMI :/ivaldi/Scripts/ota_signing$ python3 generate_signing_artifacts.py
Usage:
       generate_signing_artifacts.py ca_name country code country_name state organization

       ca_name: Name of CA for image signature chain of trust

       country code: GB/US

       country_name: CA Country Name

       state: CA Country State

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

Figure 20. generate_signing_artifacts.py description, usage, and logs

The script has been run from the Windows Linux subsystem, but it can be run from any terminal.

The Ivaldi tools should have access to the password used in the previous step for running the `generate_signing_artifacts.py` script. To achieve this, two files must be modified:

- Open the `Scripts/ota_signing/sign/sign_me.py` file and add the password on line 49 (example: PKEY_PASS = 'my_password').
- Open the `Scripts/ota_signing/sign/bundle_generate_tlhmi.py` file and add the password on line 139 (PKEY_PASS = 'my_password').

Note: *This approach of providing the password is not recommended due to security reasons, but may be used for a quick test of the setup.*

Navigate into the `Scripts/ota_signing/sign` folder and run the `sign_me.py` tool with the name of the binary to sign (for example `ais_ffs_demo` binary) and the certificate name (for example, the `prod.app.a` that we have generated in the previous step) for the entity.

5.1.4.2 Formatting the CA and the application certificate

For the device to be able to verify the image signature, it must have the root CA certificate. **(`ca/certs/<cert_name>.root.ca.crt.pem`)** and the application certificate derived from the signing entity **(`ca/ certs/ <cert_name>.app.a.crt.pem`)** .

The certificates do not have a specific address at which to be written, both need to be included in the filesystem. The obtained filesystem is going to be transformed into binary format and loaded with the rest of the images. It is done when running the `open_prog_full.py` script. Generate all the needed certificates before running the script.

5.1.5 OTA Workflow with AWS IoT Console

On the device side, if the filesystem has been properly loaded and the board is connected to a WiFi network, the application creates a secure MQTT connection with the AWS cloud. MQTT connection is used to receive push update requests from the AWS cloud.

To use Amazon OTA, configure various roles to allow AWS IoT access to the S3 Bucket (this is the server that holds your images). The following link was used by NXP to configure their OTA service:https://docs.aws.amazon.com/freertos/latest/userguide/ota-prereqs.html

To create an OTA Job, follow these steps:

1. Navigate to the following link: https://docs.aws.amazon.com/freertos/latest/userguide/ota-console-workflow.html. Focus on the area named “Use my custom-signed firmware image” as this is the process that focuses on custom-signed image creation. No other way of deploying images is currently supported. Click the **Create job** button inside the **AWS IoT > Jobs tab**.
2. A new window appears. Inside this window, select Create FreeRTOS OTA update job as shown in Figure 21:

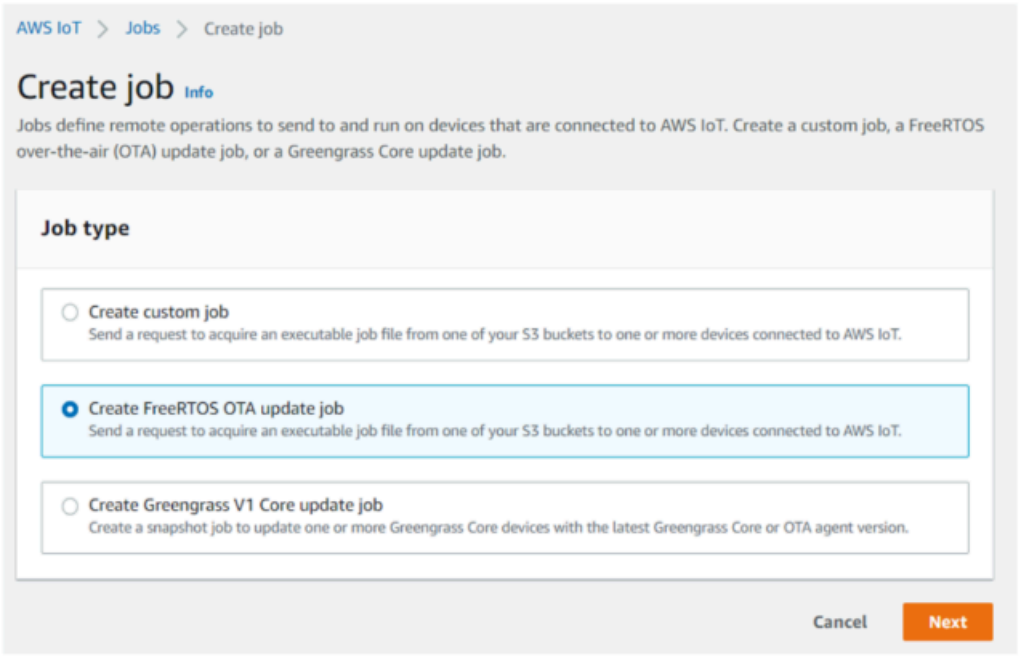


Figure 21. Create OTA job – Job types

3. The OTA Job Properties window appears. Provide a job name as shown Figure 22:

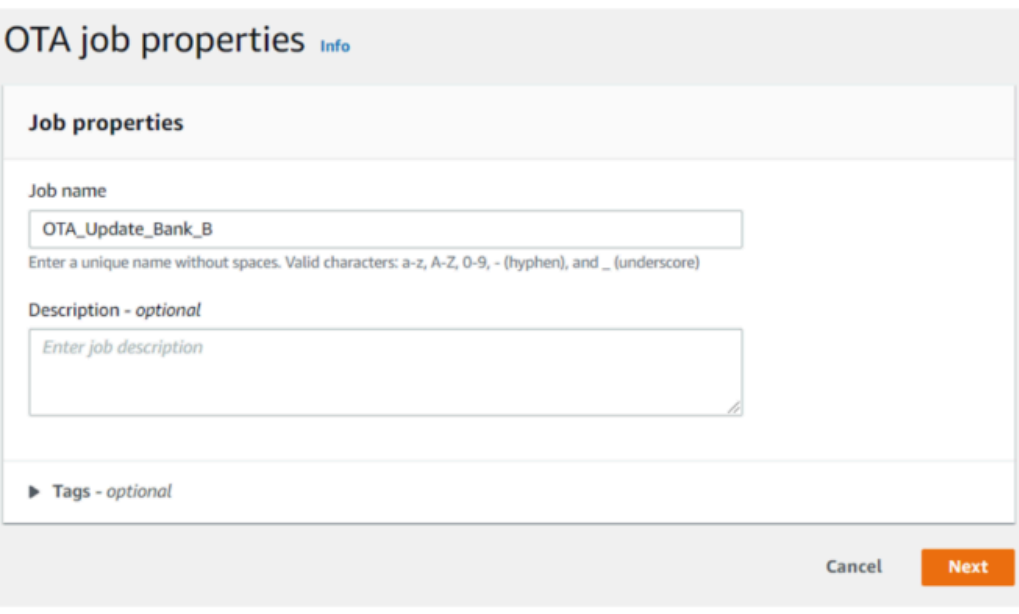


Figure 22. Create OTA job – Job name

4. The OTA File Configuration window appears. Specify the serial numbers of the devices to be updated. Select the MQTT option as the protocol for file transfer as shown in Figure 23 :

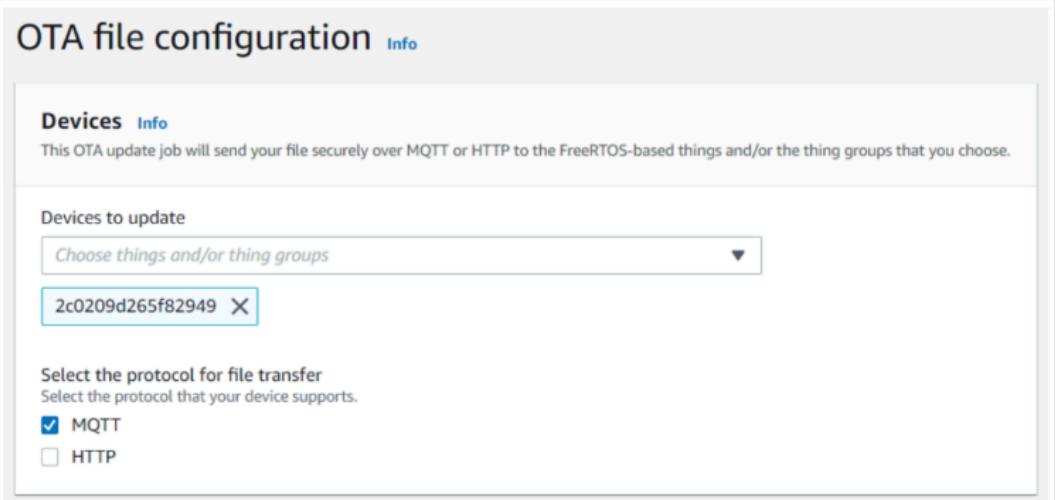


Figure 23. Create OTA job – Devices to update and protocol for file transfer

5. Select the image that is going to be delivered to the remote device. To do this, select **Use my custom signed file** and copy in the **Signature** textbox the content that has been obtained as the output of the Signing Tool (`sln_demo_new_img.bin.sha256.txt`). The following fields must be properly set:
   - Original hash algorithm - `SHA-256`
   - Original encryption algorithm - `RSA`
   - Path name of code signing certificate on device - `app_a_sign_cert.dat` (check littlefs_file_list.py for the name of the file)

Check the images below for more information.

If a new image is going to be loaded, check **Upload a new file**, click **Choose file** and select the image. S3 storage address must be specified in the "`S3 URL`" field. If the loaded binary image already exists in the location, the user can select the checkbox corresponding to **Select an existing file** and use the existing image.

The binary size increases exponentially when adding the GUI resources. Almost 70 % of the total size is occupied by those. To speed up the development and to decrease the load on the updating mechanism, the image has been split into **code and resources**, both with the fixed address in the flash. Update operation can be done on components, or all together into a bundle. Right now the OTA can be used to update:

- Main Application
- Resources
- Bundle update (Main App + Resources)

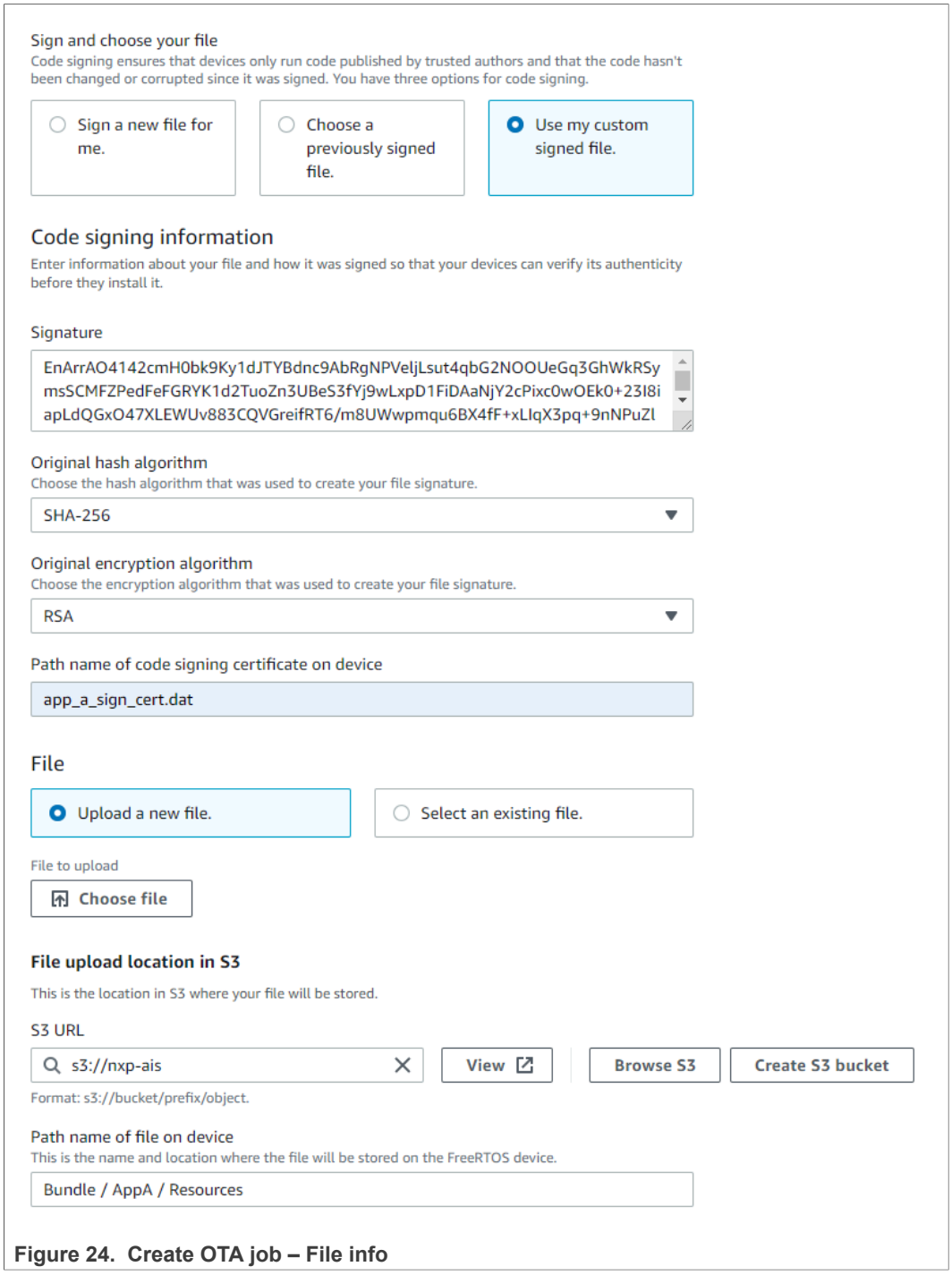


Figure 24. Create OTA job – File info

Until now the configuration for the update was the same. The difference, as was for the MSD, is in the name of the file that must be updated in the **Path name of the file on the device**. The files should be completed with:

- AppA , when updating the main application
- Resources, when updating only resources
- Bundle, update both at the same time

5.1.5.1 Update main application

Because of the remap functionality enabled in the bootloader, this binary can be placed in each of the three banks and still work as it is running from the base address. When receiving an OTA request, the OTA_Agent checks for the unused bank. The empty bank is erased to prepare it for the update. All the erase is done before starting to receive actual data. It is a measure to work around the not-in-order MQTT packets' arrival. After the new image has been written, verification is done to check the signature. Using the **Signature** field and **Path name of the code signing certificate on device** field, the main application can start validating the new image. If everything is right, a resource copy is done, and the empty bank is set as an active bank. It means that during the update procedure the resources stay the same.

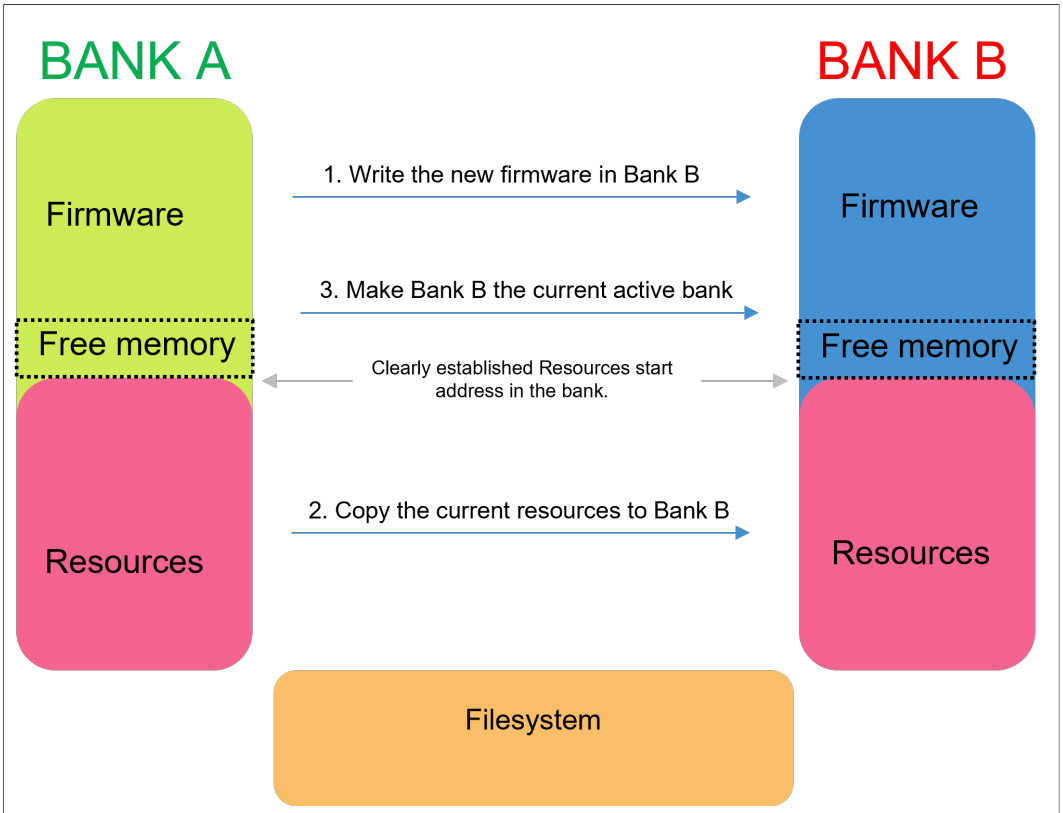


Figure 25. Update main application

5.1.5.2 Update resources

Similarly to updating the main application, the OTA_Agent on request checks for active bank and writes the binary in the opposite one. A complete erase is done beforehand. After the write is completed, the older firmware is copied, and the new bank is activated.

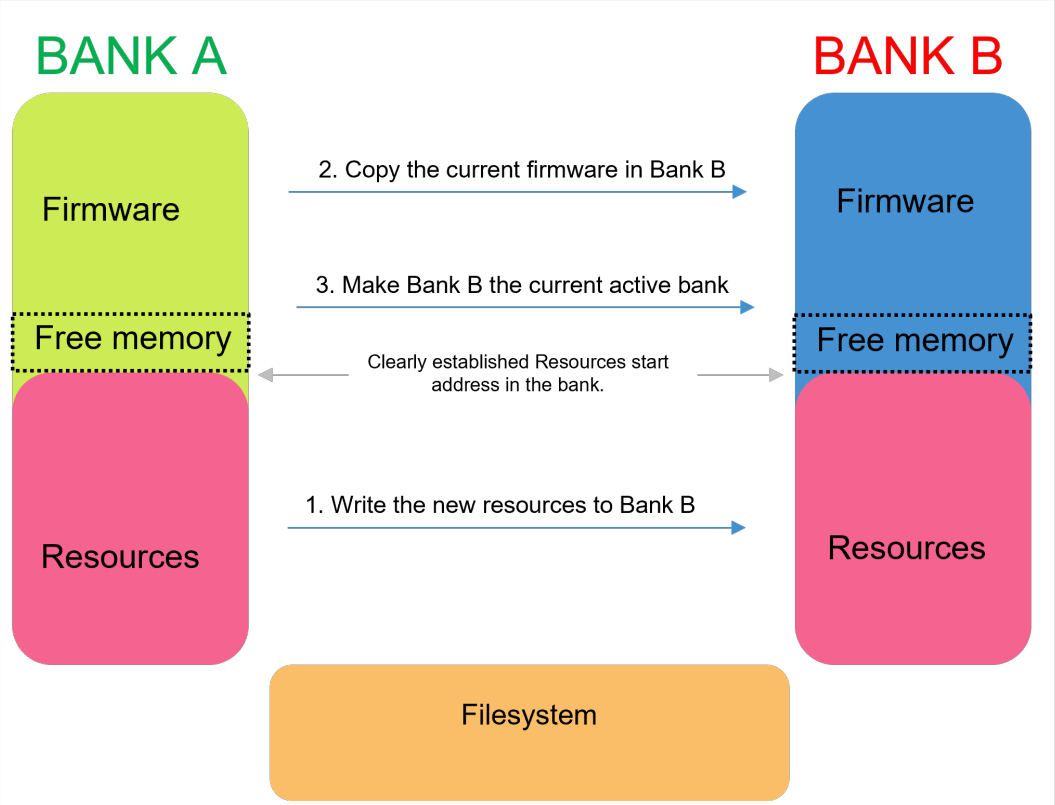


Figure 26. Update resources

5.1.5.3 Update with Bundle

To update with a bundle, a python script is used to generate the bundle. The script is part of the ivaldi suites of scripts that are delivered to the customer. The script is called `bundle_generate_tlhmi.py`. When calling it, two parameters must be set, both being the location of two important files:

- bundle configuration file (-bf) - contains a list of files that are going to be fused to generate the bundle
- board configuration file (-cf) - position of the files in flash to build the metadata.

After running the script, there is no need to pass the binary through the singing process as this script generates a signature used by the device to validate the new image.

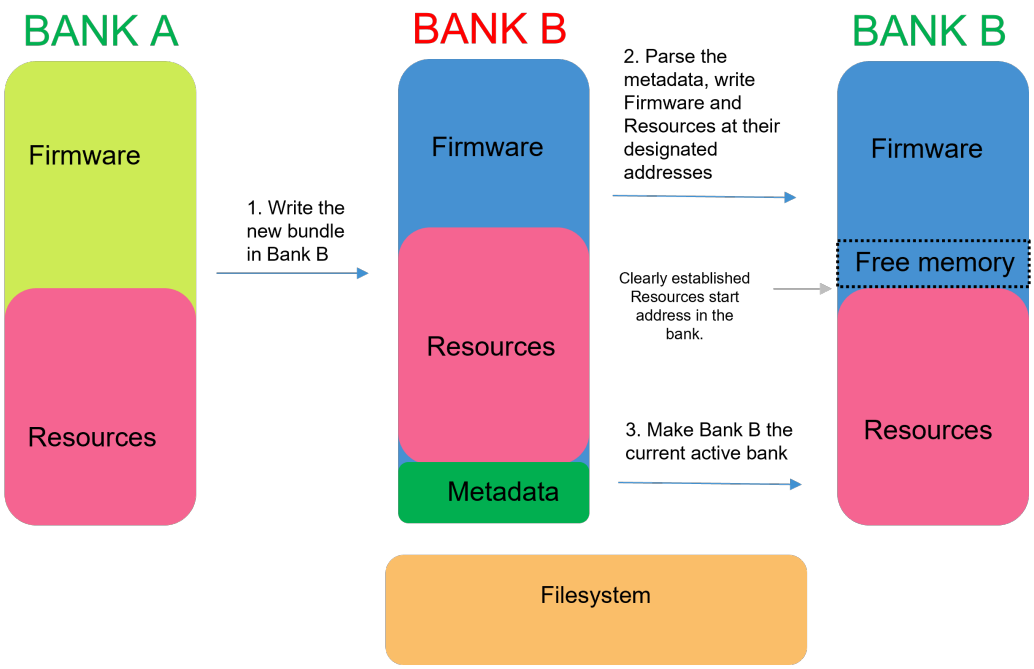


Figure 27. Update bundle

The current firmware sets all the images in the right positions based on the metadata. After the parsing of the bundle is complete and all images are placed accordingly to the `fica_definitions.h` file, the new bank is activated.

After completion, the application reboots in self-test mode. For now, nothing is done in self-test mode except checking for the version of the new application. Reboot to make sure self-test mode is not used.

6 Framework

6.1 Framework introduction

This section describes the architectural design of the framework. The application is primarily designed around the use of a "framework" architecture that is composed of several different parts.

The constituent parts include:

- Device Managers
- Hardware Abstraction Layer (HAL) Devices
- Messages/Events

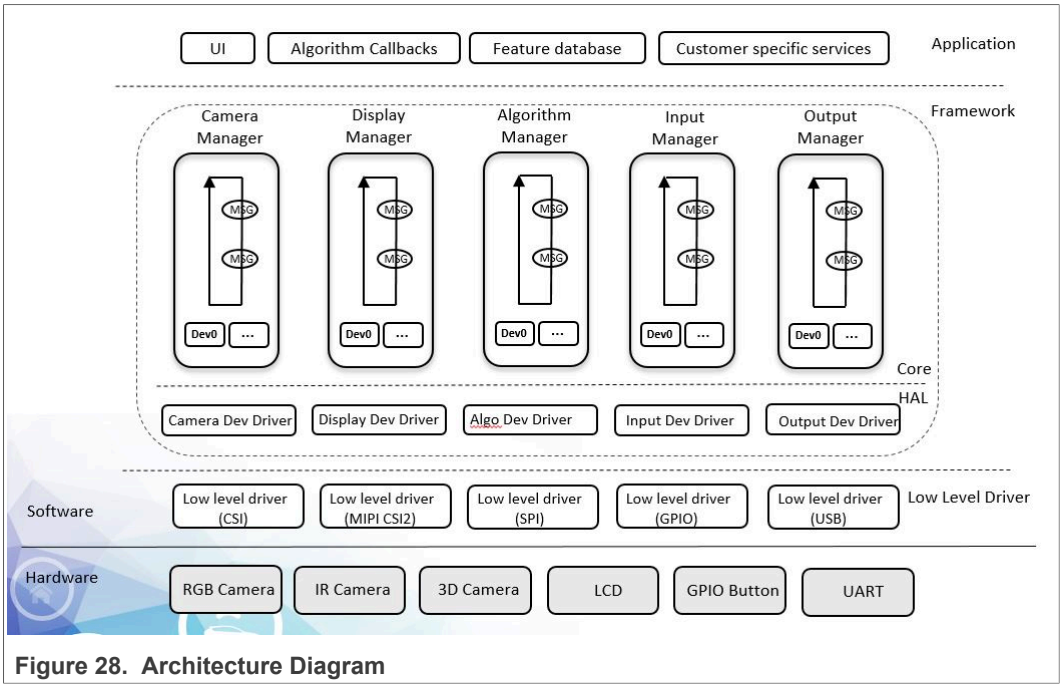


Figure 28. Architecture Diagram

Each of these different components is discussed in detail in the following sections.

6.1.1 Design goals

The architectural design of the framework was centered around 3 primary goals:

1. Ease-of-use
2. Flexibility/Portability
3. Performance

In the course of project development, many problems can arise which hinder the speed of that development. The framework architecture was designed to help combat those problems.

The framework is designed with the goal of speeding up the time to market for vision and other machine-learning applications. To ensure a speedy time to market, it is critical that the software itself is easy to understand and modify. Keeping this goal in mind, the architecture of the framework is easy to modify without being restrictive, and without coming at the cost of performance.

6.1.2 Relevant files

The files which pertain to the framework architecture can primarily be found in the `framework/` folder of the specific application. Because the application is designed around the use of the framework architecture, it is likely that the bulk of a developer's efforts will be focused on the contents of these folders.

6.2 Naming conventions

The framework code adheres to a set of naming conventions for making the code easily readable and searchable using modern code completion tools.

Note: *The naming conventions described below apply *only* to framework-related code that is primarily located in the* `framework` *folder and* `source` *folder of the application.*

6.2.1 Functions

Functions names follow the format of `{APP/FWK/HAL}_{DevType}_{DevName}_{Action}`.

For example:

```
hal_input_status_t HAL_InputDev_PushButtons_Start(const
 input_dev_t *dev);
```

To increase searchability using code completion tools, functions for each framework component have their own prefix denoting the component they relate to:

- `APP` - app-specific function. Usually device registration or event handler-related.
- `FWK` - framework-specific function. Usually framework API function.
- `HAL` - HAL-specific function. Usually HAL device operators.

Additionally, an underscore `_` may be placed in front of a function name to indicate that the function is `static`/`private`.

Note: *Static functions oftentimes exclude all but the underscore and the `Action` as the component, devType, and devName are implicit.*

For example:

```
static shell_status_t _VersionCommand(shell_handle_t
 shellContextHandle, int32_t argc, char **argv);
static shell_status_t _ResetCommand(shell_handle_t
 shellContextHandle, int32_t argc, char **argv);
static shell_status_t _SaveCommand(shell_handle_t
 shellContextHandle, int32_t argc, char **argv);
static shell_status_t _AddCommand(shell_handle_t
 shellContextHandle, int32_t argc, char **argv);
static shell_status_t _DelCommand(shell_handle_t
 shellContextHandle, int32_t argc, char **argv);
```

One of the above prefixes is the device type of the device defining the function.

- `InputDev`
- `OutputDev`
- `CameraDev`
- `DisplayDev`
- and so forth.

As the device type is the name of the device, the name must match the name of the device specified in the filename.

For example:

```
hal_input_status_t HAL_InputDev_PushButtons_Start(const
 input_dev_t *dev);
```

The name of the device is the "action" performed on/by the device. It could be anything including `Start`, `Stop`, `Register`, and so on.

Below are several examples of different function names:

```
void APP_InputDev_Shell_RegisterShellCommands(shell_handle_t
 shellContextHandle,
                                                input_dev_t
 *shellDev,

 input_dev_callback_t callback)
{
    s_InputCallback            = callback;
    s_SourceShell              = shellDev;
    s_ShellHandle              = shellContextHandle;
    s_FrameworkRequest.respond = _FrameworkEventsHandler;
    SHELL_RegisterCommand(shellContextHandle,
 SHELL_COMMAND(version));
    SHELL_RegisterCommand(shellContextHandle,
 SHELL_COMMAND(reset));
    SHELL_RegisterCommand(shellContextHandle,
 SHELL_COMMAND(save));
    SHELL_RegisterCommand(shellContextHandle,
 SHELL_COMMAND(add));
```

```
int HAL_InputDev_PushButtons_Register()
{
    int error = 0;
    LOGD("input_dev_push_buttons_register");
    error =
 FWK_InputManager_DeviceRegister(&s_InputDev_PushButtons);
    return error;
}
```

```
hal_input_status_t HAL_InputDev_PushButtons_Init(input_dev_t
 *dev, input_dev_callback_t callback);
hal_input_status_t HAL_InputDev_PushButtons_Deinit(const
 input_dev_t *dev);
hal_input_status_t HAL_InputDev_PushButtons_Start(const
 input_dev_t *dev);
hal_input_status_t HAL_InputDev_PushButtons_Stop(const
 input_dev_t *dev);
hal_input_status_t HAL_InputDev_PushButtons_InputNotify(const
 input_dev_t *dev, void *param);
```

6.2.2 Variables

Local and global variables use `camelCase`.

```
static hal_output_status_t
 HAL_OutputDev_RgbLed_InferComplete(const output_dev_t *dev,

 output_algo_source_t source,

 void *inferResult)
{
    vision_algo_result_t *visionAlgoResult =
 (vision_algo_result_t *)inferResult;
    hal_output_status_t error              =
 kStatus_HAL_OutputSuccess;
```

Static variables are prefixed with `s_PascalCase`

For example:

```
static event_common_t s_CommonEvent;
static event_face_rec_t s_FaceRecEvent;
static event_recording_t s_RecordingEvent;
static input_event_t s_InputEvent;
static framework_request_t s_FrameworkRequest;
static input_dev_callback_t s_InputCallback;
static input_dev_t *s_SourceShell; /* Shell device that
 commands are sent over */
static shell_handle_t s_ShellHandle;
```

6.2.3 Typedefs

Type definitions are written in `snake_case` and end in `_t`.

For example:

```
typedef struct
{
    fwk_task_t task;
    input_task_data_t inputData;
} input_task_t;
```

6.2.4 Enums

Enumerations are written in the the form `kEventType_State`.

For example:

```
typedef enum _rgb_led_color
{
    kRGBLedColor_Red,    /*!< LED Red Color */
    kRGBLedColor_Orange, /*!< LED Orange Color */
    kRGBLedColor_Yellow, /*!< LED Yellow Color */
    kRGBLedColor_Green,  /*!< LED Green Color */
    kRGBLedColor_Blue,   /*!< LED Blue Color */
    kRGBLedColor_Purple, /*!< LED Purple Color */
    kRGBLedColor_Cyan,   /*!< LED Cyan Color */
    kRGBLedColor_White,  /*!< LED White Color */
    kRGBLedColor_Off,    /*!< LED Off */
} rgbLedColor_t;
```

Enumerations for a status specifically must be written in the form `kStatus_{Component}_{State}`.

For example:

```
/*! @brief Error codes for input hal devices */
typedef enum _hal_input_status
{
    kStatus_HAL_InputSuccess = 0,
                         /*!< Successfully */
```

```
    kStatus_HAL_InputError   =
 MAKE_FRAMEWORK_STATUS(kStatusFrameworkGroups_Input, 1), /*!<
 Error occurs */
} hal_input_status_t;
```

6.2.5 Macros and Defines

Defines are written in all caps.

For example:

```
#define INPUT_DEV_PB_WAKE_GPIO        BOARD_USER_BUTTON_GPIO
#define INPUT_DEV_PB_WAKE_GPIO_PIN
 BOARD_USER_BUTTON_GPIO_PIN
#define INPUT_DEV_SW1_GPIO            BOARD_BUTTON_SW1_GPIO
#define INPUT_DEV_SW1_GPIO_PIN        BOARD_BUTTON_SW1_PIN
#define INPUT_DEV_SW2_GPIO            BOARD_BUTTON_SW2_GPIO
#define INPUT_DEV_SW2_GPIO_PIN        BOARD_BUTTON_SW2_PIN
#define INPUT_DEV_SW3_GPIO            BOARD_BUTTON_SW3_GPIO
#define INPUT_DEV_SW3_GPIO_PIN        BOARD_BUTTON_SW3_PIN
#define INPUT_DEV_PUSH_BUTTONS_IRQ    GPIO13_Combined_0_31_IRQn
#define INPUT_DEV_PUSH_BUTTON_SW1_IRQ BOARD_BUTTON_SW1_IRQ
#define INPUT_DEV_PUSH_BUTTON_SW2_IRQ BOARD_BUTTON_SW2_IRQ
#define INPUT_DEV_PUSH_BUTTON_SW3_IRQ BOARD_BUTTON_SW3_IRQ
```

6.3 Device managers

6.3.1 Overview

As the name would imply, device managers are responsible for "managing" devices used by the system. Each device type (input, output, and so on) has its own type-specific device manager.

A device manager serves two primary purposes:

- Initializing and starting each device registered to that manager
- Sending data to and receiving data from each device registered to that manager

This section avoids low-level implementation details of the device managers and instead focus on the device manager APIs and the startup flow for the device managers. The device managers themselves are provided as a library binary file to, in part, help abstract the underlying implementation details and encourage developers to focus on the HAL devices being managed instead.

Note: *The device managers themselves are provided as a library binary file in the* `framework` *folder, while the APIs for each manager can be found in the* `framework/inc` *folder.*

6.3.1.1 Initialization flow

Before a device manager can properly manage devices, it must follow a specific startup process. The startup process for device managers is summarized as follows:

1. Initialize managers
2. Register each device to their respective manager
3. Start managers

This process is clearly demonstrated in the `main` function found in `source/main.cpp`

```
/*
 * @brief   Application entry point.
 */
int main(void)
{
    /* Init board hardware. */
    APP_BoardInit();
    LOGD("[MAIN]:Started");
    /* init the framework*/
    APP_InitFramework();

    /* register the hal devices*/
    APP_RegisterHalDevices();

    /* start the framework*/
    APP_StartFramework();

    // start
    vTaskStartScheduler();

    while (1)
    {
        LOGD("#");
    }

    return 0;
}
```

As part of a manager's `start` routine, the manager calls the `init` and `start` functions of each of its registered devices.

Note: *In general, developers must only be concerned aout adding/removing devices from the `APP_RegisterHalDevices()` function as the `init` and `start` functions for each manager are already called by default inside the `APP_InitFramework()` and `APP_StartFramework()` functions in `main()`.*

6.3.2 Vision input manager

The Vision input manager manages the input HAL devices that can be registered into the system.

6.3.2.1 APIs

6.3.2.1.1 FWK_InputManager_Init

```
/**
 * @brief Init internal structures for input manager.
 * @return int Return 0 if the init process was successful
 */
int FWK_InputManager_Init();
```

6.3.2.1.2 FWK_InputManager_DeviceRegister

```
/**
```

```
 * @brief Register an input device. All input devices need to
 be registered before FWK_InputManager_Start is called.
 * @param dev Pointer to a display device structure
 * @return int Return 0 if registration was successful
 */
int FWK_InputManager_DeviceRegister(input_dev_t *dev);
```

6.3.2.1.3 FWK_InputManager_Start

```
/**
 * @brief Spawn Input manager task which will call init/start
 for all registered input devices
 * @return int Return 0 if the starting process was successful
 */
int FWK_InputManager_Start();
```

6.3.2.1.4 FWK_InputManager_Deinit

```
/**
 * @brief Denit internal structures for input manager.
 * @return int Return 0 if the deinit process was successful
 */
int FWK_InputManager_Deinit();
```

Note: *Calling this function is unnecessary in most applications and must be used with caution.*

6.3.3 Output manager

The Output manager manages the output HAL devices that can be registered into the system.

6.3.3.1 APIs

6.3.3.1.1 FWK_OutputManager_Init

```
/**
 * @brief Init internal structures for output manager.
 * @return int Return 0 if the init process was successful
 */
int FWK_OutputManager_Init();
```

6.3.3.1.2 FWK_OutputManager_DeviceRegister

```
/**
 * @brief Register a display device. All display devices need
 to be registered before FWK_OutputManager_Start is called.
 * @param dev Pointer to an output device structure
 * @return int Return 0 if registration was successful
 */
int FWK_OutputManager_DeviceRegister(output_dev_t *dev);
```

6.3.3.1.3 FWK_OutputManager_Start

```
/**
 * @brief Spawn output manager task which will call init/start
 for all registered output devices.
 * @return int Return 0 if starting was successful
 */
int FWK_OutputManager_Start();
```

6.3.3.1.4 FWK_OutputManager_Deinit

```
/**
 * @brief DeInit internal structures for output manager.
 * @return int Return 0 if the deinit process was successful
 */
int FWK_OutputManager_Deinit();
```

```
Calling this function is unnecessary in most applications and
 should be used with caution.
```

```
/**
 * @brief A registered output device doesn't need to be also
 active. After the start procedure, the output device
 *          can register a handler of capabilities to receive
 events.
 * @param dev Device that register the handler
 * @param handler Pointer to a handler
 * @return int Return 0 if the registration of the event
 handler was successful
 */
int FWK_OutputManager_RegisterEventHandler(const output_dev_t
 *dev, const output_dev_event_handler_t *handler);
```

6.3.3.1.5 FWK_OutputManager_UnregisterEventHandler

```
/**
 * @brief A registered output device doesn't need to be also
 active. A device can call this function to unsubscribe
 * from receiving events
 * @param dev Device that unregister the handler
 * @return int Return 0 if the deregistration of the event
 handler was successful
 */
int FWK_OutputManager_UnregisterEventHandler(const output_dev_t
 *dev);
```

6.3.4 Camera manager

Camera manager manages the camera HAL devices that can be registered into the system.

6.3.4.1 APIs

6.3.4.1.1 FWK_CameraManager_Init

```
/**
 * @brief Init internal structures for Camera manager.
 * @return int Return 0 if the init process was successful
 */
int FWK_CameraManager_Init();
```

6.3.4.1.2 FWK_CameraManager_DeviceRegister

```
/**
 * @brief Register a camera device. All camera devices need to
 be registered before FWK_CameraManager_Start is called
 * @param dev Pointer to a camera device structure
 * @return int Return 0 if registration was successful
 */
int FWK_CameraManager_DeviceRegister(camera_dev_t *dev);
```

6.3.4.1.3 FWK_CameraManager_Start

```
/**
 * @brief Spawn Camera manager task which will call init/start
 for all registered camera devices
 * @return int Return 0 if the starting process was successul
 */
int FWK_CameraManager_Start();
```

6.3.4.1.4 FWK_CameraManager_Deinit

```
/**
 * @brief Deinit CameraManager
 * @return int Return 0 if the deinit process was successful
 */
int FWK_CameraManager_Deinit();
```

Note: *Calling this function is unnecessary in most applications and must be used with caution.*

6.3.5 Display manager

The Display manager manages the display HAL devices that can be registered into the system.

6.3.5.1 APIs

6.3.5.1.1 FWK_DisplayManager_Init

```
/**
 * @brief Init internal structures for display manager.
 * @return int Return 0 if the init process was successful
 */
```

```
int FWK_DisplayManager_Init();
```

6.3.5.1.2 FWK_DisplayManager_DeviceRegister

```
/**
 * @brief Register a display device. All display devices need
 to be registered before FWK_DisplayManager_Start is
 * called.
 * @param dev Pointer to a display device structure
 * @return int Return 0 if registration was successful
 */
int FWK_DisplayManager_DeviceRegister(display_dev_t *dev);
```

6.3.5.1.3 FWK_DisplayManager_Start

```
/**
 * @brief Spawn Display manager task which will call init/start
 for all registered display devices. Will start the flow
 * to recive frames from the camera.
 * @return int Return 0 if starting was successful
 */
int FWK_DisplayManager_Start();
```

6.3.5.1.4 FWK_DisplayManager_Deinit

```
/**
 * @brief Init internal structures for display manager.
 * @return int Return 0 if the init process was successful
 */
int FWK_DisplayManager_Deinit();
```

Note: *Calling this function is unnecessary in most applications and must be used with caution.*

6.3.6 Vision algorithm manager

The Vision algorithm manager manages the vision algorithm HAL devices that can be registered into the system.

6.3.6.1 APIs

6.3.6.1.1 FWK_VisionAlgoManager_Init

```
/**
 * @brief Init internal structures for VisionAlgo manager.
 * @return int Return 0 if the init process was successful
 */
int FWK_VisionAlgoManager_Init();
```

6.3.6.1.2 FWK_VisionAlgoManager_DeviceRegister

```
/**
 * @brief Register a vision algorithm device. All algorithm
 devices need to be registered before
```

```
 * FWK_VisionAlgoManager_Start is called
 * @param dev Pointer to a vision algo device structure
 * @return int Return 0 if registration was successful
 */
int FWK_VisionAlgoManager_DeviceRegister(vision_algo_dev_t
 *dev);
```

6.3.6.1.3 FWK_VisionAlgoManager_Start

```
/**
 * @brief Spawn VisionAlgo manager task which will call init/
start for all registered VisionAlgo devices
 * @return int Return 0 if the starting process was successul
 */
int FWK_VisionAlgoManager_Start();
```

6.3.6.1.4 FWK_VisionAlgoManager_Deinit

```
/**
 * @brief Deinit VisionAlgoManager
 * @return int Return 0 if the deinit process was successful
 */
int FWK_VisionAlgoManager_Deinit();
```

Note: *Calling this function is unnecessary in most applications and must be used with caution.*

6.3.7 Voice algorithm manager

The Voice algorithm manager manages the voice algorithm HAL devices that can be registered into the system.

6.3.7.1 APIs

6.3.7.1.1 FWK_VoiceAlgoManager_Init

```
/**
 * @brief Init internal structures for VisionAlgo manager.
 * @return int Return 0 if the init process was successful
 */
int FWK_VoiceAlgoManager_Init();
```

6.3.7.1.2 FWK_VoiceAlgoManager_DeviceRegister

```
/**
 * @brief Register a voice algorithm device. All algorithm
 devices need to be registered before
 * FWK_VoiceAlgoManager_Start is called
 * @param dev Pointer to a vision algo device structure
 * @return int Return 0 if registration was successful
 */
int FWK_VoiceAlgoManager_DeviceRegister(voice_algo_dev_t *dev);
```

6.3.7.1.3 FWK_VoiceAlgoManager_Start

```
/**
 * @brief Spawn VisionAlgo manager task which will call init/
start for all registered VisionAlgo devices
 * @return int Return 0 if the starting process was successful
 */
int FWK_VoiceAlgoManager_Start();
```

6.3.7.1.4 FWK_VoiceAlgoManager_Deinit

```
/**
 * @brief Deinit VisionAlgoManager
 * @return int Return 0 if the deinit process was successful
 */
int FWK_VoiceAlgoManager_Deinit();
```

Note: *Calling this function is unnecessary in most applications and must be used with caution.*

6.3.8 Low-Power device manager

The Low-Power device manager is unique among the managers because it does not have the typical `Init` and `Start` functions that the other managers do. Instead, the Low-Power Manager has APIs to register a device (only one at a time), configure how the board should enter deep sleep, enable sleep mode, and more.

Note: *Due to the unique nature of low-power devices being an abstract "virtual" device, only one LPM device can be registered to the LPM manager at a time. However, there must be no need for more than one LPM device because other devices can configure the current low-power mode states by using the Low-Power Manager APIs.*

6.3.8.1 APIs

6.3.8.1.1 FWK_LpmManager_DeviceRegister

```
/**
 * @brief Register a low power mode device. Currently, only one
 low power mode device can be registered at a time.
 * @param dev Pointer to a low power mode device structure
 * @return int Return 0 if registration was successful
 */
int FWK_LpmManager_DeviceRegister(lpm_dev_t *dev);
```

6.3.8.1.2 FWK_LpmManager_RegisterRequestHandler

```
int FWK_LpmManager_RegisterRequestHandler(hal_lpm_request_t
 *req);
```

6.3.8.1.3 FWK_LpmManager_UnregisterRequestHandler

```
int FWK_LpmManager_UnregisterRequestHandler(hal_lpm_request_t
 *req);
```

6.3.8.1.4 FWK_LpmManager_RuntimeGet

```
int FWK_LpmManager_RuntimeGet(hal_lpm_request_t *req);
```

6.3.8.1.5 FWK_LpmManager_RuntimePut

```
int FWK_LpmManager_RuntimePut(hal_lpm_request_t *req);
```

6.3.8.1.6 FWK_LpmManager_RuntimeSet

```
int FWK_LpmManager_RuntimeSet(hal_lpm_request_t *req, int8_t
 count);
```

6.3.8.1.7 FWK_LpmManager_RequestStatus

```
int FWK_LpmManager_RequestStatus(unsigned int
 *totalUsageCount);
```

6.3.8.1.8 FWK_LpmManager_SetSleepMode

```
/**
 * @brief Configure the sleep mode to use when entering sleep
 * @param sleepMode sleep mode to use when entering sleep.
 Examples include SNVS and other "lighter" sleep modes
 * @return int Return 0 if successful
 */
int FWK_LpmManager_SetSleepMode(hal_lpm_mode_t sleepMode);
```

6.3.8.1.9 FWK_LpmManager_EnableSleepMode

```
/**
 * @brief Configure sleep mode on/off status
 * @param enable used to set sleep mode on/off; true is enable,
 false is disable
 * @return int Return 0 if successful
 */
int FWK_LpmManager_EnableSleepMode(hal_lpm_manager_status_t
 enable);
```

6.3.9 Audio processing manager

The Audio processing manager manages the audio processing HAL devices that can be registered into the system.

6.3.9.1 APIs

6.3.9.1.1 FWK_AudioProcessing_Init

```
/**
 * @brief Init Audio Processing manager
 *
 * @return int Return 0 if the init process was successful
```

```
 */
int FWK_AudioProcessing_Init(void);
```

6.3.9.1.2 FWK_AudioProcessing_DeviceRegister

```
/**
 * @brief Register an audio processing device
 *
 * @param dev Pointer to an Audio Processing device
 * @return int Return 0 if the register was successful
 */
int FWK_AudioProcessing_DeviceRegister(audio_processing_dev_t
 *dev);
```

6.3.9.1.3 FWK_AudioProcessing_Start

```
/**
 * @brief Start Audio Processing manager
 *
 * @return int Return 0 if the starting process was successful
 */
int FWK_AudioProcessing_Start(void);
```

6.3.9.1.4 FWK_AudioProcessing_Deinit

```
/**
 * @brief Deinit Audio Processing manager
 *
 * @return int Return 0 if the deit process was successful
 */
int FWK_AudioProcessing_Deinit(void);
```

Note: *Calling this function is unnecessary in most applications and must be used with caution.*

6.3.10 Flash manager

The Flash manager is used to provide an abstraction for an underlying filesystem implementation.

Due to the unique nature of the filesystem being an abstract "virtual" device, only one flash device can be registered at a time. However, generally there should be no need to have more than one filesystem. It means the Flash manager's API functions essentially act as wrappers that call the operators of the underlying flash HAL device.

Warning: *Flash access is exclusive, one request at a time.*

Note: *When working with the Flash Manager, unlike most other managers,* `FWK_Flash_DeviceRegister` ***must be called _before_*** `FWK_Flash_Init`.

6.3.10.1 Device APIs

6.3.10.1.1 FWK_Flash_DeviceRegister

```
/**
```

```
 * @brief Only one flash device is supported. Registered a
 flash filesystem device
 * @param dev Pointer to a flash device structure
 * @return int Return 0 if registration was successful
 */
int FWK_Flash_DeviceRegister(const flash_dev_t *dev);
```

Note: *Unlike the flow for most other managers, this function must be called before* `FWK_Flash_Init`.

6.3.10.1.2 FWK_Flash_Init

```
/**
 * @brief Init internal structures for flash.
 * @return int Return 0 if the init process was successful
 */
sln_flash_status_t FWK_Flash_Init();
```

6.3.10.1.3 FWK_Flash_Deinit

```
/**
 * @brief Deinit internal structures for flash.
 * @return int Return 0 if the init process was successful
 */
sln_flash_status_t FWK_Flash_Deinit();
```

6.3.10.2 Operations APIs

The Flash Manager and underlying flash HAL device define only a few operations in order to keep the API simple and easy to implement. These API functions include:

- Format
- Save
- Delete
- Read
- Make Directory
- Make File
- Append
- Rename
- Cleanup

While it might limit filesystem functionality, it also helps to keep the code readable, portable, and maintainable.

Note: *If the default list of APIs does not satisfy the requirements of a use-case, the API can always be extended or bypassed in the code directly.*

6.3.10.2.1 FWK_Flash_Format

```
/**
 * @brief Format the filesystem
 * @return the status of formatting operation
 */
sln_flash_status_t FWK_Flash_Format();
```

6.3.10.2.2 FWK_Flash_Save

```
/**
 * @brief Save the data into a file from the file system
 * @param path Path of the file in the file system
 * @param buf  Buffer which contains the data that is going to
 be saved
 * @param size Size of the buffer
 * @return the status of save operation
 */
sln_flash_status_t FWK_Flash_Save(const char *path, void *buf,
 unsigned int size);
```

6.3.10.2.3 FWK_Flash_Append

```
/**
 * @brief Append the data to an existing file.
 * @param path Path of the file in the file system
 * @param buf  Buffer which contains the data that is going to
 be append
 * @param size Size of the buffer
 * @param overwrite Boolean parameter. If true the existing
 file will be truncated. Similar to SLN_flash_save
 * @return the status of append operation
 */
 sln_flash_status_t FWK_Flash_Append(const char *path, void
 *buf, unsigned int size, bool overwrite);
```

6.3.10.2.4 FWK_Flash_Read

```
/**
 * @brief Read from a file
 * @param path Path of the file in the file system
 * @param buf  Buffer in which to store the read value
 * @param offset If reading in chunks, set offset to file
 current position
 * @param size Size that was read.
 * @return the status of read operation
 */
sln_flash_status_t FWK_Flash_Read(const char *path, void *buf,
 unsigned int offset, unsigned int *size);
```

6.3.10.2.5 FWK_Flash_Mkdir

```
/**
 * @brief Make directory operation
 * @param path Path of the directory in the file system
 * @return the status of mkdir operation
 */
sln_flash_status_t FWK_Flash_Mkdir(const char *path);
```

6.3.10.2.6 FWK_Flash_Mkfile

```
/**
 * @brief Make file with specific attributes
```

```
 * @param path Path of the file in the file system
 * @param encrypt Specify if the files should be encrypted.
 Based on FS implementation
 * this param can be neglected
 * @return the status of mkfile operation
 */
sln_flash_status_t FWK_Flash_Mkfile(const char *path, bool
 encrypt);
```

6.3.10.2.7 FWK_Flash_Rm

```
/**
 * @brief Remove file
 * @param path Path of the file that shall be removed
 * @return the status of rm operation
 */
sln_flash_status_t FWK_Flash_Rm(const char *path);
```

6.3.10.2.8 FWK_Flash_Rename

```
/**
 * @brief Rename existing file
 * @param OldPath Path of the file that is renamed
 * @param NewPath New Path of the file
 * @return status of rename operation
 */
sln_flash_status_t FWK_Flash_Rename(const char *oldPath, const
 char *newPath);
```

6.3.10.2.9 FWK_Flash_Cleanup

```
/**
 * @brief Cleanup function. Might imply defragmentation, erased
 unused sectors etc.
 *
 * @param timeout Time consuming operation. Set a time
 constrain to be sure that is not disturbing the system.
 *               Timeout = 0 means no timeout
 * @return status of cleanup operation
 */
sln_flash_status_t FWK_Flash_Cleanup(uint32_t timeout);
```

6.3.11 Multicore manager

The Multicore manager manages the multicore HAL device that can be registered into the system. In the current framework implementation, there are two ways of making a message multicore:

1. isMulticoreMessage flag set to 1
   A message constructed with isMulticoreMessage set to 1, becomes automatically a multicast message and is sent to both cores. The **taskId** field specifies the task that must handle the message from the other core. The below code snip shows how the

message is sent to both CM4/CM7 with the Multicore manager as **the man in the middle**.

```
pVAlgoResMsg->multicore.isMulticoreMessage = 1;
pVAlgoResMsg->multicore.taskId             =
 kFWKTaskID_Output;
FWK_Message_Put(kFWKTaskID_VisionAlgo, &pVAlgoResMsg);
```

If the message has been sent by the CM7/Camera_Manager, the message is sent to CM7/VisionAlgo and to CM4/Output via Multicore Manager

```
FWK_Message_Put(kFWKTaskID_VisionAlgo, &pVAlgoResMsg);
├── Message send to CM7/kFWKTaskID_VisionAlgo
└── Message send to CM7/Multicore Manager -> Deep Copy ->
 Message send to CM4/Multicore Manager -> Message send to
 CM4/pVAlgoResMsg.taskId
```

2. isMulticoreMessage field set to 0
   A message constructed with isMulticoreMessage set to 0 is a unicast message sent only to the task specified in the `FWK_Message_Put`. If the task is Multicore, an additional **taskId** must be specified:

```
pAudioReqMsg->multicore.isMulticoreMessage = 0;
pAudioReqMsg->multicore.taskId             =
 kFWKTaskID_Output;
FWK_Message_Put(kFWKTaskID_Multicore, &pAudioReqMsg);
```

If the message has been sent by the CM7/Camera_Manager, the message is sent only to CM4/Output via Multicore Manager

```
FWK_Message_Put(kFWKTaskID_Multicore, &pAudioReqMsg);
└── Message send to CM7/Multicore Manager -> Deep Copy ->
 Message send to CM4/Multicore Manager -> Message send to
 CM4/pAudioReqMsg.taskId
```

When sending a message, a deep copy of the message is done by the Multicore Manager. The purpose of the deep copy is to avoid sending references from untouchable regions (for example, CM7 sending a reference that points to internal TCM memory that cannot be seen by CM4). Deep copy ensures that the messages are stored in a shared buffer, therefore the messages must be small.

If bigger buffers must be sent, they have to be in a shared memory area and passed by reference (camera buffers).

6.3.11.1 APIs

6.3.11.1.1 FWK_MulticoreManager_Init

```
/**
 * @brief Init internal structures for Multicore Manager
 * @return int Return 0 if the init process was successful
 */
int FWK_MulticoreManager_Init();
```

6.3.11.1.2 FWK_MulticoreManager_DeviceRegister

```
/**
 * @brief Register a Multicore device. Only one multicore
 device is supported. The dev needs to be registered before
```

```
 * FWK_MulticoreManager_Start is called
 * @param dev Pointer to a camera device structure
 * @return int Return 0 if registration was successful
 */
int FWK_MulticoreManager_DeviceRegister(multicore_dev_t *dev);
```

6.3.11.1.3 FWK_MulticoreManager_Start

```
/**
 * @brief Spawn Multicore manager task which will call init/
start for all registered multicore devices
 * @param taskPriority the priority of the Multicore manager
 task
 * @return int Return 0 if the starting process was successful
 */
int FWK_MulticoreManager_Start(int taskPriority);
```

6.3.11.1.4 FWK_MulticoreManager_Deinit

```
/**
 * @brief Deinit MulticoreManager
 * @return int Return 0 if the deinit process was successful
 */
int FWK_MulticoreManager_Deinit();
```

6.4 HAL devices

6.4.1 Overview

One of the most important steps in the creation of any embedded software project is peripheral integration. This step can often be one of the most time-intensive steps of the process. Additionally, peripheral drivers are often heavily tied to the specific platform those drivers were originally written for. It makes upgrading/moving to another platform difficult and costly.

The **Hardware Abstraction Layer (HAL)** component of the framework architecture was designed in direct response to these issues.

HAL devices are designed to be written "on top of" lower-level driver code, helping to increase code understandability by abstracting many of the underlying details. HAL devices can be reused across different projects and NXP platforms, increasing code reuse, which can help cut down on development time.

6.4.1.1 Device Registration

In order for a manager to communicate with a HAL device, that device must first be registered with its respective manager. Registration of each HAL device takes place at the beginning of application startup when `main()` calls the `APP_RegisterHalDevices()` function as shown below:

```
int main(void)
{
    /* Init board hardware. */
    APP_BoardInit();
    LOGD("[MAIN]:Started");
```

```
    /* init the framework*/
    APP_InitFramework();

    /* register the hal devices*/
    APP_RegisterHalDevices();

    /* start the framework*/
    APP_StartFramework();

    // start
    vTaskStartScheduler();

    while (1)
    {
        LOGD("#");
    }

    return 0;
}
```

To register a device to its manager, each HAL device implements a registration function that is called prior to starting the managers themselves. For example, the "register" function for the push button input device looks as follows:

```
int HAL_InputDev_PushButtons_Register()
{
    int error = 0;
    LOGD("input_dev_push_buttons_register");
    error =
 FWK_InputManager_DeviceRegister(&s_InputDev_PushButtons);
    return error;
}
```

As HAL devices do not have header `.h` files associated with them, the registration function for each device is exposed via the `board_define.h` file found inside the `boards` folder. To be registered on startup, each HAL device must be added to the `APP_RegisterHalDevices` function in the `board_hal_registration.c` file. The `board_hal_registration.c` file is also found in the `boards` folder.

6.4.1.2 Device Types

There are several different device types to encapsulate the various peripherals that a user may wish to incorporate into their project. These device types include:

- Input
- Output
- Camera
- Display
- VAlgo (Vision/Voice)

As well as a few others which are not listed here.

Each device type has specific methods and fields based on the unique characteristics of that device type. For example, the camera HAL device definition looks as follows:

```
/**
 * @brief Callback function to notify camera manager that one
 frame is dequeued
```

```
 * @param dev Device structure of the camera device calling
 this function
 * @param event id of the event that took place
 * @param param Parameters
 * @param fromISR True if this operation takes place in an irq,
 0 otherwise
 * @return 0 if the operation was successfully
 */
typedef int (*camera_dev_callback_t)(const camera_dev_t *dev,
 camera_event_t event, void *param, uint8_t fromISR);

/*! @brief Operation that needs to be implemented by a camera
 device */
typedef struct _camera_dev_operator
{
    /* initialize the dev */
    hal_camera_status_t (*init)(camera_dev_t *dev, int width,
 int height, camera_dev_callback_t callback, void *param);
    /* deinitialize the dev */
    hal_camera_status_t (*deinit)(camera_dev_t *dev);
    /* start the dev */
    hal_camera_status_t (*start)(const camera_dev_t *dev);
    /* enqueue a buffer to the dev */
    hal_camera_status_t (*enqueue)(const camera_dev_t *dev,
 void *data);
    /* dequeue a buffer from the dev */
    hal_camera_status_t (*dequeue)(const camera_dev_t *dev,
 void **data, pixel_format_t *format);
    /* postProcess a buffer from the dev */
    /*
     * Only do the minimum determination(data point and the
 format) of the frame in the dequeue.
     *
     * And split the CPU based post process(IR/Depth/...
 processing) to postProcess as they will eat CPU
     * which is critical for the whole system as camera manager
 is running with the highest priority.
     *
     * Camera manager will do the postProcess if there is a
 consumer of this frame.
     *
     * Note:
     * Camera manager will call multiple times of the
 posProcess of the same frame determinted by dequeue.
     * The HAL driver needs to guarantee the postProcess only
 do once for the first call.
     *
     */
    hal_camera_status_t (*postProcess)(const camera_dev_t *dev,
 void **data, pixel_format_t *format);
    /* input notify */
    hal_camera_status_t (*inputNotify)(const camera_dev_t *dev,
 void *data);
} camera_dev_operator_t;

/*! @brief Structure that characterize the camera device. */
typedef struct
{
    /* buffer resolution */
    int height;
```

```
    int width;
    int pitch;
    /* active rect */
    int left;
    int top;
    int right;
    int bottom;
    /* rotate degree */
    cw_rotate_degree_t rotate;
    /* flip */
    flip_mode_t flip;
    /* swap byte per two bytes */
    int swapByte;
} camera_dev_static_config_t;
```

In many ways, HAL devices can be thought of as similar to interfaces in C++ and other object-oriented languages.

6.4.1.3 Anatomy of a HAL device

HAL devices are made up of several components which can vary by device type. However, each HAL device regardless of type has at least 3 components:

- `id`
- `name`
- `operators`

The `id` field is a unique device identifier that is assigned by the device's manager when the device is first registered.

The `name` field is used to help identify the device during various function calls and when debugging.

The `operators` field is a struct that contains function pointers to each of the functions that the HAL device is required to implement. The operators a device is required to implement vary based on the device type.

A HAL device's definition is stored in a struct that gets passed to that device's respective manager when the device is registered. It gives the manager information about the device and allows the manager to call the device's operators when necessary.

6.4.1.3.1 Operators

Operators are functions that "operate" on the device itself and are used by the device's manager to control the device and/or augment its behavior. Operators are used for initializing, starting, and stopping devices, as well as serving many other functions depending on the device.

As mentioned previously, the operators a HAL device must implement varies based on device type. For example, input devices must implement an `init`, `deinit`, `start`, `stop`, and `inputNotify` function.

```
typedef struct
{
    /* initialize the dev */
    hal_input_status_t (*init)(input_dev_t *dev,
 input_dev_callback_t callback);
    /* deinitialize the dev */
    hal_input_status_t (*deinit)(const input_dev_t *dev);
```

```
    /* start the dev */
    hal_input_status_t (*start)(const input_dev_t *dev);
    /* stop the dev */
    hal_input_status_t (*stop)(const input_dev_t *dev);
    /* notify the input_dev */
    hal_input_status_t (*inputNotify)(const input_dev_t *dev,
 void *param);
} input_dev_operator_t;
```

Generally, each device regardless of type has at least a `start`, `stop`, `init`, and `deinit` function. Additionally, most devices also implement an `inputNotify` function that is used for event handling.

Note: *Failing to implement a function does not prevent the HAL device from being registered, but is likely to prevent certain functionality from working. For example, failing to provide an implementation for a HAL device's* ***start*** *function prevents its respective manager from starting that device.*

6.4.1.4 Configs

Note: *This section describes a feature which is being developed.*

Configs represent the individual, configurable attributes specific to a HAL device. The configs available for a device varies from device to device, but can be altered during runtime via user input or by other devices and can be saved to flash to retain the same value through power cycles.

For example, the HAL device for the IR/White LEDs may only have a "brightness" config, while a speaker device may have configs for "volume", "left/right balance", and so on.

Note: *Each device can have a maximum of* `MAXIMUM_CONFIGS_PER_DEVICE` *configs (see* `framework/inc/fwk_common.h`*).*

Each device config regardless of device type has the same fields:

- `name`
- `expectedValue`
- `description`
- `value`
- `get`
- `set`

6.4.1.4.1 Name

A string containing the name of the config. The string length must be less than `DEVICE_CONFIG_NAME_MAX_LENGTH`.

```
char name[DEVICE_CONFIG_NAME_MAX_LENGTH];
```

6.4.1.4.2 ExpectedValue

A string that provides a description of the valid values associated with the config. The length of the string must be less than `DEVICE_CONFIG_EXPECTED_VAL_MAX_LENGTH`.

```
char expectedValue[DEVICE_CONFIG_EXPECTED_VAL_MAX_LENGTH];
```

6.4.1.4.3 Description

A string that provides a description of the config. The length of the string should be less than `DEVICE_CONFIG_DESCRIPTION_MAX_LENGTH`.

```
char description[DEVICE_CONFIG_DESCRIPTION_MAX_LENGTH];
```

6.4.1.4.4 Value

An int that stores the internal value of the config. The `value` must be set using the `set` function and retrieved using the `get` function.

```
uint32_t value;
```

6.4.1.4.5 Get

A function that returns the `value` of the config.

```
status_t (*get)(char *valueToString);
```

6.4.1.4.6 Set

A function that sets the `value` of the config.

```
status_t (*set)(char *configName, uint32_t value);
```

6.4.2 Input devices

The `Input` HAL device provides an abstraction to implement various devices that may capture data in many different ways, and the data can represent many different things. The Input HAL device definition is designed to encapsulate everything from physical devices like push buttons, to "virtual" devices like a command-line interface using UART.

Input devices are used to acquire external input data and forward that data to other HAL devices via the Input Manager so that those devices can respond to that data accordingly. The Input Manager communicates to other devices within the framework using `inputNotify` event messages. For more information about events and event handling, see Events.

As with other device types, `Input` devices are controlled via their manager. The Input Manager is responsible for managing all registered input HAL devices, and invoking input device operators (`init`, `start`, `dequeue`, and so on) as necessary. Additionally, the Input Manager allows for multiple input devices to be registered and operate at once.

6.4.2.1 Device definition

The HAL device definition for `Input` devices can be found under `framework/hal_api/hal_input_dev.h` and is reproduced below:

```
/*! @brief Attributes of an input device */
typedef struct _input_dev
{
    /* unique id which is assigned by input manager during the
 registration */
    int id;
```

```
    /* name of the device */
    char name[DEVICE_NAME_MAX_LENGTH];
    /* operations */
    const input_dev_operator_t *ops;
    /* private capability */
    input_dev_private_capability_t cap;
} input_dev_t;
```

The device operators associated with input HAL devices are as shown below:

```
/*! @brief Operation that needs to be implemented by an input
 device */
typedef struct
{
    /* initialize the dev */
    hal_input_status_t (*init)(input_dev_t *dev,
 input_dev_callback_t callback);
    /* deinitialize the dev */
    hal_input_status_t (*deinit)(const input_dev_t *dev);
    /* start the dev */
    hal_input_status_t (*start)(const input_dev_t *dev);
    /* start the dev */
    hal_input_status_t (*stop)(const input_dev_t *dev);
    /* notify the input_dev */
    hal_input_status_t (*inputNotify)(const input_dev_t *dev,
 void *param);
} input_dev_operator_t;
```

The device capabilities associated with input HAL devices are as shown below:

```
typedef struct
{
    /* callback */
    input_dev_callback_t callback;
} input_dev_private_capability_t;
```

6.4.2.2 Operators

Operators are functions that "operate" on a HAL device itself. Operators are akin to "public methods" in object oriented-languages and are used by the Input Manager to set up, start, and so on, each of its registered input devices.

For more information about operators, see Section 6.4.1.3.1.

6.4.2.2.1 Init

```
/* initialize the dev */
hal_input_status_t (*init)(input_dev_t *dev,
 input_dev_callback_t callback);
```

Initialize the input device.

`Init` should initialize any hardware resources the input device requires (I/O ports, IRQs, and so on), turn on the hardware, and perform any other setup the device requires.

The callback function to the device's manager is typically installed as part of the `Init` function as well.

This operator will be called by the Input Manager when the Input Manager task first starts.

6.4.2.2.2 Deinit

```
/* deinitialize the dev */
hal_input_status_t (*deinit)(const input_dev_t *dev);
```

"Deinitialize" the input device.

`DeInit` should release any hardware resources the input device uses (I/O ports, IRQs, and so on), turn off the hardware, and perform any other shutdown the device requires.

This operator will be called by the Input Manager when the Input Manager task ends[1].

```
[1]The `DeInit` function generally will not be called under
 normal operation.
```

6.4.2.2.3 Start

```
/* start the dev */
hal_input_status_t (*start)(const input_dev_t *dev);
```

Start the input device.

The `Start` operator will be called in the initialization stage of the Input Manager's task after the call to the `Init` operator. The startup of the display sensor and interface should be implemented in this operator. This includes, for example, starting the interface and enabling the IRQ of the DMA used by the interface.

6.4.2.2.4 Stop

```
/* start the dev */
hal_input_status_t (*stop)(const input_dev_t *dev);
```

Stop the input device.

The `Stop` operator functions as the inverse of the `Start` function and is not called under normal operation.

6.4.2.2.5 InputNotify

```
/* notify the input_dev */
hal_input_status_t (*inputNotify)(const input_dev_t *dev, void
 *param);
```

Handle input events.

The `InputNotify` operator is called by the Input Manager whenever a `kFWKMessageID_InputNotify` message received by and forwarded from the Input Manager's message queue.

For more information regarding events and event handling, see Events.

6.4.2.3 Capabilities

```
typedef struct
{
    /* callback */
    input_dev_callback_t callback;
} input_dev_private_capability_t;
```

The `capabilities` struct is primarily used for storing a callback to communicate information from the device back to the Input Manager. This callback function is typically installed via a device's `init` operator.

6.4.2.3.1 callback

```
/**
 * @brief callback function to notify input manager with an
 async event
 * @param dev Device structure
 * @param eventId Id of the event that took place
 * @param receiverList List with managers that should be notify
 * @param event Pointer to a event structure.
 * @param size If size is 0 event should be in a persistent
 memory zone else the framework will allocate memory for the
 * object Note the message delivery might go slow if the size
 is too much.
 * @param fromISR True if this operation takes place in an irq,
 0 otherwise
 * @return 0 if the operation was successfully
 */
typedef int (*input_dev_callback_t)(const input_dev_t *dev,
                                    input_event_id_t eventId,
                                    unsigned int receiverList,
                                    input_event_t *event,
                                    unsigned int size,
                                    uint8_t fromISR);
```

Callback to the Input Manager.

The `capabilities` struct is primarily used for storing a callback to communicate information from the device back to the Input Manager.

The Vision Algorithm manager provides the callback to the device when the `init` operator is called. As a result, the HAL device should make sure to store the callback in the `init` operator's implementation.

```
static hal_input_status_t
 HAL_InputDev_PushButtons_Init(input_dev_t *dev,
 input_dev_callback_t callback)
{
    hal_input_status_t error = 0;

    /* PERFORM INIT FUNCTIONALITY HERE */

    /* Installing callback function from manager... */
    memset(&dev->cap, 0, sizeof(dev->cap));
    dev->cap.callback = callback;

    return ret;
```

```
}
```

The HAL device invokes this callback to notify the vision algorithm manager of specific events.

The definition for `valgo_dev_callback_t` is as shown below:

```
typedef int (*input_dev_callback_t)(const input_dev_t *dev,
                                    input_event_id_t eventId,
                                    unsigned int receiverList,
                                    input_event_t *event,
                                    unsigned int size,
                                    uint8_t fromISR);
```

The fields passed as part of the callback are described in more detail below.

6.4.2.3.2 EventId

```
typedef enum _input_event_id
{
    kInputEventID_Recv,
    kInputEventID_AudioRecv,
    kInputEventID_FrameworkRecv,
} input_event_id_t;
```

Describes the type of source event being sent/received.

6.4.2.3.3 ReceiverList

```
typedef enum _fwk_task_id
{
    kFWKTaskID_Camera = 0, /* This should always stay first */
    kFWKTaskID_Display,
    kFWKTaskID_VisionAlgo,
    kFWKTaskID_VoiceAlgo,
    kFWKTaskID_Output,
    kFWKTaskID_Input,
    kFWKTaskID_Audio,
    kFWKTaskID_APPStart, /* APP task ID should always start
 from here */
    kFWKTaskID_COUNT = (kFWKTaskID_APPStart + APP_TASK_COUNT)
} fwk_task_id_t;
```

List of device managers meant to receive the input event message.

6.4.2.3.4 Event

```
typedef struct _input_event
{
    union
    {
        /* Valid when message is kInputEventID_RECV */
        void *inputData;

        /* Valid when eventId is kInputEventID_AudioRECV */
        void *audioData;
```

```
        /* Valid when framework information is needed
 GET_FRAMEWORK_INFO*/
        framework_request_t *frameworkRequest;
    };
} input_event_t;
```

6.4.2.4 Example

The project has several input devices implemented for use as-is or for use as reference for implementing new input devices. Source files for these input HAL devices can be found under `HAL/common/` and `HAL/face_rec`.

Below is an example of a push button input HAL device driver:

```
static input_event_t inputEvent;

const static input_dev_operator_t s_InputDev_ExampleDevOps = {
    .init        = HAL_InputDev_ExampleDev_Init,
    .deinit      = HAL_InputDev_ExampleDev_Deinit,
    .start       = HAL_InputDev_ExampleDev_Start,
    .stop        = HAL_InputDev_ExampleDev_Stop,
    .inputNotify = HAL_InputDev_ExampleDev_InputNotify,
};

static input_dev_t s_InputDev_ExampleDev = {
    .name = "buttons",
    .ops = &s_InputDev_ExampleDevOps,
    .cap = {
        .callback = NULL
    },
};

/* here assume buttons push event will call this handler */
void HAL_InputDev_ExampleDev_EvtHandler(void)
{
    /* Add manager task list need notify, the id is from
 fwk_task_id_t.
     * Note: here can set not only one task manager.
     */
    receiverList = 1 << kFWKTaskID_Display;

    /* load input data */
    inputEvent.inputData = NULL;

    /* callback inputmanager notify the corresponding manager
 from receiverList */
    inputDev.cap.callback(&inputDev, kInputEventID_Recv,
 receiverList, &inputEvent, 0, fromISR);
}

hal_input_status_t HAL_InputDev_ExampleDev_Init(input_dev_t
 *dev, input_dev_callback_t callback)
{
    hal_input_status_t ret = kStatus_HAL_InputSuccess;

    /* install manager callback for device */
    dev->cap.callback = callback;

    /* put hardware init here */
```

```
    return ret;
}

hal_input_status_t HAL_InputDev_ExampleDev_Deinit(const
 input_dev_t *dev)
{
    hal_input_status_t ret = kStatus_HAL_InputSuccess;

    /* put device deinit here */

    return ret;
}

hal_input_status_t HAL_InputDev_ExampleDev_Start(const
 input_dev_t *dev)
{
    hal_input_status_t ret = kStatus_HAL_InputSuccess;

    /* put device start here */

    return ret;
}

hal_input_status_t HAL_InputDev_ExampleDev_Stop(const
 input_dev_t *dev)
{
    hal_input_status_t ret = kStatus_HAL_InputSuccess;

    /* put device stop here */

    return ret;
}

hal_input_status_t HAL_InputDev_ExampleDev_InputNotify(const
 input_dev_t *dev, void *param)
{
    hal_input_status_t ret = kStatus_HAL_InputSuccess;

    /* add device notify handler here */

    return ret;
}

int HAL_InputDev_ExampleDev_Register(void)
{
    int ret = 0;
    ret =
 FWK_InputManager_DeviceRegister(&s_InputDev_ExampleDev);
    return ret;
}
```

6.4.3 Output devices

The Output HAL devices are used to represent any device that produces output (excluding specific devices that have their own specific device types like cameras and displays).

The Output devices respond to events passed by other HAL devices and produce corresponding output. It includes changing the UI overlay in response to a "face recognized" event or changing the volume of the speaker in response to a specific shell command.

Multiple output devices can be registered at a time per the design of the framework.

6.4.3.1 Subtypes

Currently, output devices can be divided into 3 "subtypes" to better represent the specific nuances of a wider variety of output devices without creating entirely new HAL device types:

- "General" output devices
- "Overlay/UI" output devices
- "Audio" output devices

6.4.3.1.1 General devices

"General"/generic output devices describe most output devices and include devices like LEDs.

6.4.3.1.2 UI devices

Overlay/UI output devices are used for output devices that act as an overlay that sits on top of a camera preview surface.

Overlay/UI devices require that a frame buffer be allocated when initializing a device of this subtype.

6.4.3.1.3 Audio devices

Audio output HAL devices represent devices that act as recipients of audio data. Audio output HAL devices typically process audio data so that they can play a sound in response to an event like a face being registered, or sleep mode triggering.

6.4.3.2 Device definition

The HAL device definition for output devices can be found under `framework/hal_api/hal_output_dev.h` and is reproduced below:

```
/*! @brief definition of an output device */
typedef struct _output_dev
{
    /* unique id and assigned by Output Manager when this
 device register */
    int id;
    /* device name */
    char name[DEVICE_NAME_MAX_LENGTH];
    /* attributes */
    output_dev_attr_t attr;
    /* optional config for private configuration of special
 output device */
    hal_device_config configs[MAXIMUM_CONFIGS_PER_DEVICE];

    /* operations */
    const output_dev_operator_t *ops;
}output_dev_t;
```

The operators associated with output HAL devices are as shown below:

```
/*! @brief Operation that needs to be implemented by an output
 device */
typedef struct _output_dev_operator
{
    /* initialize the dev */
    hal_output_status_t (*init)(const output_dev_t *dev);
    /* deinitialize the dev */
    hal_output_status_t (*deinit)(const output_dev_t *dev);
    /* start the dev */
    hal_output_status_t (*start)(const output_dev_t *dev);
    /* stop the dev */
    hal_output_status_t (*stop)(const output_dev_t *dev);

} output_dev_operator_t;
```

The device attributes associated with output HAL devices are as shown below:

```
/*! @brief Attributes of an output device */
typedef struct _output_dev_attr_t
{
    /* the type of output device */
    output_dev_type_t type;
    union
    {
        /* if the type of output device is OverlayUI, it need
 to allocate overlay surface */
        gfx_surface_t *pSurface;
        /* reserve for other type of output device*/
        void *reserve;
    };
} output_dev_attr_t;
```

6.4.3.3 Operators

Operators are functions that "operate" on a HAL device itself. Operators are akin to "public methods" in object-oriented languages and are used by the Output Manager to set up, start, and so on, each of its registered output devices.

For more information about operators, see Section 6.4.1.3.1.

6.4.3.3.1 Init

```
hal_output_status_t (*init)(const output_dev_t *dev);
```

The `Init` function is used to initialize the output device, `Init` should initialize any hardware resources the output device requires (I/O ports, IRQs, and so on), turn on the hardware, and perform any other setup the device requires.

This operator will be called by the Output Manager when the Output Manager task first starts.

6.4.3.3.2 DeInit

```
hal_output_status_t (*deinit)(const output_dev_t *dev);
```

The `DeInit` function is used to initialize the output device, `DeInit` should release any hardware resources the output device uses (I/O ports, IRQs, and so on), turn off the hardware, and perform any other shutdown the device requires.

This operator will be called by the Output Manager when the Output Manager task ends[1].

```
[1]The `DeInit` function generally will not be called under
 normal operation.
```

6.4.3.3.3 Start

```
hal_output_status_t (*start)(const output_dev_t *dev);
```

Starts the output device. The `Start` method will usually call `FWK_OutputManager_RegisterEventHandler` to register event handlers with the Output Manager so that when the Output Manager receives an output event (like an "inference complete" event or an "input notify" event), the corresponding event handler function is executed.

This operator is called by the Output Manager when the Output Manager task first starts.

6.4.3.3.4 Stop

```
hal_output_status_t (*stop)(const output_dev_t *dev);
```

Stops the output device. The `Stop` method will usually call `FWK_OutputManager_UnRegisterEventHandler` to unregister an event handler from the Output Manager. It prevents the device's event handlers from executing when an event is triggered.

6.4.3.4 Attributes

6.4.3.4.1 Type

The type of output device. If the type is `kOutputDevType_UI`, the `pSurface` parameter must be set. Otherwise, `pSurface` can safely be ignored.

```
output_dev_type_t type;
```

The type struct is shown below:

```
/*! @brief Types of output devices */
typedef enum _output_dev_type
{
    kOutputDevType_UI,     /* for Overlay UI */
    kOutputDevType_Audio,  /* for Audio output */
    kOutputDevType_Other,  /* for other general output, like
 LED, Console, etc */
} output_dev_type_t;
```

6.4.3.4.2 pSurface

The `pSurface` variable is used by `Overlay/UI` output devices to hold a frame buffer.

If the device type "subtype" is not a `kOuptutDevType_UI` device, then this parameter can be safely ignored.

```
gfx_surface_t * pSurface;
```

The `gfx_surface` struct is shown below:

```
typedef struct _gfx_surface
{
    int height;  /* the height of surface */
    int width;   /* the width of surface */
    int pitch;   /* the pitch of surface */
    int left;    /* the left coordinate of surface */
    int top;     /* the top coordinate of surface */
    int right;   /* the right coordinate of surface */
    int bottom;  /* the bottom coordinate of surface */
    int swapByte; /* For each 16 bit word of surface
 framebuffer, set true to swap the two bytes. */
   pixel_format_t format; /* the pixel format of surface, like
 kPixelFormat_RGB565 */
   void *buf;  /* the pointer for the framebuffer */
   void *lock; /* the mutex lock for the surface, is
 determined by hal and set to null if not use in hal*/
} gfx_surface_t;
```

6.4.3.5 Example

The project has several output devices implemented for use as-is or for use as a reference for implementing new output devices. Source files for these output HAL devices can be found under `HAL/common/`.

Below is an example of the RGB LED HAL device driver `HAL/common/hal_output_rgb_led.c`:

```
static hal_output_status_t
 HAL_OutputDev_RgbLed_Init(output_dev_t *dev);
static hal_output_status_t HAL_OutputDev_RgbLed_Start(const
 output_dev_t *dev);
static hal_output_status_t
 HAL_OutputDev_RgbLed_InferComplete(const output_dev_t *dev,

 output_algo_source_t source,

 void *inferResult);

const static output_dev_event_handler_t
 s_OutputDev_RgbLedHandler = {
    .inferenceComplete = HAL_OutputDev_RgbLed_InferComplete,
    .inputNotify       = NULL,
};

/* output device operators*/
const static output_dev_operator_t s_OutputDev_RgbLedOps = {
    .init   = HAL_OutputDev_RgbLed_Init,
    .deinit = NULL,
    .start  = HAL_OutputDev_RgbLed_Start,
    .stop   = NULL,
};
```

```
/* output device */
static output_dev_t s_OutputDev_RgbLed = {
    .name         = "rgb_led",
    .attr.type    = kOutputDevType_Other,
    .attr.reserve = NULL,
    .ops          = &s_OutputDev_RgbLedOps,
};

/* RGB LED output device Init function*/
static hal_output_status_t
 HAL_OutputDev_RgbLed_Init(output_dev_t *dev)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;
    /* put RGB LED hardware initialization here*/
    ...
    return error;
}

/* RGB LED output device start function*/
static hal_output_status_t HAL_OutputDev_RgbLed_Start(const
 output_dev_t *dev)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;
    /* registered special event handler for this output device
 */
    if (FWK_OutputManager_RegisterEventHandler(dev,
 &s_OutputDev_RgbLedHandler) != 0)
    {
        error = kStatus_HAL_OutputError;
    }
    return error;
}

static hal_output_status_t
 HAL_OutputDev_RgbLed_InferComplete(const output_dev_t *dev,

 output_algo_source_t source,

 void *inferResult)
{
    hal_output_status_t error   = kStatus_HAL_OutputSuccess;
    /* algorithm_result_t is defined by special algorithm
 device registered into vision pipeline */
    algorithm_result_t *result = (algorithm_result_t
 *)inferResult;
    if (pResult != NULL)
    {
        /* do RGB LED hardware setting according to inference
 result from valgorithm manager*/
        ...
    }
    return error;
}

int HAL_OutputDev_RgbLed_Register()
{
    int error = 0;
    LOGD("output_dev_rgb_led_register");
```

```
    error =
 FWK_OutputManager_DeviceRegister(&s_OutputDev_RgbLed);
    return error;
}
```

An example of an Overlay UI Output device can be found at `HAL/face_rec/hal_smart_lock_ui.c`.

```
static hal_output_status_t HAL_OutputDev_OverlayUi_Init(const
 output_dev_t *dev);
static hal_output_status_t HAL_OutputDev_OverlayUi_Start(const
 output_dev_t *dev);
static hal_output_status_t
 HAL_OutputDev_OverlayUi_InferComplete(const output_dev_t *dev,

  output_algo_source_t source,

  void *infer_result);
static hal_output_status_t
 HAL_OutputDev_OverlayUi_InputNotify(const output_dev_t *dev,
 void *data);

/* Overlay UI surface */
static gfx_surface_t s_UiSurface;
/* the framebuffer for Overlay UI surface */
SDK_ALIGN(static char s_AsBuffer[UI_BUFFER_WIDTH *
 UI_BUFFER_HEIGHT * UI_BUFFER_BPP], 32);
/* event handler */
const static output_dev_event_handler_t s_OutputDev_UiHandler =
 {
    .inferenceComplete = HAL_OutputDev_OverlayUi_InferComplete,
    .inputNotify       = HAL_OutputDev_OverlayUi_InputNotify,
};

/* output device operators */
const static output_dev_operator_t s_OutputDev_UiOps = {
    .init   = HAL_OutputDev_OverlayUi_Init,
    .deinit = NULL,
    .start  = HAL_OutputDev_OverlayUi_Start,
    .stop   = NULL,
};

/* output device */
static output_dev_t s_OutputDev_Ui = {
    .name         = "ui",
    .attr.type    = kOutputDevType_UI,
    .attr.pSurface = &s_UiSurface,
    .ops          = &s_OutputDev_UiOps,
};

/* Overlay UI output device Init function*/
static hal_output_status_t
 HAL_OutputDev_OverlayUi_Init(output_dev_t *dev)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;
    /* init overlay ui surface */
    s_UiSurface.left   = 0;
    s_UiSurface.top    = 0;
    s_UiSurface.right  = UI_BUFFER_WIDTH - 1;
    s_UiSurface.bottom = UI_BUFFER_HEIGHT - 1;
```

```
    s_UiSurface.height = UI_BUFFER_HEIGHT;
    s_UiSurface.width  = UI_BUFFER_WIDTH;
    s_UiSurface.pitch  = UI_BUFFER_WIDTH * 2;
    s_UiSurface.format = kPixelFormat_RGB565;
    s_UiSurface.buf    = s_AsBuffer;
    s_UiSurface.lock   = xSemaphoreCreateMutex();

    return error;
}

/* Overlay UI output device start function*/
static hal_output_status_t HAL_OutputDev_OverlayUi_Start(const
 output_dev_t *dev)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;
    /* registered special event handler for this output device
 */
    if (FWK_OutputManager_RegisterEventHandler(dev,
 &s_OutputDev_UiHandler) != 0)
        error = kStatus_HAL_OutputError;
    return error;
}

/* Overlay UI inferenceComplete event handler function*/
static hal_output_status_t
 HAL_OutputDev_OverlayUi_InferComplete(const output_dev_t *dev,

  output_algo_source_t source,

  void *infer_result)
{
    hal_output_status_t error    = kStatus_HAL_OutputSuccess;
    /* algorithm_result_t is defined by special algorithm
 device registered into vision pipeline */
    algorithm_result_t *pResult = (algorithm_result_t
 *)infer_result;

    if (pResult != NULL)
    {
        /* lock overlay surface to avoid conflict with PXP
 composing overlay surface */
        if (s_UiSurface.lock)
        {
            xSemaphoreTake(s_UiSurface.lock, portMAX_DELAY);
        }

        /* draw overlay surface here according to inference
 result from valgorithm manager */
        ...

        /* unlock */
        if (s_UiSurface.lock)
        {
            xSemaphoreGive(s_UiSurface.lock);
        }
    }
    return error;
}

/* Overlay UI inputNotify event handler function*/
```

```
static hal_output_status_t
 HAL_OutputDev_OverlayUi_InputNotify(const output_dev_t *dev,
 void *data)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;
    event_base_t eventBase    = *(event_base_t *)data;

    if (eventBase != NULL)
    {
        /* lock overlay surface to avoid conflict with PXP
 composing overlay surface */
        if (s_UiSurface.lock)
        {
            xSemaphoreTake(s_UiSurface.lock, portMAX_DELAY);
        }

        /* draw overlay surface here according to input notify
 event from input manager*/
        ...

        /* unlock */
        if (s_UiSurface.lock)
        {
            xSemaphoreGive(s_UiSurface.lock);
        }
    }
    return error;
}

int HAL_OutputDev_UiSmartlock_Register()
{
    int error = 0;
    LOGD("output_dev_ui_smartlock_register");
    error = FWK_OutputManager_DeviceRegister(&s_OutputDev_Ui);
    return error;
}
```

6.4.4 Camera devices

The `Camera` HAL device provides an abstraction to represent many different camera devices which may have different resolutions, color formats, and even connection interfaces.

For example, the same GC0308 RGB camera can connect with CSI or via a FlexIO interface.

```
A camera HAL device represents a camera sensor + interface,
meaning a separate device driver is required for the same
 camera sensor using different interfaces.
```

As with other device types, camera devices are controlled via their manager. The Camera Manager is responsible for managing all registered camera HAL devices, and invoking camera device operators (`init`, `start`, `dequeue`, and so on) as necessary. Additionally, the Camera Manager allows for multiple camera devices to be registered and operated at once.

6.4.4.1 Device definition

The HAL device definition for `Camera` devices can be found under `framework/hal_api/hal_camera_dev.h` and is reproduced below:

```
typedef struct _camera_dev camera_dev_t;
/*! @brief Attributes of a camera device. */
struct _camera_dev
{
    /* unique id which is assigned by camera manager during
 registration */
    int id;
    /* state in which the device is found */
    hal_device_state_t state;
    /* name of the device */
    char name[DEVICE_NAME_MAX_LENGTH];

    /* operations */
    const camera_dev_operator_t *ops;
    /* static configs */
    camera_dev_static_config_t config;
    /* private capability */
    camera_dev_private_capability_t cap;
};
```

The device operators associated with camera HAL devices are as shown below:

```
/*! @brief Operation that needs to be implemented by a camera
 device */
typedef struct _camera_dev_operator
{
    /* initialize the dev */
    hal_camera_status_t (*init)(camera_dev_t *dev, int width,
 int height, camera_dev_callback_t callback, void *param);
    /* deinitialize the dev */
    hal_camera_status_t (*deinit)(camera_dev_t *dev);
    /* start the dev */
    hal_camera_status_t (*start)(const camera_dev_t *dev);
    /* enqueue a buffer to the dev */
    hal_camera_status_t (*enqueue)(const camera_dev_t *dev,
 void *data);
    /* dequeue a buffer from the dev */
    hal_camera_status_t (*dequeue)(const camera_dev_t *dev,
 void **data, pixel_format_t *format);
    /* postProcess a buffer from the dev */
    /*
     * Only do the minimum determination(data point and the
 format) of the frame in the dequeue.
     *
     * And split the CPU based post process(IR/Depth/...
 processing) to postProcess as they will eat CPU
     * which is critical for the whole system as Camera Manager
 is running with the highest priority.
     *
     * Camera Manager will do the postProcess if there is a
 consumer of this frame.
     *
     * Note:
```

```
     * Camera Manager will call multiple times of the
 posProcess of the same frame determinted by dequeue.
     * The HAL driver needs to guarantee the postProcess only
 do once for the first call.
     *
     */
    hal_camera_status_t (*postProcess)(const camera_dev_t *dev,
 void **data, pixel_format_t *format);
    /* input notify */
    hal_camera_status_t (*inputNotify)(const camera_dev_t *dev,
 void *data);
} camera_dev_operator_t;
```

The static configs associated with camera HAL devices are as shown below:

```
/*! @brief Structure that characterize the camera device. */
typedef struct
{
    /* buffer resolution */
    int height;
    int width;
    int pitch;
    /* active rect */
    int left;
    int top;
    int right;
    int bottom;
    /* rotate degree */
    cw_rotate_degree_t rotate;
    /* flip */
    flip_mode_t flip;
    /* swap byte per two bytes */
    int swapByte;
} camera_dev_static_config_t;
```

The device capabilities associated with camera HAL devices are as shown below:

```
/*! @brief Structure that capability of the camera device. */
typedef struct
{
    /* callback */
    camera_dev_callback_t callback;
    /* param for the callback */
    void *param;
} camera_dev_private_capability_t;
```

6.4.4.2 Operators

Operators are functions that "operate" on a HAL device itself. Operators are akin to "public methods" in object-oriented languages, and are used by the Camera Manager to set up, start, and so on, each of its registered camera devices.

For more information about operators, see Section 6.4.1.3.1.

6.4.4.2.1 Init

```
hal_camera_status_t (*init)(camera_dev_t *dev,
                            int width,
```

```
                                int height,
                                camera_dev_callback_t callback,
                                void *param);
```

Initialize the camera device.

`Init` should initialize any hardware resources the camera device requires (I/O ports, IRQs, and so on), turn on the hardware, and perform any other setup the device requires.

This operator is called by the Camera Manager when the Camera Manager task first starts.

6.4.4.2.2 Deinit

```
hal_camera_status_t (*deinit)(camera_dev_t *dev);
```

"Deinitialize" the camera device.

`DeInit` must release any hardware resources the camera device uses (I/O ports, IRQs, and so on), turn off the hardware, and perform any other shutdown the device requires.

This operator will be called by the Camera Manager when the Camera Manager task ends[1].

```
[1]The `DeInit` function generally will not be called under
 normal operation.
```

6.4.4.2.3 Start

```
hal_camera_status_t (*start)(const camera_dev_t *dev);
```

Start the camera device.

The `Start` operator will be called in the initialization stage of the Camera Manager's task after the call to the `Init` operator. The startup of the camera sensor and interface should be implemented in this operator. It includes, for example, starting the interface and enabling the IRQ of the DMA used by the interface.

6.4.4.2.4 Enqueue

```
hal_camera_status_t (*enqueue)(const camera_dev_t *dev,
                                void *data);
```

Enqueue a single frame.

The `Enqueue` operator is called by the Camera Manager to submit an empty buffer into the camera device's buffer queue. Once the submitted buffer is filled by the camera device, the camera device should call the Camera Manager's callback function and pass a `kCameraEvent_SendFrame` event.

6.4.4.2.5 Dequeue

```
hal_camera_status_t (*enqueue)(const camera_dev_t *dev,
                                void *data);
```

Dequeue a single frame.

The `Dequeue` operator will be called by the Camera Manager to get a camera frame from the device. The frame address and the format will be determined by this operator.

6.4.4.2.6 PostProcess

```
hal_camera_status_t (*postProcess)(const camera_dev_t *dev,
                                   void **data,
                                   pixel_format_t *format);
```

Handles the post-processing of the camera frame.

The `PostProcess` operator is called by the Camera Manager to perform any required post-processing of the camera frame. For example, if a frame must be converted from one format to another in some way before it is useable by the display and/or a vision algo device, it would take place in the `PostProcess` operator.

6.4.4.2.7 InputNotify

```
hal_camera_status_t (*inputNotify)(const camera_dev_t *dev,
 void *data);
```

Handle input events.

The `InputNotify` operator is called by the Camera Manager whenever a `kFWKMessageID_InputNotify` message is received by and forwarded from the Camera Manager's message queue.

For more information regarding events and event handling, see Events.

6.4.4.3 Static configs

Static configs, unlike regular, dynamic configs, are set at compile time and cannot be changed on-the-fly.

6.4.4.3.1 Height

```
int height;
```

The height of the camera buffer.

6.4.4.3.2 Width

```
int width;
```

The width of the camera buffer.

6.4.4.3.3 Pitch

```
int pitch;
```

The total number of bytes in a single row of a camera frame.

6.4.4.3.4 Left

```
int left;
```

The left edge of the active area in a camera buffer.

6.4.4.3.5 Top

```
int top;
```

The top edge of the active area in a camera buffer.

6.4.4.3.6 Right

```
int right;
```

The right edge of the active area in a camera buffer.

6.4.4.3.7 Bottom

```
int bottom;
```

The bottom edge of the active area in a camera buffer.

6.4.4.3.8 Rotate

```
typedef enum _cw_rotate_degree
{
    kCWRotateDegree_0 = 0,
    kCWRotateDegree_90,
    kCWRotateDegree_180,
    kCWRotateDegree_270
} cw_rotate_degree_t;
```

```
cw_rotate_degree_t rotate;
```

The rotate degree of the camera sensor.

6.4.4.3.9 Flip

```
typedef enum _flip_mode
{
    kFlipMode_None = 0,
    kFlipMode_Horizontal,
    kFlipMode_Vertical,
    kFlipMode_Both
} flip_mode_t;
```

```
flip_mode_t flip;
```

Determines whether to flip the frame while processing the frame for the algorithm and display.

6.4.4.3.10 SwapByte

```
int swapByte;
```

Determines whether to enable swapping bytes while processing a frame for algorithm and display devices.

6.4.4.4 Capabilities

```
typedef struct
{
    /* callback */
    camera_dev_callback_t callback;
    /* param for the callback */
    void *param;
} camera_dev_private_capability_t;
```

The `capabilities` struct is primarily used for storing a callback to communicate information from the device back to the Camera Manager. This callback function is typically installed via a device's `init` operator.

6.4.4.4.1 Callback

```
/**
* @brief Callback function to notify Camera Manager that one
 frame is dequeued
* @param dev Device structure of the camera device calling this
 function
* @param event id of the event that took place
* @param param Parameters
* @param fromISR True if this operation takes place in an irq,
 0 otherwise
* @return 0 if the operation was successfully
*/
typedef int (*camera_dev_callback_t)(const camera_dev_t *dev,
                                     camera_event_t event,
                                     void *param,
                                     uint8_t fromISR);
```

```
camera_dev_callback_t callback;
```

Callback to the Camera Manager.

The HAL device invokes this callback to notify the Camera Manager of specific events like "frame dequeued."

The Camera Manager provides this callback to the device when the `init` operator is called. As a result, the HAL device should make sure to store the callback in the `init` operator's implementation.

```
static hal_camera_status_t HAL_CameraDev_ExampleDev_Init(
    camera_dev_t *dev, int width, int height,
 camera_dev_callback_t callback, void *param)
{
    hal_camera_status_t ret = kStatus_HAL_CameraSuccess;
```

```
    /* PERFORM INIT FUNCTIONALITY HERE */

    ...

    /* Installing callback function from manager... */
    dev->cap.callback  = callback;

    return ret;
}
```

6.4.4.4.2 Param

```
void *param;
```

The parameter of the callback for `kCameraEvent_SendFrame` event. The Camera Manager provides the parameter while calling the `Init` operator, so this param should be stored in the HAL device's struct as part of the implementation of the `Init` operator.

```
This param should be provided when calling the [`Callback`]
(#callback) function.
```

6.4.4.5 Example

The project has several camera devices implemented for use as-is or for use as reference for implementing new camera devices. Source files for these camera HAL devices can be found under `HAL/common/`.

Below is an example of the GC0308 RGB FlexIO camera HAL device driver `HAL/common/hal_camera_flexio_gc0308.c`.

```
hal_camera_status_t HAL_CameraDev_FlexioGc0308_Init(
    camera_dev_t *dev, int width, int height,
 camera_dev_callback_t callback, void *param);
static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Deinit(camera_dev_t *dev);
static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Start(const camera_dev_t *dev);
static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Enqueue(const camera_dev_t *dev,
 void *data);
static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Dequeue(const camera_dev_t *dev,

 void **data,

 pixel_format_t *format);
static int HAL_CameraDev_FlexioGc0308_Notify(const camera_dev_t
 *dev, void *data);

/* The operators of the FlexioGc0308 Camera HAL Device */
const static camera_dev_operator_t s_CameraDev_FlexioGc0308Ops
 = {
    .init        = HAL_CameraDev_FlexioGc0308_Init,
    .deinit      = HAL_CameraDev_FlexioGc0308_Deinit,
    .start       = HAL_CameraDev_FlexioGc0308_Start,
    .enqueue     = HAL_CameraDev_FlexioGc0308_Enqueue,
    .dequeue     = HAL_CameraDev_FlexioGc0308_Dequeue,
```

```
    .inputNotify = HAL_CameraDev_FlexioGc0308_Notify,
};

/* FlexioGc0308 Camera HAL Device */
static camera_dev_t s_CameraDev_FlexioGc0308 = {
    .id   = 0,
    .name = CAMERA_NAME,
    .ops  = &s_CameraDev_FlexioGc0308Ops,
    .cap =
        {
            .callback = NULL,
            .param    = NULL,
        },
};

hal_camera_status_t HAL_CameraDev_FlexioGc0308_Init(
    camera_dev_t *dev, int width, int height,
 camera_dev_callback_t callback, void *param)
{
    hal_camera_status_t ret = kStatus_HAL_CameraSuccess;
    LOGD("camera_dev_flexio_gc0308_init");

    /* store the callback and param for late using*/
    dev->cap.callback  = callback;
    dev->cap.param     = param;

    /* init the low level camera sensor and interface */

    return ret;
}

static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Deinit(camera_dev_t *dev)
{
    hal_camera_status_t ret = kStatus_HAL_CameraSuccess;
    /* Currently do nothing for the Deinit as we didn't support
 the runtime de-registraion of the device */
    return ret;
}

static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Start(const camera_dev_t *dev)
{
    hal_camera_status_t ret = kStatus_HAL_CameraSuccess;

    /* start the low level camera sensor and interface */

    return ret;
}

static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Enqueue(const camera_dev_t *dev,
 void *data)
{
    hal_camera_status_t ret = kStatus_HAL_CameraSuccess;

    /* submit one free buffer into the camera's buffer queue */

    return ret;
}
```

```
static hal_camera_status_t
 HAL_CameraDev_FlexioGc0308_Dequeue(const camera_dev_t *dev,

 void **data,

 pixel_format_t *format)
{
    hal_camera_status_t ret = kStatus_HAL_CameraSuccess;

    /* get the buffer from camera's buffer queue and determine
 the format of the frame */

    return ret;
}

static int HAL_CameraDev_FlexioGc0308_Notify(const camera_dev_t
 *dev, void *data)
{
    int error              = 0;
    event_base_t eventBase = *(event_base_t *)data;

    /* handle the events which are interested in */
    switch (eventBase.eventId)
    {
        default:
            break;
    }

    return error;
}
```

6.4.5 Display devices

The `Display` HAL device provides an abstraction to represent many different display panels which may have different controllers, resolutions, color formats, and event connection interfaces.

Note: *A display HAL device represents a display panel + interface. For example, the `hal_display_lcdif_rk024hh298.c` is the display HAL device driver for the rk024hh298 panel with eLCDIF interface. It means a separate device driver is required for the same display using different interfaces.*

As with other device types, display devices are controlled via their manager. The Display Manager is responsible for managing all registered display HAL devices, and invoking display device operators (`init`, `start`, and so on) as necessary.

6.4.5.1 Device definition

The HAL device definition for display devices can be found under `framework/hal_api/hal_display_dev.h` and is reproduced below:

```
typedef struct _display_dev display_dev_t;
/*! @brief Attributes of a display device. */
struct _display_dev
{
    /* unique id which is assigned by Display Manager during
 the registration */
```

```
    int id;
    /* name of the device */
    char name[DEVICE_NAME_MAX_LENGTH];
    /* operations */
    const display_dev_operator_t *ops;
    /* private capability */
    display_dev_private_capability_t cap;
};
```

The operators associated with display HAL devices are as shown below:

```
/*! @brief Operation that needs to be implemented by a display
 device */
typedef struct _display_dev_operator
{
    /* initialize the dev */
    hal_display_status_t (*init)(
        display_dev_t *dev,
        int width, int height,
        display_dev_callback_t callback,
        void *param);
    /* deinitialize the dev */
    hal_display_status_t (*deinit)(const display_dev_t *dev);
    /* start the dev */
    hal_display_status_t (*start)(const display_dev_t *dev);
    /* blit a buffer to the dev */
    hal_display_status_t (*blit)(const display_dev_t *dev,
                                 void *frame,
                                 int width,
                                 int height);
    /* input notify */
    hal_display_status_t (*inputNotify)(const display_dev_t
 *dev, void *data);
} display_dev_operator_t;
```

The capabilities associated with display HAL devices are as shown below:

```
/*! @brief Structure that characterize the display device. */
typedef struct _display_dev_private_capability
{
    /* buffer resolution */
    int height;
    int width;
    int pitch;
    /* active rect */
    int left;
    int top;
    int right;
    int bottom;
    /* rotate degree */
    cw_rotate_degree_t rotate;
    /* pixel format */
    pixel_format_t format;
    /* the source pixel format of the requested frame */
    pixel_format_t srcFormat;
    void *frameBuffer;
    /* callback */
    display_dev_callback_t callback;
    /* param for the callback */
```

```
    void *param;
} display_dev_private_capability_t;
```

6.4.5.2 Operators

Operators are functions which "operate" on a HAL device itself. Operators are akin to "public methods" in object oriented-languages, and are used by the Display Manager to set up, start, and so on, each of its registered display devices.

For more information about operators, see Section 6.4.1.3.1.

6.4.5.2.1 Init

```
hal_display_status_t (*init)(display_dev_t *dev,
                             int width,
                             int height,
                             display_dev_callback_t callback,
                             void *param);
```

Initialize the display device.

`Init` should initialize any hardware resources the display device requires (I/O ports, IRQs, and so on), turn on the hardware, and perform any other setup the device requires.

The callback function to the device's manager is typically installed as part of the `Init` function as well.

This operator will be called by the Display Manager when the Display Manager task first starts.

6.4.5.2.2 Deinit

```
hal_display_status_t (*deinit)(const display_dev_t *dev);
```

"Deinitialize" the display device.

`DeInit` should release any hardware resources the display device uses (I/O ports, IRQs, and so on), turn off the hardware, and perform any other shutdown the device requires.

This operator will be called by the Display Manager when the Display Manager task ends.

Note: *The `DeInit` function generally will not be called under normal operation.*

6.4.5.2.3 Start

```
hal_display_status_t (*start)(const display_dev_t *dev);
```

Start the display device.

The `Start` operator is called in the initialization stage of the Display Manager's task after the call to the `Init` operator. The startup of the display sensor and interface should be implemented in this operator. It includes, for example, starting the interface and enabling the IRQ of the DMA used by the interface.

6.4.5.2.4 Blit

```
hal_display_status_t (*blit)(const display_dev_t *dev,
                             void *frame,
                             int width,
                             int height);
```

Sends a frame to the display panel and "blits" the frame with any additional required components (UI overlay, and so on).

`Blit` is called by the Display Manager once a previously requested frame of the matching srcFormat has been sent by a camera device. The sending of the frame from the Display Manager to the display panel should be take place in this operator.

`kStatus_HAL_DisplaySuccess` must be returned if the frame was successfully sent to the display panel. After calling this operator, the Display Manager will request a new frame.

```
If the `Blit` operator is working in asynchronous mode, the
 hardware will continue sending the frame buffer even after the
 return of the `Blit` function call.
In this case, `kStatus_HAL_DisplayNonBlocking` should be
 returned instead,
and the Display Manager will not issue a new display frame
 request after this `Blit` call.

To request a new frame, the device should invoke the Display
 Manager's callback using a `kDisplayEvent_RequestFrame` event
 to notify the completion of the sending of the previous frame.
Once the Display Manager sees this new request, it will
 requesting a new frame.
```

6.4.5.2.5 InputNotify

```
    hal_display_status_t (*inputNotify)(const display_dev_t
 *dev, void *data);
```

Handle input events.

The `InputNotify` operator is called by the Display Manager whenever a `kFWKMessageID_InputNotify` message is received by and forwarded from the Display Manager's message queue.

For more information regarding events and event handling, see Events.

6.4.5.3 Capabilities

```
/*! @brief Structure that characterizes the display device. */
typedef struct _display_dev_private_capability
{
    /* buffer resolution */
    int height;
    int width;
    int pitch;
    /* active rect */
    int left;
    int top;
```

```
    int right;
    int bottom;
    /* rotate degree */
    cw_rotate_degree_t rotate;
    /* pixel format */
    pixel_format_t format;
    /* the source pixel format of the requested frame */
    pixel_format_t srcFormat;
    void *frameBuffer;
    /* callback */
    display_dev_callback_t callback;
    /* param for the callback */
    void *param;
} display_dev_private_capability_t;
```

The `capabilities` struct is primarily used for storing a callback to communicate information from the device back to the Display Manager. This callback function is typically installed via a device's `init` operator.

Display devices also maintain information regarding the size of the display, pixel format, and other information pertinent to the display.

6.4.5.3.1 Height

```
int height;
```

The height of the display buffer.

6.4.5.3.2 Width

```
int width;
```

The width of the display buffer.

6.4.5.3.3 Pitch

```
int pitch;
```

The total number of bytes in one row of the display buffer.

6.4.5.3.4 Left

```
int left;
```

The left edge of the active areain the display frame buffer.

Note: *The active area indicates the area of the display frame buffer that will be utilized.*

6.4.5.3.5 Top

```
int top;
```

The top edge of the active area in the display frame buffer.

6.4.5.3.6 Right

```
int right;
```

The right edge of the active area in the display frame buffer.

6.4.5.3.7 Bottom

```
int bottom;
```

The bottom edge of the active area in the display frame buffer.

6.4.5.3.8 Rotate

```
typedef enum _cw_rotate_degree
{
    kCWRotateDegree_0 = 0,
    kCWRotateDegree_90,
    kCWRotateDegree_180,
    kCWRotateDegree_270
} cw_rotate_degree_t;
```

```
cw_rotate_degree_t rotate;
```

The rotate degree of the display frame buffer.

6.4.5.3.9 Format

```
typedef enum _pixel_format
{
    /* 2d frame format */
    kPixelFormat_RGB,
    kPixelFormat_RGB565,
    kPixelFormat_BGR,
    kPixelFormat_Gray888,
    kPixelFormat_Gray888X,
    kPixelFormat_Gray,
    kPixelFormat_Gray16,
    kPixelFormat_YUV1P444_RGB,   /* color display sensor */
    kPixelFormat_YUV1P444_Gray,  /* ir display sensor */
    kPixelFormat_UYVY1P422_RGB,  /* color display sensor */
    kPixelFormat_UYVY1P422_Gray, /* ir display sensor */
    kPixelFormat_VYUY1P422,

    /* 3d frame format */
    kPixelFormat_Depth16,
    kPixelFormat_Depth8,

    kPixelFormat_YUV420P,

    kPixelFormat_Invalid
} pixel_format_t;
```

The format of the display frame buffer.

6.4.5.3.10 srcFormat

The source format of the requested display frame buffer.

Because there may be multiple display devices operating at a time, the display checks the `srcFormat` property of the frame to determine whether it is from the display device it is expecting. It prevents the display from displaying a 3D depth image when the user expects an RGB image, for example.

6.4.5.3.11 frameBuffer

Pointer to the display frame buffer.

6.4.5.3.12 callback

```
/**
 * @brief callback function to notify Display Manager that an
 async event took place
 * @param dev Device structure of the display device calling
 this function
 * @param event id of the event that took place
 * @param param Parameters
 * @param fromISR True if this operation takes place in an irq,
 0 otherwise
 * @return 0 if the operation was successfully
 */
typedef int (*display_dev_callback_t)(const display_dev_t *dev,
                                      display_event_t event,
                                      void *param,
                                      uint8_t fromISR);
```

```
display_dev_callback_t callback;
```

Callback to the Display Manager. The HAL device invokes this callback to notify the Display Manager of specific events.

```
Currently, only the `kDisplayEvent_RequestFrame` event callback
 is implemented in the Display Manager.
```

The Display Manager provides this callback to the device when the `init` operator is called. As a result, the HAL device must make sure to store the callback in the `init` operator's implementation.

```
hal_display_status_t HAL_DisplayDev_ExampleDev_Init(
    display_dev_t *dev, int width, int height,
 display_dev_callback_t callback, void *param)
{
    hal_display_status_t ret = kStatus_HAL_DisplaySuccess;

    /* PERFORM INIT FUNCTIONALITY HERE */

    ...

    /* Installing callback function from manager... */
    dev->cap.callback    = callback;

    return ret;
```

```
}
```

The HAL device invokes this callback to notify the Display Manager of specific events.

6.4.5.3.13 param

```
void *param;
```

The parameter of the Display Manager callback.

```
The `param` field is not currently used by the framework in any
 way.
```

6.4.5.4 Example

The project has several display devices implemented for use as-is or as reference for implementing new display devices. The source files for these display HAL devices can be found under `HAL/common/`.

Below is an example of the "rk024hh298" display HAL device driver `HAL/common/hal_display_lcdif_rk024hh298.c`.

```
hal_display_status_t
 HAL_DisplayDev_LcdifRk024hh2_Init(display_dev_t *dev,
                                                                int
 width,
                                                                int
 height,

 display_dev_callback_t callback,
                                                                void
 *param);
hal_display_status_t HAL_DisplayDev_LcdifRk024hh2_Uninit(const
 display_dev_t *dev);
hal_display_status_t HAL_DisplayDev_LcdifRk024hh2_Start(const
 display_dev_t *dev);
hal_display_status_t HAL_DisplayDev_LcdifRk024hh2_Blit(const
 display_dev_t *dev,
                                                                void
 *frame,
                                                                int
 width,
                                                                int
 height);
static hal_display_status_t
 HAL_DisplayDev_LcdifRk024hh2_InputNotify(const display_dev_t
 *receiver,

     void *data);

/* The operators of the rk024hh298 Display HAL Device */
const static display_dev_operator_t s_DisplayDev_LcdifOps = {
    .init        = HAL_DisplayDev_LcdifRk024hh2_Init,
    .deinit      = HAL_DisplayDev_LcdifRk024hh2_Uninit,
    .start       = HAL_DisplayDev_LcdifRk024hh2_Start,
    .blit        = HAL_DisplayDev_LcdifRk024hh2_Blit,
    .inputNotify = HAL_DisplayDev_LcdifRk024hh2_InputNotify,
```

```
};

/* rk024hh298 Display HAL Device */
static display_dev_t s_DisplayDev_Lcdif = {
    .id   = 0,
    .name = DISPLAY_NAME,
    .ops  = &s_DisplayDev_LcdifOps,
    .cap  = {
        .width       = DISPLAY_WIDTH,
        .height      = DISPLAY_HEIGHT,
        .pitch       = DISPLAY_WIDTH * DISPLAY_BYTES_PER_PIXEL,
        .left        = 0,
        .top         = 0,
        .right       = DISPLAY_WIDTH - 1,
        .bottom      = DISPLAY_HEIGHT - 1,
        .rotate      = kCWRotateDegree_0,
        .format      = kPixelFormat_RGB565,
        .srcFormat   = kPixelFormat_UYVY1P422_RGB,
        .frameBuffer = NULL,
        .callback    = NULL,
        .param       = NULL
        }
    };

hal_display_status_t
 HAL_DisplayDev_LcdifRk024hh2_Init(display_dev_t *dev,
                                                                int
 width,
                                                                int
 height,

 display_dev_callback_t callback,
                                                                void
 *param)
{
    hal_display_status_t ret = kStatus_HAL_DisplaySuccess;

    /* init the capability */
    dev->cap.width       = width;
    dev->cap.height      = height;
    dev->cap.frameBuffer = (void *)&s_FrameBuffers[1];

    /* store the callback and param for late using */
    dev->cap.callback    = callback;

    /* init the low level display panel and interface */

    return ret;
}

hal_display_status_t HAL_DisplayDev_LcdifRk024hh2_Uninit(const
 display_dev_t *dev)
{
    hal_display_status_t ret = kStatus_HAL_DisplaySuccess;
    /* Currently do nothing for the Deinit as we didn't support
 the runtime de-registraion of the device */
    return ret;
}
```

```
hal_display_status_t HAL_DisplayDev_LcdifRk024hh2_Start(const
 display_dev_t *dev)
{
    hal_display_status_t ret = kStatus_HAL_DisplaySuccess;

    /* start the display pannel and the interface */

    return ret;
}

hal_display_status_t HAL_DisplayDev_LcdifRk024hh2_Blit(const
 display_dev_t *dev, void *frame, int width, int height)
{
    hal_display_status_t ret = kStatus_HAL_DisplayNonBlocking;

    /* blit the frame to the real display pannel */

    return ret;
}

static hal_display_status_t
 HAL_DisplayDev_LcdifRk024hh2_InputNotify(const display_dev_t
 *receiver, void *data)
{
    hal_display_status_t error           =
 kStatus_HAL_DisplaySuccess;
    event_base_t eventBase               = *(event_base_t
 *)data;
    event_status_t event_response_status = kEventStatus_Ok;

    /* handle the events which are interested in */
    if (eventBase.eventId == kEventID_SetDisplayOutputSource)
    {

    }

    return error;
}
```

6.4.6 Vision algorithm devices

The Vision Algorithm HAL device type represents an abstraction for computer vision algorithms which are used for the analysis of digital images, videos, and other visual inputs.

The crux of the design for Vision Algorithm devices is centered around the use of "infer complete" events that communicate information about the results of inferencing that is handled by the device. For example, in the current application, the Vision Algorithm may receive a camera frame containing a recognized face, perform an inference on that data, and communicate a "face recognized" message to other devices so that they may act accordingly. For more information about events and event handling, see Events.

Currently, only one vision algorithm device can be registered to the Vision Manager at a time per the design of the framework.

6.4.6.1 Device definition

The HAL device definition for vision algorithm devices can be found under `framework/hal_api/hal_valgo_dev.h` and is reproduced below:

```
/*! @brief definition of a vision algo device */
typedef struct _vision_algo_dev
{
    /* unique id which is assigned by vision algorithm manager
 during the registration */
    int id;
    /* name to identify */
    char name[DEVICE_NAME_MAX_LENGTH];
    /* private capability */
    valgo_dev_private_capability_t cap;
    /* operations */
    vision_algo_dev_operator_t *ops;
    /* private data */
    vision_algo_private_data_t data;
} vision_algo_dev;
```

The operators associated with the vision algo HAL device are as shown below:

```
/*! @brief Operation that needs to be implemented by a vision
 algorithm device */
typedef struct
{
    /* initialize the dev */
    hal_valgo_status_t (*init)(vision_algo_dev_t *dev,
 valgo_dev_callback_t callback, void *param);
    /* deinitialize the dev */
    hal_valgo_status_t (*deinit)(vision_algo_dev_t *dev);
    /* run the inference */
    hal_valgo_status_t (*run)(const vision_algo_dev_t *dev,
 void *data);
    /* recv events */
    hal_valgo_status_t (*inputNotify)(const vision_algo_dev_t
 *receiver, void *data);

} vision_algo_dev_operator_t;
```

The capabilities associated with the vision algo HAL device are as shown below:

```
typedef struct _valgo_dev_private_capability
{
    /* callback */
    valgo_dev_callback_t callback;
    /* param for the callback */
    void *param;
} valgo_dev_private_capability_t;
```

The private data fields associated with the vision algo HAL device are as shown below:

```
typedef struct
{
    int autoStart;
    /* frame type definition */
    vision_frame_t frames[kVAlgoFrameID_Count];
```

```
} vision_algo_private_data_t;
```

6.4.6.2 Operators

Operators are functions that "operate" on a HAL device itself. Operators are akin to "public methods" in object-oriented languages and are used by the Vision Algorithm Manager to set up, start, and so on, its registered vision algo device.

For more information about operators, see Section 6.4.1.3.1.

6.4.6.2.1 Init

```
hal_valgo_status_t (*init)(vision_algo_dev_t *dev,
 valgo_dev_callback_t callback, void *param);
```

Initialize the vision algo HAL device.

`Init` must initialize any hardware resources the device requires (I/O ports, IRQs, and so on), turn on the hardware, and perform any other setup required by the device.

The callback function to the device's manager is typically installed as part of the `Init` function as well.

This operator is called by the vision algorithm manager when the output manager task first starts.

6.4.6.2.2 Deinit

```
hal_valgo_status_t (*deinit)(vision_algo_dev_t *dev);
```

The `DeInit` function is used to "deinitialize" the algorithm device. `DeInit` must release any hardware resources the device uses (I/O ports, IRQs, and so on), turn off the hardware, and perform any other shutdown required by the device.

This operator is called by the Vision Algorithm Manager when the Vision Algorithm Manager task ends.

Note: *The `DeInit` function generally is not called under normal operation.*

6.4.6.2.3 Run

```
hal_valgo_status_t (*run)(const voice_algo_dev_t *dev, void
 *data);
```

Begin running the vision algorithm.

The `run` operator is used to start running algorithm inference and processing camera frame data.

This operator is called by the Vision Algorithm manager when a "camera frame ready" message is received from the Camera Manager and forwarded to the algorithm device via the Vision Algorithm Manager.

Once the Vision Algorithm device finishes processing the camera frame data, its manager forwards this message to the Output Manager in the form of an "inference complete" message.

6.4.6.2.4 InputNotify

```
hal_valgo_status_t (*inputNotify)(const vision_algo_dev_t
 *receiver, void *data);
```

Handle input events.

The `InputNotify` operator is called by the Vision Algorithm Manager whenever a `kFWKMessageID_InputNotify` message is received and forwarded from the Vision Algorithm Manager's message queue.

For more information regarding events and event handling, see Events.

6.4.6.3 Capabilities

The `capabilities` struct is primarily used for storing a callback to communicate information from the device back to the Vision Algorithm Manager. This callback function is typically installed via a device's `init` operator.

6.4.6.3.1 Callback

```
/*!
 * @brief Callback function to notify managers the results of
 inference
 * valgo_dev* dev Pointer to an algorithm device
 * valgo_event_t event Event which took place
 * void* param Pointer to a struct of data that needs to be
 forwarded
 * unsigned int size Size of the struct that needs to be
 forwarded. If size = 0, param should be a pointer to a
 * persistent memory area.
 */

typedef int (*valgo_dev_callback_t)(int devId, valgo_event_t
 event, void *param, unsigned int size, uint8_t fromISR);
```

```
valgo_dev_callback_t callback;
```

Callback to the Vision Algorithm Manager.

The Vision Algorithm manager provides the callback to the device when the `init` operator is called. As a result, the HAL device should make sure to store the callback in the `init` operator's implementation.

```
static hal_valgo_status_t
 HAL_VisionAlgoDev_ExampleDev_Init(vision_algo_dev_t *dev,

 valgo_dev_callback_t callback,
                                                        void
 *param)
{
    hal_valgo_status_t ret = kStatus_HAL_ValgoSuccess;

    /* PERFORM INIT FUNCTIONALITY HERE */

    ...

    /* Installing callback function from manager... */
```

```
    memset(&dev->cap, 0, sizeof(dev->cap));
    dev->cap.callback = callback;

    return ret;
}
```

The HAL device invokes this callback to notify the Vision Algorithm manager of specific events.

6.4.6.3.2 Param

```
void *param;
```

The param for the callback (optional).

6.4.6.4 Private Data

6.4.6.4.1 AutoStart

```
int autoStart;
```

The flag for automatic start of the algorithm.

If `autoStart` is 1, the Vision Algorithm Manager automatically starts requesting camera frames for this algorithm device after its `init` operator is executed.

6.4.6.4.2 Frames

```
vision_frame_t frames[kVAlgoFrameID_Count];
```

The three kinds of frames that are currently supported by the vision framework are `RGB`, `IR`, and `Depth` images.

The vision algorithm device must specify information for each kind of frame so that the framework properly converts and passes only the frames which correspond to this algorithm device's requirement.

For example, older Solution's projects like SLN-VIZN3D-IOT use both 3D Depth and IR camera images to perform liveness detection and face recognition, while using RGB frames solely for use as user feedback help with aligning a user's face, and so on. Therefore, the algorithm device must ensure that it is receiving only the 3D and IR frames and not any RGB frames.

The definition of `vision_frame_t` is as shown below:

```
typedef struct _vision_frame
{
    /* is supported by the device for this type of frame */
    /* Vision Algorithm Manager will only request the supported
 frame for this device */
    int is_supported;

    /* frame resolution */
    int height;
    int width;
    int pitch;
```

```
    /* rotate degree */
    cw_rotate_degree_t rotate;
    flip_mode_t flip;
    /* swap byte per two bytes */
    int swapByte;

    /* pixel format */
    pixel_format_t format;

    /* the source pixel format of the requested frame */
    pixel_format_t srcFormat;
    void *data;
} vision_frame_t;
```

6.4.6.5 Example

As only one Vision Algorithm device can be registered at a time per the design of the framework, the project has one Vision Algorithm device implemented.

Note: *This example is implemented using NXP's OasisLite face recognition algorithm, which is the core vision computing algorithm used in all projects.*

This example is reproduced below:

```
static hal_valgo_status_t
 HAL_VisionAlgoDev_OasisLite_Init(vision_algo_dev_t *dev,

 valgo_dev_callback_t callback,
                                                                void
 *param);
static hal_valgo_status_t
 HAL_VisionAlgoDev_OasisLite_Deinit(vision_algo_dev_t *dev);
static hal_valgo_status_t HAL_VisionAlgoDev_OasisLite_Run(const
 vision_algo_dev_t *dev, void *data);
static hal_valgo_status_t
 HAL_VisionAlgoDev_OasisLite_InputNotify(const
 vision_algo_dev_t *receiver, void *data);

/* vision algorithm device operators */
const static vision_algo_dev_operator_t
 s_VisionAlgoDev_OasisLiteOps = {
    .init        = HAL_VisionAlgoDev_OasisLite_Init,
    .deinit      = HAL_VisionAlgoDev_OasisLite_Deinit,
    .run         = HAL_VisionAlgoDev_OasisLite_Run,
    .inputNotify = HAL_VisionAlgoDev_OasisLite_InputNotify,
};

/* vision algorithm device */
static vision_algo_dev_t s_VisionAlgoDev_OasisLite3D = {
    .id   = 0,
    .name = "OASIS_3D",
    .ops  = (vision_algo_dev_operator_t
 *)&s_VisionAlgoDev_OasisLiteOps,
    .cap  = {.param = NULL},
};

/* vision algorithm device Init function*/
static hal_valgo_status_t
 HAL_VisionAlgoDev_OasisLite_Init(vision_algo_dev_t *dev,
```

```
 valgo_dev_callback_t callback,
                                                                  void
 *param)
{
    LOGI("++HAL_VisionAlgoDev_OasisLite_Init");
    hal_valgo_status_t ret = kStatus_HAL_ValgoSuccess;

    // init the device
    memset(&dev->cap, 0, sizeof(dev->cap));
    dev->cap.callback = callback;

    /* set parameters of the requested frames that this vision
 algorithm dev asks for*/
    /* for example oasisLite algorithm asks for two kind of
 frames: one is IR, the other is Depth */
    /* firstly set parameters of the requested IR frames */
    dev->data.autoStart                          = 1;
    dev->data.frames[kVAlgoFrameID_IR].height       =
 OASIS_FRAME_HEIGHT;
    dev->data.frames[kVAlgoFrameID_IR].width        =
 OASIS_FRAME_WIDTH;
    dev->data.frames[kVAlgoFrameID_IR].pitch        =
 OASIS_FRAME_WIDTH * 3;
    dev->data.frames[kVAlgoFrameID_IR].is_supported = 1;
    dev->data.frames[kVAlgoFrameID_IR].rotate       =
 kCWRotateDegree_0;
    dev->data.frames[kVAlgoFrameID_IR].flip         =
 kFlipMode_None;
    dev->data.frames[kVAlgoFrameID_IR].format    =
 kPixelFormat_BGR;
    dev->data.frames[kVAlgoFrameID_IR].srcFormat =
 kPixelFormat_Gray16;
    int oasis_lite_rgb_frame_aligned_size        =
 SDK_SIZEALIGN(OASIS_FRAME_HEIGHT * OASIS_FRAME_WIDTH * 3, 64);
    dev->data.frames[kVAlgoFrameID_IR].data      =
 pvPortMalloc(oasis_lite_rgb_frame_aligned_size);

    if (dev->data.frames[kVAlgoFrameID_IR].data == NULL)
    {
        OASIS_LOGE("[ERROR]: Unable to allocate memory for
 kVAlgoFrameID_IR.");
        ret = kStatus_HAL_ValgoMallocError;
        return ret;
    }
    /* secondly set parameters of the requested Depth frames */
    dev->data.frames[kVAlgoFrameID_Depth].height       =
 OASIS_FRAME_HEIGHT;
    dev->data.frames[kVAlgoFrameID_Depth].width        =
 OASIS_FRAME_WIDTH;
    dev->data.frames[kVAlgoFrameID_Depth].pitch        =
 OASIS_FRAME_WIDTH * 2;
    dev->data.frames[kVAlgoFrameID_Depth].is_supported = 1;
    dev->data.frames[kVAlgoFrameID_Depth].rotate       =
 kCWRotateDegree_0;
    dev->data.frames[kVAlgoFrameID_Depth].flip         =
 kFlipMode_None;

    dev->data.frames[kVAlgoFrameID_Depth].format    =
 kPixelFormat_Depth16;
```

```
    dev->data.frames[kVAlgoFrameID_Depth].srcFormat =
 kPixelFormat_Depth16;
    int oasis_lite_depth_frame_aligned_size         =
 SDK_SIZEALIGN(OASIS_FRAME_HEIGHT * OASIS_FRAME_WIDTH * 2, 64);
    dev->data.frames[kVAlgoFrameID_Depth].data      =
 pvPortMalloc(oasis_lite_depth_frame_aligned_size);

    if (dev->data.frames[kVAlgoFrameID_Depth].data == NULL)
    {
        OASIS_LOGE("Unable to allocate memory for
 kVAlgoFrameID_IR");
        ret = kStatus_HAL_ValgoMallocError;
        return ret;
    }

    /* do private Algorithm Init here */
    ...

    LOGI("--HAL_VisionAlgoDev_OasisLite_Init");
    return ret;
}

/* vision algorithm device DeInit function*/
static hal_valgo_status_t
 HAL_VisionAlgoDev_OasisLite_Deinit(vision_algo_dev_t *dev)
{
    hal_valgo_status_t ret = kStatus_HAL_ValgoSuccess;
    LOGI("++HAL_VisionAlgoDev_OasisLite_Deinit");

    /* release resource here */
    ...

    LOGI("--HAL_VisionAlgoDev_OasisLite_Deinit");
    return ret;
}

/* vision algorithm device inference run function*/
static hal_valgo_status_t HAL_VisionAlgoDev_OasisLite_Run(const
 vision_algo_dev_t *dev, void *data)
{
    hal_valgo_status_t ret = kStatus_HAL_ValgoSuccess;
    OASIS_LOGI("++HAL_VisionAlgoDev_OasisLite_Run");

    vision_algo_result_t result;
    /* do inference run, derive meaningful information from the
 current frame data in dev private data */
    /* for example, oasisLite will inference according to two
 kinds of input frames:
       void* frame1 = dev->data.frames[kVAlgoFrameID_IR].data
       void* frame2 = dev-
>data.frames[kVAlgoFrameID_Depth].data
       result = oasisLite_run(frame1, frame2, ......);
    */
    ...

    /* execute algorithm manager callback to inform algorithm
 manager the result */
    if (dev != NULL && result != NULL && dev->cap.callback !=
 NULL)
    {
```

```
        dev->cap.callback(dev->id,
 kVAlgoEvent_VisionResultUpdate, result,
 sizeof(vision_algo_result_t), 0);
    }

    OASIS_LOGI("--HAL_VisionAlgoDev_OasisLite_Run");
    return ret;
}

/* vision algorithm device InputNotify function*/
static hal_valgo_status_t
 HAL_VisionAlgoDev_OasisLite_InputNotify(const
 vision_algo_dev_t *receiver, void *data)
{
    hal_valgo_status_t ret = kStatus_HAL_ValgoSuccess;
    OASIS_LOGI("++HAL_VisionAlgoDev_OasisLite_InputNotify");
    event_base_t eventBase = *(event_base_t *)data;

    /* do proess according to different input notify event */
    ...

    LOGI("--HAL_VisionAlgoDev_OasisLite_InputNotify");
    return ret;
}

/* register vision algorithm device to vision algorithm manager
 */
int HAL_VisionAlgoDev_OasisLite3D_Register()
{
    int error = 0;
    LOGD("HAL_VisionAlgoDev_OasisLite3D_Register");
    error = FWK_VisionAlgoManager_DeviceRegister(

 &s_VisionAlgoDev_OasisLite3D);

    return error;
}
```

6.4.7 Voice algorithm devices

The Voice Algorithm HAL device type represents an abstraction to do voice recognition based on clean stream AFE generated.

After the Voice Algorithm manager receives the clean stream, the Voice Algorithm Hal device `run` method is called. If a voice command is detected, the device outputs the inference result and transfer result to the Output HAL device through `valgo_dev_callback_t callback`. For more information about events and event handling, see Events.

Currently, only one voice algorithm device can be registered to the Voice Manager at a time per the design of the framework.

6.4.7.1 Device definition

The HAL device definition for voice algorithm devices can be found under `framework/hal_api/hal_valgo_dev.h` and is reproduced below:

```
/*! @brief Attributes of a voice algo device */
```

```
struct _voice_algo_dev
{
    /* unique id which is assigned by algorithm manager during
 the registration */
    int id;
    /* name to identify */
    char name[DEVICE_NAME_MAX_LENGTH];
    /* private capability */
    valgo_dev_private_capability_t cap;
    /* operations */
    voice_algo_dev_operator_t *ops;
    /* private data */
    voice_algo_private_data_t data;
};
```

The operators associated with the voice algo HAL device are as shown below:

```
/*! @brief Operation that needs to be implemented by a voice
 algorithm device */
typedef struct voice_algo_dev_operator_t
{
    /* initialize the dev */
    hal_valgo_status_t (*init)(voice_algo_dev_t *dev,
 valgo_dev_callback_t callback, void *param);
    /* deinitialize the dev */
    hal_valgo_status_t (*deinit)(voice_algo_dev_t *dev);
    /* start the dev */
    hal_valgo_status_t (*run)(const voice_algo_dev_t *dev, void
 *data);
    /* recv events */
    hal_valgo_status_t (*inputNotify)(const voice_algo_dev_t
 *receiver, void *data);

} voice_algo_dev_operator_t;
```

The capabilities associated with the voice algo HAL device are as shown below:

```
typedef struct _valgo_dev_private_capability
{
    /* callback */
    valgo_dev_callback_t callback;
    /* param for the callback */
    void *param;
} valgo_dev_private_capability_t;
```

The private data fields associated with the voice algo HAL device is as shown below:

```
typedef struct _voice_algo_private_data
{
} voice_algo_private_data_t;
```

6.4.7.2 Operators

Operators are functions that "operate" on a HAL device itself. Operators are akin to "public methods" in object-oriented languages, and are used by the Voice Algorithm Manager to init, run, and so on its registered voice algo device.

For more information about operators, see Section 6.4.1.3.1.

6.4.7.2.1 Init

```
hal_valgo_status_t (*init)(voice_algo_dev_t *dev,
 valgo_dev_callback_t callback, void *param);
```

Init the voice algo HAL device.

`Init` performs all setups the device requires, such as preparing memory for voice algorithm runtime consumption, loading AI models, running library initialization API and so on.

The callback function to the device's manager is typically installed as part of the `Init` function as well.

This operator is called by the voice algorithm manager when the voice manager task first starts.

6.4.7.2.2 Deinit

```
hal_valgo_status_t (*deinit)(voice_algo_dev_t *dev);
```

The `DeInit` function is used to "deinitialize" the algorithm device. `DeInit` must release any hardware resources the device uses (heap memory, handles created by device, and so on), turn off the hardware, and perform any other shutdown required by the device.

This method is not called in AFE Manager based on current framework version.

Note: *The `DeInit` function generally is not called under normal operation.*

6.4.7.2.3 Run

```
hal_valgo_status_t (*run)(const voice_algo_dev_t *dev, void
 *data);
```

Begin running the voice algorithm.

The `run` operator is used to start running algorithm inference and processing voice frame data.

This operator is called by the Voice Algorithm manager when the `kFWKMessageID_VAlgoASRInputProcess` message is received from the AFE Manager and forwarded to the algorithm device via the Voice Algorithm Manager.

Once the Voice Algorithm device finishes processing the voice frame data, its manager forwards the inference result to the Output Manager. If Wake Word is detected, Voice manager forwards a message indicating length of wake word to AFE manager.

6.4.7.2.4 InputNotify

```
hal_valgo_status_t (*inputNotify)(const voice_algo_dev_t
 *receiver, void *data);
```

Handle input events.

The `InputNotify` operator is called by the Voice Algorithm Manager whenever the `kFWKMessageID_InputNotify` message is received and forwarded from the Voice Algorithm Manager's message queue.

For more information regarding events and event handling, see Events.

6.4.7.3 Capabilities

The `capabilities` struct is primarily used for storing a callback to communicate information from the device back to the Voice Algorithm Manager. This callback function is typically installed via a device's `init` operator.

6.4.7.3.1 Callback

```
/*!
 * @brief Callback function to notify managers the results of
 inference
 * valgo_dev* dev Pointer to an algorithm device
 * valgo_event_t event Event which took place
 * void* param Pointer to a struct of data that needs to be
 forwarded
 * unsigned int size Size of the struct that needs to be
 forwarded. If size = 0, param should be a pointer to a
 * persistent memory area.
 */

typedef int (*valgo_dev_callback_t)(int devId, valgo_event_t
 event, void *param, unsigned int size, uint8_t fromISR);
```

```
valgo_dev_callback_t callback;
```

Callback to the Voice Algorithm Manager.

The Voice Algorithm manager provides the callback to the device when the `init` operator is called. As a result, the HAL device must make sure to store the callback in the `init` operator's implementation.

The HAL device invokes this callback to notify the Voice Algorithm manager of specific events.

6.4.7.3.2 Param

```
void *param;
```

The param for the callback (optional).

6.4.7.4 Example

Because only one Voice Algorithm device can be registered at a time per the design of the framework, the SLN-TLHMI-IOT project has two Voice Algorithm devices(DSMT/VIT) implemented.

Note: *This example is implemented using the DSMT algorithm*

This example is reproduced below:

```
hal_valgo_status_t voice_algo_dev_asr_init(voice_algo_dev_t
 *dev, valgo_dev_callback_t callback, void *param)
static hal_valgo_status_t
 HAL_VisionAlgoDev_OasisLite_Deinit(vision_algo_dev_t *dev);
```

```
hal_valgo_status_t voice_algo_dev_asr_run(const
 voice_algo_dev_t *dev, void *data)
hal_valgo_status_t voice_algo_dev_input_notify(const
 voice_algo_dev_t *dev, void *data)

const static voice_algo_dev_operator_t voice_algo_dev_asr_ops =
 {
    .init        = voice_algo_dev_asr_init,
    .deinit      = NULL,
    .run         = voice_algo_dev_asr_run,
    .inputNotify = voice_algo_dev_input_notify
};

static voice_algo_dev_t voice_algo_dev_asr = {
    .id  = 0,
    .ops = (voice_algo_dev_operator_t
 *)&voice_algo_dev_asr_ops,
    .cap = {.param = NULL},
};

hal_valgo_status_t voice_algo_dev_asr_init(voice_algo_dev_t
 *dev, valgo_dev_callback_t callback, void *param)
{
    hal_valgo_status_t ret = kStatus_HAL_ValgoSuccess;
    uint32_t timerId       = 0;

    /* Set callback function */
    dev->cap.callback = callback;

    ...

    /* Initialize the ASR engine */
    initialize_asr();

    ...

    return ret;
}

/* voice algorithm device inference run function*/
hal_valgo_status_t voice_algo_dev_asr_run(const
 voice_algo_dev_t *dev, void *data)
{
    hal_valgo_status_t status    = kStatus_HAL_ValgoSuccess;
    static asr_events_t asrEvent = ASR_SESSION_ENDED;
    struct asr_inference_engine *pInfWW;
    struct asr_inference_engine *pInfCMD;
    char **cmdString;
    int16_t *pi16Sample;

    audio_msg_payload_t *audioIn = (audio_msg_payload_t *)data;

    ...

    /* Wake Word detection. Check all enabled languages, but
 stop on first match. */
    for (pInfWW = s_AsrEngine.voiceControl.infEngineWW;
 pInfWW != NULL; pInfWW = pInfWW->next)
    {
```

```
        if (asr_process_audio_buffer(pInfWW->handler,
 pi16Sample, NUM_SAMPLES_AFE_OUTPUT, pInfWW->iWhoAmI_inf) ==
 kAsrLocalDetected)

        {
            LOGI("Trust: %d, SGDiff: %d\r\n",
 s_AsrEngine.voiceControl.result.trustScore,

 s_AsrEngine.voiceControl.result.SGDiffScore);
        }
    }

    ...

    return status;
}

hal_valgo_status_t voice_algo_dev_input_notify(const
 voice_algo_dev_t *dev, void *data)
{
    hal_valgo_status_t error = kStatus_HAL_ValgoSuccess;
    event_voice_t event      = *(event_voice_t *)data;
    const char *language_str = NULL;

    ...

    return error;
}

int HAL_VoiceAlgoDev_Asr_Register()
{
    int error = 0;
    LOGD("HAL_VoiceAlgoDev_Asr_Register");
    error =
 FWK_VoiceAlgoManager_DeviceRegister(&voice_algo_dev_asr);
    return error;
}
```

6.4.8 Audio processing device

Audio Processing Device is used for Audio Front End (AFE) processing. In the following sections, we abridge 'Audio Processing Device' as 'AFE device'. And also use 'AFE manager' instead of 'audio_processing manager'.

The `AFE` HAL device provides an abstraction to represent audio front-end(AFE)handling.

AFE provides several subalgorithm modules, finally outputting a clean stream for the ASR engine. AFE supports Beamformer, AEC, NS, and DOA. Beamformer eliminates reverberation and background noise. AEC (Acoustic Echo Cancellation) can support multi-channel systems, which is used for suppressing local speaker stream. DOA (Direction Of Arrival) tracking has 1-degree resolution.

The AFE device receives microphone streams and reference streams (speaker streams) and outputs a clean stream for the ASR engine.

As with other device types, the AFE device is controlled via the AFE manager. The AFE manager is responsible for managing all registered AFE HAL devices, and invoking AFE device operators (`init`, `start`, `run`, `stop`, and so on) as necessary. Additionally, the

AFE Manager allows for multiple AFE devices to be registered and operate at once. Based on real project requirements, in most cases, only one AFE device is needed.

6.4.8.1 Device definition

The HAL device definition for `AFE` devices can be found under `framework/hal_api/hal_audio_processing_dev.h` and is reproduced below:

```
typedef struct _audio_processing_dev audio_processing_dev_t;
/*! @brief Attributes of an audio processing device. */
struct _audio_processing_dev
{
    /* unique id which is assigned by audio processing manager
 during registration */
    int id;
    /* name of the device */
    char name[DEVICE_NAME_MAX_LENGTH];
    /* operations */
    const audio_processing_dev_operator_t *ops;
    /* private capability */
    audio_processing_dev_private_capability_t cap;
};
```

The device operators associated with AFE HAL devices are as shown below:

```
/*! @brief Operation that needs to be implemented by a audio
 processing device */
typedef struct _audio_processing_dev_operator
{
    /* initialize the dev */
    hal_audio_processing_status_t (*init)
(audio_processing_dev_t *dev, audio_processing_dev_callback_t
 callback);
    /* deinitialize the dev */
    hal_audio_processing_status_t (*deinit)(const
 audio_processing_dev_t *dev);
    /* start the dev */
    hal_audio_processing_status_t (*start)(const
 audio_processing_dev_t *dev);
    /* start the dev */
    hal_audio_processing_status_t (*stop)(const
 audio_processing_dev_t *dev);
    /* notify the audio_processing_dev_t */
    hal_audio_processing_status_t (*run)(const
 audio_processing_dev_t *dev, void *param);
    /* notify the audio_processing_dev_t */
    hal_audio_processing_status_t (*inputNotify)(const
 audio_processing_dev_t *dev, void *param);
} audio_processing_dev_operator_t;
```

The device capabilities associated with AFE HAL devices are as shown below:

```
/*! @brief Structure that capability of the AFE device. */
typedef struct _audio_processing_dev_private_capability
{
    /* callback */
    audio_processing_dev_callback_t callback;
} audio_processing_dev_private_capability_t;
```

6.4.8.2 Operators

Operators are functions which "operate" on a HAL device itself. Operators are akin to "public methods" in object-oriented languages, and are used by the AFE Manager to set up, start, and so on, each of its registered AFE devices.

6.4.8.2.1 Init

```
hal_audio_processing_status_t (*init)(audio_processing_dev_t
 *dev, audio_processing_dev_callback_t callback);
```

Initialize the AFE device.

`Init` performs all setups that the device requires, such as preparing memory for AFE runtime consumption, microphone number and position, and so on.

This operator is called by the AFE Manager when the AFE Manager task first starts.

6.4.8.2.2 Deinit

```
hal_audio_processing_status_t (*deinit)(const
 audio_processing_dev_t *dev);
```

De-initialize the AFE device.

`DeInit` releases all memory resources allocated in initialization stage. Set all handles created to NULL.

This operator is not called in AFE Manager based on current framework version.

Note: *The `DeInit` function is not called under normal operation.*

6.4.8.2.3 Start

```
hal_audio_processing_status_t (*start)(const
 audio_processing_dev_t *dev);
```

Start the AFE device.

The `Start` operator is called in the initialization stage of the AFE Manager's task after the call to the `Init` operator. Since AFE device is a pure software device, there is not Clock/GPIO, or any peripheral bus depended. In most cases, the `Start` method can return `kStatus_HAL_AudioProcessingSuccess` directly.

6.4.8.2.4 Stop

```
hal_audio_processing_status_t (*stop)(const
 audio_processing_dev_t *dev);
```

`Stop` is reverted operation compared to `Start`. Return `kStatus_HAL_AudioProcessingSuccess` if there is nothing needed to be done to device.

For the AFE device SDK implemented, this method returns `kStatus_HAL_AudioProcessingSuccess` directly. And it is not called in AFE Manager based on current framework version.

6.4.8.2.5 Run

```
hal_audio_processing_status_t (*run)(const
 audio_processing_dev_t *dev, void *param);
```

Execute AFE engine for handling microphone stream and outputting clean stream.

The `Run` operator will be called by the AFE Manager to handle audio frame with 160 samples.

6.4.8.2.6 InputNotify

```
 hal_audio_processing_status_t (*inputNotify)(const
 audio_processing_dev_t *dev, void *param);
```

Handle input events.

The `InputNotify` operator is called by the AFE Manager whenever a `kFWKMessageID_InputNotify` message is received by and forwarded from the AFE Manager's message queue.

For more information regarding events and event handling, see Events.

6.4.8.3 Capabilities

```
typedef struct _audio_processing_dev_private_capability
{
    /* callback */
    audio_processing_dev_callback_t callback;
} audio_processing_dev_private_capability_t;
```

The `capabilities` struct is primarily used for storing a callback to communicate information from the device back to the AFE Manager. This callback function is typically installed via a device's `init` operator.

6.4.8.3.1 Callback

```
/**
 * @brief Callback function to notify audio processing manager
 that an async event took place
 * @param dev Device structure of the audio processing device
 calling this function
 * @param event id of the event that took place
 * @param param Parameters
 * @param fromISR True if this operation takes place in an irq,
 0 otherwise
 * @return 0 if the operation was successfully
 */
typedef int (*audio_processing_dev_callback_t)(
    const audio_processing_dev_t *dev,
    audio_processing_event_t event,
    void *param, unsigned int size,
    uint8_t fromISR);
```

Callback to the AFE Manager.

The HAL device invokes this callback to notify the AFE Manager of specific events like "audio processing done or audio dumping event."

The AFE Manager provides this callback to the device when the `init` operator is called. As a result, the HAL device should make sure to store the callback in the `init` operator's implementation.

```
hal_audio_processing_status_t
 audio_processing_afe_init(audio_processing_dev_t *dev,

 audio_processing_dev_callback_t callback)
{
    hal_audio_processing_status_t error =
 kStatus_HAL_AudioProcessingSuccess;

    sln_afe_status_t afeStatus = kAfeSuccess;
    sln_afe_config_t afeConfig = {0};

    dev->cap.callback = callback;

    afeConfig.numberOfMics    = AUDIO_PDM_MIC_COUNT;
    afeConfig.afeMemBlock     = s_afeExternalMemory;
    ....
    return error;
}
```

6.4.8.3.2 Param

```
void *param;
```

The parameter of the callback points to audio data AFE outputting.

6.4.8.4 Example

The SLN-TLHMI-IOT project implements one AFE device for use as-is or for use as reference for implementing new AFE devices. Source files for these AFE HAL devices can be found under `hal/voice/hal_audio_processing_afe.c`.

```
const static audio_processing_dev_operator_t
 audio_processing_afe_ops = {
    .init        = audio_processing_afe_init,
    .deinit      = audio_processing_afe_deinit,
    .start       = audio_processing_afe_start,
    .stop        = audio_processing_afe_stop,
    .run         = audio_processing_afe_run,
    .inputNotify = audio_processing_afe_notify,
};

static audio_processing_dev_t audio_processing_afe = {
    .id = 1, .name = "AFE", .ops =
 &audio_processing_afe_ops, .cap = {.callback = NULL}};

hal_audio_processing_status_t
 audio_processing_afe_init(audio_processing_dev_t *dev,

 audio_processing_dev_callback_t callback)
{
```

```
    hal_audio_processing_status_t error =
 kStatus_HAL_AudioProcessingSuccess;
 /*
     * Prepare AFE memory and configuration parameters needed,
     * and then initialize AFE library.
 */

    return error;
}

hal_audio_processing_status_t audio_processing_afe_deinit(const
 audio_processing_dev_t *dev)
{
    hal_audio_processing_status_t error =
 kStatus_HAL_AudioProcessingSuccess;
    return error;
}

hal_audio_processing_status_t audio_processing_afe_start(const
 audio_processing_dev_t *dev)
{
    hal_audio_processing_status_t error =
 kStatus_HAL_AudioProcessingSuccess;
    return error;
}

hal_audio_processing_status_t audio_processing_afe_stop(const
 audio_processing_dev_t *dev)
{
    hal_audio_processing_status_t error =
 kStatus_HAL_AudioProcessingSuccess;
    return error;
}

hal_audio_processing_status_t audio_processing_afe_notify(const
 audio_processing_dev_t *dev, void *param)
{
    hal_audio_processing_status_t error =
 kStatus_HAL_AudioProcessingSuccess;
    event_voice_t event                = *(event_voice_t
 *)param;

 /* Parse event structure and do further handling */

    return error;
}

hal_audio_processing_status_t audio_processing_afe_run(const
 audio_processing_dev_t *dev, void *param)
{
    hal_audio_processing_status_t error =
 kStatus_HAL_AudioProcessingSuccess;
    event_voice_t event                = *(event_voice_t
 *)param;

    /* Parse event structure and execute AFE engine for
 handling microphone streams */

    return error;
}
```

6.4.9 Flash devices

The flash HAL device represents an abstraction used to implement a device that handles all operations dealing with flash (permanent) storage.

Note: *Even though the word "flash" is used in the terminology of this device, the user is technically capable of implementing an FS that uses a volatile memory instead. One potential reason for doing so would be to run logic/sanity checks on the filesystem API's before implementing them on a flash device. Ultimately, the flash HAL device is useful for abstracting not only flash operations, but memory operations in general.*

The flash HAL device is primarily used as a wrapper over an underlying filesystem, be it LittleFS, FatFS, and so on. As a result, the Flash Manager only allows one flash device to be registered because there is usually no need for multiple file systems operating at the same time.

General information

Because only one flash device can be registered at a time, it means that API calls to the Flash Manager essentially act as wrappers over the flash HAL device's operators.

In terms of functionality, the flash HAL device provides:

- Read/Write operations
- Cleanup methods to handle defragmentation and/or emptying flash sectors during idle time
- Information about underlying flash mapping and flash type

6.4.9.1 Device definition

The HAL device definition for flash devices can be found under `framework/hal_api/hal_flash_dev.h` and is reproduced below:

```
/*! @brief Attributes of a flash device */
struct _flash_dev
{
    /* unique id */
    int id;
    /* operations */
    const flash_dev_operator_t *ops;
};
```

The device operators associated with flash HAL devices are as shown below:

```
/*! @brief Callback function to timeout check requester list
 busy status. */
typedef int (*lpm_manager_timer_callback_t)(lpm_dev_t *dev);

/*! @brief Operation that needs to be implemented by a flash
 device */
typedef struct _flash_dev_operator
{
    sln_flash_status_t (*init)(const flash_dev_t *dev);
    sln_flash_status_t (*deinit)(const flash_dev_t *dev);
    sln_flash_status_t (*format)(const flash_dev_t *dev);
    sln_flash_status_t (*save)(const flash_dev_t *dev, const
 char *path, void *buf, unsigned int size);
    sln_flash_status_t (*append)(const flash_dev_t *dev, const
 char *path, void *buf, unsigned int size, bool overwrite);
```

```
    sln_flash_status_t (*read)(const flash_dev_t *dev, const
 char *path, void *buf, unsigned int offset, unsigned int
 *size);
    sln_flash_status_t (*mkdir)(const flash_dev_t *dev, const
 char *path);
    sln_flash_status_t (*mkfile)(const flash_dev_t *dev, const
 char *path, bool encrypt);
    sln_flash_status_t (*rm)(const flash_dev_t *dev, const char
 *path);
    sln_flash_status_t (*rename)(const flash_dev_t *dev, const
 char *oldPath, const char *newPath);
    sln_flash_status_t (*cleanup)(const flash_dev_t *dev,
 unsigned int timeout_ms);
} flash_dev_operator_t;
```

6.4.9.2 Operators

Operators are functions that "operate" on a HAL device itself. Operators are akin to "public methods" in object oriented-languages.

For more information about operators, see Section 6.4.1.3.1 .

6.4.9.2.1 Init

```
sln_flash_status_t (*init)(const flash_dev_t *dev);
```

Initialize the flash and filesystem.

`Init` must initialize any hardware resources required by the flash device (pins, ports, clock, and so on) In addition to initializing the hardware, the `init` function should also mount the filesystem.

Note: *An application that runs from flash (does XiP) must not initialize/deinitialize any hardware. If a hardware change is truly needed, the change must be performed with caution.*

Note: *Some lightweight FS may not require mounting and can be prebuilt/preloaded on the flash instead. Regardless, the `init` function must result in the filesystem being in a usable state.*

6.4.9.2.2 Deinit

```
hal_lpm_status_t (*deinit)(const lpm_dev_t *dev);
```

"Deinitialize" the flash and filesystem.

`DeInit` must release any hardware resources a flash device might use (I/O ports, IRQs, and so on), turn off the hardware, and perform any other shutdown the device requires.

6.4.9.2.3 Format

```
sln_flash_status_t (*format)(const flash_dev_t *dev);
```

Clean and format the filesystem.

6.4.9.2.4 Save

```
sln_flash_status_t (*save)(const flash_dev_t *dev, const char
 *path, void *buf, unsigned int size);
```

Save a file with the contents of `buf` to `path` in the filesystem.

6.4.9.2.5 Append

```
sln_flash_status_t (*append)(const flash_dev_t *dev, const char
 *path, void *buf, unsigned int size, bool overwrite);
```

Append the contents of `buf` to an existing file at `path`.

Setting `overwrite` equal to true causes append from the beginning of the file instead.

Note: *`overwrite == true` makes this function nearly equivalent to the save function, the only difference is that this does not create a new file.*

6.4.9.2.6 Read

```
sln_flash_status_t (*read)(const flash_dev_t *dev, const char
 *path, void *buf, unsigned int offset, unsigned int *size);
```

Read a file from the filesystem located at `path` and storing the contents in `buf`.

To find the needed space for the buf, call read with `buf` set to NULL. In case there is not enough space in memory to read the whole file, read with offset can be use while specifying the chunk size.

Note: *It is up to the user to guarantee that the buffer supplied will fit the contents of the file being read.*

6.4.9.2.7 Make directory

```
sln_flash_status_t (*mkdir)(const flash_dev_t *dev, const char
 *ph);
```

Create a directory at `path`.

```
If the filesystem in use does not support directories,
this operator can be set to `NULL`.
```

6.4.9.2.8 Make file

```
sln_flash_status_t (*mkfile)(const flash_dev_t *dev, const char
 *path, bool encrypt);
```

Creates the file mentioned by the path. If the information needs to stored not in plain text, encryption can be enabled.

6.4.9.2.9 Remove

```
sln_flash_status_t (*rm)(const flash_dev_t *dev, const char
 *path);
```

Remove the file at `path`.

```
If the filesystem in use does not support directories,
this operator can be set to `NULL`.
```

6.4.9.2.10 Rename

```
sln_flash_status_t (*rename)(const flash_dev_t *dev, const char
 *oldPath, const char *newPath);
```

Rename/move a file from `oldPath` to `newPath`.

6.4.9.2.11 Cleanup

```
sln_flash_status_t (*cleanup)(const flash_dev_t *dev, unsigned
 int timeout_ms);
```

Clean up the filesystem.

This function is used to help minimize delays introduced by things like fragmentation caused during "erase sector" operations that can lead to unwanted delays when searching for the next available sector.

`timeout_ms` specifies how much time to wait while performing cleanup. This helps prevent multiple HAL devices calling `cleanup` and stalling the filesystem.

6.4.9.3 Example

As only one flash device can be registered at a time per the design of the framework, the project has only one filesystem implemented.

The source file for this flash HAL device can be found at `HAL/common/hal_flash_littlefs.c`.

In this example, we demonstrate a way to integrate Littlefs in our framework.

Littlefs is a lightweight file-system that is designed to handle random power failures. The architecture of the file-system allows having directories and files. As a result, this example uses the following file layout:

```
root-directory
├── cfg
│   ├── Metadata
│   ├── fwk_cfg - stores framework related information.
│   └── app_cfg - stores app specific information.
├── oasis
│   ├── Metadata
│   └── faceFiles - the number of files that stores faces are
 up to 100
├── app_specific
└── wifi_info
```

```
    └── wifi_info
```

6.4.9.3.1 Littlefs device

```
static sln_flash_status_t _lfs_init()
{
    int res = kStatus_HAL_FlashSuccess;
    if (s_LittlefsHandler.lfsMounted)
    {
        return kStatus_HAL_FlashSuccess;
    }
    s_LittlefsHandler.lock = xSemaphoreCreateMutex();
    if (s_LittlefsHandler.lock == NULL)
    {
        LOGE("Littlefs create lock failed");
        return kStatus_HAL_FlashFail;
    }

    _lfs_get_default_config(&s_LittlefsHandler.cfg);
#if DEBUG
    BOARD_InitFlashResources();
#endif
    SLN_Flash_Init();
    if (res)
    {
        LOGE("Littlefs storage init failed: %i", res);
        return kStatus_HAL_FlashFail;
    }

    res = lfs_mount(&s_LittlefsHandler.lfs,
 &s_LittlefsHandler.cfg);
    if (res == 0)
    {
        s_LittlefsHandler.lfsMounted = 1;
        LOGD("Littlefs mount success");
    }
    else if (res == LFS_ERR_CORRUPT)
    {
        LOGE("Littlefs corrupt");
        lfs_format(&s_LittlefsHandler.lfs,
 &s_LittlefsHandler.cfg);
        LOGD("Littlefs attempting to mount after
 reformatting...");
        res = lfs_mount(&s_LittlefsHandler.lfs,
 &s_LittlefsHandler.cfg);
        if (res == 0)
        {
            s_LittlefsHandler.lfsMounted = 1;
            LOGD("Littlefs mount success");
        }
        else
        {
            LOGE("Littlefs mount failed again");
            return kStatus_HAL_FlashFail;
        }
    }
    else
    {
        LOGE("Littlefs error while mounting");
```

```
    }

    return res;
}

static sln_flash_status_t _lfs_cleanupHandler(const flash_dev_t
 *dev,

 unsigned int timeout_ms)
{
    sln_flash_status_t status              =
 kStatus_HAL_FlashSuccess;
    uint32_t usedBlocks[LFS_SECTORS/32]    = {0};
    uint32_t emptyBlocks                   = 0;
    uint32_t startTime                     = 0;
    uint32_t currentTime                   = 0;

    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }

    /* create used block list */
    lfs_fs_traverse(&s_LittlefsHandler.lfs,
 _lfs_traverse_create_used_blocks,
                                 &usedBlocks);

    startTime = sln_current_time_us();

    /* find next block starting from free.i */
    for (int i = 0; i < LFS_SECTORS; i++)
    {
        currentTime = sln_current_time_us();
        /* Check timeout */
        if ((timeout_ms) && (currentTime >= (startTime +
 timeout_ms * 1000)))
        {
            break;
        }

        lfs_block_t block = (s_LittlefsHandler.lfs.free.i + i)
 % LFS_SECTORS;

        /* take next unused marked block */
        if (!_is_blockBitSet(usedBlocks, block))
        {
            /* If the block is marked as free but not yet
 erased, try to erase it */
            LOGD("Block %i is unused, try to erase it", block);
            _lfs_qspiflash_erase(&s_LittlefsConfigDefault,
 block);
            emptyBlocks += 1;
        }
    }

    LOGI("%i empty_blocks starting from %i available in %ims",
                emptyBlocks, s_LittlefsHandler.lfs.free.i,
 (sln_current_time_us() - startTime)/1000);
```

```
    _unlock();
    return status;
}

static sln_flash_status_t _lfs_formatHandler(const flash_dev_t
 *dev)
{
    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }
    lfs_format(&s_LittlefsHandler.lfs, &s_LittlefsHandler.cfg);
    _unlock();
    return kStatus_HAL_FlashSuccess;
}

static sln_flash_status_t _lfs_rmHandler(const flash_dev_t
 *dev, const char *path)
{
    int res;

    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }

    res = lfs_remove(&s_LittlefsHandler.lfs, path);
    if (res)
    {
        LOGE("Littlefs while removing: %i", res);
        _unlock();
        if (res == LFS_ERR_NOENT)
        {
            return kStatus_HAL_FlashFileNotExist;
        }

        return kStatus_HAL_FlashFail;
    }
    _unlock();
    return kStatus_HAL_FlashSuccess;
}

static sln_flash_status_t _lfs_mkdirHandler(const flash_dev_t
 *dev, const char *path)
{
    int res;

    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }

    res = lfs_mkdir(&s_LittlefsHandler.lfs, path);

    if (res == LFS_ERR_EXIST)
    {
        LOGD("Littlefs directory exists: %i", res);
```

```
        _unlock();
        return kStatus_HAL_FlashDirExist;
    }
    else if (res)
    {
        LOGE("Littlefs creating directory: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }
    _unlock();
    return kStatus_HAL_FlashSuccess;
}

static sln_flash_status_t _lfs_writeHandler(const flash_dev_t
 *dev, const char *path, void *buf, unsigned int size)
{
    int res;
    lfs_file_t file;

    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }

    res = lfs_file_opencfg(&s_LittlefsHandler.lfs, &file, path,
 LFS_O_CREAT, &s_FileDefault);
    if (res)
    {
        LOGE("Littlefs opening file: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }

    res = lfs_file_write(&s_LittlefsHandler.lfs, &file, buf,
 size);
    if (res < 0)
    {
        LOGE("Littlefs writing file: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }

    res = lfs_file_close(&s_LittlefsHandler.lfs, &file);
    if (res)
    {
        LOGE("Littlefs closing file: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }

    _unlock();
    return kStatus_HAL_FlashSuccess;
}

static sln_flash_status_t _lfs_appendHandler(const flash_dev_t
 *dev,

 const char *path,
```

```
 void *buf,

 unsigned int size,

 bool overwrite)
{
    int res;
    lfs_file_t file;

    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }

    res = lfs_file_opencfg(&s_LittlefsHandler.lfs, &file, path,
 LFS_O_APPEND, &s_FileDefault);
    if (res)
    {
        LOGE("Littlefs opening file: %i", res);
        _unlock();
        if (res == LFS_ERR_NOENT)
        {
            return kStatus_HAL_FlashFileNotExist;
        }
        return kStatus_HAL_FlashFail;
    }

    if (overwrite == true)
    {
        res = lfs_file_truncate(&s_LittlefsHandler.lfs, &file,
 0);

        if (res < 0)
        {
            LOGE("Littlefs truncate file: %i", res);
            _unlock();
            return kStatus_HAL_FlashFail;
        }
    }

    res = lfs_file_write(&s_LittlefsHandler.lfs, &file, buf,
 size);
    if (res < 0)
    {
        LOGE("Littlefs writing file: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }

    res = lfs_file_close(&s_LittlefsHandler.lfs, &file);
    if (res)
    {
        LOGE("Littlefs closing file: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }

    _unlock();
```

```
    return kStatus_HAL_FlashSuccess;
}

static sln_flash_status_t _lfs_readHandler(const flash_dev_t
 *dev, const char *path, void *buf, unsigned int size)
{
    int res;
    int offset = 0;
    lfs_file_t file;

    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }
    res = lfs_file_opencfg(&s_LittlefsHandler.lfs, &file, path,
 LFS_O_RDONLY, &s_FileDefault);
    if (res)
    {
        LOGE("Littlefs opening file: %i", res);
        _unlock();
        if (res == LFS_ERR_NOENT)
        {
            return kStatus_HAL_FlashFileNotExist;
        }
        return kStatus_HAL_FlashFail;
    }

    do
    {
        res = lfs_file_read(&s_LittlefsHandler.lfs, &file, (buf
 + offset), size);
        if (res < 0)
        {
            LOGE("Littlefs reading file: %i", res);
            _unlock();
            return kStatus_HAL_FlashFail;
        }
        else if (res == 0)
        {
            LOGD("Littlefs reading file \"%s\": Read only %d.
 %d bytes not found ", path, offset, size);
            break;
        }

        offset += res;
        size -= res;
    } while (size > 0);

    res = lfs_file_close(&s_LittlefsHandler.lfs, &file);
    if (res)
    {
        LOGE("Littlefs closing file: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }

    _unlock();
    return kStatus_HAL_FlashSuccess;
}
```

```
static sln_flash_status_t _lfs_renameHandler(const flash_dev_t
 *dev, const char *OldPath, const char *NewPath)
{
    int res;

    if (_lock())
    {
        LOGE("Littlefs _lock failed");
        return kStatus_HAL_FlashFail;
    }

    res = lfs_rename(&s_LittlefsHandler.lfs, OldPath, NewPath);
    if (res)
    {
        LOGE("Littlefs renaming file: %i", res);
        _unlock();
        return kStatus_HAL_FlashFail;
    }
    _unlock();
    return kStatus_HAL_FlashSuccess;
}

const static flash_dev_operator_t s_FlashDev_LittlefsOps = {
    .init   = _lfs_init,
    .deinit = NULL,
    .format = _lfs_formatHandler,
    .append = _lfs_appendHandler,
    .save   = _lfs_writeHandler,
    .read   = _lfs_readHandler,
    .mkdir  = _lfs_mkdirHandler,
    .rm     = _lfs_rmHandler,
    .rename = _lfs_renameHandler,
    .cleanup= _lfs_cleanupHandler,
};

static flash_dev_t s_FlashDev_Littlefs = {
    .id  = 0,
    .ops = &s_FlashDev_LittlefsOps,
};

int HAL_FlashDev_Littlefs_Init()
{
    int error = 0;
    LOGD("++HAL_FlashDev_Littlefs_Init");
    _lfs_init();

    LOGD("--HAL_FlashDev_Littlefs_Init");
    error = FWK_Flash_DeviceRegister(&s_FlashDev_Littlefs);

    FWK_LpmManager_RegisterRequestHandler(&s_LpmReq);
    return error;
}
```

Note: *The information presented here shows only the operators described above. For more information regarding Littlefs configuration, FlexSPI configuration, optimization done, check the full code base.*

6.4.10 Multicore devices

The multicore HAL device represents an abstraction used to implement a device that handles all multicore message passing.

The Multicore HAL device is primarily used as a wrapper over known multicore message libraries, be it MU/Mailbox peripheral registers, rpmsg_lite, eRPC, and so on.

In terms of functionality, the multicore HAL device provides:

- Send operation
- Receive operation

6.4.10.1 Device definition

The HAL device definition for multicore devices can be found under `framework/hal_api/hal_multicore_dev.h` and is reproduced below:

```
/*! @brief Attributes of a multicore device. */
struct _multicore_dev
{
    /* unique id which is assigned by multicore manager during
 the registration */
    int id;
    /* name of the device */
    char name[DEVICE_NAME_MAX_LENGTH];
    /* operations */
    const multicore_dev_operator_t *ops;
    /* private capability */
    multicore_dev_private_capability_t cap;
};
```

The device operators associated with multicore HAL devices are as shown below:

```
/*! @brief Operation that needs to be implemented by a
 multicore device */
typedef struct _multicore_dev_operator
{
    /* initialize the dev */
    hal_multicore_status_t (*init)(multicore_dev_t *dev,
 multicore_dev_callback_t callback, void *param);
    /* deinitialize the dev */
    hal_multicore_status_t (*deinit)(const multicore_dev_t
 *dev);
    /* start the dev */
    hal_multicore_status_t (*start)(const multicore_dev_t
 *dev);
    /* Multicore Send the message */
    hal_multicore_status_t (*send)(const multicore_dev_t *dev,
 void *data, unsigned int size);
    /* input notify */
    hal_multicore_status_t (*inputNotify)(const multicore_dev_t
 *dev, void *data);
} multicore_dev_operator_t;
```

In order to achieve a two-way communication between cores, hal devices need to implement both send and receive operations. The send is triggered by the multicore

manager, while receive is async, the other core being able to send at any moment. All async operations are handled within Multicore manager callback.

```
/**
 * @brief callback function to notify multicore manager that an
 async event took place
 * @param dev Device structure of the multicore device calling
 this function
 * @param event  the event that took place
 * @param fromISR True if this operation takes place in an irq,
 0 otherwise
 * @return 0 if the operation was successfully
 */
typedef int (*multicore_dev_callback_t)(const multicore_dev_t
 *dev, multicore_event_t event, uint8_t fromISR);

/*! @brief Structure that characterizes the multicore device.
 */
typedef struct _multicore_dev_private_capability
{
    /* callback */
    multicore_dev_callback_t callback;

} multicore_dev_private_capability_t;
```

6.4.10.2 Operators

Operators are functions which "operate" on a HAL device itself. Operators are akin to "public methods" in object oriented-languages.

For more information about operators, see Section 6.4.1.3.1.

6.4.10.2.1 Init

```
/* initialize the dev */
hal_multicore_status_t (*init)(multicore_dev_t *dev,
 multicore_dev_callback_t callback, void *param);
```

`Init` should initialize any hardware resources required by the multicore device (pins, ports, clock, and so on).

6.4.10.2.2 Deinit

```
/* deinitialize the dev */
hal_multicore_status_t (*deinit)(const multicore_dev_t *dev);
```

"Deinitialize" the multicore device.

`DeInit` should release any hardware resources a multicore device might use (I/O ports, IRQs, and so on), turn off the hardware, and perform any other shutdown the device requires.

6.4.10.2.3 Start

```
/* start the dev */
hal_multicore_status_t (*start)(const multicore_dev_t *dev);
```

`Start` should start the flow. Handshake protocol can be implemented. The purpose of a handshake protocol is to verify that both cores initialized properly the multicore unit.

6.4.10.2.4 Send

```
/* Multicore Send the message */
hal_multicore_status_t (*send)(const multicore_dev_t *dev, void
 *data, unsigned int size);
```

Multicore manager passes a buffer to the underlying level. The multicore device must send the message, characterized by the size, to the counterpart device from the other core. On the other side, after receiving the message, the hal device is responsible to call the callback, to make the multicore manager aware of the new message.

6.4.10.3 FreeRTOS message buffer device

Message buffers from FreeRTOS are used for one-way communication between two threads. In order to create a two-way communication, a send task and receive task must be created on both cores. Multicore Manager acts as a `send task`, while the `receive task` is created within the Hal device init. The receive task also inherits the priority of the send task. The send and receive task should be built having a non-blocking design pattern in mind and they should be initialized with highest priority in order to have the best response time.

The number of shared buffers that must be allocated is two, one for each one way communication. The size is at least the maximum message size, after a deep copy has been performed. They should be allocated statically at compile or a procedure to advertise between cores the address should be implemented.

- CM7/ Write Buffer = CM4/ Read Buffer
- CM4/ Write Buffer = CM7/ Read Buffer

For more information about RTOS Message Buffers API, check FreeRTOS documentation

```
void vGenerateMulticoreInterrupt(void *xUpdatedMessageBuffer)
{
    /* Trigger the inter-core interrupt using the MCMGR
 component.
       Pass the APP_MESSAGE_BUFFER_EVENT_DATA as data that
 accompany
       the kMCMGR_FreeRtosMessageBuffersEvent event. */

 (void)MCMGR_TriggerEventForce(kMCMGR_FreeRsMessageBuffersEvent,
 kMulticore_DataEvent);
}

static void RemoteAppReadyEventHandler(uint16_t eventData, void
 *context)
{
    *(bool *)context = (bool)eventData;
}

static void FreeRtosMessageBuffersEventHandler(uint16_t
 eventData, void *context)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
```

```
    /* Make sure the message has been addressed to us. Using
 eventData that accompany
       the event of the kMCMGR_FreeRtosMessageBuffersEvent
 type, we can distinguish
       different consumers. */
    if (kMulticore_DataEvent == eventData)
    {
        /* Call the API function that sends a notification to
 any task that is
    blocked on the xUpdatedMessageBuffer message buffer waiting
 for data to
    arrive. */

 (void)xMessageBufferSendCompletedFromISR(xReadMessageBuffer,
 &xHigherPriorityTaskWoken);
    }

    /* Normal FreeRTOS "yield from interrupt" semantics, where
    HigherPriorityTaskWoken is initialzed to pdFALSE and will
 then get set to
    pdTRUE if the interrupt unblocks a task that has a priority
 above that of
    the currently executing task. */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

    /* No need to clear the interrupt flag here, it is handled
 by the mcmgr. */
}

static void _HAL_MulticoreDev_MessageBuffer_RcvMsgHandler(void
 *param)
{
    /* Size to cover on MAX message. Can be lowered if we know
 what we send */
    static uint8_t pMessageBufferRcv[MB_STORAGE_BUFFER_SIZE];

    while (1)
    {
        size_t xReceivedBytes =
 xMessageBufferReceive(xReadMessageBuffer, (void
 *)pMessageBufferRcv,

 sizeof(pMessageBufferRcv), portMAX_DELAY);

        LOGI("Remote Message receive, size = %d",
 xReceivedBytes);
        if ((xReceivedBytes != 0) && (

 s_MulticoreDev_MessageBuffer.cap.callback != NULL))
        {
            multicore_event_t multicore_event;
            multicore_event.eventId =
 kMulticoreEvent_MsgReceive;
            multicore_event.data    = pMessageBufferRcv;
            multicore_event.size    = xReceivedBytes;
  s_MulticoreDev_MessageBuffer.cap.callback(
                              &s_MulticoreDev_MessageBuffer,
 multicore_event, false);
        }
    }
```

```
}

static hal_multicore_status_t
 HAL_MulticoreDev_MessageBuffer_Deinit(const multicore_dev_t
 *dev)
{
    hal_multicore_status_t status =
 kStatus_HAL_MulticoreSuccess;

    return status;
}

static hal_multicore_status_t
 HAL_MulticoreDev_MessageBuffer_Send(const multicore_dev_t
 *dev, void *data, uint32_t size)
{
    hal_multicore_status_t status =
 kStatus_HAL_MulticoreSuccess;

    if ((data != NULL) && (size != 0))
    {
        uint32_t streamFreeSpace =
 xStreamBufferSpacesAvailable(xWriteMessageBuffer);
        if (streamFreeSpace < size)
        {
            status = kStatus_HAL_MulticoreError;
            LOGE("Not enough space, free %x needed %x",
 streamFreeSpace, size);
        }

        if (status == kStatus_HAL_MulticoreSuccess)
        {
            (void)xMessageBufferSend(xWriteMessageBuffer, data,
 size, 0);
            LOGI("MulticoreDev_send: Send %d bytes", size);
        }
    }
    else
    {
        LOGD("MulticoreDev_send: Nothing to send");
    }

    return status;
}

static hal_multicore_status_t
 HAL_MulticoreDev_MessageBuffer_InputNotify(const
 multicore_dev_t *dev, void *data)
{
    hal_multicore_status_t status =
 kStatus_HAL_MulticoreSuccess;

    return status;
}

static hal_multicore_status_t
 HAL_MulticoreDev_MessageBuffer_Start(const multicore_dev_t
 *dev)
{
```

```
    hal_multicore_status_t status =
 kStatus_HAL_MulticoreSuccess;

    /* Wait until the secondary core application signals it is
 ready to communicate. */
    while (true != s_SecondCoreReady)
    {
        (void)MCMGR_TriggerEvent(kMCMGR_RemoteApplicationEvent,
 true);
        vTaskDelay(pdMS_TO_TICKS(10));
    };

    /* Send one more event to be sure the other core got it */
    (void)MCMGR_TriggerEvent(kMCMGR_RemoteApplicationEvent,
 true);

    if
 (xTaskCreate(_HAL_MulticoreDev_MessageBuffer_RcvMsgHandler,
 MULTICORE_RCV_TASK_NAME, MULTICORE_RCV_TASK_STACK,
                  NULL, uxTaskPriorityGet(NULL), NULL) !=
 pdPASS)
    {
        LOGE("[MessageBuffer] Task creation failed!.");
        while (1)
            ;
    }

    return status;
}

static hal_multicore_status_t
 HAL_MulticoreDev_MessageBuffer_Init(multicore_dev_t *dev,

   multicore_dev_callback_t callback,

   void *param)
{
    hal_multicore_status_t status =
 kStatus_HAL_MulticoreSuccess;
    LOGD("Start Multicore MessageBuffer INIT");

    s_MulticoreDev_MessageBuffer.cap.callback = callback;

    xWriteMessageBuffer = xMessageBufferCreateStatic(
        /* The buffer size in bytes. */
        MB_STORAGE_BUFFER_SIZE,
        /* Statically allocated buffer storage area. */
        &ucWriteMessageBufferStorage,
        /* Message buffer handle. */
        &xWriteMessageBufferStruct);


 (void)MCMGR_RegisterEvent(kMCMGR_FreeRtosMessageBuffersEvent,
 FreeRtosMessageBuffersEventHandler, ((void *)0));
    (void)MCMGR_RegisterEvent(kMCMGR_RemoteApplicationEvent,
 RemoteAppReadyEventHandler, (void *)&s_SecondCoreReady);

    /* We initied we are ready to rcv messages */
    LOGD("Exit Multicore MessageBuffer INIT");
    return status;
```

```
}
```

6.5 Events

6.5.1 Overview

Events are a means by which information is communicated between different devices via their managers.

6.5.1.1 Event triggers

Events can correspond to many different happenings during the runtime of the application, and can include things like:

- Button pressed
- Face detected
- Shell command received

When an event is triggered, the device that first received the event communicates that event to its manager, that in turn notifies other managers designated to receive the event.

For example, when a button is pressed, a flow similar to the following takes place:

1. The "Push Button" HAL device receives an interrupt corresponding to the button that was pressed.
2. Inside the HAL device's interrupt handler, the device associates an event with the button that was pressed.
3. The HAL device specifies which managers should receive the event.
4. The HAL device forwards the event to its manager.

The code that corresponds to this scenario can be seen in the below excerpts from `HAL/common/hal_input_push_buttons.c`" and `source/event_handlers/smart_lock_input_push_buttons.c`, respectively.

```
void _HAL_InputDev_IrqHandler(button_data_t *button,
 switch_press_type_t pressType)
{
    if (s_InputDev_PushButtons.cap.callback != NULL)
    {
        uint32_t receiverList;
        if (APP_InputDev_PushButtons_SetEvent(button->buttonId,
 pressType, &s_pEvent, &receiverList) == kStatus_Success)
        {
            s_inputEvent.inputData = s_pEvent;
            uint8_t fromISR        = __get_IPSR();

 s_InputDev_PushButtons.cap.callback(&s_InputDev_PushButtons,
 kInputEventID_Recv, receiverList,
                                                 &s_inputEvent,
 0, fromISR);
        }
        else
        {
            LOGE("No valid event associated with SW%d button %s
 press", button->buttonId,
                 pressType == kSwitchPressType_Short ?
 "short" : "long");
```

```
        }
    }
}
```

```
The "callback" function in the above code refers to an
 internal callback function inside the [Input Manager](../
device_managers/input_manager.md)
which relays input events to each of the managers specified in
 an event's `receiverList`.
```

```
    switch (button)
    {
        case kSwitchID_1:
            if (pressType == kSwitchPressType_Long)
            {
                LOGD("Long PRESS Detected.");
                unsigned int totalUsageCount;
                FWK_LpmManager_RequestStatus(&totalUsageCount);

 FWK_LpmManager_EnableSleepMode(kLPMManagerStatus_SleepEnable);
            }
            break;

        case kSwitchID_2:
            if ((pressType == kSwitchPressType_Short) ||
 (pressType == kSwitchPressType_Long))
            {
                *receiverList                    = 1 <<
 kFWKTaskID_VisionAlgo;
                s_FaceRecEvent.eventBase.eventId =
 kEventFaceRecID_DelUser;
                s_FaceRecEvent.delFace.hasName   = false;
                s_FaceRecEvent.delFace.hasID     = false;
                *event                           =
 &s_FaceRecEvent;
            }
            break;

        case kSwitchID_3:
            if ((pressType == kSwitchPressType_Short) ||
 (pressType == kSwitchPressType_Long))
            {
                *receiverList                    = 1 <<
 kFWKTaskID_VisionAlgo;
                s_FaceRecEvent.eventBase.eventId =
 kEventFaceRecID_AddUser;
                s_FaceRecEvent.addFace.hasName   = false;
                *event                           =
 &s_FaceRecEvent;
            }
            break;

        default:
            ret = kStatus_Fail;
            break;
    }

    return ret;
```

6.5.1.2 Types of events

Events can be used to communicate all sorts of information, but the two types of events defined by default are **InferComplete** events and **InputNotify** events.

Both types of events represent different information being communicated to and by the HAL devices.

6.5.1.2.1 InferComplete events

Inference events are used to indicate that a vision/voice algorithm HAL device has completed a stage in its inference pipeline.

Note: *Only output HAL devices can respond to `InferComplete` events. This is not true of `InputNotify` events.*

In the current application, it can refer to several things, including:

- Face detected
- Face recognized
- Fake face detected

Output HAL devices can respond to inference events by implementing an `inferComplete` method. When an "InferComplete" event is triggered, the output manager attempts to call the `inferComplete` event handler of each of its devices, (assuming the device has implemented an `inferComplete` function).

As part of the `inferComplete` function call, the output manager also communicates the HAL device from which the event originated, the ID of the event received, as well as any additional information related to the event that was generated.

For example, a "Face Recognized" event also includes the ID of the face being recognized. Below is an example of how the RGB LED HAL device responds to several different events.

```
static hal_output_status_t
 HAL_OutputDev_RgbLed_InferComplete(const output_dev_t *dev,

 output_algo_source_t source,

 void *inferResult)
{
    vision_algo_result_t *visionAlgoResult =
 (vision_algo_result_t *)inferResult;
    hal_output_status_t error              =
 kStatus_HAL_OutputSuccess;

    if (visionAlgoResult != NULL)
    {
        if (visionAlgoResult->id == kVisionAlgoID_OasisLite)
        {
            oasis_lite_result_t *result = &(visionAlgoResult-
>oasisLite);
            if (source == kOutputAlgoSource_Vision)
            {
                if ((result->face_recognized) && (result-
>face_id >= 0))
                {
                    RGB_LED_SET_COLOR(kRGBLedColor_Green);
                }
```

```
                else if (result->face_count)
                {
                    RGB_LED_SET_COLOR(kRGBLedColor_Red);
                }
                else
                {
                    RGB_LED_SET_COLOR(kRGBLedColor_Off);
                }
            }
        }
```

For more information about handling events, see Event handlers.

6.5.1.2.2 InputNotify events

Input events are events that indicate that input has been received by an input HAL device.

```
Only input HAL devices can generate an "InputNotify" event.
However, all HAL devices
(with the exception of LPM, Flash, and Graphics devices)
are able to respond to an "InputNotify" event.
```

Examples of input events include:

- Button pressed
- Shell command received
- Wi-Fi/BLE input received

The event to generate for a given input is decided by the device which receives the input.

For example, the Push-Button device associates different events based on the different button presses and the duration of those button presses, either long or short presses.

```
    switch (button)
    {
        case kSwitchID_1:
            if (pressType == kSwitchPressType_Long)
            {
                LOGD("Long PRESS Detected.");
                unsigned int totalUsageCount;
                FWK_LpmManager_RequestStatus(&totalUsageCount);

FWK_LpmManager_EnableSleepMode(kLPMManagerStatus_SleepEnable);
            }
            break;

        case kSwitchID_2:
            if ((pressType == kSwitchPressType_Short) ||
(pressType == kSwitchPressType_Long))
            {
                *receiverList                    = 1 <<
kFWKTaskID_VisionAlgo;
                s_FaceRecEvent.eventBase.eventId =
kEventFaceRecID_DelUser;
                s_FaceRecEvent.delFace.hasName   = false;
                s_FaceRecEvent.delFace.hasID     = false;
                *event                           =
&s_FaceRecEvent;
```

```
            }
            break;

        case kSwitchID_3:
            if ((pressType == kSwitchPressType_Short) ||
(pressType == kSwitchPressType_Long))
            {
                *receiverList                    = 1 <<
kFWKTaskID_VisionAlgo;
                s_FaceRecEvent.eventBase.eventId =
kEventFaceRecID_AddUser;
                s_FaceRecEvent.addFace.hasName   = false;
                *event                           =
&s_FaceRecEvent;
            }
            break;

        default:
            ret = kStatus_Fail;
            break;
    }
```

Alongside an input event, the HAL device from which the event originated may also relay additional information . Depending on the event, this may correspond to the button that was pressed, the shell command and args that were received, and so on.

In the above example, we can see that pressing the SW3 push-button generates a `kEventFaceRecID_AddUser` event, specifying that there is no name for the face to add.

```
A list of general events can be found in
 `hal_event_descriptor_common.h`, while a list of
 face recognition-specific events can be found in
 `hal_event_descriptor_face_rec.h`.
It is recommended that new events be added to the
 `hal_event_descriptor_common.h` file.
```

To respond to an "InputNotify" event, a HAL device must implement an `inputNotify` handler function. When an "InputNotify" event is triggered, each manager which receives the event attempts to call the `inputNotify` method of every one of its devices (assuming the device has implemented an `inputNotify` method).

For more information regarding event handlers, see Event handlers.

6.5.2 Event handlers

Because events are the primary means by which the framework communicates between devices, a mechanism to respond to those events is necessary for them to be useful. Event handlers were created for this explicit purpose.

There are two kinds of event handler:

- Default Handlers
- App-specific Handlers

Event handlers, like other device operators, are passed via the device's operator struct to its manager.

```
const static display_dev_operator_t s_DisplayDev_LcdifOps = {
```

```
    .init        = HAL_DisplayDev_LcdifRk024hh2_Init,
    .deinit      = HAL_DisplayDev_LcdifRk024hh2_Uninit,
    .start       = HAL_DisplayDev_LcdifRk024hh2_Start,
    .blit        = HAL_DisplayDev_LcdifRk024hh2_Blit,
    .inputNotify = HAL_DisplayDev_LcdifRk024hh2_InputNotify,
};
```

Each HAL device may define its own handlers for any given event. For example, a developer may want the RGB LEDs to turn green when a face is recognized, but have the UI display a specific overlay for that same event. To do it, the RGB Output HAL device and the UI Output HAL device can each implement an `InferComplete` handler which will be called by their manager when an "InferComplete" event is received.

```
A HAL device does NOT have to implement an event handler for
 any specific event,
nor does it have to implement an `InputNotify` handler
 (applicable for most device types)
or an `InferComplete` handler (applicable only for output
 devices).
```

6.5.2.1 Default handlers

Default event handlers are exactly what their name would suggest -- the default means by which a device handles events. A HAL device's default event handlers (`InputNotify`, `InferComplete`, and so on) can be found in the HAL device driver itself.

Nearly every device has a default handler implemented, although most devices will only actually handle a few types of events.

Note: *Devices that do not have a handler implemented can be extended to have one by using a similar device as an example.*

```
static hal_display_status_t
 HAL_DisplayDev_LcdifRk024hh2_InputNotify(const display_dev_t
 *receiver, void *data)
{
    hal_display_status_t error           =
 kStatus_HAL_DisplaySuccess;
    event_base_t eventBase               = *(event_base_t
 *)data;
    event_status_t event_response_status = kEventStatus_Ok;

    if (eventBase.eventId == kEventID_SetDisplayOutputSource)
    {
        event_common_t event             = *(event_common_t
 *)data;
        s_DisplayDev_Lcdif.cap.srcFormat =
 event.displayOutput.displayOutputSource;
        s_NewBufferSet                   = true;
        if (eventBase.respond != NULL)
        {
            eventBase.respond(eventBase.eventId,
 &event.displayOutput, event_response_status, true);
        }
        LOGI("[display_dev_inputNotify]:
 kEventID_SetDisplayOutputSource devID %d, srcFormat %d",
 receiver->id,
```

```
            event.displayOutput.displayOutputSource);
    }
    else if (eventBase.eventId ==
 kEventID_GetDisplayOutputSource)
    {
        display_output_event_t display;
        display.displayOutputSource =
 s_DisplayDev_Lcdif.cap.srcFormat;
        if (eventBase.respond != NULL)
        {
            eventBase.respond(eventBase.eventId, &display,
 event_response_status, true);
        }
        LOGI("[display_dev_inputNotify]:
 kEventID_GetDisplayOutputSource devID %d, srcFormat %d",
 receiver->id,
            display.displayOutputSource);
    }

    return error;
}
```

Some devices will not handle any events at all and will instead return `0` after performing no action.

```
hal_camera_status_t HAL_CameraDev_CsiGc0308_InputNotify(const
 camera_dev_t *dev, void *data)
{
    hal_camera_status_t ret = kStatus_HAL_CameraSuccess;

    return ret;
}
```

Alternatively, some devices which do not require an event handler may simply return a `NULL` pointer instead.

```
const static display_dev_operator_t s_DisplayDev_LcdifOps = {
    .init        = HAL_DisplayDev_Lcdifv2Rk055ah_Init,
    .deinit      = HAL_DisplayDev_Lcdifv2Rk055ah_Deinit,
    .start       = HAL_DisplayDev_Lcdifv2Rk055ah_Start,
    .blit        = HAL_DisplayDev_Lcdifv2Rk055ah_Blit,
    .inputNotify = NULL,
};
```

Managers will not call the `InputNotify` or other handler if that handler points to `NULL`.

A device's default handler whether for `InputNotify` events or `InferComplete` or otherwise can be overridden by an "app-specific" handler.

6.5.2.2 App-specific handlers

App-specific handlers are device handlers which are defined for a specific "app".

Not every device must implement an app-specific handler, but because default handlers are implemented using `WEAK` functions, any device which has a default event handler can have that handler overridden.

Note: *Some devices may not have implemented their default handlers using `WEAK` functions, but may be updated to do so in the future.*

For example, the IR + White LEDs may not require project-specific handlers because they will always react the same way to a `kEventID_SetConfig`/`kEventID_GetConfig` command. Alternatively, an application may wish to override and/or extend that default event handling behavior so that, for example, the LEDs increase in brightness when an "Add Face" event is received.

To help denote an app-specific handler, App-specific handlers start with the `APP` prefix. If an app-specific handler for a device exists, it can be found in `source/event_handlers/{APP_NAME}_{DEV_TYPE}_{DEV_NAME}.c`

7 Coffee machine

7.1 Introduction

This Coffee Machine application demonstrates the Coffee machine use-case with the following core functionalities:

- Coffee machine GUI with touch support
- Local voice command to control the use cases of Coffee machine
- Face recognition to store user's coffee preferences automatically

For leveraging the full computational power of the RT107H, the image has been split into two images that are running in parallel on the CM7 and CM4 cores. The Coffee Machine CM7 acts as an AI block, handling all the machine learning operations, such as face recognition and voice command. The operation has been optimized to obtain the best performance on this type of MCU. The Coffee Machine CM4 holds the user interaction (display, shell, buttons). The CM4 image is loaded into the memory by the CM7 core.

By default, i.MX RT117H boot from CM7. By fusing BT_CORE_SEL (Bit 12 in 0x960), the chip switches to CM4 as the main core. For more info on this topic, check AN13264.

The Coffee Machine uses the following HW components and peripherals:

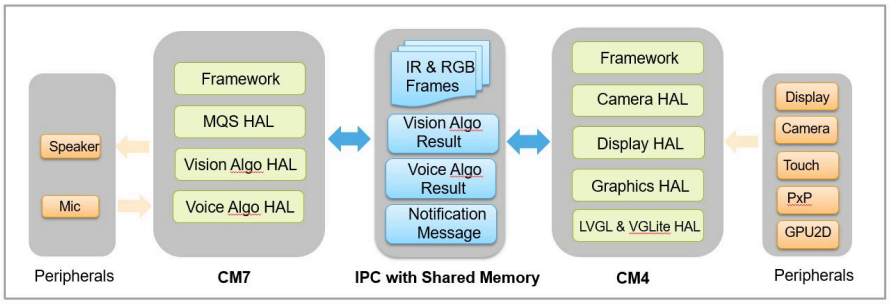
- 2 x PDM MIC - configured to work with 16 kHz sampling. The conversion to PCM is done in hardware using the PDM microphone interface.
- 16 KHz raw data to RT117x MQS HW peripheral that generates PWM data output.
- External filtering and coupling.
- Analog audio amplifier
- MIPI GC2145 Camera - configured to work with 600x800 resolution.
- LCDIFV2 Rocktech RK055MHD091 - configured to work at the HD resolution of 1280x720

To change this configuration, check HAL code and Section 9.1

It uses NXP's below core technologies:

- LVGL-based GUI
- Local voice command algorithm
- Face recognition algorithm
- Dual core architecture based on multicore manager (mcmgr) middleware component.

7.2 Architecture



CM7 (Vision & Voice algorithm accelerator):

- Framework
- Vision algorithm with VGA input frames
- Voice algorithm (AFE + ASR) with mic input
- MQS audio playback
- IPC communication with shared memory

CM4 (UI & System control unit):

- Framework
- CSI/MIPI Camera preview @VGA
- LVGL GUI @720p with VGLite 2D GPU acceleration
- Vision algorithm input frames color space conversion with PxP
- Touch panel input
- IPC communication with shared memory

Figure 29. Architecture diagram

7.3 Software block diagram

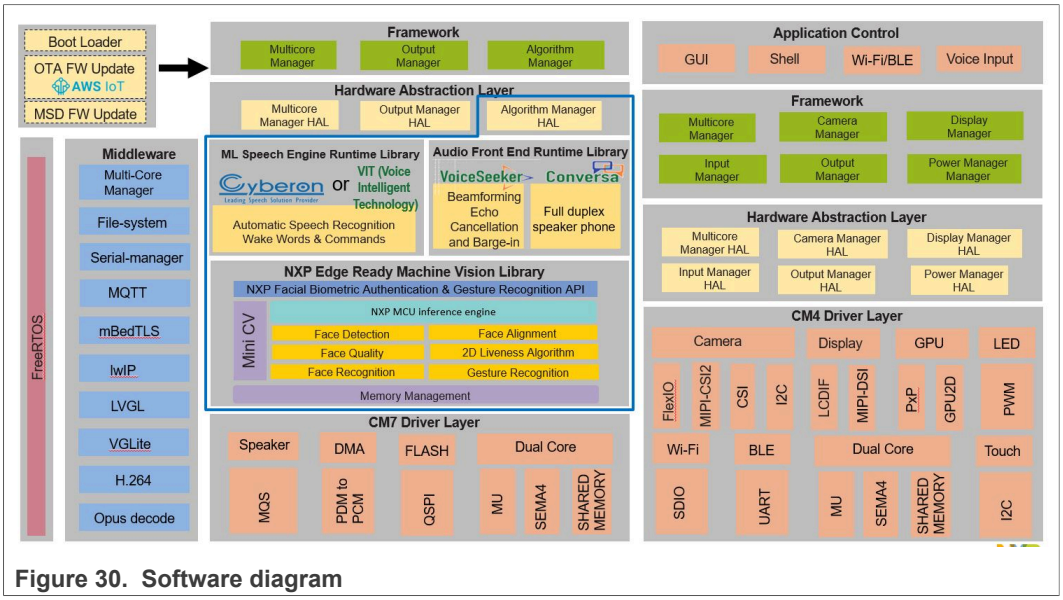


Figure 30. Software diagram

It includes two projects as below:

-Host CM7 project

-Slave CM4 project

Each project uses a two layer architecture containing the **Framework + HAL** layer, and the **Application** layer. For the details, refer to the documentation on each project.

7.4 Coffee machine CM7

This Coffee Machine CM7 host project runs on the CM7 core.

It is linked to flash with the combination of the CM4 project.

The CM7 was designed to focus on the vision and voice algorithms' processing to get the best performance.

7.5 Main functionalities

- Vision algorithm
- Voice algorithm
- Audio playback
- Microphone stream input
- Multicore communication
- Littlefs format filesystem

7.6 Boot sequence

The "main" entry of this project is located in the `../coffee_machine/cm7/source/sln_smart_tlhmi_cm7.cpp` file. The basic boot-up flow is:

- Initialize board level
- Initialize framework
- Register HAL devices
- Start the framework
- Start the freeRTOS scheduler

```
int main(void)
{
    /* init the board */
    APP_BoardInit();

    ...

    /* init the framework*/
    APP_InitFramework();

    /* register the hal devices*/
    APP_RegisterHalDevices();

    /* start the framework*/
    APP_StartFramework();

    vTaskStartScheduler();

    for (;;)
    {
    }
}
```

7.7 Board level initialization

The board-level initialization is implemented in the `APP_BoardInit()` entry which is located in `../coffee_machine/cm7/source/sln_smart_tlhmi_cm7.cpp`. Below is the main flow:

- Relocate vector table into RAM

- Configure MPU, Clock, and Pins
- Debug console with hardware semaphore initialization
- System time stamp start
- Load resources from flash into the share memory region
- Multicore manager init and boot slave core

```
void APP_BoardInit(void)
{
    BOARD_RelocateVectorTableToRam();

    BOARD_ConfigMPU();
    BOARD_InitBootPins();
    BOARD_InitBootClocks();

    BOARD_InitDebugConsole();
    Time_Init(1);

    APP_LoadResource();

    /* Initialize the HW Semaphore */
    SEMA4_Init(BOARD_SEM4_BASE);

#if defined(ENABLE_MASTER) && ENABLE_MASTER
    /* Initialize MCMGR before calling its API */
    (void)MCMGR_Init();

    /* Boot Secondary core application */
    (void)MCMGR_StartCore(kMCMGR_Core1, (void *)(char
 *)CORE1_BOOT_ADDRESS, 0, kMCMGR_Start_Synchronous);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */
}
```

7.8 Framework managers

The below framework managers are enabled on the cm7 side with the following priorities:

- Vision algorithm manager - P3
- Voice algorithm manager - P3
- Audio processing manager - P2
- Input manager - P1
- Output manager - P4
- Multicore manager - P0
- Flash device manager

Where P0 is the highest priority and P4 is the least prioritized.

Note: *Choosing the right priority for the manager is something that must be addressed based on the requirements. Our recommendation is to keep Vision manager equal to or less than Voice manager, or the audio sample can be lost.*

Refer to the framework documentation (`../framework/docs`) for a detailed description of these framework managers.

Note: *To prepare the environment for other framework managers, initialize the file system and application configuration first.*

```
int APP_InitFramework(void)
```

```
{
    int ret = 0;

    HAL_FLASH_DEV_REGISTER(Littlefs, ret);
    HAL_OutputDev_SmartTlhmiConfig_Init();

    FWK_MANAGER_INIT(VisionAlgoManager, ret);
    FWK_MANAGER_INIT(VoiceAlgoManager, ret);
    FWK_MANAGER_INIT(AudioProcessing, ret);
    FWK_MANAGER_INIT(OutputManager, ret);
    FWK_MANAGER_INIT(InputManager, ret);
#if defined(ENABLE_MASTER) && ENABLE_MASTER
    FWK_MANAGER_INIT(MulticoreManager, ret);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */

    return ret;
}

int APP_StartFramework(void)
{
    int ret = 0;

    FWK_MANAGER_START(VisionAlgoManager,
 VISION_ALGO_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(OutputManager,
 OUTPUT_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(AudioProcessing,
 AUDIO_PROCESSING_TASK_PRIORITY, ret);
    FWK_MANAGER_START(InputManager,
 INPUT_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(VoiceAlgoManager,
 VOICE_ALGO_MANAGER_TASK_PRIORITY, ret);
#if defined(ENABLE_MASTER) && ENABLE_MASTER
    FWK_MANAGER_START(MulticoreManager,
 MULTICORE_MANAGER_TASK_PRIORITY, ret);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */

    return ret;
}
```

7.9 Framework HAL devices

The enabled HAL devices are configured in the `../coffee_machine/cm7/board/board_define.h` file as shown below:

```
#define ENABLE_INPUT_DEV_PdmMic
#define ENABLE_AUDIO_PROCESSING_DEV_Afe
#define ENABLE_DSMT_ASR
#define ENABLE_OUTPUT_DEV_MqsAudio
#define ENABLE_OUTPUT_DEV_SmartTlhmiConfig
#define ENABLE_VISIONALGO_DEV_Oasis_CoffeeMachine
#define ENABLE_FLASH_DEV_Littlefs
#define ENABLE_FACEDB
#define USE_CAMERA_MipiGc2145
#if defined(ENABLE_MASTER) && ENABLE_MASTER
#define ENABLE_MULTICORE_DEV_MessageBuffer
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */
```

The registration of the enabled HAL devices is implemented in the `APP_RegisterHalDevices(...)` function which is located in `../coffee_machine/cm7/source/sln_smart_tlhmi_cm7.cpp`:

Note: *`APP_RegisterHalDevices(...)` must be called after the framework initialization `APP_InitFramework(...)` and before framework startup `APP_StartFramework(...)`.*

```
int APP_RegisterHalDevices(void)
{
    int ret = 0;

    HAL_OUTPUT_DEV_REGISTER(MqsAudio, ret);
    HAL_AUDIO_PROCESSING_DEV_REGISTER(Afe, ret);
    HAL_INPUT_DEV_REGISTER(PdmMic, ret);
    HAL_VOICEALGO_DEV_REGISTER(Asr, ret);
    HAL_VALGO_DEV_REGISTER(OasisCoffeeMachine, ret);
#if defined(ENABLE_MASTER) && ENABLE_MASTER
    HAL_MULTICORE_DEV_REGISTER(MessageBuffer, ret);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */
    HAL_INPUT_DEV_REGISTER(WiFiAWAM510, ret);

    return ret;
}
```

7.10 Logging

Both the CM7 and CM4 projects are leveraging the FreeRTOS logging library.

The FreeRTOS logging library code is located in the logging folder where you can find the detailed document located in `../coffee_machine/cm7/freertos/libraries/logging/README.md`.

The CM7 and CM4 share the low-level LPUART12 peripheral for the logging output. The hardware semaphore mechanism is used to guarantee the concurrence access of the LPUART12 peripheral. They share a low-level timer to get the unified timestamp of the logging information.

7.10.1 Log Task Init

The application calls the `xLoggingTaskInitialize(...)` API to create the logging task in the `main()` entry of this project and is located in `../coffee_machine/cm7/source/sln_smart_tlhmi_cm7.cpp`:

```
xLoggingTaskInitialize(LOGGING_TASK_STACK_SIZE,
 LOGGING_TASK_PRIORITY, LOGGING_QUEUE_LENGTH);
```

7.10.2 Log Macros

There are four kinds of logging that can be used in both cm7 and cm4, which you can find in `../framework/inc/fwk_log.h`

```
#ifndef LOGV
#define LOGV(fmt, args...) {implement...}
...
#endif
```

```
#ifndef LOGD
#define LOGD(fmt, args...) {implement...}
#endif

#ifndef LOGI
#define LOGI(fmt, args...) {implement...}
#endif

#ifndef LOGE
#define LOGE(fmt, args...) {implement...}
#endif
```

7.11 Coffee Machine database

The Coffee Machine application uses framework flash operations with the low-level littlefs file system to store the recognized user-faces database and user coffee information. The detailed usage API is located in files `../framework/hal/vision/hal_sln_facedb.h` and `../coffee_machine/cm7/source/hal_sln_coffeedb.h`. The face database and user coffee information database entry are bound together using the user id. The user id is a unique identifier on one device.

To make it easier for users to add their database with personal attributes, we split the face database from user database. The user should create something similar with `hal_sln_coffeedb.h` and add attributes like in the `coffee_attribute_t` structure.

7.11.1 Face recognition database usage

`g_facedb_ops` handles all kinds of face database operations.

```
typedef struct _facedb_ops
{
    facedb_status_t (*init)(uint16_t featureSize);
    facedb_status_t (*saveFace)(void);
    facedb_status_t (*addFace)(uint16_t id, char *name, void
 *face, int size);
    facedb_status_t (*delFaceWithId)(uint16_t id);
    facedb_status_t (*delFaceWithName)(char *name);
    facedb_status_t (*updNameWithId)(uint16_t id, char *name);
    facedb_status_t (*updFaceWithId)(uint16_t id, char *name,
 void *face, int size);
    facedb_status_t (*getFaceWithId)(uint16_t id, void
 **pFace);
    facedb_status_t (*getIdsAndFaces)(uint16_t *face_ids, void
 **pFace);
    facedb_status_t (*getIdWithName)(char *name, uint16_t *id);
    facedb_status_t (*genId)(uint16_t *new_id);
    facedb_status_t (*getIds)(uint16_t *face_ids);
    bool (*getSaveStatus)(uint16_t id);
    int (*getFaceCount)(void);
    char *(*getNameWithId)(uint16_t id);
} facedb_ops_t;

extern const facedb_ops_t g_facedb_ops;
```

7.11.2 User coffee information database usage

`g_coffedb_ops` handles all kinds of user information database operations.

```
ypedef enum _coffee_type
{
    Coffee_Espresso,
    Coffee_Americano,
    Coffee_Cappuccino,
    Caffee_Latte,
} coffee_type_t;

typedef enum _coffee_size
{
    Coffee_Small,
    Coffee_Medium,
    Coffee_Large,
} coffee_size_t;

typedef enum _coffee_strength
{
    Coffee_Soft,
    Coffee_Mild,
    Coffee_Strong,
} coffee_strength_t;

typedef struct _coffee_attribute
{
    uint16_t id;
    uint8_t type;
    uint8_t size;
    uint8_t strength;
    uint8_t reserved[16];
} coffee_attribute_t;

typedef struct _coffeedb_ops
{
    coffeedb_status_t (*init)(void);
    coffeedb_status_t (*deinit)(void);
    coffeedb_status_t (*addWithId)(uint16_t id,
 coffee_attribute_t *attr);
    coffeedb_status_t (*delWithId)(uint16_t id);
    coffeedb_status_t (*updWithId)(uint16_t id,
 coffee_attribute_t *attr);
    coffeedb_status_t (*getWithId)(uint16_t id,
 coffee_attribute_t *attr);
} coffeedb_ops_t;

extern const coffeedb_ops_t g_coffedb_ops;
```

7.12 Coffee machine CM4

This Coffee Machine CM4 slave project runs on the CM4 core.

It is linked to SDRAM and is embedded into the CM7 project.

The CM7 project handles the loading of this CM4 project into SDRAM and launching it.

7.13 Main functionalities

- Main GUI based on LVGL with Vglite graphics acceleration
- Camera with 2D PxP graphics acceleration
- Display for the camera preview and LVGL GUI
- USB shell
- LED indicator
- Multicore with messaging and shared memory communication

7.14 LVGL GUI screens and widgets

All the LVGL GUI screens and widgets are generated with NXP's GUI Guider tools.

Refer the GUI Guider home page for more information.

7.15 LVGL and Vglite library

The LVGL and Vglite components are directly ported from RT1170 SDK and we did not modify them in our solution.

Also the code for the LVGL GUI screens and widgets, which are generated by NXP's GUI guider, is not frequently changed.

To speed up the building of the whole project, we moved these components into one static library and linked the library into the CM4 application project.

This LVGL and Vglite library project is located in the `coffee_machine/lvgl_vglite_lib` folder.

7.16 Boot sequence

Below is the core boot up flow:

- Board level initialization
- Framework initialization
- HAL devices registration
- Framework startup
- FreeRTOS scheduler startup

The `main()` entry of this project is located in `../coffee_machine/cm4/source/sln_smart_tlhmi_cm4.cpp` file:

```
int main(void)
{
    /* init the board */
    APP_BoardInit();
    ...
    /* init the framework*/
    APP_InitFramework();

    /* register the hal devices*/
    APP_RegisterHalDevices();

    /* start the framework*/
    APP_StartFramework();

    vTaskStartScheduler();
```

```
    for (;;)
    {
    } /* should never get here */
    return 0;
}
```

7.17 Board level initialization

The board level initialization is implemented in the `APP_BoardInit()` entry which is located in the `../coffee_machine/cm4/source/sln_smart_tlhmi_cm4.cpp` file.

Below is the main flow:

- MPU, Clock, and Pins configuration
- Multicore manager init and slave startup
- Peripherals initialization

```
void APP_BoardInit()
{
    BOARD_ConfigMPU();
    BOARD_BootClockRUN();
    BOARD_InitBootPins();

#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    uint32_t startupData, i;
    mcmgr_status_t status;
    (void)MCMGR_Init();
    /* Get the startup data */
    do
    {
        status = MCMGR_GetStartupData(&startupData);
    } while (status != kStatus_MCMGR_Success);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    ...
    BOARD_MIPIPanelTouch_I2C_Init();
    BOARD_InitEDMA();
    Time_Init(1);
}
```

7.18 LVGL image resource and icon resource loading

All the LVGL images, data, and icon data are merged into one continuous binary block with the 64 Bytes aligned of each image/icon.

The cm7 loads this resource binary block into the dedicated memory region `res_sh_mem`.

The following two functions load each of these LVGL images and icons from this region during the boot.

Setup the LVGL images is implemented in `../coffee_machine/cm4/generated/gui_guider.c`:

```
void setup_imgs(unsigned char *base)
{
    brewing_animimg_brewingf01.data = (base + 0);
    brewing_animimg_brewingf02.data = (base + 120000);
    brewing_animimg_brewingf03.data = (base + 240000);
```

```
    ....
}
```

Load the icons (`../framework/hal/output/hal_output_ui_coffee_machine.c`):

```
void LoadIcons(void *base)
{
    s_Icons[ICON_PROGRESS_BAR] = (base + 0);

    s_Icons[ICON_VIRTUAL_FACE_BLUE]  = (base + 6720);
    s_Icons[ICON_VIRTUAL_FACE_GREEN] = (base + 364608);
    s_Icons[ICON_VIRTUAL_FACE_RED]   = (base + 722496);
    // Icons Total: 0x00107c40  1080384
}
```

7.19 Framework managers

The below framework managers are enabled on the cm4 side with the following priorities:

- Low-power manager
- Camera manager - P2
- Display manager - P2
- Multicore manager - P0
- Output manager - P1
- Input manager - P2

Where P0 is the highest priority and P3 is the least prioritized.

For a more detailed description of these framework managers, refer to the framework documentation (`../framework/docs/introduction.md`).

Framework initialization (`../coffee_machine/cm4/source/sln_smart_tlhmi_cm4.cpp`):

```
int APP_InitFramework(void)
{
    int ret = 0;

    FWK_MANAGER_INIT(LpmManager, ret);
    FWK_MANAGER_INIT(CameraManager, ret);
    FWK_MANAGER_INIT(DisplayManager, ret);
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    FWK_MANAGER_INIT(MulticoreManager, ret);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    FWK_MANAGER_INIT(OutputManager, ret);
    FWK_MANAGER_INIT(InputManager, ret);

    return ret;
}
```

Framework startup (`../coffee_machine/cm4/source/sln_smart_tlhmi_cm4.cpp`):

```
int APP_StartFramework(void)
{
    int ret = 0;
```

```
    FWK_MANAGER_START(LpmManager, 0, ret);
    FWK_MANAGER_START(CameraManager,
 CAMERA_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(DisplayManager,
 DISPLAY_MANAGER_TASK_PRIORITY, ret);
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    FWK_MANAGER_START(MulticoreManager,
 MULTICORE_MANAGER_TASK_PRIORITY, ret);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    FWK_MANAGER_START(OutputManager,
 OUTPUT_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(InputManager,
 INPUT_MANAGER_TASK_PRIORITY, ret);

    return ret;
}
```

7.20 Framework HAL devices

The enabled HAL devices are configured in the `../coffee_machine/cm4/board/board_define.h` file as shown below:

```
#define ENABLE_GFX_DEV_Pxp
#define ENABLE_DISPLAY_DEV_LVGLCoffeeMachine
#define ENABLE_CAMERA_DEV_MipiGc2145
#define ENABLE_OUTPUT_DEV_RgbLed
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
#define ENABLE_MULTICORE_DEV_MessageBuffer
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
#define ENABLE_INPUT_DEV_ShellUsb
#define ENABLE_OUTPUT_DEV_UiCoffeeMachine
#define ENABLE_LPM_DEV_Standby
```

The registration of the enabled HAL devices is implemented in the `APP_RegisterHalDevices(...)` function which is located in `../coffee_machine/cm4/source/sln_smart_tlhmi_cm4.cpp`:

Note: *`APP_RegisterHalDevices(...)` must be called after the framework initialization `APP_InitFramework(...)` and before framework startup `APP_StartFramework(...)`.*

```
int APP_RegisterHalDevices(void)
{
    int ret = 0;

    HAL_GFX_DEV_REGISTER(Pxp, ret);
    HAL_DISPLAY_DEV_REGISTER(LVGLCoffeeMachine, ret);
    HAL_CAMERA_DEV_REGISTER(MipiGc2145, ret);
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    HAL_MULTICORE_DEV_REGISTER(MessageBuffer, ret);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    HAL_OUTPUT_DEV_REGISTER(RgbLed, ret);
    HAL_INPUT_DEV_REGISTER(ShellUsb, ret);
    HAL_OUTPUT_DEV_REGISTER(UiCoffeeMachine, ret);
    HAL_LPM_DEV_REGISTER(Standby, ret);
#ifdef ENABLE_OUTPUT_DEV_AudioDump
    HAL_OUTPUT_DEV_REGISTER(AudioDump, ret);
```

```
#endif /* ENABLE_OUTPUT_DEV_AudioDump */
    /* Add new HAL device registrations here */

    return ret;
}
```

7.20.1 MipiGc2145 camera HAL device

This HAL device driver is located in `../framework/hal/camera/hal_camera_mipi_gc2145.c`.

Below is the configuration of this camera device located in `../coffee_machine/cm4/board/board_define.h`.

```
#ifdef ENABLE_CAMERA_DEV_MipiGc2145
#define CAMERA_DEV_MipiGc2145_BUFFER_COUNT 2
#define CAMERA_DEV_MipiGc2145_HEIGHT       600 // 720
#define CAMERA_DEV_MipiGc2145_WIDTH        800 // 1280
#define CAMERA_DEV_MipiGc2145_LEFT         0
#define CAMERA_DEV_MipiGc2145_TOP          0
#define CAMERA_DEV_MipiGc2145_RIGHT        799 // 1279
#define CAMERA_DEV_MipiGc2145_BOTTOM       599 // 719
#define CAMERA_DEV_MipiGc2145_ROTATE       kCWRotateDegree_0
#define CAMERA_DEV_MipiGc2145_FLIP         kFlipMode_None
#define CAMERA_DEV_MipiGc2145_SWAPBYTE     0
#define CAMERA_DEV_MipiGc2145_FORMAT
 kPixelFormat_YUV1P444_RGB
#define CAMERA_DEV_MipiGc2145_BPP          4
#endif /* ENABLE_CAMERA_DEV_MipiGc2145 */
```

7.20.2 PxP graphics HAL device

This HAL device driver is located in `../framework/hal/misc/hal_graphics_pxp.c`.

It represents the 2D graphics device to handle the 2D graphics operations.

7.20.3 LVGLCoffeeMachine display HAL device

This HAL device driver is located in `../framework/hal/display/hal_display_lvgl_coffeemachine.c`.

Below is the configuration of this display device located in `../coffee_machine/cm4/board/board_define.h`.

```
#ifdef ENABLE_DISPLAY_DEV_LVGLCoffeeMachine
#define DISPLAY_DEV_LVGLCoffeeMachine_BUFFER_COUNT 1
#define DISPLAY_DEV_LVGLCoffeeMachine_HEIGHT       640
#define DISPLAY_DEV_LVGLCoffeeMachine_WIDTH        480
#define DISPLAY_DEV_LVGLCoffeeMachine_StartX       80
#define DISPLAY_DEV_LVGLCoffeeMachine_StartY       50
#define DISPLAY_DEV_LVGLCoffeeMachine_LEFT         0
#define DISPLAY_DEV_LVGLCoffeeMachine_TOP          0
#define DISPLAY_DEV_LVGLCoffeeMachine_RIGHT        479
#define DISPLAY_DEV_LVGLCoffeeMachine_BOTTOM       639
#define DISPLAY_DEV_LVGLCoffeeMachine_ROTATE
 kCWRotateDegree_270
```

```
#define DISPLAY_DEV_LVGLCoffeeMachine_FORMAT
 kPixelFormat_RGB565
#ifdef ENABLE_CAMERA_DEV_MipiGc2145
#define DISPLAY_DEV_LVGLCoffeeMachine_SRCFORMAT
 kPixelFormat_YUV1P444_RGB
#else
#define DISPLAY_DEV_LVGLCoffeeMachine_SRCFORMAT
 kPixelFormat_UYVY1P422_RGB
#endif /* ENABLE_CAMERA_DEV_MipiGc2145 */
#define DISPLAY_DEV_LVGLCoffeeMachine_BPP 2
#endif /* ENABLE_DisplayDev_LVGLCoffeeMachine */
```

This LVGLCoffeeMachine-display-HAL-device launches the main LVGL task loop for the UI flashing.

```
static void _LvglTask(void *param)
{
#if LV_USE_LOG
    lv_log_register_print_cb(_PrintCb);
#endif /* LV_USE_LOG */

    lv_port_pre_init();
    lv_init();
    lv_port_disp_init();
    lv_port_indev_init();
    g_LvglInitialized = true;

    setup_imgs((unsigned char *)APP_LVGL_IMGS_BASE);
    setup_ui(&guider_ui);
    events_init(&guider_ui);
    custom_init(&guider_ui);
    while (1)
    {
        lv_task_handler();
        vTaskDelay(pdMS_TO_TICKS(5));
    }
}
```

It also handles the camera preview request from the framework in `HAL_DisplayDev_LVGLCoffeeMachine_Blit` function:

```
hal_display_status_t
 HAL_DisplayDev_LVGLCoffeeMachine_Blit(const display_dev_t
 *dev, void *frame, int width, int height)
{
    hal_display_status_t ret = kStatus_HAL_DisplaySuccess;
    LOGI("++HAL_DisplayDev_LVGLCoffeeMachine_Blit");

    // Show the new frame.
    void *lcdFrameAddr             = s_LcdBuffer[0];
    static int camerPreviewLayerOn = 0;

    // enable camera preview layer in screen with camera
 preview.
    if (lv_scr_act() == guider_ui.home && g_PreviewMode ==
 PREVIEW_MODE_CAMERA)
    {
        if (camerPreviewLayerOn == 0)
        {
```

```
            lv_enable_camera_preview(lcdFrameAddr, true);
            camerPreviewLayerOn = 1;
        }
    }
    else
    {
        // disable camera preview layer in screen without
 camera preview.
        if (camerPreviewLayerOn == 1)
        {
            camerPreviewLayerOn = 0;
            lv_enable_camera_preview(lcdFrameAddr, false);
        }
    }

    LOGI("--HAL_DisplayDev_LVGLCoffeeMachine_Blit");
    return ret;
}
```

7.20.4 UiCoffeeMachine UI output HAL device

This HAL device driver is located in `../framework/hal/output/hal_output_ui_coffee_machine.c`.

The whole UI state machine is driven by this output HAL device with the below event sources:

7.20.4.1 LVGL touch events

All the event callbacks of the LVGL widget are implemented in `../coffee_machine/cm4/generated/events_init.c`.

7.20.4.2 Vision and Voice algorithm inference result

The vision and voice inference result is notified by the output manager with below `HAL_OutputDev_UiCoffeeMachine_InferComplete` operator:

```
static hal_output_status_t
 HAL_OutputDev_UiCoffeeMachine_InferComplete(const output_dev_t
 *dev,output_algo_source_t source,void *inferResult)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;

    if (inferResult == NULL)
    {
        return error;
    }

    coffee_machine_screen_id_t currentScreenId =
 get_current_screen();

    if (currentScreenId == SCR_INVALID)
    {
        return error;
    }

    if (source == kOutputAlgoSource_Vision)
    {
```

```
        _InferComplete_Vision(dev, inferResult,
 currentScreenId);
    }
    else if (source == kOutputAlgoSource_Voice)
    {
        _InferComplete_Voice(dev, inferResult,
 currentScreenId);
    }

    return error;
}
```

7.20.5 RgbLed output HAL device

This HAL device driver is located in `../framework/hal/output/hal_output_rgb_led.c`.

It flashes the RGB led with different pattern according to the `HAL_OutputDev_RgbLed_InferComplete` or `HAL_OutputDev_RgbLed_InputNotify` operators below:

```
static hal_output_status_t
 HAL_OutputDev_RgbLed_InferComplete(const output_dev_t *dev,
 output_algo_source_t source, void *inferResult)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;
    uint32_t timerOn          = 0;

 _SetLedColor(APP_OutputDev_RgbLed_InferCompleteDecode(source,
 inferResult, &timerOn));

    if (timerOn != 0)
    {
        xTimerChangePeriod(OutputRgbTimer,
 pdMS_TO_TICKS(timerOn), 0);
    }
    return error;
}
```

```
static hal_output_status_t
 HAL_OutputDev_RgbLed_InputNotify(const output_dev_t *dev, void
 *data)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;

    _SetLedColor(APP_OutputDev_RgbLed_InputNotifyDecode(data));

    return error;
}
```

7.20.6 MessageBuffer multicore HAL device

This HAL device driver is located in `../framework/hal/misc/hal_multicore_messageBuffer.c`.

It handles the multicore messaging based on the multicore manager message buffer mechanism.

Refer the ../framework/docs/hal_devices/multicore.md file in the framework documentation for the detailed description of this HAL device.

7.20.7 ShellUsb input HAL device

This HAL device driver is located in ../framework/hal/input/hal_input_shell_cdc.c.

It populates one USB CDC device and generates the shell.

This driver only includes one **weak** shell command registration function as below:

```
__attribute__((weak)) void
 APP_InputDev_Shell_RegisterShellCommands(shell_handle_t
 shellContextHandle,  input_dev_t shellDev,
 input_dev_callback_t callback)
{
}
```

The application must overwrite this function to register the exactly shell commands.

The implementation of this overwritten function for the **Coffee Machine** application is in ../coffee_machine/cm4/source/event_handlers/smart_tlhmi_input_shell_commands.c:

```
void APP_InputDev_Shell_RegisterShellCommands(shell_handle_t
 shellContextHandle, input_dev_t *shellDev,
 input_dev_callback_t callback)
{
    s_InputCallback            = callback;
    s_SourceShell              = shellDev;
    s_ShellHandle              = shellContextHandle;
    s_FrameworkRequest.respond = _FrameworkEventsHandler;

    if (s_ThingName == NULL)
    {
        APP_GetHexUniqueID(&s_ThingName);
    }

    SHELL_RegisterCommand(shellContextHandle,
 SHELL_COMMAND(version));
    ...
}
```

7.20.8 Standby LPM HAL device

This HAL device driver is located in ../framework/hal/misc/hal_lpm_standby.c.

Refer to ../framework/docs/hal_devices/low_power.md in the framework documentation for the detailed description of this LPM device.

This standby HAL device implements the standby mode of this application. The backlight is turned off and the main display layer is disabled.

```
static void _EnterStandbyMode(void)
{
    LOGD("[Standby] Enter standby mode");
```

```
    BOARD_BacklightControl(0);
    lv_enable_ui_preview(0);
}
```

7.21 Logging

Both the CM7 and CM4 projects are leveraging the FreeRTOS logging library.

The FreeRTOS logging library code is located in the logging folder where you can find the detailed document `../coffee_machine/cm4/freertos/libraries/logging/README.md`.

The CM7 and CM4 share the low-level LPUART12 peripheral for the logging output. The hardware semaphore mechanism is used to guarantee the concurrence access of theLPUART12 peripheral.

They share a low-level timer to get the unified timestamp of the logging information.

7.21.1 Logging Task Init

Application calls `xLoggingTaskInitialize(...)` API to create the logging task in the `main()` entry of this project is located in the `../coffee_machine/cm4/source/sln_smart_tlhmi_cm4.cpp` file:

```
xLoggingTaskInitialize(LOGGING_TASK_STACK_SIZE,
 LOGGING_TASK_PRIORITY, LOGGING_QUEUE_LENGTH);
```

7.21.2 Logging Macros

The logging Macros are defined in `../framework/inc/fwk_log.h`.

All the modules must use these unified logging Macros for logging.

```
#ifndef LOGV
#define LOGV(fmt, args...) {implement...}
...
#endif

#ifndef LOGD
#define LOGD(fmt, args...) {implement...}
#endif

#ifndef LOGI
#define LOGI(fmt, args...) {implement...}
#endif

#ifndef LOGE
#define LOGE(fmt, args...) {implement...}
#endif
```

8 Elevator

8.1 Introduction

This Elevator application demonstrates the elevator use-case with the core functionalities:

- Elevator GUI with touch support
- Local voice command to control the use cases of the elevator
- Face recognition to store user's floor information automatically

For leveraging the full computational power of the RT107H, the image has been split into two images that are running in parallel on the CM7 and CM4 cores. The Elevator CM7 acts as an AI block, handling all the machine learning operations, such as face recognition and voice command. The operation has been optimized to obtain the best performance on this type of MCU. Elevator CM4 holds the user interaction (display, shell, buttons). The CM4 image is loaded into memory by the CM7 core.

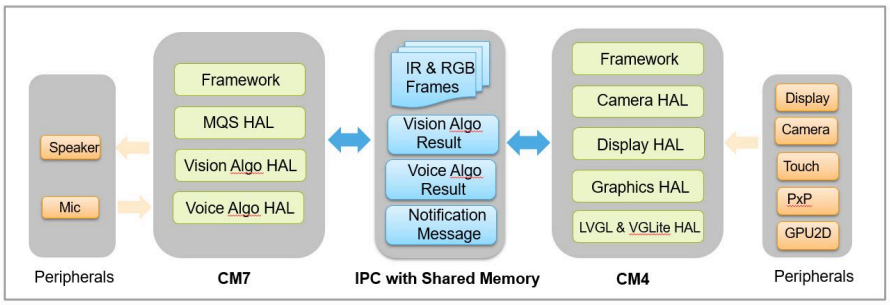
By default, i.MX RT117H is boot from CM7. By fusing BT_CORE_SEL (Bit 12 in 0x960), the chip switches to CM4 as the main core. For more information on this topic, check AN13264.

The Elevator Application uses the following HW components and peripherals:

- 2 x PDM MIC - configured to work with 16 kHz sampling. The conversion to PCM is done in hardware using the PDM microphone interface.
- 16 KHz raw data to RT117x MQS HW peripheral that generates PWM data output.
- External filtering and coupling.
- Analog audio amplifier.
- MIPI GC2145 Camera - configured to work at 600x800 resolution.
- LCDIFV2 Rocktech RK055MHD091 - configured to work at HD resolution of 1280x720.

To change this configuration, check HAL code and Section 9.1

8.2 Architecture



CM7 (Vision & Voice algorithm accelerator):

- Framework
- Vision algorithm with VGA input frames
- Voice algorithm (AFE + ASR) with mic input
- MQS audio playback
- IPC communication with shared memory

CM4 (UI & System control unit):

- Framework
- CSI/MIPI Camera preview @VGA
- LVGL GUI @720p with VGLite 2D GPU acceleration
- Vision algorithm input frames color space conversion with PxP
- Touch panel input
- IPC communication with shared memory

Figure 31. Architecture diagram

8.3 Software block diagram

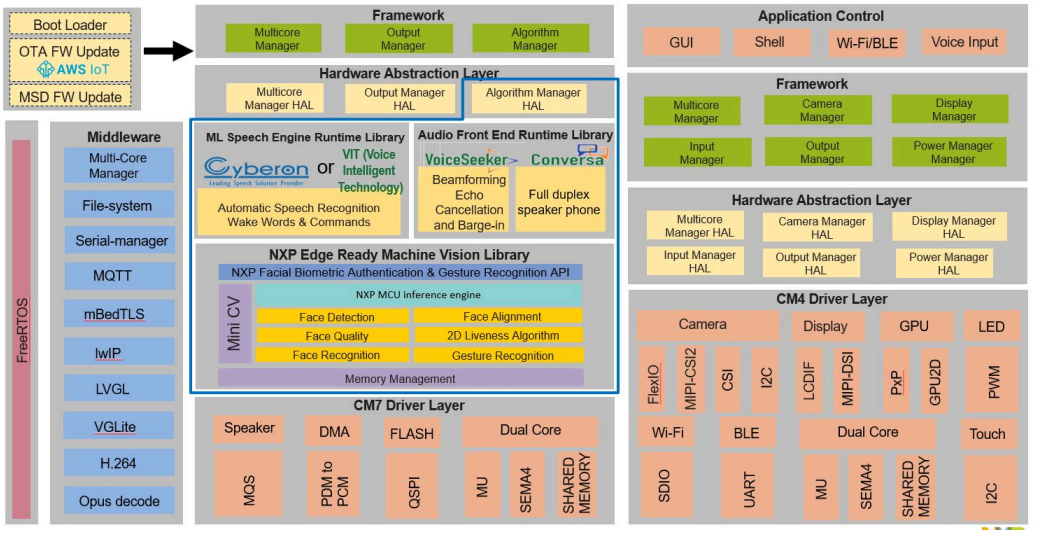


Figure 32. Software diagram

It includes two projects as below:

-Host CM7 project

-Slave CM4 project

Each project uses two-layer architecture containing the **Framework + HAL** layer, and the **Application** layer. For more information, refer to the documentation on each project..

8.4 Elevator CM7

This Elevator CM7 host project runs on the CM7 core. It is linked to flash with the combination of the CM4 project. CM7 was designed to focus on the vision and voice algorithms' processing to get the best performance.

8.5 Main functionalities

- Vision algorithm
- Voice algorithm
- Audio playback
- Microphone stream input
- Multicore communication
- Elevator database

8.6 Boot sequence

The "main" entry of this project is in the `../elevator/CM7/source/sln_smart_tlhmi_CM7.cpp` file. The basic boot up flow is:

- Initialize board level
- Initialize framework
- Register HAL devices
- Start the framework

- Start the freeRTOS scheduler

```
int main(void)
{
    /* init the board */
    APP_BoardInit();

    ...

    /* init the framework*/
    APP_InitFramework();

    /* register the hal devices*/
    APP_RegisterHalDevices();

    /* start the framework*/
    APP_StartFramework();

    vTaskStartScheduler();

    for (;;)
    {
    }
}
```

8.7 Board level initialization

The board-level initialization is implemented in the `APP_BoardInit()` entry which is located in `../elevator/CM7/source/sln_smart_tlhmi_CM7.cpp`. Below is the main flow:

- Relocate vector table into RAM
- Configure MPU, Clock, and Pins
- Debug console with hardware semaphore initialization
- System time stamp start
- Load resource from flash into share memory region
- Multicore manager init and boot slave core

```
void APP_BoardInit(void)
{
    BOARD_RelocateVectorTableToRam();

    BOARD_ConfigMPU();
    BOARD_InitBootPins();
    BOARD_InitBootClocks();

    BOARD_InitDebugConsole();
    Time_Init(1);

    APP_LoadResource();

    /* Initialize the HW Semaphore */
    SEMA4_Init(BOARD_SEM4_BASE);

#if defined(ENABLE_MASTER) && ENABLE_MASTER
    /* Initialize MCMGR before calling its API */
    (void)MCMGR_Init();
```

```
    /* Boot Secondary core application */
    (void)MCMGR_StartCore(kMCMGR_Core1, (void *)(char
 *)CORE1_BOOT_ADDRESS, 0, kMCMGR_Start_Synchronous);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */
}
```

8.8 Framework managers

The below framework managers are enabled in the CM7 side with the following priorities:

- Vision algorithm manager - P3
- Voice algorithm manager - P3
- Audio processing manager - P2
- Input manager - P1
- Output manager - P4
- Multicore manager - P0

Refer to the framework documentation (`../framework/docs`) for a detailed description of these framework managers.

Note: *To prepare the environment for other framework managers, initialize the file system and application configuration first.*

```
int APP_InitFramework(void)
{
    int ret = 0;

    HAL_FLASH_DEV_REGISTER(Littlefs, ret);
    HAL_OutputDev_SmartTlhmiConfig_Init();

    FWK_MANAGER_INIT(VisionAlgoManager, ret);
    FWK_MANAGER_INIT(VoiceAlgoManager, ret);
    FWK_MANAGER_INIT(AudioProcessing, ret);
    FWK_MANAGER_INIT(OutputManager, ret);
    FWK_MANAGER_INIT(InputManager, ret);
#if defined(ENABLE_MASTER) && ENABLE_MASTER
    FWK_MANAGER_INIT(MulticoreManager, ret);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */

    return ret;
}

int APP_RegisterHalDevices(void)
{
    int ret = 0;

    HAL_OUTPUT_DEV_REGISTER(MqsAudio, ret);
    HAL_AUDIO_PROCESSING_DEV_REGISTER(Afe, ret);
    HAL_INPUT_DEV_REGISTER(PdmMic, ret);
    HAL_VOICEALGO_DEV_REGISTER(Asr, ret);
    HAL_VALGO_DEV_REGISTER(OasisElevator, ret);
#if defined(ENABLE_MASTER) && ENABLE_MASTER
    HAL_MULTICORE_DEV_REGISTER(MessageBuffer, ret);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */

    return ret;
}
```

```
int APP_StartFramework(void)
{
    int ret = 0;

    FWK_MANAGER_START(VisionAlgoManager,
 VISION_ALGO_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(OutputManager,
 OUTPUT_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(AudioProcessing,
 AUDIO_PROCESSING_TASK_PRIORITY, ret);
    FWK_MANAGER_START(InputManager,
 INPUT_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(VoiceAlgoManager,
 VOICE_ALGO_MANAGER_TASK_PRIORITY, ret);
    // FWK_MANAGER_START(CameraManager,
 CAMERA_MANAGER_TASK_PRIORITY, ret);
#if defined(ENABLE_MASTER) && ENABLE_MASTER
    FWK_MANAGER_START(MulticoreManager,
 MULTICORE_MANAGER_TASK_PRIORITY, ret);
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */

    return ret;
}
```

8.9 Framework HAL devices

The enabled HAL devices are configured in the `../elevator/CM7/board/board_define.h` file as shown below:

```
#define ENABLE_INPUT_DEV_PdmMic
#define ENABLE_AUDIO_PROCESSING_DEV_Afe
#define ENABLE_DSMT_ASR
#define ENABLE_OUTPUT_DEV_MqsAudio
#define ENABLE_OUTPUT_DEV_SmartTlhmiConfig
#if defined(ENABLE_MASTER) && ENABLE_MASTER
#define ENABLE_MULTICORE_DEV_MessageBuffer
#endif /* defined(ENABLE_MASTER) && ENABLE_MASTER */
```

8.10 Logging

Both CM7 and CM4 projects are leveraging the FreeRTOS logging library.

The FreeRTOS logging library code is located in the logging folder where you can find the detailed document `../coffee_machine/cm7/freertos/libraries/logging/README.md`.

The CM7 and CM4 share low-level LPUART12 peripheral for the logging output. The hardware semaphore mechanism is used to guarantee the concurrence access of LPUART12 peripheral. And they also share low-level timer to get the unified timestamp of the logging information.

8.10.1 Log task init

The application calls the `xLoggingTaskInitialize(...)` API to create the logging task in the `main()` entry of this project and is located in `elevator/cm7/source/sln_smart_tlhmi_cm7.cpp`:

```
xLoggingTaskInitialize(LOGGING_TASK_STACK_SIZE,
 LOGGING_TASK_PRIORITY, LOGGING_QUEUE_LENGTH);
```

8.10.2 Log usage

There are four kinds of logging that can use both CM7 and CM4, that you can find in `../framework/inc/fwk_log.h`.

```
#ifndef LOGV
#define LOGV(fmt, args...) {implement...}
...
#endif

#ifndef LOGD
#define LOGD(fmt, args...) {implement...}
#endif

#ifndef LOGI
#define LOGI(fmt, args...) {implement...}
#endif

#ifndef LOGE
#define LOGE(fmt, args...) {implement...}
#endif
```

8.11 Elevator database

The Elevator application uses framework flash operation with low-level littlefs file system to store the recognized user-faces database and user elevator information. The detailed usage API is located in files `../framework/vision/hal_sln_facedb.h` and `../source/hal_sln_elevatordb.h`. The face database and elevator user information database entry are bound together using user id. The user id is a unique identifier on one device.

To make it easier for users to add their own database with personal attributes, we split the face database from user database. The user must create something similar with `hal_sln_elevator.h` and add attributes like in the `elevator_attr_t` structure. If the purpose is to extend the current elevator database, use a reserved field from the structure below.

8.11.1 Face recognize database usage

`g_facedb_ops` handles all kinds of face database operation.

```
typedef struct _facedb_ops
{
    facedb_status_t (*init)(uint16_t featureSize);
    facedb_status_t (*saveFace)(void);
```

```
    facedb_status_t (*addFace)(uint16_t id, char *name, void
 *face, int size);
    facedb_status_t (*delFaceWithId)(uint16_t id);
    facedb_status_t (*delFaceWithName)(char *name);
    facedb_status_t (*updNameWithId)(uint16_t id, char *name);
    facedb_status_t (*updFaceWithId)(uint16_t id, char *name,
 void *face, int size);
    facedb_status_t (*getFaceWithId)(uint16_t id, void
 **pFace);
    facedb_status_t (*getIdsAndFaces)(uint16_t *face_ids, void
 **pFace);
    facedb_status_t (*getIdWithName)(char *name, uint16_t *id);
    facedb_status_t (*genId)(uint16_t *new_id);
    facedb_status_t (*getIds)(uint16_t *face_ids);
    bool (*getSaveStatus)(uint16_t id);
    int (*getFaceCount)(void);
    char *(*getNameWithId)(uint16_t id);
} facedb_ops_t;

extern const facedb_ops_t g_facedb_ops;
```

8.11.2 Elevator user information database usage

`g_elevatordb_ops` handles all kinds of user information database operation.

```
typedef struct _elevator_attribute
{
    uint16_t id;
    uint32_t floor;
    uint8_t reserved[16];
} elevator_attr_t;

typedef struct _elevatordb_ops
{
    elevatordb_status_t (*init)(void);
    elevatordb_status_t (*deinit)(void);
    elevatordb_status_t (*addWithId)(uint16_t id,
 elevator_attr_t *attr);
    elevatordb_status_t (*delWithId)(uint16_t id);
    elevatordb_status_t (*updWithId)(uint16_t id,
 elevator_attr_t *attr);
    elevatordb_status_t (*getWithId)(uint16_t id,
 elevator_attr_t *attr);
} elevatordb_ops_t;

extern const elevatordb_ops_t g_elevatordb_ops;
```

8.12 Elevator CM4

This Elevator CM4 slave project runs on the CM4 core.

It is linked to SDRAM and will be embedded into the CM7 project.

The CM7 project handles the loading of this CM4 project into SDRAM and launching it.

8.13 Main functionalities

- Main GUI based on LVGL with Vglite graphics acceleration

- Camera with 2D PxP graphics acceleration
- Display for the camera preview and LVGL GUI
- USB shell
- LED indicator
- Multicore with messaging and shared memory communication

8.14 LVGL GUI screens and widgets

All the LVGL GUI screens and widgets are generated with NXP's GUI Guider tools.

Refer to the GUI Guider home page for more detailed information.

8.15 LVGL and Vglite library

LVGL and Vglite components are directly ported from RT1170 SDK where we did not modify them in our solution.

The code for LVGL GUI screens and widgets, which are generated by NXP's GUI guider, is not frequently changed.

To speed up the building of the whole project, we moved these components into one static library and linked the library to the CM4 application project.

This LVGL and Vglite library project is located in the `../elevator/lvgl_vglite_lib` folder.

8.16 Boot sequence

Below is the core boot-up flow:

- Board level initialization
- Framework initialization
- HAL devices registration
- Framework startup
- FreeRTOS scheduler startup

The `main()` entry of this project is located in the `../elevator/cm4/source/sln_smart_tlhmi_cm4.cpp` file:

```
int main(void)
{
    /* init the board */
    APP_BoardInit();
    ...
    /* init the framework*/
    APP_InitFramework();

    /* register the hal devices*/
    APP_RegisterHalDevices();

    /* start the framework*/
    APP_StartFramework();

    vTaskStartScheduler();
    for (;;)
    {
    } /* should never get here */
```

```
    return 0;
}
```

8.17 Board level initialization

The board level initialization is implemented in the `APP_BoardInit()` entry which is located in the `../elevator/cm4/source/sln_smart_tlhmi_cm4.cpp` file.

Below is the main flow:

- MPU, Clock, and Pins configuration
- Multicore manager init and slave startup
- Peripherals initialization

```
void APP_BoardInit()
{
    BOARD_ConfigMPU();
    BOARD_BootClockRUN();
    BOARD_InitBootPins();

#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    uint32_t startupData, i;
    mcmgr_status_t status;
    (void)MCMGR_Init();
    /* Get the startup data */
    do
    {
        status = MCMGR_GetStartupData(&startupData);
    } while (status != kStatus_MCMGR_Success);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    ...
    BOARD_MIPIPanelTouch_I2C_Init();
    BOARD_InitEDMA();
    Time_Init(1);
}
```

8.18 LVGL image resource loading

All the LVGL images, data, and icon data are merged into one continuous binary block with the 64 Bytes aligned of each image/icon.

The cm7 loads this resource binary block into the dedicated memory region `res_sh_mem`.

The below two function loads each of these LVGL images and icons from this region during the boot.

Setup the LVGL images is implemented in `../elevator/cm4/generated/gui_guider.c`:

```
void setup_imgs(void *base)
{
    _TLHMI_Elevator_Main_Screen_1280x720.data
 = (base + 0);
    _TLHMI_Elevator_Virtual_Face_Blue_180x180.data
 = (base + 2764800);
    _TLHMI_Elevator_Button_Call_alpha_90x90.data
 = (base + 2862016);
```

```
    ....
}
```

8.19 Framework managers

The below framework managers are enabled on the cm4 side:

- Low-power manager
- Camera manager
- Display manager
- Multicore manager
- Output manager
- Input manager

Refer to `framework/docs/introduction.md` for a more detailed description of these framework managers.

Framework initialization (`../elevator/cm4/source/sln_smart_tlhmi_cm4.cpp`):

```
int APP_InitFramework(void)
{
    int ret = 0;

    FWK_MANAGER_INIT(LpmManager, ret);
    FWK_MANAGER_INIT(CameraManager, ret);
    FWK_MANAGER_INIT(DisplayManager, ret);
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    FWK_MANAGER_INIT(MulticoreManager, ret);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    FWK_MANAGER_INIT(OutputManager, ret);
    FWK_MANAGER_INIT(InputManager, ret);

    return ret;
}
```

Framework startup (`../elevator/cm4/source/sln_smart_tlhmi_cm4.cpp`):

```
int APP_StartFramework(void)
{
    int ret = 0;

    FWK_MANAGER_START(LpmManager, 0, ret);
    FWK_MANAGER_START(CameraManager,
 CAMERA_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(DisplayManager,
 DISPLAY_MANAGER_TASK_PRIORITY, ret);
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    FWK_MANAGER_START(MulticoreManager,
 MULTICORE_MANAGER_TASK_PRIORITY, ret);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    FWK_MANAGER_START(OutputManager,
 OUTPUT_MANAGER_TASK_PRIORITY, ret);
    FWK_MANAGER_START(InputManager,
 INPUT_MANAGER_TASK_PRIORITY, ret);

    return ret;
```

```
}
```

8.20 Framework HAL devices

The enabled HAL devices are configured in the `../elevator/cm4/board/board_define.h` file as below:

```
#define ENABLE_GFX_DEV_Pxp
#define ENABLE_DISPLAY_DEV_LVGLElevator
#define ENABLE_CAMERA_DEV_MipiGc2145
#define ENABLE_OUTPUT_DEV_RgbLed
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
#define ENABLE_MULTICORE_DEV_MessageBuffer
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
#define ENABLE_INPUT_DEV_ShellUsb
#define ENABLE_OUTPUT_DEV_UiElevator
#define ENABLE_LPM_DEV_Standby
```

The registration of the enabled HAL devices is implemented in the `APP_RegisterHalDevices(...)` function, which is located in `../elevator/cm4/source/sln_smart_tlhmi_cm4.cpp`

Note: *The `APP_RegisterHalDevices(...)` must be called after the framework initialization `APP_InitFramework(...)` and before framework startup `APP_StartFramework(...)`.*

```
int APP_RegisterHalDevices(void)
{
    int ret = 0;

    HAL_GFX_DEV_REGISTER(Pxp, ret);
    HAL_DISPLAY_DEV_REGISTER(LVGLElevator, ret);
    HAL_CAMERA_DEV_REGISTER(MipiGc2145, ret);
#if defined(ENABLE_SLAVE) && ENABLE_SLAVE
    HAL_MULTICORE_DEV_REGISTER(MessageBuffer, ret);
#endif /* defined(ENABLE_SLAVE) && ENABLE_SLAVE */
    HAL_OUTPUT_DEV_REGISTER(RgbLed, ret);
    HAL_INPUT_DEV_REGISTER(ShellUsb, ret);
    HAL_OUTPUT_DEV_REGISTER(UiElevator, ret);
    HAL_LPM_DEV_REGISTER(Standby, ret);
    /* Add new HAL device registrations here */

    return ret;
}
```

8.20.1 MipiGc2145 camera HAL device

This HAL device driver is located in `../framework/hal/camera/hal_camera_mipi_gc2145.c`

Below is the configuration of this camera device, which is located in `../elevator/cm4/board/board_define.h`

```
#ifdef ENABLE_CAMERA_DEV_MipiGc2145
#define CAMERA_DEV_MipiGc2145_BUFFER_COUNT 2
#define CAMERA_DEV_MipiGc2145_HEIGHT       600 // 720
#define CAMERA_DEV_MipiGc2145_WIDTH        800 // 1280
```

```
#define CAMERA_DEV_MipiGc2145_LEFT          0
#define CAMERA_DEV_MipiGc2145_TOP           0
#define CAMERA_DEV_MipiGc2145_RIGHT         799 // 1279
#define CAMERA_DEV_MipiGc2145_BOTTOM        599 // 719
#define CAMERA_DEV_MipiGc2145_ROTATE        kCWRotateDegree_0
#define CAMERA_DEV_MipiGc2145_FLIP          kFlipMode_None
#define CAMERA_DEV_MipiGc2145_SWAPBYTE      0
#define CAMERA_DEV_MipiGc2145_FORMAT
 kPixelFormat_YUV1P444_RGB
#define CAMERA_DEV_MipiGc2145_BPP           4
#endif /* ENABLE_CAMERA_DEV_MipiGc2145 */
```

8.20.2 PxP graphics HAL device

This HAL device driver is located in `../framework/hal/misc/hal_graphics_pxp.c`

It represents the 2D graphics device to handle the 2D graphics operations.

8.20.3 LVGLElevator display HAL device

This HAL device driver is located in `../framework/hal/display/hal_display_lvgl_elevator.c`

Below is the configuration of this display device, which is located in the `../elevator/cm4/board/board_define.h`

```
#ifdef ENABLE_DISPLAY_DEV_LVGLElevator
#define DISPLAY_DEV_LVGLElevator_BUFFER_COUNT 1
#define DISPLAY_DEV_LVGLElevator_HEIGHT       640
#define DISPLAY_DEV_LVGLElevator_WIDTH        480
#define DISPLAY_DEV_LVGLElevator_StartX       80
#define DISPLAY_DEV_LVGLElevator_StartY       50
#define DISPLAY_DEV_LVGLElevator_LEFT         0
#define DISPLAY_DEV_LVGLElevator_TOP          0
#define DISPLAY_DEV_LVGLElevator_RIGHT        479
#define DISPLAY_DEV_LVGLElevator_BOTTOM       639
#define DISPLAY_DEV_LVGLElevator_ROTATE
 kCWRotateDegree_270
#define DISPLAY_DEV_LVGLElevator_FORMAT
 kPixelFormat_RGB565
#ifdef ENABLE_CAMERA_DEV_MipiGc2145
#define DISPLAY_DEV_LVGLElevator_SRCFORMAT
 kPixelFormat_YUV1P444_RGB
#else
#define DISPLAY_DEV_LVGLElevator_SRCFORMAT
 kPixelFormat_UYVY1P422_RGB
#endif
#define DISPLAY_DEV_LVGLElevator_BPP 2
#endif /* ENABLE_DisplayDev_LVGLElevator */
```

This LVGLElevator display HAL device launches the main LVGL task loop for the UI flashing.

```
static void _LvglTask(void *param)
{
#if LV_USE_LOG
    lv_log_register_print_cb(_PrintCb);
```

```
#endif /* LV_USE_LOG */

    lv_port_pre_init();
    lv_init();
    lv_port_disp_init();
    lv_port_indev_init();
    g_LvglInitialized = true;

    setup_imgs((unsigned char *)APP_LVGL_IMGS_BASE);
    setup_ui(&guider_ui);
    events_init(&guider_ui);
    custom_init(&guider_ui);
    while (1)
    {
        lv_task_handler();
        vTaskDelay(pdMS_TO_TICKS(5));
    }
}
```

8.20.4 UiElevator UI output HAL device

This HAL device driver is located in `../framework/hal/output/hal_output_ui_elevator.c`

The whole UI state machine is driven by this output HAL device with the below event sources:

8.20.4.1 LVGL touch events

All the event callbacks of the LVGL widget are implemented in `../elevator/cm4/generated/events_init.c`

8.20.4.2 Vision and Voice algorithm inference result

The vision and voice inference result is notified by the output manager with below `HAL_OutputDev_UiElevator_InferComplete` operator:

```
static hal_output_status_t
 HAL_OutputDev_UiElevator_InferComplete(const output_dev_t
 *dev,output_algo_source_t source,void *inferResult)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;

    if (inferResult == NULL)
    {
        return error;
    }

    if (source == kOutputAlgoSource_Vision)
    {
        _InferComplete_Vision(dev, inferResult);
    }
    else if (source == kOutputAlgoSource_Voice)
    {
        _InferComplete_Voice(dev, inferResult);
    }

    return error;
```

```
}
```

8.20.5 RgbLed output HAL device

This HAL device driver is located in `../framework/hal/output/hal_output_rgb_led.c`

It flashes the RGB led with different pattern according to the `HAL_OutputDev_RgbLed_InferComplete` or `HAL_OutputDev_RgbLed_InputNotify` operators below:

```
static hal_output_status_t
 HAL_OutputDev_RgbLed_InferComplete(const output_dev_t *dev,
 output_algo_source_t source, void *inferResult)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;
    uint32_t timerOn          = 0;

 _SetLedColor(APP_OutputDev_RgbLed_InferCompleteDecode(source,
 inferResult, &timerOn));

    if (timerOn != 0)
    {
        xTimerChangePeriod(OutputRgbTimer,
 pdMS_TO_TICKS(timerOn), 0);
    }
    return error;
}
```

```
static hal_output_status_t
 HAL_OutputDev_RgbLed_InputNotify(const output_dev_t *dev, void
 *data)
{
    hal_output_status_t error = kStatus_HAL_OutputSuccess;

    _SetLedColor(APP_OutputDev_RgbLed_InputNotifyDecode(data));

    return error;
}
```

8.20.6 MessageBuffer multicore HAL device

This HAL device driver is located in `../framework/hal/misc/hal_multicore_messageBuffer.c`

It handles multicore messaging based on the multicore manager message buffer mechanism.

For the detailed description of this HAL device, refer to `../framework/docs/hal_devices/multicore.md` in the framework documentation.

8.20.7 ShellUsb input HAL device

This HAL device driver is located in `../framework/hal/input/hal_input_shell_cdc.c`

It populates one USB CDC device and generates the shell.

This driver only includes one **weak** shell command registration function as below:

```
__attribute__((weak)) void
 APP_InputDev_Shell_RegisterShellCommands(shell_handle_t
 shellContextHandle,  input_dev_t shellDev,
 input_dev_callback_t callback)
{
}
```

The application must overwrite this function to register the exactly shell commands.

You can find the implementation of this overwritten function for the **Elevator** application from `../elevator/cm4/source/event_handlers/smart_tlhmi_input_shell_commands.c`:

```
void APP_InputDev_Shell_RegisterShellCommands(shell_handle_t
 shellContextHandle, input_dev_t *shellDev,
 input_dev_callback_t callback)
{
    s_InputCallback            = callback;
    s_SourceShell              = shellDev;
    s_ShellHandle              = shellContextHandle;
    s_FrameworkRequest.respond = _FrameworkEventsHandler;

    if (s_ThingName == NULL)
    {
        APP_GetHexUniqueID(&s_ThingName);
    }

    SHELL_RegisterCommand(shellContextHandle,
 SHELL_COMMAND(version));
    ...
}
```

8.20.8 Standby LPM HAL device

This HAL device driver is located in `../framework/hal/misc/hal_lpm_standby.c`.

For the detailed description of this LPM device, refer to `../framework/docs/hal_devices/low_power.md` in the framework documentation.

This standby HAL device implements the standby mode of this application. The backlight is turned off and the main display layer is disabled.

```
static void _EnterStandbyMode(void)
{
    LOGD("[Standby] Enter standby mode");
    BOARD_BacklightControl(0);
    lv_enable_ui_preview(0);
}
```

8.21 Logging

Both the CM7 and CM4 projects are leveraging the FreeRTOS logging library.

The FreeRTOS logging library code is located in the logging folder where you can find the detailed document `../elevator/cm4/freertos/libraries/logging/README.md`

The CM7 and CM4 share low-level LPUART12 peripheral for the logging output. The hardware semaphore mechanism is used to guarantee the concurrence access of LPUART12 peripheral.

They share a low-level timer to get the unified timestamp of the logging information.

8.21.1 Logging task init

Application calls `xLoggingTaskInitialize(...)` API to create the logging task in the `main()` entry of this project is located in the `../elevator/cm4/source/sln_smart_tlhmi_cm4.cpp`:

```
xLoggingTaskInitialize(LOGGING_TASK_STACK_SIZE,
 LOGGING_TASK_PRIORITY, LOGGING_QUEUE_LENGTH);
```

8.21.2 Logging macros

The logging Macros are defined in `../framework/inc/fwk_log.h`.

All the modules must use these unified logging Macros for logging.

```
#ifndef LOGV
#define LOGV(fmt, args...) {implement...}
...
#endif

#ifndef LOGD
#define LOGD(fmt, args...) {implement...}
#endif

#ifndef LOGI
#define LOGI(fmt, args...) {implement...}
#endif

#ifndef LOGE
#define LOGE(fmt, args...) {implement...}
#endif
```

9 Customization

9.1 How to develop a user application

9.1.1 Introduction

We created a template to demonstrate the Smart HMI application with LVGL GUI, Face Recognition, and Far-Field Voice Recognition AI/ML algorithms integrated.

You can leverage this template to quickly build your own applications:

-- Create your fancy GUI with an open-source LVGL library

-- Use Face Recognition as the user identity

-- Use Far-Field Voice Recognition as touchless interface

9.1.2 Build the LVGL GUI

LVGL is a free and open-source embedded graphic library with features that enable you to create embedded GUIs with intuitive graphical elements, beautiful visual effects, and a low memory footprint. The complete graphic framework includes various widgets for you to use in the creation of your GUI, and supports more advanced functions such as animations and anti-aliasing.

To learn more about LVGL, visit https://lvgl.io/

9.1.2.1 Design and create the GUI with NXP's free GUI Guider tool

GUI Guider is a user-friendly graphical user interface development tool from NXP that enables rapid development of high quality displays with the open-source LVGL graphics library. GUI Guider's drag-and-drop editor makes it easy to utilize the many features of LVGL such as widgets, animations, and styles to create a GUI with minimal or no coding at all.

To learn more about GUI Guider, visit https://www.nxp.com/design/software/development-software/gui-guider:GUI-GUIDER

Refer to our full GUI Guider project for Coffee Machine and Elevator demo below:

-- Coffee Machine `coffee_machine/gui_guider/coffee_machine.guiguider`

-- Elevator `elevator/gui_guider/elevator.guiguider`

9.1.2.2 Integrate your generated LVGL GUI code

The whole GUI code is running in the CM4 core and is built into the CM4 project.

By default, the function below is the main entry of the whole LVGL GUI that is located in your generated GUI code `../coffee_machine/cm4/generated/gui_guider.c`

```
void setup_ui(lv_ui *ui)
{
    setup_scr_standby(ui);
    lv_scr_load(ui->standby);
}
```

We created the LVGL Display HAL device to handle the LVGL initialization and the GUI launch. The `void setup_ui(lv_ui *ui)` is called in this HAL device, therefore you must replace the "generated" folder with your GUI code in the CM4 project, and the whole UI be launched during the start-up.

Refer LVGL Display HAL device implementation for the Coffee Machine demo and Elevator demo as below:

-- Coffee Machine `../framework/hal/display/hal_display_lvgl_coffeemachine.c`

-- Elevator `../framework/hal/display/hal_display_lvgl_elevator.c`

To learn more about Display HAL device, refer to `../framework/docs/hal_devices/display.md`

9.1.3 Build the phoneme-based voice recognition model

We enabled Far-Field Voice Recognition by phoneme-based Automatic Speech Recognition (ASR) engine. NXP partners with Cyberon for generating phoneme-based voice engines. For more information on how to build your phoneme-based voice engine, refer to `../voice/dsmt_instructions.md`.

We created the Voice Algorithm HAL device to handle the whole voice recognition.

Refer the Voice algorithm HAL device implementation for the Coffee Machine demo and Elevator demo as below:

-- Coffee Machine and Elevator Voice Algorithm HAL `../framework/hal/voice/hal_voice_algo_dsmt_asr.c`

-- Coffee Machine voice recognition models `../coffee_machine/cm7/local_voice/>local_voice` folder

-- Elevator voice recognition models `../elevator/cm7/local_voice>local_voice` folder

The voice recognition is running in CM7 and the whole Voice algorithm HAL device and voice models are built into CM7 project.

9.1.4 Bind the user's profile data with face recognition

The face recognition algorithm and face feature database have been implemented. You can use them as the user identity for your application.

They are all running on CM7 and are built into the CM7 project.

You can refer the implementation for the Coffee Machine demo and Elevator demo as below:

-- Face recognition algorithm for Coffee Machine `../framework/hal/vision/hal_vision_algo_oasis_coffeemachine.c`

-- Face recognition algorithm for Elevator `../framework/hal/vision/hal_vision_algo_oasis_elevator.c`

-- Face feature database `../framework/hal/vision/hal_sln_facedb.c`

We have implemented the framework flash APIs based on the little fs. You can define the user's profile data structure and implement the user's profile database base on these well-defined APIs.

You can refer the user's profile database implementation for the Coffee Machine demo and Elevator demo as below:

-- User's profile data base for Coffee Machine `../coffee_machine/cm7/source/hal_sln_coffeedb.c`

-- User's profile data base for Elevator `../elevator/cm7/source/hal_sln_elevatordb.c`

9.1.5 Implement the use case flow for your application

We created the UI Output HAL device to handle the APP use case flow. It controls the face recognition HAL device, voice recognition HAL device and the LVGL UI. The inference results from face recognition HAL device and voice recognition HAL device are posted into this output device.

To learn more about Output HAL device, refer to `../framework/docs/hal_devices/output.md`

You can refer the UI Output HAL device implementation for the Coffee Machine demo and Elevator demo as below:

-- Coffee Machine `../framework/hal/output/hal_output_ui_coffee_machine.c`

-- Elevator `../framework/hal/output/hal_output_ui_elevator.c`

9.2 Application resource build

9.2.1 Introduction

This section is focused on the use of the resource build tool, which can easily generate the required binary file from the user's source and description file.

9.2.2 Source files

The source files are placed in the `resource` folder of each project. The files generally contain three types of images, icons, and sounds, placed in the corresponding folders respectively. The build tool has certain requirements on the code format of the source files, as shown below.

9.2.2.1 Format of Image file

The image files are generated by `GUI-Guide` and automatically saved in the `gui_guide/generated/images` folder. The installation package for GUI-Guide V1.3.0 can be found at this address: GUI-Guide Tool.

There are two types of image files, one is big-endian and the other is little-endian, so only data of the required image type must be generated.

```
const uint8_t _Americano_250x250_map[] = {
#if LV_COLOR_DEPTH == 16 && LV_COLOR_16_SWAP == 0
  0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0xff,
 0xff, ...
#endif
#if LV_COLOR_DEPTH == 16 && LV_COLOR_16_SWAP != 0
  0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0xff,
 0xff, ...
#endif
};
```

9.2.2.2 Format of Icon file

The format of the icon file must be consistent with the following.

```
#ifndef _NXP_LOGO_H_
#define _NXP_LOGO_H_

#define NXP_LOGO_W 240
#define NXP_LOGO_H 86

static const unsigned short nxp_logo_240x86[] = {
```

```
  0xFDA4, 0xFD83, 0xFD83, 0xFD83, 0xFD83, 0xFD83, 0xFD83,
 0xFD83, ...
};
#endif /* _NXP_LOGO_H_ */
```

9.2.2.3 Format of Sound file

The format of the sound file must be consistent with the following.

```
/***********************************
* Written by WAVToCode
* FileName:   Can_I_help.h
* Signed:   Yes
* No. of channels: 1
* No. of samples: 14211
* Bits/Sample:  16
***********************************/

#define WW_DETECT_EN_LEN  sizeof(ww_detect_en)

short ww_detect_en[14211] = {
  0,   0,   0,   1,  -2,   2,  -1,   0, /* 0-7 */
  1,  -1,   1,  -2,   2,  -1,   0,   1, /* 8-15 */
  ...
  2,   0,  -1}; /* 14208-14210 */
```

Note: *The sound files can be generated using open-source Audacity and WavToCode, and the sampling rate is set to 16,000 hz, with 16 bits per channel.*

9.2.3 Description file

Each application has a description file to contain all the resources to be built. The resource build tool reads this description file to build the final resource binary file.

Here is the basic design for the description file. Each line represents a source file to build, and the format is `<Type File_Name>`.

- Type: **image/icon/sound**
- File_Name: the path of source files is **relative** to the build tool.

```
// resource_build coffee_machine_resource.txt
/*
image ../../coffee_machine/resource/images/
brewing_animimg_brewingf01.c
icon ../../coffee_machine/resource/icons/process_bar_240x14.h
sound ../../coffee_machine/resource/sounds/common/
confirm_tone.h
...
*/
```

9.2.4 Resource build tool

Provide `bat` for Windows and `bash` for Linux to invoke the corresponding build tools to generate resource binary file and offset table file from the source files and the description file.

The build tools are placed in the `tools/resource_build` folder. The bat and bash tools are placed in the `resource` folder of each project.

Modify the description file, binary file name, and image file format in bat and bash tools to generate the required binary file.

- description file: the name of the description file;
- binary file name (optional): the name of the generated binary file, default is "`resources.bin`";
- image file format (optional): 0/1, default is 0 (LV_COLOR_16_SWAP == 0);

```
<build tool path> <description file> <binary file name> <image
 file format>
```

Generate the binary file by running `project.bat` as administrator in Windows or executing `bash project.sh` in Linux command shell.

9.3 Cyberon DSMT speech model instructions

9.3.1 Getting started with phoneme-based voice engine tool

NXP partners with Cyberon for generating phoneme-based voice engines. The voice engine supports speaker-independent recognition and there is no need to collect speech data for training specific commands in advance. With the generation tool, you can create your own custom voice engine by simply typing text.

The TLHMI solution supports Far-Field voice recognition enabled by phoneme-based Automatic Speech Recognition (ASR) engine, digital signal processing (DSP), and audio front end (AFE). This chapter describes:

1. How to create or modify phoneme-based voice engine in various languages
2. How to integrate a generated voice engine into TLHMI solution software
3. Guide for voice recognition improvement
4. Technical specification information of the voice engine

9.3.2 Installation

The generation tool requires you to log in. To get access to the tool, contact NXP (local-commands@nxp.com) with the following information.

1. Company name
2. User's name
3. User's e-mail address
4. Physical address (MAC address) of PC's network interface.

We reach out to let you know when the account is created. The installation package for Cyberon DSpotter Modeling Tool (DSMT) V2 can be found at this address: DSpotter Modeling Tool

The installation package contains the following items.

1. Cyberon DSpotter Modeling Tool (DSMT) V2
2. DSpotter Offline Test Tool V2
3. DSpotter Online Test Tool V2 You are required to install all of them. While installing the modeling tool, you are prompted to install the offline / online test tools.

Install the Cyberon DSpotter GarbGen Tool from this address: DSpotter GarbGen Tool

9.3.3 Load the project template

Note: *This guide focuses on exemplifying how DSMT tool works by using the Coffee Machine demo template for English language.*

First, copy the `coffee_machine/oob_demo_en.dsmt` file in the MCUXpresso project at the location below.

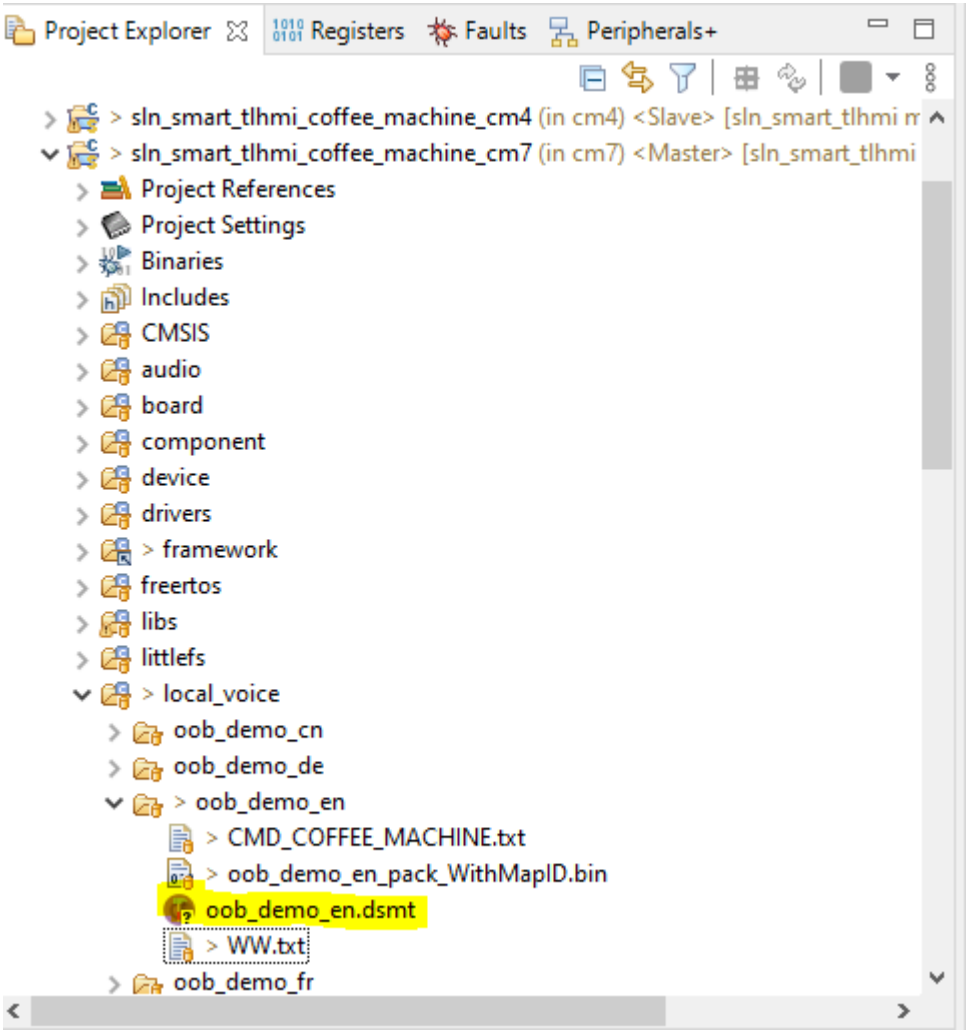


Figure 33. Coffee Machine DSMT en template

Ensure that the DSpotter Modeling Tool (DSMT) is installed. To load the project template:

1. Launch the application.
2. A window prompts you to enter your credentials. Log in with your credentials.
3. Click **File > Load Project**

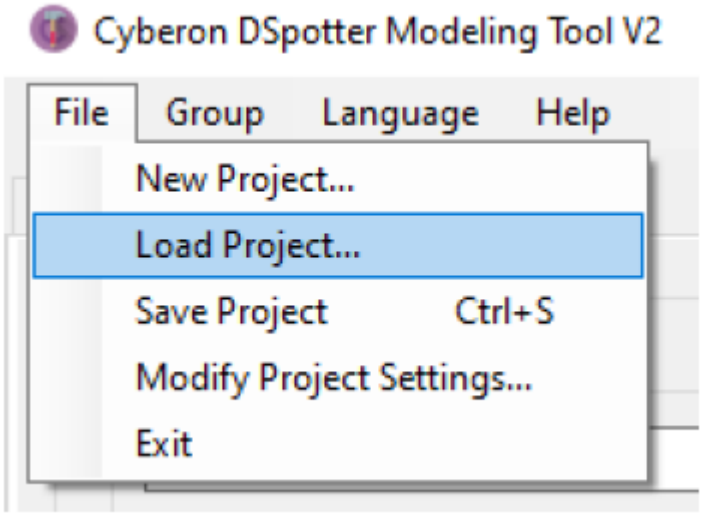


Figure 34. Load DSMT template

4. Open the DSMT project previously copied into the workspace.

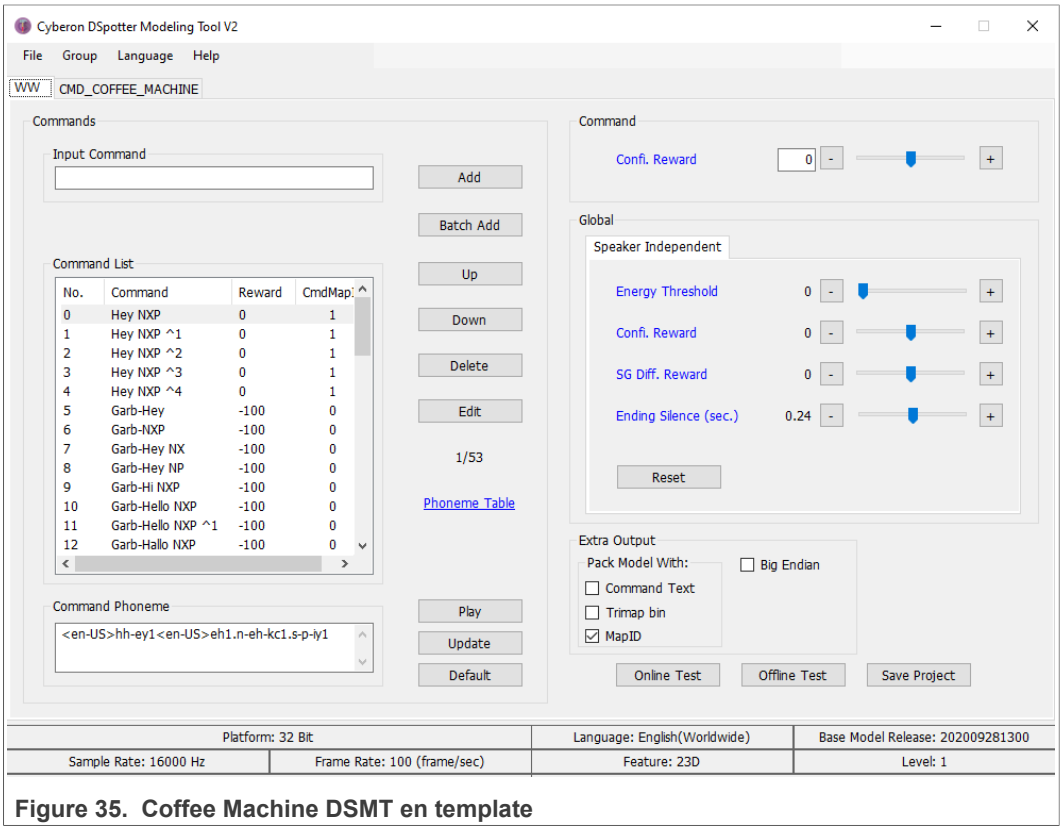


Figure 35. Coffee Machine DSMT en template

9.3.4 Add a new command into the Coffee Machine demo

Note: *For an easier demonstration, we remove the garbage words here. Delete all entries after "Deregister" command.*

To add a new command into the Coffee Machine demo:

1. Click CMD_COFFEE_MACHINE tab on the DSMT tool.
2. Type a new command, then press on "Add". For example, "Mochaccino". This command is inserted at the end, as shown below (this is the reason for which we have deleted the garbage words, we would have needed to press the "Up" button for more than 300 times to bring the new command on the position from the image below.)

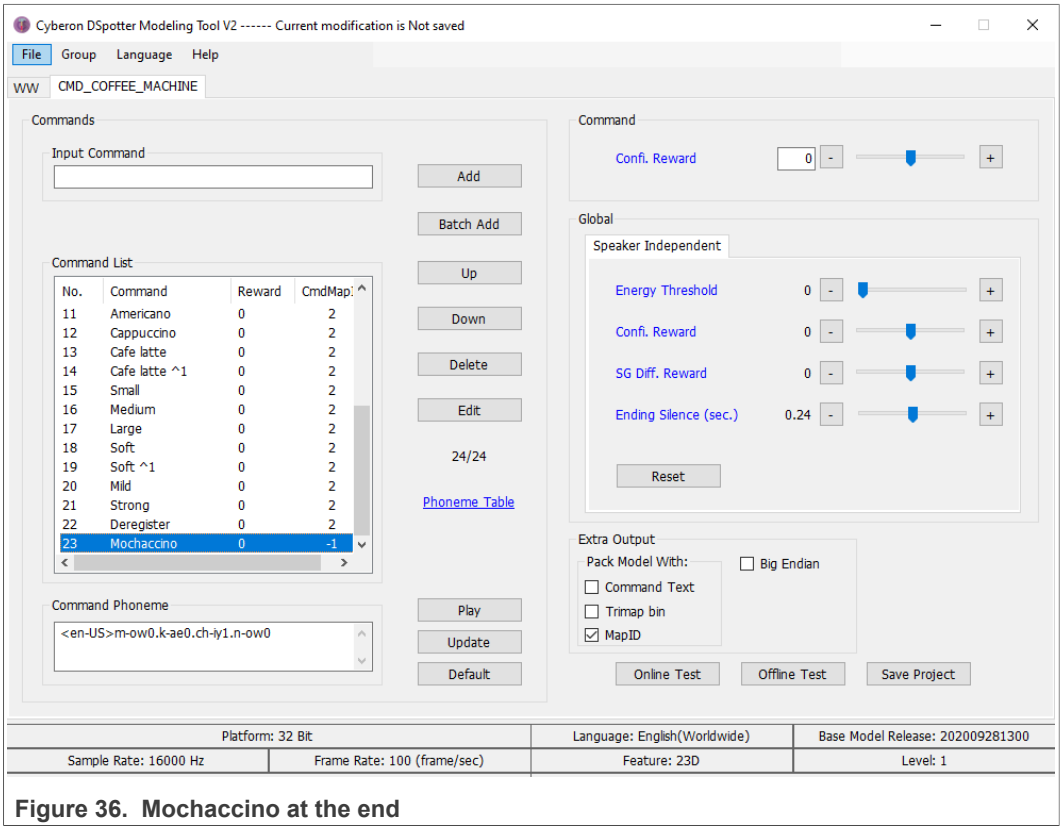


Figure 36. Mochaccino at the end

3. Edit `CmdMapId` from -1 to the one used for the other commands of this command group, which is 2. To do this, double-click the command.

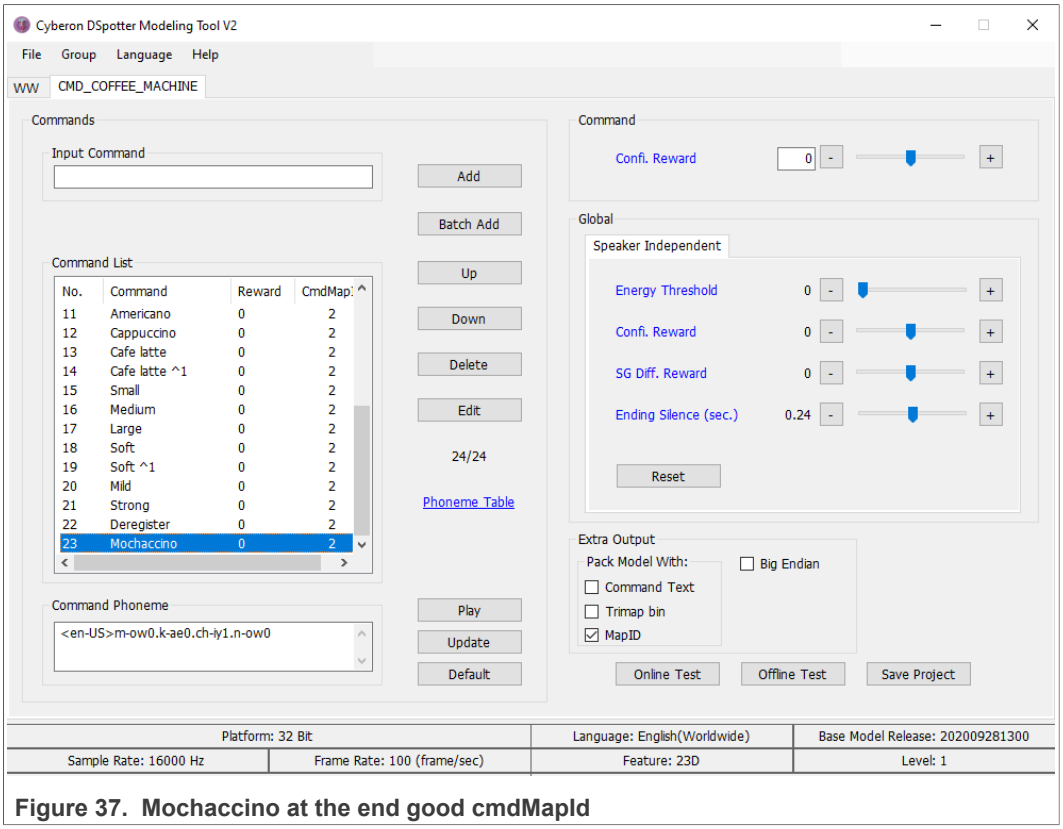


Figure 37. Mochaccino at the end good cmdMapId

4. Save the project (by pressing Ctrl + S or clicking the **Save Project** button.)

9.3.4.1 Integrate the voice engine in MCUXpresso project

If the DSMT template was copied into the folder mentioned above, the binary containing the speech model is automatically updated when you save the DSMT project.

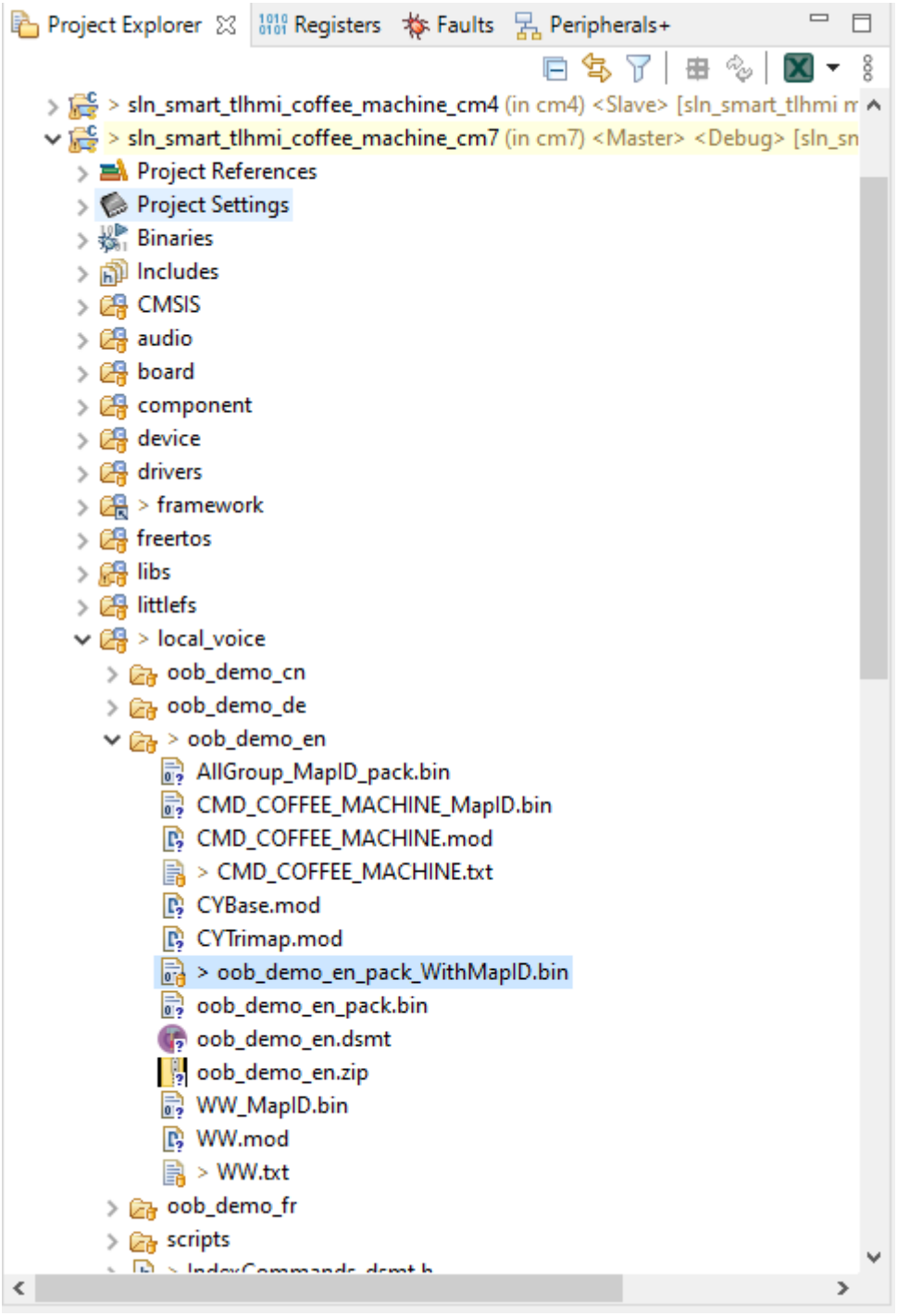


Figure 38. Updated dsmt binary

We now must update a few things in the firmware to add support for the new command. For the sake of the example, we do the same action on the GUI for Mochaccino as we are doing for Cappuccino.

1. Update `IndexCommands_dsmt.h`. Increase the total number of commands by 1 and also add an action in `action_coffee_machine_en`, specifying that we have the same action as for Cappuccino.

IndexCommands_dsmt.h | IndexToCommand_en.h

```
18 /* These defines are currently used only for displaying in
19 #define NUMBER_OF_COFFEE_MACHINE_CMDS_EN 24
20 #define NUMBER_OF_COFFEE_MACHINE_CMDS_CN 20
21 #define NUMBER_OF_COFFEE_MACHINE_CMDS_FR 16
22 #define NUMBER_OF_COFFEE_MACHINE_CMDS_DE 15
23
24 unsigned int action_coffee_machine_en[] = {
25     kCoffeeMachineActionStart,      // "Start"
26     kCoffeeMachineActionCancel,     // "Cancel"
27     kCoffeeMachineActionConfirm,    // "Confirm"
28     kCoffeeMachineActionConfirm,    // "Confirm"
29     kCoffeeMachineActionConfirm,    // "Confirm"
30     kCoffeeMachineActionConfirm,    // "Confirm"
31     kCoffeeMachineActionConfirm,    // "Confirm"
32     kCoffeeMachineActionConfirm,    // "Confirm"
33     kCoffeeMachineActionConfirm,    // "Confirm"
34     kCoffeeMachineActionConfirm,    // "Confirm"
35     kCoffeeMachineActionEspresso,   // "Espresso"
36     kCoffeeMachineActionAmericano,  // "Americano"
37     kCoffeeMachineActionCappuccino, // "Cappuccino"
38     kCoffeeMachineActionLatte,      // "Cafe latte"
39     kCoffeeMachineActionLatte,      // "Cafe latte"
40     kCoffeeMachineActionSmall,      // "Small"
41     kCoffeeMachineActionMedium,     // "Medium"
42     kCoffeeMachineActionLarge,      // "Large"
43     kCoffeeMachineActionSoft,       // "Soft"
44     kCoffeeMachineActionSoft,       // "Soft"
45     kCoffeeMachineActionMild,       // "Mild"
46     kCoffeeMachineActionStrong,     // "Strong"
47     kCoffeeMachineActionDeregister, // "Deregister"
48     kCoffeeMachineActionCappuccino, // "Mochaccino"
49 };
```

Figure 39. Update IndexCommands_dsmt

2. Update `IndexToCommand_en.h`. Add a string representation of the new command.

IndexCommands_dsmt.h | IndexToCommand_en.h

```
1  #ifndef INDEXCOMMANDS_EN_H_
2  #define INDEXCOMMANDS_EN_H_
3
4  char *ww_en[] = {"Hey NXP", "Hey NXP", "Hey NXP", "Hey NXP", "Hey NXP"};
5
6  char *cmd_coffee_machine_en[] = {"Start",      "Cancel",     "Confirm",    "Confirm",    "Confirm",
7                                   "Confirm",    "Confirm",    "Confirm",    "Confirm",    "Confirm",
8                                   "Espresso",   "Americano",  "Capuccino",  "Cafe latte",
9                                   "Cafe latte", "Small",      "Medium",     "Large",      "Soft",
10                                  "Soft",       "Mild",       "Strong",     "Deregister", "Mochaccino"};
11
12 #endif /* INDEXCOMMANDS_EN_H_ */
```

Figure 40. Update IndexToCommand_en

3. Build and flash the project. You must now be able to see the command "Mochaccino" being detected and also triggering the same action as the "Cappuccino" command.

9.3.5 Add a new language into the Coffee Machine demo

1. Open DSMT and login

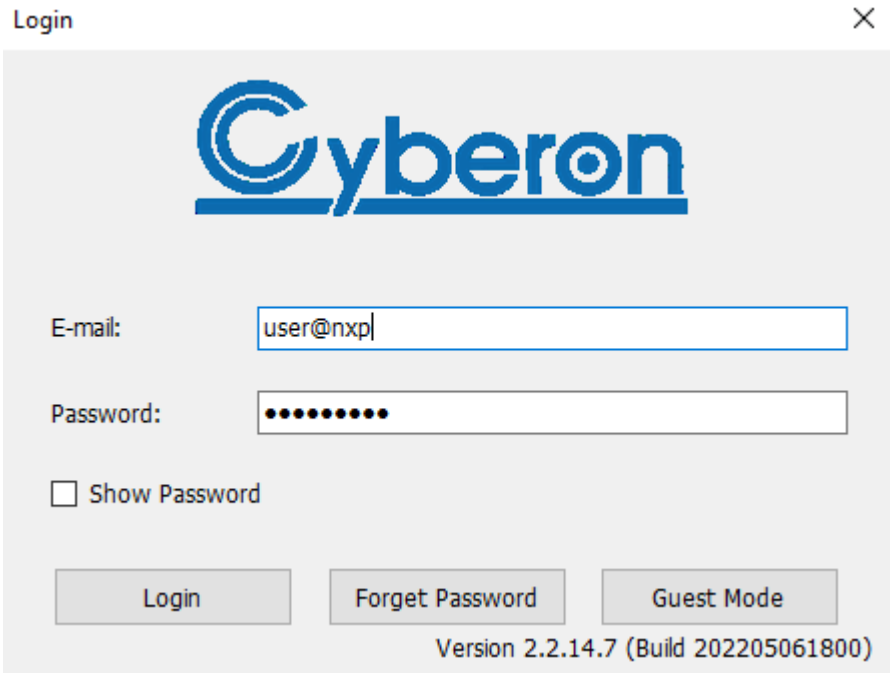


Figure 41. Dsmt login

2. **File -> New Project**. Use the name `oob_demo_it`, choose the Italian language. Click **OK**.

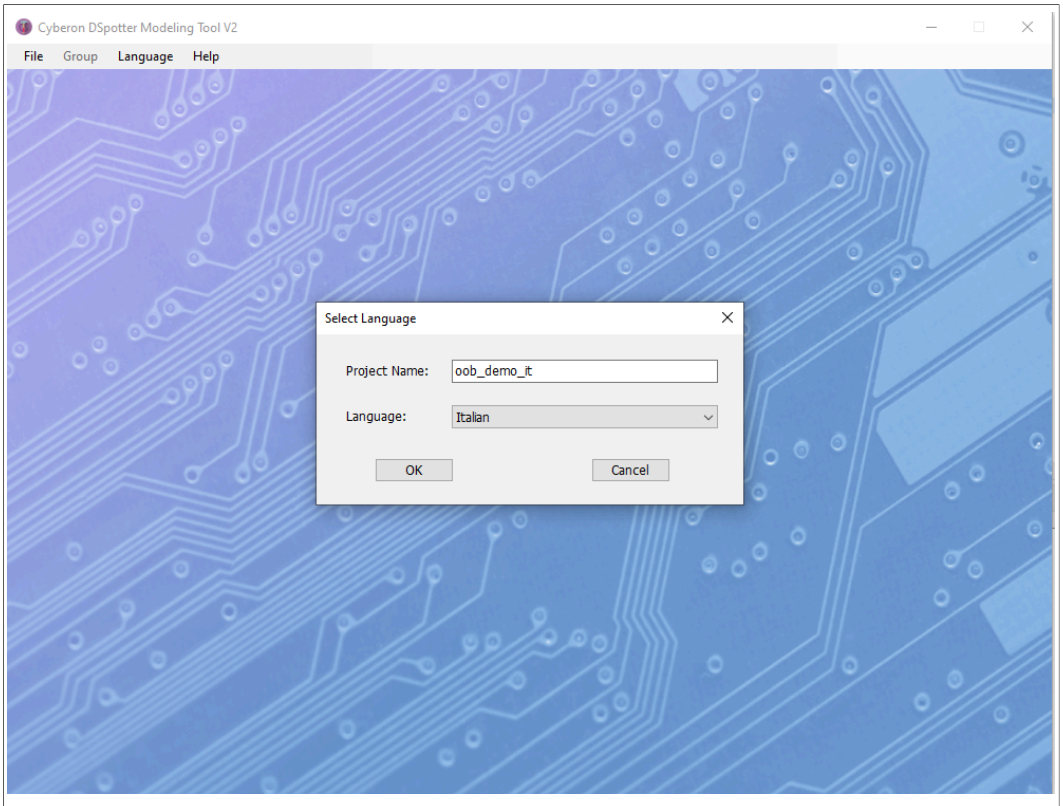


Figure 42. New dsmt project

3. Use the default settings. Click **OK**.

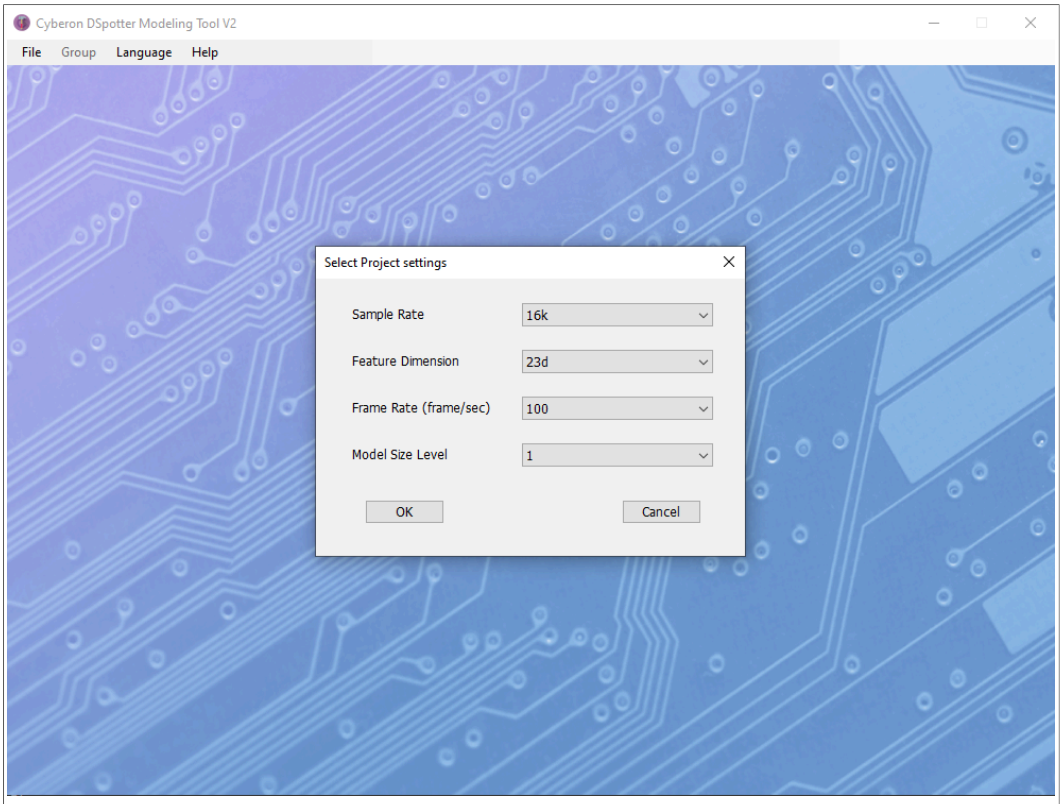


Figure 43. New dsmt project settings

4. When asked about the Folder where the project should be saved, go to the workspace location of the **cm7 Coffee Machine demo project -> local_voice folder.**

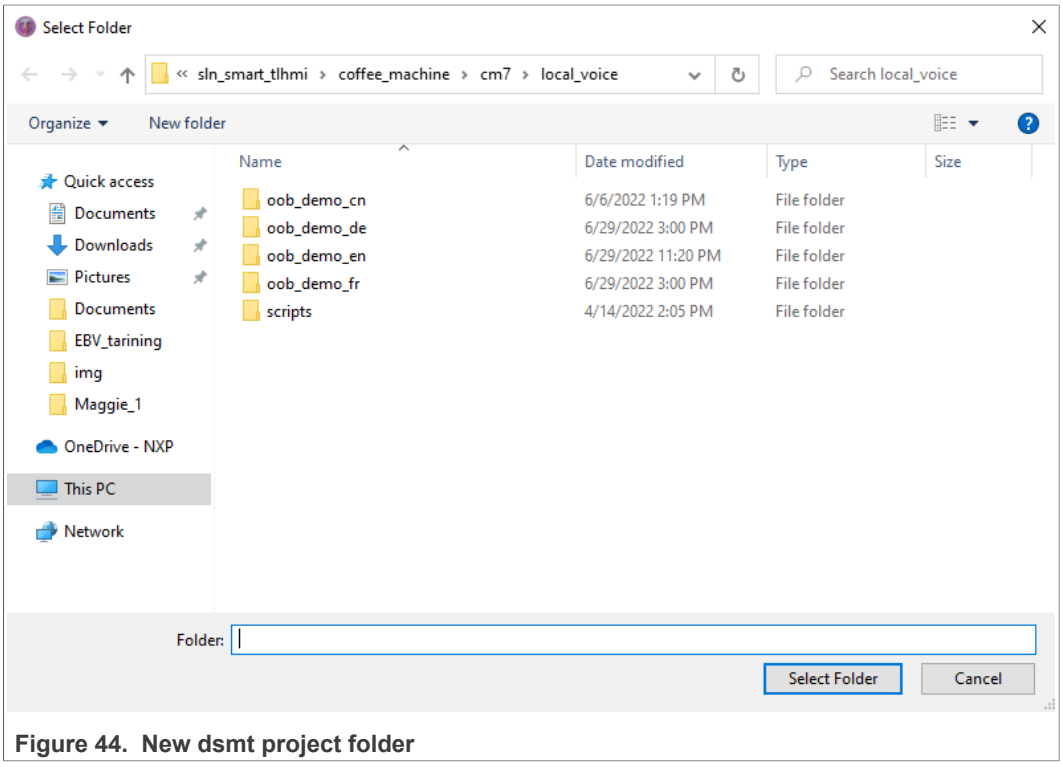


Figure 44. New dsmt project folder

5. Rename Group_1 to WW by selecting **Group -> Rename**.

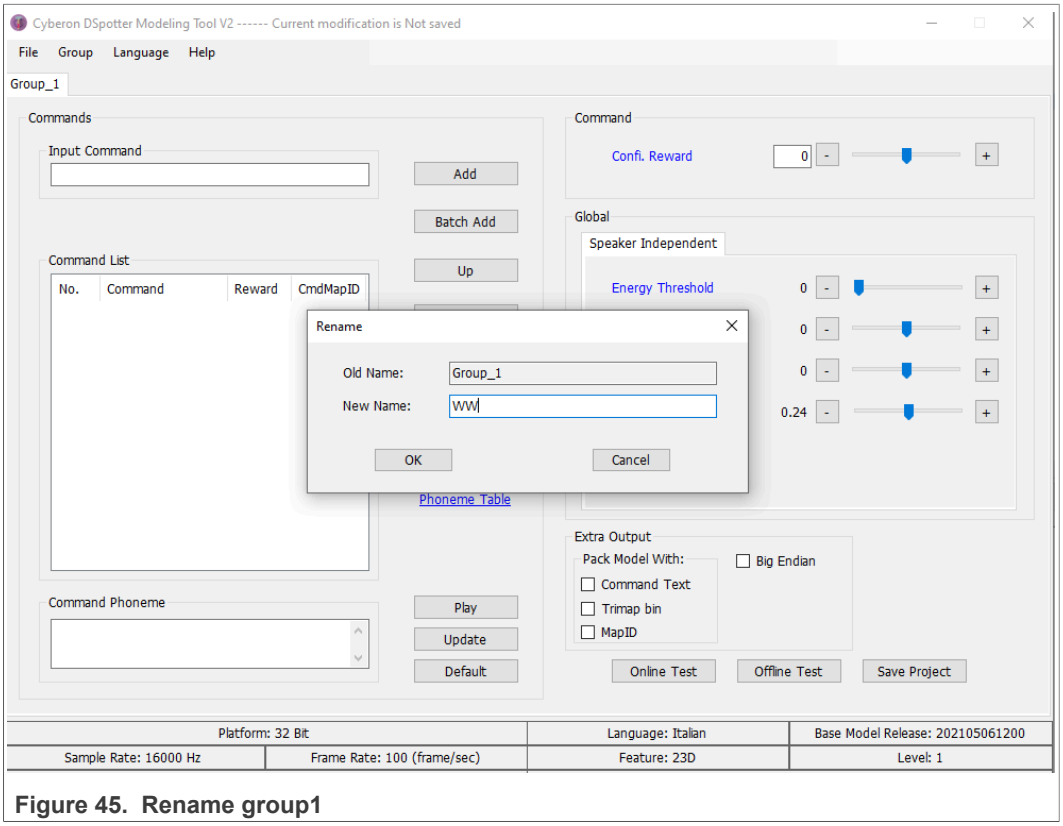


Figure 45. Rename group1

6. Add a simple wake word - let us use "Ciao NXP". By default CmdMapId has value -1. Change that to value 1 by double-clicking the wake word.

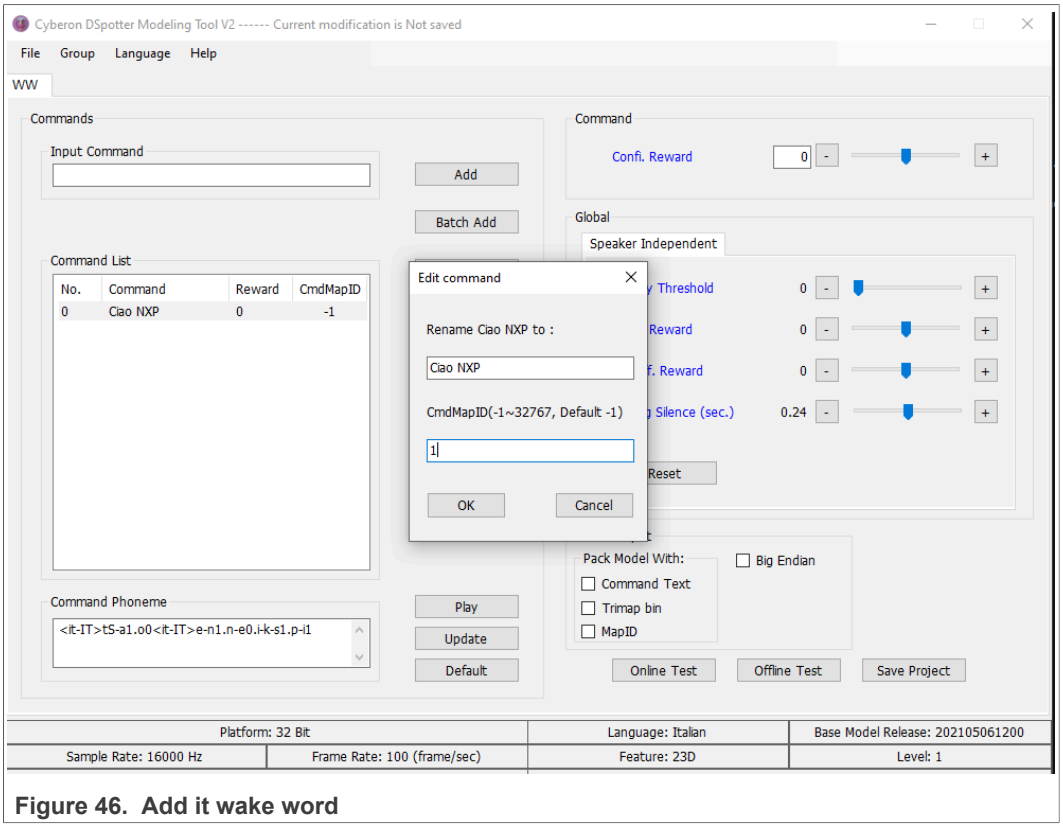


Figure 46. Add it wake word

7. Add a new group by selecting **Group -> Insert**. Change the group name to CMD_COFFEE_MACHINE.
   Add the commands below and change CmdMapId value to 2 for all of them.
   Inizia, Annulla, Confermare, Caffè espresso, Caffè americano, Cappuccino, Caffè Latte, Piccolo, Medio, Grande, Leggero, Mite, Forte, Annullare la registrazione.

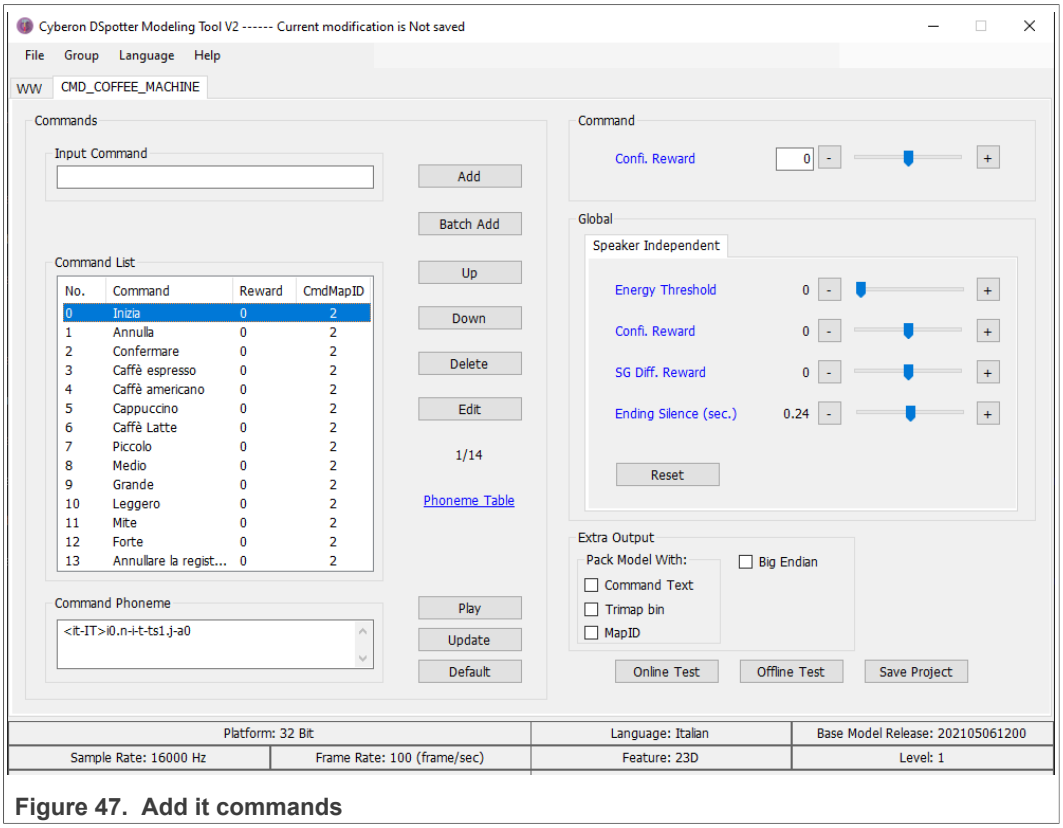


Figure 47. Add it commands

8. Very important: Check the MapID checkbox, otherwise the binary we must integrate into our project will not be generated.

Figure 48. Add it commands

9. Save the DSMT project (Ctrl + S or **File -> Save project**).
10. Now we modify the source code to use the newly generated Italian speech model. It is easier to replace one of the existing models, like French.
    - create `IndexToCommand_it.h`

```
#ifndef INDEXCOMMANDS_IT_H_
#define INDEXCOMMANDS_IT_H_

char *ww_it[] = {"Ciao NXP"};

char *cmd_coffee_machine_it[] = {"Inizia",         "Annulla",        "Confermare",     "CaffÃ¨ espresso",
                                 "CaffÃ¨ americano",     "Cappuccino",     "CaffÃ¨ Latte",
                                 "Piccolo",     "Medio",          "Grande",     "Leggero",          "Mite",
                                 "Forte",          "Annullare la registrazione"};

#endif /* INDEXCOMMANDS_IT_H_ */
```

Figure 49. Index to cmd it

Replace the following symbols in your workspace:

- `ASR_FRENCH` with `ASR_ITALIAN`

- `NUMBER_OF_COFFEE_MACHINE_CMDS_FR` with `NUMBER_OF_COFFEE_MACHINE_CMDS_IT` (must add that in `IndexCommands_dsmt.h`). `NUMBER_OF_COFFEE_MACHINE_CMDS_IT` should be 14.
- `action_coffee_machine_fr` with the equivalent `action_coffee_machine_it`
- `action_coffee_machine_fr` can be removed from `IndexCommands_dsmt.h`
- In `IndexCommands_dsmt.h` include `IndexToCommand_it.h` instead of `IndexToCommand_fr.h`
- `action_coffee_machine_it` must be defined, as shown below

```
92  unsigned int action_coffee_machine_it[] = {
93      kCoffeeMachineActionStart,      // "Inizia"
94      kCoffeeMachineActionCancel,     // "Annulla"
95      kCoffeeMachineActionConfirm,    // "Confermare"
96      kCoffeeMachineActionEspresso,   // "Caffè espresso"
97      kCoffeeMachineActionAmericano,  // "Caffè americano"
98      kCoffeeMachineActionCappuccino, // "Cappuccino"
99      kCoffeeMachineActionLatte,      // "Caffè Latte"
100     kCoffeeMachineActionSmall,      // "Piccolo"
101     kCoffeeMachineActionMedium,     // "Medio"
102     kCoffeeMachineActionLarge,      // "Grande"
103     kCoffeeMachineActionSoft,       // "Leggero"
104     kCoffeeMachineActionMild,       // "Mite"
105     kCoffeeMachineActionStrong,     // "Forte"
106     kCoffeeMachineActionDeregister, // "Annullare la registrazione"
107 };
108
```

Figure 50. Coffee machine it commands

- replace `oob_demo_fr_begin` with `oob_demo_it_begin` everywhere in the workspace
- use `oob_demo_it_pack_WithMapID.bin` in `local_voice_model.s`

Modified, not staged File: coffee_machine/cm7/local_voice/local_voice_model.s

```
@@ -11,11 +11,11 @@
 .align 4

 .global oob_demo_en_begin
 .global oob_demo_cn_begin
 .global oob_demo_de_begin
-.global oob_demo_fr_begin
+.global oob_demo_it_begin


 oob_demo_en_begin:
 .incbin "../local_voice/oob_demo_en/oob_demo_en_pack_WithMapID.bin"
 oob_demo_en_end:
@@ -26,9 +26,9 @@ oob_demo_cn_end:

 oob_demo_de_begin:
 .incbin "../local_voice/oob_demo_de/oob_demo_de_pack_WithMapID.bin"
 oob_demo_de_end:

-oob_demo_fr_begin:
-.incbin "../local_voice/oob_demo_fr/oob_demo_fr_pack_WithMapID.bin"
-oob_demo_fr_end:
+oob_demo_it_begin:
+.incbin "../local_voice/oob_demo_it/oob_demo_it_pack_WithMapID.bin"
+oob_demo_it_end:
```

Figure 51. Index to cmd it

11. Replace `s_memPoolWLangFr` with `s_memPoolWLangIt`.
12. Delete the cm7 debug folder and rebuild afterwards. Flash the project. You must now be able to interact with the dev kit through voice.

9.3.6 Cyberon tools

Check the video tutorials: Cyberon demos

10 VIT speech model instructions

10.1 Getting started with VIT

Smart HMI demos use DSMT as Audio Speech Recognition technology by default. To enable VIT ASR in Smart HMI SDK demos, do the following code modifications:

1. In `cm7 board_define.h` comment `ENABLE_DSMT_ASR` and uncomment `ENABLE_VIT_ASR` (path toward header: coffee_machine `../coffee_machine/cm7/board/board_define.h` and elevator `../elevator/cm7/board/board_define.h`
2. At the moment of this release, French is not supported on VIT. Hiding it from the available languages menu is done putting `FRENCH_LANG_SUPPORTED` define on 0 in this file from both coffee_machine `../coffee_machine/cm4/custom/custom.h` and elevator `../elevator/cm4/custom/custom.h`.
3. After modifying the files, build the lvgl library, then build the cm7 project and flash it

10.2 Barge-in support when VIT is enabled

At the moment of this release, VIT is not compatible with the AFE which is integrated into the Smart HMI SDK. As a consequence, barge-in is not available when VIT is enabled. It should change in the future, as compatibility between VIT and Voice Seeker is planned.

10.3 Obtaining a new VIT model

To obtain a new model, submit a request at this address: https://vit.nxp.com/#/

Note: *To do this, you need an nxp.com account.*

10.4 Integrating a new VIT model

Place the newly obtained model in the same folder as the currently existing models, as shown in Figure 52.

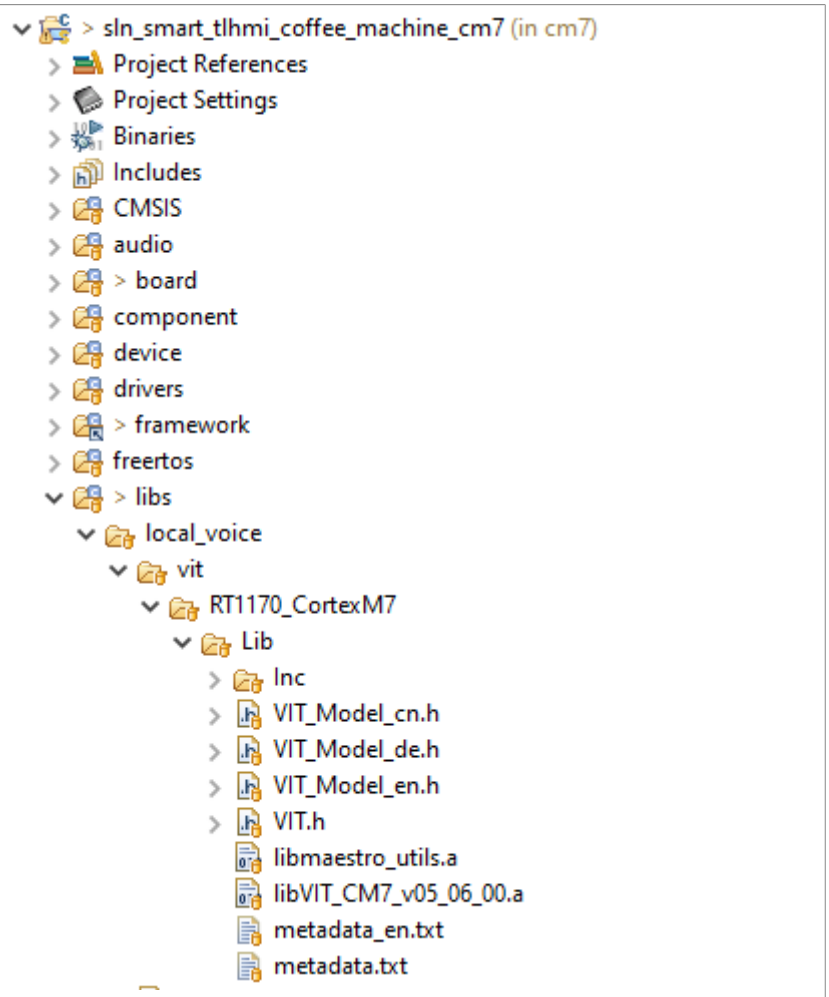


Figure 52. VIT models

Other files that must be updated for VIT integration are the ones highlighted below.

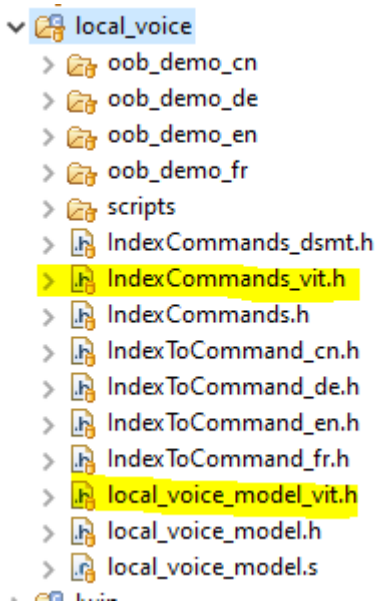


Figure 53. VIT files

10.5 Multilanguage support

VIT does not support listening for multiple wake words from different languages at the same time, as it is the case with DSMT. Hence, you will be able to say only one wake word at a time. To change to a different language, use the language menu from the display.

10.6 Additional info and resources

For documentation and other resources, see: VIT page

11 Revision history

Table 1. Revision history

Revision number	Date	Substantive changes
0	25 October 2022	Initial release

12 Legal information

12.1 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.

12.2 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 http://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 data sheet 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.

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.

12.3 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.

Contents

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

Date of release: 25 October 2022
Document identifier: MCU-SMHMI-SDUG