SLN-VIZN3D-IOTDG

SLN-VIZN3D-IOT SW Developer Guide

Rev. 0 — 19 August 2022 User guide

Document information

Information	Content
Keywords	SLN-VIZN3D-IOT
Abstract	The purpose of this guide is to help developers gain a better understanding of the software design and architecture of the Smart Lock application in order to more easily and efficiently implement applications using the SLN-VIZN3D-IOT.

1 Introduction

Welcome to the Developer Guide for the SLN-VIZN3D-IOT!

The purpose of this guide is to help developers gain a better understanding of the software design and architecture of the Smart Lock application in order to more easily and efficiently implement applications using the SLN-VIZN3D-IOT.

This guide covers topics including Bootloader, Framework + HAL Architecture Design, and the project-specific features of Smart Lock application which may be relevant to developing Machine Vision applications for the SLN-VIZN3D-IOT.

1.1 Smart Lock application layout

The Smart Lock application for the SLN-VIZN3D-IOT provides a fully integrated HW + SW solution, which allows for the rapid prototyping and development of Machine Vision-based applications. The Smart Lock application comes with full source code as well as hardware reference designs to help get developers up and running as quickly as possible.

The design of the Smart Lock app falls into two distinct layers: an underlying Framework + HAL layer, and a top-level Application layer.

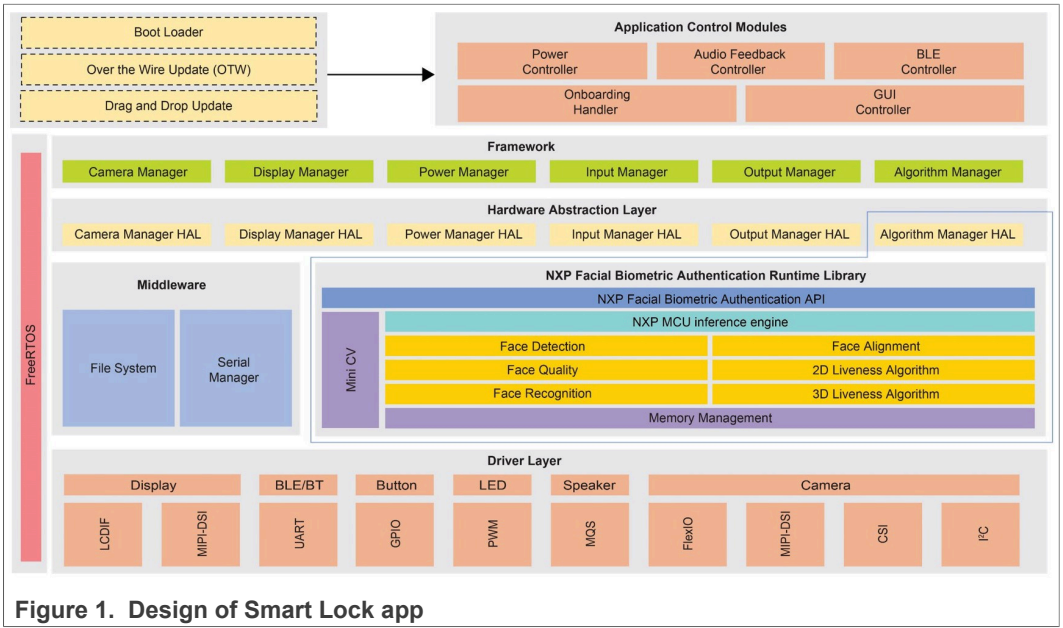

Figure 1. Design of Smart Lock app

The bottom Framework + HAL layer acts as a message routing system which allows the peripherals connected to the board to interact with one another.

The Framework was designed with code portability in mind, with the idea that low-level driver bindings would connect to higher-level, platform-agnostic **Hardware Abstraction Layer drivers** which do not depend on the underlying pin assignments, and so on. They are specific to the board. This design allows for the easy migration from one platform to another, helping alleviate platform lock-in and make code easier to read, write, modify, and maintain.

The top Application layer contains all application-specific code including the various sounds, icons, UI elements, and so on. In addition, the Application layer registers all the

devices relevant to the application, as well as their Event Handlers which react to events triggered by other devices.

Separating the **Application** and **Framework + HAL** layers from each other encourages code reuse between different projects because the underlying Framework code can be reused in almost its entirety, while primarily only the **Application** layer code need modifications.

Note: *Be sure to check out the Getting Started Guide for an overview of the out-of-box features available in the SLN-VIZN3D-IOT Smart Lock application.*

2 Setup and installation

This section focuses on the setup up and installation of the tools necessary to begin developing applications using the framework architecture of NXP.

Note: *This guide focuses on the use of MCUXpresso IDE for development.*

2.1 MCUXpresso IDE

The 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. The 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. The 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 about the MCUXpresso IDE, see the NXP website.

2.2 Install the toolchain

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

Figure 2. Install the toolchain

Once the download has completed, simply follow the instructions in the installer to get started.

2.3 Install the SDK

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

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

Figure 3. INstall the SDK

To build the SDK for your preferred setup, perform the following steps:

1. Select your preferred OS (Windows, Mac, Linux).
2. Configure the SDK to be compatible your preferred toolchain (MCUXpresso IDE, ARMGCC).
3. Use the **Select All** button to ensure that all necessary SDK components are included in the SDK package.
4. Click the **Download SDK** button.

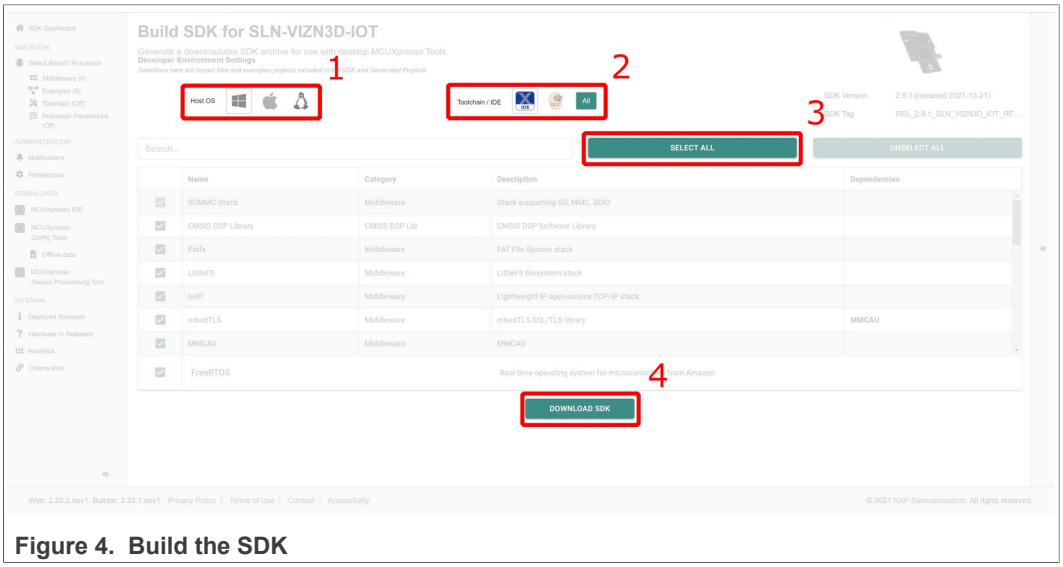

Figure 4. Build the SDK

Once the SDK has been downloaded, it can be installed into MCUXpresso IDE by dragging and dropping the zip file into the **Installed SDKs** window in MCUXpresso IDE.

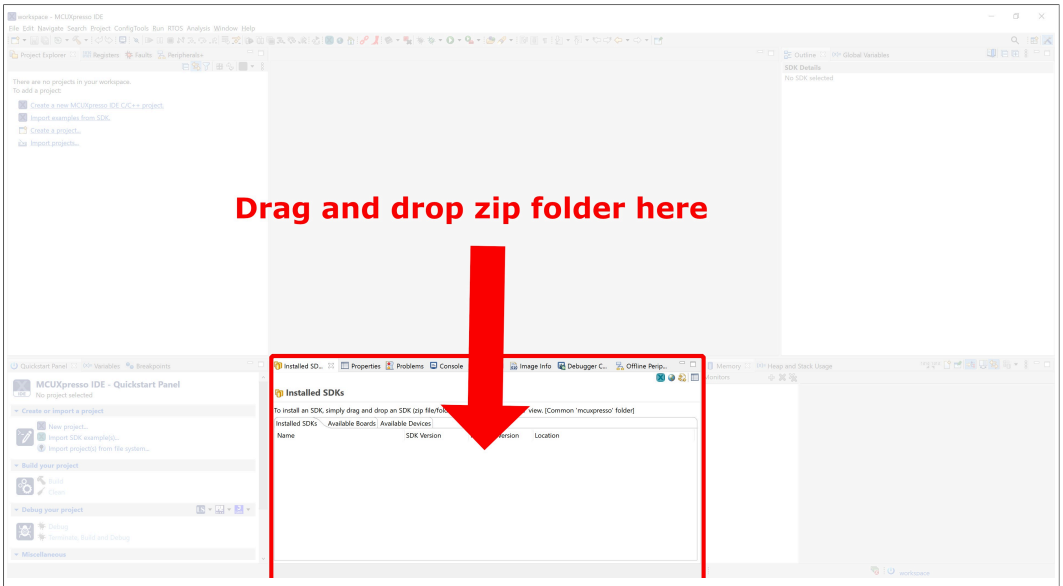

Figure 5. SDK installed

2.4 Import example projects

Note: *You **MUST** have a compatible MCUXpresso SDK package for the SLN-VIZN3D-IOT installed to build example projects you import regardless of how they are imported.*

MCUXpresso IDE allows you to import example projects from a variety of sources including:

- A compatible MCUXpresso SDK package
- A zipped MCUXpresso project folder
- An unzipped MCUXpresso project folder

This section discusses both methods officially supported for importing the Smart Lock application into MCUXpresso.

2.4.1 Import from GitHub (Option 1)

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

The latest software updates for the SLN-VIZN3D-IOT Smart Lock application can be downloaded from the official GitHub repository. Here, you will find the most up-to-date version of the code which contains the newest features available for the Smart Lock project.

To import the SLN-VIZN3D-IOT Smart Lock application into MCUXpresso IDE using GitHub, perform the following steps:

1. Clone the **vizn3d_smart_lock** repository.

```
git clone https://github.com/NXP/vizn3d_smartlock_oobe.git
```

   Note: *Cloning directly to your MCUXpresso workspace location is recommended, but not required.*

2. In MCUxpresso IDE, navigate to the **QuickStart Panel** and click **Import Project(s) from File System. . .**.

3. Select **Browse. . .** next to the **Root Directory** box.

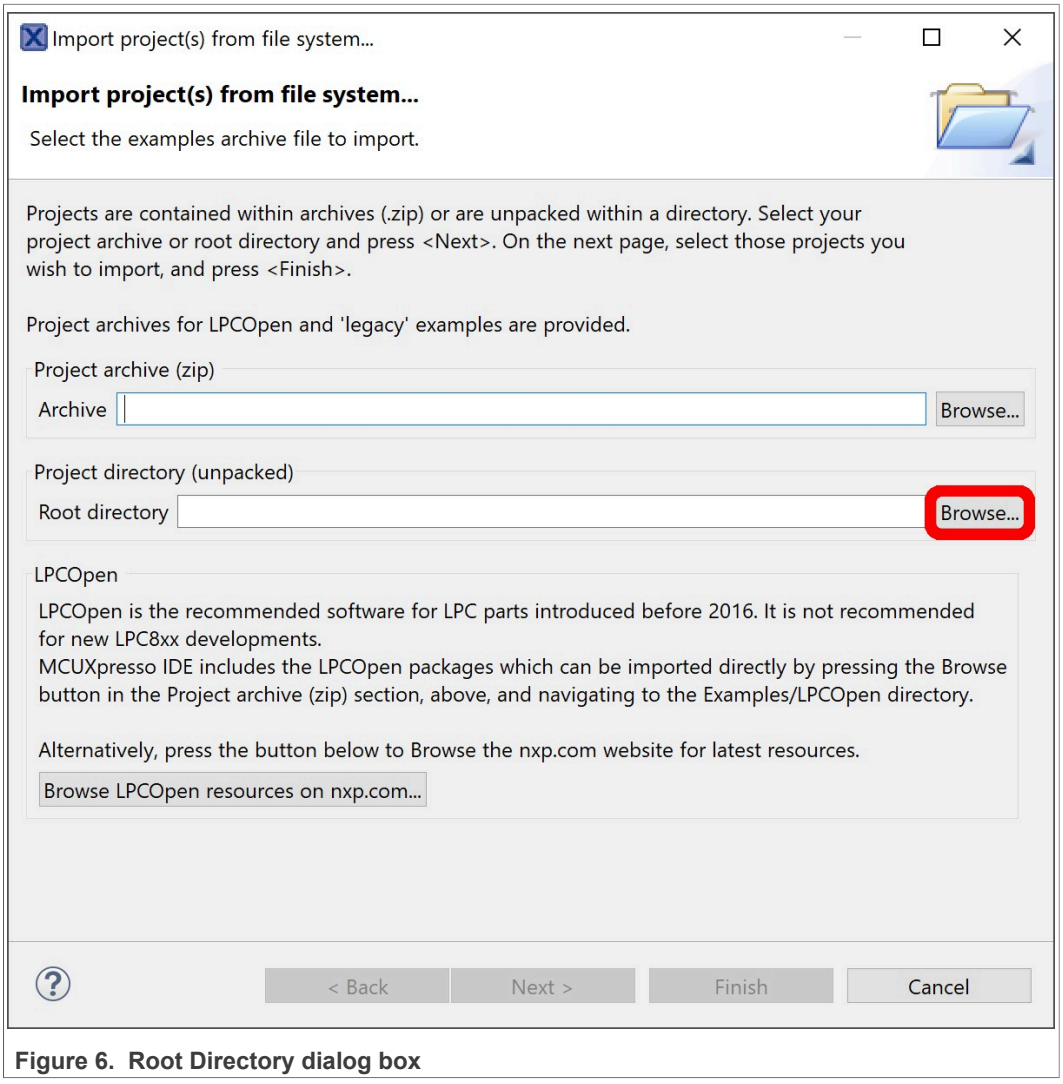

Figure 6. Root Directory dialog box

4. Navigate to the file path of the project cloned in the first step and click **Select Folder**.

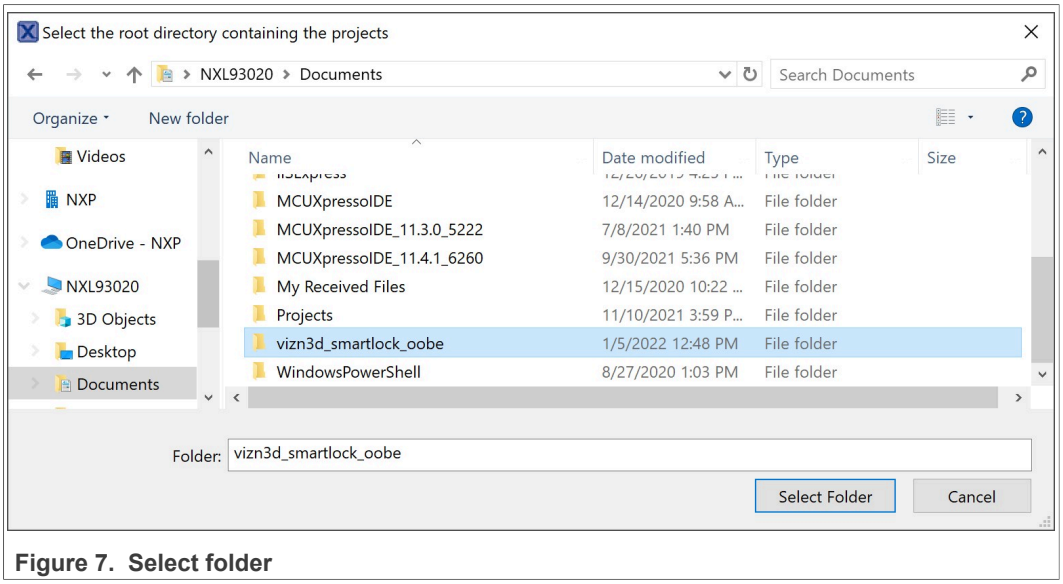

Figure 7. Select folder

5. Click **Next** and check the box next to each project (`sln_vizn3d_iot_smart_lock`, and `sln_vizn3d_iot_bootloader`) you wish to import. Click **Finish** after selecting the projects you want to import.

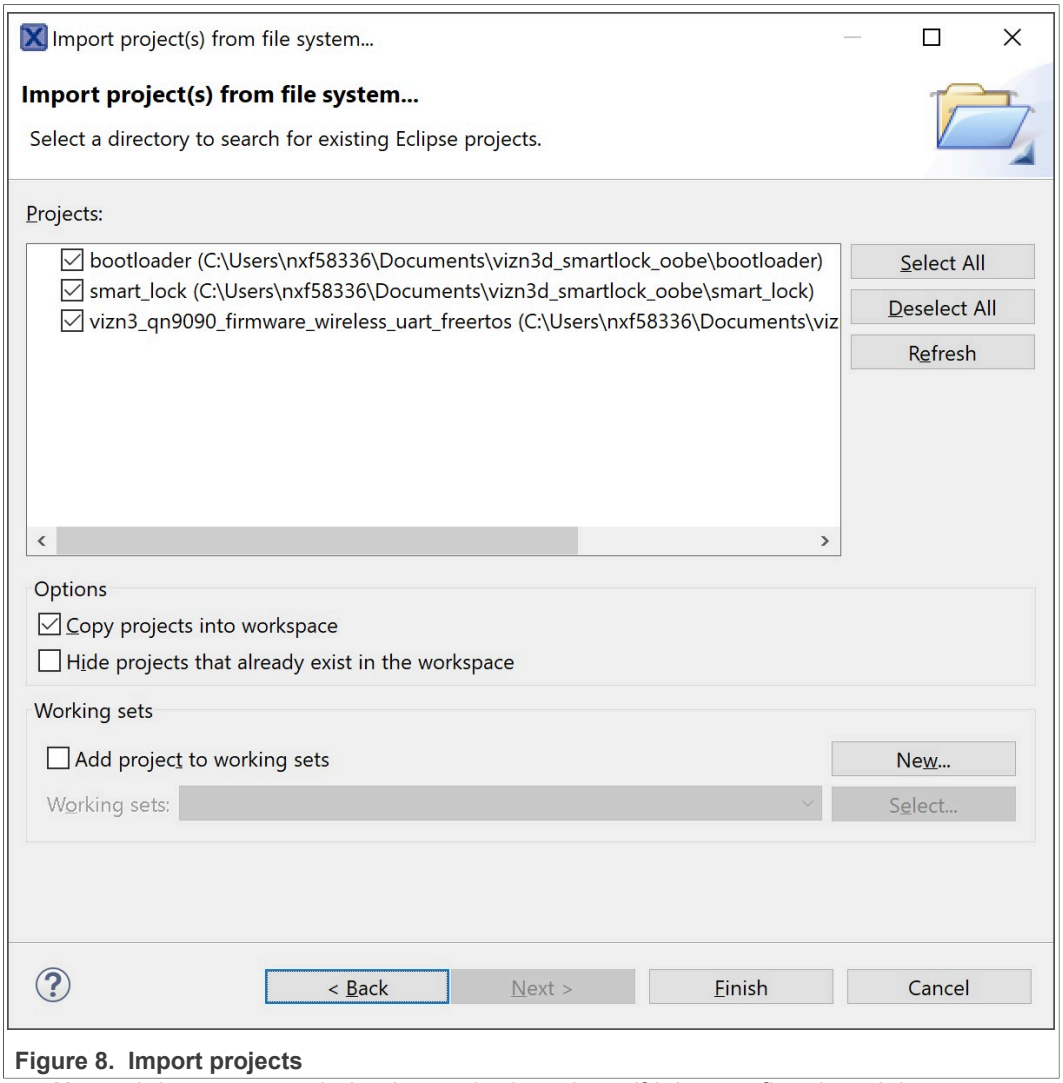

Figure 8. Import projects

Note: *It is recommended to import both projects if it is your first time doing so.*
Note: *Deselect* ***Copy projects into workspace*** *if you want any changes you make in the original repo (git pull, etc.) to be reflected in MCUXpresso IDE.*

6. After following the above steps, confirm that the projects can be found in the **Project Explorer**.

Figure 9. Project completed

2.4.2 Import from MCUXpresso SDK (Option 2)

Note: *For the latest version of the Smart Lock application, clone the example project directly from the GitHub repository.*

For instructions to import SDK examples using the MCUXpresso SDK, see the Getting Started Guide for the SLN-VIZN3D-IOT.

3 BOOTLOADER

3.1 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:

- Launch the main application and, if necessary, initialize 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[1]

3.1.1 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 that precautions be taken to ensure the integrity and stability of the firmware before, during, and after an update. The bootloader application is simply one measure to help provide this assurance.

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

3.1.2 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 should fail midway through Bank B will still contain a fully operational backup.

In the SLN-VIZN3D-IOT, Bank A is located at `0x30100000` while Bank B is located at `0x30700000`. Specify the flash address of an application. Then, to compile the application, select **Properties** -> **MCU Settings** menu, as shown in Figure 10.

1 The SLN-VIZN3D-IOT does not currently support any bootloader security features.

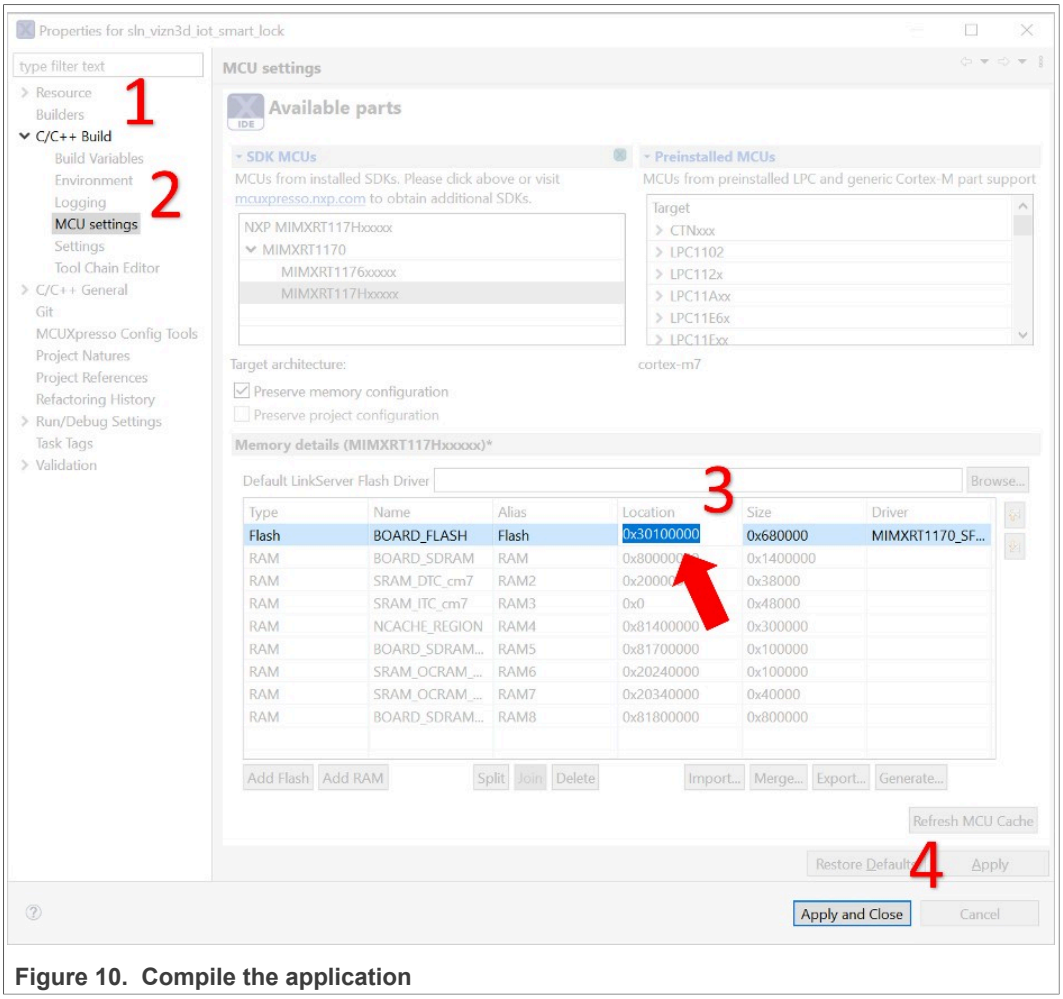

Figure 10. Compile the application

During the MSD, OTA, and OTW updates, it is crucial to provide an application binary built for the proper application bank address. The failure to do so results 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.*

3.1.3 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. It 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 in the bootloader, simply build and run the bootloader in the Release build mode configuration. To do so, right-click the bootloader project in the Project Explorer view and navigate to **Build Configurations -> Set Active -> Release**, as shown in Figure 11.

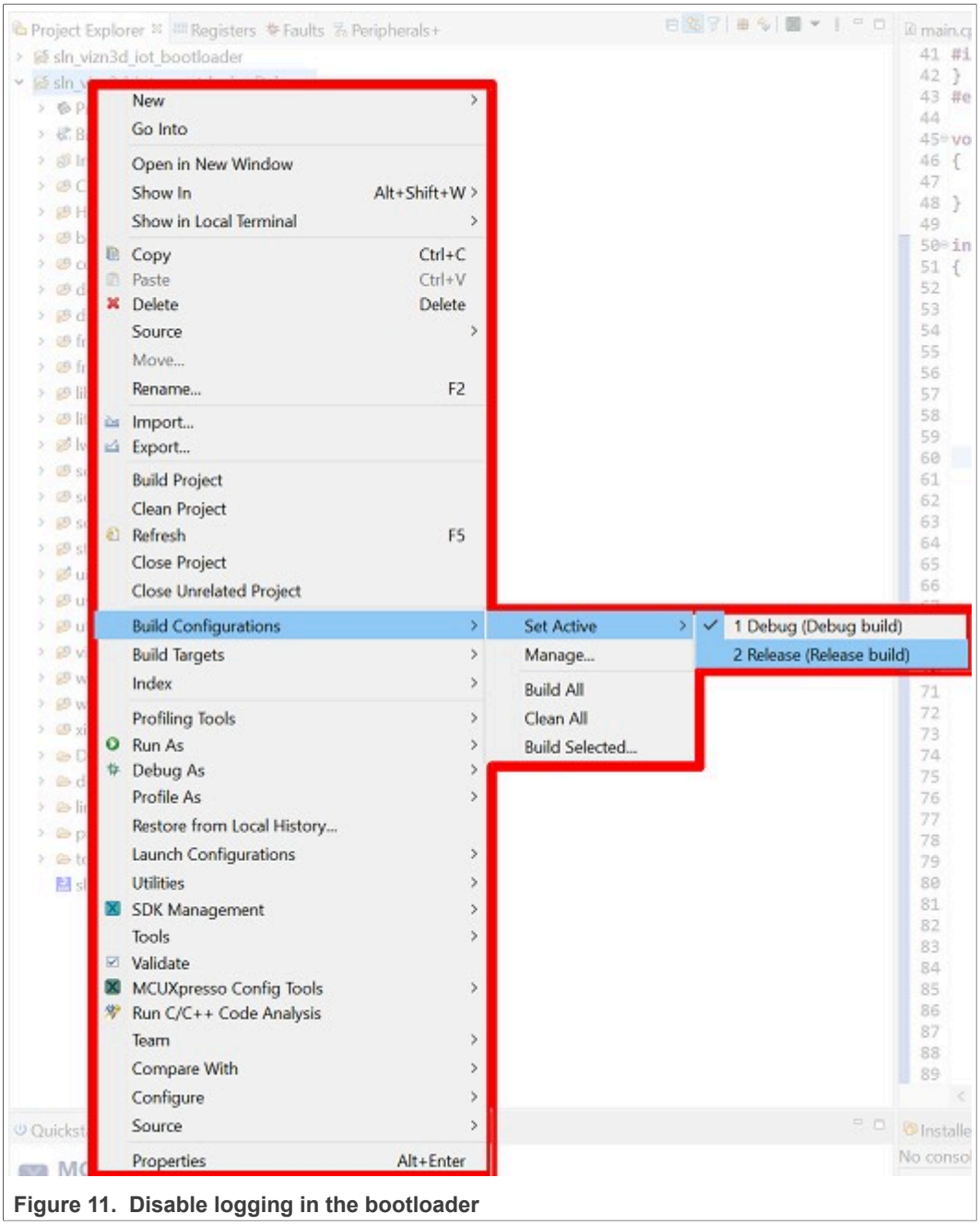

Figure 11. Disable logging in the bootloader

To use 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

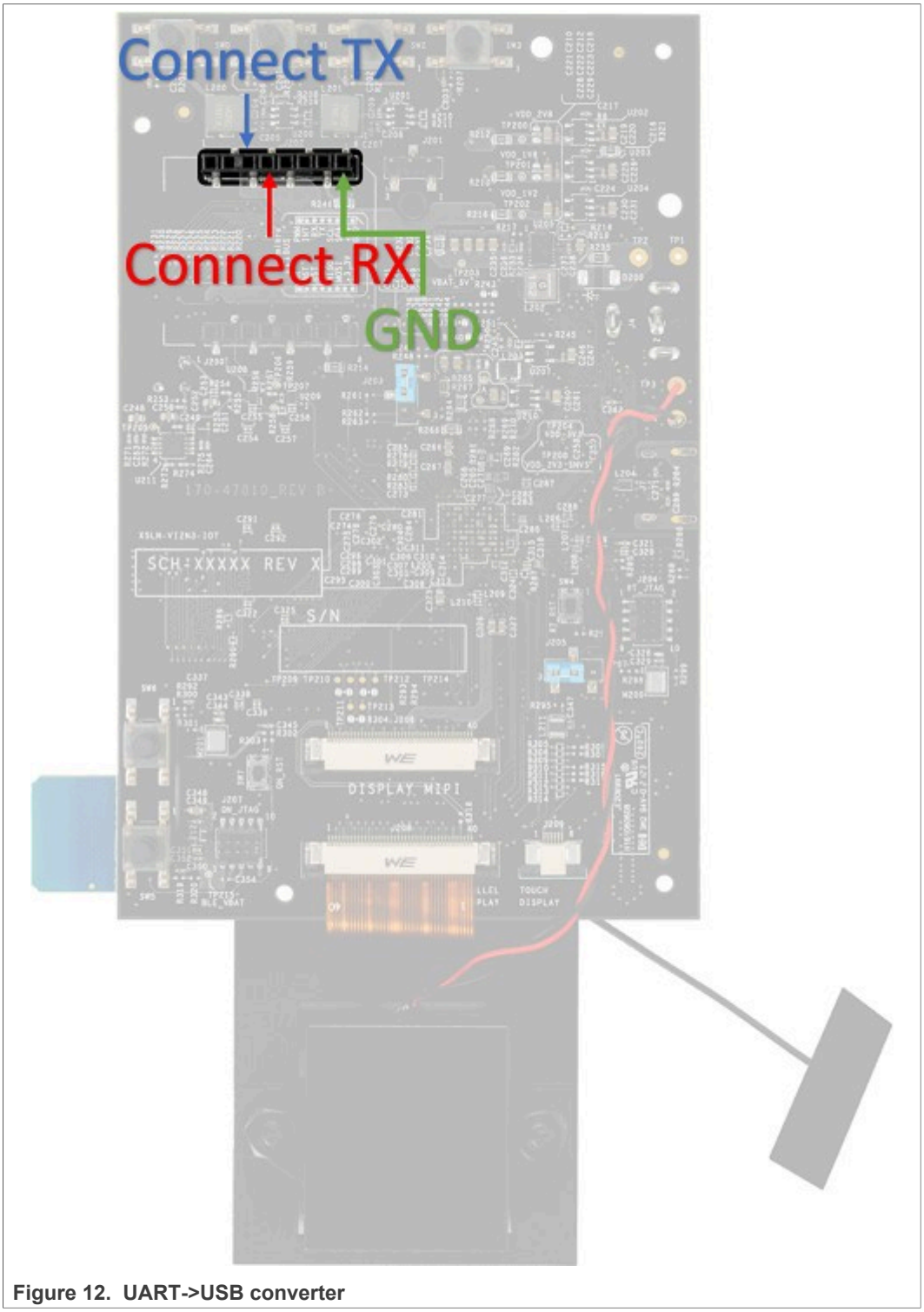

Figure 12. UART->USB converter

Once the converter has been properly attached, connect to the board using a serial terminal emulator like **PuTTY** or **Tera Term**. The serial settings are as below:

- Speed: 115200
- Data: 8 Bit
- Parity: None

- Stop Bits: 1 bit
- Flow Control: None

3.2 Overview

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

- Normal boot mode: It is the default boot mode in which the bootloader simply loads the main application.
- Mass Storage Device (MSD) boot mode: It 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 can receive and flash a new binary by copying that binary to the board as one would for a regular USB storage device.

For more information on these modes, see the subsequent sections in this document.

3.2.1 How is boot mode determined?

To determine the boot mode to enter, the bootloader checks several boot flags which get set based on various conditions being met.

For each boot mode (excluding Normal boot mode which is taken by default), there is a different corresponding boot flag. The means by which a boot flag gets set depends on the boot mode in question and the platform being used. On the SLN-VIZN3D-IOT, for example, the MSD boot flag is set when the SW1 button is held during bootup.

3.3 Normal boot mode

By default, if no other boot flags are set during the boot phase, the Normal boot mode is used. During the Normal boot, the bootloader simply 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.

3.4 Mass Storage Device (MSD) boot mode

The Mass Storage Device (MSD) boot mode is a means by which application binaries can be flashed to the board via a drag-and-drop interface like one would use with a USB flash drive or similar device. MSD mode is useful for deploying quick updates to marketers and engineers in the field without access to debugging tools like a Segger J-Link.

3.4.1 Enabling MSD mode

To enable the MSD mode on the SLN-VIZN3D-IOT, press and hold the SW1 button while powering on the board. If the operation is correct, the onboard LED of the board changes to purple and begins blinking at an interval of roughly 1 second.

Figure 13. Enabling MSD mode

Additionally, if connected to a Windows PC, your computer makes a sound indicating that a new USB device has been connected and a new USB Storage Device is shown in the file explorer.

3.4.2 Flashing a new binary

To flash a new binary while Mass Storage device mode is enabled, you must first verify the application bank which is currently in use. For the information, use the version shell command while the main app is running.

```
version
App running in Bank A
Version 1.0.4
Shell>>
```

Once the current application bank in use has been identified, you must compile a binary for the alternate flash bank. For example, if Bank A is currently in use, you must compile a Bank B binary and vice versa. For Instructions on how to compile a specific flash bank, see Application banks.

After compiling a binary for the proper flash bank, activate MSD mode by following Enabling MSD mode.

To begin flashing the binary, simply drag and drop the binary onto the device listing for the USB Storage device associated with your board. While flashing is in progress, a pop-up window indicates the current progress of the firmware download, as shown in Figure 14.

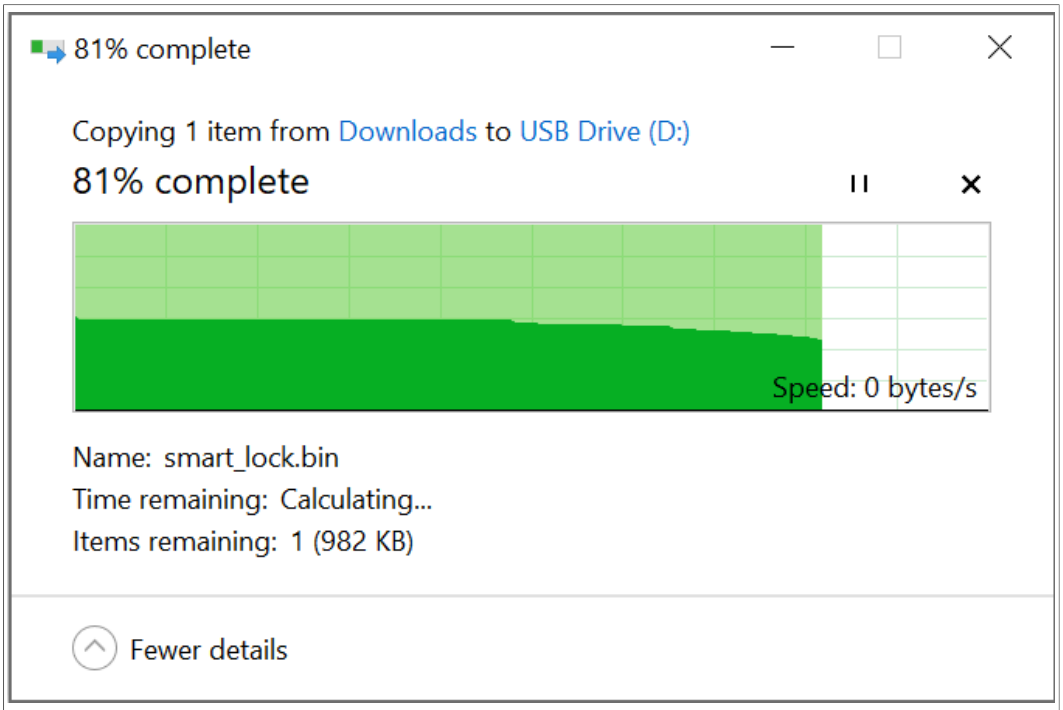

Figure 14. Flash the binary

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

3.5 Application banks

- Dual application flash banks, **Bank A** and **Bank B**.
- Provides a redundancy mechanism used by the update mechanisms of the bootloader.

The SLN-VIZN3D-IOT utilizes a series of dual **application flash banks** used as redundancy mechanism when updating the firmware via the update mechanisms of one of the bootloader.

3.5.1 Addresses

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

- Bank A - 0x30100000
- Bank B - 0x30780000

3.5.2 Configuring Flash bank in MCUXpresso IDE

Before compiling a project, configure the flash bank in MCUXpresso IDE.

1. Right-click the **sln_vizn3d_iot_smart_lock** project in the **Project Explorer** window.
2. Go to **Properties**.

3. Click on **MCU Settings**.
4. Change **FLASH_BANK** from `0x30100000` to `0x30780000` or vice versa.
5. Build the project.

3.5.2.1 Converting .axf to .bin

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

Fortunately, a **.axf** file can be converted to **.bin** in MCUXpresso without any additional setup.

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

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

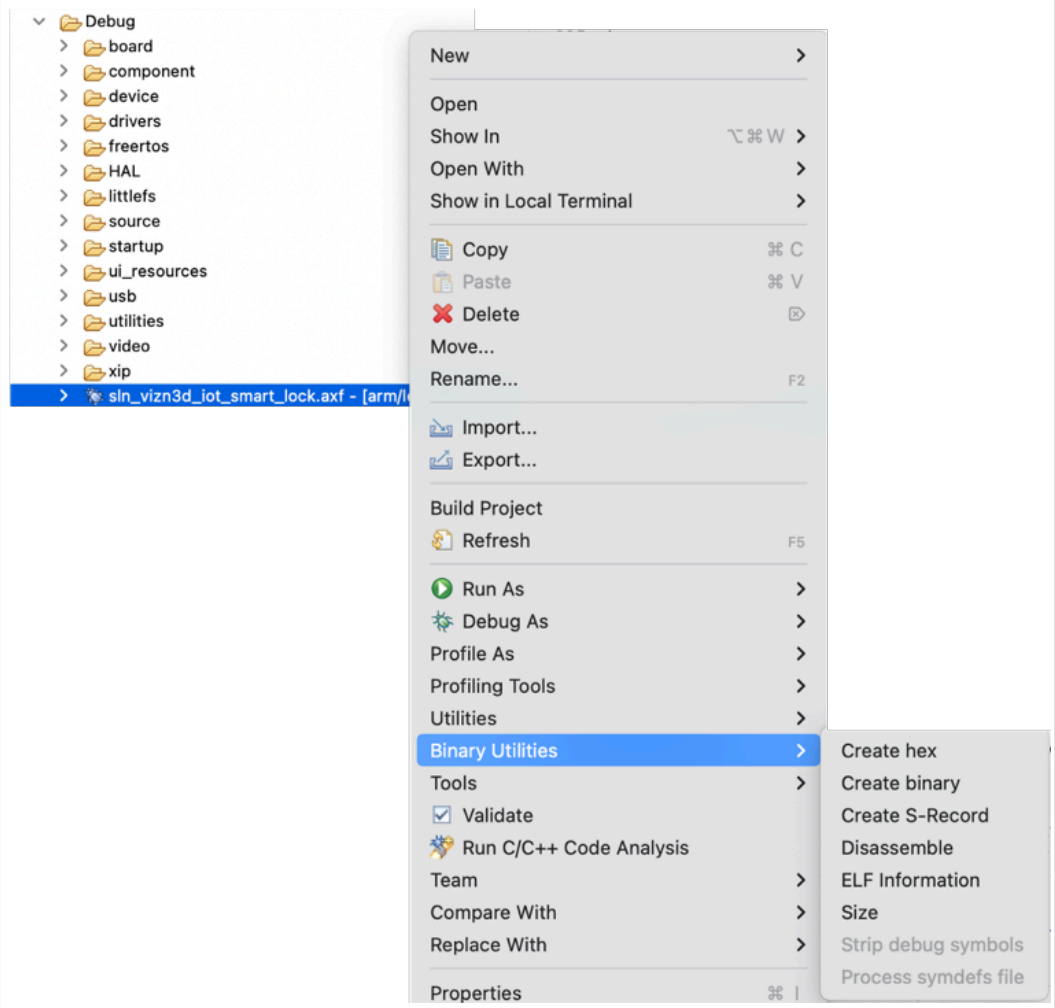

Figure 15. Context menu

Verify that the binary has successfully been created.

4 FRAMEWORK

4.1 Framework introduction

This section discusses the architecture design of the Smart Lock application. The Smart Lock application is primarily designed around the use of a **framework** architecture which is composed of several different parts.

These constituent parts include:

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

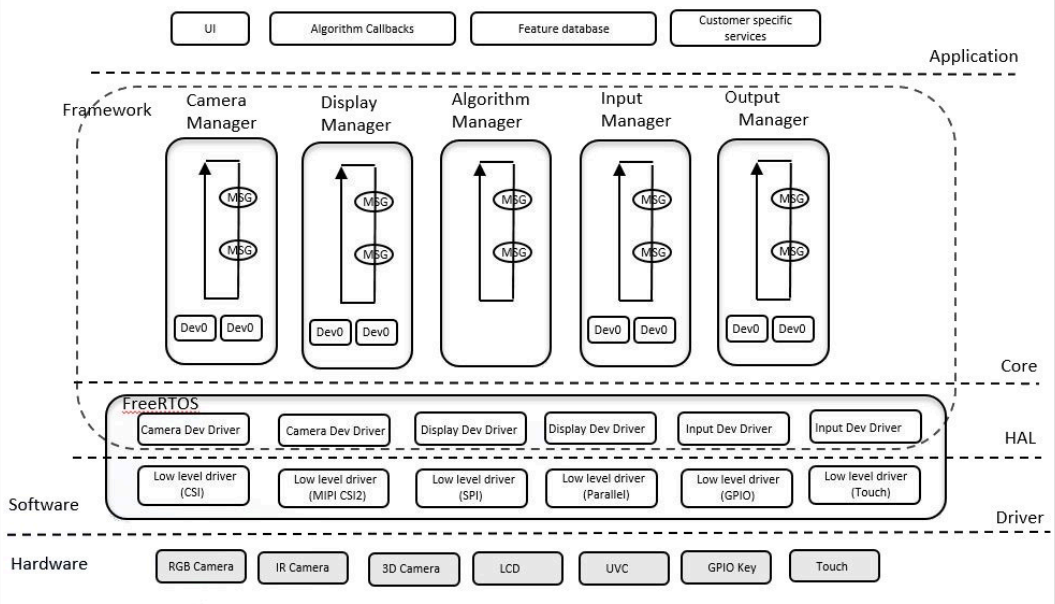

Figure 16. Smart Lock application

The following sections discuss each component.

4.1.1 Design goals

The architectural design of the Smart Lock application software was centered around three primary goals:

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

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

The SLN-VIZN3D-IOT platform 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 easy to modify. Keeping this goal in mind, the architecture of the Smart Lock software was

designed to be easy to modify without being restrictive, and without coming at the cost of performance.

4.1.2 Relevant files

The files which pertain to the framework architecture can primarily be found in the **source**/**framework**/**HAL** folders of the **sln_vizn3d_iot_smart_lock** application. Because the Smart Lock 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.

4.2 Device manager overview

Device managers are responsible for **managing** devices used by the system. Each device type (input, output, etc.) 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 will avoid 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.

4.2.1 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 the start routine of a manager, the manager will call the **init** and start functions of each of its registered devices.

Note: *In general, developers should only be concerned with adding/removing devices from the **APP_RegisterHalDevices()** function as the **Init** and **Start** functions for each manager is already called by default inside the **APP_InitFramework()** and **APP_StartFramework()** functions in **main()**.*

4.3 Vision input manager

4.3.1 APIs

- FWK_InputManager_Init

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

- 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);
```

- 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();
```

- 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 should be used with caution.*

4.4 Output manager

4.4.1 APIs

- FWK_OutputManager_Init

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

- 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);
```

- 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();
```

- FWK_OutputManager_Deinit

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

Note: *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);
```

- 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);
```

4.5 Camera manager

4.5.1 APIs

- FWK_CameraManager_Init

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

- 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);
```

- 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();
```

- 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 should be used with caution.*

4.6 Display manager

4.6.1 APIs

- FWK_DisplayManager_Init

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

```
 */
int FWK_DisplayManager_Init();
```

- 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);
```

- 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.
 *n@return int Return 0 if starting was successful
 */
int FWK_DisplayManager_Start();
```

- 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 should be used with caution.*

4.7 Vision algorithm manager

4.7.1 APIs

- FWK_VisionAlgoManager_Init

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

- 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);
```

- 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();
```

- 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 should be used with caution.*

4.8 Voice algorithm manager

4.8.1 APIs

- FWK_VoiceAlgoManager_Init

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

- 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);
```

- 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();
```

- 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 should be used with caution.*

4.9 Low power manager

The Low Power Device Manager is unique amongst 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 deep a sleep the board should enter, enable sleep mode, and more.

Note: *Due to the unique nature of the low power devices being an abstract **virtual** device, only one LPM device can be registered to the LPM manager at a time. However, there should 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.*

4.9.1 APIs

- 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);
```

- FWK_LpmManager_RegisterRequestHandler

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

- FWK_LpmManager_UnregisterRequestHandler

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

- FWK_LpmManager_RuntimeGet

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

- FWK_LpmManager_RuntimePut

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

- FWK_LpmManager_RuntimeSet

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

- FWK_LpmManager_RequestStatus

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

- 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);
```

- 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);
```

4.10 Audio processing manager

4.10.1 APIs

- FWK_AudioProcessing_Init

```
/**
 * @brief Init Audio Processing manager
 *
 * @return int Return 0 if the init process was successful
 */
int FWK_AudioProcessing_Init(void);
```

- 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);
```

- FWK_AudioProcessing_Start

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

- 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 should be used with caution.*

4.11 Flash manager

The Flash Manager provides 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. This means that API functions of the Flash Manager essentially act as wrappers which calls the Operators of the underlying flash HAL device.

Note: *When working with the Flash Manager, unlike most other managers,* ***FWK_Flash_DeviceRegister*** *should be called before* ***FWK_Flash_Init****.*

4.11.1 Device APIs

- ```
  /**
   * @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);
  ```

  Unlike the flow for most other managers, this function should be called *before* FWK_Flash_Init.
- 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();
  ```
- 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();
  ```

4.11.2 Operations APIs

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

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

While this 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.*

- FWK_Flash_Format

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

- 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);
```

- 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);
```

- FWK_Flash_Read

  Note: *For now, reading with an offset has not been implemented. This is currently planned for a future release.*

```
/**
 * @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 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 size);
```

  Note: *For now read with an offset has not been implemented. This is currently planned for a future release.*

- 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);
```

- 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);
```

- 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);
```

- 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);
```

4.12 HAL overview

One of the most important steps in the creation of any embedded software project is peripheral integration. Unfortunately, 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 which those drivers were originally written for, which 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 are also designed to be reused across different projects and even different NXP platforms, increasing code reuse which can help cut down on development time.

4.12.1 Device registration

To communicate with a HAL device, the manager must first be registered to 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 which 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;
}
```

Because 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. Each HAL device to be registered on startup 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.

4.12.2 Device types

There are several device types to encapsulate the various peripherals which 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 regarded as similar to interfaces in C++ and other object-oriented languages.

4.12.3 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 three components:

- ID
  The **ID** field is a unique device identifier which is assigned by the manager of the device when the device is first registered.
- Name
  The **Name** field is used to identify the device during various function calls and when debugging.
- Operators
  The **Operators** field is a structure which contains function pointers to each of the functions that the HAL device is required to implement. The operators which a device is required to implement varies with the device type.

The definition of an HAL device is stored in a structure which gets passed to the respective manager of that device when the device is registered. This gives the manager information about the device and allows the manager to call the operators of the device when necessary.

4.12.3.1 Operators

Operators are functions that **operate** on the device itself. Operators are used by the manager of the device 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 an 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 will have at least a **start**, **stop**, **init**, and **deinit** function. Additionally, most devices will also implement an **inputNotify** function which is used for Event handlers.

Note: *Failing to implement a function will 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 the **start** function of an HAL device will prevent its respective manager from starting that device.*

4.12.4 Configs

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

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

- expectedValue
  A string which provides a description of the valid values associated with the config. The length of the string should be less than DEVICE_CONFIG_EXPECTED_VAL_MAX_LENGTH.

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

- description

A string which provides a description of the config. The length of the string should be less than DE- VICE_CONFIG_DESCRIPTION_MAX_LENGTH.

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

- value
  An int which stores the internal value of the config. **value** should be set using the set function and retrieved using the get function.

```
uint32_t value;
```

- get
  A function which returns the value of the config.

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

- set
  A function which sets the value of the config.

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

4.13 Input devices

The Input HAL device provides an abstraction to implement a variety of devices which may capture data in many different ways, and whose 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 Event triggers.

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.

4.13.1 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;
```

4.13.2 Operators

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

For more information about operators, see Operators.

- 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, etc.), turn on the hardware, and perform any other setup the device requires.
The callback to the manager of the device 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.

- 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, etc.), 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.[2].

- 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.

- 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 will generally not be called under normal operation.

- 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 message queue of the Input Manager.
For more information regarding events and event handling, see Event triggers.

4.13.3 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 the **init** operator of a device.

- 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
```

2 The DeInit function generally will not be called under normal operation.

```
  * 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 will provide 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.

- 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.

- 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.

- 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;
```

4.13.4 Example

The SLN-VIZN3D-IOT Smart Lock 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;
}
```

4.14 Output devices

The Output HAL devices is used to represent any device which produces output (excluding specific devices which have their own specific device type like cameras and displays).

Output devices will respond to events passed by other HAL devices and produce corresponding output. This 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.

4.14.1 Subtypes

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

- General devices
- UI devices
- Audio devices

4.14.1.1 General devices

A **general**/generic output devices describes the majority of output devices, and includes devices like LEDs.

4.14.1.2 UI devices

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

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

4.14.1.3 Audio devices

Audio output HAL devices represent devices which act as a 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.

4.14.2 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;
```

4.14.3 Operators

Operators are functions which **operate** on an HAL device itself. Operators are akin to **public methods** in object oriented-languages, and are used by the Output Manager to setup, start, etc., each of its registered output devices.

For more information about operators, see Operators.

- 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, etc.), 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.

- 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, etc.), 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[3].
The DeInit function generally will not be called under normal operation.

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

- 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. This prevents the event handlers of the device from executing when an event is triggered.

3 The DeInit function generally will not be called under normal operation.

4.14.4 Attributes

- type
  The type of output device. If the type is **kOutputDevType_UI**, the `pSurface` parameter will need to 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;
```

- pSurface
  The pSurface variable is used by UI 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;
```

4.14.5 Example

The SLN-VIZN3D-IOT Smart Lock project has several output devices implemented for use as-is or for use as 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;
}
```

4.15 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.

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, etc.) as necessary. Additionally, the Camera Manager allows for multiple camera devices to be registered and operate at once.

4.15.1 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;
```

4.15.2 Operators

Operators are functions which **operate** on an HAL device itself. Operators are akin to **public methods** in object oriented-languages, and are used by the Camera Manager to setup, start, etc. each of its registered camera devices.

For more information about operators, see Operators.

- 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, etc.), turn on the hardware, and perform any other setup the device requires. This operator will be called by the Camera Manager when the Camera Manager task first starts.

- DeInit

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

Deinitialize the camera device.
DeInit should release any hardware resources the camera device uses (I/O ports, IRQs, etc.), 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[4].

- 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 task of Camera Manager after the call to the Init operator. The startup of the camera sensor and

4 The DeInit function generally will not be called under normal operation.

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.

- 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 buffer queue of camera device. 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.

- 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.

- 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 needs to be converted from one format to another in some way before it is useable by the display and/or a vision algo device, this would take place in the PostProcess operator.

- 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 message queue of Camera Manager.
For more information regarding events and event handling, see Event triggers.

4.15.3 Static configs

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

- height

```
int height;
```

The height of the camera buffer.

- width

```
int width;
```

The width of the camera buffer.

- pitch

```
int pitch;
```

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

- left

```
int left;
```

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

- top

```
int top;
```

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

- right

```
int right;
```

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

- bottom

```
int bottom;
```

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

- 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.

- 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.

- swapByte

```
int swapByte;
```

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

4.15.4 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 the **init** operator of a device.

- 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 will provide 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 implementation of init operator.

```
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;
}
```

- param

```
void *param;
```

The parameter of the callback for **kCameraEvent_SendFrame** event. The Camera Manager will provide the parameter while calling the Init operator, so this param should be stored in the struct of HAL device as part of the implementation of the Init operator.
Note: *This param should be provided when calling the callback function.*

4.15.5 Example

The SLN-VIZN3D-IOT Smart Lock 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;
}
```

4.16 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.

For example, in the VIZN3D kit, the **rk024hh298** panel is connected via the eLCDIF interface and the rk055ahd091 panel is connected via the LCDIF v2 interface.

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, etc.) as necessary.

4.16.1 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;
```

4.16.2 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 setup, start, etc. each of its registered display devices.

For more information about operators, see Operators.

- 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, etc.), turn on the hardware, and perform any other setup the device requires. The callback function to the manager of device 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.

- 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, etc.), 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[5].

- Start

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

Start the display device.
The **Start** operator will be called in the initialization stage of the task of Display Manager 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.

- 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, etc.).
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** should be returned if the frame was successfully sent to the display panel. After calling this operator, the Display Manager will request a new frame.
Note:
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.*

- InputNotify

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

Handle input events.

5 The DeInit function generally will not be called under normal operation.

The **InputNotify** operator is called by the Display Manager whenever a `kFWKMessageID_InputNotify` message is received by and forwarded from the message queue of Display Manager.
For more information regarding events and event handling, see Event triggers.

4.16.3 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 the **init** operator of a device.

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

- height

```
int height;
```

The height of the display buffer.

- width

```
int width;
```

The width of the display buffer.

- pitch

```
int pitch;
```

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

- left

```
int left;
```

The left edge of the active area[6] in the display frame buffer.

- top

```
int top;
```

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

- right

```
int right;
```

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

- bottom

```
int bottom;
```

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

- 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.

- 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.

- srcFormat
  The source format of the requested display frame buffer.

6 The active area indicates the area of the display frame buffer that will be utilized.

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

- frameBuffer
  Pointer to the display frame buffer.
- 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.
Note: *Currently, only the kDisplayEvent_RequestFrame event callback is implemented in the Display Manager.*
The Display Manager will provide 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 implementation of the init operator.

```
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.

- param

```
void *param;
```

The parameter of the Display Manager callback.
Note: *The param field is not currently used by the framework in any way.*

4.16.4 Example

The SLN-VIZN3D-IOT Smart Lock 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;
}
```

4.17 VAlgo devices

The Vision Algorithm HAL device type represents an abstraction for computer vision algorithms which are used for 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 which communicate information about the results of inferencing which is handled by the device. For example, in the Smart Lock 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 Event triggers.

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

4.17.1 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 is as shown below:

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

4.17.2 Operators

Operators are functions which **operate** on an HAL device itself. Operators are akin to **public methods** in object oriented-languages, and are used by the Vision Algorithm Manager to setup, start, etc. its registered vision algo device.

For more information about operators, see Operators.

- Init

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

Init the vision algo HAL device.
Init should initialize any hardware resources the device requires (I/O ports, IRQs, etc.), turn on the hardware, and perform any other setup required by the device.
The callback to the manager is of device typically installed as part of the Init function as well. This operator will be called by the vision algorithm manager when the output manager task first starts.

- Deinit

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

The DeInit function is used to **deinitialize** the algorithm device. DeInit should release any hardware resources the device uses (I/O ports, IRQs, etc.), turn off the hardware, and perform any other shutdown required by the device.
This operator will be called by the Vision Algorithm Manager when the Vision Algorithm Manager task ends[7].

- 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 will forward this message to the Output Manager in the form of an **inference complete** message.

- 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 Event triggers.

4.17.3 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 the **init** operator of a device.

- 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
```

7 The DeInit function generally will not be called under normal operation.

```
 * 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 will provide 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 implementation of the init operator.

```
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.

- param

```
void *param;
```

The param for the callback (optional).

4.17.4 Private data

- autoStart

```
int autoStart;
```

The flag for automatically starting the algorithm.
If autoStart is 1, the Vision Algorithm Manager will automatically start requesting camera frames for this algorithm device after its init operator is executed.

- frames

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

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

The vision algorithm device needs to specify information for each kind of frame, so that the framework will properly convert and pass only the frames which correspond to the requirement of this algorithm device.
For example, the Smart Lock application uses both 3D Depth and IR camera images to perform liveness detection and face recognition, while using RGB frames solely for use as user feedback to help with aligning the face of the user, and so on. Therefore, the algorithm device needs to 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;
```

4.17.5 Example

Because only one Vision Algorithm device can be registered at a time per the design of the framework, the SLN- VIZN3D-IOT Smart Lock project has one Vision Algorithm device implemented.

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;
}
```

4.18 Low power devices

The Low Power/LPM HAL device represents an abstraction used to implement a device which controls the power management of the device by configuring the chip-level power mode (normal operation, SNVS, and so on).

Unlike other devices which may represent a real, physical device, the low power HAL device is purely a **virtual** abstraction mechanism representing the power regulation controls of the chip. As a result, the low power HAL device is platform-dependent because it relies on the different power modes and configuration options made available by the platform being used. Additionally, only one low power HAL device can (and is necessary to) be registered at a time because a chip's power regulatory functionality will not typically require multiple disparate components. This means that the API calls to the Low Power Manager are essentially wrappers over the operators of the single LPM device.

As for functionality, the low power HAL device provides:

- Multi-level low-power switching
- Manual power state configuration
- Automatic power state configuration via periodic idle checks and other flags

The low power mode device also provides an exit mechanism which is called before entering low power mode, to ensure components are properly shut down before sleeping. This is achieved by using a series of timers, one as a periodic idle

check to wait for a specified timeout period before shutting down, and the other as an **exit timer** which reserves a sufficient amount of time for other HAL devices to properly shutdown.

4.18.1 Device definition

The HAL device definition for LPM devices can be found under *framework/hal_api/hal_lpm_dev.h* and is reproduced below:

```
/*! @brief Attributes of a lpm device */
struct _lpm_dev
{
    /* unique id which is assigned by lpm manager during the
 registration */ int id;

    /* operations */
    const lpm_dev_operator_t *ops;
    /* timer */
    TimerHandle_t timer;
    /* pre-enter sleep timer */
    TimerHandle_t preEnterSleepTimer;
    /* lock */
    SemaphoreHandle_t lock;
    /* callback */
```

```
    lpm_manager_timer_callback_t callback;
    /* preEnterSleepCallback */
    lpm_manager_timer_callback_t preEnterSleepCallback;
};
```

The device operators associated with LPM 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 lpm
 device */ typedef
struct _lpm_dev_operator
{

    hal_lpm_status_t (*init)(lpm_dev_t *dev,
                                lpm_manager_timer_callback_t
 callback,
                                lpm_manager_timer_callback_t
 preEnterSleepTimer);
    hal_lpm_status_t (*deinit)(const lpm_dev_t *dev);
 hal_lpm_status_t (*openTimer)(const
    lpm_dev_t *dev);
    hal_lpm_status_t (*stopTimer)(const lpm_dev_t *dev);
    hal_lpm_status_t (*openPreEnterTimer)(const lpm_dev_t
 *dev);
    hal_lpm_status_t (*stopPreEnterTimer)(const lpm_dev_t
 *dev);
    hal_lpm_status_t (*enterSleep)(const lpm_dev_t *dev,
 hal_lpm_mode_t mode);
    hal_lpm_status_t (*lock)(const lpm_dev_t *dev);
    hal_lpm_status_t (*unlock)(const lpm_dev_t *dev); }
lpm_dev_operator_t;

typedef struct _hal_lpm_request
{
    void *dev; /* request dev handle */
    char name[LPM_REQUEST_NAME_MAX_LENGTH]; /* request
name */ } hal_lpm_request_t;
```

4.18.2 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 Low Power Manager to setup, start, etc. its registered low power device.

For more information about operators, see Operators.

- Init

```
hal_lpm_status_t (*init)(lpm_dev_t *dev,
 lpm_manager_timer_callback_t callback,
                     lpm_manager_timer_callback_t#
‹→preEnterSleepTimer);
```

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

The callback to the manager of the device 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.

- Deinit

```
hal_lpm_status_t (*deinit)(const lpm_dev_t *dev);
```

Deinitialize the lpm device.
DeInit should release any hardware resources the lpm device uses (I/O ports, IRQs, etc.), 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[8].

- OpenTimer

```
hal_lpm_status_t (*openTimer)(const lpm_dev_t *dev);
```

Kicks off the periodic idle check timer.

- StopTimer

```
hal_lpm_status_t (*stopTimer)(const lpm_dev_t *dev);
```

Stops the periodic idle check timer.
After all busy requests (Bluetooth LE connection established, face registration in progress) have ceased, this function will be called and begin the shutdown process for other HAL devices.

- OpenPreEnterTimer

```
hal_lpm_status_t (*openPreEnterTimer)(const lpm_dev_t *dev);
```

Kicks off the preEnterSleepTimer.
The **preEnterSleepTimer** is used to provide other HAL devices sufficient time to properly shutdown before the board enters sleep mode. This function will be called after the periodic idle check timer has stopped (due to a timeout).

- StopPreEnterTimer

```
hal_lpm_status_t (*stopPreEnterTimer)(const lpm_dev_t *dev);
```

Stops the preEnterSleepTimer.
This function is called to stop the timer associated with the pre-sleep shutdown process. After this timer ends, the EnterSleep function will be called and the device will power down.

- EnterSleep

```
hal_lpm_status_t (*enterSleep)(const lpm_dev_t *dev,
 hal_lpm_mode_t mode);
```

Enter sleep mode using the low power mode specified in the function call[9].

- Lock

```
hal_lpm_status_t (*lock)(const lpm_dev_t *dev);
```

Acquire the lock for the low power device.
The low power manager uses a lock-based system to prevent accidentally entering sleep mode before all devices are ready to enter sleep. The Lock function is called by the Low Power manager in response to an HAL device signaling that it is performing a critical function which requires that the board does not enter sleep until complete.

8 The DeInit function generally will not be called under normal operation.
9 The power modes available vary based on the platform in use.

- Unlock

```
hal_lpm_status_t (*unlock)(const lpm_dev_t *dev);
```

Release the lock for the low power device.
The low power manager uses a lock-based system to prevent accidentally entering sleep mode before all devices are ready to enter sleep. The Unlock function is called by the Low Power manager in response to an HAL device signaling that it is finished performing a critical function which required that the board did not enter sleep until it was completed.

4.18.3 Components

- timer

```
/* timer */
TimerHandle_t timer;
```

This timer is use to periodically check busy requests from other HAL devices.

- preEnterSleepTimer

```
/* pre-enter sleep timer */
TimerHandle_t preEnterSleepTimer;
```

This timer is used to provide a sufficient amount of time for HAL devices to shutdown prior to entering sleep mode.

- lock

```
/* lock */
SemaphoreHandle_t lock;
```

This lock is used to maintain thread safety when multiple task need to call the Low Power Manager, and is managed by the Low Power Manager.

- callback

```
/* callback */
lpm_manager_timer_callback_t callback;
```

Callback to the Low Power Manager. The HAL device invokes this callback to notify the vision algorithm manager of specific events.
The Low Power Manager will provide 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 implementation of the init operator.

```
hal_lpm_status_t HAL_LpmDev_Init(lpm_dev_t *dev,
                    lpm_manager_timer_callback_t callback,
                    lpm_manager_timer_callback_t
 preEnterSleepCallback)
{
    int ret = kStatus_HAL_LpmSuccess;

    dev->callback = callback; dev->preEnterSleepCallback =
    preEnterSleepCallback;
```

- PreEnterSleepCallback

```
/* preEnterSleepCallback */
lpm_manager_timer_callback_t preEnterSleepCallback;
```

Callback function which is called after the “preEnterSleep” timer terminates.
Note: *This callback comes from the LPM Manager*

4.18.4 Example

Because only one low power device can be registered at a time per the design of the framework, the SLN-VIZN3D-IOT Smart Lock project has only one low power device implemented.

The source file for this low power device can be found at *HAL/common/hal_sln_lpm.c*.

In this example, we will demonstrate the use of a low power device (using FreeRTOS for timers, etc.) in conjunction with a device/manager of a different type.

The LPM manager device implements all the power switching functionality we need, while the *secondary device/manager* will attempt to make busy requests (lock the LPM device) and enable/disable low power mode.

4.18.4.1 LPM manager device

```
/* Here call periodic callback to check idle status. */ static
 void
HAL_LpmDev_TimerCallback(TimerHandle_t handle)
{
    if (handle == NULL)
    {
        return;
    }

    lpm_dev_t *pDev = (lpm_dev_t *)pvTimerGetTimerID(handle);
 if
    (pDev->callback != NULL)
    {
        pDev->callback(pDev);
    }
}

/* Here call preEnterSleepCallback. Duing this time, all device
 have already exit. So# ‹→this callback
will call enterSleep operator to enter low power mode. */
static void HAL_LpmDev_PreEnterSleepTimerCallback(TimerHandle_t
 handle)
{
    if (handle == NULL)
    {
        return;
    }

    lpm_dev_t *pDev = (lpm_dev_t *)pvTimerGetTimerID(handle);
 if
    (pDev->preEnterSleepCallback != NULL)
    {
        pDev->preEnterSleepCallback(pDev);
    }
}

hal_lpm_status_t HAL_LpmDev_Init(lpm_dev_t *dev,
                                 lpm_manager_timer_callback_t
 callback,
                                 lpm_manager_timer_callback_t
 preEnterSleepCallback)
{
    int ret = kStatus_HAL_LpmSuccess;
```

```
    dev->callback        = callback;
    dev->preEnterSleepCallback = preEnterSleepCallback;

    /* put low power hardware init here */

    /* put periodic timer create and init here */
    dev->timer = xTimerCreate("LpmTimer", pdMS_TO_TICKS(1000),
 pdTRUE, (void *)dev, HAL_
→LpmDev_TimerCallback);
    if (dev->timer == NULL)
    {
        return kStatus_HAL_LpmTimerNull;
    }

    /* put exit timer create and init here */
    dev->preEnterSleepTimer =
 xTimerCreate("LpmPreEnterSleepTimer", pdMS_TO_TICKS(1500),#
→pdTRUE, (void *)dev,

 HAL_LpmDev_PreEnterSleepTimerCallback);
    if (dev->preEnterSleepTimer == NULL)
    {
        return kStatus_HAL_LpmTimerNull;
    }

    /* put lock create and init here */
    dev->lock = xSemaphoreCreateMutex();
    if (dev->lock == NULL)
    {
        return kStatus_HAL_LpmLockNull;
    }

    /* put init low power mode and status here, detial can find
 in lpm_manager. */
    FWK_LpmManager_SetSleepMode(kLPMMode_SNVS);

 FWK_LpmManager_EnableSleepMode(kLPMManagerStatus_SleepDisable);

    return ret;
}

hal_lpm_status_t HAL_LpmDev_Deinit(const lpm_dev_t *dev)
{
    int ret = kStatus_HAL_LpmSuccess;

    return ret;
}

hal_lpm_status_t HAL_LpmDev_OpenTimer(const lpm_dev_t *dev)
{
    int ret = kStatus_HAL_LpmSuccess;

    if (dev->timer == NULL)
    {
        return kStatus_HAL_LpmTimerNull;
    }

    if (xTimerStart(dev->timer, 0) != pdPASS)
    {
```

```
          ret = kStatus_HAL_LpmTimerFail;
    }

    return ret;
}

hal_lpm_status_t HAL_LpmDev_StopTimer(const lpm_dev_t *dev)
{
    int ret = kStatus_HAL_LpmSuccess;

    if (dev->timer == NULL)
    {
          return kStatus_HAL_LpmTimerNull;
    }

    if (xTimerStop(dev->timer, 0) != pdPASS)
    {
          ret = kStatus_HAL_LpmTimerFail;
    }

    return ret;
}

hal_lpm_status_t HAL_LpmDev_OpenPreEnterSleepTimer(const
 lpm_dev_t *dev) {

    int ret = kStatus_HAL_LpmSuccess;

    if (dev->preEnterSleepTimer == NULL)
    {
          return kStatus_HAL_LpmTimerNull;
    }

    if (xTimerStart(dev->preEnterSleepTimer, 0) != pdPASS)
    {
          ret = kStatus_HAL_LpmTimerFail;
    }

    return ret;
}

hal_lpm_status_t HAL_LpmDev_StopPreEnterSleepTimer(const
 lpm_dev_t *dev) {

    int ret = kStatus_HAL_LpmSuccess;

    if (dev->preEnterSleepTimer == NULL)
    {
          return kStatus_HAL_LpmTimerNull;
    }

    if (xTimerStop(dev->preEnterSleepTimer, 0) != pdPASS)
    {
          ret = kStatus_HAL_LpmTimerFail;
    }

    return ret;
}
```

```
hal_lpm_status_t HAL_LpmDev_EnterSleep(const lpm_dev_t *dev,
 hal_lpm_mode_t mode) {

    int ret = kStatus_HAL_LpmSuccess;
    switch (mode)
    {
         case kLPMMode_SNVS:
         {
             /* put enter SNVS low power mode here*/
         }
         break;

         default:
             break;
    }

    return ret;
}

hal_lpm_status_t HAL_LpmDev_Lock(const lpm_dev_t *dev)
{
    uint8_t fromISR = __get_IPSR();

    if (dev->lock == NULL)
    {

         return kStatus_HAL_LpmLockNull;
    }
    if (fromISR)
    {
         BaseType_t HigherPriorityTaskWoken = pdFALSE;
         if (xSemaphoreTakeFromISR(dev->lock,
 &HigherPriorityTaskWoken) != pdPASS)
         {
               return kStatus_HAL_LpmLockError;
         }
    }
    else
    {
         if (xSemaphoreTake(dev->lock, portMAX_DELAY) !=
 pdPASS)
         {
               return kStatus_HAL_LpmLockError;
         }
    }

    return kStatus_HAL_LpmSuccess;
}

hal_lpm_status_t HAL_LpmDev_Unlock(const lpm_dev_t *dev)
{
    uint8_t fromISR = __get_IPSR();

    if (dev->lock == NULL)
    {
         return kStatus_HAL_LpmLockNull;
    }

    if (fromISR)
    {
```

```
         BaseType_t HigherPriorityTaskWoken = pdFALSE;
         if (xSemaphoreGiveFromISR(dev->lock,
 &HigherPriorityTaskWoken) != pdPASS)
        {
             return kStatus_HAL_LpmLockError;
         }
    }
    else
    {
         if (xSemaphoreGive(dev->lock) != pdPASS)
         {
             return kStatus_HAL_LpmLockError;
         }
    }

    return kStatus_HAL_LpmSuccess;
}

static lpm_dev_operator_t s_LpmDevOperators = {
    .init          = HAL_LpmDev_Init,
    .deinit        = HAL_LpmDev_Deinit,
    .openTimer     = HAL_LpmDev_OpenTimer,
    .stopTimer     = HAL_LpmDev_StopTimer,
    .openPreEnterTimer    = HAL_LpmDev_OpenPreEnterSleepTimer,
    .stopPreEnterTimer    = HAL_LpmDev_StopPreEnterSleepTimer,
    .enterSleep    = HAL_LpmDev_EnterSleep,
    .lock          = HAL_LpmDev_Lock,
    .unlock        = HAL_LpmDev_Unlock,
};

static lpm_dev_t s_LpmDev = {
    .id    = 0,
    .ops   = &s_LpmDevOperators,
};

int HAL_LpmDev_Register()
{
    int ret = 0;

    FWK_LpmManager_DeviceRegister(&s_LpmDev);

    return ret;
}
```

4.18.4.2 Requesting device

As part of this example, we assume an LPM device is running at the same time as a **requesting device** (camera, vision algo, etc.) of a different type which is performing some critical functionality.

Supposing this example **requesting device** (aptly named **ExampleDev**) performs some critical functionality inside `HAL_InputDev_ExampleDev_Critical` will set the request busy by calling `FWK_LpmManager_RuntimeGet`, thus acquiring the lock which prevents changes to the current power mode state.

After the device has completed its critical functionality, it will use `FWK_LpmManager_RuntimePut` to release the lock which prevents changes to the current power mode state.

```
static hal_lpm_request_t s_LpmReq = {
    .dev = &s_InputDev,
    .name = "lpm device",
};

int HAL_InputDev_ExampleDev_Critical(void)
{
    FWK_LpmManager_RuntimeGet(&s_LpmReq);

    /* perform critical function here */

    FWK_LpmManager_RuntimePut(&s_LpmReq);
}

int HAL_InputDev_ExampleDev_Register(void)
{
    hal_input_status_t status = kStatus_HAL_InputSuccess;

    status = FWK_LpmManager_RegisterRequestHandler(&s_LpmReq);

    return status;
}
```

4.19 Flash devices

The flash HAL device represents an abstraction used to implement a device which handles all operations dealing with flash[10] (permanent) storage.

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 File manager only allows one flash device to be registered because there is usually no need for multiple file systems operating at the same time.

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

4.19.1 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 */
```

10 Even though the word **flash** is used in the terminology of this device, the user is technically capable of implementing a FS which uses a volatile memory instead. One potential reason for doing so would be to run logic/sanity checks on the APIs of the filesystem before implementing them on 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 size);
    sln_flash_status_t (*mkdir)(const flash_dev_t *dev, const
 char *path);
    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;
```

4.19.2 Operators

Operators are functions which **operate** on an HAL device itself. Operators are akin to **public methods** in object oriented-languages.

For more information about operators, see Operators.

- Init

```
sln_flash_status_t (*init)(const flash_dev_t *dev);
```

Initialize the flash & filesystem.
Init should initialize any hardware resources required by the flash device (pins, ports, clock, etc)[11]. In addition to initializing the hardware, the init function should also mount the filesystem. [12]

11 An application that runs from flash (does XiP) should not initialize/deinitialize any hardware. If a hardware change is truly needed, the change should be performed with caution.

- Deinit

```
hal_lpm_status_t (*deinit)(const lpm_dev_t *dev);
```

Deinitialize the flash and filesystem.
DeInit should release any hardware resources a flash device might use (I/O ports, IRQs, etc.), turn off the hardware, and perform any other shutdown the device requires[13].

- Format

```
sln_flash_status_t (*format)(const flash_dev_t *dev);
```

Clean and format the filesystem.

- 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.

- 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 located at path.
Setting overwrite[14] equal to true will cause append from the beginning of the file instead.

- Read

```
sln_flash_status_t (*read)(const flash_dev_t *dev, const char
 *path, void *buf, unsigned#
‹→int size);
```

Read a file from the filesystem located at path and store the contents in buf[15].
Note: *For now, the current functionality of read is limited. In the future we plan to extend this function to support offset read, file physical address, true length return, and more.*

- Make Directory

```
sln_flash_status_t (*mkdir)(const flash_dev_t *dev, const char
 *path);
```

Create a directory located at path.
Note: *If the filesystem in use does not support directories, this operator can be set to NULL.*

- Remove

```
sln_flash_status_t (*rm)(const flash_dev_t *dev, const char
 *path);
```

Remove the file located at path.

12 Some lightweight FS may not require mounting and can be prebuilt/preloaded on the flash instead. Regardless, the init function should result in the filesystem being in a usable state.
13 An application that runs from flash (does XiP) should not initialize/deinitialize any hardware. If a hardware change is truly needed, the change should be performed with caution.
14 overwrite = true makes this function nearly equivalent to the save function, the only difference being that this will not create a new file.
15 It is up to the user to guarantee that the buffer supplied will fit the contents of the file being read.

Note: *If the filesystem in use does not support directories, this operator can be set to NULL.*

- 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.

- 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 which can lead to unwanted delays when searching for the next available sector.
`timeout_ms` specifies how much time to wait before performing another cleanup. This helps prevent against multiple HAL devices calling cleanup and stalling the filesystem.

4.19.3 Example

Because only one flash device can be registered at a time per the design of the framework, the SLN-VIZN3D-IOT Smart Lock 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 will demonstrate a way to integrate the well known Littlefs device with the framework.

Littlefs is a lightweight file-system that is designed to handle random power failures. The architecture of the file-system allows having both 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
└── wifi_info
    ├── wifi_info
    └── ftp_info
```

4.19.3.1 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;
}
```

4.20 Event overview

Events are a means by which information is communicated between different devices via their managers.

4.20.1 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 which first received the event will communicate that event to its manager, which in turn will notify other managers designated to receive the event.

For example, when a button is pressed, a flow similar to the following will take place:

1. The **Push Button** HAL device will receive an interrupt corresponding to the button that was pressed.
2. Inside the interrupt handler of the HAL device, the device will associate an event with the button that was pressed.
3. The HAL device will specify which managers should receive the event.
4. The HAL device will forward the event to its manager.

The code which 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");
        }
    }
}
```

```
    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;
```

4.20.2 Types of events

Events can be used to communicate all sorts of information. The default two types of events are defined as InferComplete events and InputNotify Events.

Both types of events represent different information being communicated to and by the HAL devices.

4.20.2.1 InferComplete events

Inference events are used to indicate that a vision/voice algorithm HAL device has completed a stage in its inference pipeline.

In the Smart Lock application, this 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 will also communicate 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 will also include 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.

4.20.2.2 InputNotify Events

Input events are events which indicate that input has been received by an input HAL device. 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 as well. 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.

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 triggers.

4.21 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 operator struct of the device 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 this, 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.

Note: *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).*

4.21.1 Default handlers

Default event handlers are exactly what their name would suggest that the default means by which a device handles events. A HAL device's default event handlers (InputNotify, InferComplete, etc.) can be found in the HAL device driver itself.

Nearly every device has a default handler implemented[16], although most devices will only actually handle a few types of events.

```
static hal_display_status_t
 HAL_DisplayDev_LcdifRk024hh2_InputNotify(const display_dev_t#
↪*receiver, void *data)
```

16 Devices which do not have a handler implemented can be extended to have one by using a similar device as an example.

```
{
    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 know not to 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 handlers.

4.21.2 App-specific handlers

App-specific handlers are device handlers which are defined for a specific **app**, such as, the Smart Lock project or the upcoming Touchless HMI project.

Not every device will need to implement an app-specific handler, but because default handlers are implemented using

WEAK functions[17], 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 will 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*.

4.22 Naming conventions

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

Note: *The naming conventions described below apply ONLY to framework-related code which is primarily located in the HAL, framework, and source folders.*

4.22.1 Functions

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

17 The power modes available vary based on the platform in use.

- Ex.

```
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 which 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.*

- Ex.

```
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);
```

Following one of the above prefixes is the device type of the device defining the function.

– InputDev

– OutputDev

– CameraDev

– DisplayDev

– etc.

Following the device type is the name of the device. This name should match the name of the device specified in the file name.

- Ex.

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

Finally, following the name of the device is the **action** which is being performed on/by the device. This 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);
```

4.22.2 Variables

Local and global variables both 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`.

- Ex.

```
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;
```

4.22.3 Typedefs

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

- Ex.

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

4.22.4 Enums

Enumerations are written in the form `kEventType_State`.

- Ex.

```
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 are be written in the form kStatus_{Component}_{State}.

- Ex.

```
/*! @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;
```

4.22.5 Macros and Defines

Defines are written in all caps.

- Ex.

```
#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
```

5 Smart lock

5.1 Smart lock introduction

As briefly mentioned in Introduction, the Smart Lock application uses a two layer architecture containing a Framework + HAL layer, and an **Application** layer.

The **Smart Lock** application is a demo reference project which uses proprietary 3D face recognition of NXP and detection engine to implement all the functionality necessary for a full-fledged Smart Lock product. The Smart Lock application comes with many features out of the box, including:

- Local (offline) face registration + recognition
- Remote face registration + recognition via smartphone/tablet
- **Liveness detection** for protection against spoof attacks
- Low power integration for battery-based applications

Note: *Be sure to check out the Getting Started Guide and SLN-VIZN3D-IOT Kit User Guide (document SLN-VIZN3D-IOT-UG) for an overview of the out of box features available in the SLN-VIZN3D-IOT Smart Lock application.*

This section will focus on the Application Layer code which is written on top of the underlying Framework + HAL layer, and is specific to the Smart Lock application.

5.2 HAL WiFi introduction

The SLN-VIZN3D-IOT board is equipped with an **Azurewave Technologies AW-AM510 WiFi Module**, which supports 802.11a/b/g/n on the 2.4 GHz and 5 GHz bands, as well as supporting Bluetooth 5.1.

For the time being, the requirements for the WiFi cover:

- Module hardware initialization
- Local access point connection
- Connect to a non-secure FTP server with anonymous credentials

5.2.1 Integrating the WiFi module

The WiFi module is a special type of peripheral which acts as both an input and an output. Despite this fact, integrate the WiFi module with the framework by registering both an input device and an output device.

Note: *Because the module is registered to two managers at the same time, some degree of synchronization needs to be implemented to avoid various race conditions.*

5.2.1.1 WiFi task

The WiFi initialization task will automatically create several other tasks to handle TCP Connections, module communication, and the WiFi dispatcher. However, because WiFi operations are slow compared to the speed at which the CPU operates, the calling task will block waiting for an answer[18].

To avoid blocking important resources like Output manager task or Vision input manager task, the WiFi device must maintain a standalone execution task in which waiting operations can be performed with minimal impact to the rest of the system.

The communication between the WiFi, Input Manager, and Output Manager tasks uses an event-driven design which can help to decrease the number of potential errors caused by race conditions.

Figure 17 presents the components involved.

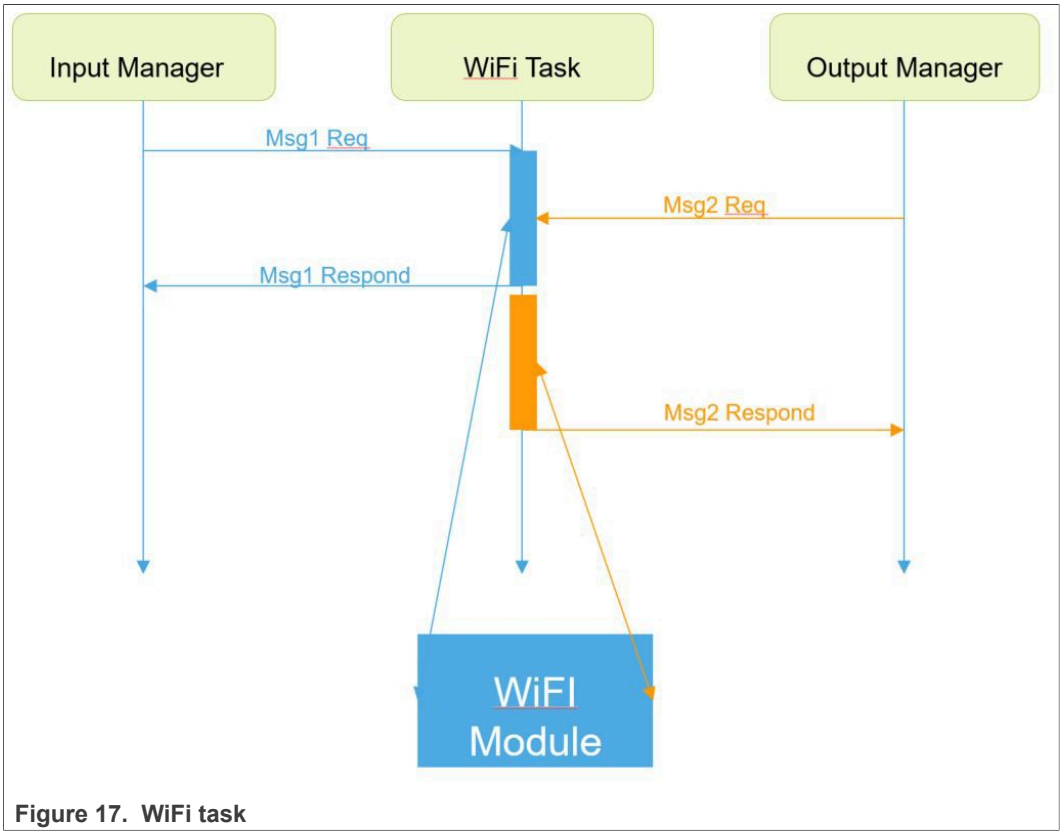

Figure 17. WiFi task

To connect to an access point, the function logic is as follows:

1. `BOARD_InitWIFIAW_AM510Resource()`: Initialize hardware pins.
2. `WPL_Init()`: Load the WiFi firmware on the AW-AM510 chip.
3. `WiFi_GetCredentials` (&credentials): Get the saved credentials.
4. `WPL_Join` (credentials): Join the network specified by the credentials.

18 To alleviate this issue, the NXP driver supports asynchronous operation, however the code complexity will grow and the need for synchronization will increase.

5.2.1.2 FTP client

The Smart Lock application is capable of maintaining brief videos which have been captured by the camera. This feature can be useful in applications where a user wants to review Important activity captured by the camera.

After encoding a captured image frame, the output manager will send a message to the WiFi task via the InferenceComplete callback which contains a pointer to the location and the size of the video. After that, the WiFi task will kick off the procedure to establish a connection with a FTP server.

Connecting to a FTP server uses a handshake procedure similar to the process of connecting to an access point. The steps are as follows:

1. FTP_Init
2. FTP_ConnectBlocking
3. FTP_StoreBlocking
4. FTP_DisconnectBlocking

```
FTP_Init();
ftp_session_handle_t handle = FTP_ConnectBlocking();
if (handle != NULL)
{
    FTP_StoreBlocking(handle, remote_path, recordedDataAddress,
 recordedDataSize);
    FTP_DisconnectBlocking(handle);
}
```

```
/**
 * @brief Fetches the Server Info from the flash, init internal
 structures.
 *
 * @return kStatus_Success on success
 */
status_t FTP_Init(void);

/**
 * @brief Connect to the server specified by saved server_info
 *
 * @return Return a handler which is to be used with the store/
disconnect function. NULL#
‹→if no connection was done
 */

ftp_session_handle_t FTP_ConnectBlocking(void);
```

Note: *For now only anonymous unsecured FTP connections are supported. In order to supply credentials for the FTP server, modify the FTP_ConnectBlocking function in ftp_client.c.*

```
    ftpSession.user = "anonymous" /* set manually the user */;
    ftpSession.pass = "anonymous@domain.com" /* set manually
 the password */;
```

After changing the code, rebuild and reflash the application.

```
```c
```
```

```
/**
 * @brief Store the data at a specific remote location.
 *
 * @param sessionHandler Session handler obtain after the
 connect operation took place
 * @param remote_path Remote path at which to save the file.
 Should contain the name of#
‹→the file
 * @param data_source Data to be saved
 * @param len Length of the file
 * @return kStatus_Success on success. If the status was fail,
 automatic disconnection#
‹→takes place
 */
status_t FTP_StoreBlocking(ftp_session_handle_t sessionHandler,
 const char *remotePath,#
‹→char *dataSource, uint32_t len);
/**
 * @brief Disconnect from a connected server
 *
 * @param sessionHandler
 * @return kStatus_Success on success
 */
status_t FTP_DisconnectBlocking(ftp_session_handle_t
 sessionHandler);
```

6 Revision history

Rev.	Date	Description
0	19 August 2022	Initial release

7 Legal information

7.1 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.

7.2 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.

7.3 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

Date of release: 19 August 2022
Document identifier: SLN-VIZN3D-IOTDG