Overview

AWS IoT Smart Home Device SDK (Beta)

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

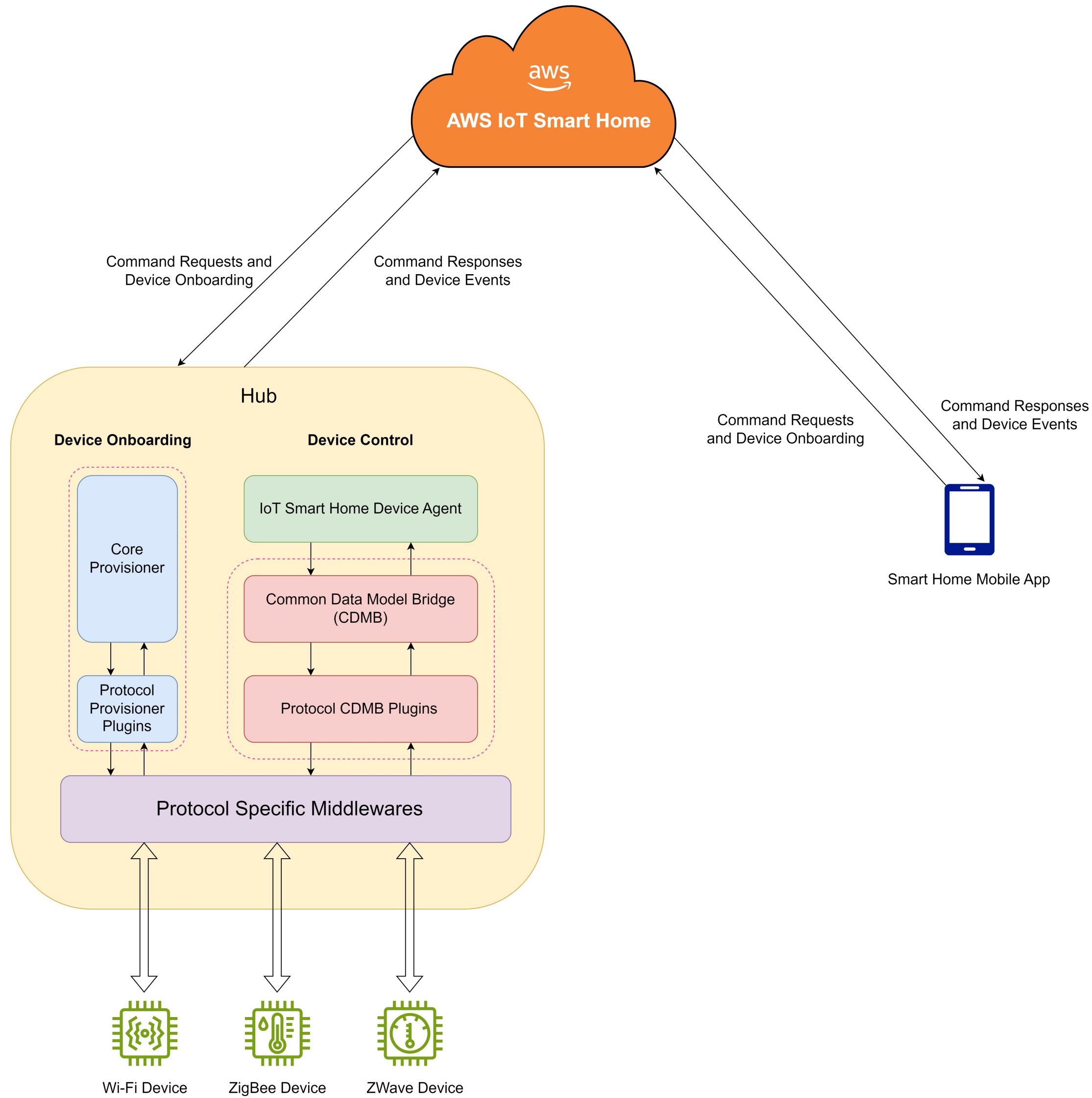
Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

This prerelease documentation is confidential and is provided under the terms of your nondisclosure agreement with Amazon Web Services (AWS) or other agreement governing your receipt of AWS confidential information.

AWS IoT Smart Home being provided to you by AWS is a "Beta Service" as defined in the AWS Service Terms. Your use of the Beta Service is governed by your Agreement with AWS and the AWS Service Terms. AWS is providing you access to the Beta Service for your non- public facing, internal testing only. You may not use any data from your customers as part of your use of the Beta Service; you may only use test data and endpoints. Your use of the Beta Service is at your own risk and you are solely responsible for ensuring your use of the Beta Service is safe, reasonable, and compliant with all applicable laws. The Beta Service is confidential and you may not discuss the features, functionality, or documentation related to the Beta Service with any parties that are not authorized by AWS. You may experience some disruption as we update the Beta Service with fixes or otherwise. This Beta Service
is also subject to change and cancellation at any time, and AWS may remove your access to the Beta Service at any time, upon which all of your data in the Beta Service will be lost unless you yourself have previously backed it up. The Beta Service is not intended for use
in production environments, or to support your production workloads, and your experience with the Beta Service may materially differ from your experience with any production equivalent AWS makes generally available.

1. Introduction

AWS IoT Smart Home is a fully-managed service that enables smart home service providers to quickly build solutions by leveraging a unified control experience across different device protocols and connectivity types. The service includes a Device SDK and a cloud service to support devices connected over Z-Wave, ZigBee, and Wi-Fi. The service also includes device onboarding flows, including bar-code scanning and zero-touch provisioning.



AWS IoT Smart Home can primarily be used for following two use cases.

1. **Device Onboarding**
2. **Device Control**

1.1 Device Onboarding

For device onboarding, AWS IoT Smart Home provides an abstraction layer for discovering and onboarding IoT devices to the network. It provides a unified interface to customers in a protocol agnostic manner, so customers can focus on building end-user focused applications. A new device can be onboarded by the end customer using the following three ways:

1. **Simple Setup (SS):** In this method, the end-user powers on the IoT device. Then scan the QR code present on the device using the AWS customer's smart home app. This will enroll the device on the AWS IoT smart home cloud and connect it to the IoT Hub.
2. **User Guided Setup (UGS)**: In this method, the end-user powers on the device and interacts with IoT Hub and/or device to onboard the device to AWS IoT Smart Home. End-user may need to press button on IoT hub (or soft button in AWS customer's smart home app), or press buttons on both Hub and device. This is also the failover mode when SS mode fails.
3. **Zero Touch Setup (ZTS) (Optional):** In this method, the end-user powers on the device (e.g., lightbulb) and the IoT Hub automatically initiates the device onboarding to AWS IoT Smart Home. One of the key requirements for enabling Zero Touch Setup (ZTS) is that the device has to be pre-associated to the end-user's account. The pre-association of the device takes place during the fulfillment process. Fulfillment can be executed by the smart home service providers or a 3P retailer as long as the device can be associated to an end-user's AWS IoT smart home account.

ZTS can only be enabled for the devices which are pre-associated to an end-user's account. C2C(cloud-to-cloud) devices (e.g., Nest Thermostat) require the device to be first onboarded to the 3rd party cloud provider's platform, and therefore cannot be supported for a ZTS experience.

1.2 Device Control

In case of device control, AWS IoT Smart Home service is responsible for registering devices in and executing command/control for such devices. AWS IoT Smart Home service provides a vendor/protocol agnostic device management experience and enables smart home service provides to build end-user experiences without having knowledge of device specific protocols.

AWS IoT Smart Home service will allow customers to view device resources (e.g., *Brightness* of the light bulb, *on/off* state of the device) and also view & modify Device Resource States (e.g., set *Brightness* to xyz, *open* garage door). AWS IoT Smart Home service will also emit events for every state change received from devices, thereby allowing Customers to build analytics, rules/routines, monitoring over the state changes.

High-level feature of device control functionality are as follows:

1. **Modify/Read device state**: Allows the customers to view device attributes/capabilities and send commands to change the device attributes. These attributes/commands dependent on the device types and the C2C APIs available from 3P cloud providers for integration. Following states can be accessed by the customer.

   a. Device State: Value of the device attributes.
   b. Connectivity State: Whether the device is reachable (connected, offline)
   c. Health Status: Value of attributes such as battery level, received signal strength indicator (RSSI) strength, etc.
2. **State Change Notification**: Emit events for changes in device attributes/connectivity states (e.g., *brightness* of the lightbulb is *XX*, front door *status* is *unlocked*).
3. **Local Control:** Even if the IoT hub is not connected to the internet, the devices can send commands to the other devices connected to the same IoT hub. Once the device comes back online, the current state of the devices will be updated on the cloud.
4. **Synchronize state changes**: It will synchronize the state changes received from disparate sources. For example, end-user changes thermostat temperate via a) Smart home app, b) device manufacturer app, and c) manually on device.

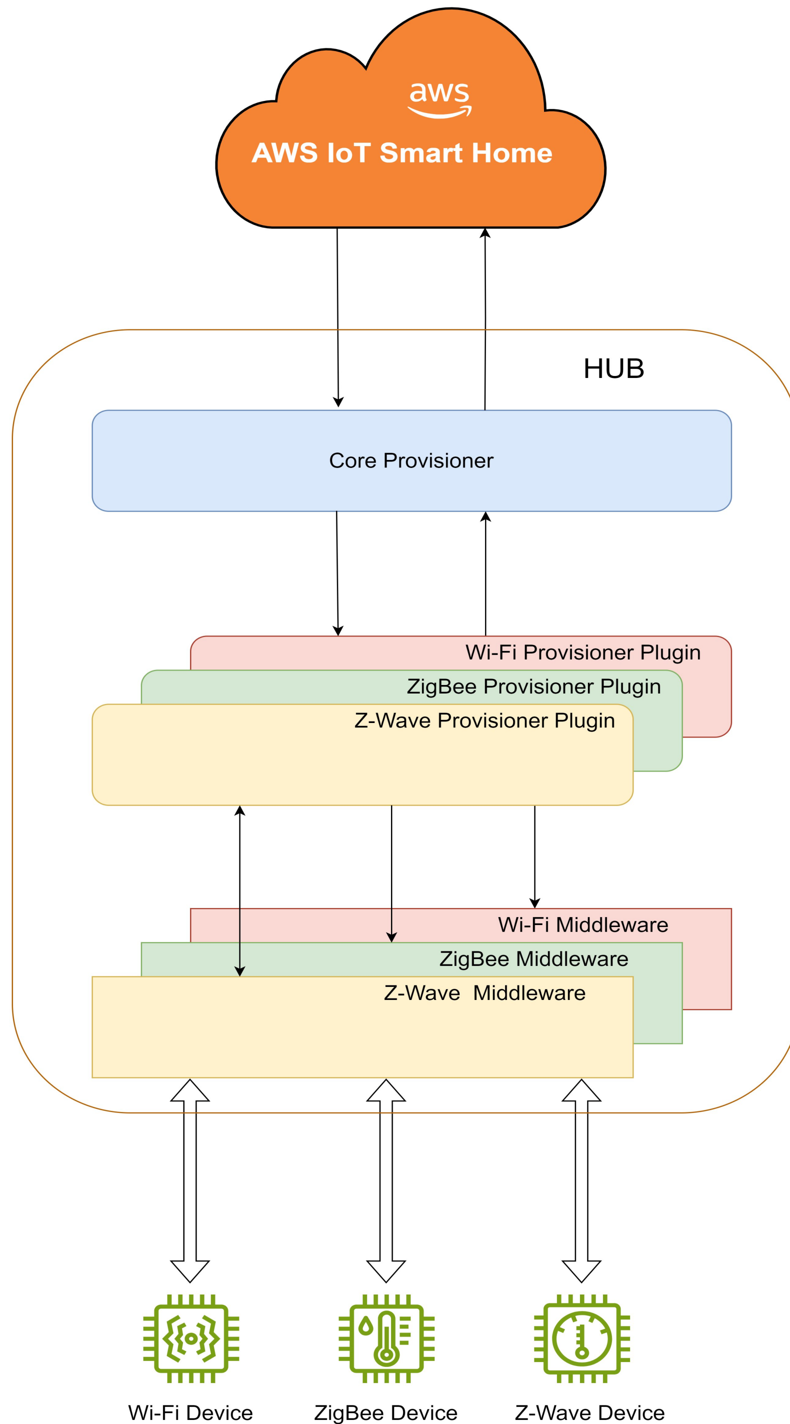
2. Smart Home Device SDK Architecture

To enable both **Device Onboarding** and **Device Control** on the IoT Hub, the Smart Home device SDK has been created. Smart Home device SDK consist of the following components which enable the onboarding and control of a device connected to the IoT hub.

- **IoT Smart Home Edge Agent:** It acts as a gateway between the IoT Hub and AWS IoT Smart Home cloud.
- **Common Data Model Bridge (CDMB):** It is the interface between the IoT Smart Home Device Agent and the Protocol specific CDMB plugins.
- **Protocol specific CDMB plugins:** These are responsible for converting the command in AWS IoT Smart Home cloud format to protocol specific command.
- **Core Provisioner:** It is the key component which interacts with both AWS IoT Smart Home cloud and protocol specific provisioner plugins to enable device onboarding on the IoT Hub.
- **Protocol specific provisioner plugins:** These are responsible to perform protocol specific tasks to onboard a new device to the IoT Hub.
- **Protocol specific Middlewares:** Protocol specific middlewares have a critical role of interacting with underlying protocol stacks. Both device onboarding and device control components use it to interact with end device.

Each component except Middleware is either used for to enable device onboarding or device control on the IoT Hub. The next two sections contain the details of each Smart Home device SDK component based on their function.

2.1 Device SDK components enabling Device Onboarding



The above diagram represent the device SDK components used for device onboarding and their interactions.
Core Provisioner and Protocol specific provisioner plugins run as a single process on the hub.

2.1.1 Core Provisioner

Core Provisioner is the key component which interacts with both AWS IoT Smart Home cloud and protocol specific provisioner plugins to enable device onboarding on the IoT Hub. Responsibilities of core provisioner are as follows:

1. **MQTT connection**: Establish the connection with the AWS IoT Smart Home MQTT broker, allowing the provisioner to publish and subscribe to relevant topics on the cloud.
2. **Message Queue and Handler**: All incoming messages are serialized and saved in a queue, and are processed in an orderly manner. These messages contain requests to add/remove end devices from the IoT Hub.
3. **Interface with Z-Wave/ZigBee/Matter Plugin**: It interfaces with the protocol specific provisioner plugins to facilitate device onboarding by passing the relevant auth material and setting the correct joining modes in the radios.
4. **IPC Client to CDMB**: To receive the device capability report from CDMB. Device capability report is created by the protocol specific CDMB plugins and then sent to core Provisioner via CDMB. Once the core Provisioner has received the device capability report, it then sends it to the AWS IoT Smart Home cloud.

2.1.2 Protocol specific provisioner plugins.

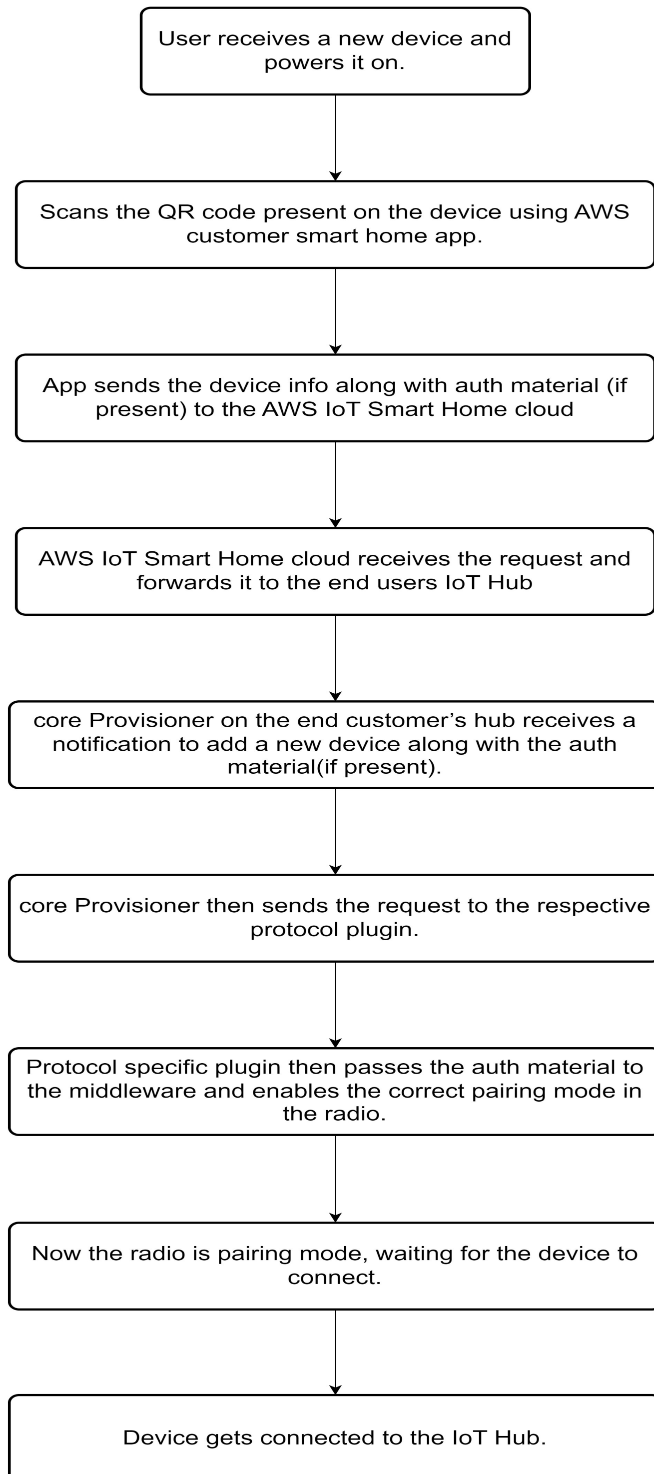
The Protocol specific provisioner plugin libraries are responsible for the following functions.

1. Initialize the protocol specific Middleware.
2. In case of ZTS, receive the protocol specific auth material from the core Provisioner to add a new device on the hub and pass it to the middleware.
3. Setup the correct joining mode in the radio based on the request received from the core Provisioner to add a new device.
4. When a request to remove a device is received from the core Provisioner, then delete the relevant auth material from the Middleware and call the Middleware APIs to remove the device from the network.

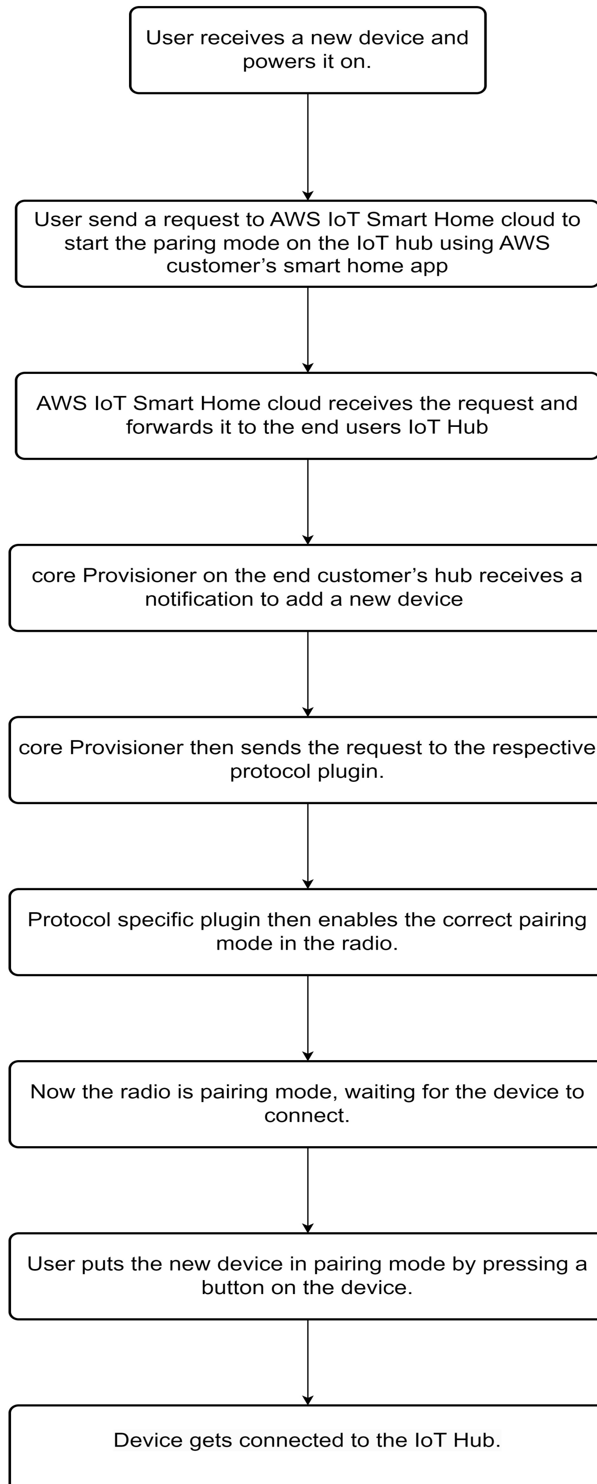
2.1.3 Protocol specific Middlewares.

Protocol specific middlewares have a critical role of interacting with underlying protocol stacks. They are used for both device onboarding and device control. So, Middleware architecture and use cases are separately discussed later in the middleware section.

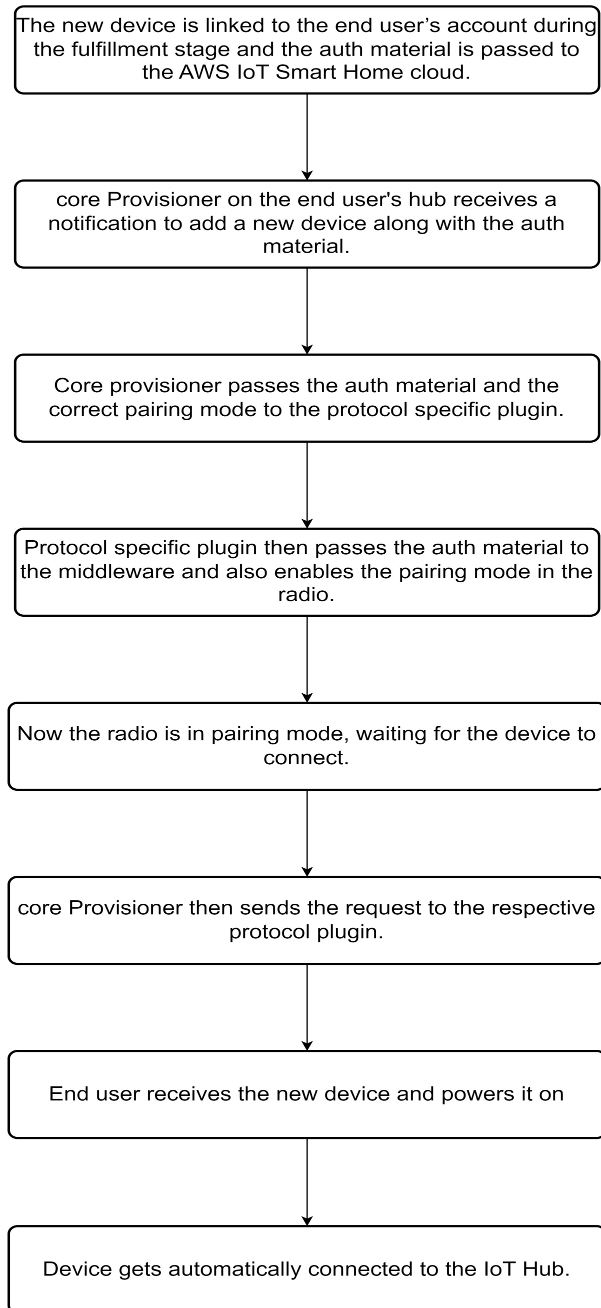
2.1.4 Simple Setup device onboarding flow



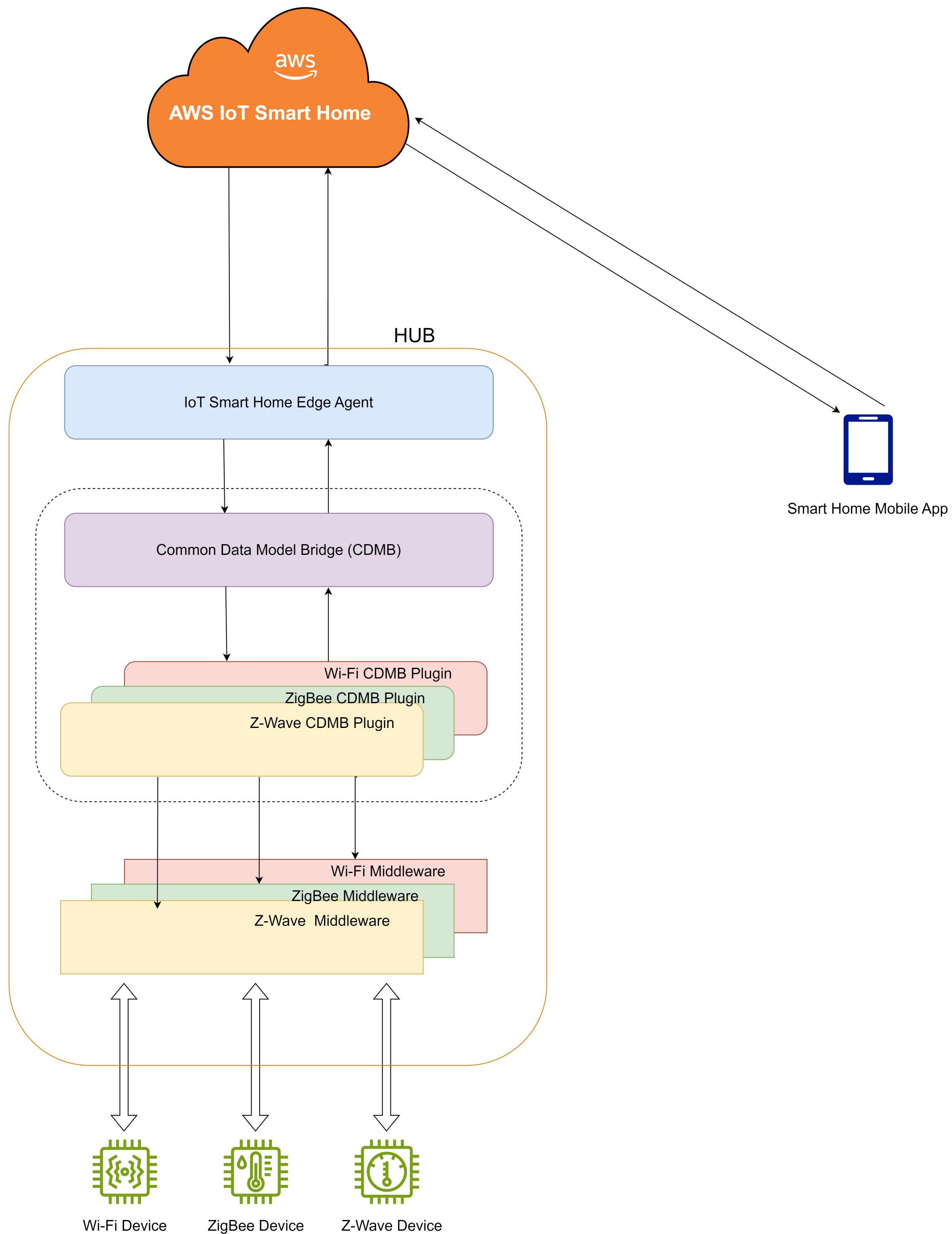
2.1.5 User Guided Setup (UGS) device onboarding flow



2.1.6 Zero touch setup (ZTS) device onboarding flow



2.2 Device SDK components enabling Device Control



The above diagram represent the device SDK components used for device control and their interactions.

These components run as the following processes on the hub.

1. IoT Smart Home Edge Agent run as one process.
2. CDMB and Protocol specific CDMB plugins run as another process.
3. Middleware run as a separate process or processes.

2.2.1 IoT Smart Home Edge Agent

IoT Smart Home Edge Agent acts as a gateway between the IoT Hub and AWS IoT Smart Home cloud. The responsibilities of the agent are as follows:

1. Establish and maintain connection with the AWS IoT Smart Home cloud.
2. Receive commands from the Smart Home cloud via MQTT, validate them and pass it to the CDMB.
3. Send back the command response received from the CDMB to the AWS IoT Smart Home cloud.
4. Send the unsolicited device events and device state notifications received from the CDMB to the cloud.
5. Maintain a local shadow of each device and update it for each device state change.
6. Periodically sync the local shadow and cloud shadow of each device connected to the IoT Hub.
7. Maintain a local copy of device schema for each device connected to the IoT Hub. Also, periodically update it with the device schema stored to the cloud.

All of the communication between Agent and cloud happens via MQTT using AWS Smart Home Internal Data Model format which follows the Matter Data Model.

2.2.2 Common Data Model Bridge (CDMB)

Common Data Model Bridge (CDMB) is the interface between the IoT Smart Home Device Agent and the Protocol specific CDMB plugins. The responsibilities of the CDMB are as follows:

1. Receive the command requests from the IoT Smart Home Device Agent in the AWS Smart Home Internal Data Model format.
2. Parse the received command request and extract the protocol of the end device for which the command is intended.
3. Create a CDMB task for each request and send the task to respective protocol plugin based on the protocol of the end device.
4. Receive back the command response from the protocol specific plugins and convert it to the AWS Smart Home Internal Data Model format.
5. Receive unsolicited events/device state notifications from the protocol specific plugins and send it to the IoT Smart Home Device Agent.

2.2.3 Protocol Specific CDMB Plugins

The Protocol specific CDMB plugin libraries are responsible for the following three functions.

1. **Task Handler**: It receives the tasks from CDMB and translates it to the protocol specific commands. Then, it calls the protocol specific Middleware APIs to send the command to the device. Also, it handles the response received from the device for each task and sends it back to the CDMB.
2. **Event Handler**: It receives the unsolicited events from the end devices through protocol specific Middleware, converts it to the AWS Smart Home Data Model report format and sends it to the CDMB.
3. **Device Capability Report Handler**: Protocol specific CDMB plugin libraries receive an event whenever a new end device is connected to the IoT Hub. Then, the plugin will fetch all of the device capabilities using the protocol specific middleware APIs. After receiving all of the device capabilities, the plugin will convert them to AWS Smart Home Data Model device capability report format and send it to the CDMB.

2.2.4 Protocol Specific Middlewares

Protocol specific middlewares have a critical role of interacting with underlying protocol stacks. They are used for both device onboarding and device control. So, Middleware architecture and use cases are separately discussed later in the middleware section.

2.2.5 End to end device control flow example

For this example, the end device in consideration is a ZigBee smart plug, and the end customer is trying to turn on the smart plug.

1. First the end customer will open the smart home app on their mobile device where they can see the ZigBee smart plug onboarded to their IoT hub.
2. Then the end customer can send a "Turn On" command to the ZigBee smart plug using the smart home mobile app.
3. The smart home mobile app will then create a command request and send it to the AWS IoT Smart Home cloud in AWS Smart Home Data Model command request format.
4. Once the AWS IoT Smart Home cloud has received the command, then it will forward that command to end customer's IoT Hub via MQTT.
5. IoT Smart Home Device Agent on the IoT Hub will then receive the command request from the cloud in AWS Smart Home Internal Data Model command request format.
6. IoT Smart Home Device Agent will then validate the command request and forward it to CDMB in AWS Smart Home Internal Data Model command request format.
7. Once the CDMB receive the command request, then it will parse it and extract the end device protocol which is ZigBee in this case. CDMB will then forward the command request to the ZigBee CDMB plugin.
8. After receiving the command request from CDMB, the ZigBee CDMB plugin will extract the command and parameters from it.
9. ZigBee CDMB plugin will then convert the command and parameters present in the AWS Smart Home Internal Data Model format to corresponding ZCL command. After that it will call the ZigBee MW APIs to send the ZCL command to the end device.

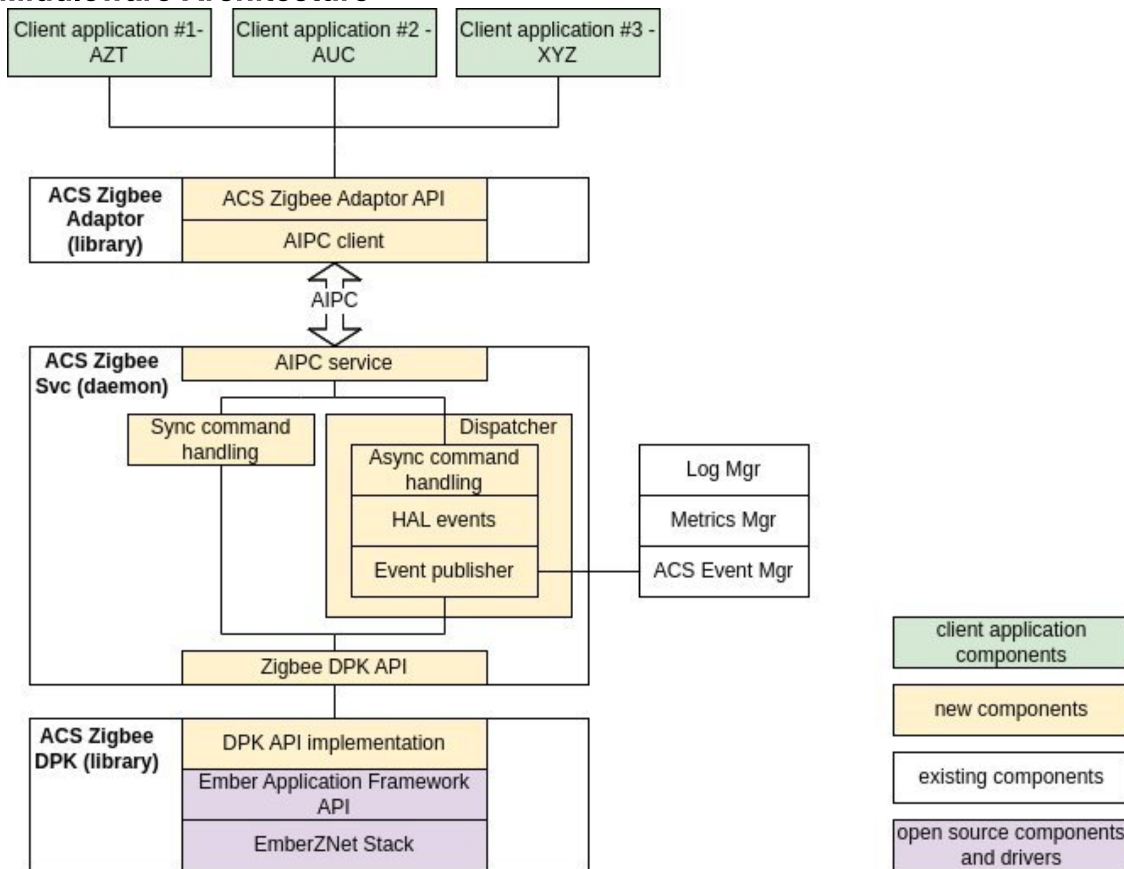
10. Finally, the ZigBee Middleware will send the ZCL command to the end device by calling ZigBee chipset vendor APIs.

3. Middleware

Protocol specific middlewares have a critical role of interacting with underlying protocol stacks. The main responsibilities of the middleware are as follows:

1. Abstracts the APIs from the device protocol stacks from different vendors by providing a common set of APIs.
2. Provides software execution management such as thread scheduler, event queue management and data cache.
3. Provides protocol-specific application stack such as ZigBee Cluster Library (ZCL) and BLE mesh.

3.1 Middleware Architecture



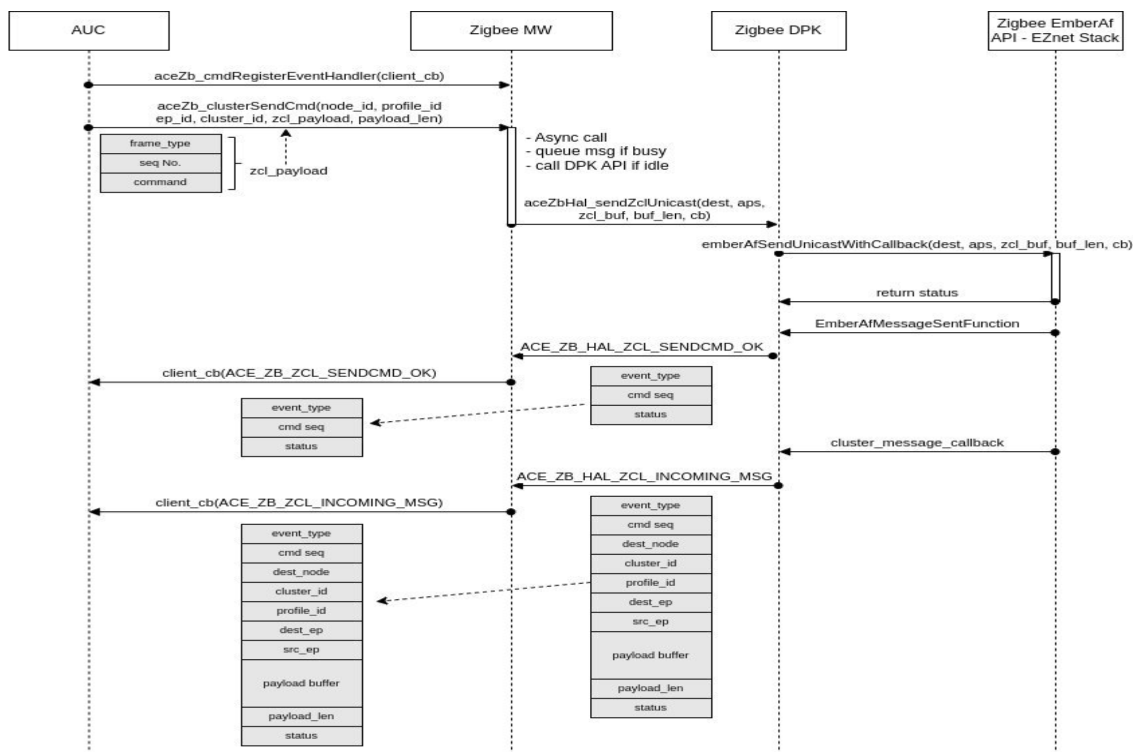
The above block diagram represents the architecture of the ZigBee Middleware. The architecture of middlewares of other protocols like Z-Wave is also similar.

There are 3 main components present in the Middleware.

1. **ACS ZigBee DPK:** ZigBee Device Porting Kit (DPK) is used to provide abstraction from underlying hardware and operating system, thereby enabling portability. Basically, this can be considered as the Hardware Abstraction Layer (HAL) which provides a common set APIs to control and communicate with the ZigBee radios from different vendors. The current ZigBee middleware contains DPK API implementation for the Silicon Labs ZigBee Application framework.
2. **ACS ZigBee Service:** ZigBee service runs as a dedicated daemon. It includes an API handler serving the API calls from client applications through the IPC channels. AIPC is used as the IPC channel between ZigBee adaptor and ZigBee service. It provides other functionalities like handling both async/sync commands, handling events from the HAL and using ACS Event Manager for event registering/publishing.
3. **ACS ZigBee Adaptor:** ZigBee adaptor is a library running within the application process (in this case, the application is the CDMB plugin). The ZigBee adaptor provides a set of APIs which are consumed by client applications like CDMB/Provisioner protocol plugins to control and communicate with the end device.

3.2 End to End Middleware command flow example

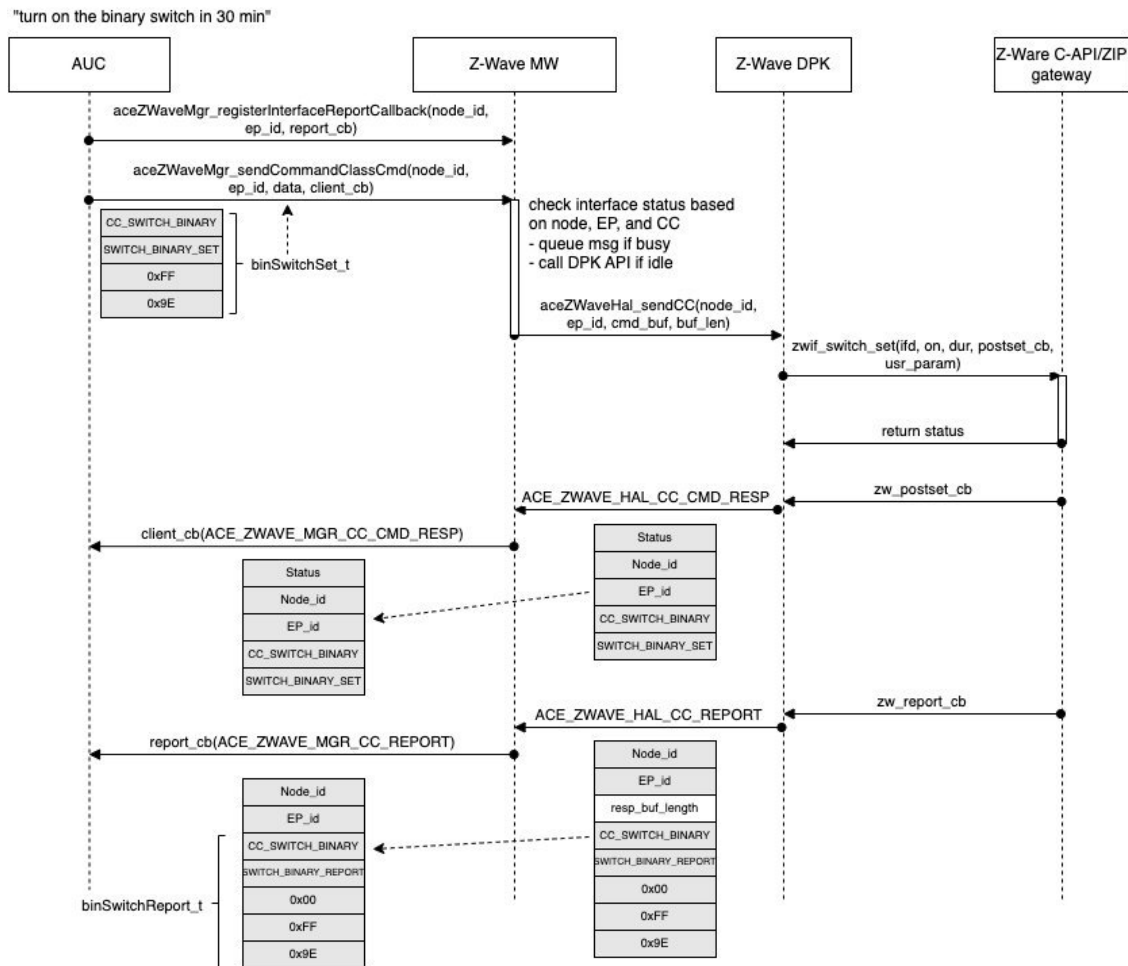
Here is an example of the command flow through the ZigBee Middleware.



Note: AUC component in this diagram refers to the ZigBee CDMB plugin.

Here is an example of the command flow through the Z-Wave Middleware.

Note: AUC component in this diagram refers to the Z-Wave CDMB plugin.



3.3 Middleware Code Organization

This section contains information about the location of the code for each component inside the IoTSmartHomeMiddlewares repository. Following is the high level folder structure IoTSmartHomeMiddlewares repo.

```
./IoTSmartHomeMiddlewares
├── greengrass
├── telus-iot-ace-dpk
├── telus-iot-ace-general
```

```
├── telus-iot-ace-project
├── telus-iot-ace-z3-gateway
├── telus-iot-ace-zware
└── telus-iot-ace-zwave-mw
```

3.3.1 ZigBee Middleware Code Organization

1. **ACS ZigBee DPK:** Code for ZigBee DPK is located inside `IoTSmartHomeMiddlewares/telus-iot-ace-dpk/telus/dpk/ace_hal/zigbee` folder. At this location, there are folders which contains DPK implementation for different protocols like ZigBee, Z-Wave and Wi-Fi.

```
./IoTSmartHomeMiddlewares/telus-iot-ace-dpk/telus/dpk/ace_hal/
├── common
│   ├── fxnDbusClient
│   └── include
├── kvs
├── log
├── wifi
│   ├── include
│   ├── src
│   └── wifid
│       ├── fxnWifiClient
│       └── include
├── zigbee
│   ├── include
│   ├── src
│   └── zigbeed
│       ├── ember
│       └── include
└── zwave
    ├── include
    ├── src
    └── zwaved
        ├── fxnZwaveClient
        ├── include
        └── zware
```

2. **Silicon Labs ZigBee SDK:** Code for the Silicon labs SDK is present inside `IoTSmartHomeMiddlewares/telus-iot-ace-z3-gateway` folder. The above ACS ZigBee DPK layer is implemented for this Silicon labs SDK.

```
IoTSmartHomeMiddlewares/telus-iot-ace-z3-gateway/
├── autogen
├── config
└── gecko_sdk_4.3.2
    ├── platform
```

```
    ├── protocol
    └── util
```

3. **ACS ZigBee Service:** Code for ZigBee Service is located inside `IoTSmartHomeMiddlewares/telus-iot-ace-general/middleware/zigbee/` folder. The `src and include` folder at this location contain all the files related to ACS ZigBee service.

```
./IoTSmartHomeMiddlewares/telus-iot-ace-
general/middleware/zigbee/src/
├── zb_alloc.c
├── zb_callbacks.c
├── zb_database.c
├── zb_discovery.c
├── zb_log.c
├── zb_main.c
├── zb_region_info.c
├── zb_server.c
├── zb_svc.c
├── zb_svc_pwr.c
├── zb_timer.c
├── zb_util.c
├── zb_zdo.c
└── zb_zts.c
./IoTSmartHomeMiddlewares/telus-iot-ace-
general/middleware/zigbee/include/
├── init.zigbeeservice.rc
├── zb_ace_log_uml.h
├── zb_alloc.h
├── zb_callbacks.h
├── zb_client_aipc.h
├── zb_client_event_handler.h
├── zb_database.h
├── zb_discovery.h
├── zb_log.h
├── zb_region_info.h
├── zb_server.h
├── zb_svc.h
├── zb_svc_pwr.h
├── zb_timer.h
├── zb_util.h
├── zb_zdo.h
└── zb_zts.h
```

4. **ACS ZigBee Adaptor:** Code for ZigBee Adaptor is located inside `IoTSmartHomeMiddlewares/telus-iot-ace-`

`general/middleware/zigbee/api` folder. The `src and include` folder at this location contain all the files related to ACS ZigBee Adaptor library.

```
./IoTSmartHomeMiddlewares/telus-iot-ace-
general/middleware/zigbee/api/src/
├── zb_client_aipc.c
├── zb_client_api.c
├── zb_client_event_handler.c
└── zb_client_zcl.c
./IoTSmartHomeMiddlewares/telus-iot-ace-
general/middleware/zigbee/api/include/
└── ace
    ├── zb_adapter.h
    ├── zb_command.h
    ├── zb_network.h
    ├── zb_types.h
    ├── zb_zcl.h
    ├── zb_zcl_cmd.h
    ├── zb_zcl_color_control.h
    ├── zb_zcl_hvac.h
    ├── zb_zcl_id.h
    ├── zb_zcl_identify.h
    ├── zb_zcl_level.h
    ├── zb_zcl_measure_and_sensing.h
    ├── zb_zcl_onoff.h
    └── zb_zcl_power.h
```

3.3.2 Z-Wave Middleware Code Organization

1. **ACS Z-Wave DPK:** Code for Z-Wave DPK is located inside `IoTSmartHomeMiddlewares/telus-iot-ace-dpk/telus/dpk/ace_hal/zwave` folder.

```
./IoTSmartHomeMiddlewares/telus-iot-ace-dpk/telus/dpk/ace_hal/
├── common
│   ├── fxnDbusClient
│   └── include
├── kvs
├── log
├── wifi
│   ├── include
│   ├── src
│   └── wifid
│       ├── fxnWifiClient
│       └── include
├── zigbee
│   ├── include
│   ├── src
```

```
│   └── zigbeed
│       ├── ember
│       └── include
└── zwave
    ├── include
    ├── src
    └── zwaved
        ├── fxnZwaveClient
        ├── include
        └── zware
```

2. **Silicon labs ZWare and Zip Gateway:** Code for the Silicon labs ZWare and Zip Gateway is present inside `IoTSmartHomeMiddlewares/telus-iot-ace-zware` folder. The above ACS Z-Wave DPK layer is implemented for Z-Wave C-APIs and Zip gateway.

```
IoTSmartHomeMiddlewares/telus-iot-ace-zware/
├── config
├── demos
│   ├── add_node
│   ├── basic
│   ├── bin_sensor
│   ├── bin_switch
│   ├── controller_app
│   ├── gw_discovery
│   ├── nw_reset
│   └── rm_node
├── doc
│   └── doxygen
├── external
├── include
│   └── zwave
├── lib
└── src
```

3. **ACS Z-Wave Service:** Code for Z-Wave Service is located inside `IoTSmartHomeMiddlewares/telus-iot-ace-zwave-mw` folder. The `src and include` folder at this location contain all the files related to ACS Z-Wave service.

```
IoTSmartHomeMiddlewares/telus-iot-ace-zwave-mw/src/
├── zwave_mgr.c
├── zwave_mgr_cc.c
├── zwave_mgr_ipc_aipc.c
├── zwave_svc.c
├── zwave_svc_dispatcher.c
├── zwave_svc_hsm.c
```

```
├── zwave_svc_ipc_aipc.c
├── zwave_svc_main.c
└── zwave_svc_publish.c
IoTSmartHomeMiddlewares/telus-iot-ace-zwave-mw/include/
├── ace
│   ├── zwave_common_cc.h
│   ├── zwave_common_cc_battery.h
│   ├── zwave_common_cc_doorlock.h
│   ├── zwave_common_cc_firmware.h
│   ├── zwave_common_cc_meter.h
│   ├── zwave_common_cc_notification.h
│   ├── zwave_common_cc_sensor.h
│   ├── zwave_common_cc_switch.h
│   ├── zwave_common_cc_thermostat.h
│   ├── zwave_common_cc_version.h
│   ├── zwave_common_types.h
│   ├── zwave_mgr.h
│   └── zwave_mgr_cc.h
├── zwave_log.h
├── zwave_mgr_internal.h
├── zwave_mgr_ipc.h
├── zwave_svc_hsm.h
├── zwave_svc_internal.h
└── zwave_utils.h
```

4. **ACS Z-Wave Adaptor:** Code for Z-Wave Adaptor is located inside `IoTSmartHomeMiddlewares/telus-iot-ace-zwave-mw/cli/` folder. The `src and include` folder at this location contain all the files related to ACS Z-Wave Adaptor library.

```
IoTSmartHomeMiddlewares/telus-iot-ace-zwave-mw/cli/
├── include
│   └── zwave_cli.h
└── src
    ├── zwave_cli.yaml
    ├── zwave_cli_cc.c
    ├── zwave_cli_event_monitor.c
    ├── zwave_cli_main.c
    └── zwave_cli_net.c
```

4. Smart Home Device SDK Integration to a new Hub

Smart Home Device SDK is designed in such a way that it should be easier for a Smart Home service provider or ODM to integrate and run it on their Hubs. All of the Device SDK components except the Middleware are hardware independent. So, all of the device SDK components like IoT Smart Home Device Agent, Common Data Model Bridge (CDMB), Core

Provisioner and Protocol specific CDMB/provisioner plugins can be used without any modifications. These components just need to be cross compiled and can then run on the new hub.

Middleware integration on the new hub is discussed in the next two sub sections.

4.1 DPK API Implementation

To integrate any Chipset vendor SDK with the middleware a standard API interface is provided by the DPK(Device porting kit) layer of the Middleware. The AWS smart home service providers/ODMs need to implement these APIs based on the vendor SDK supported by the ZigBee/Z-wave/Wi-Fi chipsets used on their IoT Hubs.

4.2 Reference implementation and code organization

Reference implementation of the middleware is based on the Silicon labs SDK for ZigBee and Z-Wave. If the Z-Wave and ZigBee chipsets used in the new hub are supported by the Silicon labs SDK present in the middleware, then the reference middleware can be used without any modifications. Middleware also just needs to be cross compiled and it can run on the new hub.

DPK APIs for ZigBee can be found in `acehal_zigbee.c` and reference implementation of the DPK APIs is present inside the `zigbeed` folder.

```
IoTSmartHomeMiddlewares/telus-iot-ace-
dpk/telus/dpk/ace_hal/zigbee/
├── CMakeLists.txt
├── include
│   └── zigbee_log.h
├── src
│   └── acehal_zigbee.c
└── zigbeed
    ├── CMakeLists.txt
    ├── ember
    │   ├── ace_ember_common.c
    │   ├── ace_ember_ctrl.c
    │   ├── ace_ember_hal_callbacks.c
    │   ├── ace_ember_network_creator.c
    │   ├── ace_ember_power_settings.c
    │   └── ace_ember_zts.c
    └── include
        ├── zbd_api.h
        ├── zbd_callbacks.h
        ├── zbd_common.h
        ├── zbd_network_creator.h
```

```
        ├── zbd_power_settings.h
        └── zbd_zts.h
```

DPK APIs for Z-Wave can be found in `acehal_zwave.c` and reference implementation of the DPK APIs is present inside the `zwaved` folder.

```
IoTSmartHomeMiddlewares/telus-iot-ace-
dpk/telus/dpk/ace_hal/zwave/
├── CMakeLists.txt
├── include
│   └── zwave_log.h
├── src
│   └── acehal_zwave.c
└── zwaved
    ├── CMakeLists.txt
    ├── fxnZwaveClient
    │   ├── zwave_client.c
    │   └── zwave_client.h
    ├── include
    │   ├── zwaved_cc_intf_api.h
    │   ├── zwaved_common_utils.h
    │   └── zwaved_ctrl_api.h
    └── zware
        ├── ace_zware_cc_intf.c
        ├── ace_zware_common_utils.c
        ├── ace_zware_ctrl.c
        ├── ace_zware_debug.c
        ├── ace_zware_debug.h
        └── ace_zware_internal.h
```

As a starting point to implement the DPK layer for a different vendor SDK, the reference implementation can be used and modified. Following two modifications will be need to support a different vendor SDK:

1. Replace the current vendor SDK with the new vendor SDK in the repository.
2. Implement the middleware DPK (Device porting kit) APIs according to the new vendor SDK.