HomeKit ADK Integration Guide

For ADK 2.0

November 2018

Contents

This guide is intended for use with a beta version of the ADK and is not intended for use in the development of Proposed Products or Licensed Products under an MFi License.

1. Introduction

The HomeKit Accessory Development Kit (ADK) abstracts the details of the HomeKit Accessory Protocol (HAP) so that a chipset vendor can ultimately focus on their hardware platform and an accessory manufacturer on the application layer.

This document describes how a chipset vendor can use the HomeKit Accessory Development Kit (ADK) to develop a HomeKit SDK, and how an accessory manufacturer can use the HomeKit ADK to bring HomeKit functionality to their home automation accessory.

1.1. HomeKit SDK Development

Chipset vendors can base their HomeKit SDK on the ADK provided by Apple. Using the ADK can help reduce initial development effort and maintenance costs for HomeKit protocol updates.

1.2. HomeKit Accessory Development

Accessory manufacturers can work directly with the ADK. However, it is recommended an accessory manufacturer works with a HomeKit SDK that supports a platform type that is the same or similar to the accessory manufacturer's target platform. This will minimize the amount of platform development required to support HomeKit technology.

1.3. How This Guide is Organized

This guide consists of the following sections:

- Architecture Overview: Discusses the elements of the HomeKit ADK, including Apple's implementation of the HomeKit protocols. This is relevant for all users of the ADK.
- Accessory Logic Development: Describes how to set up a HomeKit profile for the accessory, with the necessary services and characteristics, and how to write the code that implements the accessory behavior. This is relevant mostly for accessory manufacturers.
- Platform Development: Describes how to write a Platform Abstraction Layer (PAL) for a platform. A PAL decouples Apple's protocol implementation from the specific hardware and software of a platform. It provides an overview of the platform requirements, descriptions of the modules that make up a PAL, and how a PAL can be packaged into a complete HomeKit SDK. This is relevant mostly for chipset vendors.

- Appendix A: Set Up the POSIX SDK: Contains a guide for setting up a Raspberry Pi with the POSIX sample SDK. It is an example implementation for chipset vendors and accessory manufacturers targeting Internet Protocol (IP)-based accessories. This SDK is of interest to both vendors of IP chipsets and manufacturers of IP accessories.

1.4. Conventions Used in this Guide

The following typographical conventions are used in this guide:

Underline	Indicates URLs, email addresses, or a reference to another section of this document or other document.
`Constant width`	Used for program listings or program elements such as function names, for file names and paths, for commands or other input to be typed by a user, or for output generated by a program.

1.5. Integration of the HomeKit ADK

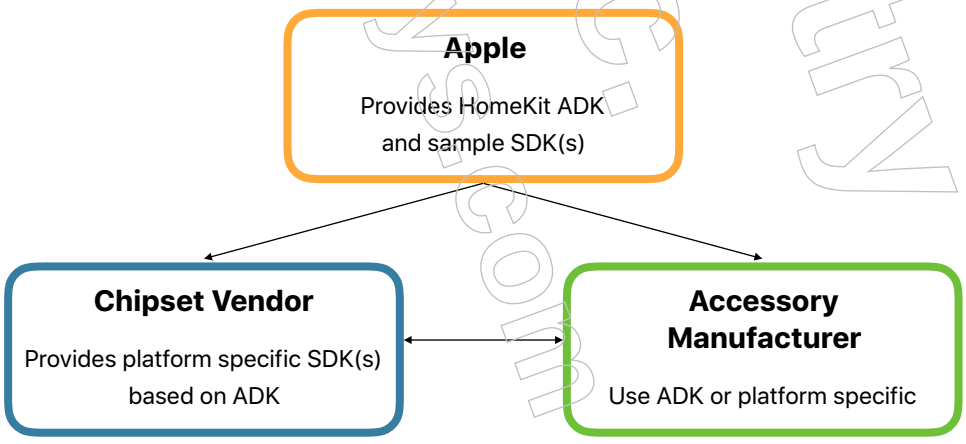
The HomeKit ADK supports the following HAP variants:

- HAP over Wi-Fi or Ethernet (Referred to as IP)
- HAP over Bluetooth Low Energy (BLE)

Either HAP over IP or HAP over BLE is supported, but not both at the same time.

Developing the HomeKit software for an accessory consists of two tasks:

- Accessory logic development: Use the most appropriate sample application in the ADK as a starting point to develop the accessory logic (the application-specific part of the accessory's software).
- Platform development: The HAP implementation provided by Apple in the ADK is independent of the platform it will run on. To make this possible, the platform-dependent parts of a HomeKit implementation are encapsulated in the PAL. The ADK provides an empty PAL as a template, and also sample PALs for IP and BLE accessories. These PALs can be used as illustrations and starting points for new PALs. They are not intended to be used *as is*; platform-specific adaptations or optimizations will usually be necessary.



Systems integrators may be able to assist in creating a new PAL or in developing accessory logic.

1.6. HAP over Wi-Fi or Ethernet

A sample SDK for POSIX and similar platforms, in particular Linux is provided with the ADK. It supports Raspberry Pi 3 Model B as a reference platform that can be used for evaluation, training and during development of a new IP-based target platform. Even while targeting a non-POSIX accessory with Wi-Fi or Ethernet, the POSIX SDK is a good starting point. Complete build scripts are provided, for partners to modify, build and deploy the sample applications on a Raspberry Pi. More details are given in Appendix A: Set Up the POSIX SDK.

The Raspberry Pi with the Raspbian Linux distribution is to be used as an easily available reference platform only, not a target platform for actual accessory products.

A HomeKit PAL for Linux has dependencies on each Linux platform unique own configurations. This is most critical regarding networking, e.g. how `dhcpcd` is used as part of the Bonjour support required by HomeKit. It is the accessory manufacturer's responsibility to appropriately configure its Linux platform for HomeKit, e.g., for Bonjour conformance, security, etc.

Support for HAP over iCloud is planned for a future release of the ADK.

1.7. HAP over BLE

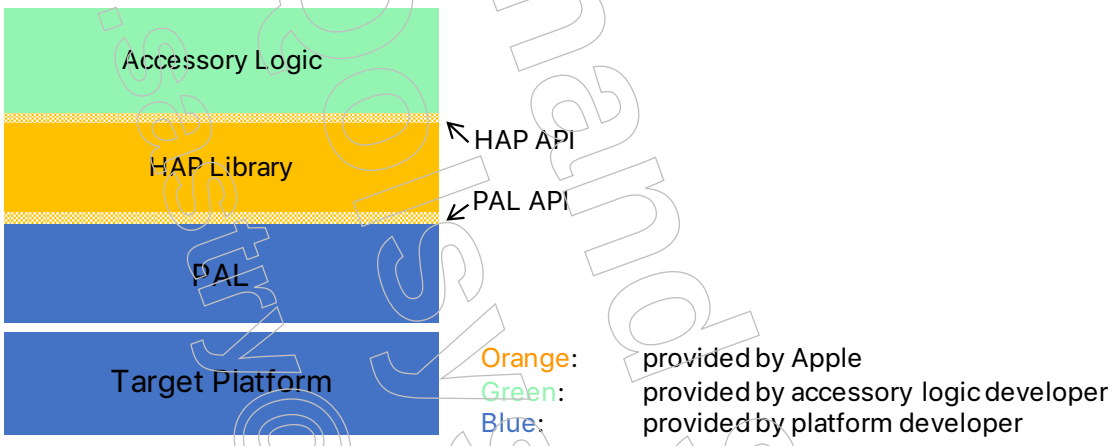
For Bluetooth Low Energy (BLE) accessories, sample source code is provided for the complete PAL for a BLE SoC. It is intended to demonstrate the level of effort required to support BLE using the ADK and unlike the POSIX SDK, it is not a complete and ready-to-use SDK. It is mainly of interest to vendors of BLE chipsets.

2. Architecture Overview

This section provides an overview of the contents of the HomeKit ADK and how the elements of the ADK execute on an accessory at runtime.

2.1. HAP at Runtime

At runtime, the HomeKit-related software on an accessory consists of three major elements: the accessory logic, the HAP Library, and the PAL. The HAP Library implements the HomeKit Accessory Protocol. It is a software layer that resides between the accessory's application layer software (e.g., control software for a ceiling fan) and the target platform (e.g., a custom-designed Linux device). These elements, and the interfaces between them (APIs), are discussed below, starting from the top - the accessory logic.



2.1.1. Accessory Logic

The accessory logic contains the application-layer code of an accessory. It is written by an accessory manufacturer, using its particular domain expertise, such as heating and cooling of rooms for thermostat accessories, or audio / video processing for IP camera accessories. The ADK contains samples for light bulb, thermostat, door lock, camera and HomeKit bridge accessories. To implement the accessory logic, its developer mainly uses the HAP API.

2.1.2. HAP API

The HAP API, defined by Apple, allows developers to declare the services and characteristics of the accessory. After initialization of the target platform, the HomeKit accessory server which handles all HAP communication over IP or BLE can be started. When a request comes in (e.g., a request from an iPhone to change the set point of a thermostat) the HAP Library calls the suitable function pointers provided during the declaration of the services. As these calls go from the HAP Library "upwards" into the accessory logic, they are called upcalls. Upcalls issued by the HAP Library never overlap in order to avoid hard-to-find multithreading issues, to make it easy to integrate the HAP Library with other code, and to make the library usable even on "bare metal" systems without operating system and multithreading support. The HAP API is simple, and flexible enough to support custom characteristics and services in addition to the Apple-defined HomeKit profiles.

2.1.3. HAP Library

Apple provides an up-to-date implementation of the HAP Library. It implements the HAP pairing, session, and security protocols. It is intended to be used by all accessories and HomeKit SDKs based on the ADK. When the HAP protocol is updated, rebuilding the accessory software with an updated version of the HAP Library is often sufficient. To ensure integrity, the HAP Library is distributed as a binary library file. Variants for many popular toolchains and instruction sets are provided.

The single-threaded design of the HAP Library makes it easy to integrate into different environments (e.g., into device software that already exists and uses a proprietary cloud service for data storage).

2.1.4. PAL API

The PAL API is defined by Apple but unlike the HAP API, it is not implemented by Apple. It is used by the HAP Library, and must be implemented in a PAL by a platform developer. The PAL API is highly modular and supports mechanisms including timers, persistent storage, random number generator, and more. It is designed to provide abstractions for every service that the HAP Library needs, in a way that enables easy implementation with minimal overhead.

Note: As a general rule, an accessory logic developer does not need to know the PAL API except for its initialization functions, and a platform developer does not need to know the HAP API.

2.1.5. PAL

Depending on the SDK and its supplier, a PAL is provided as a binary library or as source code that the developer can modify. An SDK may provide several alternate implementations of a PAL module (e.g., to support different Wi-Fi chips that implement the same Wi-Fi-related portion of the PAL API). If one PAL module needs modifications, the other PAL modules are often not affected.

If an accessory manufacturer intends to develop a new generation of accessories, they should use a HomeKit SDK that is optimized for the chosen hardware/software platform. This may result in minimal or no need for PAL modifications. If an accessory manufacturer is developing a derivative of an existing accessory that did not previously support HomeKit, its hardware/software platform may not be supported by an existing SDK. This may require additional effort for platform development.

If a chipset vendor wants to make it easier for their products to be used in HomeKit accessories, they should create a PAL optimized for their chipsets. As starting point, an empty sample PAL in the ADK can be used as a template, or the PAL in the POSIX SDK. If a chipset vendor already has its own HomeKit SDK, then it should be possible to repackage the code into a PAL by removing the existing protocol code and replacing it with the HAP Library.

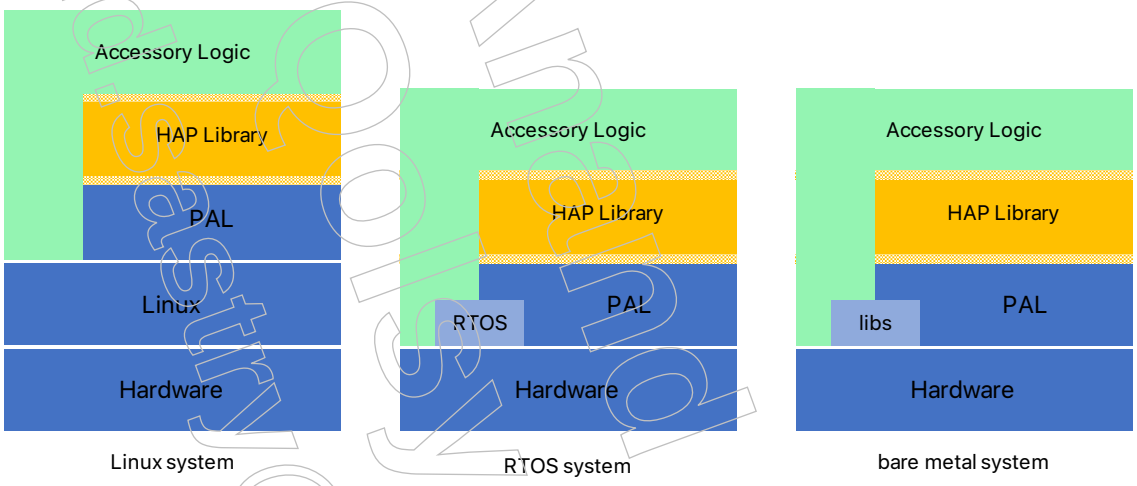
2.1.6. Target Platform

A PAL is created for a specific target platform. An operating system (OS) like Linux provides its own abstractions. Therefore, a single PAL for such an OS covers a broad spectrum of target platforms. The device drivers of the OS provide a platform abstraction layer of their own, typically called a hardware abstraction layer. This can simplify development for a platform developer but may pose challenges regarding the correct and secure configuration of the OS.

A PAL can be implemented without an OS, using a single thread of execution, which is called a bare metal implementation. No complete hardware abstraction exists in a bare metal system; therefore, a bare metal PAL is specific for a particular hardware, its system libraries, and the toolchain provided by a chipset vendor. When using integrated SoCs the chipset largely determines the target platform and can be fully supported by a PAL. Therefore, modification to this PAL may not be required.

For some resource-constrained accessories, in particular those with IP network stacks, developers often take advantage of a real-time operating system (RTOS). For example, FreeRTOS, ThreadX, RTX, Contiki, NuttX or Mbed. While an RTOS can be used for its real-time capabilities in other types of projects, these special capabilities are less applicable to the development of a HomeKit accessory. For HomeKit, such real-time operating systems are more relevant due to their small memory footprints, which enable low BOM costs. Due to the minimal hardware abstractions provided by a typical RTOS, creating a PAL for an RTOS platform can require more work compared to an operating system like Linux. On the other hand, a full operating system may pose its own challenges by providing too much abstraction (e.g., preventing sufficiently flexible access to the network stack).

This diagram shows an architecture comparison between these different types of systems:

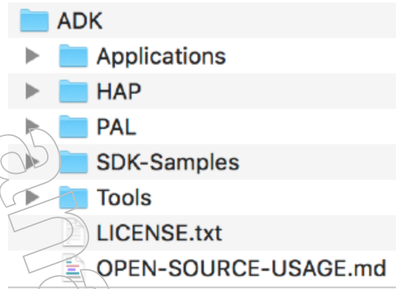


Orange: provided by Apple
Green: provided by accessory logic developer
Blue: provided by platform developer

The RTOS typically has little impact on the effort needed to create a PAL. The more significant effort is usually the adaptation to a BLE or IP protocol stack. IP stacks have APIs that are often similar to Berkeley sockets. The challenge with IP stacks can be with dual IPv4 / IPv6 support which is required by HomeKit, Bonjour, and Wi-Fi Accessory Configuration 2 (WAC2). Linux can resolve most of these challenges. BLE stack APIs exhibit more variety compared to IP stack APIs; therefore, the effort required can differ considerably between BLE stacks.

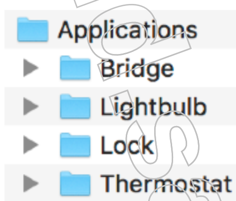
2.2. ADK Directory Structure

The previous section explained how the software of a HomeKit accessory works at runtime. Key elements of this software, in particular the HAP Library, are provided by Apple in the ADK, or are redistributed by chipset vendors in their HomeKit SDKs. The elements delivered in the ADK archive are `Applications`, `HAP`, `PAL`, and `SDK-Samples` subdirectories:



2.2.1. Applications

The `Applications` subdirectory contains source code of typical accessories.



For example, the `Lightbulb` subfolder contains the accessory attribute database definition for a light bulb accessory and a simple implementation of its callbacks using log output (HAPLog). These files contain no platform-specific code, have minimal complexity and are fully portable.

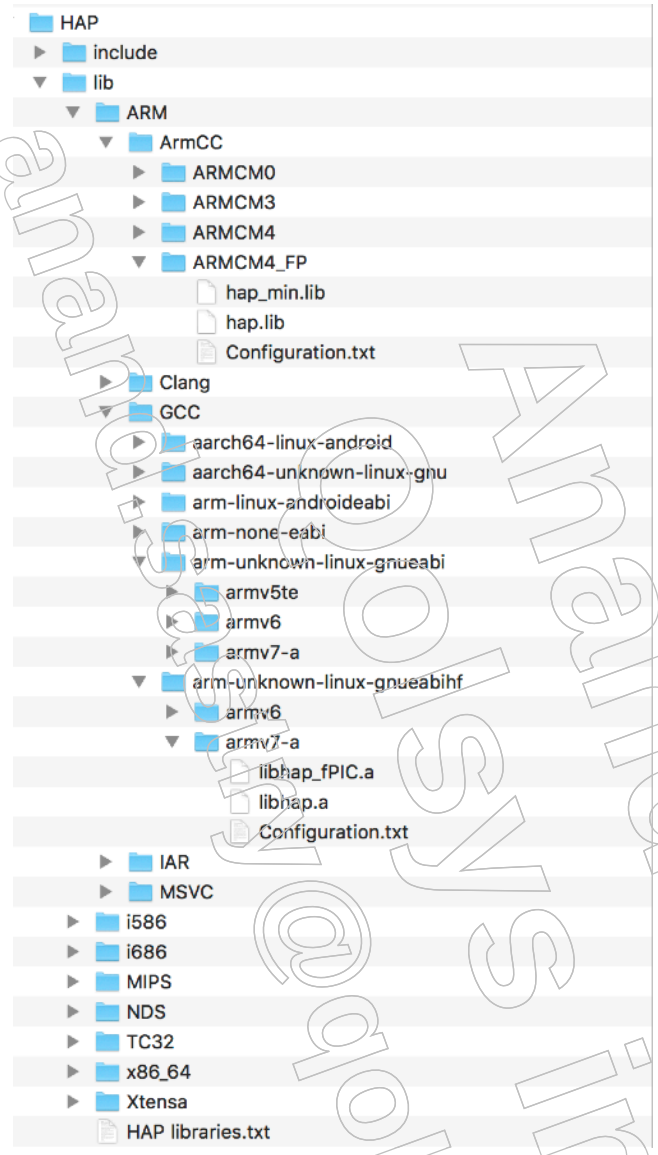
For complete source code examples with an initialization of the platform and the starting of the HAP server, review the corresponding POSIX SDK sample `ADK/SDK–Samples/POSIX/Raspi/Applications/Lightbulb`; it contains a complete example including the platform-specific Main.c file and makefile.

The SDK samples also contain additional samples that use platform-specific I/O mechanisms and are more fully functional. For example, see the `LightbulbLED` sample `ADK/SDK–Samples/POSIX/Raspi/Applications/LightbulbLED`.

Note: The ADK header files can be used with C++ code. However, the accessory logic samples, and also all PAL code in the SDK samples, are C99 code, which is not fully C++ compatible. For this reason, it must not be compiled in C++ mode. When using C++, it is recommended to use GCC, not g++.

2.2.2. HAP

The HAP API is provided in the `HAP/include` subfolder. The actual implementation of HAP is in the ADK/HAP/lib subfolder and only contains binary library files. The `libhap.a` (for GCC) and `hap.lib` (for ARM Keil MDK, aka ArmCC) files are organized by processor architectures and toolchains:



The list of currently supported formats is described in the document HAP `libraries.txt`. Every HAP Library binary is placed in its own directory, which also contains a `Configuration.txt` file describing the library in more details, and in some cases a `.config` file that contains the input to Crosstool-NG for creating a GCC toolchain for the library. The number of library files offered may increase to support a more diverse set of platforms in the future.

The most common instruction set architectures, and processor cores implementing them, are already supported:

Compilers	Instruction Set Architecture	Processor / Core Examples	Remarks
ARM Keil MDK 5, GCC, IAR	ARMv6-M	Cortex-M0, Cortex-M0+	Compiler-specific ARM embedded ABIs. `_min` variant provided.
ARM Keil MDK 5, GCC, IAR	ARMv7-M	Cortex-M3	Compiler-specific ARM embedded ABIs. `_min` variant provided.

Compilers	Instruction Set Architecture	Processor / Core Examples	Remarks
ARM Keil MDK 5, GCC, IAR	ARMv7E-M	Cortex-M4	Compiler-specific ARM embedded ABIs. No use of FPU. Can also be used for Cortex-M4F systems that use the soft float ABI. `_min` variant provided.
ARM Keil MDK 5, GCC, IAR	ARMv7E-M with FPv4-SP	Cortex-M4F	Compiler-specific ARM embedded ABIs, single-precision hard float. `_min` variant provided.
GCC	ARMv7-R	Cortex-R4	Compiler-specific ARM embedded ABI. No use of FPU. Can also be used for Cortex-RF systems that use the soft float ABI. `_min` variant provided.
GCC	ARMv7-R with FPv3-SP	Cortex-R4F	Compiler-specific ARM embedded ABI, single-precision hard float. `_min` variant provided.
GCC	ARMv5TE	ARM926	Linux soft float ABI. Compiled with `–march=armv5te –marm –mfloat-abi=soft`.
GCC	ARMv6	ARM1176	Linux soft float ABI. Compiled with `–march=armv6 –marm mfloat-abi=soft –mfpu=vfpv3-d16`.
GCC	ARMv6	ARM1176	Linux hard float ABI. Compiled with `–march=armv6 –marm mfloat-abi=hard –mfpu=vfpv3-d16`.
GCC	ARMv7-A	Cortex-A7	Linux soft float ABI. Compiled with `–march=armv7-a –mthumb mfloat-abi=soft –mfpu=neon-vfpv4`.
GCC	ARMv7-A	Cortex-A7	Linux hard float ABI. Compiled with `–march=armv7-a –mthumb mfloat-abi=hard –mfpu=neon-vfpv4`.
GCC, Clang	ARMv7-A	Cortex-A7	Android soft float ABI.
Visual Studio 2008 with Service Pack 1	ARMv7-A	Cortex-A7	For Windows Embedded Compact 7 (formerly Windows CE).
GCC, Clang	ARMv8-A	Cortex-A73	Android hard float ABI.
GCC	x86-64	Xeon	Supported by nearly all Intel and AMD processors after the i386. Linux ABI.
Clang	x86-64	Xeon	Supported by nearly all Intel and AMD processors after the i386. macOS ABI.
Visual Studio 2017	x86-64	Xeon	Supports Windows 7 and newer.
GCC	i586, i686	CE5310	Intel 32-bit processors.
GCC	MIPS32 R2	MIPS 24K	Linux o32 soft float ABI, little endian.
GCC	MIPS32 R2	MIPS 24K	Linux o32 soft float ABI, big endian.
GCC	LX6	ESP32	Xtensa.

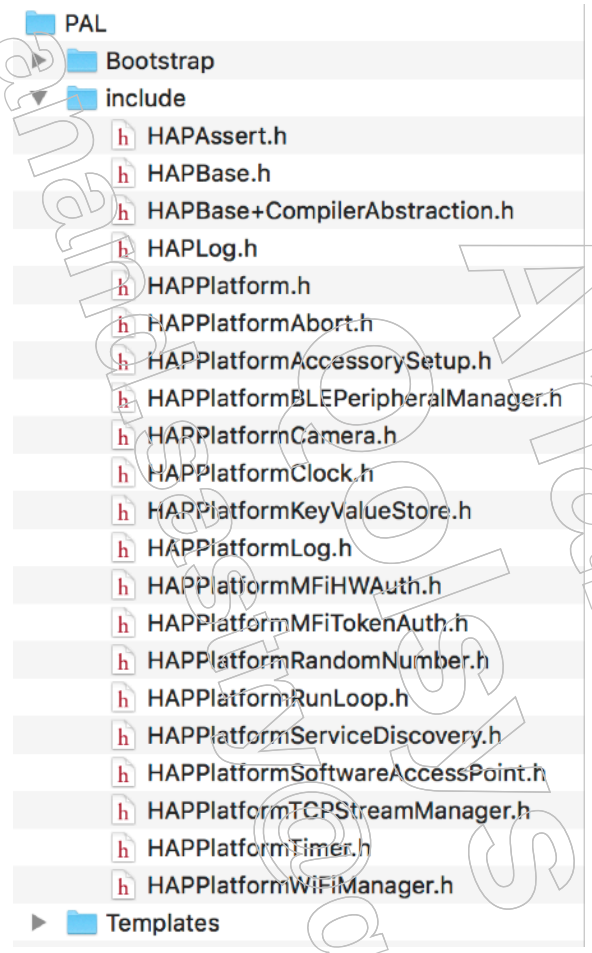
Compilers	Instruction Set Architecture	Processor / Core Examples	Remarks
GCC	LX106	ESP8266	Xtensa.
XCC	QCA4010	QCA4010	Xtensa.
GCC	TC32	TLSR8269	Telink.
GCC	NDS32 N10		Andes.

For ARM Cortex-A microprocessors, HAP Library variants built with and without the `fPic` compiler option are provided.

For microcontroller-based systems, use the standard binaries during development if sufficient memory is available, as they provide log output that is helpful for debugging purposes. For a licensed accessory, the provided `_min` binaries are more suitable, as they have a considerably smaller memory footprint.

2.2.3. PAL

The PAL API is in the `PAL/include` subfolder.



The `HAPPlatform.h` file is an umbrella header file for the header files of the individual PAL module APIs, namely `HAPPlatform<ModuleName>.h`. A PAL developer must provide implementations for all `HAPPlatform<ModuleName>.h` files. The other header files are helpers that need not be modified:

- `HAPAssert.h` that the HAP Library uses for raising assertions, e.g., through precondition checks.
- `HAPBase.h` defines common base types needed in different PAL modules and in HAP.h.
- `HAPBase+CompilerAbstraction.h` makes it possible to support a variety of compilers.
- `HAPLog.h` defines the logging levels supported by the ADK for the HAP Library.

The layering of these files is shown below, i.e., a header file may contain dependencies to one or several lower layer header files, but not vice versa. For example, `HAPPlatform.h` includes `HAPPlatformTimer.h`, `HAPPlatformRunLoop.h`, `HAPPlatformAbort.h`, etc., but `HAPPlatformAbort.h` must not include e.g. `HAPAssert.h`:

- **`HAPPlatform.h`**
- **`HAPPlatform<ModuleName>.h`**

- HAPBase.h
- HAPAssert.h
- HAPLog.h
- **HAPPlatformAbort.h**
- HAPBase+CompilerAbstraction.h

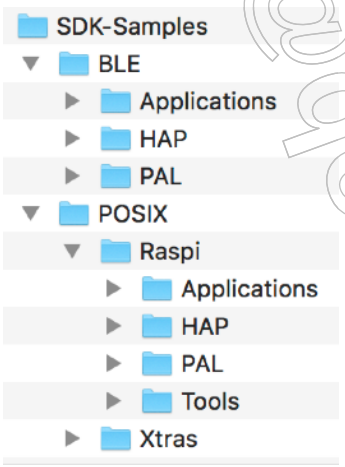
The files to be implemented are listed above in bold. More information about each of them is provided in the PAL Modules chapter. A platform developer should include HAPPlatform.h in order to include all header file dependencies.

An empty "null" PAL is located in the ADK/PAL/Templates folder. This PAL is a set of template files where the function implementations of the various PAL modules produce "not yet implemented" log output and then abort the program. They can be used as a starting point for implementing a new PAL.

To decrease the time until a first complete prototype of a new PAL is working, simple portable implementations of several PAL modules are provided in ADK/PAL/Bootstrap. For example, one is an implementation of a random number generator that only produces pseudo-random numbers instead of cryptographically secure random numbers. Bootstrap modules **must not** be used in final products.

2.2.4. SDK Samples

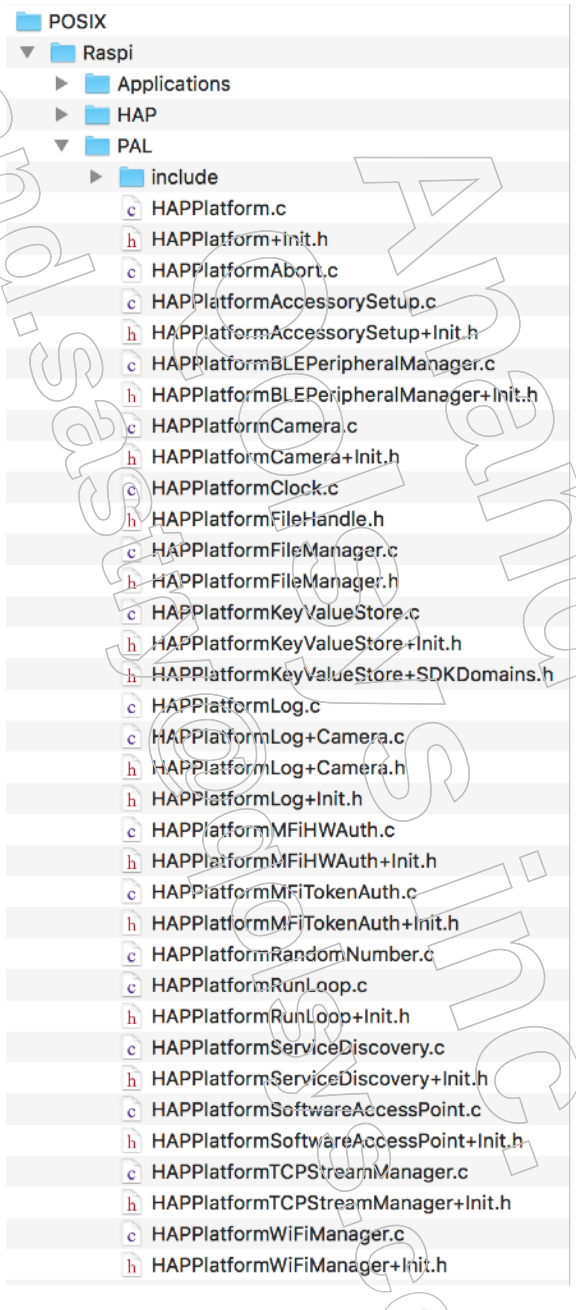
The SDK–Samples subdirectory contains an example of a HomeKit SDK for IP-based accessories (ADK/SDK–Samples/POSIX) and another one for BLE-based accessories (ADK/SDK-Samples/BLE). These SDKs are self-contained so that they can be copied or deleted as needed.



Both the BLE and the POSIX sample folders contain Applications, HAP and PAL subfolders.

- The Applications folder contains a superset of the samples given in ADK/Applications. It includes more comprehensive native samples that illustrate platform-specific features, e.g., a LightbulbLED sample that shows how to access an on-board LED of a BLE development kit, or of the Raspberry Pi which is used as a reference platform for the POSIX sample SDK.
- The HAP folder contains a subset of the HAP Library binaries given in ADK/HAP. It is the subset that is applicable to the platform(s) supported by the SDK.

- The PAL folder contains an implementation of the PAL defined in ADK/PAL. In the POSIX SDK, a PAL for the Raspberry Pi reference platform is given in the Raspi subfolder. Alternate implementations that may be useful for other Linux configurations, or non-Linux POSIX platforms, are given in the `Xtras` subfolder.

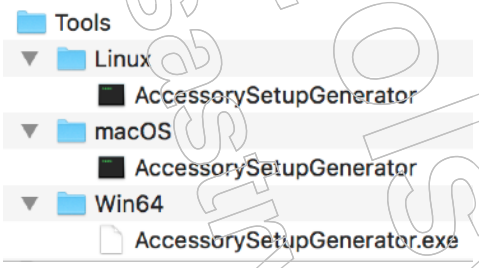


  - The header files in `ADK/SDK–Samples/POSIX/Raspi/PAL/include` are direct copies of the `ADK/PAL/include` header files.
  - The `ADK/SDK–Samples/POSIX/Raspi/PAL/HAPPlatform<ModuleName>.c` files contain the actual PAL for the Raspberry Pi. They can be used as starting points, in particular for Linux platforms. Depending on the configurations of the Linux platform being used, modifications may be necessary. It is the responsibility of the platform developer to ensure correctness and security of the PAL for their target platform.

- The header files in the ADK/SDK–Samples/POSIX/Raspi/PAL directory that have names ending in +Init.h provide Raspberry Pi specific initialization functions for those PAL modules that need explicit initialization. Platform developers may modify these initialization functions as needed for their platforms, unlike the other PAL header files, which must not be modified. This is possible because initialization of the PAL is not performed by the HAP Library, but by the initialization code of the accessory logic (e.g., in its Main.c file).
- The remaining files, e.g., HAPPlatformFileHandle.h, define Raspberry Pi specific (or rather, Linux-specific) internal abstractions that are useful for several PAL module implementations. HAPPlatformFileManager.h/.c is a helper for reading setup information from a POSIX file.

- Additional folders may be provided in a HomeKit SDK, e.g., a Tools folder with useful tools or a Dependencies folder that contains system libraries needed specifically for the HomeKit PAL, in case the underlying operating system configuration does not by default.

2.2.5. Tools



The ADK/Tools directory contains the AccessorySetupGenerator tool, in Linux and macOS and Windows 64-bit variants. This tool can be used by provisioning mechanisms (e.g., the script in ADK/SDK–Samples/POSIX/SDK/Raspi/Tools/Provision) to generate information for the provisioning of a HomeKit accessory, such as a setup code, a corresponding SRP salt and verifier, and a setup ID. The setup code is used by the controller for setting up an encrypted link with the accessory during HomeKit pairing. The setup ID is used to identify the accessory to which a scanned label belongs. *Start the tool to see the options that it supports.* For more information, see Set up provisioning.

The output of the generator is sent to stdout, whereas stderr is used for error conditions.

From the Mac, you can create the provisioning data, e.g., for Lightbulb:

```
ADK/Tools/macOS/AccessorySetupGenerator --ip --category 5
```

This produces the following output in stdout:

```
1

111-22-333

3E5961962E116C229ADB36CF7D3A33FA

A27D3DE652F1942DF5A03862E4F7B1A955F8684F2B7469469932A960605644EF0EA4E622948A4958CC7
A1493B359391DE4DBD98E1345A5F06BD33392651235AA9E086472FA0F966ABC992BADFCDCEAD9B6628A
4B215B20E3643D61D7AD6BCA334727B4154A0D218B0FE7D2D62D5C0E165C07C9BBCE3D3FE2F4DF77FCD
57E4807A0FB4237E0278069D5957331815CC5DC7C28EE3069F5A529EDE9F9C84964FC9DBBDDF1D10DB7
73F42F64327D0BBD3ECD2A9F360937156F6DEB38BA2919DB940853C407D893BCF7249E294CA289E0F9F
1BE900AB731B3A26272750FE334145232EB0CD70AA4B48149B65B243D9C30B8875939ACAA0CD6A6F95C
```

```
6FCACF81A2BD720E5EE041B804AD713AB5A7B95ABDE01CCC3081F5012A6F4EC0FFF8A3C59BE8CFBCB1A
7C914CCEE26B8EC976215D5683EE4B0DDC225DF1F20E6AB4355F293D1AC9F6A1A91AC8B728655F70583
0B7F46C0FFE0A37E8F24126E3B7EA47E0F38AB01BA8BE6FD14CAF74FEB942658B3453231F87EEF19789
0C484AFE72FD9C3751A94

ACME

X-HM://00527813XACME
```

The POSIX `Provision` script writes the generated values into files in folder `.KeyValueStore`:

File `.HomeKitStore/40.10` contains SRP salt and verifier

File `.HomeKitStore/40.11` contains setup ID

The following random number generators are used by the tool:

- macOS: `/dev/urandom`
- Linux: `/dev/urandom`
- Windows: `BcryptGenRandom` with `BCRYPT_RNG_ALGORITHM`. This function complies with the NIST SP800-90 standard, specifically the `CTR_DRBG` portion of that standard, and is not compatible with older versions of Windows. Prior to ADK build 15F41 the algorithm specified by `BCRYPT_USE_SYSTEM_PREFERRED_RNG` was used.

Note: For the Windows version of the tool, install the following package: https://aka.ms/vs/15/release/vc_redist.x64.exe *via:* https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads. *Windows versions older than Windows 7 are not supported, due to the use of the* `BCryptGenRandom` *function.*

2.2.6. Migration between ADK versions

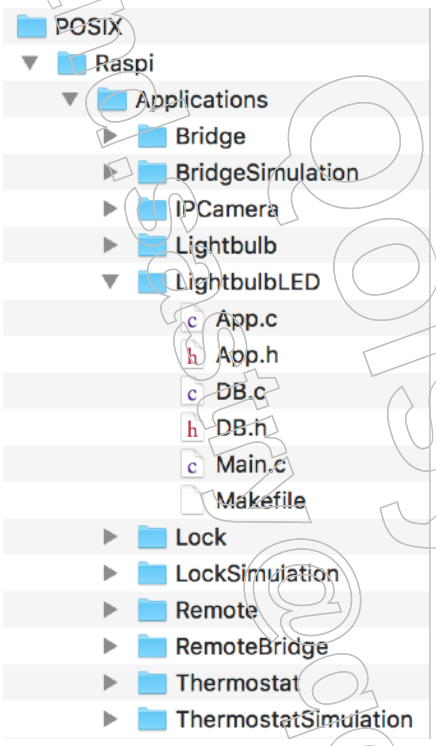
To migrate from one ADK version to a newer one, it is often sufficient if you replace the contents of the following directories with the one in the newer ADK version: `HAP/include`, `HAP/lib` and `PAL/include`. Often this will be sufficient to get your code to compile again, and possibly to work again right away.

3. Accessory Logic Development

An accessory logic developer can typically start from one of the sample applications provided in the SDK and modify it based on development needs.

3.1. Application Samples

The sample code provided in the POSIX SDK is structured as follows:



There are two types of accessory logic samples included:

- Basic samples that are good starting points for understanding how accessory logic may look like, as they are portable and minimal. These are the `Bridge`, `Lightbulb`, `Lock`, and `Thermostat` samples. They are identical to the ones in `ADK/Applications` and produce log output, without supporting user interaction.
- More complete samples that use platform-specific I/O (e.g., LEDs) and user interaction features (e.g., Unix signals). They are not as portable, but would already pass the applicable HomeKit self-certification tests.

3.1.1. Makefile

The makefile is similar for all accessory types.

3.1.2. DB.h and DB.c

DB.c contains a declaration of the HomeKit services and characteristics that the accessory should support, called the accessory attribute database. It is a C data structure that also contains pointers to functions that will be called by the HAP Library when the accessory needs to perform an action. The functions themselves are implemented in the separate files App.h and App.c.

For an overview of the Apple-defined HomeKit profiles, please review the current version of the *HomeKit Accessory Protocol Specification*.

3.1.3. App.h and App.c

App.c contains the implementation of the actual accessory logic, as functions that are called by the HAP Library whenever the accessory needs to perform an action. For example, the `HandleThermostatTargetTemperatureWrite` function must be implemented for a thermostat. It switches on a heating element when the user increases the thermostat's target temperature.

The original `DB.h / DB.c / App.c / App.h` samples provided by Apple are portable across platforms. Console output is used to simulate the behavior of these accessories. Chipset vendors may provide additional samples that demonstrate vendor-specific aspects, e.g., how to access GPIOs or PWMs for controlling an LED. Accessory manufacturers modify these files according to the needs of their accessories.

3.1.4. Main.c

This C file contains the main function with code for initializing the accessory logic state and the PAL, possibly termination code, and code for controlling the HomeKit accessory server that is implemented in the HAP Library.

3.2. Implement the Accessory Logic

This section describes the steps for implementing the accessory logic for a new product. A single HomeKit API is needed for this purpose: the HAP API, which is provided in the header file `ADK/HAP/include/HAP.h`. This API was designed to impose minimal structure on the accessory logic. It is designed to enable use of interrupt handlers, threads or OS processes of the platform.

3.2.1. Select sample code

Use the `LightbulbLED` or `ThermostatSimulation` sample code as a starting point for light bulbs or thermostats. The same samples can be used as starting points for other relatively simple accessories (e.g., leak sensor, switch, fan, etc.).

Use the Lock sample code as a starting point for door locks.

Use the `IPCamera` sample code in the POSIX sample SDK as starting point for cameras and video doorbells.

Use the `BridgeSimulation` sample code in the POSIX sample SDK for HomeKit bridges.

3.2.2. Define accessory attribute database

Define the accessory attribute database with the IIDs of the involved services and characteristics (e.g., in `DB.c`) by modifying the selected sample code. Definition is done in a declarative way, by setting up C data structures for accessories, services and characteristics. The details of these elements are described in the document *HomeKit Accessory Protocol Specification*, in the chapters *Apple-defined Characteristics*, *Apple-defined Services* and *Apple-defined Profiles*.

In `App.c`, an accessory object is defined in this manner:

```
static HAPAccessory accessory = {

    .aid = 1,
```

```
    .category = kHAPAccessoryCategory_Lighting,
    .name = "Acme Lightbulb LED",
    .manufacturer = "Acme",
    .model = "Lightbulb1,1",
    .serialNumber = "099DB48E9E28",
    .firmwareVersion = "1",
    .hardwareVersion = "1",
    .services = (const HAPService *const[]) {
        &accessoryInformationService,
        &hapProtocolInformationService,
        &pairingService,
        &lightbulbService,
        NULL
    },
    .cameraStreamConfigurations = NULL,
    .callbacks = {
        .identify = IdentifyAccessory
    }
};
```

- The accessory instance id `aid` must be set to 1 for non-bridged accessories and for HomeKit bridges themselves. For bridged accessories, it must be a number in the range 2 to 2^{64} - 1.
- The accessory object contains an array of service objects. In this case an `Accessory Information` service, a `Protocol Information` service, a `Pairing` service, and a `Lightbulb` service. Every accessory object must provide an `Accessory Information` service and the `Protocol Information` service. Every BLE accessory must provide the Pairing service, which is ignored on IP accessories.
- The `const` keyword is used in order to enable the toolchain to put the accessory attribute database into flash memory on systems where RAM is scarce.
- After a firmware update, the field `firmwareVersion` must be incremented. Everything else required according to the HomeKit Accessory Protocol specification is done by the HAP Library, e.g., incrementing the config number (CN). Note that firmware downgrades are not allowed.

To decouple the aspects of an accessory that are product-specific from those that are typically the same, or very similar, for all accessories of a particular HomeKit profile, the rest of the accessory attribute database is placed in the file DB.c. There, declarations of instance IDs for the supported services and their characteristics are needed:

```
#define kIID_AccessoryInformation                     ((uint64_t) 0x0001)
```

```
#define kIID_AccessoryInformationIdentify            ((uint64_t) 0x0002)
#define kIID_AccessoryInformationManufacturer        ((uint64_t) 0x0003)
#define kIID_AccessoryInformationModel               ((uint64_t) 0x0004)
#define kIID_AccessoryInformationName                ((uint64_t) 0x0005)
#define kIID_AccessoryInformationSerialNumber        ((uint64_t) 0x0006)
#define kIID_AccessoryInformationFirmwareRevision    ((uint64_t) 0x0007)
#define kIID_AccessoryInformationHardwareRevision    ((uint64_t) 0x0008)
#define kIID_AccessoryInformationADKVersion          ((uint64_t) 0x0009)

#define kIID_HAPProtocolInformation                  ((uint64_t) 0x0010)
#define kIID_HAPProtocolInformationServiceSignature  ((uint64_t) 0x0011)
#define kIID_HAPProtocolInformationVersion           ((uint64_t) 0x0012)

#define kIID_Pairing                                 ((uint64_t) 0x0020)
#define kIID_PairingPairSetup                        ((uint64_t) 0x0022)
#define kIID_PairingPairVerify                       ((uint64_t) 0x0023)
#define kIID_PairingPairingFeatures                  ((uint64_t) 0x0024)
#define kIID_PairingPairingPairings                  ((uint64_t) 0x0025)

#define kIID_Lightbulb                               ((uint64_t) 0x0030)
#define kIID_LightbulbServiceSignature               ((uint64_t) 0x0031)
#define kIID_LightbulbName                           ((uint64_t) 0x0032)
#define kIID_LightbulbOn                             ((uint64_t) 0x0033)

HAP_STATIC_ASSERT(kAttributeCount == 9 + 3 + 5 + 4,
AttributeCount_mismatch);
```

- An instance ID `iid` must be in the range of 1 to 2^{64} - 1 for IP accessories, 1 to 2^{16} - 1 for BLE accessories. Once an accessory is paired with its controller, an IID must remain the same throughout the lifetime of the accessory pairing. This means in particular across firmware updates, and also when someone migrates an accessory from a non-ADK based HomeKit SDK to one that is based on the ADK. Also, an IID that is not used anymore, after a firmware update, must not be reused for anything else. IIDs are chosen from a single ID pool for all characteristics and services of an accessory, except for a HomeKit bridge where every bridged accessory has its own pool.

- The IIDs above are grouped by services. The sample accessory supports the `Accessory Information`, `HAP-BLE Protocol Information`, `Pairing` and `Lightbulb` services. The first two are mandatory HomeKit services. The `Pairing` service is a HomeKit service required only

for BLE accessories. It is recommended to define it even for IP accessories, so that adding BLE support later on would be simplified.

- The samples are structured such that their implementation of all three HomeKit system services (`Accessory Information, HAP-BLE Protocol Information`, and `Pairing`) can be reused as is, with only their IIDs possibly having to be modified. This is possible because accessory-specific information that is exposed particularly in the Accessory Information service, such as `manufacturer` and `serialNumber`, are already provided in the `HAPAccessory` object in the separate file `App.c`.
- The `AccessoryInformation` service must contain the `ADKVersion` characteristic, see the `DB.c` files of the samples.
- The assertion check after the IID definitions helps detect inconsistencies, especially after changing the set of services or after adding characteristics. `kAttributeCount` is the total number of IIDs and must be correct. Both characteristics and services are attributes with IIDs, and therefore must be counted.

A `Lightbulb` service is defined in this manner:

```
const HAPService lightbulbService = {
    .iid = kIID_Lightbulb,
    .serviceType = &kHAPServiceType_Lightbulb,
    .debugDescription = kHAPServiceDebugDescription_Lightbulb,
    .name = "Lightbulb",
    .properties = {
        .primaryService = true,
        .hidden = false,
        .ble = {
            .supportsConfiguration = false
        }
    },
    .linkedServices = NULL,
    .characteristics = (const HAPCharacteristic *const[]) {
        &lightbulbServiceSignatureCharacteristic,
        &lightbulbNameCharacteristic,
        &lightbulbOnCharacteristic,
        NULL
    }
};
```

- The service object contains an array of characteristic objects. In this case a `Service Signature` characteristic, a Name characteristic, and an On characteristic. The latter controls the on/off state of the lightbulb.

An On characteristic is defined in this manner:

```
const HAPBoolCharacteristic lightbulbOnCharacteristic = {
    .format = kHAPCharacteristicFormat_Bool,
    .iid = kIID_LightbulbOn,
    .characteristicType = &kHAPCharacteristicType_On,
    .debugDescription = kHAPCharacteristicDebugDescription_On,
    .manufacturerDescription = NULL,
    .properties = {
        .readable = true,
        .writable = true,
        .supportsEventNotification = true,
        .hidden = false,
        .requiresTimedWrite = false,
        .supportsAuthorizationData = false,
        .ip = {
            .controlPoint = false
        },
        .ble = {
            .supportsBroadcastNotification = true,
            .supportsDisconnectedNotification = true,
            .readableWithoutSecurity = false,
            .writableWithoutSecurity = false
        }
    },
    .callbacks = {
        .handleRead = HandleLightbulbOnRead,
        .handleWrite = HandleLightbulbOnWrite
    }
};
```

- This is a Boolean characteristic, as indicated by its type `HAPBoolCharacteristic`, so its value at runtime will be either `true` or `false`. `HAP.h` supports a number of other types for characteristics, e.g., integer and floating point numbers, strings, etc.
    - Make sure that the type matches the format, e.g., a `HAPBoolCharacteristic` must have `kHAPCharacteristicFormat_Bool`, a `HAPUInt8Characteristic` must have `kHAPCharacteristicFormat_UInt8`, etc.
- The characteristic object contains two callbacks: `handleRead` and `handleWrite`. They are bound to the functions `HandleLightbulbOnRead` and `HandleLightbulbOnWrite`.
- The `lightbulbServiceSignatureCharacteristic` is ignored on IP accessories, but is required on BLE accessories for services that are *linked*, *primary*, *hidden* or *support configuration*. See `HAP.h` (search for "service properties") or the *HomeKit Accessory Protocol Specification* for more information.

3.2.3. Implement callbacks

Implement the callbacks that provide the actual behavior of the accessory, e.g., to change the set point of a thermostat (in `App.c`).

Callbacks are blocking, meaning it is the accessory logic's responsibility to immediately process them without waiting. This is possible because HAP is essentially a protocol for the remote reading and writing of variables (characteristics) over a network, which should not involve lengthy processing on the accessory.

In the lightbulb sample, these two functions are typical examples of callback implementations:

```
HAPError HandleLightbulbOnRead(
    HAPAccessoryServerRef *server __unused,
    const HAPBoolCharacteristicReadRequest *request __unused,
    bool *value,
    void *_Nullable context __unused)
{
    *value = accessoryConfiguration.state.lightbulbOn;
    HAPLogInfo(&kHAPLog_Default, "%s: %s", __func__, *value ? "true" :
"false");
    return kHAPError_None;
}
and
HAPError HandleLightbulbOnWrite(
    HAPAccessoryServerRef *server,
    const HAPBoolCharacteristicWriteRequest *request,
    bool value,
    void *_Nullable context __unused)
```

```
{
    HAPLogInfo(&kHAPLog_Default, ”%s: %s", __func__, value ? "true" :
”false”);
    if (accessoryConfiguration.state.lightbulbOn != value) {
        accessoryConfiguration.state.lightbulbOn = value;
       if (value) {
            TurnOnLightbulb();
        } else {
            TurnOffLightbulb();
        }
        SaveAccessoryState();
        HAPAccessoryServerRaiseEvent(
            server,
            request->characteristic,
            request->service,
            request->accessory);
   }
   return kHAPError_None;
}
```

- `HAPLogInfo` is used to produce log output that can be easily inspected during development. In a real accessory, the implementations of `TurnOnLightbulb()` and `TurnOffLightbulb()` would contain the code to query the platform for the state of the GPIO, PWM or other mechanism for controlling the light-emitting element of the lightbulb.

  `HAPLogInfo` relies on the portable `HAPStringWithFormat` function for string-formatting to minimize dependencies on the C standard library. Especially on small embedded systems, the various C standard libraries do not support all possible features such as `%ll` or `%z` format specifiers. `HAPStringWithFormat` only offers a subset of the full `printf` style format specifiers:

   - flags: 0, +, ' '
   - width: number
   - length: l, ll, z
   - types: %, d, i, u, x, X, p, s, c

  Precisions as well as dynamic widths (*) are not supported.

- The variable `accessoryConfiguration` contains the accessory logic state, in this case the state of the lightbulb (on/off).

- `HAPAccessoryServerRaiseEvent` tells the HAP server that this characteristic has changed, which may require sending of notifications. This function must always be called after the value of a characteristic was changed by the accessory logic.

- `HAPAccessoryServerRef` is an example of an opaque pointer. Accessory logic must access its internals only via the access functions provided. For custom implementations (e.g., `HAPPlatformKeyValueStoreRef` in the PAL), the ADK only accesses it via the access functions needed, never directly.

3.2.4. Start the accessory server

After setting up the accessory attribute database descriptors with their callbacks, initialize the platform and then call the functions `HAPAccessoryServerCreate` and `HAPAccessoryServerStart` with the descriptors as arguments. Then call the `HAPPlatformRunLoopRun`, which handles all protocol processing and invokes the callbacks when needed. This whole initialization part can be platform-specific, and therefore is placed in a separate file for the sample program (`Main.c`). Also, it may be specific to the transport being used (here: IP only, without BLE) and specific to the accessory category (here: lightbulb).

The central object is a HomeKit accessory server, declared as follows:

```
static HAPAccessoryServerRef accessoryServer;
```

In `main()`, first the platform is initialized. This is where the accessory logic developer needs to know something of the PAL, namely the initialization functions that it provides, which can be different from PAL to PAL. Then the memory for the BLE or IP stack is provided, in the form of static variables. Note that the HAP Library does not allocate any memory dynamically.

Worst-case memory requirements depend on the number and size of the characteristic values in the accessory database. In the case of HomeKit bridges, it also depends on the number of bridged accessories. It may take a few runs with the HomeKit Certification Assistant (HCA) to find values that are small but still large enough to handle all requests from a HomeKit controller.

In `Main.c`, the samples use HAP-defined default constants for IP buffer sizes and the like. Start with these defaults, and then change the values as needed. These are the minimal values, appropriate for simple HomeKit accessories like lightbulbs or sensors:

- Replace `kHAPIPSessionStorage_DefaultNumElements` by 9 to reduce the number of HomeKit connections that can be open simultaneously.

- Replace `kHAPIPSession_DefaultInboundBufferSize` by 768 to reduce the buffer space for messages coming from a HomeKit controller (one buffer per connection). With ADK 1.2 and later, this size is sufficient for a minimal Lightbulb sample to pass all certification tests. For a HomeKit bridge, the value should not be less than 1536.

- Replace `kHAPIPSession_DefaultOutboundBufferSize` by 1536 to reduce the buffer space for messages going to a HomeKit controller (one buffer per connection).

- Replace `kHAPIPSession_DefaultScratchBufferSize` by 1536 to reduce the buffer size for various protocol operations (one buffer per accessory/bridge).

If you replace one or more of the above values, be sure to replace all occurrences. Make sure that the number of parallel HomeKit sessions that is supported is correctly configured. On a PAL that is derived from the POSIX PAL, this means that `ipAccessoryServerStorage.numSessions` (see below) must

have the same value as the option `HAPPlatformTCPStreamManagerOptions.maxConcurrentTCPStreams`.

The code will be similar to this:

```
HAPAssert(HAPGetCompatibilityVersion() == HAP_COMPATIBILITY_VERSION);

HAPAccessoryServerCreate(&accessoryServer,
    &(const HAPAccessoryServerOptions) {
        .maxPairings = kHAPPairingStorage_MinElements,
        .ip = {
            .available = true,
            .accessoryServerStorage = &ipAccessoryServerStorage,
            .wacAvailable = true,
        }
    },
    &(const HAPPlatform) {
        .accessorySetup = &platform.accessorySetup,
        .keyValueStore = &platform.keyValueStore,
        .ip = {
            .tcpStreamManager = &platform.tcpStreamManager,
            .serviceDiscovery = &platform.serviceDiscovery,
            .wac = {
                .softwareAccessPoint = &platform.softwareAccessPoint,
                .wiFiManager = &platform.wiFiManager
            }
        },
        .authentication = {
            .mfiHWAuth = &platform.mfiHWAuth
            // Set to NULL if no Apple authentication coprocessor available.
        }
    },
    &(const HAPAccessoryServerCallbacks) {
        .handleUpdatedState = AccessoryServerHandleUpdatedState
    },
    /* context: */ NULL);
```

```
AppCreate(&accessoryServer, &platform.keyValueStore);

HAPAccessoryServerStart();

HAPPlatformRunLoopRun();
```

- `HAPAccessoryServerStart` starts HAP for the `lightbulbAccessory` object that has been defined earlier.
- Before that, an `accessoryServer` object must have been created with `HAPAccessoryServerCreate`.
- `HAPAccessoryServerCreate` has a `HAPAccessoryServerOptions` parameter, which controls whether HAP over IP or HAP over BLE is used, and for HAP over IP, whether or not Wi-Fi (and thus WAC2) is used.
- Storage needed for the IP or BLE stack is provided in the `HAPAccessoryServerOptions` parameter.
- `HAPAccessoryServerCreate` has a `HAPPlatform` parameter, which must embody the PAL services to be used by the HAP library. In the example, the platform parameters have first been combined in a global variable to make platform dependencies easy to find:

```
static struct {

    HAPPlatformKeyValueStore keyValueStore;
    HAPPlatformAccessorySetup accessorySetup;
    HAPPlatformTCPStreamManager tcpStreamManager;
    HAPPlatformServiceDiscovery serviceDiscovery;
    HAPPlatformSoftwareAccessPoint softwareAccessPoint;
    HAPPlatformWiFiManager wiFiManager;
    HAPPlatformMFiHWAuth mfiHWAuth;

} platform;
```

- The `platform` variable contains those PAL services that need to be instantiated, and that are relevant for the given accessory. They are initialized individually and will be used at runtime once the server has been started. In contrast, the camera part of the PAL is not instantiated here, because it is not relevant for a lightbulb. The random number generator in the PAL is an example of a service that exists only as a single instance and thus needs no explicit instantiation.
- For an IP accessory, an IP manager object must be provided.
- For a BLE accessory, a BLE manager object must be provided.
- Currently, concurrent IP and BLE operations are not supported, i.e., an accessory is either an IP or a BLE accessory. Either an IP or a BLE accessory must be provided, but not both.

- Providing an IP manager object is only possible if the PAL being used has implemented the PAL module `HAPPlatformTCPStreamManager`. In this case, `HAPPlatformBLEPeripheralManager` need not be implemented (meaning that the empty implementation is used, where all functions return fatal errors when called).

- Providing a BLE manager object is only possible if the PAL being used has implemented the PAL module `HAPPlatformBLEPeripheralManager`. In this case, `HAPPlatformTCPStreamManager` need not be implemented (meaning that the empty implementation is used, where all functions return fatal errors when called).

- All memory provided through arguments to `HAPAccessoryServerCreate` must stay valid until the accessory server has been released (`HAPAccessoryServerRelease`). Be careful to ensure that this condition holds if you call `HAPAccessoryServerCreate` from a scope in which the arguments do not automatically stay valid until the accessory server has been released.

- The assertion `HAPAssert(HAPGetCompatibilityVersion() == HAP_COMPATIBILITY_VERSION)` checks whether the HAP Library's version, as returned by `HAPGetCompatibilityVersion`, has been taken from the same ADK version as the `HAP.h` header file which contains the definition of `HAP_COMPATIBILITY_VERSION`, to detect inadvertent version mixups.

- `AppCreate` is a helper function that initializes the accessory and its platform. For example, it sets up the key-value store of the PAL and uses it to load the previous accessory state.

- **Important**: The HAP Library runs as a single execution context (e.g., a thread) in one loop, and *all* HAP* functions (with only one exception) must be called from this execution context. The blocking function `HAPPlatformRunLoopRun` implements the run loop, which manages and dispatches I/O events and drives the scheduling of timer events. When calling a HAP* function from another execution context, it must be scheduled by calling `HAPPlatformRunLoopScheduleCallback`. This is the only function that is always safe to call from any execution context (e.g., some other thread).

    - For example in `LightbulbLED`, `HAPAccessoryServerStop` is called in `HandleSignalCallback`, which is scheduled using `HAPPlatformRunLoopScheduleCallback` in `HandleSignal`.

    - As another example in `LightbulbLED`, `HAPAccessoryServerRaiseEvent` is called in `ToggleLightbulbState`, which is called in `HandleSignalCallback` and therefore also scheduled for execution by the run loop.

    - Callbacks issued by the HAP Library, from `HAPPlatformRunLoopRun`, are already on the ADK's execution context. This is the reason why in `LightbulbLED`, the call to `HAPAccessoryServerRaiseEvent` in `HandleLightbulbOnWrite` is not scheduled.

3.2.5. Stop the accessory server

Normally, the HomeKit accessory server is never stopped as long as an accessory is powered. To stop it anyway, follow this sequence of steps as illustrated in the `LightbulbLED` sample:

- Make sure that you are running on the ADK execution context, see: `HAPPlatformRunLoopScheduleCallback` in `HandleSignal` in `App.c`.

- Stop the accessory server, see:
  `HAPAccessoryServerStop` in `HandleSignalCallback` in `App.c`.
- Wait for the accessory server to transition to the idle state. Then stop the run loop. See:
  `HAPPlatformRunLoopStop` in `AccessoryServerHandleUpdatedState` in `App.c`.
- Release the accessory server, see:
  `HAPAccessoryServerRelease` in `Main.c`.
- Release the run loop, see:
  `HAPPlatformRunLoopRelease` in `DeinitializePlatform` in `Main.c`.

3.2.6. Troubleshooting

Before starting to use your own accessory logic code, ensure that the basic, portable lightbulb sample (`ADK/Applications/Lightbulb`) works correctly on your platform (Not the `LightbulbLED` sample since it assumes a Raspberry Pi 3 as its platform). See Developer Technical Support for ADK for more information about requesting code-level technical support from Apple.

If there are issues during compilation or linking:

- Use the current ADK version.
- Use a toolchain that is compatible with the chosen HAP Library binary, with compatible compiler flags. The toolchains that Apple has used for creating the HAP Library binaries, along with the used compiler flags, are described in the `Configuration.txt` that comes with a HAP Library binary, e.g., `ADK/HAP/lib/ARM/GCC/arm-none-eabi/thumb/v7e-m/Configuration.txt`.
- Make sure the chosen HAP Library binary, the toolchain, and the compiler flags assume the existence of suitable hardware floating-point support - or assume the absence of such support. The floating-point ABI must match like the rest of the ABI.
- If during linking you get a linker error similar to undefined reference to 'fmodf' then your clib is missing some required functionality, typically floating-point operations.
- Include a description of the toolchain, the build number of the HAP Library binary, the compiler/linker flags, and the compiler/linker log.

If there are issues at runtime:

- Make sure that—if there is sufficient memory on the development platform—the system is built with debug logging enabled. For example, this is done with make debug on the Raspberry Pi. Adapt the makefiles accordingly, by setting the `-DHAP_LOG_LEVEL=3` flag. The log will be written to the console.
- If there is an error in the log that says "precondition failed", the log message also says what is expected in this place. A precondition failure means that a function is called with illegal arguments or that the system is in an erroneous state that is not acceptable for this function to run correctly. For example, a function may be called with a null value for a non-nullable pointer parameter. The ADK systematically checks for precondition failures, which always indicate programming errors ("contract violations"). If one is detected, the ADK is aborted immediately ("fail fast") so that the program error is not masked.

Note: In general when dealing with the ADK, only pointers marked as `_Nullable` are optional.

- If you randomly encounter errors, e.g., leading to ADK aborts: please check that all calls to the HAP Library occur in the correct execution context. They must not occur from arbitrary threads. See Start the accessory server and Run loop control for more information.
- If text output lines of the HAL Library log look incomplete or corrupted: please check that all calls to the HAP Library occur in the correct execution context. They must not occur from arbitrary threads. See Start the accessory server and Run loop control for more information.
- Make sure use of the Apple Authentication Coprocessor is not the cause of the failure, by disabling it (temporarily during early development, not for a shipping product). You can do this with the build option `USE_MFI_HW_AUTH=0`.
- If your system fails during startup of the HAP Library, make sure that the modules `HAPPlatformAbort` and `HAPPlatformLog` work, by writing small test programs that you can use to test them individually, without the lightbulb or the HAP Library code being involved.
- If you suspect file system issues (e.g., because you changed POSIX PAL code related to storage), replace the Raspi implementation of the key-value store and accessory setup modules by the bootstrap implementations in `ADK/PAL/Bootstrap`. With these bootstrap modules, no file operations are performed. So, if the system starts up with them, then there is a problem with the file operation changes you have made.
- If you use a PAL that is derived from the POSIX PAL and you get the "Failed to allocate IP session" error message in the HAP log, then check that the option `HAPAccessoryServerOptions.ip.accessoryServerStorage.numSessions` has the same value as the option `HAPPlatformTCPStreamManagerOptions.maxConcurrentTCPStreams`.
- If you get the "Closing connection (inbound buffer too small)" error message in the HAP log, increase the value of `kHAPIPSession_DefaultInboundBufferSize`.
- For every characteristic in the accessory attribute database, make sure that the characteristic format matches the characteristic structure, i.e., a `HAPUInt8Characteristic` **must** have `kHAPCharacteristicFormat_UInt8`.
- Make sure that the necessary accessory setup information is provisioned (see Set up provisioning) and that the relevant `HAPPlatformAccessorySetup` functions properly return it (see Accessory Setup).
- If there is an issue with camera video or audio, make sure to switch on extended logging for these data streams. See Log for more information.
- If there are issues with floating point numbers, especially on microcontroller-based systems, the problem could be inconsistent use of ABIs for floating point arguments. The ABI dictates how float arguments are passed between functions. Be careful not to mix "soft-float" and "hard-float" code - the compiler may not always warn you of such errors.
- If there is an issue during firmware update testing, check that existing IIDs have remained the same as before the firmware update. This is particularly important if a firmware update now uses the ADK and the ADK samples and their IIDs are copied as is, without adapting them to their pre-ADK values.

- If you contact Apple Developer Technical Support, include a description of the toolchain and the target platform (hardware and software), the `compiler / linker` log, and the *complete* HAP log. The HAP log should be in the format that Apple uses for its ADK samples, i.e., as implemented in the POSIX sample SDK. Without the complete ADK log output as generated by the HAP Library, key information may be missing that is needed for technical support. The log may be supplemented by log output specific to your target platform, as long as it also contains the HAP Library's log output. Please only send full logs to Apple.

If there are issues with a profile implementation:

- Pairing with HAT and consulting its trace view for errors may assist with diagnostics.
- Make sure that the accessory attribute database is set up according to the profile definition of the latest HAP specification. If this is not the case, the HomeKit controller may abort pairing.

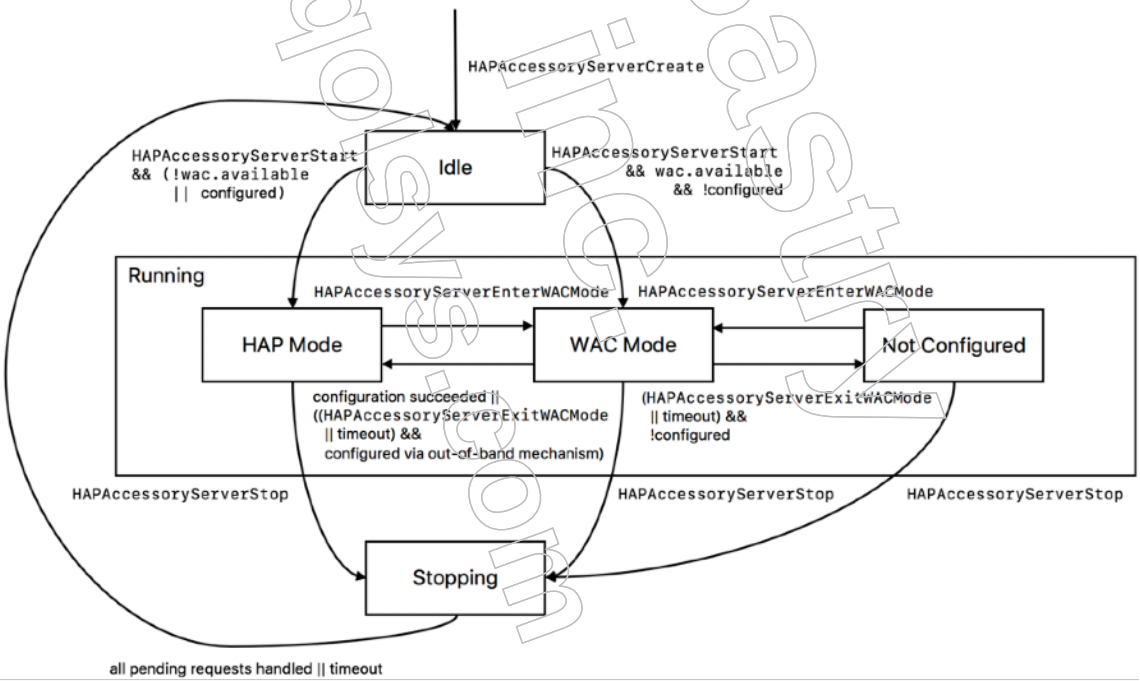
3.3. Protocols

This section discusses protocol aspects that are helpful to know when developing and debugging accessory logic code.

3.3.1. Wi-Fi Accessory Configuration 2 (WAC2)

A HomeKit Wi-Fi accessory that has not been used yet, or has been reset to factory settings, has no information on how to join the owner's Wi-Fi network, i.e., no network name (SSID) nor the password. HomeKit uses the WAC2 protocol to provide this information to the accessory during pairing, without the user having to enter the information manually.

The WAC process goes through several states. Knowing them can help during testing of an accessory. The following diagram shows the WAC state machine as implemented in the ADK:



- `Idle` state:
  An unpaired accessory goes into the `Idle` state after booting up and after `HAPAccessoryServerCreate` has been called by the accessory logic. This is the only state in which the accessory logic is allowed to call `HAPAccessoryServerStart`, which transitions the accessory into the Running state.

- `Running` state:
  The Running state has three substates: `HAP Mode`, `WAC Mode` and `Not Configured`. If the Wi-Fi credentials are already configured - which might have happened with some other mechanism than WAC on accessories that do not only support HomeKit - or if the accessory logic has set the `ip.wac.available` flag in the `HAPAccessoryServerOptions` to false, the accessory enters HAP Mode when started.

  - `HAP Mode` substate is the normal operational state when an accessory is running.

  - `WAC Mode` substate is entered automatically when the server has been created for an unconfigured accessory, or when a user requests it explicitly: typically via a hardware button. If no pairing occurs for 15 minutes, the accessory goes into the `Not Configured` mode, otherwise into `HAP Mode`. An accessory's user interface may also allow the user to explicitly `exit WAC Mode`, e.g., by pressing the WAC button again.

  - `Not Configured` substate is entered after a failed pairing attempt. The user can request the accessory to re-enter `WAC Mode`.

- `Stopping` state:
  When `HAPAccessoryServerStop` is called by the accessory logic, the accessory enters the Stopping state. In this state, pending HAP requests are processed, but no new requests are accepted anymore. A timeout ensures that the Idle state is entered eventually even if outstanding requests cannot be completed for some reason.

Some remarks regarding the accessory server functions:

- `HAPAccessoryServerStop` may be called anytime after `HAPAccessoryServerCreate`. It always causes a state transition to `Stopping` if not in `Idle`, or it remains there if already in `Idle`. This can be useful e.g. for setting up the Wi-Fi credentials through some other mechanism, see below.

- `HAPAccessoryServerStart` may only be called in `Idle` state, otherwise a precondition check fails.

- `HAPAccessoryServerEnterWACMode` may be called anytime after `HAPAccessoryServerCreate`. It has no effect, except if in `HAP Mode`, where it deletes the Wi-Fi credentials and transitions to `WAC Mode`.

- `HAPAccessoryServerExitWACMode` may be called anytime after `HAPAccessoryServerCreate`. It has no effect, except if in `WAC Mode`, where it transitions to `HAP Mode` or `Not Configured`, depending on whether or not the Wi-Fi network has been configured in the meantime by an out-of-band mechanism.

- `HAPAccessoryServerIsInWACMode` indicates whether the accessory is currently in the `WAC Mode` substate of state `Running`. Relevant state changes are also signaled through the

`HAPAccessoryServerCallbacks.handleUpdatedState` callback. Also refer to `HAPAccessoryServerGetState`.

Note: On BLE and Ethernet accessories, which do not require IWAC2, it is nevertheless permitted to use the WAC-related API functions of `HAP.h`. The function `HAPAccessoryServerIsInWACMode` then simply returns false.

3.3.2. Pairing

A HomeKit controller can only use a HomeKit accessory after it has been paired with it. A new HomeKit accessory, or an accessory that has been factory reset, is not paired with any HomeKit controller. An accessory can be paired after it has been powered on. A Wi-Fi accessory needs to be in `WAC Mode` (see previous section) for pairing, or must be added to the Wi-Fi network out-of-band. For accessories with a QR code or a read-only NFC tag (optional), nothing else is required. In contrast, for accessories that support programmable NFC tags (specify USE_NFC in the makefile) or displays (specify USE_DISPLAY in the `makefile`), the accessory logic must call function `HAPAccessoryServerEnterPairingMode` in order to enable pairing.

`HAPAccessoryServerEnterPairingMode` causes a dynamically-generated setup code to be shown on the display, or a static setup code to be advertised over programmable NFC. (To do this, `HAPAccessoryServerEnterPairingMode` internally calls function `HAPPlatformAccessorySetupUpdateSetupPayload` of PAL module Accessory setup.) Once the special pairing mode has been entered, the HomeKit controller's camera or NFC hardware can be used to read the setup code and to run the HomeKit pairing protocol. After a pairing attempt (whether successful or failed) or if no pairing attempt occurs within five minutes, this special mode is automatically left again. While a pairing is in progress, the pairing mode timer is extended indefinitely until a pairing attempt is registered or pairing is cancelled. The setup code does not change while a pairing is in progress.

if you need to be notified about pairing status changes, use the callback `handleStateUpdated` in `HAPAccessoryServerOptions` (see `HAP.h` for more information). In this callback, you may call `HAPAccessoryServerIsPaired` if you need that information.

Note: An unpaired accessory is always pairable when the accessory server is running, even when the special pairing mode has not been entered (e.g., by using the iOS accessory picker). `HAPAccessoryServerEnterPairingMode` just pre-generates the QR code.

For debugging the pairing behavior of IP accessories, it can be helpful to observe the Bonjour TXT status flags (see also `Table 6-8 Bonjour TXT Status Flags` in the HAP specification) using a Bonjour browser app such as `Discovery - DNS-SD Browser` from the App Store. These flags, which are broadcast over the IP network, are also shown in the ADK log output:

- The value of `sf` should be `1` if the accessory is not paired. In this case, the accessory should be visible in the `Add accessory` screen of the HomeKit controller.
- The value of `sf` should be `0` if the accessory is paired.

For accessories that support displays or programmable NFC, it is the accessory manufacturer's responsibility to provide an appropriate mechanism for entering this special mode. The mechanism must require physical presence of a user, so that no remote attacks can cause the pairing mode to be entered.

Typically, a dedicated hardware button is provided that triggers the call to `HAPAccessoryServerEnterPairingMode`.

The following is required for licensed accessories:

- The accessory must require users to explicitly trigger NFC pairing mode via a physical interaction.
- Accessories implementing programmable NFC must not implicitly be ready to pair once pairing is removed.

Note: When the user initiates a manual pairing in the Home app's accessory browser (list of unpaired HomeKit accessories), the HAP Library automatically enters the pairing mode without further user intervention. This behavior must be modified for programmable NFC tags when going through production to require user interaction in this case, i.e., by not running `HAPAccessoryServerStart` until the user triggers pairing mode.

Note: The POSIX samples in the ADK have only two signals available (SIGUSR1 and SIGUSR2) and these are already used for other purposes. As no other signal is available for triggering `HAPAccessoryServerEnterPairingMode`, this function is triggered automatically after every power cycle (switching accessory off and on again). The BLE samples behave in the same way for consistency. This workaround is only intended for internal testing and debugging.

3.3.3. Firmware Updates

HomeKit does not specify a mechanism for firmware updates, but defines some requirements in the specification, and provides basic support in the ADK:

- After a firmware update, the field `firmwareVersion` in the `HAPAccessory` object must be incremented by the accessory logic. The remaining necessary state changes are automatically done by the HAP Library, e.g., incrementing the config number (CN).
- An IID must remain the same throughout the lifetime of the accessory pairing. This means in particular across firmware updates. If an IID is not used anymore after a firmware update, it must not be reused for anything else.
- See also section <u>Versioning</u> in this guide and the relevant sections of the HAP specification.

3.4. Profiles

This section discusses advanced topics for selected HomeKit profiles in more details.

3.4.1. Programmable Switch

There are three `Programmable Switch` events:

- Single Press
- Double Press
- Long Press

When defining the `Programmable Switch` event characteristic, make sure that the readable property is set to `true`:

```
static const HAPUInt8Characteristic
    programmableSwitchEventCharacteristic = {

    .format = kHAPCharacteristicFormat_UInt8,

    .iid = kIID_ProgrammableSwitchEvent,

    .characteristicType = &kHAPCharacteristicType_ProgrammableSwitchEvent,

    .debugDescription =
        kHAPCharacteristicDebugDescription_ProgrammableSwitchEvent,

    .manufacturerDescription = …,

    .properties = {

        .readable = true,

        .writable = false,
```

If this is not the case, iOS may not be able to pair with the accessory.

handleRead for this characteristic is only invoked in response to `RaiseEvent`. Its implementation should return the last event triggered. This ensures that the code works as specified on both IP and BLE.

3.4.2. Remote

The Apple TV `Remotes` profile allows a HomeKit accessory to control a target, which is an Apple TV or a HomePod. Please refer to the accessory logic samples in `ADK/SDK–Samples/POSIX/Raspi/Applications/Remote` and `ADK/SDK–Samples/POSIX/Raspi/Applications/RemoteBridge`.

As this profile needs relatively complex logic that is the same for all products, that common logic has been factored out into the helper files `Remote.h/.c`. In order to simplify migration to future ADK versions, it is recommended that a licensee not modify these helper files - so that they can easily be replaced by newer versions.

The Apple TV `Remotes` profile uses the *HomeKit Data Stream* (HDS) protocol for connections to a HomeKit hub (e.g., an Apple TV or HomePod). A `Remote` service requires at least 8 KB of RAM for HDS, 16 KB are recommended. In addition, about 160 KB of RAM is needed for audio buffers and other data structures used in the `Remote` accessory logic and in the microphone PAL module (e.g., GStreamer buffers).

The maximum number of supported targets can be configured. In the `Remote` sample, every target gets its own key-value store domain, defined in `Remote.c`:

```
/**
 * Key value store domain for the first target of the remote.
 */
#define kAppKeyValueStoreDomain_TargetBegin    ((HAPPlatformKeyValueStoreDomain) 0x01)


/**
```

```
 * Key value store domain for the last target of the remote.
 */
#define kAppKeyValueStoreDomain_TargetEnd      ((HAPPlatformKeyValueStoreDomain) 0x14)
```

In such a domain, the configuration of the target (e.g., Apple TV), is stored.

From this information, the maximum number of targets is computed as `kRemote_MaxTargets`.

The configuration of the `Remote` itself is stored in another domain (`kAppKeyValueStoreDomain_Configuration`).

3.4.3. HomeKit Bridge

A HomeKit bridge enables HomeKit controllers to access non-HomeKit accessories. In a situation without bridge, HomeKit establishes end-to-end security, with a HomeKit controller being one of the endpoints and an accessory being the other endpoint. In a situation with a HomeKit bridge, the bridged accessories by definition do not support HAP and so the bridge becomes the endpoint of the secure connection. As a result, the bridged accessories do not give the same strict security guarantees that HomeKit accessories normally provide. To mitigate this fact, HomeKit defines restrictions on the profiles and protocols that may be supported by a bridge. See the *HomeKit Accessory Protocol Specification* for more details. A HomeKit bridge itself is also a HomeKit accessory and must be an IP accessory (Wi-Fi or Ethernet).

Accessory attribute database of a bridge:

A bridge must represent the accessory attribute databases of the accessories that it currently bridges to, as separate HAP accessory objects. A bridge may define an upper limit of HAP accessory objects (at most 150, according to the specification). Sufficient memory for this number of accessories should be statically allocated, to ensure robust operation. In the samples, this is done in App.c like this:

```
static AccessoryState accessoryState[kAppState_NumLightbulbs];
```

The maximum number of bridged lightbulbs is defined as the constant `kAppState_NumLightbulbs` in file `DB.h`. See `ADK/SDK-Samples/POSIX/Applications/Bridge` and `ADK/SDK-Samples/POSIX/Applications/BridgeSimulation` for more details of bridges for `Lightbulb`-compatible accessories, and ADK/SDK-Samples/POSIX/Applications/RemoteBridge for Remote accessories.

The amount of memory that needs to be allocated depends on the accessory attribute databases of the bridged accessories, i.e., on their profiles, more exactly on the number of attributes that have to be represented. For a simple lightbulb, the number of attributes is calculated like this (see `ADK/Applications/Lightbulb/DB.c`):

```
kAttributeCount = 9 + 3 + 5 + 4
```

i.e., the sum of the number of attributes for the `Accessory Information` service (9), `Protocol Information` service (3), `Pairing` service (5), and the `Lightbulb` service (4).

For bridges, each bridged accessory requires another Accessory Information service plus the number of attributes for the bridged accessories themselves:

```
kAttributeCount = 9 + 3 + 5 + (8 + 4) * kAppState_NumLightbulbs
```

Note that the bridged accessory itself exposes the ADKVersion characteristic - as it is implemented using the ADK - while the bridged lightbulbs do not - as they are not implemented using the ADK. This explains why it is (8 + 4) and not (9 + 4) in the expression above.

Bridge reconfiguration:

Sometimes, bridges need to be reconfigured, e.g., for adding another bridged accessory. In such a case, the HomeKit accessory server must be stopped using function `HAPAccessoryServerStopBridge`. Once the accessory server has stopped, the callback `handleUpdatedState` from `HAPAccessoryServerCallbacks` is called. Then the list of bridged accessories is changed (note that the list must be null-terminated), and then the accessory server must be started again using function `HAPAccessoryServerStartBridge` with argument `configurationChanged` set to `true`. See also sections Stop the accessory server and Start the accessory server.

Callbacks for bridged characteristics:

When the HAP Library invokes a callback function in order to read or write a characteristic value, then the callback must return immediately. It must never wait for a bridged accessory that is connected via a slow communication channel. This means that for communication technologies such as ZigBee, the characteristic values must be cached.

In a read callback, the cached value of the characteristic is returned along with `kHAPError_None`. A cached value should be returned even if it is not known to be completely up-to-date. The cache is then updated as fast as the communication channel allows. If the previously cached value is not valid anymore and needs to be replaced with a new value, the HomeKit controller(s) should be notified of the new value by calling `HAPAccessoryServerRaiseEvent`.

In a write callback, the cached value of the characteristic is updated and `kHAPError_None` returned. If the new value is different from the previously cached value, the bridged accessory is updated as fast as the communication channel allows. If the bridged accessory signals that this update failed, the cache should be reverted to the old value and the HomeKit controller(s) should be notified of the old value by calling `HAPAccessoryServerRaiseEvent`.

For a HAP multi-characteristic write, the ADK invokes the callbacks of all involved characteristics in one go, meaning that any timer or other callbacks scheduled to the run loop will not execute before all pending characteristic write callbacks have been invoked.

4. Platform Development

A platform developer for an accessory manufacturer typically starts with an existing PAL implementation and then modifies the PAL as needed. Chipset vendors can provide optimized PAL implementations for their chipsets, to make HomeKit integration easier for accessory manufacturers and to ensure that their platform is implemented in an optimal way.

This section provides an overview over the tasks that accessory manufacturers face when using the POSIX SDK as starting point, outlines the tasks that chipset vendors face when developing platform support for their chipsets, describes the most important technical requirements that a platform must meet in order to support HomeKit, and describes the modules that make up a PAL.

4.1. Use a HomeKit SDK

The major tasks involved in using an ADK-based HomeKit SDK are described here for the POSIX SDK that is provided as part of the ADK in `ADK/SDK-Samples/POSIX`. For third-party SDKs, their respective documentation should be consulted. If no SDK is available for the target platform, see Create a HomeKit SDK.

4.1.1. Check feasibility

The typical hardware for a Linux system, with an Apple Authentication Coprocessor attached via I2C, usually meets the basic HomeKit hardware requirements. In terms of software, some additional services may have to be installed and configured as required by the PAL, e.g., the implementation of an mDNS client. IP cameras and video doorbells may require more powerful hardware (e.g., video encoders) and additional system libraries may be needed for the audio/video pipeline.

More information about the technical requirements is provided in the Memory Requirements and PAL Modules sections below.

4.1.2. Select a HAP Library

In the ADK, check the document `ADK/HAP/lib/HAP libraries.txt`. It contains a list of the toolchains and binaries currently supported by Apple. For every HAP Library, the path to a directory is given that contains the library as a binary file plus a text file `Configuration.txt` with more information about the library, e.g., the compiler flags that have been used to create it. Choose and set up a toolchain that supports the target platform, e.g., a GCC compiler compatible with the target platform's ARM or MIPS microprocessor. Adapt the makefiles as needed. Additional toolchains will be supported in the future.

4.1.3. Configure the operating system

The POSIX SDK supports the Raspberry Pi 3 as reference platform. Complete makefiles for this platform are provided, as well as a preconfigured Raspbian image on which the necessary open source dependencies have already been installed. The target platform may have to be adapted accordingly: configure the target operating system as needed before HomeKit is deployed.

Make sure that the operating system is configured for a high level of security, e.g.:

- Only the minimally needed software packages are installed.
- Only the minimally needed services are started.
- Except for those that are absolutely necessary, all network ports and other I/Os are inactive.

- Memory accesses close to the address 0x0 are detected and prevented, e.g., by not mapping the null page.
- Ensure that all software can be updated quickly when vulnerabilities are discovered.
- For a production image: all development, debugging, manufacturing, and test mechanisms have been removed or disabled.

Configuration decisions regarding the operating system can directly affect PAL modules. For example, the decision to use Avahi instead of mDNSResponder means that the PAL module for Bonjour (aka service discovery) must be implemented in a different way.

4.1.4. Adapt the POSIX PAL

Go through the list of all PAL modules, adapt those that are relevant for the target platform and accessory category as needed.

Once the PAL is adapted and the operating system configured, the samples (e.g., `Lightbulb` sample) can be used for testing the PAL with HomeKit test tools (e.g., HAT and HCA). See section Troubleshooting for more information on how to troubleshoot if problems arise.

4.1.5. Develop or integrate the accessory logic

In the ADK, several accessory logic samples are provided in the `ADK/SDK–Samples/POSIX/Raspi/Applications` directory. Use one of them as a starting point for developing the product's accessory logic, or for integration with existing accessory logic.

For security and privacy reasons, make sure that only the data that is strictly necessary is acquired, communicated and stored.

4.1.6. Setup provisioning

Create a concept for how an accessory, during production or in the field, is provisioned with a setup code and related data.

For a licensed accessory, a random setup code and related data **must** be generated for every individual accessory according to the rules described in the HomeKit Accessory Protocol Specification, and as summarized here:

- Generate a random setup code in format XXX-XX-XXX with X being a digit from 0–9. The code must be generated using a cryptographically secure random number generator.
  - Codes that only consist of a repeating digit are not allowed.
  - 123–45–678 and 876–54–321 are not allowed.
- Generate a random SRP salt (16 random bytes) using a cryptographically secure random number generator.
- Derive the corresponding SRP verifier from the setup code and SRP salt.
- Generate a random setup ID in format XXXX with X being a digit from 0-9 or a character from A-Z. The ID must be generated using a cryptographically secure random number generator.
  - Lowercase characters are not allowed.

- Derive the setup payload from the setup code and setup ID.
  - For manual validation of this task, the HomeKit Accessory Simulator may be used by opening it, then pressing Cmd-2.

- Deploy the SRP salt, the SRP verifier and the setup ID to the accessory.
  - If the accessory supports programmable NFC - and only then - also deploy the setup code.
  - Depending on how these items are deployed to an accessory, `HAPPlatformAccessorySetup.c` must be adapted accordingly.

- Create a QR code sticker that contains the setup payload and setup code. The sticker will be used by users every time they want to pair the accessory to an iOS device.
  - Implement the QR code as specified in the HomeKit Identity Guidelines.

If the accessory has a display that supports displaying a setup code or QR code, only the setup ID has to be generated and deployed in the manufacturing process. The other steps involving setup code, SRP salt, SRP verifier, setup payload, and the QR code label may be ignored.

The following provisioning script serves as a reference implementation of the provisioning mechanism: `ADK/SDK-Samples/POSIX/SDK/Raspi/Tools/Provision`. With this tool, a target accessory's key-value store is provisioned with data that is compatible with the provided implementation of PAL module `HAPPlatformAccessorySetup`. If the PAL implementation is modified, e.g., in order to use some other storage mechanism for storing the provisioning information, then this tool must be adapted accordingly.

The `Provision` tool can be used for hardware authentication (Apple Authentication Coprocessor) as well as software authentication. Its usage is printed if `Provision` is run without any arguments.

4.1.7. Provide a secure firmware update mechanism

See section Versioning in chapter PAL Modules, and the HomeKit Accessory Protocol Specification for more details regarding updates of the HomeKit part of an accessory's firmware specifically.

Note *HomeKit Accessory Protocol Specification Addendum*: *App must ensure that HomeKit accessories are not updatable to a firmware version that is incompatible with the iOS version of the controller.*

4.1.8. Check before release

Before releasing the accessory logic code to production, ensure that:

- Only securely generated setup codes etc. are used - not a fixed code such as the 111-22-333 example code used in the SDK samples
- No log data is being produced
- MFi authentication is activated
- For Wi-Fi accessories: WAC2 support is activated
- For accessories with displays: display support is activated
- The `ADKVersion` characteristic is included in the `AccessoryInformation` service

4.2. Create a HomeKit SDK

The major tasks involved in creating an ADK-based HomeKit SDK are:

4.2.1. Check feasibility

Check whether the selected chipset meets the minimum requirements to support HAP. More information about the technical requirements is provided in the Memory Requirements and PAL Modules sections.

4.2.2. Select a toolchain and a HAP Library

In the ADK, check the document `ADK/HAP/lib/HAP libraries.txt`. It contains a list of the toolchains and binaries currently supported by Apple. For every HAP Library binary, the path to a directory is given that contains the library as a binary file plus a text file `Configuration.txt` with more information about the library. Choose one or more toolchains that support the selected chipsets, e.g., a GCC compiler for a Linux-based accessory or an ARM Keil compiler for a Cortex-M4F product. Additional toolchains will be supported in the future.

For resource-constrained, microcontroller-based accessories, two binaries are provided for every toolchain: a standard library that is especially useful during development because it produces log output that is helpful for debugging, plus a library with the `_min suffix` for production purposes, which is optimized for minimal size. Use the standard library during development if memory resources of your development platform allow this. Use the `_min library` for the target platform for production purposes (or the standard Library binary, but with logging disabled or a null implementation used for the Log PAL module).

4.2.3. Define a directory structure

Define the directory structure of the HomeKit SDK. Use the same structure as demonstrated by the sample SDKs, which are located at `ADK/SDK-Samples`, if possible. To start:

- Copy all applicable application samples from `ADK/Applications`
- Copy the applicable library (or libraries) from `ADK/HAP/lib`.
- Copy the complete contents of `ADK/PAL`.

4.2.4. Develop a PAL

Implement the PAL modules required for an accessory. For a high-level description of the PAL modules, refer to PAL Modules. The header files are located in `ADK/PAL/include`. "Null" implementations that can be used as templates are provided in `ADK/PAL/Templates`. Their functions are not actually implemented but write error messages to the log and then fail fatally, indicating that they are not yet implemented. For licensed accessories, these implementations must be modified according to the specification given in this document - unless a module is not needed for the given type of accessory. For example, the null implementation of the IP Camera module should be left as is for all accessories except IP cameras and video doorbells.

In addition to null implementations, bootstrap implementations are provided for some PAL modules, i.e., implementations that are not platform-dependent, must not be used for licensed accessories, but are useful to allow end-to-end testing of a working prototype as quickly as possible. For example, a key-value store that works in main memory instead of making the key-value pairs persistent in some way (which would be platform-dependent). These files are located in `ADK/PAL/Bootstrap`.

For chipset vendors that have an existing HomeKit SDK, it should be possible to repackage the equivalent functionality of the SDK into a PAL implementation, adhering to the PAL API that is prescribed by the ADK.

A PAL can be distributed as a single binary file, or as a collection of library and/or source files in the SDK.

The ADK contains the source code of a complete PAL for POSIX - with Raspberry Pi 3 as reference platform and a PAL for a BLE chipset. The files are located in `ADK/SDK-Samples/POSIX/Raspi/PAL` and `ADK/SDK-Samples/BLE/PAL`, respectively. These PALs may be useful references for new HomeKit SDK implementations.

The POSIX SDK includes all open source dependencies needed to build and run the samples (makefiles are provided). In contrast, the BLE SDK is purely for demonstration purposes. The BLE chipset native SDK is not included, nor makefiles for building accessory logic.

To implement the PAL modules, use the `ADK/PAL/Templates` modules as starting points.

Start replacing the function bodies of the null function implementations with real code.

First, implement versioning, abort and logging functionality, so that useful debug information can be obtained during development. Then, all remaining relevant PAL modules could be implemented before trying to run HomeKit based on this new PAL. In practice, it has advantages to instead get as quickly as possible to a point where HomeKit runs end-to-end, i.e., the Home app on an iOS device actually works with the accessory - and only then complete the PAL module implementations. To that end, the ADK provides bootstrap implementations of some PAL modules that can be used as temporary implementations during development:

- Random number generator. The bootstrap implementation provides a pseudo-random generator which is not cryptographically secure.
- Key-value store. The bootstrap implementation works in main memory, so it is not persistent.
- Accessory setup. The bootstrap implementation uses an example setup code not to use in a licensed accessory.

Then build the system, using the `Lightbulb` accessory and the selected HAP Library binary (or binaries). The makefile from the POSIX SDK sample may be adapted for this purpose. The makefile can be structured so that it is easy to switch between bootstrap implementation (where available), template implementation, and the real implementation for the target platform.

Now start the HAP server with the `Lightbulb` sample. The log output will show fatal errors that indicate where in the PAL a null function is not yet implemented. Implement these functions until no fatal errors occur anymore. The functions can also be found by searching for the text pattern `[NYI]` in the source files (not all PAL modules need to be implemented for every HomeKit accessory though, see below).

Then replace the bootstrap implementations with real implementations - accessories must **not** be shipped with the bootstrap PAL module implementations.

Once the PAL is adapted and the operating system configured as necessary, the PAL can be tested using Apple's HAT and HCA HomeKit test tools.

4.2.5. Complete the accessory logic samples

In the ADK, the portable parts of accessory logic samples are provided in the `ADK/Applications` directory, e.g., a `Lightbulb` sample with its `DB.h`, `DB.c`, `App.h` and `App.c` files. For the SDK, add corresponding Main.c files that contain the code for platform initialization and for starting the accessory server. A complete lightbulb example with Main.c for POSIX is provided in `ADK/SDK-Samples/POSIX/Raspi/Applications/Lightbulb`, for the nRF52832 chipset is provided in `ADK/SDK-Samples/BLE/Applications/Lightbulb`. It is recommended to provide additional native samples that

demonstrate the use of native features of the platform, e.g., a `LightbulbLED` sample that actually switches an on-board LED of the platform's reference hardware board on and off, controlled via HomeKit. Ideally, these native samples could pass self-certification to make it as easy as possible for accessory makers to bring products to market.

4.2.6. Provide a secure firmware update mechanism

See section Versioning in chapter PAL Modules, and the HomeKit Accessory Protocol Specification for more details.

Refer to *HomeKit Accessory Protocol Specification Addendum R1.4* rule 18: *App must ensure that HomeKit accessories are not updatable to a firmware version that is incompatible with the iOS version of the controller.*

4.2.7. Check before release

Before releasing the PAL to production, ensure that:

- No bootstrap modules are used anymore, i.e.,
  - a cryptographically secure random number generator is used,
  - a persistent key-value store implementation is used,
  - an accessory setup implementation is provided that can be used by accessory manufacturers for their provisioning mechanisms.

If there appear to be issues with the used HAP Library binary, see Troubleshooting for more information on troubleshooting.

4.3. Memory Requirements

The following data is an estimate of the minimum HAP memory requirements currently recommended for a BLE SoC that implements the `Lightbulb` profile:

- Flash: 160 KB (for program code, Cortex-Mx code built with Keil toolchain, _min library variant)
- RAM: 25 KB (for data structures and stack)
- Secure storage: 4 KB of writable non-volatile storage (flash or EEPROM)

This assumes that code executes from flash, otherwise available RAM needs to be correspondingly larger. The code and data consumed by the BLE stack and the operating system, if any, are not included here.

For IP accessories (Wi-Fi or Ethernet), more resources are needed. The following data is an estimate of the minimum HAP memory requirements currently recommended for an IP accessory that implements the `Lightbulb` profile:

- Flash: 400 KB (for program code, ARMv6 code built with GCC toolchain)
- RAM: 60 KB (for data structures and stack)
- Secure storage: 4 KB of writable non-volatile storage (flash or EEPROM)

This assumes that code executes from flash, otherwise available RAM needs to be correspondingly larger. The code and data consumed by the IP stack and the operating system, if any, are not included here.

Updating persistent storage in a robust, transactional way may require some additional flash storage (e.g., an additional sector, depending on flash memory architecture). The firmware update mechanism for a platform may require a considerable amount of additional flash memory, especially if it completely retains the old firmware image while installing a new one.

Precise memory requirements also depend on the supported HomeKit profiles. Most HomeKit profiles are similar to the `Lightbulb` profile regarding the RAM requirements for their accessory logic: a few bytes of global state only. However, a few profiles require substantially more memory (e.g., door locks with logs, IP cameras and video doorbells, HomeKit bridges).

For IP accessories that need no audio/video support and are not bridges, communication buffers typically dominate the requirements for HomeKit-specific RAM. With ADK 1.2 and later, it is usually possible to work with the following rule of thumb for such accessories:

*min-RAM-size = max-number-of-parallel-connections * 6 KB rounded up to the next multiple of 10*

where *max-number-of-parallel-connections* is at least 9. This results in at least 60 KB. For HomeKit bridges, use at least twice as much memory per connection.

In the ADK, HAP Library binaries are provided that contain support for both HAP over IP and HAP over BLE, and produce extensive log output (if the `HAPPlatformLog` PAL module is implemented). For very resource-constrained accessories, minimized variants of the libraries are provided that do not contain this logging code and the associated strings. RAM use is not affected, but code size differs considerably: less than 160 KB program code for the minimized variant compared to more than 500 KB for the standard variant with logging (depending on the toolchain). Nevertheless, the standard variant can be useful during development and debugging if there exists a development system with sufficiently large program memory. Minimized variants of the libraries can be recognized by the `_min` in their names.

Note: Depending on the supported instruction set architectures and quality of the used compilers, the code size, and thus flash storage consumption, can differ substantially. Code size and performance can also depend on the specific version of a compiler being used.

Note: HAP Library never dynamically allocates memory itself, i.e., it does not call `malloc`. Instead, if dynamic memory allocation is used at all, this happens within the accessory logic or the PAL. In the sample BLE and POSIX SDKs, the accessory logic samples `Main.c` initialization sections statically allocate and pass the necessary memory for the BLE / IP buffers to the HAP Library, in their calls to `HAPAccessoryCreateServer`. If you want to use dynamic memory allocation, you can deallocate memory provided to HomeKit after `HAPAccessoryServerRelease` has been called.

4.4. Clib Dependencies

Platform and accessory developers may use any suitable clib, including modified clibs for resource-constrained accessories that leave out all functions that are not needed for their accessories. The HAP Library implementation does not explicitly call any clib functions. However, the compiler used for creating the HAP Library binary might have implicitly generated calls to clib functions, in particular `mem*` functions, and possibly floating-point related functions.

4.5. PAL Modules

The PAL API is designed to be modular. All modules are described in detail in the following sections. Here is an overview:

PAL module	Must be implemented for
Versioning	All accessories
Abort	All accessories
Log	All accessories
Random number generator	All accessories
Clock	All accessories
Timers	All accessories
Run loop control	All accessories
Key-value store	All accessories
Accessory setup	All accessories
Authentication	All accessories
TCP stream manager	IP accessories, i.e., Ethernet or Wi-Fi
Service discovery (Bonjour)	IP accessories, i.e., Ethernet or Wi-Fi
Wi-Fi manager (WAC2)	Wi-Fi accessories
Software access point (WAC2)	Wi-Fi accessories
BLE peripheral manager	BLE accessories
IP camera	Camera and video doorbell IP accessories
Audio	Camera and video doorbell IP accessories
Video	Camera and video doorbell IP accessories
Microphone	Remote accessories

Including IP-, BLE- and camera-specific functions, the PAL defines about 70 functions in total. A BLE accessory requires the implementation of less than 40 of these functions, a Wi-Fi IP camera less than 60 of these functions.

Accessory logic developers usually do not need the PAL API for their development, except for initialization purposes.

All PAL functions are available from the `ADK/PAL/include/HAPPlatform.h` file.

Some PAL module implementations are optional, e.g., the IP Manager module need only be implemented for IP accessories (Ethernet or Wi-Fi).

In terms of implementation, "optional" means that the PAL developer can simply use null implementations of those PAL modules that are not needed. They are provided in `ADK/PAL/Templates`. Their functions call `HAPFatalError()`, but if an optional PAL module is not needed, its functions are never called by the HAP Library. For example, only IP cameras and video doorbells need an actual implementation of the IP Camera PAL module.

The PAL module headers use some common data types, and there are some commonly useful helper functions defined in `ADK/PAL/include/HAPBase.h`. A PAL developer need not know the details of these definitions.

In the following detailed descriptions of PAL modules, a *default* implementation of a function means code that *may* be modified, a *template* implementation means code that *must* be replaced for production code unless the PAL module is optional. For example, functions that interface with a BLE stack must be implemented for BLE accessories, but their null implementations must be used for IP accessories, without modifications. A *bootstrap* implementation can be used during development but must not be used for a licensed accessory.

The modules are presented in the order in which it makes most sense to implement them.

4.5.1. Versioning

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatform.h`

Summary:

An accessory must be able to provide a version number of its HomeKit platform (i.e., its hardware / software / toolchain combination), plus a compatibility version number that indicates which HAP Library version it is compatible with. This allows the HAP Library to detect inadvertent use of an old PAL version.

Expected behavior:

The function `HAPPlatformGetCompatibilityVersion` must return the version number of the HAP Library with which the platform is compatible. If the PAL is correctly compiled against the corresponding ADK version's header files, the appropriate version number is used automatically, as defined by `HAP_PLATFORM_COMPATIBILITY_VERSION` in `HAPPlatform.h`.

Before a firmware update, if a newer ADK version is used, the PAL must be compiled and tested against its new PAL header files so that `HAPPlatformGetCompatibilityVersion` correctly reflects the HAP Library version in use in the accessory.

The function `HAPPlatformGetVersion` must return a string containing the current version of the accessory platform, e.g., following semantic versioning rules. After a firmware update, if the PAL is modified in any way, a new version number must be returned. For semantic versioning, use the following rules:

Major change: (breaking API changes, so that code changes in client code are required)

- Increment `HAP_COMPATIBLE_VERSION`
- Increment major version number, reset minor version and revision numbers

Minor change: (backwards compatible API changes, must work properly with recompile but without code changes)

- Increment `HAP_COMPATIBLE_VERSION`
- Increment minor version number, reset revision number

Revision change: (no API changes, must work properly with recompile but without code changes)

- Do not increment `HAP_COMPATIBLE_VERSION`
- Increment revision

Note: As the PAL API is defined by Apple, and the HAP Library is the main client of the PAL, this versioning scheme is mainly relevant for the parts of the PAL that are visible to the accessory logic, namely the platform-specific `+Init.h` files.

The function `HAPGetIdentification` must return a string that identifies the PAL. The format of the returned string is defined by the platform developer.

Implementations:

ADK/PAL/Templates/HAPPlatform.c:
The function HAPPlatformGetVersion exits with a fatal error.

ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatform.c:
The function HAPPlatformGetVersion returns the version of the Raspberry Pi PAL implementation.

ADK/SDK-Samples/BLE/PAL/nRF52832/HAPPlatform.c:
The function HAPPlatformGetVersion returns the version of the BLE PAL implementation.

4.5.2. Abort

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformAbort.h`

Summary:

There may be error situations from which the HAP Library cannot recover. This PAL module implements the accessory behavior when this occurs.

These are the two most common policies to handle a fatal error:

- Fail-restart system: The complete accessory is reset and rebooted. This is recommended for resource-constrained platforms and for accessories that only support HAP.
- Fail-restart subsystem: Only the HomeKit subsystem is reset and goes through a "cold start" with a complete reinitialization of its resources. This is recommended for accessories that run a full operating system with isolated processes (e.g., with address spaces protected from other processes) and run other protocols besides HAP.

In special cases, especially if a mechanical actuator of the accessory cannot be reliably brought back to a well-defined and safe position as part of the restart process, it may be best to notify the user or administrator of the accessory of the problem and then stop the complete accessory (fail-stop system) or the critical subsystem (fail-stop subsystem).

Expected behavior:

A single function must be provided by `HAPPlatformAbort`. This function implements the behavior of the accessory after the HAP Library has detected a fatal error from which it cannot recover; therefore, the function must not return.

Implementations:

`ADK/PAL/Templates/HAPPlatformAbort.c`:
Null implementation that executes an endless loop. Must be implemented for all accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformAbort.c`:
Default implementation that terminates the HAP server process with a non-zero exit code.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformAbort.c`:
Sample implementation that terminates the HAP server by using the ARM core's interrupt mechanism to perform a system reset.

4.5.3. Log

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformLog.h`

Summary:

The HAP Library is only provided in binary form, so it appears as a black box. To make its behavior observable requires logging support from the PAL. The logging concept is based on Apple's *Unified Logging*, see Apple Developer Logging Documentation.

Logging support is available from all HAP Library binaries, except for those whose names end in _min. The latter are provided for resource-constrained, microcontroller-based accessories, produce no log output, and are considerably smaller.

The HAP Library writes no security-critical information to the log (e.g., no key data in plain text).

Here is an example of a log output:

```
2017-09-07'T'17:08:20'Z'  Debug
[com.apple.mfi.HomeKit.HM.Server.AccessoryServer] socket:0xedf490:>

    0000 47455420 2f636861    GET /cha

    0008 72616374 65726973    racteris

    0010 74696373 3f69643d    tics?id=

    0018 312e3531 20485454    1.51 HTT

    0020 502f312e 310d0a48    P/1.1..H

    0028 6f73743a 2041636d    ost: Acm

    0030 655c3033 324c6967    e\032Lig

    0038 68746275 6c622e5f    htbulb._

    0040 6861702e 5f746370    hap._tcp

    0048 2e6c6f63 616c0d0a    .local..

    0050 0d0a                 ..

2017-09-07'T'17:08:20'Z'  Info  [com.apple.mfi.HomeKit.App.App]
lightbulb_on_read: false

2017-09-07'T'17:08:20'Z'  Debug
[com.apple.mfi.HomeKit.HM.Server.AccessoryServer] socket:0xedf490:<

    0000 48545450 2f312e31    HTTP/1.1

    0008 20323030 204f4b0d     200 OK.

    0010 0a436f6e 74656e74    .Content
```

```
0018 2d547970 653a2061    -Type: a
 0020 70706c69 63617469    pplicati
 0028 6f6e2f68 61702b6a    on/hap+j
 0030 736f6e0d 0a436f6e    son..Con
 0038 74656e74 2d4c656e    tent-Len
 0040 6774683a 2035340d    gth: 54.
 0048 0a0d0a7b 22636861    ...{"cha
 0050 72616374 65726973    racteris
 0058 74696373 223a5b7b    tics":[{
 0060 22616964 223a312c    "aid":1,
 0068 22696964 223a3531    "iid":51
 0070 2c227661 6c756522    ,"value"
 0078 3a66616c 73657d5d    :false}]
 0080 7d                   }
```

Expected behavior:

Two functions must be provided by the PAL:

`HAPPlatformLogIsEnabled` indicates whether a specific type of logging, such as `default, info`, `debug`, `error`, or `fault`, is enabled for a specific subsystem / category. This allows the logging for different subsystems / categories to be controlled individually. The `default`, `error` and `fault` level messages are always enabled. Log levels are documented for type `HAPLogType` in the header file `ADK/PAL/include/HAPLog.h`. Here a summary:

- Debug-level messages contain information that may be useful during developing, or for troubleshooting a specific problem.
- Info-level messages contain information that may be helpful, but isn't essential, for troubleshooting errors.
- Default-level messages contain information about things that may result in a failure.
- Error-level messages are intended for reporting component-level errors.
- Fault-level messages are intended for reporting system-level errors (more than one component).

The makefiles of the POSIX sample SDK set the log level used for an accessory:

`HAP_LOG_LEVEL=0`	No logs are displayed.
`HAP_LOG_LEVEL=1`	Default, Error and Fault-level logs are displayed.
`HAP_LOG_LEVEL=2`	Default, Error, Fault-level and Info logs are displayed.
`HAP_LOG_LEVEL=3`	Default, Error, Fault-level, Info and Debug logs are displayed.

`HAPPlatformLogCapture` should write a log message to the target platform. If no log output is desired, its function body should be made empty. Small SoCs typically do not have sufficient memory, RAM nor flash, to store log output. Thus log output on such chipsets is usually forwarded directly to a serial port, typically a UART port. To observe an accessory's behavior, a Mac or PC with a terminal program can be attached. On the smallest systems, there may not be sufficient flash memory available, for them the `_min` variants of the HAP Library binaries are provided that *emit no log output at all* and have considerably smaller code sizes.

Please ensure that complete logs can be produced before requesting technical support.

Warning: Make sure that for licensed accessories, log output is disabled (`HAP_LOG_LEVEL = 0`), or that implementation of `HAPPlatformLog` produces no log output.

Implementations:

`ADK/PAL/Templates/HAPPlatformLog.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformLog.c`:
Default implementation that mainly uses `fprintf` for creating a log message.

Normally, only the most important RTP-related information is logged: key frames, errors and statistics. For debugging camera-related issues, more extensive logging can be switched on by setting variable `logVideoRTP` in `HAPPlatformLog.c` to `true`. The format of these packet log lines is documented in the header of `HAPPlatformLog+Camera.c`. The main parts of the video packet log lines are:

```
[<id>] h264: <sequence number>, <time stamp>, <nri>, <NAL-type>
```

plus additional fields for fragmented and aggregated packets.

Similarly, there is a variable `logAudioRTP` in the same header file for the audio data stream that is also set to `false` by default. The format of the audio packet log lines is:

```
[<id>] audio: <sequence number>, <time stamp>
```

`ADK/SDK-Samples/BLE/PAL/HAPPlatformLog.c`:
Sample implementation that uses SEGGER RTT for writing a log message to a serial port.

4.5.4. Random number generator

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformRandomNumber.h`

Summary:

A HomeKit accessory uses random numbers for various functions (e.g., for creating long-term keys during pairing with an iPhone). Such random numbers **must be cryptographically secure**.

Expected behavior:

The functions `HAPPlatformRandomNumberFill` must be implemented to fill a buffer with random data. Use built-in hardware random number generators if available.

Implementations:

`ADK/PAL/Templates/HAPPlatformRandomNumber.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for licensed accessories.

`ADK/PAL/Bootstrap/HAPPlatformRandomNumber.c`:
Bootstrap implementation using a deterministic pseudo-random number generator. Must not be used for all accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformRandomNumber.c`:
Default implementation based on getrandom().

`ADK/SDK-Samples/BLE/PAL/HAPPlatformRandomNumber.c`:
Sample implementation based on a hardware true random number generator.

As an alternative to the Raspberry Pi implementation, the following implementation is provided:

`ADK/SDK-Samples/POSIX/SDK/Xtras/PAL/RandomNumber/POSIX/HAPPlatformRandomNumber.c`:
Default implementation based on `/dev/urandom`. Use it as a fallback solution for POSIX platforms that do not provide `getrandom()` support. Use this implementation for older Linux kernel or Clib versions.

4.5.5. Clock

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformClock.h`

Summary:

`HAPPlatformClock` together with `HAPPlatformTimer` are used for timing and timeouts in the HAP Library. In addition, this module may be used for creating time stamps for log output. It is not essential that the time be synchronized to universal (real) time, but it must be monotonic (never jumping backwards).

Expected behavior:

If the hardware provides a real-time clock, use this RTC for implementing the module. Otherwise, use a timer to indicate the time that has passed since the last reboot.

If the hardware clock can be set from the outside (e.g., by the user), the PAL must compensate for the clock being set to an earlier time, to ensure that it is monotonically increasing.

The function `HAPPlatformClockGetCurrent` must return the time in milliseconds relative to an implementation-defined time in the past.

Implementations:

`ADK/PAL/Templates/HAPPlatformClock.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformClock.c`:
Default implementation based on `gettimeofday`.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformClock.c`:
Sample implementation based on a hardware timer/counter.

4.5.6. Timers

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformTimer.h`

Summary:

A HomeKit implementation needs multiple timers for various internal actions and timeouts.

Expected behavior:

Typically, only one hardware timer is used to implement a flexible number of timers in software.

The function `HAPPlatformTimerRegister` must add a new timer with a callback function and a deadline after which the callback function is called. The deadline is given in milliseconds as absolute time after an implementation-defined time in the past (must be the same as for `HAPPlatformClock`, see above). Whenever a timer has expired, i.e., its deadline is less than or equal to "now", its callback is invoked as soon as possible.

Timers must fire in ascending order of their deadlines and timers registered with the same deadline must fire in order of registration. Callbacks must be synchronized with (i.e., run on the same execution context as) the run loop (see `HAPPlatformRunLoop`). It is permissible to start a timer with `deadline == 0`, meaning that its handler should be called as soon as possible.

Implementations:

`ADK/PAL/Templates/HAPPlatformTimer.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories.

`ADK/SDK-Samples/POSIX/PAL/Raspi/HAPPlatformTimer.c`:
Default implementation of a dynamic list of timers based on the `HAPPlatformClock` module (see above).

`ADK/SDK-Samples/BLE/PAL/HAPPlatformTimer.c`:
Sample implementation of an array of timers based on a hardware timer.

4.5.7. Run loop control

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformRunLoop.h`

Summary:

The ADK runs in one loop, here called the *run loop*, and is responsible for managing and dispatching I/O events from all attached event sources, and it also drives the scheduler of timer events. In addition, `HAPPlatformRunLoop` provides a way for scheduling callbacks that will be called from the run loop. It must be safe to call this function from execution contexts other than the main loop, e.g., from another thread, from a signal handler, or an interrupt handler. It is the only synchronization mechanism of the ADK. The complete remaining ADK functionality executes in a single-threaded way from the run loop.

Callbacks are blocking, meaning it is the accessory logic's responsibility to immediately process them without waiting. This is possible because HAP is essentially a protocol for the remote reading and writing of variables (characteristics), which should not involve lengthy processing on the accessory. I/O in the PAL is generally non-blocking.

Expected behavior:

The following functions must be implemented:

`HAPPlatformRunLoopRun`

`HAPPlatformRunLoopStop`

`HAPPlatformRunLoopScheduleCallback`

Function `HAPPlatformRunLoopScheduleCallback` must be called by the accessory logic in order to schedule a callback on the run loop execution context *from any other execution context*. To minimize resource consumption, it should not be called from code that already executes on the run loop execution context.

A HomeKit controller may send a write request for multiple characteristics, e.g., requesting to turn on a light bulb, at 50% intensity, with blue color. When such a request has been received, all handlers for these characteristics must be called in an uninterrupted sequence, i.e., if there are pending scheduled callbacks or timer callbacks they must be delayed accordingly.

Implementations:

`ADK/PAL/Templates/HAPPlatformRunLoop.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories.

`ADK/SDK-Samples/POSIX/PAL/Raspi/HAPPlatformRunLoop.c`:
Default main loop implementation for POSIX that handles file descriptor events for service discovery and the HAP accessory server. This module also drives the timer scheduler `HAPPlatformTimer`. Callbacks from other execution contexts are synchronized via an internal pipe, a mechanism also known as a 'self-pipe'.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformRunLoop.c`:
Sample run loop implementation that handles BLE events, schedules timers, and synchronizes callbacks via event queues.

4.5.8. Key-value store

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformKeyValueStore.h`

Summary:

HAP expects to have a few KB of mutable persistent memory for its keys and other information. The data is stored as a set of key-value pairs.

Expected behavior:

Several instances of a key-value store must be supported (domains), so that information owned by the accessory, by the SDK provider and by the HAP implementation can be separately managed. The following domains are defined:

- 0x00–0x3F is reserved for accessory developers
- 0x40-0x7F is reserved for platform developers
- 0x80-0xFF is reserved for the HAP Library

In every domain, keys in the range 0x00 to 0xFF can be used.

Note that the function HAPRestoreFactorySettings only touches domains 0x80 through 0xFF. This means that provisioning information that is stored in the key-value store (domain 0x40 in the POSIX and BLE sample PALs) survives such a factory reset as required by the HAP specification. However, it is up to the platform developer where the provisioning information is stored, it does not have to be in the key-value store. Adjust HAPPlatformAccessorySetup if you want to use another backing store for provisioning.

Storage of key-value pairs is possible in files or directly on a micro-controller on-chip flash or EEPROM memory. A flash-based implementation of the key-value store may require up to twice as much flash memory compared to what HAP may require. The additional flash memory may be required e.g. because a copy of a flash sector has to be written before deleting the old sector.

The maximum theoretical possible capacity of the key-value store is 256 * 256 key-value pairs. The HAP Library owns 128 * 256 of these pairs, the PAL owns 64 * 256 pairs, and the accessory logic also owns 64 * 256 pairs. Note that PAL and accessory logic are not required to use the parts of the key-value store that are defined for them. These parts are merely defined for convenience, and also used in this way in the ADK reference implementations.

In practice, only few pairs are used (sparse structure). Actual use depends on parameters like the number of pairings, the given accessory attribute database, IP vs BLE capabilities, etc. It is expected that a total size of 4 KB is sufficient if the accessory logic does not use the key-value store.

Values have different sizes depending on the key; the ones used by HAP Library stay within reasonable bounds (up to 128 bytes for the longest). In the PAL, there are larger values, for example to store SRP salt and verifier. But PAL code is provided in source form, so these values can be split across multiple keys if necessary.

About half a dozen functions must be implemented.

Implementations:

`ADK/PAL/Templates/HAPPlatformKeyValueStore.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories.

`ADK/PAL/Bootstrap/HAPPlatformKeyValueStore.c`:
Bootstrap implementation that implements a KVS in main memory (no persistence). Must not be used in a licensed accessory.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformKeyValueStore.c`:
Default implementation based on the Linux file system. The key-value store is backed by the .`HomeKitStore` directory, with file names denoting <Domain>.<Key>, for example 90.10 for domain 0x90 and key 0x10. Some of the (POSIX-specific) uses are:

- File `.HomeKitStore/40.10` contains SRP salt and verifier
- File `.HomeKitStore/40.11` contains setup ID

`ADK/SDK-Samples/BLE/PAL/HAPPlatformKeyValueStore.c`:
Sample implementation based on flash memory.

4.5.9. Accessory setup

Required:	For every accessory
Header file:	`ADK/PAL/include/HAPPlatformAccessorySetup.h`

Summary:

During pairing, a setup code or setup ID is used to ensure that the end user of the accessory does the pairing, and not a "man in the middle" attacker. The HAP Library requires a way to access this information. Accessories with displays must display setup codes that are generated dynamically by the HAP Library. Accessories with programmable NFC must be able to update the data that they provide, as requested by the HAP Library.

HAPPlatformAccessorySetupUpdateSetupPayload is called when you should update your display. This function is not called in response to pairing status changes: if you need to be notified about pairing status changes, use the callback handleStateUpdated in HAPAccessoryServerOptions (see HAP.h for more information).

Expected behavior:

As with the key-value store (see Key-value store), the setup data may be stored in a file, directly in flash, or in EEPROM chips. The key-value store may be used for this purpose.

About half a dozen functions must be implemented.

Implementations:

`ADK/PAL/Templates/HAPPlatformAccessorySetup.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories.

`ADK/PAL/Bootstrap/HAPPlatformAccessorySetup.c`:
Bootstrap implementation that provides setup information based on an example setup code. Must not be used for licensed accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformAccessorySetup.c`:
Same as the bootstrap implementation. Must not be used for licensed accessories.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformAccessorySetup.c`:
Sample implementation that supports static, dynamic and NFC setup codes.

4.5.10. Hardware-based Authentication (Apple Authentication Coprocessor)

Required:	For every accessory (if an Apple Authentication Coprocessor is attached)
Header file:	`ADK/PAL/include/HAPPlatformMFiHWAuth.h`

Summary:

For authentication, byte blocks from a HomeKit controller are forwarded to the Apple Authentication Coprocessor via its I2C interface, and vice versa. The platform developer does not need to understand the format or meaning of the data being forwarded.

Expected behavior:

Apart from a few functions for enabling/disabling the Apple Authentication Coprocessor, two functions must be implemented: one for writing and one for reading a byte block over I2C.

Apple Authentication Coprocessors 2.0C and 3.0 are supported by the HAP Library.

Implementations:

`ADK/PAL/Templates/HAPPlatformMFiHWAuth.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories that use hardware authentication.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformMFiHWAuth.c`: Default implementation based on the `/dev/i2c` interface.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformMFiHWAuth.c`:
Sample implementation based on the I2C interface.

4.5.11. Software Authentication

Required:	For every accessory (if no Apple Authentication Coprocessor is attached - this feature is only available in ADK 1.1 and later and requires iOS 11.3 or later)
Header file:	`ADK/PAL/include/HAPPlatformMFiTokenAuth.h`

Summary:

For software authentication, a token is provisioned in the accessory either at the factory during the time of manufacturing or via firmware upgrade for accessories that are already shipping and do not yet support HomeKit. When a controller adds an accessory to HomeKit, it requests the token from the accessory during the Pair Setup process.

Expected behavior:

A token must be stored persistently, e.g., using a key-value store. Writing and updating a token must happen atomically (i.e., a power failure during a token update must leave the previous token intact). If the token update is asynchronous, communication via BLE / IP must be suspended until the token update is complete. The token can only be used once to activate an accessory. To reactivate the accessory (e.g., due to a factory reset), a new token will be issued by the controller that needs to be provisioned on the accessory and used for the next activation.

Please refer to the `Provision` script in `ADK/SDK-Samples/POSIX/Raspi/Tools` for correct encoding of the token UUID and token blob.

> *Note: The `--mfi-token <Software Token UUID> <Software Token>` allows the provisioning of information required for Software Authentication. The `<Software Token UUID>` and initial `<Software Token>` are deployed to the key-value store.*
>
> *The `--mfi-token` preserve option needs to be used to preserve a token between tests, otherwise it is deleted / overwritten.*
>
> *Note: Within the ADK, the encoding of software token UUIDs (type `HAPPlatformMFiTokenAuthUUID`) is inverted, i.e., network byte order backwards.*

Implementations:

`ADK/PAL/Templates/HAPPlatformMFiTokenAuth.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for all accessories that use software authentication.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformMFiTokenAuth.c`: Default implementation based on the key-value store.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformMFiTokenAuth.c`:
Default implementation based on the key-value store.

4.5.12. TCP stream manager

Required:	For every IP accessory (Ethernet, Wi-Fi)
Header file:	`ADK/PAL/include/HAPPlatformTCPStreamManager.h`

Summary:

An IP stack that supports several TCP connections is the basis for all HAP over IP accessories. The HAP server uses a stream abstraction for its IP connections. The PAL implementation must wrap the IP stack functionality for open, close, read, and write.

Expected behavior:

An IP protocol stack must meet the following requirements:

- A POSIX dual IPv4 / IPv6 stack based on Berkeley sockets, also known as BSD sockets
- Concurrent IPv4 and IPv6 connections must be supported
- At least 8 concurrent open connections must be supported

To interface with the IP stack, about 20 functions must be implemented, for managing TCP stream listeners and TCP streams, and for reading and writing on TCP streams.

Implementations:

`ADK/PAL/Templates/HAPPlatformTCPStreamManager.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for IP accessories, must be used as is for BLE accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformTCPStreamManager.c`:
Default implementation for POSIX network interface. On Linux, the built-in IP stack is used. Most current Linux distributions meet the HomeKit requirements.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformTCPStreamManager.c`:
Same as the null implementation, must be used as is for BLE accessories.

4.5.13. Service discovery (Bonjour)

Required:	For every IP accessory (Ethernet, Wi-Fi)
Header file:	`ADK/PAL/include/HAPPlatformServiceDiscovery.h`

Summary:

HomeKit uses Apple's Bonjour protocol for finding accessories in a local IP network. Bonjour consists of the mDNS protocol plus additional behavior that is usually associated with ZeroConfig.

Expected behavior:

A Bonjour implementation must meet the following requirements:

- `mDNS` must be supported
- The implementation must pass Apple's Bonjour Conformance Test (BCT)

Implementation:

The following functions must be implemented:

`HAPPlatformServiceDiscoveryRegister`

`HAPPlatformServiceDiscoveryUpdate`

`HAPPlatformServiceDiscoveryStop`

Implementations:

`ADK/PAL/Templates/HAPPlatformServiceDiscovery.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for IP accessories, must be used as is for BLE accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformServiceDiscovery.c`:
Default implementation that requires installation of Apple's mDNSResponder (open source). The Unix command to check that `mDNSResponder` is working is

```
ps aux | grep [m]dns
```

If the command does not return any output, `mDNSResponder` should be installed and its service started.

`mDNSResponder` only implements the higher layers of Bonjour, it does not cover some ARP-level functionality (`ZeroConfig`) that is required in order to pass the Bonjour Conformance Test.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformServiceDiscovery.c`:
Same as the null implementation, must be used as is for BLE accessories.

As an alternative to the `mDNSResponder` implementation, the following implementation is provided:

`ADK/SDK-Samples/POSIX/Xtras/PAL/Service/Discovery/Avahi/HAPPlatformServiceDiscovery.c`:

Default implementation that requires installation of Avahi (open source). Avahi must be enabled as the active mDNS service. The Unix command to check that `Avahi` is working is

```
ps aux | grep [a]vahi
```

If the command does not return any output, `Avahi` should be installed and its service started. For `Avahi` it is also necessary to change the build command to

```
make SDLIB=Avahi
```

instead of

```
make SDLIB=Apple
```

4.5.14. Wi-Fi manager

Required:	For every Wi-Fi accessory
Header file:	`ADK/PAL/include/HAPPlatformWiFiManager.h`

Summary:

Many HomeKit IP accessories are based on Wi-Fi and need some specific support for taking part in Wi-Fi networks.

Expected behavior:

Four functions need to be implemented, to enable a Wi-Fi accessory to join and to leave a Wi-Fi network.

Implementations:

`ADK/PAL/Templates/HAPPlatformWiFi.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for Wi-Fi accessories, must be used as is for Ethernet and BLE accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformWiFi.c`:
Default implementation for Linux. Check whether the dependencies documented in this file are working on the target platform. Check whether this code needs adaptations to the target platform's Wi-Fi hardware. Check whether the underlying Linux platform needs modifications, e.g., at the time of this writing, a Linux kernel driver patch is necessary to make the code work on the Raspberry Pi 3 reference platform.

`ADK/SDK-Samples/BLE/PAL/Raspi/WiFi/HAPPlatformWiFi.c`:
Same as the null implementation, must be used as is for BLE accessories.

4.5.15. Software access point (used for legacy WAC/WAC2)

Required:	For every Wi-Fi accessory
Header file:	`ADK/PAL/include/HAPPlatformSoftwareAccessPoint.h`

Summary:

HomeKit uses the WAC2 protocol to make pairing of an accessory easy for the user (no manual entry of network credentials). WAC itself is already implemented within the HAP Library, but some functionality of the Wi-Fi hardware must be used to enable it.

Expected behavior:

A Wi-Fi accessory must meet the following requirements for WAC2:

- Wi-Fi hardware must support Soft Access Point mode and Station mode
- It must be possible to programmatically switch between Soft Access Point mode and Station mode without rebooting the device
- It must be possible to broadcast custom IE frames
- It must be possible to join a Wi-Fi network programmatically with a given SSID and pass phrase
- The Soft Access Point must run as an unsecured network, i.e., without a password
- The Soft Access Point must support Berkeley sockets (also known as BSD sockets)
- The Soft Access Point network interface must support Bonjour and TCP socket listener
- The Soft Access Point must provide a DHCP server

The following functions must be implemented:

`HAPPlatformSoftwareAccessPointStart`

`HAPPlatformSoftwareAccessPointStop`

Note: ADK 1.1 implements WAC2. WAC2 is only compatible with iOS 11.3 or later. A software update that upgrades from ADK 1.0 to ADK 1.1 automatically upgrades the target accessories from legacy WAC to WAC2, without changes to PAL or accessory logic code.

Implementations:

`ADK/PAL/Templates/HAPPlatformSoftwareAccessPoint.c:`
Null implementation where all functions exit with fatal errors. Must be implemented for Wi-Fi accessories, must be used as is for Ethernet and BLE accessories.

`ADK/SDK–Samples/POSIX/Raspi/PAL/HAPPlatformSoftwareAccessPoint.c`:
Default implementation for Linux. Check whether the dependencies documented in this file are working on the target platform. Check whether this code needs adaptations to the target platform's Wi-Fi hardware. Even the underlying Linux platform may need modifications, e.g., at the time of this writing, a Linux kernel driver patch is necessary to make the code work on the Raspberry Pi 3 reference platform.

`ADK/SDK–Samples/BLE/PAL/HAPPlatformSoftwareAccessPoint.c:`
Same as the null implementation, must be used as is for BLE accessories.

4.5.16. BLE peripheral manager

Required:	For every BLE accessory
Header file:	`ADK/PAL/include/HAPPlatformBLEPeripheralManager.h`

Summary:

A BLE accessory comes with a BLE stack, which is interfaced through this PAL module. The PAL implementation must wrap the BLE stack functionality for advertising, services, characteristics, and read/write events.

Expected behavior:

The requirements for a BLE stack are as follows:

- The BLE stack must allow the HAP Library to handle read and write requests directly. This is required for HomeKit session security. For read requests, the response value and length are dynamically computed on a per-request basis. For write requests, the application must be able to decide whether an error response or a write response is returned. Therefore, the stack must not automatically respond with cached values.
- The BLE stack must allow delaying the response to a read or write request by several seconds.
- It must be possible to disconnect from the central without responding to a read or write request. It must be possible to immediately disconnect after a read response has been transmitted.
- It is recommended that the GATT database size be configurable by the application and that characteristic values be stored in application memory. It is also recommended to store 128-bit UUIDs in a compact format. The same descriptor UUIDs are used for each characteristic and Apple-defined characteristics use 128-bit UUIDs based on a common base UUID (similar to the way 16-bit UUIDs assigned by the Bluetooth SIG operate).

Implementation:

To interface with the BLE stack, about a dozen functions must be implemented, for managing device addresses and names, advertisements and connections, and for exposing services, characteristics and descriptors.

Implementations:

`ADK/PAL/Samples/HAPPlatformBLEPeripheralManager.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for BLE accessories, must be used as is for IP accessories.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformBLEPeripheralManager.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for BLE accessories, must be used as is for IP accessories.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformBLEPeripheralManager.c`:
Sample implementation for a BLE chipset.

4.5.17. IP Camera

Required:	For every IP camera or video doorbell accessory
Header file:	`ADK/PAL/include/HAPPlatformIPCamera.h`

Summary:

Cameras and video doorbells are special in that the HomeKit Accessory Protocol is used for setting up the video (and audio) streams for such an accessory, but the actual communication is done via standard protocols like SRTP and using standard data formats like H.264. In essence, a HomeKit service is used to configure, start and stop SRTP streams that are otherwise completely independent of HomeKit.

Therefore, supporting IP camera functionality in a HomeKit accessory consists of two tasks:

1. Handle the HomeKit part of the communication, e.g., to obtain information from the HomeKit controller about the desired video resolution for the video stream. This task mainly requires HomeKit know-how. A complete implementation is provided in `ADK/SDK-Samples/POSIX/Raspi/Applications/IPCamera`. Change its configuration part to reflect the properties of the camera accessory.

2. Implement what is called the *A/V pipeline*, which for video means capturing video frames from the camera, encoding them into H.264, and streaming the result using the SRTP protocol via a UDP socket to the HomeKit controller. For audio it means capturing audio data from the microphone, encoding it using one of the permitted audio formats (Opus or AA-ELD), and streaming it using the SRTP protocol via another UDP socket to the HomeKit controller. In addition, there is an audio pipeline for decoding received audio data and forwarding it to a speaker. This task mainly requires A/V know-how, experience with the special video processors that are typically used, and possibly experience with a software library such as `GStreamer` (https://gstreamer.freedesktop.org) which implements many of the protocol elements needed for implementing the A/V pipeline. The result is an implementation of the `HAPPlatformIPCamera` API, plus the setup of whatever additional system software is needed, e.g., GStreamer and its plugins.

Requirements:

IP cameras have special requirements. Of the HomeKit profiles defined so far, they require the largest amounts of RAM by far. They also require considerable processing power, for this reason at least two processor cores, at least 500 MHz clock speed, and support from a dedicated GPU are recommended.

The hardware must support the following video requirements:

- Codec: H.264, main profile, level 4
- Resolution: 1920x1080, 1280x720, 640x360, 480x270, 320x180, 1280x960, 1024x768, 640x480, 480x360, 320x240
- Support for at least two parallel streams: one must support at least 1080p@30, the others at least 720p@30.
- In addition, support for a snapshot stream
- On-the-fly reconfiguration of the video streams (resolution and bitrate)

The hardware must support the following audio requirements:

- Codec: AAC-ELD or Opus, variable bit rate mode
- Sample rate: 16 k or 24 k samples per second
- The RTP timestamp frequency must be equal to the sample frequency, even for Opus

The chipset-specific drivers must also support these requirements. Typically, the chipset vendor needs to supply software that acquires the raw data from camera and microphone and encodes it into the standard formats mentioned above. This task is performance-critical. Even if a processor with its GPU may have enough raw horsepower to meet all HomeKit requirements, the available software may not be sufficient. For example, on a Raspberry Pi 3, the programs `raspivid` and `raspistill` appear to monopolize the GPU and only support one stream at the time.

For more information, see the HomeKit Accessory Protocol Specification.

Note: H.264 and AAC-ELD require licenses, which may be included with processors that provide special hardware support for these standards. It is the licensee's responsibility to ensure that valid licenses are available.

Once the incoming data is in the correct format, in can be forwarded to another component that performs streaming over IP. This software must meet the following requirements:

- Protocol: SRTP with AES_CM_128_HMAC_SHA1_80 and AES_256_CM_HMAC_SHA1_80
- RTP and RTCP packets multiplexed on same port
- separate ports for audio and video
- RTCP must support extended feedback and codec control including PLI, FIR, TMMB and TST messages

HAP-RTP:

The ADK provides an implementation of RTP in the HAP Library, called HAP-RTP. Apple's HAP-RTP is a stable, efficient, and light-weight protocol implementation of RTP, RTCP and SRTP that meets the HomeKit specifications.Technically, this implementation is shipped as part of the HAP Library. Logically, it is independent of HomeKit and may or may not be used in a PAL. Its purpose is to speed up development of HomeKit-compliant IP cameras even if an accessory manufacturer does not yet have access to a HomeKit-compliant RTP implementation.

Implementation approaches:

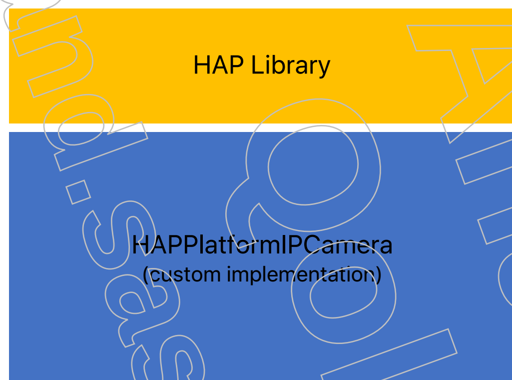
There are several ways how the `HAPPlatformIPCamera` PAL module can be implemented. They differ in whether HAP-RTP is used or not, and whether GStreamer functionality is used (and how much of it).

The following combinations are possible:

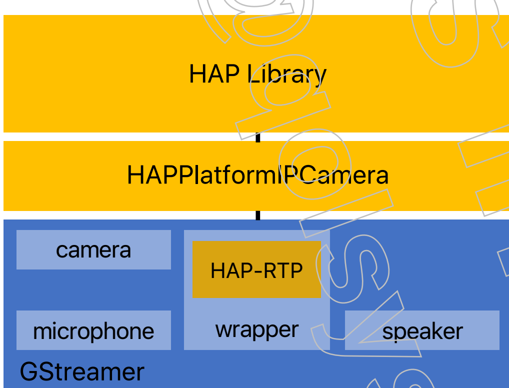
- With HAP-RTP, with GStreamer
- With HAP-RTP, without GStreamer
- Without HAP-RTP, with GStreamer
- Without HAP-RTP, without GStreamer

In all cases, HomeKit support relies on the small number of streaming operations in `HAPPlatformIPCamera.h` that must be implemented in some way. Here the most important approaches, one of them directly supported by sample code in the ADK:

1. Custom implementation: Start with the stub implementation described further below. Any hardware and/or software implementation of the A/V pipeline that meets the HomeKit specification may be used. In this approach, HAP-RTP is not used, so RTP, RTCP and SRTP code must be provided by the platform developer. GStreamer use is optional.



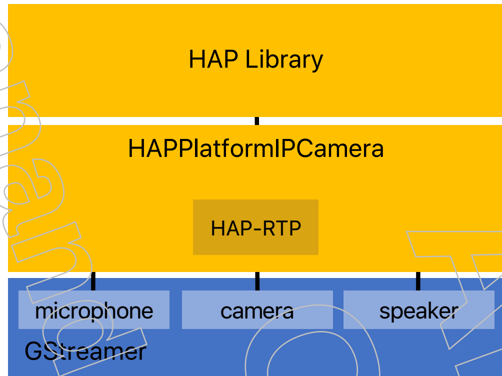
2. Wrapper implementation: Wrap the HAP-RTP software component of the ADK into a GStreamer plugin, so that the A/V pipeline can be fully implemented in GStreamer. The PAL module doesn't directly interact with HAP-RTP, only with GStreamer and its plugins. Take this approach if you have experience with GStreamer and want to work as much as possible within its framework but still use Apple's HAP-RTP code.



Note: HAP-RTP is a separate component as depicted here, but shipped as part of the normal HAP Library file.

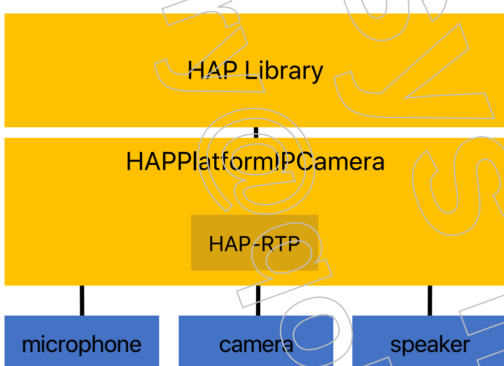
3. Modular implementation: Start with the full, working implementation for the Raspberry Pi, described further below. This implementation of the `HAPPlatformIPCamera` API directly uses HAP-RTP (not integrated into GStreamer) and introduces a new API for accessing cameras: `HAPPlatformIPCamera.h`. The implementations of video capture and encoding for the camera, audio capture and encoding for the microphone, and audio decoding for the speaker are done using GStreamer. This is the current default implementation, as provided in the POSIX PAL in file `HAPPlatformIPCamera.c`.
Since ADK 2.0, `HAPPlatformIPCamera.c` has become largely portable, by using the newly

introduced PAL module APIs HAPPlatformAudio.h and `HAPPlatformVideo.h`. Only these two APIs must be implemented in order to get HomeKit-compatible IP Camera functionality.



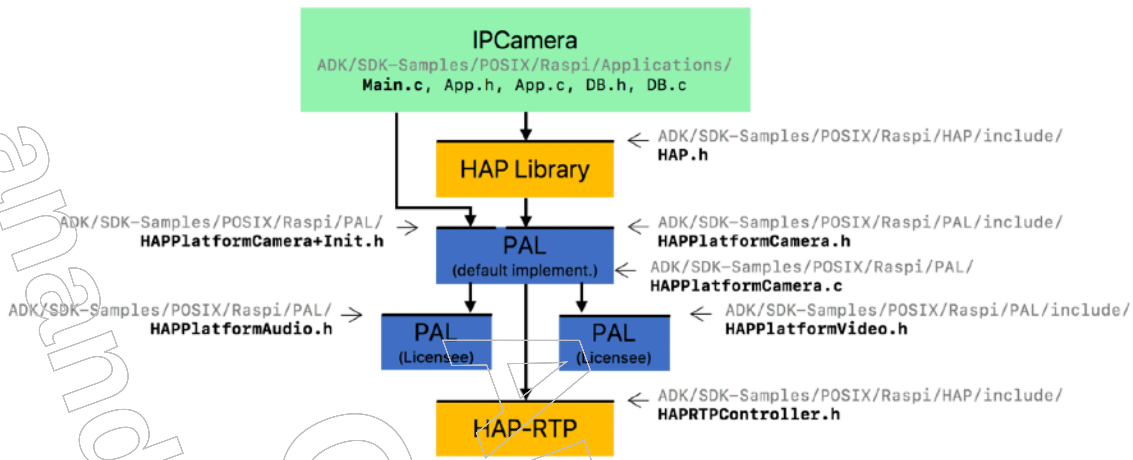
Take this approach to minimize the use of GStreamer. Also start with this approach to work without using GStreamer at all, because it requires only replacing these simpler GStreamer plugins, rather than the more complex GStreamer pipelines of approach 2. After eliminating the GStreamer dependencies, it results in a minimal implementation like the following:

4. Minimal implementation: In this approach, there are no dependencies on GStreamer.



Dependencies of the modular implementation approach:

The IP Camera sample code for the Raspberry Pi implements approach 3 as described above. For this approach, the following diagram shows the relationship of the camera-related files; an arrow denotes a *uses* relation:



- The IPCamera sample files (`DB.h`, `DB.c`, `App.h`, `App.c`, `Main.c`) are only provided in the POSIX sample SDK, not in `ADK/Applications`, as there exists no portable, generic implementation of the IP camera profile that would work with a HomeKit controller. The sample code, shown in green, contains the HomeKit integration code of the camera accessory.

- `Main.c` uses the `HAPPlatformCamera+Init.h` interface to initialize the PAL. This interface can be defined by the platform developer as needed for the specific PAL implementation.

- The other interface of the camera PAL module, `HAPPlatformCamera.h`, must be implemented as defined by the ADK. It controls the A/V streams of the accessory. It is recommended to use the default implementation HAPPlatformCamera.c, which typically needs minimal changes for target platforms that support Berkeley sockets. If this default implementation is used, it means that you only need to implement the two auxiliary PAL modules HAPPlatformAudio and HAPPlatformVideo.

- If (and only if) the platform developer wants to take advantage of HAP-RTP, then the PAL needs to use the `HAPRTPController.h` interface. In this implementation approach, the use of HAP-RTP is encapsulated in the file `HAPPlatformCamera.c`.

- Not shown in the diagram: during development, the helper files `HAPPlatformLog+Camera.h/.c` are useful for diagnosing the data being streamed by HAP-RTP. For initialization purposes, there are the files `HAPPlatformAudio+Internal.h` and `HAPPlatformVideo+Internal.h`. The *uses* relation here is a bit indirect:
  `HAPPlatformCamera.c > HAPPlatformCamera+Init.h >`
  `HAPPlatformAudio+Internal.h > HAPPlatformAudio.h`
  and analogous for video streams.

Implementations:

`ADK/PAL/Templates/HAPPlatformIPCamera.c`:
Null implementation where all functions exit with fatal errors. Must be implemented for IP cameras and video doorbells, must be used as is for all other types of accessories.

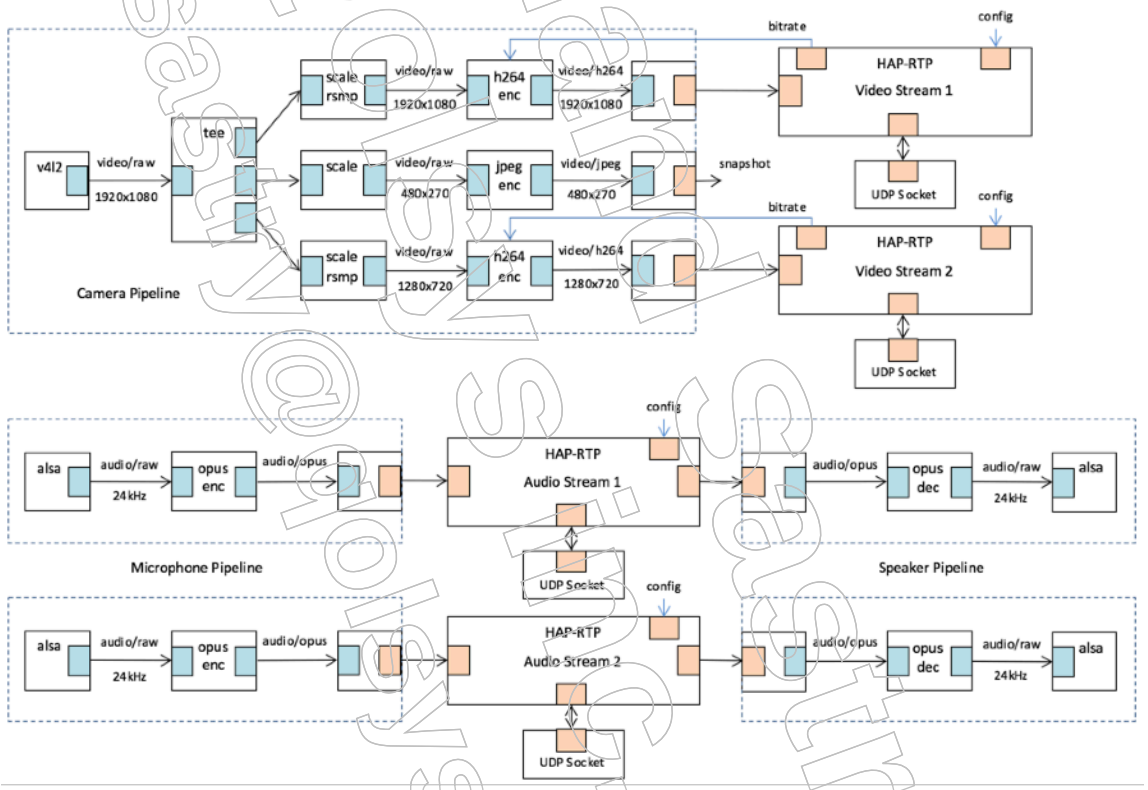
`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformIPCamera.c`:
Default implementation for Raspberry Pi 3 that implements approach 3. using GStreamer as described above. Make sure that GStreamer and compatible camera, microphone and speaker plug-ins for GStreamer are installed.

`ADK/SDK-Samples/BLE/PAL/HAPPlatformIPCamera.c`:
Same as the null implementation. Must be used as is for BLE accessories.

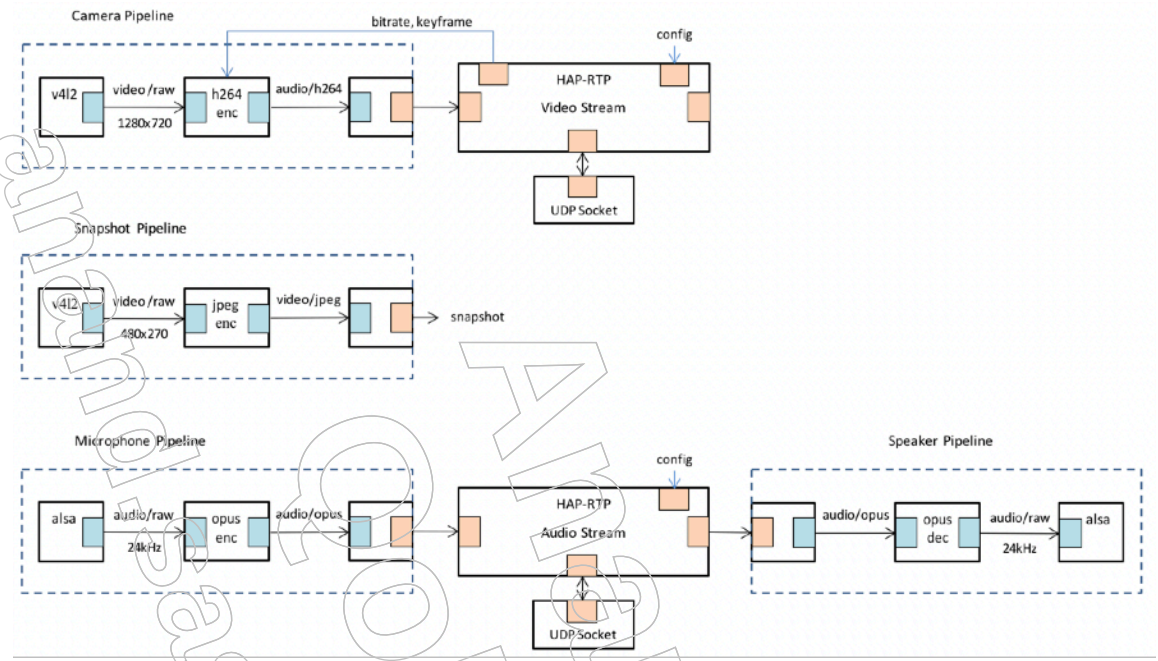
To eliminate all GStreamer dependencies, modify this implementation as needed for the A/V libraries. To support dynamic changes to the volume and mute characteristics (which is optional according to the HomeKit specification), then also modify `ADK/SDK-Samples/POSIX/Raspi/Applications/IPCamera/App.c` accordingly.

As a peculiarity of this implementation, the video stream is not directly encoded on the GPU of the Raspberry Pi's microprocessor. Instead, it is sent via the GPU to the CPU, and back again for H.264 encoding. This detour is due to a limitation of the available GStreamer elements, in order to enable more than one video stream at the same time.

The implementation supports the HomeKit requirements for at least two A/V streams plus a stream of stills images (snapshots). The following diagram shows a complete configuration:



Note: In the current Raspberry Pi reference platform implementation, only one A/V stream is supported due to limitations of the used GStreamer components and drivers. Apple is planning on removing this limitation in a later release of the ADK. So the current implementation on Raspberry Pi actually implements this A/V pipeline:



The boxes with dotted outlines indicate GStreamer pipelines with their plugins. The other boxes are part of HAP-RTP.

- The `v4l2src` GStreamer plug-in uses the Video for Linux driver to automatically detect an attached camera. It continuously reads a stream of video images and forwards the stream to other plug-ins in the camera pipeline.
- The `alsasrc` GStreamer plug-in uses the Advanced Linux Sound Architecture driver to automatically detect an attached microphone. It continuously reads a stream of audio samples and forwards the stream to other plug-ins in the microphone pipeline.
- The `alsasink` GStreamer plug-in uses the Advanced Linux Sound Architecture driver to automatically detect an attached speaker. It receives a stream of audio samples from other plug-ins in the speaker pipeline and sends the stream to the speaker.

For more information on GStreamer, see the API manual at https://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/manual.pdf. For more information on how to set up a Raspberry Pi as an IP Camera demonstrator, see chapter Raspberry Pi as IP Camera Demonstrator.

4.5.18. Audio

Required:	For every IP camera or video doorbell accessory that uses the default camera PAL implementation (`HAPPlatformCamera.c`)
Header file:	`ADK/SDK-Samples/POSIX/Raspi/PAL/include/HAPPlatformAudio.h`

Summary:

Audio stream abstraction. It is only needed if the default camera PAL implementation of Apple is used.

For more information, see IP Camera above.

Implementations:

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformAudio.c`:
Implementation that uses GStreamer and ALSA.

4.5.19. Video

Required:	For every IP camera or video doorbell accessory that uses the default camera PAL implementation (`HAPPlatformCamera.c`)
Header file:	`ADK/SDK-Samples/POSIX/Raspi/PAL/include/HAPPlatformVideo.h`

Summary:

Video stream abstraction. It is only needed if the default camera PAL implementation of Apple is used.

For more information, see IP Camera above.

Implementations:

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformVideo.c`:
Implementation that uses GStreamer.

4.5.20. Microphone

Required:	For every `Remote` accessory
Header file:	`ADK/SDK-Samples/POSIX/Raspi/PAL/include/HAPPlatformMicrophone.h`

Summary:

Microphone abstraction. Similar to the microphone-related part of `HAPPlatformAudio.h`.

This PAL module delivers sound data from a microphone by invoking `HAPPlatformMicrophoneDataCallback`. Along with the Opus-encoded sound samples that you provide through the arguments `bytes`, `numBytes` and `sampleTime` of this function, you must also provide a root mean square value in argument `rms`. It is an average volume computed like this:

```
rms = sqrt( sum( i = 0 to n-1, s[i]^2 ) / n )
```

where `s[i]` are the individual *raw* audio samples and `n` is the number of audio samples provided in this callback.

The resulting value, which you then pass to the `rms` argument, must be in the range of 0 to 1 for all representable audio signals. The necessary scaling can be done by scaling the raw audio samples to the range -1 to +1 before the rms calculation. The scale factor is simply `1/max` where `max` is the positive maximum of the values representable in the used format. For floating point formats (F32 or F64), no scaling is needed, as they already are in the range -1 to +1.

Alternatively, the rms could be computed on the unscaled samples, followed by a scaling of the resulting rms. The same scale factor is needed as in the first approach.

These are the scale factors for some commonly used formats:

Audio format	Scale factor
S8	$1 / 2^7$
S16	$1 / 2^{15}$
S24	$1 / 2^{23}$
F32	1
F64	1

For more information on audio formats, see e.g. https://gstreamer.freedesktop.org/documentation/design/mediatype-audio-raw.html.

In a typical implementation, the `HAPPlatformMicrophoneDataCallback` calls are invoked by a separate thread. The called code is in the `Remote` accessory logic, where it must be synchronized with the main thread as needed.

`ADK/SDK-Samples/POSIX/Raspi/PAL/HAPPlatformMicrophone.c`:
Implementation that uses GStreamer and ALSA.

5. Developer Technical Support for ADK

Apple's Developer Technical Support (DTS) team may assist with code-level and circuit-level questions or provide guidance to the right documentation, schematics, and code. All DTS communication is conducted via email.

For instructions on requesting code-level technical support, search for article DS100 in the "Articles" section of the MFi Portal. Be sure to include *complete* accessory logs and all other information requested in the Troubleshooting section.

Appendix A: Set Up the POSIX SDK

This appendix describes how to set up and use the SDK to build and run a sample light bulb accessory implementation on a Raspberry Pi. The SDK is set up using a SD card with a Linux image (including the ADK's POSIX SDK) for the Raspberry Pi.

To learn how to write application code, refer to chapter Implement the Accessory Logic. To review an actual example of a light bulb accessory implementation, refer to the files in the `ADK/SDK–Samples/POSIX/Applications` directory.

To learn how to go from a Raspberry Pi to some other POSIX-based target platform, refer to chapter Platform Development. To study an actual example of a completely functional PAL, refer to the files in the `ADK/SDK–Samples/POSIX/Raspi/PAL` directory.

1. Requirements

To get started, the following items are required:

- Development platform
  - Mac running macOS High Sierra 10.13 or later
  - Administrator password for the Mac
  - SD Card reader (built into the Mac or connected via an external adapter)
  - microSD to SD Card adapter
- Internet connection
- Reference platform
  - Raspberry Pi 3 Model B (not B+). Other models of the Raspberry Pi, such as the *Raspberry Pi Zero W*, may also work, but have not been tested and no technical support is provided for them.
  - microSD Card with at least 16 GB capacity

    During the process described in this guide, a suitable minimal distribution of Raspbian (a Raspberry Pi adaptation of the Debian Linux distribution) will be written to the card. The card need not be empty; its content will be overwritten.
  - Power supply for the Raspberry Pi
  - Apple Authentication Coprocessor board (refer to Connect the Authentication Coprocessor). For first tests with Ethernet only, this board need not be attached to the Raspberry Pi, but it will be needed for Wi-Fi with WAC2.
- Connection between reference and development platforms via an Ethernet network.

2. Getting Started with Raspberry Pi

This section describes how to install the POSIX SDK, set up a Raspberry Pi 3 Model B, build a simple HomeKit accessory server as an example, deploy it on the Raspberry Pi, and how to control and observe it from the Mac using the HomeKit Accessory Tester (HAT) application.

2.1. Download files to a Mac

1. Download the latest ADK zip archive from the MFi Portal to a Mac. The directory `ADK/SDK-Samples/POSIX` contains the elements of the ADK that are needed for the Raspberry Pi. In the following text, `POSIX` is used as a shorthand for this directory.

2. Download the zip archive of the Raspberry Pi 3 image. The file is available for download in the Technical Specifications section of the MFi Portal. It contains an SD card image for the Raspberry Pi. This image is based on Raspbian Stretch Lite, a minimal Debian Stretch image from the Raspberry Pi Foundation. The following items are preinstalled:

   - `raspbian-stretch-lite 2018-06-27` (https://www.raspberrypi.org/downloads/raspbian/). This image already contains the Wi-Fi driver patch that had to be installed separately in earlier releases of the ADK.
   - `hostapd 2.4-1` (apt-get package: hostapd)
   - `dnsmasq 2.76-5` (apt-get package: dnsmasq)
   - `mDNSResponder 878.30.4` (https://opensource.apple.com/source/mDNSResponder/mDNSResponder-878.30.4/)
   - `libavahi-client-dev 0.6.32` (https://packages.debian.org/stretch/libavahi-client-dev)
   - `libusb 1.0.21` (apt-get package: libusb)
   - `libnfc 1.7.1` (https://github.com/nfc-tools/libnfc/release)
   - `GStreamer 1.10.4` (apt-get packages: `gstreamer1.0-plugins-base, gstreamer1.0-x, gstreamer1.0-tools, libgstreamer1.0-dev, gstreamer1.0-doc, gstreamer1.0-plugins-good, gstreamer1.0-plugins-ugly, gstreamer1.0-plugins-bad, gstreamer1.0-omx, gstreamer1.0-alsa`)
   - `libssl 1.1.0` (http://raspbian.raspberrypi.org/raspbian/pool/main/o/openssl)
   - `libasound2 1.1.3` (http://archive.raspberrypi.org/debian/pool/main/a/alsa-lib)
   - `wiringPi 2.46` (http://wiringpi.com/download-and-install/)
   - miscellaneous tools for simplifying development (`clang`, `valgrind`, `vim`, `tmux`, `mercurial`, `git`)

3. Download the *HomeKit Application Tester* (*HAT*) from the MFi Portal.

Note: Use the Raspbian image provided by Apple. The use of other Linux images for the Raspberry Pi as a reference platform is not supported. For licensed accessories, with different Linux distributions or dependencies, the POSIX PAL implementation of the ADK may require adaptations.

Warning: Do not use apt-get upgrade *on the the Raspbian image provided by Apple. An upgrade might de-install the Wi-Fi kernel patch and WAC2 would not work as a result. This patch is a temporary solution. In the future, the official Raspbian image is expected to include the kernel patch.*

Here it is assumed that the downloaded and unzipped archive is located in the default `Downloads` directory of the Mac.

2.2. Set up the SD card

Follow these steps to copy the Raspbian image to an SD Card. The Raspberry Pi is not needed yet, nor an Internet connection, only the SD card.

4. Ensure that the SD card is ready but not plugged into the Mac. It need not be empty, as it will be erased during the process.

5. Ensure that the Raspberry Pi is disconnected from its power supply.

6. Locate the `POSIX` directory in the Finder.

7. Open a `Terminal` window on the Mac.

8. Change the current directory to POSIX by typing "cd " (without the double quotes but with the space character) into the terminal window, then select the directory icon for `POSIX`, drop the directory icon into the terminal window, click into the terminal window again, and then press the return key.

9. Locate the `Downloads` directory in Finder.

10. Type `./Raspi/Tools/SetupRaspiSDCard` with a trailing space in the terminal.

11. Drop the icon of the downloaded Raspberry Pi image file into the terminal window and press the return key.

12. Follow the instructions written to the terminal window.

    Insert the SD card into the card reader on the Mac when the script requests this step. It requires entering the correct drive path name of the SD card reader. The available drive path name can be seen in the output in the terminal window: compare the terminal display before

```
================================================================================
/dev/disk0 (internal):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        1.0 TB     disk0
   1:                        EFI EFI                     314.6 MB   disk0s1
   2:          Apple_CoreStorage Macintosh HD           999.6 GB   disk0s2
   3:                 Apple_Boot Recovery HD            650.0 MB   disk0s3

/dev/disk1 (internal, virtual):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:                  Apple_HFS Macintosh HD           +999.2 GB   disk1
                                 Logical Volume on disk0s2
                                 76FF8E3D-F896-4751-BC77-FD8F530B0613
                                 Unlocked Encrypted

================================================================================
Please insert Raspberry Pi SD card into host.
Press [ENTER] to continue.
```

    and after insertion of the SD Card:

```
================================================================================
```

```
/dev/disk0 (internal):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        1.0 TB     disk0
   1:                        EFI EFI                     314.6 MB   disk0s1
   2:          Apple_CoreStorage Macintosh HD            999.6 GB   disk0s2
   3:                 Apple_Boot Recovery HD             650.0 MB   disk0s3

/dev/disk1 (internal, virtual):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:                 Apple_HFS Macintosh HD           +999.2 GB   disk1
                                Logical Volume on disk0s2
                                76FF8E3D-F896-4751-BC77-FD8F530B0613
                                Unlocked Encrypted

================================================================================
Please insert Raspberry Pi SD card into host.
Press [ENTER] to continue.
================================================================================
/dev/disk0 (internal):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        1.0 TB     disk0
   1:                        EFI EFI                     314.6 MB   disk0s1
   2:          Apple_CoreStorage Macintosh HD            999.6 GB   disk0s2
   3:                 Apple_Boot Recovery HD             650.0 MB   disk0s3

/dev/disk1 (internal, virtual):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:                 Apple_HFS Macintosh HD           +999.2 GB   disk1
                                Logical Volume on disk0s2
                                76FF8E3D-F896-4751-BC77-FD8F530B0613
                                Unlocked Encrypted

/dev/disk2 (internal, physical):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:     FDisk_partition_scheme                        *31.9 GB    disk2

   1:             Windows_FAT_32 NO NAME                31.9 GB    disk2s1

================================================================================
Please type Raspberry Pi SD card device name: /dev/disk
```

Note the new disk `/dev/disk2`. This is the SD card drive, recognizable by its relatively small size.

So in this case, add a 2 after `/dev/disk` and press the `return` key.

Warning: All existing data on the SD Card will be deleted.

When asked whether to erase the disk, type Y.

Note: The script may ask for a password one or more times. Enter the administrator password of the Mac and press the return key. While typing in the password no visible feedback is displayed.

Note: This script may run for about ten minutes without much terminal output. Check progress by pressing control-T (not command-T) in the terminal window from time to time.

13. The script will ask to remove the SD card from the Mac:

```
4404019200 bytes transferred in 68.284103 secs (64495527 bytes/sec)
Disk /dev/rdisk2 ejected
SD card set up completed.
You can now remove SD card from host!
```

14. The script will ask for a new host name for the Raspberry Pi and for a new password for the user pi:

```
================================================================================
Please enter hostname for Raspberry Pi ([ENTER] to keep 'raspberrypi'):
Please enter desired password for user 'pi' ([ENTER] to keep default):
Please retype password:

Note: If you press Return, the default hostname 'raspberrypi' and the default
password 'raspberry' for user 'pi' are used. It is recommended to change the
hostname to avoid conflicts in networks with more than one connected Raspberry
Pi.
```

15. The script asks for the country in which the Raspberry Pi will be used to configure the Wi-Fi module according to the regulations in this country:

```
Please enter the country in which the Raspberry Pi will be used (Press return to
show list of country codes).
Country code:
```

16. The script will ask to insert the SD card into the Raspberry Pi (which is still unpowered):

```
================================================================================
1. Put SD card into Raspberry Pi.
2. Connect Raspberry Pi to host via Ethernet / USB.
3. Power Raspberry Pi.
Press [ENTER] to continue.
```

17. Insert the SD card into the Raspberry Pi. Connect the Ethernet port of the Raspberry Pi directly to the Mac or to the same Network router as the one the Mac is connected to. Attach the power cable to the Raspberry Pi. Press return to continue with the installation script. It will configure the additional libraries needed for HomeKit and reboot the Raspberry Pi. This can take several minutes. Ignore messages like `Re-reading the partition table failed.: Device or resource`.

Note: Connect the Raspberry Pi as directly as possible to the Mac. Adapters between the two may cause connection issues.

18. The following output indicates that the script has ended:

```
pi@raspberrypi:~ $ sudo reboot
Connection to raspberrypi.local closed by remote host.
Connection to raspberrypi.local closed.
================================================================================
Setup complete.
Log in with `ssh pi@raspberrypi.local` and the previously supplied password.
================================================================================
Testusers-iMac:POSIX testuser$
```

2.3. Copy POSIX SDK to Raspberry Pi

The HomeKit accessory logic, e.g., the lightbulb sample, is built not on the Mac but on the Raspberry Pi itself. To make this possible, the POSIX sample SDK must first be copied to the Raspberry Pi.

Use the following script to copy the SDK-`Samples/POSIX` directory contents from the Mac to the Raspberry Pi, which takes a few seconds:

```
./Raspi/Tools/CopyRaspiSamples
```

Note: This guides assumes a Raspberry Pi with the default host name `raspberrypi`. Instead, use the hostname provided in step 14 above.

The script will ask for the hostname and password of the Raspberry Pi:

```
Testusers-iMac:POSIX testuser$ ./Raspi/Tools/CopyRaspiSamples
================================================================================
Copying POSIX SDK with samples to Raspberry Pi.
================================================================================

Please enter hostname for Raspberry Pi ([return] to use 'raspberrypi').
Hostname: raspberrypi

Please enter password for Raspberry Pi ([return] to use 'raspberry').
Password:
================================================================================
Copying POSIX SDK with samples to Raspberry Pi.
spawn rsync -avz --no-o --no-g --super --relative -e ssh -o
StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null --rsync-path=sudo rsync --
exclude=.DS_Store /Users/testuser/Desktop/HomeKit-ADK_1.0_15D1/ADK/SDK-Samples/
POSIX/Raspi/Tools/../.././Raspi/Applications /Users/testuser/Desktop/HomeKit-
ADK_1.0_15D1/ADK/SDK-Samples/POSIX/Raspi/Tools/../.././Raspi/PAL /Users/testuser/
Desktop/HomeKit-ADK_1.0_15D1/ADK/SDK-Samples/POSIX/Raspi/Tools/../.././Raspi/HAP/
include /Users/testuser/Desktop/HomeKit-ADK_1.0_15D1/ADK/SDK-Samples/POSIX/Raspi/
Tools/../.././Raspi/HAP/lib/ARM/GCC/ /Users/testuser/Desktop/HomeKit-ADK_1.0_15D1/
ADK/SDK-Samples/POSIX/Raspi/Tools/../.././Raspi/Tools/helpers/Pi /Users/testuser/
Desktop/HomeKit-ADK_1.0_15D1/ADK/SDK-Samples/POSIX/Raspi/Tools/../.././Xtras
pi@raspberrypi.local:~
Warning: Permanently added
'raspberrypi.local,fe80::b873:a26a:f8d0:e175%en0' (ECDSA) to the list of known
hosts.
pi@raspberrypi.local's password:
building file list ... done
Raspi/
Raspi/Applications/
Raspi/Applications/Bridge/
Raspi/Applications/Bridge/App.c
Raspi/Applications/Bridge/App.h
Raspi/Applications/Bridge/DB.c
Raspi/Applications/Bridge/DB.h
...
Xtras/PAL/WiFiManager/Null/HAPPlatformWiFiManager.c

sent 1471961 bytes  received 4464 bytes  984283.33 bytes/sec
total size is 5497078  speedup is 3.72
================================================================================
Copying to RaspberryPi complete.
Log in with `ssh pi@raspberrypi.local` and the previously supplied password.
================================================================================
Testusers-iMac:POSIX testuser$
```

2.4. Provision LightbulbLED example with a setup code

Use the following script to generate accessory setup information and deploy it to the key-value store of the `LightbulbLED` example on the RaspberryPi:

```
./Raspi/Tools/Provision --category 5 pi@raspberrypi.local:~/Raspi/
Applications/LightbulbLED/.HomeKitStore
```

Note: This script must be run on the Mac.

The script will display a random setup code, the setup payload, and its corresponding QR code:

```
================================================================================

                            Setup Code: 111-22-333

                         Setup Payload: X-HM://00527813XACME

================================================================================
```

Use SSH to connect from the Mac's `Terminal` window to the Raspberry Pi shell. Type in the following command:

```
ssh pi@raspberrypi.local
```

and then press return:

```
Testusers-iMac:POSIX testuser$ ssh pi@raspberrypi.local
```

When asked whether to continue connecting, type `yes` and `return`:

```
Testusers-iMac:POSIX testuser$ ssh pi@raspberrypi.local
The authenticity of host 'raspberrypi.local (fe80::b873:a26a:f8d0:e175%en0)' can't
be established.
ECDSA key fingerprint is SHA256:S076znlRKY28v/Wd7HjtqXAVuivnL5trQNSja2TIh9Q.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added
'raspberrypi.local,fe80::b873:a26a:f8d0:e175%en0' (ECDSA) to the list of known
hosts.
pi@raspberrypi.local's password:
```

The warning means that the Mac doesn't yet know this Raspberry Pi, which is correct.

Now enter the password for the Raspberry Pi. This is the password provided in step 14. The following output is displayed:

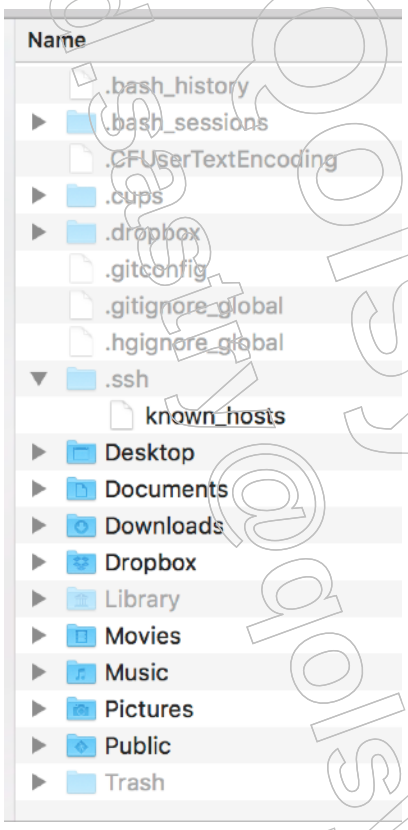
```
Linux raspberrypi 4.9.67-v7 #1 SMP Wed Nov 29 04:08:24 UTC 2017 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
```

In the case that you've previously opened ssh to the Raspberry Pi, the following message is displayed and additional steps are required:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
```

```
SHA256:Gnk2bFWoGXmoLEDkhEepqFexLQ+77j2sAp5u2thHvTA.
Please contact your system administrator.
Add correct host key in /Users/testuser/.ssh/known_hosts to get rid of this
message.
Offending ECDSA key in /Users/testuser/.ssh/known_hosts:2
ECDSA host key for raspberrypi.local has changed and you have requested strict
checking.
Host key verification failed.
lost connection
Testusers-iMac:POSIX testuser$
```

This resulted in an error because the Mac has stored the MAC address of the Raspberry Pi during the first attempt. Execute open `~/.ssh/known_hosts`, or go to the Finder and select the `Go > Home` menu command to see the following:



Open the `known_hosts` text file in the `.ssh` directory with a text editor, and delete the entry for the Raspberry Pi. Save the file and then retry the SSH command.

Note: Type `exit` followed by `return` to exit SSH.

2.5. Build and deploy LightbulbLED example

It is assumed that an SSH connection is now open, i.e., the following prompt is displayed:

```
pi@raspberrypi:~ $
```

To change to one of the sample accessory app directories - the light bulb, use the cd (change directory) command:

```
pi@raspberrypi:~ $ cd Raspi/Applications/LightbulbLED/
```

This example simulates a Lightbulb by writing log messages to the console and simulates a light bulb with a green LED on the Raspberry Pi.

Note: This is the LED that normally is used by the operating system for indicating network or SD Card activities. For `LightbulbLED`, this functionality is switched off.

2.6. LightbulbLED via Ethernet

Assuming Ethernet is used rather than Wi-Fi (otherwise, skip to the next section), enter these two commands to build the example, each followed by `return`:

```
sudo make clean

sudo make debug
```

Example output in the terminal window:

```
pi@raspberrypi:~/Raspi/Applications/LightbulbLED $ sudo make clean
Platform: Linux 4.14.50-v7+ #1122 SMP Tue Jun 19 12:26:26 BST 2018 armv7l
rm -f LightbulbLED ../../PAL/HAPPlatformRandomNumber.o ../../PAL/
HAPPlatformMFiHWAuth.o ../../PAL/HAPPlatformServiceDiscovery.o App.o DB.o
Main.o ../../../Xtras/PAL/Camera/Null/HAPPlatformCamera.o ../../PAL/
HAPPlatform.o ../../PAL/HAPPlatformAbort.o ../../PAL/
HAPPlatformAccessorySetup.o ../../PAL/HAPPlatformBLEPeripheralManager.o ../../PAL/
HAPPlatformClock.o ../../PAL/


pi@raspberrypi:~/Raspi/Applications/LightbulbLED $ sudo make debug
Platform: Linux 4.14.50-v7+ #1122 SMP Tue Jun 19 12:26:26 BST 2018 armv7l
cc -DLED_PORT=\"/sys/class/leds/led0/brightness\" -DLED_TRIGGER=\"/sys/class/leds/
led0/trigger\" -DHAVE_MFI_HW_AUTH=1 -std=c99 -pedantic -pedantic-errors -Werror -
g0 -Os -fno-strict-overflow -fno-strict-aliasing -Wall -Wextra -pthread -I./ -I../
-I../../HAP/include -I../../PAL -I../../PAL/include -I../../Dependencies/hidapi -
I../../Dependencies/hidapi/hidapi -fno-delete-null-pointer-checks -
D_POSIX_C_SOURCE=200112 -D_DEFAULT_SOURCE  -DDEBUG -DHAP_LOG_LEVEL=3 -MMD -MP  -c
-o ../../PAL/HAPPlatformRandomNumber.o ../../PAL/HAPPlatformRandomNumber.c
cc -lm -L../../HAP/lib/ARM/GCC/arm-unknown-linux-gnueabihf/armv7-a -Wl,--gc-
sections  ../../PAL/HAPPlatformRandomNumber.o ../../PAL/
HAPPlatformMFiHWAuth.o ../../PAL/HAPPlatformServiceDiscovery.o App.o DB.o
Main.o ../../../Xtras/PAL/Camera/Null/HAPPlatformCamera.o ../../PAL/
HAPPlatform.o ../../PAL/HAPPlatformAbort.o ../../PAL/
HAPPlatformAccessorySetup.o ../../PAL/HAPPlatformBLEPeripheralManager.o ../../PAL/
HAPPlatformClock.o ../../PAL/HAPPlatformFileManager.o ../../PAL/
HAPPlatformKeyValueStore.o ../../PAL/HAPPlatformLog.o ../../PAL/
HAPPlatformMFiTokenAuth.o ../../PAL/HAPPlatformRunLoop.o ../../PAL/
HAPPlatformSoftwareAccessPoint.o ../../PAL/HAPPlatformTCPStreamManager.o ../../
PAL/HAPPlatformWiFiManager.o  -ldns_sd -lhap -lpthread -o LightbulbLED
```

Note: If the Raspberry Pi has no connection to the Internet, the system time will be wrong. As a consequence, make issues a warning: "Clock skew detected. Your build may be incomplete.". To get rid of these warnings, set the time with the following command: sudo date `MMDDhhmmYYYY`. Now run the lightbulb sample by entering the following command, followed by return:

```
sudo ./LightbulbLED
```

This command starts a HomeKit accessory server on the Raspberry Pi that provides the Lightbulb profile.

```
pi@raspberrypi:~/Raspi/Applications/LightbulbLED $ sudo ./LightbulbLED
```

```
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Platform:KeyValueStore]
Storage configuration: keyValueStore = 4
2018-11-01'T'15:37:52'Z' Default
[com.apple.mfi.HomeKit.Platform:AccessorySetup] Storage configuration:
accessorySetup = 12
2018-11-01'T'15:37:52'Z' Default
[com.apple.mfi.HomeKit.Platform:AccessorySetup] Using display: false /
Programmable NFC: false
2018-11-01'T'15:37:52'Z' Default
[com.apple.mfi.HomeKit.Platform:TCPStreamManager] Storage configuration:
tcpStreamManager = 60
2018-11-01'T'15:37:52'Z' Default
[com.apple.mfi.HomeKit.Platform:TCPStreamManager] Storage configuration:
maxTCPStreams = 17
2018-11-01'T'15:37:52'Z' Default
[com.apple.mfi.HomeKit.Platform:TCPStreamManager] Storage configuration:
tcpStreams = 408
2018-11-01'T'15:37:52'Z' Default
[com.apple.mfi.HomeKit.Platform:ServiceDiscovery] Storage configuration:
serviceDiscovery = 552
2018-11-01'T'15:37:52'Z' Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth]
HAPPlatformMFiHWAuthCreate
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Platform:MFiHWAuth]
Storage configuration: mfiHWAuth = 8
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Platform:RunLoop]
SystemInfo:
Linux 4.14.50-v7+ #1122 SMP Tue Jun 19 12:26:26 BST 2018 armv7l GNU/Linux
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Platform:RunLoop]
Storage configuration: runLoop = 320
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Platform:RunLoop]
Storage configuration: fileHandle = 28
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Platform:RunLoop]
Storage configuration: timer = 24
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:AccessoryServer]
Version information:
libhap: /ARM/GCC/arm-unknown-linux-gnueabihf/armv7-a/
  - Version: 2.0 (16A61) - compatibility version 4
Using platform: Raspi
  - Version: 2.0 (16A61) - compatibility version 4
```

Note the version information with the libhap variant and the build number (16A61). Your ADK version may have a newer build version. In case of issues with the HAP Library binaries, send complete logs including this key version information to Apple for analysis.

Now there follow further storage configuration messages …

```
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:AccessoryServer]
Storage configuration: server = 1864
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: ipAccessoryServerStorage = 40
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: numSessions = 17
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: sessions = 14416
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: sessions[0...16].inboundBuffer.numBytes = 32768
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: sessions[0...16].outboundBuffer.numBytes = 32768
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: sessions[0...16].numEventNotifications = 21
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: sessions[0...16].eventNotifications = 504
2018-11-01'T'15:37:52'Z' Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: numReadContexts = 21
```

```
2018-11-01'T'15:37:52'Z'  Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: readContexts = 1008
2018-11-01'T'15:37:52'Z'  Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: numWriteContexts = 21
2018-11-01'T'15:37:52'Z'  Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: writeContexts = 1344
2018-11-01'T'15:37:52'Z'  Default    [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Storage configuration: scratchBuffer.numBytes = 32768
2018-11-01'T'15:37:52'Z'  Info  AppCreate
2018-11-01'T'15:37:52'Z'  Info  ConfigureIO
2018-11-01'T'15:37:52'Z'  Info  DeviceConfigureLED
2018-11-01'T'15:37:52'Z'  Info  DeviceConfigureLED: Disable LED trigger.
2018-11-01'T'15:37:53'Z'  Info  DeviceDisableLED
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:AccessoryServer]
Accessory server starting.
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:AccessoryServer]
Firmware version: 1.0.0
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:AccessoryServer] [1.0.0]
Storing initial firmware version.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Core:AccessoryServer]
Registering accessories.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Core:AccessoryServer] Loading
accessory identity.
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:AccessoryServer]
<private> Generated new LTSK.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Core:AccessoryServer]
Checking if admin pairing exists.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
Starting server engine.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:TCPStreamManager]
setsockopt(7, SOL_SOCKET, SO_REUSEADDR, <buffer>);
    0000  01000000
....
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:TCPStreamManager]
TCP stream listener interface index: 0
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:TCPStreamManager]
bind(7, <buffer>);
    0000  0a000000 00000000 00000000 00000000 00000000 00000000 00000000
.......................
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:TCPStreamManager]
TCP stream listener port: 34633
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:TCPStreamManager]
listen(7, 64);
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:MFiHWAuth] Turning on
Apple Authentication Coprocessor.
2018-11-01'T'15:37:53'Z'  Default    [com.apple.mfi.HomeKit.Platform:Clock] Using
'clock_gettime' with 'CLOCK_MONOTONIC_RAW'.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x05.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 05
    0000  00
.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x00.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 00
    0000  05
.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x01.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 01
    0000  01
.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x02.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 02
```

```
    0000  02
.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x03.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 03
    0000  00
.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x05.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 05
    0000  00
.
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:MFiHWAuth] Apple
Authentication Coprocessor information:
- Device Version: 2.0C (0x05)
- Firmware Version: 1
- Authentication Protocol Version: 2.0
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi >
    0000  4001
@.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi write
complete.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x40.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 40
    0000  c0
.
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:DeviceID] Generated new
Device ID.
    0000  5452ed5c 1637
TR.\.7
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Core:IPServiceDiscovery]
Registering _hap._tcp service.
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
interfaceIndex: 0
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
name: "Acme Lightbulb LED"
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
protocol: "_hap._tcp"
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
port: 34633
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[0]: "c#"
    0000  31
1
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[1]: "ff"
    0000  31
1
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[2]: "id"
    0000  35343a35 323a4544 3a35433a 31363a33 37
54:52:ED:5C:16:37
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[3]: "md"
    0000  4c696768 7462756c 62312c31
Lightbulb1,1
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[4]: "pv"
    0000  312e31
1.1
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[5]: "s#"
    0000  31
1
```

```
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[6]: "sf"
    0000  31
1
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[7]: "ci"
    0000  35
5
2018-11-01'T'15:37:53'Z'  Debug [com.apple.mfi.HomeKit.Platform:ServiceDiscovery]
txtRecord[8]: "sh"
    0000  54666b72 38413d3d
Tfkr8A==
2018-11-01'T'15:37:53'Z'  Info  [com.apple.mfi.HomeKit.Platform:RunLoop] Entering
run loop.
2018-11-01'T'15:37:53'Z'  Info  Accessory Server State did update: Running.
2018-11-01'T'15:37:56'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi read
0x4d.
2018-11-01'T'15:37:56'Z'  Debug [com.apple.mfi.HomeKit.Platform:MFiHWAuth] MFi < 4d
    0000  00
.
2018-11-01'T'15:37:56'Z'  Debug [com.apple.mfi.HomeKit.Core:MFiHWAuth] System Event
Counter = 0.
2018-11-01'T'15:37:56'Z'  Info  [com.apple.mfi.HomeKit.Core:MFiHWAuth] Turning off
Apple Authentication Coprocessor.
```

The log output during startup of the accessory server, as shown above, indicates that the server attempted to turn on the authentication coprocessor, but failed because no authentication coprocessor is attached (I2C read timed out). In this case, use of the authentication coprocessor is not strictly enforced to allow for very early prototyping.

In order to toggle the lightbulb state between on and `off`, send the `SIGUSR1` signal to the `LightbulbLED` process - the paired controllers will be notified about such state changes. To send the signal, open a second terminal on the Mac and open a second remote shell on the Raspberry Pi:

```
ssh pi@raspberrypi.local
```

Then execute the following command:

```
sudo killall -SIGUSR1 LightbulbLED
```

The green LED on the Raspberry Pi will toggle its state and in the log of the first terminal you can observe the change, as well:

```
2018-11-01'T'15:39:26'Z'    Info ToggleLightbulbState: true
2018-11-01'T'15:39:26'Z'    Info DeviceEnableLED
```

Note: While the `LightbulbLED` sample supports signal handling, the minimal `Lightbulb` sample does not.

The log output shows that a new Device ID was generated. This ID will be stored and no new ID will be generated when the accessory server starts up again.

Go to the next chapter to see what happens when a HomeKit controller accesses the running server.

Notes:

Stopping the accessory server is done by entering `Control–C` in the terminal window, which leads back to the SSH prompt.

Enter the sudo `./LightbulbLED` command to restart the stopped accessory server.

Use the command sudo `rm –r .HomeKitStore/` in the same directory to delete Device ID and keys from the accessory ("factory reset"). Use the command `sudo reboot` to reboot the Raspberry Pi.

2.7. LightbulbLED via Wi-Fi including WAC2

On the Mac, generate and deploy a setup code for a lightbulb that supports WAC2:

```
./Raspi/Tools/Provision ––wac ––category 5 pi@raspberrypi.local:~/Raspi/
Applications/LightbulbLED/.HomeKitStore
```

Note: The `Provision` tool supports QR codes, NFC, and accessories with displays. The sample applications provide corresponding build options. For example, a setup code provisioned for NFC requires build option `USE_NFC=1` during `make`.

Note: The Provision tool supports hardware authentication with the MFi authentication coprocessor and software authentication via tokens. The available tool options are displayed if `Provision` is run without arguments. For software authentication, tokens can only be used once – the accessory will maintain a token between tests, but if you provision with the same token again, it will not be usable a second time.

If an authentication coprocessor is attached (see corresponding section), build the example by entering the following command on the Raspberry Pi, followed by `return`:

```
sudo make clean USE_WAC=1

sudo make debug USE_WAC=1
```

Now run the lightbulb sample by entering the following command, followed by return:

```
sudo ./LightbulbLED
```

Note: Without an authentication coprocessor or software authentication, WAC2 does not work. But it is possible to run the lightbulb example described in the previous section and use Wi-Fi by providing the Wi-Fi credentials manually. For this purpose, ssid and password can be set by editing file `/etc/wpa_supplicant/wpa_supplicant.conf` for very early prototyping.

Note: `LightbulbLED` will only go through WAC2 if Wi-Fi is not yet configured. To make `LightbulbLED` go through WAC2 again, restore `LightbulbLED` to factory settings by executing the following command on the Raspberry Pi:

```
sudo killall –SIGUSR2 LightbulbLED
```

The interpretation of signals is application dependent. `LightbulbLED` uses `SIGUSR2` for factory reset, in `LockSimulation` `SIGUSR2` is used to simulate the jamming of a lock.

Note: If there are issues connecting the HomeKit controller (e.g., iPhone) with the Raspberry Pi, check the following:

- *No legacy security is used: WEP, WPA1*
- *No enterprise security is used: WPA2 Enterprise*
- *The 2.4 GHz band is used*

- *No special characters in the Wi-Fi name or too long Wi-Fi names are used*
- *MAC addresses are not filtered out*
- *The HomeKit controller is not currently a Wi-Fi access point*

Note: The bandwidth of a Wi-Fi connection from a HomeKit controller to the Raspberry Pi is considerably lower, and packet losses occur more frequently, than in the other direction. This does not happen when using Ethernet.

2.8. Summary of build options

The following build options are supported (several options can be combined in one `make` command):

Command	Description	Default
`make`	Production build, no logs.	-
`make debug`	Debug build, highest log level.	-
`make USE_MFI_HW_AUTH=0`	Switches off use of authentication coprocessor.	ON
`make USE_DISPLAY=1`	Assumes use of display. This is emulated via log output, so logging must be switched on as well.	OFF
`make USE_WAC=1`	Switches on WAC.	OFF
`make USE_NFC=1`	Switches on NFC.	OFF
`make SDLIB=Avahi`	Switches service discovery to Avahi. See note below.	mDNSResponder
`make USE_BCT_NAME_CHANGE=1`	Switches on BCT name change. This option is specific to `LightbulbLED`.	OFF

The `SDLIB=Avahi` build option requires Avahi to be installed and enabled as the system's mDNS service. The Raspberry Pi image has both mDNSResponder and Avahi preinstalled. By default, mDNSResponder is enabled (Avahi is switched off, because only one mDNS service should be enabled). To enable Avahi as mDNS service, execute the following commands in a shell on the Raspberry Pi:

```
sudo systemctl stop host-local.service

sudo systemctl disable host-local.service

sudo systemctl stop mdns.service

sudo systemctl mask mdns.service


sudo systemctl unmask avahi-daemon.service

sudo systemctl unmask avahi-daemon.socket

sudo systemctl enable avahi-daemon.service

sudo systemctl start avahi-daemon.service
```

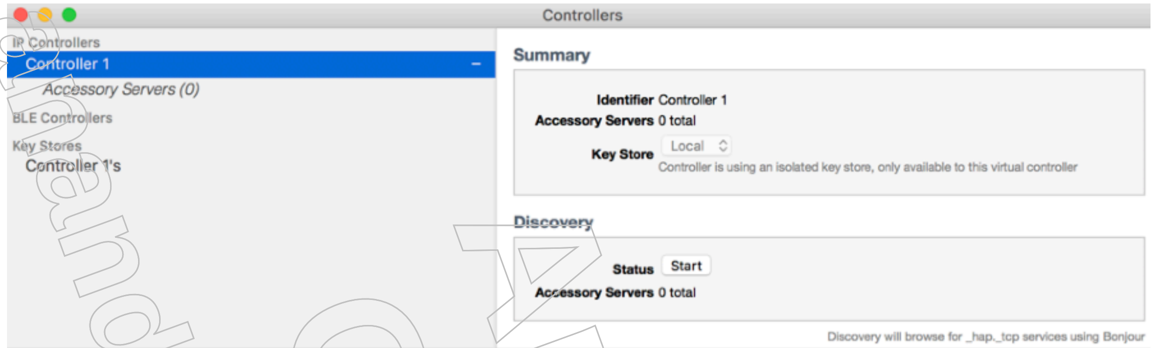
To switch back to mDNSResponder, execute the following commands:

```
sudo systemctl stop avahi-daemon.service
sudo systemctl disable avahi-daemon.service
sudo systemctl mask avahi-daemon.service
sudo systemctl mask avahi-daemon.socket

sudo systemctl unmask mdns.service
sudo systemctl start mdns.service
sudo systemctl enable host-local.service
sudo systemctl start host-local.service
```

To mask `avahi-daemon` is not strictly necessary, but protects against accidentally restarting Avahi.

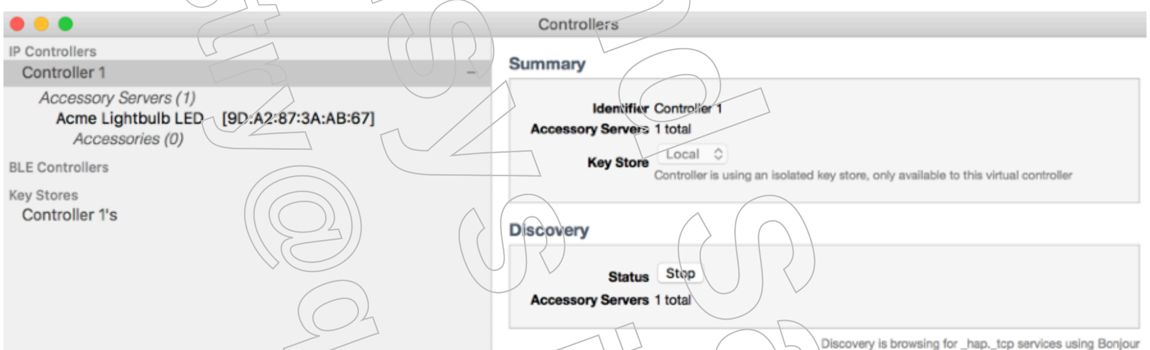
2.9. Use HAT to see the LightbulbLED accessory

Start the application *HomeKit Accessory Tester*. When the *HAT* opens, a window called `Controllers` is displayed. (If not, select the `Window > Controllers` menu item.)



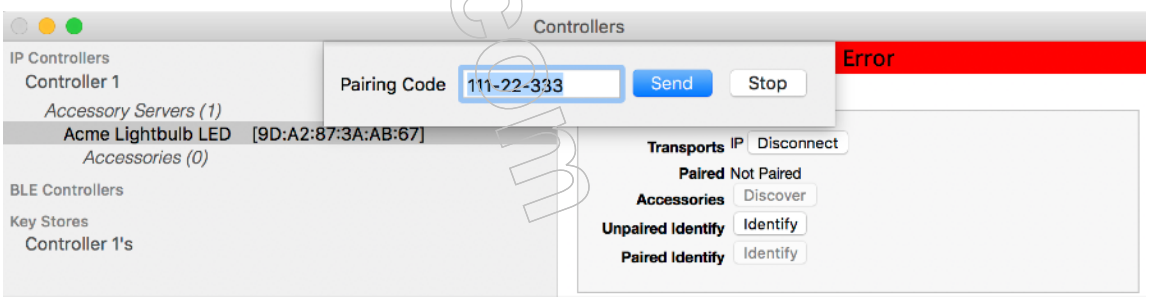
HAT controllers, such as `Controller 1`, use the HAP protocol in a similar way as iOS and the Home app do, and provide additional visibility and control over the protocol interactions.

Prior to pairing the controller with a HomeKit accessory, the controller must be able to discover available unpaired accessories. Click the `Start` button in the `Discovery` box. The Acme `Lightbulb` LED accessory server with its HomeKit Device ID appears (looks like a MAC address):

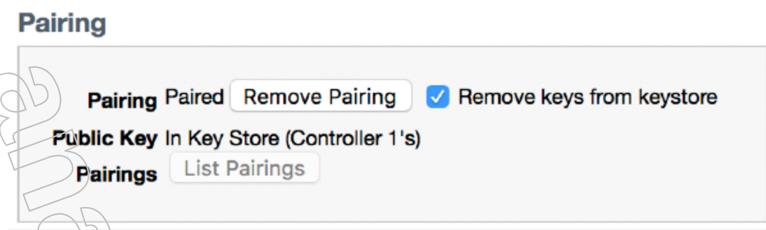


Click once on Acme `Lightbulb LED`:

Click on the `Start Pairing` button in the `Pairing` box. A dialog box opens and prompts for the setup code.



Click on the Send button. Now the pairing is performed, resulting in the following Pairing box:

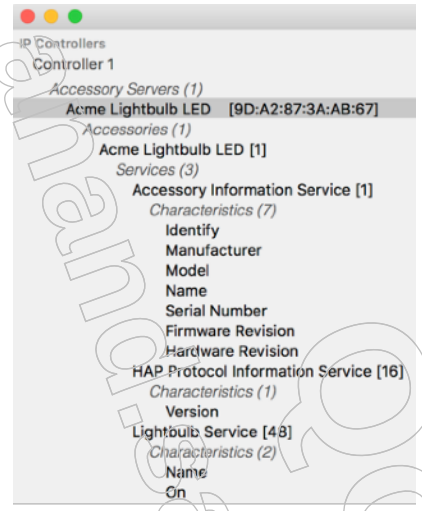


In the terminal window, new debug log output is available, displaying e.g. the HTTP response that the accessory returns to the controller and which is part of the HAP over IP protocol:

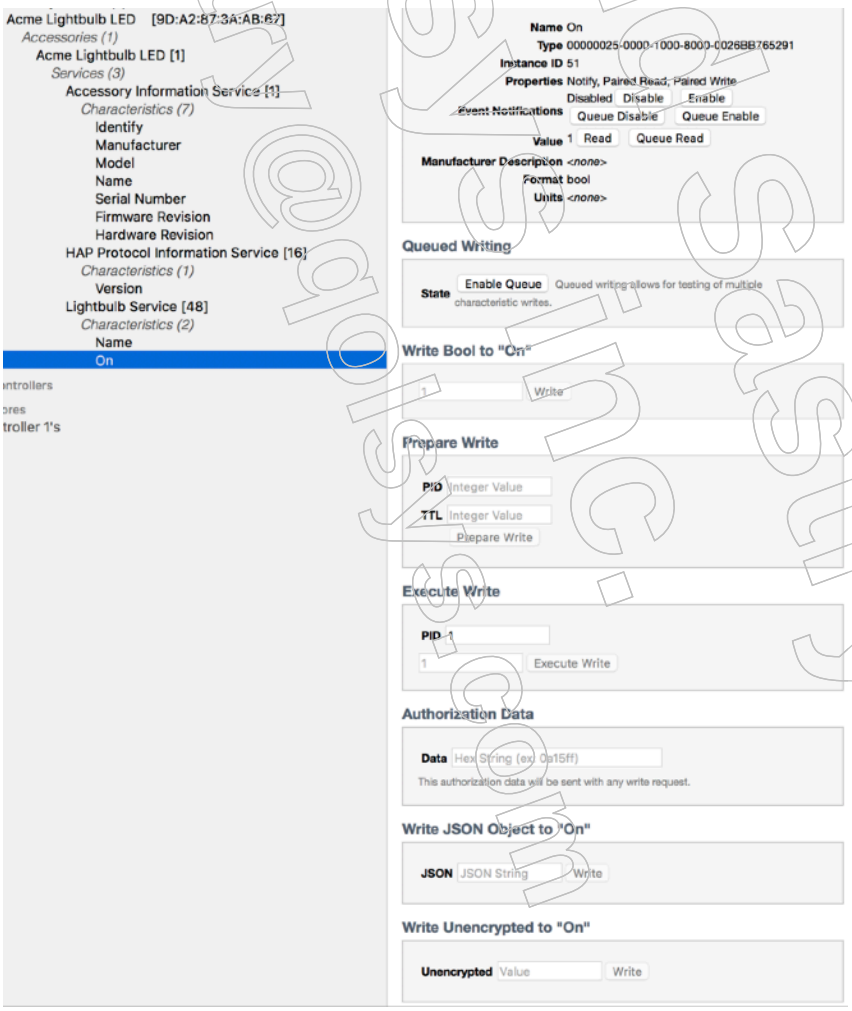
```
2018-11-01'T'15:45:09'Z'  Debug [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
session:0x6c5a8:<
    0000  48545450 2f312e31 20323030 204f4b0d
   0a436f6e 74656e74 2d547970 653a2061     HTTP/1.1 200 OK..Content-Type: a
    0020  70706c69 63617469 6f6e2f70 61697269
   6e672b74 6c76380d 0a436f6e 74656e74     pplication/pairing+tlv8..Content
    0040  2d4c656e 6774683a 20313430 0d0a0d0a
   06010203 20090063 1b0b7e8e 69208244     -Length: 140........ ..c..~.i .D
    0060  f24cf337 7e5b43d8 d9b9735a
   0056d3f6 6c78a3b8 01056585 2f05685e f79c0b10    .L.7~[C...sZ.V..lx....e./.h^....
    0080  a49dd194 d2381f0f 779b0f03 5c86e51e
   aaafed01 05436eed 135a5890 3bc3d546     .....8..w...\........Cn..ZX.;..F
    00a0  27cf6237 f239def7 c7ae4acc f79af4ef
   7f6c5a41 78baab23 2cec6c14 f5cf078b     '.b7.9....J......lZAx..#,.l.....
    00c0  3ba6b003 fcfae5fa 01de5064 f88d9cbf
   bc531fba 097ef9c9 331377bd              ;.........Pd.....S...~..3.w.
```

Close the trace window and safely click Don't Save, no information will be lost.

In the `Controllers` window, in its `Summary` box, click the button `Discover`. A list of HomeKit services with their characteristics will now be displayed:



In this case the three services `Accessory Information Service`, `HAP-BLE Protocol Information Service` and `Lightbulb Service` are shown. Then click on the On characteristic in the `Lightbulb Service`:



When clicking on the Read button in the Summary box, the controller sends a HTTP GET request to the accessory server for the On characteristic, for which the server returns `false`, as seen in the log:

```
2018-11-01'T'15:46:14'Z'  Debug [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
session:0x6c5a8:>
    0000  47455420 2f636861 72616374 65726973
   74696373 3f69643d 312e3531 20485454    GET /characteristics?id=1.51 HTT
    0020  502f312e 310d0a48 6f73743a 2041636d
   655c3033 324c6967 68746275 6c625c30    P/1.1..Host: Acme\032Lightbulb\0
    0040  33324c45 442e5f68 61702e5f 7463702e
   6c6f6361 6c0d0a43 6f6e7465 6e742d4c    32LED._hap._tcp.local..Content-L
    0060  656e6774 683a2030 0d0a0d0a
ength: 0....
2018-11-01'T'15:46:14'Z'  Info  [com.apple.mfi.HomeKit.Core:Characteristic]
[0000000000000001 Acme Lightbulb LED] [0000000000000033 on] Calling read handler.
2018-11-01'T'15:46:14'Z'  Info  HandleLightbulbOnRead: false
2018-11-01'T'15:46:14'Z'  Debug [com.apple.mfi.HomeKit.Core:IPAccessoryServer]
session:0x6c5a8:<
    0000  48545450 2f312e31 20323030 204f4b0d
   0a436f6e 74656e74 2d547970 653a2061    HTTP/1.1 200 OK..Content-Type: a
    0020  70706c69 63617469 6f6e2f68 61702b6a
   736f6e0d 0a436f6e 74656e74 2d4c656e    pplication/hap+json..Content-Len
    0040  6774683a 2035300d 0a0d0a7b 22636861
   72616374 65726973 74696373 223a5b7b    gth: 50....{"characteristics":[{
    0060  22616964 223a312c 22696964 223a3531
   2c227661 6c756522 3a307d5d 7d          "aid":1,"iid":51,"value":0}]}
```

This assumes that you have not toggled the state of the LED, otherwise the result will be true instead of false.

Now enter true (with lowercase “t”!) into the `JSON` field in the `Controller` window and click `Write`. As a result the LED will be switched on.

In order to toggle the lightbulb state between on and `off`, send the `SIGUSR1` signal to the accessory. To send the signal, execute the following command:

```
sudo killall -SIGUSR1 LightbulbLED
```

2.10. Modify the LightbulbLED example

To get started modifying the lightbulb example, go to the following path:

```
Raspi/Applications/LightbulbLED/
```

This example is based on the accessory definition in the following file:

```
App.c
```

and it implements a simple accessory server simulating a light bulb via log messages on stdout, uses the green onboard LED of the Raspberry Pi as light bulb, and handles signals. Upon receiving a SIGUSR1 signal, the lightbulb is toggled. Upon receiving a SIGUSR2 signal, a factory reset is triggered.

Note: To run a lightbulb sample on a platform that does not support signals, start with the `Lightbulb` sample instead of `LightbulbLED`.

The initialization and configuration of the platform-specific PAL modules, and the creation and start of the accessory server is done in the file:

```
Main.c
```

Refer to the source code for more details.

2.11. Configure LightbulbLED as a service that is automatically restarted on power on

The system is configured by default to not restart an accessory server after a reboot. This is convenient during development, where automatic restart interferes with the edit-compile-run cycle of the developer. During actual use of an accessory product, and sometimes for testing, automatic restart is needed. See the section Configuring the Raspberry Pi for Bonjour Conformance Test below how to configure the lightbulb as a service that is automatically restarted on power on.

2.12. User interaction with the POSIX samples

To interact with the various POSIX-specific samples, Linux signals can be sent with the following command:

```
sudo killall -SIGUSR<signal number> <sample name>
```

For example:

```
sudo killall -SIGUSR1 LightbulbLED
```

These are the supported interactions:

- For `LightbulbLED`:
  - `SIGUSR1` = toggle state of green LED between on and off
  - `SIGUSR2` = trigger a factory reset
  - `SIGTERM` = trigger a controlled shutdown
- For ThermostatSimulation:
  - `SIGUSR1` = rotate the target heating cooling state between Off, Heat, Cool and Auto
  - `SIGUSR2` = trigger a factory reset
  - `SIGTERM` = trigger a controlled shutdown
- For LockSimulation:
  - `SIGUSR1` = toggle lock state between secured (green LED on) and unsecured (green LED off)
  - `SIGUSR2` = toggle whether lock is blocked, i.e., if subsequent lock (un-)secure will fail
  - `SIGTERM` = trigger a controlled shutdown
- For `BridgeSimulation`:
  - `SIGUSR1` = toggle whether first bridged lightbulb is reachable
  - `SIGUSR2` = toggle whether second bridged lightbulb is reachable
  - `SIGTERM` = trigger a controlled shutdown
- For IPCamera:

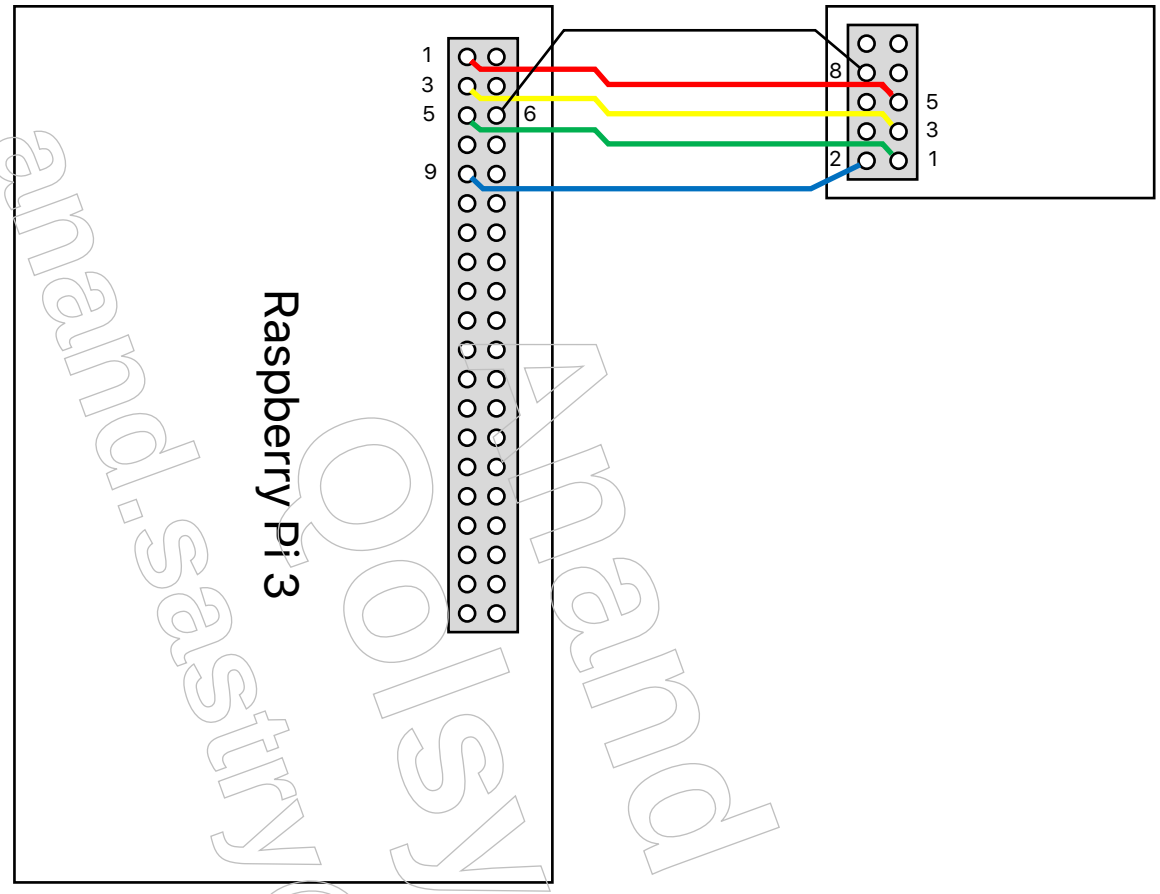
- `SIGUSR1` = enter WAC mode
- `SIGUSR2` = trigger a factory reset
- `SIGTERM` = trigger a controlled shutdown

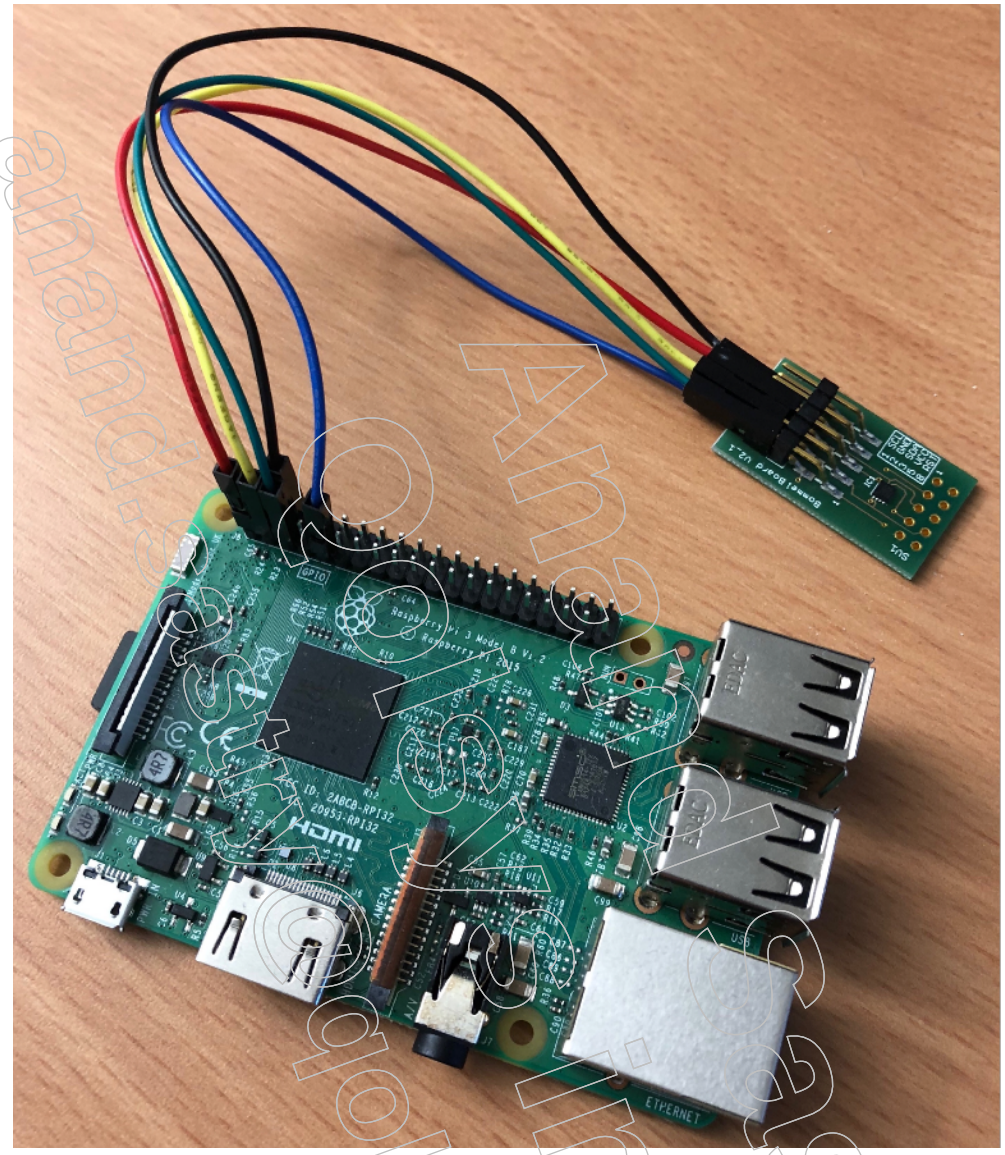
2.13. Connect the Apple Authentication Coprocessor

Use a breakout board for the authentication coprocessor and attach it to Raspberry Pi. The use of a breakout board *Bommel Board V2.1* is assumed, with an Apple Authentication Coprocessor V2.0C.

If not present yet, a male or female two-row connector must be soldered to the Raspberry Pi board. This allows the Raspberry Pi to talk to the authentication coprocessor via an I2C interface. The following table and diagram illustrate the wiring for a Raspberry Pi 3:

Raspberry Pi GPIO Connector	Signal	MFi Board SV2 Connector	Wire Color
1	3.3 V	5	red
9	Ground	2	blue
3	SDA	3	yellow
5	SDL	1	green
6	Reset, Address (= Ground)	8	black





Raspberry Pi 3 with connected authentication coprocessor

2.14. Configuring the Raspberry Pi for Bonjour Conformance Test (BCT)

Note: the folder `/home/pi/Raspi/Tools/` contains the necessary BCT configuration files. It is copied to the Raspberry Pi as part of the initial setup.

Prepare for the BCT on the Raspberry Pi, by going through the following commands.

On the Mac, open a terminal window and change the current directory to POSIX and provision the `LightbulbLED` sample with a setup code in a new directory:

```
./Raspi/Tools/Provision --category 5 pi@raspberrypi.local:/home/pi/acme-lightbulb/bin/.HomeKitStore
```

Call `make clean` in directory `~/Applications/LightbulbLED` before continuing if the lightbulb example has been run previously. Then, build the `LightbulbLED` sample:

```
sudo make -C /home/pi/Raspi/Applications/LightbulbLED/
USE_BCT_NAME_CHANGE=1
```

Configure `LightbulbLED` as a service that is automatically restarted on power on:

```
sudo cp /home/pi/Raspi/Applications/LightbulbLED/LightbulbLED /home/pi/
acme-lightbulb/bin

sudo cp /home/pi/Raspi/Tools/helpers/Pi/lib/systemd/system/acme-
lightbulb.service /lib/systemd/system/

sudo systemctl daemon-reload

sudo systemctl enable acme-lightbulb.service

sudo systemctl start acme-lightbulb.service
```

`Prevent host-local.service from interfering with acme-lightbulb.service during BCT:`

```
sudo systemctl stop host-local.service

sudo systemctl disable host-local.service
```

`Reboot Raspberry Pi:`

```
sudo reboot
```

Note: As a shortcut for the above commands (starting from sudo cp ...), the following script can be run:

`/home/pi/Raspi/Tools/helpers/Pi/setup-lightbulb-for-bonjour-conformance.sh`

Note: This script requires that directory `home/pi/acme-lightbulb/bin` already exists, prior to execution. The directory was created as part of the `Provision` script that should have run before, as described.

-
-

Note: During Bonjour Conformance Test, Raspberry Pi's hostname and IP address are changed several times. Therefore the Raspberry Pi might not be accessible via its hostname during and after the test, and a reboot is recommended after the test. During reboot, the Raspberry Pi's hostname is reset to the previously supplied hostname.

Note: Running `LightbulbLED` as a Linux service that automatically starts upon reboot is necessary for BCT and for a licensed accessory. However, during development it is inconvenient. To stop and unconfigure autostart of `LightbulbLED` as a service, use the following commands:

```
sudo systemctl stop acme-lightbulb.service

sudo systemctl disable acme-lightbulb.service
```

When the `LightbulbLED` service is disabled, the `host-local.service` should be restarted in order to re-enable hostname resolution; otherwise Raspberry Pi will not be accessible via its hostname and `ssh pi@raspberrypi.local` will fail. To restart `host-local.service`, use the following commands:

```
sudo systemctl enable host-local.service

sudo systemctl start host-local.service

sudo reboot
```

Note: One of the Bonjour Conformance Tests requires to change the name of the Bonjour service. For that reason, we use the option USE_BCT_NAME_CHANGE=1 for building the `LightbulbLED` this time. As a consequence, signal `SIGUSR2` is not used for factory reset anymore, but for changing the name from "Acme Lightbulb with LED" to "New - Bonjour Service Name". To perform the factory reset procedure, use the command `sudo rm -r .HomeKitStore/ in /home/pi/acme-lightbulb/bin`, run the `Provision` script again, and reboot the Raspberry Pi.

Note: To avoid confusion, make sure not to run the `LightbulbLED` as a service and from the command line at the same time.

2.15. Executing the Bonjour Conformance Test

Prepare environment (Mac and AirPort) and execute BCT according to the *Bonjour Conformance Test for HomeKit* document.

For a manual name change during BCT, issue the following command on the Raspberry Pi:

```
/home/pi/Raspi/Tools/helpers/Pi/rename-lightbulb.sh
```

For a cable change test during BCT on Wi-Fi interface `wlan0`, issue the following command on the Raspberry Pi:

```
/home/pi/Raspi/Tools/helpers/Pi/wlan0-down-sleep-11sec-wlan0-up.sh
```

Note: To find out the IP address of the Raspberry Pi on the network, while the acme-lightbulb.service is running, issue the following command on the Mac:

```
ADK/SDK-Samples/POSIX/Raspi/Tools/helpers/Mac/hap-sd
```

This command will output the Raspberry Pi's IP address, e.g.:

```
Acme LightbulbLED:

  169.254.169.50
```

To open an ssh connection to the Raspberry Pi with an IP address instead of the host name, issue the following command on the Mac:

```
ssh pi@<IP-Address>

e.g. ssh pi@169.254.169.50
```

3. Raspberry Pi as IP Camera Example

Beyond simple HomeKit profiles such as lightbulbs, a Raspberry Pi 3 may also be used as a demonstrator for a HomeKit IP camera. Its purpose is to show camera (or video doorbell) accessory developers not familiar with HomeKit how to quickly achieve HomeKit compatibility, so that they can focus mostly on their actual audio/video streaming code. In fact, the ADK also provides the key element of such an A/V pipeline as an option, namely an implementation of the SRTP protocol. The remaining elements - video capture and encoding, audio capture and encoding, audio decoding - are implemented in the demonstrator using the

GStreamer multimedia framework. By replacing these elements for a target platform, accessory developers can remove any dependency on GStreamer if desired.

Note: The Raspberry Pi is used as a reference platform for demonstration purposes, it is not meant as a production platform. Also, it is the MFi licensees responsibility to adapt and possibly optimize the reference IP Camera implementation provided in the ADK to the target platform as needed.

To get started with the camera demonstrator:

- Make sure that the lightbulb sample works as described earlier.
- Obtain the following hardware:
  - A Raspberry Pi Camera (https://www.adafruit.com/product/3099)
  - A microphone:
    - It is easiest to connect a USB microphone like this one: https://www.adafruit.com/product/3367.
    - More representative for a real IP camera would be an I2S microphone like this one: https://www.adafruit.com/product/3421 or an I2S sound card like this one: http://www.audioinjector.net. However, I2S microphones are far more difficult to install. In contrast, a USB microphone just works when plugged-in.
  - A headphone connected to the headphone jack for audio output. Do not use a headset with a microphone because the microphone contact is misused as a primitive video output. Better quality can be obtained if the HDMI audio output it used instead of the headphone jack. A HDMI to audio adapter is needed, such as this one: https://www.adafruit.com/product/3048.
    - Alternatively, a small "USB sound card" can be used for audio input and output (for instance https://www.adafruit.com/product/1475). No special installation is needed.
    - If a non-standard audio output is used (I2S or USB), the ALSA configuration in `HAPPlatformCamera.c` must be changed to `#define kAlsaOutputDevice "plughw:1"`.
- Use `make debug` in the `Applications/IPCamera` directory to build the application.
- Use `./IPCamera` to run the application.

4. Raspberry Pi as Remote Example

A Raspberry Pi 3 may also be used to demonstrate a remote control accessory. A Remote accessory requires a HomeKit hub: an Apple TV, an iPad connected to your home Wi-Fi network or a HomePod. Here, an Apple TV is assumed as the hub, an iPhone as the iOS controller, and a Raspberry Pi as the `Remote` accessory. The `RemoteSimple` sample allows controlling an Apple TV with configurable buttons. The `Remote` sample additionally contains support for Siri.

4.1. Device setup

iPhone setup:

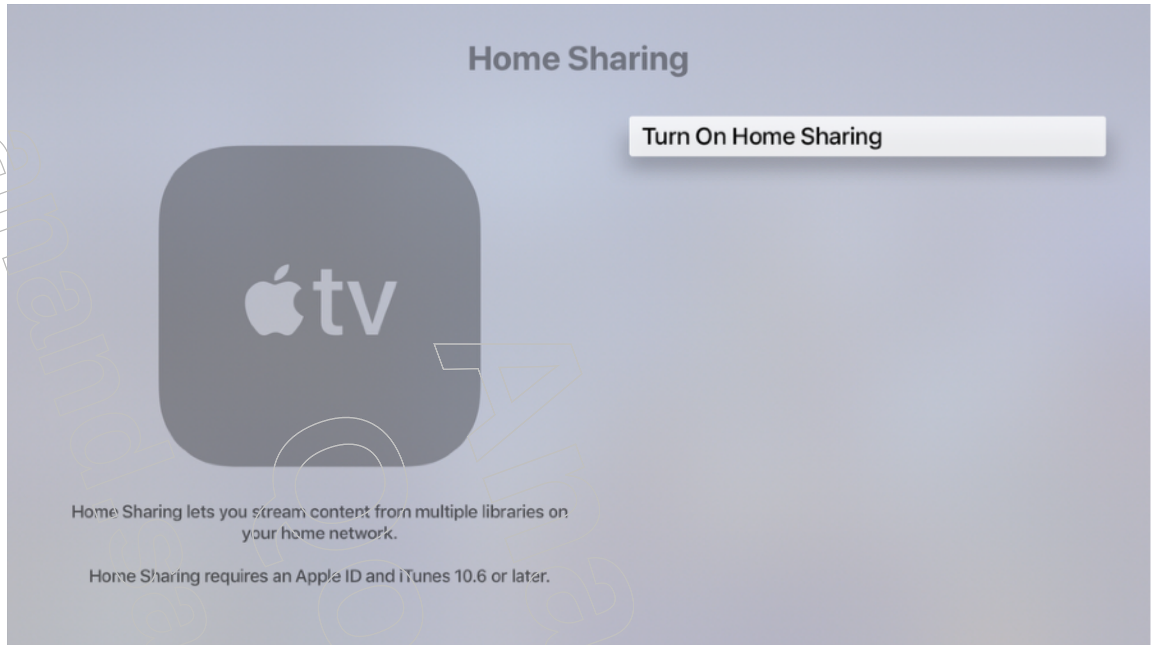
- Make sure that the iPhone runs iOS 12 or later
- Set the region to the United States and the language to English

- Set up a HomeKit home

Apple TV setup:

- Make sure that the Apple TV runs tvOS 12 or later
- Assign the Apple TV to the same Wi-Fi network as the iPhone, to the same Apple ID (iCloud ID) as the iPhone, and to the same home as the iPhone
- Set the region to the United States and the language to English
- Enable AirPlay

- Turn on `Home Sharing`



Make sure that iCloud and `Home Sharing` are connected to the same iCloud account.

Note: The remote supports connecting to multiple Apple TVs in the same network.

Remote setup without Siri:

- Generate and deploy a setup code for the `RemoteSimple` application:
  - `./Raspi/Tools/Provision --wac --category 32 pi@raspberrypi.local:~/Raspi/Applications/RemoteSimple/.HomeKitStore`
- Make sure that the accessory is connected to the same network as the controller and the Apple TV to be used.
- Use `make debug` in the `Applications/RemoteSimple` directory to build the application.
- Use `./RemoteSimple` to run the application.

Remote setup with Siri:

- If you want to use Siri, obtain a microphone for the Raspberry Pi:
  - It is easiest to connect a USB microphone like this one: https://www.adafruit.com/product/3367.
  - More representative for a real remote control would be an I2S microphone like this one: https://www.adafruit.com/product/3421 or an I2S sound card like this one: http://www.audioinjector.net. However, I2S microphones are far more difficult to install. In contrast, a USB microphone just works when plugged-in.
- Generate and deploy a setup code for the `Remote` application:
  - `./Raspi/Tools/Provision --wac --category 32 pi@raspberrypi.local:~/Raspi/Applications/Remote/.HomeKitStore`

- Make sure that the accessory is connected to the same network as the controller and the Apple TV to be used.
- Use `make debug` in the `Applications/Remote` directory to build the application.
- Use `./Remote` to run the application.

Pairing:

- Pair the remote with the iPhone
- Assign the remote to the same Home and Room as the Apple TV

4.2. Usage

The `Remote` application captures keyboard input from the console to emulate key presses. Key up and key down events are sent separately to emulate physical button events and to allow for easier testing. The following key configuration is configured in the remote:

Action	Down	Up
Siri	1	2
Menu	q	w
Play / Pause	e	r
TV / Home	t	y
Select	u	i
Arrow Up	a	s
Arrow Right	d	f
Arrow Left	g	h
Volume Up	j	k
Volume Down	i	;
Power	c	v
Generic	b	n

Before key presses can be sent, the remote needs to be paired with a controller that has at least one Apple TV configured. If multiple Apple TVs are configured, the "`m`"- key can be used to switch between the configured Apple TVs. The "`0`"-key is used to emulate a remote that controls a non-HomeKit entity.

Action	Key
Toggle Identifier / Apple TV	m
Set target to non HomeKit	0

Note: These key configurations also work for the remote bridge. Remote 1 will react to the key presses that are discussed above. Remote 2 reacts to the same key presses with the additional `Shift` key pressed (assumption: US/`International` keyboard layout).

4.3. Background information

Apple TVs are identified by an integer number. The `Active Identifier` denotes the currently active identifier, e.g., Apple TV. Each Apple TV will register itself with an identifier and the associated key configuration.

If the active identifier is set to `0`, the device indicates that it is either controlling a non-HomeKit entity or that no Apple TV has been configured yet. In this case the following log message will be shown:

```
2018-10-17'T'18:12:34'Z'    Debug    [com.apple.mfi.HomeKit.Remote:Remote]
RemoteRaiseButtonEvent: No active identifier is set for the remote registered with
remote 0x86388.
```

Note: If this message occurs even tough the `Remote` has been paired and an Apple TV is configured, then verify the Apple TV/iPhone/Remote configuration discussed earlier.

Appendix B: Change History and Migration Guide

This appendix describes what has changed between ADK releases and how to migrate between two subsequent releases.

Whenever you move from one ADK version to another, do the following:

- If you have built accessory logic against an earlier version of the ADK, completely replace the following directories by the ones provided with the new version:
  - `ADK/HAP/include`
  - `ADK/HAP/lib`
  - `ADK/PAL/include`
  - `ADK/Tools`
- If your accessory software is based on a reference platform, migrate the changes from `ADK/SDK–Samples` as well. Especially the PAL subdirectory and the files `App.c`, `DB.c` and `Main.c` should be carefully reviewed.

1. From ADK 1.2 to ADK 2.0

ADK 2.0 is compatible with HomeKit Accessory Protocol Specification R13 and Accessory Interface Specification R29.

1.1. How to migrate

- The sample implementation of an IP Camera for Raspberry Pi 3 has been improved (see below). This is more relevant for new camera accessories, so it is *not* recommended that licensees with camera code based on ADK 1.1 migrate to the new sample code architecture. Note that the `HAPPlatformCamera.h` API has not been changed. The newly introduced PAL modules `HAPPlatformAudio.h` and `HAPPlatformVideo.h` are not relevant in this case.

1.2. New features and improvements

- The new Apple TV `Remotes` profile is supported. See Remote.
- The sample implementation of an IP Camera for Raspberry Pi 3 has been made more portable, by making `HAPPlatformCamera.c` largely generic, by using the newly introduced `HAPPlatformAudio.h` and `HAPPlatformVideo.h` interfaces. This reduces the work needed for a new IP Camera implementation.
- Improved implementation of `AccessorySetupGenerator`.
- Added 64-bit Windows support to random number generator used by "AccessorySetupGenerator.exe."
- Added toolchain `mips16el-gcc4.4.2-uclibc0.9.30-linux-gnu`.

1.3. Key fixes

- Error correction in JSON serialization of characteristic manufacturer description.
- Corrected handling of `Configuration Number` update after a firmware update / bridge configuration change.
- Extended timeout for BLE requests to be processed on the controller side.
- Improved compatibility of C `stdlib` by removing usage of `%llu`.
- Always send optional `HAP-Characteristic-Configuration-Param-Broadcast-Interval`, even when default interval is configured.
- Fixed misaligned memory accesses on ARMv5-TE processors.
- Corrected memory alignment code for MSVC-based HAP Library binaries (Windows CE).
- Fixed error response when IP camera streaming is started on a busy service.
- Added a workaround for BLE reference platform `HAPPlatformKeyValueStore` implementation to avoid concurrency issues when key-value store is enumerated while a key is being deleted.

1.4. Known limitations

- The `IPCamera` sample for Raspberry Pi has limitations related to the open-source plug-ins used with GStreamer:
  - Only one A/V stream is supported.
  - Audio/video performance, memory and stability issues.

Glossary

accessory Any product able to interface with an Apple device via the HomeKit Accessory Protocol.

accessory attribute database A data structure that reflects the current state of an accessory's attributes, as it can be read or written via the HomeKit Accessory Protocol.

accessory development Platform development plus accessory logic development.

accessory development kit (ADK) Apple's implementation of HAP for accessories, which can be used by chipset vendors or accessory manufacturers to enable HomeKit on their products. It implements the server side of HAP.

accessory logic The domain-oriented "business logic" of an accessory, e.g., a thermostat's temperature control code.

accessory manufacturer Company that is an HomeKit MFi licensee and provides HomeKit-enabled accessories. For HomeKit software development it may use a systems integrator that is a HomeKit MFi developer licensee, for production it may use a HomeKit MFi manufacturing licensee.

accessory object A data structure that represents a physical accessory and contains its accessory attribute database. Most accessories contain exactly one accessory object, while a HomeKit bridge contains one accessory representing itself, plus a number of accessory objects that represent the non-HomeKit accessories that they are bridging to.

accessory server A HomeKit accessory server is server software an an accessory that manages one or several accessory objects and implements HAP. The HAP Library in the ADK contains the code for an accessory server that supports HAP over IP and HAP over BLE.

bridge A HomeKit accessory that gives a HomeKit controller access to non-HomeKit accessories.

characteristic Variables or procedures of an accessory that can be accessed from an iPhone or similar device via a network. For example, the On characteristic of a lightbulb accessory.

chipset vendor Company that provides microcontroller, microprocessor or communication chipsets and whose customers - accessory manufacturers - would benefit from HomeKit support for these products. Such support may range from the implementation of a single PAL module, e.g., for a Wi-Fi chipset, to a full HomeKit SDK containing a complete PAL implementation, e.g., for a system-on-chip product.

controller A HomeKit controller is an Apple product that implements the client side of HAP. Currently this is an iPhone, iPad, iPod, Watch, Apple TV or HomePod.

Bonjour An Apple-defined protocol that enables a HomeKit controller to automatically find HomeKit accessories in a local IP network (Ethernet or Wi-Fi).

HAP Library Apple's implementation of HAP, provided in binary form. The ADK contains HAP libraries for the most popular toolchains. A HomeKit SDK contains a subset of these libraries, possibly only one, that is relevant for the supported platform.

HomeKit Accessory Protocol (HAP) Pairing, session and security protocols specified by Apple, with the goal of providing security and interoperability between a HomeKit controller on the one hand, and HomeKit accessories on the other hand. There are two main variants of HAP: *HAP over BLE* (for Bluetooth Low

Energy) and *HAP over IP* (for Ethernet and Wi-Fi). In addition, the complementary protocol *HAP over iCloud* allows remote access to an accessory over the Internet and via an Apple TV, iPad or HomePod.

HomeKit Data Streams (HDS) An extension of the original HomeKit Accessory Protocol for reliable data streams within HomeKit.

HomeKit SDK At its core, a HomeKit SDK contains a HAP implementation plus a PAL for the type of platform that the SDK should support. A HomeKit SDK typically provides accessory logic samples in addition, plus a toolchain configuration with make files, support for a secure firmware update mechanism, etc.

HomeKit system service HomeKit service that is needed for HomeKit to work correctly and is not related to any specific profile.

hub A HomeKit hub is a HomeKit controller that supports remote access to HomeKit accessories via Internet. It also enables Remote accessories to use Apple's Siri service.

IID Every characteristic and service of a HomeKit accessory has a unique IID, which is a number from 1 to $2^{64} - 1$ for IP accessories, and $2^{16} - 1$ for BLE accessories. An IID must remain stable for the entire lifetime of a HomeKit pairing, i.e., it must never change its value, even across firmware updates. An IID that is not needed anymore, because its service or characteristic is not supported after a firmware update anymore, must not be reassigned to another characteristic or service.

licensed accessory A HomeKit accessory that has completed all MFi Program certification requirements.

platform A combination of hardware, software and toolchain that is able to support HomeKit, e.g., a microcontroller with FreeRTOS and the GCC toolchain.

platform abstraction layer (PAL) A set of modules that make it possible to run a single HAP implementation on a variety of platforms. The PAL modules have platform-independent C interfaces, defined in ADK header files. Accessory makers can use the PALs that are contained in their chipset vendors' HomeKit SDKs, or, if no such SDK is available, they can develop PALs of their own using the ADK.

platform development Developing a platform, and developing or using a PAL for this platform.

POSIX At its core, a standardized operating system interface intended to increase software portability.

profile A HomeKit profile is a set of related services, e.g., the `IP Camera profile` requires the implementation of the Camera `RTP Stream Management` service plus the `Microphone` service, and optionally the `Speaker` service. All profiles require the implementation of the `Accessory Information` service. All BLE accessories additionally require the implementation of a pairing service. Most profiles consist of only profile-specific service, e.g., the `Lightbulb` service.

sample SDK The ADK provides a few sample SDKs, which are intended for illustration purposes, to serve as starting points for the development of new PALs or full HomeKit SDKs. A sample SDK contained in the ADK may be comprehensive like the POSIX SDK, or focus on more specific topics, such as the PAL for BLE chipsets.

service A HomeKit service is a set of related characteristics, e.g., the `Lightbulb` service defines the On, `Brightness`, Hue, `Name` and `Saturation` characteristics. A service may define constraints on how the involved services and characteristics must behave.

setup code A randomly generated code that is printed on an accessory or label that comes with an accessory, in the form XXX-XX-XXX. Its purpose is to prevent man-in-the-middle attacks during initial pairing of an accessory with a HomeKit controller. For newer accessories, a setup ID and a QR code are printed instead of the setup code, to increase convenience of the secure pairing process.

setup ID A randomly generated code that is printed on an accessory or label that comes with an accessory, in the form XXXX, along with a QR code that is derived from the setup ID and a setup code. Their purpose is to make secure pairing easy to use. Setup IDs are not allowed to be generated on accessories and they must be preserved across factory resets.

setup payload Data derived from a setup code and setup ID and printed as a QR code on an accessory or label that comes with an accessory, along with the setup code. Their purpose is to make secure pairing easy to use.

SRP salt Random value used to make attacks on the SRP algorithm and the setup code more difficult.

SRP Secure Remote Password (SRP) is an algorithm that is used in HomeKit for the secure pairing of an accessory with a HomeKit controller.

SRP verifier Value derived from a setup code and SRP salt used to make attacks on the SRP algorithm and the setup code more difficult. Except for accessories with programmable NFC, only the SRP verifier is stored persistently, not the setup code.

upcall A function call from a library using a function pointer to invoke code that is not known to the library developer. Apple's HAP library uses callbacks to the accessory logic, e.g., to change the set point of a thermostat.

Wi-Fi Accessory Configuration 2 (WAC2) An Apple-defined protocol that allows a HomeKit controller to pair with a new HomeKit accessory over Wi-Fi so that the user does not have to manually enter a Wi-Fi SSID and password.

Apple Inc.

Apple Inc.
One Apple Park Way
Cupertino, CA 95014
408-996-1010