Sample Code

Configuring a home automation device

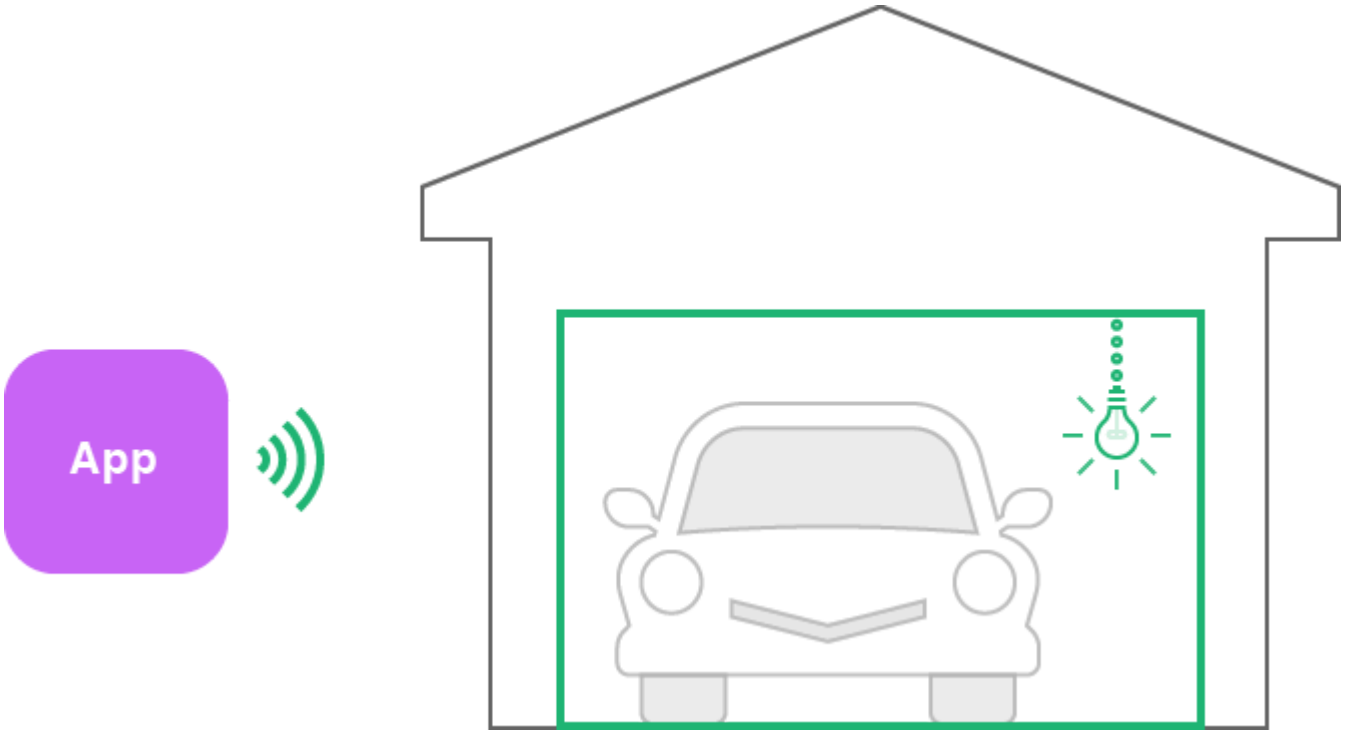
Give users a familiar experience when they manage HomeKit accessories.

Download

iOS 12.1+ | iPadOS 12.1+ | Xcode 10.2+

Overview

This sample presents a simplified version of the kind of app a HomeKit-enabled accessory manufacturer might provide. You use the app to configure and control a specific device—a garage door opener from a fictional manufacturer, Kilgo Devices. The app offers access to secondary and custom characteristics that Apple's Home app doesn't expose. It provides a user experience that's consistent with the Home app's approach and terminology, but doesn't attempt to replicate every feature of the Home app.



For more information about user interface considerations for HomeKit-enabled apps, see the HomeKit section of the Human Interface Guidelines.

Configure the sample code project

To be able to use HomeKit, you enable the HomeKit capability and include the `NSHomeKitUsageDescription` key in your app's `Info.plist` file, as described in Enabling HomeKit in Your App. In this sample app, the capability is already enabled and a usage description is provided.

Perform the following steps before building and running the app:

1. Set a valid signing team in the target's General pane so that Xcode can create a provisioning profile containing the HomeKit entitlement when you build for the first time.
2. Download and install the Home Accessory Simulator (HAS) on your Mac to be able to emulate HomeKit-enabled accessories. See Testing Your App with the HomeKit Accessory Simulator.
3. Import the `hasaccessory` file that the sample app bundles to define the specific garage door opener accessory that the app controls. Choose `File > Import Accessory` from the HAS menu. In the dialog that appears, navigate to the downloaded sample code project's `Documentation` folder, and select the `garage.hasaccessory` file.

The import creates a single accessory with a hidden accessory information service that all accessories have, and two user-interactive services: one that controls a garage door and another to control an attached light bulb. Most of the associated characteristics are standard for their service. Only one—the light bulb's fade rate—is custom. You can use HAS to inspect and manipulate all of these items.

Create a home manager and get the primary home

You always use an instance of `HMHomeManager` as the root HomeKit object. The home manager contains an array of homes, each of which has a collection of accessories. The sample app defines a `HomeStore` class for use as a singleton that holds the one and only home manager for the app:

```
class HomeStore: NSObject {
    /// A singleton that can be used anywhere in the app to access the home manager.
    static var shared = HomeStore()

    /// The one and only home manager that belongs to the home store singleton.
    let homeManager = HMHomeManager()

    /// A set of objects that want to receive home delegate callbacks.
    var homeDelegates = Set<NSObject>()
```

```
    /// A set of objects that want to receive accessory delegate callbacks.
    var accessoryDelegates = Set<NSObject>()
}
```

You create an accessory list collection view controller to present the list of connected accessories. Because this root view controller never gets deallocated, it can safely assign itself as the `HMHomeManagerDelegate` protocol delegate:

```
HomeStore.shared.homeManager.delegate = self
```

The manager tells its delegate when the list of homes changes, including the first time the home manager loads data from the HomeKit database during initialization. When this happens, the accessory list reloads to show the accessories in the primary home, or prompts the user to create a new home if none exists.

You can extend the app to allow the user to select among all known homes instead of always choosing the primary home. You can also allow users to add, remove, or rename homes, although the user performs these tasks infrequently, and typically relies on the Home app to do so.

Add new accessories

The first time you run the app, the accessory list is empty, because you haven't associated any accessories from Kilgo Devices. The app's UI presents a + button in the navigation bar that you tap to initiate a search for accessories on the local network. The button's tap handler calls the home's `addAndSetupAccessories(completionHandler:)` method.

```
home?.addAndSetupAccessories(completionHandler: { error in
    if let error = error {
        print(error)
    } else {
        // Make no assumption about changes; just reload everything.
        self.reloadData()
    }
})
```

This presents the standard HomeKit UI for locating and adding new accessories to a given home. On success, the completion handler refreshes the app's copy of the HomeKit data and redraws the display.

When your app enters the standard accessory association flow, which is the same one that the Home app uses, the user follows these steps:

1. **Scan or enter the new device's HomeKit setup code.** This code comes packaged with the device, or is available in the HAS display for the accessory.
2. **Select from a list of devices on the network that don't already have an existing HomeKit association.** This includes both real and simulated devices that can access the local area network.
3. **Wait for HomeKit to verify that the setup code entered in step 1 matches the device's code.**
4. **Assign a name and room to each service associated with the device.** HomeKit provides a default name and room for each service, one page per service, that the user can accept or change.

Each item that the user names in step 4 appears in the Home app as an "accessory". However, in HomeKit, these are `HMService` instances. They are owned by an `HMAccessory` instance that represents the physical device that you selected in step 2. To maintain a user experience consistent with the Home app, the sample app UI (and the rest of this article) also refers to each `HMService` instance as an accessory.

Show only accessories relevant to your app

You draw the display based on a copy of data from HomeKit captured into an array of Kilgo services:

```
var kilgoServices = [HMService]()    // These are called "accessories" in the UI.
```

When the accessory list reloads, either because a new home is set or because the accessory-add flow completes, you populate the above array, filtering out `HMAccessory` instances from manufacturers other than Kilgo, and `HMService` instances that aren't user interactive. Also while gathering the list, you request notifications for changes on any of the corresponding characteristics, which are the individual points of status and control for a given service:

```
for accessory in home.accessories.filter({ $0.manufacturer == "Kilgo Devices, Inc."
    accessory.delegate = HomeStore.shared

    for service in accessory.services.filter({ $0.isUserInteractive }) {
        kilgoServices.append(service)

        // Ask for notifications from any characteristics that support them.
        for characteristic in service.characteristics.filter({
            $0.properties.contains(HMCharacteristicPropertySupportsEventNotificatior
        }) {
            characteristic.enableNotification(true) { _ in }
```

```
        }
    }
}
```

As a result, the display shows only the accessories relevant to this particular app.

> **Note**
>
> Use care when crafting your filters. For example, the name “Kilgo Devices, Inc.” might not be unique among all past and future manufacturers, and therefore might not serve as a sufficient predicate in a real app.

Tailor common interactions to specific accessories

Accessories (like light bulbs) have characteristics (like power state, color temperature, brightness, and so on) that users control or observe. Users typically care about one of these characteristics above the others, because they change or read that characteristic most often. This is the primary characteristic, and you should give users quick access to it. For a light bulb, users most often want to switch it on or off, so the power state is the primary characteristic.

It’s up to you to define what the primary charactersitc is for the accessories you control. You can do that by creating a computed property on `HMService` in an extension that returns the type of primary characteristic:

```
var primaryControlCharacteristicType: String? {
    switch kilgoServiceType {
    case .lightBulb: return HMCharacteristicTypePowerState
    case .garageDoor: return HMCharacteristicTypeTargetDoorState
    case .unknown: return nil
    }
}
```

Then use this primary characteristic type to locate and return the characteristic that has that type:

```
var primaryControlCharacteristic: HMCharacteristic? {
    return characteristics.first { $0.characteristicType == primaryControlCharacteri
}
```

For Kilgo Devices, both the light bulb and garage door have binary primary state. The bulb is on or off. The target state of the door is open or closed. This lends itself to an interface where a toggle

switch is sufficient to control all primary characteristics. You can implement this as the tap handler on each item in the accessory list collection view. When the user taps the accessory, you read the current characteristic value and then write the opposite:

```
func tap() {
    if let characteristic = service?.primaryControlCharacteristic,
        let value = characteristic.value as? Bool {

        // Provide visual feedback that the item was tapped.
        bounce()

        // Write the new value to HomeKit.
        characteristic.writeValue(!value) { error in
            self.redrawState(error: error)
        }
    }
}
```

The write involves network access, so HomeKit calls a completion handler when the write completes. Use this opportunity to update the state of the interface, as shown in the snippet above.

Enable custom configuration

When the user taps an accessory's information button, the app reveals details about the accessory. From the detail view, the user can rename the accessory, assign it to a room, remove it from the home, and see device information, like the firmware version. The user can also tap Settings to reveal a list of secondary characteristics for that accessory.

Control the user experience by presenting only relevant characteristic types. The `KilgoService` extension of `HMService` defines a computed property that limits the list of displayable characteristics to those in a curated list:

```
var displayableCharacteristics: [HMCharacteristic] {
    let characteristicTypes = [HMCharacteristicTypePowerState,
                               HMCharacteristicTypeBrightness,
                               HMCharacteristicTypeHue,
                               HMCharacteristicTypeSaturation,
                               HMCharacteristicTypeTargetDoorState,
                               HMCharacteristicTypeCurrentDoorState,
                               HMCharacteristicTypeObstructionDetected,
```

```
                                HMCharacteristicTypeTargetLockMechanismState,
                                HMCharacteristicTypeCurrentLockMechanismState,
                                KilgoCharacteristicTypes.fadeRate.rawValue]

    return characteristics.filter { characteristicTypes.contains($0.characteristicTy
}
```

These are mostly HomeKit standard types, all of which are applicable to Kilgo devices. There's also one custom type—fade rate—defined earlier in the same extension:

```
enum KilgoCharacteristicTypes: String {
    case fadeRate = "7E536242-341C-4862-BE90-272CE15BD633"
}
```

Characteristic types are stored as UUID strings. The value specified in the code for fade rate matches the value found in the accessory simulator, which you can inspect in HAS. If you also build a real Kilgo device, the value used there would have to match as well.

See Also

Home Manager

Testing your app with the HomeKit Accessory Simulator

Install the HomeKit Accessory Simulator to help you debug your HomeKit-enabled app.

`class` HMHomeManager

The manager for a collection of one or more of a user's homes.