Article

# Discovering and configuring accessories

Detect nearby accessories and facilitate their setup.

## Overview

Use the AccessorySetupKit framework to simplify discovery and configuration of Bluetooth or Wi-Fi accessories. This allows the person using your app to use these devices without granting overly-broad Bluetooth or Wi-Fi access.

To discover accessories and present them in your app:

1. Declare that your app uses AccessorySetupKit in its information property list.
2. In your app, create and activate an instance of `ASAccessorySession`.
3. Provide information about your supported accessories to display a picker. This lets the person using your app discover and select nearby accessories to configure.
4. When the picker sends an accessory added event, use information about the selected device to create a Bluetooth or Wi-Fi connection.

## Declare your app's accessories

To prepare your app to discover accessories, add the `NSAccessorySetupKitSupports` key to its information property list. Set its value to an array of strings that contains one or more of the following values:

`Bluetooth`
: Add this value if your app discovers accessories using Bluetooth or Bluetooth Low Energy.

`WiFi`
: Add this value if your app discovers accessories by finding Wi-Fi SSIDs that the accessories publish.

If you add `Bluetooth` to the list of supported protocols, you also need to add the following keys and values to your app's information property list:

`NSAccessorySetupBluetoothCompanyIdentifiers`
: An array of strings that represent the Bluetooth company identifiers for accessories your app configures.

`NSAccessorySetupBluetoothNames`
: An array of strings that represent the Bluetooth device names for accessories your app configures.

`NSAccessorySetupBluetoothServices`
: An array of strings that represent the hexadecimal values of Bluetooth services for accessories your app configures.

> **Important**
>
> If your app tries to discover Bluetooth accessories during setup without supplying these keys and values, or uses identifiers, names, or services that it doesn't include in its information property list, the app crashes. This only affects use of AccessorySetupKit discovery and selection; you can use other services and properties on the accessory after the person using the app selects it.

## Activate a discovery session

The `ASAccessorySession` is how your app uses the AccessorySetupKit framework. In your app, you create an instance of `ASAccessorySession` and activate it to receive callbacks with events as the session processes events. The `activate(on:eventHandler:)` method takes a `DispatchQueue` and an event-handling block or closure. The callbacks occur on the provided queue, which defaults to `main`.

The event handler receives a single parameter of type `ASAccessoryEvent`, which has an `eventType` that you use to determine what to do with each callback. For example, shortly after activating the session, your callback receives the `ASAccessoryEventType.activated` event. The following listing comments on the meaning of these events and how you may want to handle them.

```
let session = ASAccessorySession()
override func viewDidLoad() {
    super.viewDidLoad()

    session.activate(on: DispatchQueue.main) { [weak self] event in
        guard let self else { return }
```

```
        switch event.eventType {
        case .activated:
            // Use previously-discovered accessories in
            // session.accessories, if necessary.
        case .accessoryAdded:
            // Handle addition of an accessory by person using the app.
        case .accessoryRemoved, .accessoryChanged:
            // Handle removal or change of previously-added
            // accessory, if necessary.
        case .invalidated:
            // The session is now invalid and you can't use it further.
        case .migrationComplete:
            // Handle migration.
        case .pickerDidPresent:
            // Update state for picker appearing, if necessary.
        case .pickerDidDismiss:
            // Update state for picker disappearing, if necessary.
        case .unknown:
            // Handle unknown event type, if appropriate.
        @unknown default:
            // Reserve this space for yet-to-be-defined event types.
        }
    }
}
```

## Display an accessory picker

When the session activates, its `accessories` array contains any accessories previously authorized for the app, which you can inspect. To discover new devices, your app needs to show an accessory picker. The person using the app uses this picker to choose the accessory to configure.

> **Note**
>
> If someone renames a previously-discovered Wi-Fi accessory, it becomes discoverable again by the picker.

Create instances of `ASPickerDisplayItem` that describe the items in the session that your app can configure. Collect these items in an array and pass them to the session for the operating system to present a picker:

```
private func showMyPicker() {
    var descriptor = ASDiscoveryDescriptor()
    descriptor.bluetoothServiceUUID = CBUUID(string: MY_ACCESSORY_UUID_STRING) // It

    let displayName = "My Accessory"
    guard let productImage = UIImage(named: MY_ACCESSORY_IMAGE_NAME) else { return }

    var items: [ASPickerDisplayItem] = []
    items.append (ASPickerDisplayItem(name: displayName,
                                      productImage: productImage,
                                      descriptor: descriptor))

    // Create additional picker items if you support multiple accessories
    // with different Wi-Fi SSIDs or Bluetooth service UUIDs.

    session.showPicker(for: items) { error in
        if let error {
            // Handle error.
        } else {
            // Perform any post-picker cleanup.
            // If the picker finished by selecting an item, the event
            // handler receives it as an event of type `.accessoryAdded`.
        }
    }
}
```

Each display item's `descriptor`, a property of type `ASDiscoveryDescriptor`, needs to have a `bluetoothCompanyIdentifier` or `bluetoothServiceUUID`, and at least one of the following accessory identifiers:

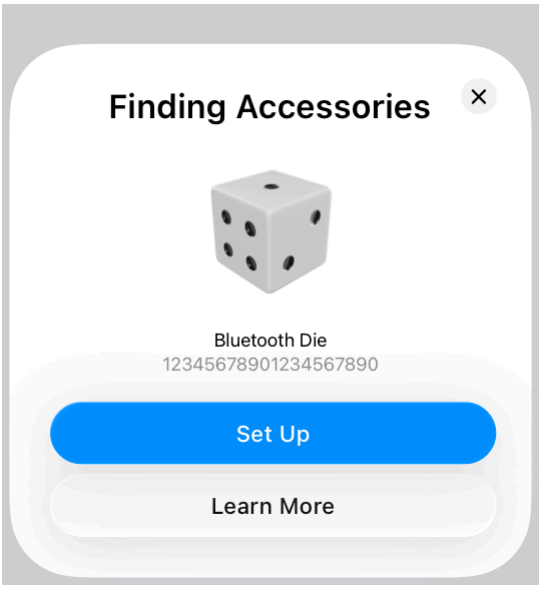
- `bluetoothNameSubstring`
- A `bluetoothManufacturerDataBlob` and `bluetoothManufacturerDataMask` set to the same length.
- A `bluetoothServiceDataBlob` and `bluetoothServiceDataMask` set to the same length.
- Either `ssid` or `ssidPrefix`, which needs to have a non-zero length. Only supply one of these; the app crashes if you supply both.

For Bluetooth accessories, the accessory identifiers you use in display items need to match the values you supply in the app's information property list.

Along with filtering matched accessories to show in the picker, the display item and its descriptor allow you to control certain behaviors of the picker interaction. You can limit the `bluetoothRange` of the descriptor to only match accessories in the immediate physical proximity of the device running the app. To specify behaviors like allowing renaming of the accessory during setup, or confirming accessory authorization before showing the setup view, set the display item's `setupOptions`.

When the picker appears, the person using the app sees a view of all nearby accessories that match the identifiers you provide. When multiple devices match a given identifier, the picker shows a separate item for each unique device. This allows the person to select a single device with the picker.

The following figure shows a single accessory selected in the picker.




## Use the picker when migrating to AccessorySetupKit

You can also perform a one-time migration of previously-configured accessories, which adds them to the AccessorySetupKit framework's list of known accessories. To do this, create instances of `ASMigrationDisplayItem` and include them in the array of items you send to `showPicker(for:completionHandler:)`.

For items you want to migrate, set one or both of the following:

- An `hotspotSSID`, which must be a full SSID and not a prefix.
- An `peripheralIdentifier`, which corresponds to the `identifier` property of the `CBPeer` type.

**Important**

Don't initialize a `CBCentralManager` before migration is complete. If you do, your callback handler receives an error and the picker fails to appear.

## Perform custom filtering

Some apps need to obtain over-the-air (OTA) data from discovered accessories and perform additional filtering before showing them in the picker. The filtering includes tasks like validating the authenticity of an accessory, testing whether it's in pairing mode, and other checks. If your app needs do this, set the picker's display settings to use the `filterDiscoveryResults` option. If you need unlimited time to perform filtering and perform additional actions like downloading product artwork, set the `discoveryTimeout` to `unbounded`.

The following code performs these setup steps:

```
let settings = ASPickerDisplaySettings.default
settings.discoveryTimeout = .unbounded
settings.options.insert(.filterDiscoveryResults)
session.pickerDisplaySettings = settings
```

When the session produces an `ASAccessoryEventType.accessoryDiscovered` event, examine the accessory and determine whether to display it in the picker. To add the accessory to the picker, create a `ASDiscoveredDisplayItem`. Using this type gives you the option to customize the item's display with a specific name and a custom image. Then call `updatePicker(showing:completionHandler:)` on the session to show the customized item.

The following example demonstrates an event handler that inspects discovered accessories and adds customized items to the picker.

```
session.activate(on: .main) { [weak self] event in
    guard let strongSelf = self else { return }
    switch event.eventType {
    case .accessoryDiscovered:
        if let accessory = event.accessory as? ASDiscoveredAccessory {
            if myShouldDisplayAccessory(advertisement: accessory.bluetoothAdvertisem
                                        rssi: accessory.bluetoothRSSI) {
                let item = ASDiscoveredDisplayItem(name: "More Specific Product Name
                                                   productImage: UIImage(named: "Ass
                                                   accessory: accessory)
                session.updatePicker(showing: [item]) { error in
```

```
                    print(error)
                }
            }
        }
    }
}
```

If your custom filtering process requires the app to finish the accessory discovery early, manually end the discovery process by calling `finishPickerDiscovery(completionHandler:)`. If your filtering process didn't add any discovered items to the picker, this call shows a timeout message in the app.

> **Tip**
>
> AccessorySetupKit limits the number of accessories it exposes for discovery. If you don't discover the accessory you expect, manually cause a picker timeout by calling `finishPickerDiscovery(completionHandler:)`. If you want to retry, wait for the `ASAccessoryEventType.pickerDidDismiss` and call `showPicker(for:completionHandler:)` again. In this scenario, you may want to suggest the person using the app verify that the accessory is powered up and nearby.

## Connect and configure the selected device

When the person picks an accessory, the picker sends an event of type `ASAccessoryEventType.accessoryAdded`, followed by an `ASAccessoryEventType.pickerDidDismiss` event when they dismiss the picker. If your app presents its own UI to configure the accessory, wait for the picker to dismiss, then use the accessory from the first event. You can handle this scenario by rewriting your event handler closure from before as follows, storing the accessory on the first event and retrieving it on the second.

```
case .accessoryAdded:
    self.pickedAccessory = event.accessory
case .pickerDidDismiss:
    guard let accessory = self.pickedAccessory else { return }
    self.pickedAccessory = nil
    self.handleAccessoryAdded(accessory)
```

The event's `accessory` property contains details of the selected device, like its `displayName` and an `bluetoothIdentifier` for Bluetooth devices or `ssid` for Wi-Fi. Use this information to connect to the accessory — using Core Bluetooth for Bluetooth or Network Extension for Wi-Fi — and begin your device-specific setup process.

```
private func handleAccessoryAdded(_ accessory: ASAccessory) {
    if let btIdentifier = accessory.bluetoothIdentifier {
        // Use Core Bluetooth to communicate with and configure the accessory.
    }
    else if let ssid = accessory.ssid {
        // Create a `NEHotspotConfiguration` with this SSID to configure.
    }
}
```

Because the app discovered the device with AccessorySetupKit, connecting to the device won't invoke the TCC or other alerts that the system normally shows when using these system frameworks.

Starting in watchOS 26, if your iOS app has a companion watchOS app, the watchOS app can use CoreBluetooth to communicate with a device that someone set up by using AccessorySetupKit in the iOS app. Unlike iOS, however, the watchOS app still shows TCC alerts when connecting.

# See Also

## Essentials

{} Setting up and authorizing a Bluetooth accessory

Discover, select, and set up a specific Bluetooth accessory without requesting permission to use Bluetooth.

`class` ASAccessorySession

A class to coordinate accessory discovery.