Working with a Wi-Fi Accessory

This thread has been locked by a moderator.

167

For important background information, read Extra-ordinary Networking before reading this.

Share and Enjoy

—

Quinn "The Eskimo!" @ Developer Technical Support @ Apple

`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

Working with a Wi-Fi Accessory

Building an app that works with a Wi-Fi accessory presents specific challenges. This post discusses those challenges and some recommendations for how to address them.

Note While my focus here is iOS, much of the info in this post applies to all Apple platforms.

Accessory Categories

I classify Wi-Fi accessories into three different categories.

A **bound accessory** is ultimately intended to join the user's Wi-Fi network. It may publish its own Wi-Fi network during the setup process, but the goal of that process is to get the accessory on to the existing network. Once that's done, your app interacts with the accessory using ordinary networking APIs.

An example of a bound accessory is a Wi-Fi capable printer.

A **stand-alone accessory** publishes a Wi-Fi network at all times. An iOS device joins that network so that your app can interact with it. The accessory never provides access to the wider Internet.

An example of a stand-alone accessory is a video camera that users take with them into the field. You might want to write an app that joins the camera's network and downloads footage from it.

A **gateway accessory** is one that publishes a Wi-Fi network that provides access to the wider Internet. Your app might need to interact with the accessory during the setup process, but after that it's useful as is.

An example of this is a Wi-Fi to WWAN gateway.

Not all accessories fall neatly into these categories. Indeed, some accessories might fit into multiple categories, or transition between categories. Still, I've found these categories to be helpful when discussing various accessory integration challenges.

Do You Control the Firmware?

The key question here is *Do you control the accessory's firmware?* If so, you have a bunch of extra options that will make your life easier. If not, you have to adapt to whatever the accessory's current firmware does.

Simple Improvements

If you *do* control the firmware, I strongly encourage you to:

- Support IPv6
- Implement Bonjour [1]

These two things are quite easy to do — most embedded platforms support them directly, so it's just a question of turning them on — and they will make your life significantly easier:

- Link-local addresses are intrinsic to IPv6, and IPv6 is intrinsic to Apple platforms. If your accessory supports IPv6, you'll always be able to communicate with it, regardless of how messed up the IPv4 configuration gets.
- Similarly, if you support Bonjour, you'll always be able to find your accessory on the network.

[1] Bonjour is an Apple term for three Internet standards:

- RFC 3927 Dynamic Configuration of IPv4 Link-Local Addresses
- RFC 6762 Multicast DNS
- RFC 6763 DNS-Based Service Discovery

WAC

For a bound accessory, support Wireless Accessory Configuration (WAC). This is a relatively big ask — supporting WAC requires you to join the MFi Program — but it has some huge benefits:

- You don't need to write an app to configure your accessory. The user will be able to do it directly from Settings.
- If you do write an app, you can use the `EAWiFiUnconfiguredAccessoryBrowser` class to simplify your configuration process.

HomeKit

For a bound accessory that works in the user's home, consider supporting HomeKit. This yields the same onboarding benefits as WAC, and many other benefits as well. Also, you can get started with the HomeKit Open Source Accessory Development Kit (ADK).

Bluetooth LE

If your accessory supports Bluetooth LE, think about how you can use that to improve your app's user experience. For an example of that, see *SSID Scanning*, below.

Claiming the Default Route, Or Not?

If your accessory publishes a Wi-Fi network, a key design decision is whether to stand up enough infrastructure for an iOS device to make it the default route.

IMPORTANT To learn more about how iOS makes the decision to switch the default route, see The iOS Wi-Fi Lifecycle and Network Interface Concepts.

This decision has significant implications. If the accessory's network becomes the default route, most network connections from iOS will be routed to your accessory. If it doesn't provide a path to the wider Internet, those connections will fail. That includes connections made by your own app.

Note It's possible to get around this by forcing your network connections to run over WWAN. See *Binding to an Interface* in Network Interface Techniques and Running an HTTP Request over WWAN. Of course, this only works if the user *has* WWAN. It won't help most iPad users, for example.

OTOH, if your accessory's network doesn't become the default route, you'll see other issues. iOS will not auto-join such a network so, if the user locks their device, they'll have to manually join the network again.

In my experience a lot of accessories choose to become the default route in situations where they shouldn't. For example, a bound accessory is never going to be able to provide a path to the wider Internet so it probably shouldn't become the default route. However, there are cases where it absolutely makes sense, the most obvious being that of a gateway accessory.

Acting as a Captive Network, or Not?

If your accessory becomes the default route you must then decide whether to act like a captive network or not.

IMPORTANT To learn more about how iOS determines whether a network is captive, see The iOS Wi-Fi Lifecycle.

For bound and stand-alone accessories, becoming a captive network is generally a bad idea. When the user joins your network, the captive network UI comes up and they have to successfully complete it to stay on the network. If they cancel out, iOS will leave the network. That makes it hard for the user to run your app while their iOS device is on your accessory's network.

In contrast, it's more reasonable for a gateway accessory to act as a captive network.

SSID Scanning

Many developers think that TN3111 iOS Wi-Fi API overview is lying when it says:

iOS does not have a general-purpose API for Wi-Fi scanning

It is not.

Many developers think that the Hotspot Helper API is a panacea that will fix all their Wi-Fi accessory integration issues, if only they could get the entitlement to use it.

It will not.

Note this comment in the official docs:

`NEHotspotHelper` *is only useful for hotspot integration. There are both technical and business restrictions that prevent it from being used for other tasks, such as accessory integration or Wi-Fi based location.*

Even if you had the entitlement you would run into these technical restrictions. The API was specifically designed to support hotspot navigation — in this context hotspots are "Wi-Fi networks where the user must interact with the network to gain access to the wider Internet" — and it does not give you access to on-demand real-time Wi-Fi scan results.

Many developers look at another developer's app, see that it's displaying real-time Wi-Fi scan results, and think there's some special deal with Apple that'll make that work.

There is not.

In reality, Wi-Fi accessory developers have come up with a variety of creative approaches for this, including:

- If you have a bound accessory, you might add WAC support, which makes this whole issue go away.
- You might build your accessory with a barcode containing the info required to join its network, and scan that from your app. This is the premise behind the Configuring a Wi-Fi Accessory to Join the User's Network sample code.
- You might configure all your accessories to have a common SSID prefix, and then take advantage of the prefix support in `NEHotspotConfigurationManager`. See *Programmatically Joining a Network*, below.
- You might have your app talk to your accessory via some other means, like Bluetooth LE, and have the accessory scan for Wi-Fi networks and return the results.

Programmatically Joining a Network

Network Extension framework has an API, `NEHotspotConfigurationManager`, to programmatically join a network, either temporarily or as a known network that supports auto-join. For the details, see Wi-Fi Configuration.

One feature that's particularly useful is it's prefix support, allowing you to create a configuration that'll join any network with a specific prefix. See the `init(ssidPrefix:)` initialiser for the details.

For examples of how to use this API, see:

- Configuring a Wi-Fi Accessory to Join the User's Network — It shows all the steps for one approach for getting a non-WAC bound accessory on to the user's network.
- NEHotspotConfiguration Sample — Use this to explore the API in general.

Secure Communication

Users expect all network communication to be done securely. For some ideas on how to set up a secure connection to an accessory, see TLS For Accessory Developers.

Revision History

- **2023-10-11** Added the *HomeKit* section. Fixed the link in *Secure Communication* to point to TLS For Accessory Developers.
- **2023-07-23** First posted.

Network

Reply

Posted 2 months ago by eskimo

Add a Comment

This site contains user submitted content, comments and opinions and is for informational purposes only. Apple disclaims any and all liability for the acts, omissions and conduct of any third parties in connection with or related to your use of the site. All postings and use of the content on this site are subject to the Apple Developer Forums Participation Agreement.

Copyright © 2023 Apple Inc. All rights reserved. Terms of Use | Privacy Policy | License Agreements