How to Configure your Raspberry Pi for SmartThings® direct-connected Devices

Introduction

This guide will cover all the steps to getting your Pi set up to successfully implement a SmartThings direct-connected device application.

By far, the easiest, and least error-prone path to success is to use the '**mastersetup**' script available here. This script will automate the majority of the steps outlined in this document to get you up and running relatively quickly. All you need to do is download the one file and execute it. Even if using the mastersetup script, you will still want to refer to two important sections in this document: (1) **Register test device in Developer Workspace** (page 8), and (2) **Device Onboarding** (page 18).

> For those who insist on performing all the setup and configuration tasks manually, this guide will provide step-by-step instructions to do that. And for those using the mastersetup script instead, it will provide all the gory details of what is being automated for you. However, a word to the wise for either scenario: If this is your first time going through this process, resist making modifications to these instructions or to the mastersetup process. Follow them to the letter your first time through, THEN you can experiment with your own modifications. There are a few important inter-dependencies among configuration elements that you might not grasp until you've successfully completed the process once, so making changes beyond the recommendations here can result in frustration and wasted time.

Information here is divided into the following sections

1. **Preparation**
2. **Install the required software**
3. **Register your device in SmartThings Developer Workspace**
4. **Configure your Pi's Wifi**
5. **Onboard your Device**

I know this guide looks daunting, and I'm surprised myself it takes this many pages of documentation! However, know that just about all of this is a one-time setup. Once you get everything working, implementing device applications that can interface to SmartThings via a simple API is quite easy to do.

In this document, there are many terminal commands I will suggest. Those are highlighted with a `cyan background` and you can copy/paste these to a terminal window. The ones in **`bold`** are generally going to be required; the ones that are optional are `not bold`. Some commands are purposely prefixed with 'sudo' where it is required, so mind those cases.

If you find anything in this document that can be improved, please don't hesitate to open an issue on my github repository.

Preparation

Get a SmartThings Developer account: https://smartthings.developer.samsung.com/
Get a Github account: www.github.com

Read the following documents:

SmartThings Developer documentation for "Direct-connected devices"

SmartThings Community Topic – "How to Build Direct Connected Devices" (how-to instructions)

*Important Note: you will not follow every step in that community post since it was not written for Raspberry Pi and is somewhat outdated at this point. But we will refer to it and follow some of the same procedures outlined there. I will refer to this resource often in this guide using the term "**how-to instructions**", and call out specific section numbers where applicable.*

Validating your model of Raspberry Pi is capable

There are multiple Pi models out there and not all include wifi chips that have the right capabilities to function in an IOT environment. If you own a Raspberry Pi Version 3 or later, chances are you'll be OK. The key wireless capability required is called "AP mode" or Access Point mode. This feature puts your Pi in the mode of being a WAP (wireless access point), similar to what your wireless router is in your home. This mode is used only during initial device onboarding to SmartThings, so it's a rather fleeting requirement. Nevertheless AP mode is required to successfully complete the device onboarding process with the SmartThings mobile app and requires some special setup on the Pi.

Even if your Pi has wireless, it doesn't necessarily mean the wifi chip supports AP mode, although I suspect that with all recent Pi models version 3B or later you won't have a problem.

To confirm that your Pi is capable of supporting AP mode, execute this command from a terminal widow:

```
iw phy0 info
```

'**phy0**' is the usual name of the physical wireless device. It's possible yours could be different.
The command above will display a whole lot of info regarding your wifi hardware support but what you are looking for is about 15 lines down from the beginning where it says **Supported interface modes**. In that section you are looking for "*** AP**". If it's listed there you are good to proceed.

As of this writing, the following Pi models have been tested and should work without issue: Model 3b, Model 4 (all), Zero W.

Raspberry Pi OS Lite

The instructions in this document generally assume you have a full Raspberry Pi OS installation on your Pi, however the **Lite** version of the OS (no desktop GUI) is also supported.

Prior to installing the required software, you will have to install two packages that are not included in the Lite OS:

```
sudo apt-get install python3-pip
sudo apt-get install git
```

You will also need an alternate way to display a QR code that you will generate for your device application. Since the Lite OS doesn't include graphics modules, you will have to do this on another system: another Pi, or a Windows or Mac computer. More information will be provided when we get to that step.

Installing the Required Software

Before you start this project, you should take this opportunity to do a full system update to your Pi to ensure you have the latest of everything. It will reduce the issues you may run into later.

By far, the easiest path to success is to have a recent Pi model (Raspberry Pi 3b+ or 4 or Pi Zero W) with Raspberry Pi OS Version 10 ("Buster") (Full or Lite), or Raspberry Pi OS Version 11 ("Bullseye") (Full or Lite) and Python 3.5 or later. OS versions back to Jessie can also work if Python3 is updated.

There are 3 parts to the software configuration:

1. Installing software dependencies
2. Cloning and building of SmartThings SDK
3. Installing & configuring the SoftAP software

Some notes about Python

The SmartThings SDK that will later be cloned to your system includes 2 important tools that you will use in the final device setup. These tools require Python **3.5** or later, so you will need to upgrade if you have an older system. You can find your Python version(s) by issuing both of these commands in a terminal window:

```
python --version
python3 --version
```

You probably have both version 2 and version 3 even with Buster OS. Confirm if your Python version 3 is at least **3.5**.

Upgrading Python is beyond the scope of this guide, so I would recommend searching the raspberrypi.org forums for instructions.

> *Beware of Googling vanilla Debian or Linux-platform Python install/upgrade instructions as they may not upgrade correctly on a Pi.*

As a precaution, I'd also recommend you confirm your version of pip (Python package installation program) by the following commands:

`pip --version` and **`pip3 --version`** << *note the double dashes*

Make sure you are using the Python 3.x version of pip in the next section below (i.e use 'pip3.x' instead of just 'pip3' if needed).

The Raspberry Pi Enabling Package

I have created this package that contains files you will need to (1) build a Pi-enabled SDK core library, and (2) configure your Pi for working as a SmartThings direct connected device. It is available from github at: https://github.com/toddaustin07/rpi-st-device

You can clone the repository to your system thusly:

```
cd ~
git clone https://github.com/toddaustin07/rpi-st-device.git
```

We'll refer back to this package shortly.

Additional Software Dependencies

We're now going to reference the SmartThings community how-to instructions post that I suggested you read through at the beginning of this document, as it contains some important next steps that we will *partially* follow.

In the 'Workstation Setup' section of the how-to instructions, there is an apt-get command provided to update a Linux system with all pre-requisite software modules:

```
sudo apt-get install gcc git cmake gperf ninja-build ccache wget make libncurses-dev flex
bison gperf python3 python3-pip python3-setuptools python3-serial python3-cryptography
python3-future python3-pyparsing python3-pyelftools libffi-dev libssl-dev
```

DON'T RUN THIS

You *do* need these modules, but some should already be installed on your Pi if you have an up-to-date system. Others are needed on Raspberry Pi in addition to those listed above. I would NOT recommend you do an 'apt-get python3' unless you really know what you are doing; this could cause problems especially on older Pi's.

Here's what I do recommend…

Before you start installing, run these command from your Pi terminal:

```
sudo apt-get update
sudo apt-get upgrade
```

I'll present the next install commands into three groups, but before you start running these individually, know there is one script available to perform all of these in one shot (identified further below).

1) This group should already be installed on an up-to-date Pi with the full Raspberry Pi OS:

```
sudo apt-get install gcc make git wget python3-serial python3-cryptography
```

2) This group will probably need to be installed:

```
sudo apt-get install cmake gperf ninja-build ccache libncurses-dev flex
bison libffi-dev libssl-dev python3-future python3-pyparsing python3-
pyelftools
```

3) These will be needed in addition to the how-to instructions list:

```
sudo apt-get install libpthread-stubs0-dev
pip3 install --user pynacl
sudo apt-get install python3-pil
pip3 install --user qrcode
```

Older Jessie or Stretch OS-based systems may also need this package:

```
pip3 install --user pillow
```

You will later use a Python script to create a qrcode for your Pi-based device application that will be used during initial onboarding. In order to use that tool successfully on a Raspberry Pi **Lite** OS (no GUI), you will need to install this *additional* module:

```
sudo apt-get install libopenjp2-7
```

You can either copy/paste the commands above to your terminal, or a simpler way is to run a script from my package that will issue all these commands for you:

```
cd ~/rpi-st-device
./installSDKdeps
```

Watch the output closely for any errors. If you see any problems regarding missing dependencies, you may have to install those missing ones first, then go back and try again.

Now that you have those packages installed, let's look at the next steps outlined in the how-to instructions…

There is a paragraph in the Workstation Setup section that suggests a way to make Python 3 the default on your system, but this is not needed if you are careful to specify 'python3' or 'python3.x' and 'pip3' or 'pip3.x' in all applicable commands. If you have interest in this topic of setting up alternative Python versions on your Pi, I'd recommend this raspberrypi.org article on the subject.

Skip the how-to instructions **pertaining to installing Espressif IDF or ESP32 toolchain**. You don't need any of those files in a Raspberry Pi setup.

The SmartThings SDK

There are actually 2 SDKs related to SmartThings direct connected devices. The one you need is the **core device library** SDK. You do NOT need the 'Reference' SDK, so ignore the **'Clone the SDK Reference'** paragraph in the how-to instructions.

Cloning the Core SDK

The SmartThings core SDK is a set of files that contain the source code to build the library that your device application will use to communicate with SmartThings.

The SDK is located on github at: https://github.com/SmartThingsCommunity/st-device-sdk-c

Be aware that you need about 224Mb of available disk space on your Pi for the SDK.

To clone the core SDK, in a terminal window navigate to your home directory and run the following command:

```
cd ~
git clone https://github.com/SmartThingsCommunity/st-device-sdk-c.git
```

Once that is complete, you will have the SDK on your local disk in the directory **~/st-device-sdk-c**.

Build the core device library

Before we issue the **make** command to build the SDK object module, there are a few modifications we need to make to the SDK files to build a library that will work on Raspberry Pi. To make this simple, run the bash script in my Pi package that will make the necessary changes:

```
cd ~/rpi-st-device
./sdkbuildsetup
```

Note that any SDK files that are replaced are first saved with 'ORIG' added to the name in case you want to examine them later.

Now we are ready to build the SDK library. Simply execute the **make** command while in the ~/st-device-sdk-c directory:

```
cd ~/st-device-sdk-c
make
```

The very first time you run a make against the SDK, it will take some time to complete since it will also be downloading and creating some additional networking libraries (Pi Zero W can be especially slow). Subsequent makes will go faster.

If you get any errors, you have missing library dependencies. If so, use **sudo apt-get** until you remove all dependency errors.

CONGRATS! You have now created the core SDK library module with Raspberry Pi support:

```
~/st-device-sdk-c/output/libiotcore.a
```

This file will be linked to your device applications when you build them.

Register Test Device in Developer Workspace

The next several steps will get a simple test switch device profile defined in the SmartThings Developer Workspace and create 2 **json** files that your Pi-based device application will need. Get signed-in to the Developer Workspace now from your web browser and follow the steps outlined in the how-to instructions starting in section **2.1 Create a new project** and through and including **2.7 Download onboarding_config.json**. (*Update: some of the DW screens have changed, so the screenshots in the how-to instructions may not be exactly right any more*)

Note that you could use any computer's browser for these steps, including your Windows PC or Mac, but since you will need to download a file to your Pi as part of this process, it may be more expedient to do this from your Pi's Chromium browser (slow as it is!).

As you proceed through the Developer Workspace steps, refer to this additional info that you will need to complete each section, and do not deviate from these values except for the Country:

Device Profile

Basic Info

Device Type: **`Switch`**

Components & Capabilities (main)

'**`Switch`**'

You can leave 'Health Check' as well, which is included by default

UI Display

State: **`Switch`**

Action: **`Switch`**

Device Onboarding

Authentication type: **`ED25519`**

Setup ID: **`001`**

Instructions: you can make these anything you like - they are customized prompts you will see on on your mobile device during the initial onboarding process of your device app. You might want to make the last prompt (Prepare Device) to remind you to start the device app on your Pi.

Confirm Method: **`Just Works`**

Product Info

Product Name: **`Switch Product`**

Category: **`Switch/dimmer`**

Availability: **`United States`** (or other)

Model Number / SKU: **`US_SKU`**

Your Test Devices - Create Unique Device Serial Number & device_info.json

At this step in the Developer Workspace (and as described in section **2.6** of the how-to instructions), you must provide a device serial number and public key for the device you will be testing. To generate these 2 items you use a tool in the SDK called **keygen**. Go to the **~/st-device-sdk-c/tools/keygen** directory on your Pi. Here you will find a Python script that you will run to generate a unique serial number and public key that will identify your Pi-based device to SmartThings.

```
cd ~/st-device-sdk-c/tools/keygen
python3 stdk-keygen.py --firmware switch_example_001
```

If you know your SmartThings-assigned ManufacturingName (aka mnmn or mnid), you can also include it as an argument to the keygen tool, e.g. **`--mnid eF8b`**, but this is completely optional.

The keygen output (serial number & public key) must be copy/pasted into the device identity fields within the Developer Workspace device setup screens as shown in the how-to instructions section **2.6**. Do not try and type these by hand - it is too easy to make a mistake (O vs 0, I vs l vs 1, etc), and if these are not correct, your device onboarding attempt will fail with cryptic error messages.

Copy device_info.json into device application directory

The keygen tool also created a **device_info.json** file that needs to be copied to the example application directory **~/st-device-sdk-c/example** (you can replace the one that is there). This file includes the generated serial number and keys that your device app will need to authenticate with SmartThings. The **keygen** tool placed this file into a new subdirectory with a name corresponding to your device serial number. Copy it into the example directory now:

```
cd ./output_STDKxxxxxxxxxxxx          << where xx…xx is your device serial number
cp device_info.json ~/st-device-sdk-c/example/device_info.json
```

Download onboarding_config.json

Next you need to download the **onboarding_config.json** file from the Developer Workspace into the SDK example directory (again, replacing the file that is already there). This file gets created when you finish the Device Onboarding steps in the Developer Workspace, so once you completed that, there should be a download link on the project Overview page. See section **2.7** in the how-to instructions.

Do NOT follow section **2.8** in the how-to instructions, but you can reference section **3.2** for more information about the json file we've just downloaded. Ignore references to the ESP32 example.

Generate a QR Code for your device

This step uses the second Python tool included in the SDK called **qrgen**. Let's run it now with the following command:

```
cd ~/st-device-sdk-c/tools/qrgen/
python3 stdk-qrgen.py --folder ~/st-device-sdk-c/example/
```

The `--folder` argument must point to the location of the **onboarding_config.json and device_info.json** files which should now both be in the example application directory.

After running **qrgen**, you will have a new **.png** file in the qrgen directory that represents a unique QR code for your device, embedded with your unique serial, key, and other device onboarding information. You will need this later when you use the SmartThings mobile app to add your device. I'd recommend making it a practice to copy this .png file into your device app directory - in this case ~/st-device-sdk-c/example.

Note: if you do not have the Raspberry Pi OS desktop GUI installed, you will have scp the .png file to a computer where you can display it during device onboarding. (It cannot be displayed in Raspberry Pi OS Lite due to missing graphics libraries.)

Build the Example Application

Assuming you have the new **device_info.json** and **onboarding_config.json** files stored in the SDK example application directory, we are now ready to run 'make' to build the example device application:

```
cd ~/st-device-sdk-c/example
rm example        << delete any previous example executable to ensure make executes
make
```

Assuming you had successfully built the SDK core library previously and have not made any changes to the example.c code, you should pretty much instantly have a successful make with no errors.

Feel free to look through the example.c file to get a feeling for how the SDK APIs are used from a device application. This example implements a simple on/off switch device.

If you try to run the device application at this point you will get errors, since we first need to get your wireless configuration set up and the SoftAP applications configured and working.

If you've made it this far, congratulations! You've now proven you can build the SDK library and the example application successfully, so you have all the software requirements for those sorted out. Now we can proceed to configuring your Pi to get it ready to actually *onboard and run* the device application that you have just built.

Configuring your Pi to support Wireless AP mode

We need an application that implements the SoftAP capability (AP mode) on the Pi. This will be accomplished with two service modules called **hostapd** and **dnsmasq**. There are other options out there that perform similar functions, but these are what I have selected as they have a proven history of successfully running as-is on Raspberry Pis and are fairly lightweight.

All SoftAP-related install and configuration tasks can be accomplish quite quickly by using a bash automation script included in the RPI package called **SoftAPconfig**.

(For those preferring a do-it-yourself approach, the manual steps are detailed farther below starting with "**Configuring Wireless Devices**")

Using the Automation Script

Before you run the automated script, have ready the answers to these configuration questions:

1) Wifi device name to use for AP mode
   normally 'wlan0' - use this for now until you are ready to experiment with other configs
2) Static IP address for your new AP
   ensure no conflicts on your home network
3) IP range that the SoftAP DHCP server will assign to connections
   make the same subnet as static IP above; ensure no conflicts on your home network
4) Channel number for the AP to use
   match the wifi channel used by your home router that you are normally connected to (1-14)
5) Country code (2-character)
   reference link *e.g. US, GB, CA, etc*

Further details on these parameters are contained in the manual steps guidance below. Don't worry if some of these are a mystery to you, the script will recommend defaults you can accept in most cases.

You can run the setup script now:

```
cd ~/rpi-st-device
sudo ./SoftAPconfig
```

*Please note that you must run this script with **root** privileges since it will update some network interface files and services configuration.*

If you have customized your /etc/dhcpcd.conf file, then you may be asked to manually edit dhcpcd_ap.conf during execution of this automation script. If so, follow the instructions at the bottom of page 12 for the required contents of the **dhcpcd_ap.conf** file, which should *replace* any other existing customized `interface wlan0` entries already in the file.

Once you've made your wifi config selections and have gotten to the last step in the script to optionally test hostapd, you can proceed to the section "**Testing your PI wireless Access Point**" in this document for more info (the SoftAPconfig script will also automate many of those final testing steps for you).

- Start of Manual Configuration Steps - *(Skip to page 16 if you ran the automated script, SoftAPconfig)*

Configuring Wireless Devices

What we'll describe here is how to get your wireless device set up on your Pi, without using the SoftAPconfig automation script. There are actually a number of ways you could configure your Pi, but for simplicity sake I'm going to approach it one way in this guide. If you have the expertise, you are welcome to explore other options which are discussed in the **Reference** section at the end of this document.

Here is some optional background info for those of you who like to know what's happening 'under the covers':

> AP mode, or Access Point mode is a special capability of your Pi wireless that is required to complete the device onboarding (or provisioning) process. Put simply, it turns your Pi into a wireless access point so that the mobile app can temporarily connect directly to it during device onboarding. If you've ever provisioned other wireless IOT devices (e.g. Ring doorbell), it's a similar process where you run the manufacturer's setup app on your phone and your phone's wireless briefly connects to the *device's* own ssid to exchange configuration information. Once the device is configured, the device then connects to your home's wireless router and lives the rest of its life as just another wireless client (also called managed or station) on your network. (And your phone's wifi connection resumes back to what it was originally connected to.) This ability for the *device* (Pi) to temporarily create its own wireless access point is called "SoftAP" in the industry, and while not the most user-friendly process, it's the current standard for provisioning wireless IOT devices. The configuration instructions that follow help setup your Pi to be able to enter this SoftAP state when required.

Determine the wireless device that will handle AP mode

First confirm what wireless devices you have with this command in a terminal window.

```
iw dev
```

Note the physical wireless device name in the first line of output - probably called '**phy0**'.
Now take note of what Interfaces are listed. If you've never messed around with your devices, you probably have just one Interface called '**wlan0**', and you can see that the **'type'** shows '**managed'**.
If it's currently connected to your home wireless, you'll also see the SSID listed, and you should take note of the **channel** number being used - you will need that later.

For our initial configuration purposes, we will assume the use **wlan0** as both our client (managed) and AP mode device. The wifi mode will be changed dynamically as needed during runtime.

Setup dhcpcd config for AP mode

We will create a new file in the /etc directory that will be utilized by dhcpcd during AP mode.

```
cd /etc
sudo cp dhcpcd.conf dhcpcd_ap.conf
sudo nano dhcpcd_ap.conf
```

In order for AP mode to work on your Pi, it must be configured with a **static** IP address while in AP mode. So add these lines to the very end of the new dhcpcd_ap.conf file:

```
interface wlan0
    static ip_address=192.168.2.1/24
    nohook wpa_supplicant
```

Where *wlan0* is the name of your wireless device.

Replace the **ip_address** value with the static IP you want to use. Be sure the IP address is outside the range that your home DHCP server assigns (which is typically defaulted to 192.168.1.1 through 192.168.1.250). In this suggested config, we will use the 192.168.**2**.x subnet for our Pi Access Point. Finally, be sure to include the 'nohook wpa_supplicant' line.

Save the **dhcpcd_ap.conf** file and exit the editor.

SoftAP Services

To install the needed SoftAP services (**hostapd** and **dnsmasq)**, issue these commands in a terminal window:

```
sudo apt-get install hostapd
sudo apt-get install dnsmasq
```

These two services are installed by default to start at boot time, but we want to prevent that for our purposes, so run these commands next:

```
sudo systemctl unmask hostapd
sudo update-rc.d hostapd disable
sudo update-rc.d dnsmasq disable
```

Note: if you intend to have AP mode running continuously and you want hostapd and dnsmasq to be enabled at boot time, then change the last two commands to **enable** instead of `disable`. This configuration can only be used if Ethernet is also connected, so it's not allowed on a Pi Zero W.

Now tell the system where it will find the hostapd configuration file:

```
sudo nano /etc/default/hostapd
```

Find the line containing "DAEMON_CONF=" (around line 13), remove the comment symbol (#), and modify it as follows:

```
DAEMON_CONF=/etc/hostapd/hostapd.conf
```

The last step in this section is to modify the SoftAP services configuration files.

First, let's grab some default config files from my package with a bash script:

```
cd ~/rpi-st-device
sudo initsoftapconf
```

This will copy initial configuration files for hostapd and dnsmasq into the appropriate directories.

hostapd Configuration File

The file **hostapd.conf** should now be located in /etc/hostapd. There are three parameters in this file

that you will need to modify now, so edit the file:

```
sudo nano /etc/hostapd/hostapd.conf
```
<< *'sudo' is mandatory*

country_code=*US* make sure it's set for your 2-character country code (US, GB, etc)
reference link

interface=*wlan0* set to whatever device name is being used for AP mode

channel=*n* This value should be set to the same channel (1-14) as wlan0 is normally connected with as a wireless client. Whatever wireless router your Pi is connected to by default (as a client), use that same channel for this hostapd configuration. If your Pi wifi is presently connected to your wireless router, you can do a `iw wlan0 info` command to see the channel it is using. Alternatively, go into your wireless router's configuration screens to see how you have it set.
See also below Sidebar on Wifi Channels.

Sidebar on Wifi Channels (experts can skip this)
Since we are operating in an IOT environment, it's worth mentioning here the importance of wifi channel selection. I assume you have a SmartThings hub in your home – hopefully not too close to your wireless router. You should be aware that the 2.4GHz wireless spectrum overlaps that used by zigbee devices. So it's important to minimize this potential conflict by choosing a wireless router channel frequency as far away as your zigbee frequency as you can. Some hubs may have the zigbee channel printed on the bottom of the physical hub. If not, you can go into the SmartThings IDE and find where you can display the details of your hub. There it will show what channel it is using for zigbee. Refer to discussions on this topic in the SmartThings community or Google 'wifi and zigbee overlap' and there are some nice graphs (like this) that will help you choose the best wireless channel to use for your wireless routers. Often it's going to be either channel 1 or channel 11. As an example, my hub uses zigbee channel 20 which is in the upper-middle part of the 2.4Ghz range. I have two wireless routers – one configured for channel 1 which is in the same room as my ST hub, and one configured for channel 11 which is farther away from the hub. That provides frequency space for zigbee to live between them and also keeps the two routers from conflicting with each other.

OK, back to hostapd.conf. Don't change any other parameters besides the 3 specified above. That goes for ssid and password as well; they will be updated dynamically during runtime. But do take note of the ssid that is in there now ('MyPiTestAccessPoint'); you'll see that later when you bring up your AP for testing.

Save the hostapd.conf file (back to /etc/hostapd/hostapd.conf) and exit the editor.

Further info on hostapd.conf
As described earlier, hostapd is the service that puts your Pi's wireless interface into Access Point mode. If you want to have your wireless permanently in AP mode (only applicable for Pi's with Ethernet connection), you can changed the SSID and password in the hostapd.conf file to the values of your choosing. During any device onboarding, hostapd.conf will be dynamically changed to a different SSID and password while the mobile app needs to connect to it. But don't worry, your settings will first be saved and then restored once device provisioning is done. So be aware that your BAU Pi Wifi AP will be temporarily interrupted during device onboarding.

dnsmasq Configuration File

Now look for the dnsmasq config file in /etc and edit it:

```
sudo nano /etc/dnsmasq.conf
```

Modify its contents as follows:

```
interface=wlan0
dhcp-range=192.168.2.2,192.168.2.10,255.255.255.0,12h
domain=wlan
address=/gw.wlan/192.168.2.1
```

Set **interface** to the device name being used for AP mode.

Set the **dhcp-range** (ip address range) you want the Pi AP to assign out to its connecting clients. Note that I have a small range of addresses shown above since I realistically don't expect anything to be connecting besides my phone. Again here I am allocating the 192.168.2 subnet to the Pi AP to avoid any conflicts with other DHCP servers on my LAN.

Replace the ip address in the **address** parameter to be the static address you defined in dhcpcd_ap.conf

Save the dnsmasq.conf file and exit the editor.

Continue following the steps in the next section, "**Testing your Pi wireless Access Point**"

- End of Manual Configuration Steps -

Testing your Pi wireless Access Point

You've reached this step either after you've run the SoftAPconfig script, or you have followed the manual instructions above. Either way, by now you should have a fully configured system to enable SoftAP capability (wireless Access Point) with the hostapd and dnsmasq services.

If you've re-booted since completing the previous steps, confirm that your wireless client capability is still operating. You should be able to use the wifi icon in the top right of the Pi desktop GUI to see what SSID you are connected to, assuming you are in regular managed/station mode.

There is a bash script that will manually start hostapd for testing, and return your wireless back to client mode when it is done. (Note that those using SoftAPconfig will have the test started and terminated for them, so this other script is not needed.)

Warning!! *This next step will turn off your wireless client operation*. If you are connected as a remote console to your Pi, you'll need to be connected instead via Ethernet, or get on to your Pi console directly. We'll restore your wireless client operation when we're done.

```
cd ~/rpi-st-device
sudo ./testhostapd
```

When hostapd loads, you may see errors like "failed to create interface mon.wlmyap" or 'Could not connect to kernel driver". Don't worry - these can be ignored. You have success if the last line of the output shows "**AP-ENABLED**" . You can verify your AP is live now by using your mobile phone to go into wireless settings where it displays all available access points in your vicinity. You should see your new Pi AP listed there as "**MyPiTestAccessPoint**" which was the default SSID for initial testing in the hostapd.conf file. *NOTE: Do not try to connect to this SSID with your mobile device. It is configured to only be functional for use by the SmartThing mobile app during an actual device onboarding.*

Once you've confirmed the AP is live, **Ctrl-c** to terminate hostapd, and the testhostapd script will restore your wireless back to client mode.

Troubleshooting

If your wireless AP interface doesn't seem to be live, the first thing I would recommend after examining any error messages, is to reboot.

Ensure there is no "soft" block on your physical wireless device:

```
sudo rfkill list all
sudo rfkill unblock n
```

<< where *n*=single numeric digit (typically 0) corresponding to the **phy0** device listed in the previous command output

Check that the network is up:

```
ip link
```

in the output, you should see the word "UP" between the < > brackets for your wireless devices
e.g. `wlan0  <BROADCAST,MULTICAST,UP,LOWER_UP>`

If your device seems to be down, then issue these commands:

```
ifdown wlan0
ifup wlan0
```

Beyond that, the failure to have a live AP working is most likely a configuration error, so go back and re-validate all your previous steps.

If you have to fix things, remember that reboots may be needed, or you need to stop (use **Ctrl-c**) and restart hostapd and/or restart dhcpcd service (`sudo systemctl restart dhcpcd`).

When you get SoftAP working successfully, remember to **Ctrl-c** out of hostapd when you are done.

If you seem to be stuck in AP mode, you can try using **~/rpi-st-device/resetAP** to restore it back to station mode (or re-boot).

Device Onboarding - Add Device in SmartThings Mobile App

Now we're ready for the real action! We will use the SmartThings mobile app to 'add' the test device to your SmartThings device inventory. You can reference section **4.3** in the how-to instructions for screenshots of much of this process, although some of them are outdated.

I recommend fully reading through everything here and in the applicable how-to instructions sections before you proceed.

As described in the how-to instructions section **4.1**, you must have '**Developer Mode**' enabled in the ST mobile app through the **settings** menu.

To ensure 'Developer Mode' is on: .

> *On the main screen of the mobile app, tap the Menu icon in the lower right, then the gear icon in the upper right. Scroll all the way down in the options and make sure the **Developer Mode** switch is **enabled**. If the switch does not appear, then long-press for 5 seconds the **About SmartThings** menu item. The Developer mode switch should then appear below, which you should turn on. Then restart the SmartThings app.*

You also need to make sure that the SmartThings mobile app has permission to use your camera (for reading a QR code), so take care of that now by going into the appropriate settings of your mobile phone's OS.

IMPORTANT: Network connections to the SmartThings MQTT server requires an accurate clock setting on the Pi, so make sure your Pi clock is set accurately.

Back in the ST mobile app, tap on the **+** symbol in the upper right and then tap **Add device**. Don't go any further than that yet.

Now open up the **QR code** image that you created earlier. On the Pi, you can use **gpicview** in a terminal command or double-click on the **.png** file from file manager. Recall the .png file was created in the **~/st-device-sdk-c/tools/qrgen** directory, and I recommended you also copy it to your application directory - in this case **~/st-device-sdk-c/example**. If you are running Raspberry Pi OS Lite, you'll have to display this file on another computer (Pi / Windows / Mac).

Back on your mobile phone on the **Add device** screen, you should see 'Scan QR code' as the first option. Tap that and point your phone's camera at the QR code image you should now have displayed on a monitor. That will kick off the next step in the mobile app.

> *Note: if you get an error here saying the QR code is invalid, then following a different path in the mobile app may get you passed this. Go back to the Add device screen and below the 'Scan QR code' and 'Scan for nearby devices' options you will see a list of icons. Be sure the 'By device type' tab is selected, then scroll all the way to the bottom of the icon list and you should see an icon labeled 'My Testing Device' or similar. Tap that, then from the list of Developer Workbench projects that is displayed, select your Pi test switch (e.g. 'Jeff's RPI Test Switch'). This will launch the onboarding process screens and when you get to the screen that launches the the QR reader, BEFORE you point it at your QR code, launch the Pi example app as below. THEN, once the example app is fully initialized and listening, point the phone to your QR code.*

Now go to the Pi example directory and launch your device application.

```
cd ~st-device-sdk-c/example
./example
```

You should see numerous informational messages flashing by as the software initializes. If you see this error message among those displayed, you can ignore it.

```
[IoT]: _iot_security_be_bsp_fs_load_from_nv(129) > iot_bsp_fs_open(/IotAPPASS) = -1208
```

Raspberry Pi Zero W users will also see a warning message that Ethernet is not found; this is normal!

The software will determine that you've never onboarded this device before and wait for the mobile app to initiate onboarding. Once the messages settle and you haven't seen any other errors, continue back on your phone with the add-device procedure that we started earlier.

On the the mobile app, after the QR code is recognized, there should now be a **Start** button displayed which you can tap. You will get to a screen that asks you what Room to put the device in, so select a room and then tap 'next' a couple times until you see a message that asks if you want to join a network SSID with a name made up of your unique device serial number. Tap yes, and after a short pause you will see some additional activity from the the Pi device app message log. Then on your mobile phone you will be presented with a list of SSIDs to connect to. This is what your Pi wifi client will be re-connected to once AP mode is over (if you have an active Ethernet connection, it's a moot point, because that will be used by default). Choose your SSID* and enter the password if required.

> **For Pi configurations with an always-on AP mode*: the selection of this SSID will have no effect on your Pi's AP configuration; your Pi AP will return to the SSID it was configured-to prior to device onboarding.*

Once you choose the SSID in the mobile app, there will be a long pause, but if everything goes well you will eventually see further activity on your Pi terminal and finally get a message from the mobile app that the device was successfully added. Press OK on the mobile app.

At this point the mobile app will show your new test switch device on your main screen, and you can try turning the switch on and off. You'll see corresponding log messages on your Pi terminal window where your device app is running and you can see a sequence number increment each time. *Sweet, it all works!*

Troubleshooting

If things don't go so well and the process craps out at some point, you'll have to do some troubleshooting. The first thing I would do is simply retry. If that's not successful, you can read through the errors on the Pi terminal and they will narrow down pretty well where the problem occurred. Note that all the Raspberry Pi-unique-code messages will contain the characters **[rpi]**.

Unfortunately any error messages or codes you may get in the mobile app are usually fairly cryptic and don't help with problem determination. But exactly where the process stalls, provides a good indication of where to look for problems:

If you never get a list of SSIDs to choose from on your mobile app, then it's likely it wasn't able to connect to your Pi. That could mean your AP is not operational, but you can confirm this by the Pi log messages. There could be a configuration error in your hostapd.conf file, but that is less likely since you verified this was working during the previous steps.

This could also mean the SSID that the mobile app is looking for is not found. The SSID is constructed from a combination of your deviceOnboardingId (from onboarding_config.json) and device serialNumber (from device_info.json). These values are also embedded in the QR code which also informs the mobile app what SSID to look for. So if values are not synchronized across

your Developer Workplace configuration, json files, and or QR code file, it would cause this connection failure. If you jump over to your mobile device's wifi settings to see the currently available SSIDs while the example app is still waiting for connection, then you can see what SSID is live. Confirm that this matches what you would expect. You can also check the Pi example app log output and there should be a log message that displays a **truncated** portion of the SSID, e.g.:

```
[IoT]: iot_easysetup_create_ssid(89) > >> Jeffs RPI Tes[IsZf] <<
```

In the event the example device application on the Pi appears hung, if you are patient it will usually time out and your wireless mode will be reset to its original state. If you Ctrl-c out of the example device application before it times out naturally, it may leave your wireless in the **SoftAP** state (up/down arrows instead of the regular wireless icon in the upper right of the desktop). If you find this is the case, and before you restart the example app to try the onboarding process again, run the **resetAP** bash utility in the rpi-st-device folder to get your wireless back to its original mode. In the rare case that doesn't work, it may be best to reboot.

When the mobile app is done successfully connecting to your Pi's access point, it will shut down the connection, and then the Pi SDK code will then shut down the AP mode and reconnect wifi to the SSID you selected in the mobile app. The mobile device should then connect back to its previous wifi router SSID and continue some additional initialization tasks with the SmartThings Servers. The Pi example app will then be listening for the SmartThings servers to contact it with the final device onboarding confirmation.

On occasion, the SmartThings servers have been known to simply fail to connect back with the Pi app in time, and the whole process fails. In those cases it is possible that the device DID in fact get created, and a restart of the example app clears things up. But more likely you'll need to restart the onboarding process from the beginning.

It may also be helpful to examine the **provisioning data files** that get stored on your Pi to see how far along in the onboarding process you had gotten. There is a utility program from my package you can use to conveniently display this data. In a terminal window, do the following:

```
cd ~/st-device-sdk-c/example
~/rpi-st-device/STProv
```

You can optionally copy this executable to your /usr/local/bin directory so you can run it from any directory (assuming you your PATH includes /usr/local/bin).
At present, there is a second executable, STProv64, for use on 64-bit OS configurations.

What you'll see when you run this utility are the contents of several files that show the core SDK provisioning status and stored info for both wifi and cloud connections. Both wifi and cloud provisioning status should show as "DONE" if your device was fully onboarded. The wifi data would show info for the AP you selected in the mobile app. The cloud server URL and Port are given to the device app by the mobile app during onboarding.

Note that if you looked at this data before a device is onboarded, the status for both wifi and cloud would show as "NONE", or the files may not even exist yet.

Generally the last bit of data that is saved during device onboarding is the Device ID, which comes from the SmartThings server once the device is fully registered.

Finally, you may want to turn debug-level messages on by un-commenting the last line in **~/st-device-sdk-c/stdkconfig** and rebuilding the SDK core library and example application:

```
cd ~/st-device-sdk-c
nano stdkconfig
make
cd example
rm example
make
```

If you are having problems, it's a good idea to capture (copy/paste) all the terminal output into a file so you can share it when seeking help.

Where to go from here

Once you have successfully onboarded, you are able to stop (Ctrl-C) and restart the device application and it will simply reconnect to the SmartThings MQTT server as a normal client. You won't have to go through the onboarding process again for this device unless you delete the device from the mobile app. You'll notice that if you stop the device application on your Pi, the SmartThings mobile app will show that device with an "Offline" status.

If you delete the device from the mobile app, SmartThings sends a notification to the Pi device application that the device has been deleted and the Pi device app simply exits with a message letting you know that the device was deleted. You can re-onboard the device with the same serial number (and QR code). Of course any *new* device apps you develop will need their *own* serial numbers and QR codes. If the devices are all the same time type, they might be able to share the same onboarding_config.json file.

Try making some changes to the example app and re-running make. Remember **you don't need to rebuild the core SDK** and you won't have to repeat the onboarding process even after changing the device app code and rebuilding just the device app. As long as you continue to use the same **json** files you originally created, SmartThings will recognize your app as the same registered device.

Develop Your Own Device App

The reason you're doing all this is because you want to write you own device application. Get to know the API you'll use, as documented here.

The SmartThings core SDK API supports C-language apps, however this package also contains a python subdirectory that includes a readme file for how to build a shared library that serves as an API wrapper to access the C-based core SDK API. There is also an example python device application you can try, and use as a template to build your own application.

Use the Developer Workspace to define your new projects and device profiles and remember that each unique device profile device will (1) need its own pair of json files (**onboarding_config.json** - from Developer Workspace, and **device_info.json** - created by keygen tool), and (2) need it's own **QR code** generated for onboarding. Plan to use a unique directory folder on your Pi for each device application, since they each require their own device_info.json files, as well as other runtime provisioning files.

If you instantiate multiple test devices under the same Developer Workspace device profile, then those devices will share the same onboarding_config.json file. However they will still each need their own device_info.json (serial number) and QR code.

Reference the example app **Makefile** to see how your C-application needs to be built and linked with the SDK core library.

*There are two bash files we have not yet mentioned in this guide that are used at runtime by the SDK core library rpi module: **softapstart** and **softapstop**. Be sure these are present either in the ~/rpi-st-device directory or in your device application directory (searched first).*

Managing json files

The example C application included with the core SDK was designed to link binary versions of the device_info and onboarding_config json files directly into the executable. While this makes sense to

do in an MCU environment, you certainly can do this differently with your Pi-based applications. Instead of hard-linking the json files during the make process, you can leave them as separate files to be read in and passed to the SmartThings API at runtime. This method is used in the python device app example included in the python directory of the rpi-st-device package.

Controlling Log Messages

You can determine what level log messages (info/warning/error/debug) are produced by the core SDK library by editing **~/st-device-sdk-c/stdkconfig** and removing or adding comment prefixes ('#') on the last 4 lines of the STDK_EXTRA_CFLAGS section of the file. Then re-make the core SDK library and your application.

Updating Packages

It's a good idea to updated your RPI and core SDK packages periodically to be sure you've picked up the latest files:

```
cd ~/rpi-st-device
git pull origin main
cd ~/st-device-sdk-c
git pull origin master
~/rpi-st-device/sdkbuildsetup
make
```

Any time you update the core SDK files, be sure to re-run the **sdkbuildsetup** utility to ensure the Raspberry Pi-specific modules are still properly installed before you re-build the SDK library.

Share your projects

Finally, please consider sharing your new Pi-based device projects on the SmartThings community and Raspberry Pi communities so we can leverage each others' work. And if you really want to get professional, you could pursue official device certification from SmartThings (I would hope they would do that for Pi-based devices!).

REFERENCE: rpi-st-device folder contents

Config/Setup Scripts

installSDKdeps	apt-get and pip installs of all SDK prerequisite software
sdkbuildsetup	update SDK with Raspberry Pi-enabling files
SoftAPconfig	auto setup and configure SoftAP capability- devices and services
mastersetup	master quick-start script
initsoftapconf	installs default config files for hostapd and dnsmasq (for manual installs)

SoftAP iniital config files

RPIhostapd.conf	hostapd config
RPIdnsmasq.conf	dnsmasq config

Tools

STProv	program to display SmartThings provisioning data store files
STProv64	same as above, but for 64-bit OS environments
testhostapd	bash script to put wlan0 into AP mode and test hostapd service
resetAP	bash script to turn off AP mode (use if runtime error left AP on)

SDK Make

RPIMakefile	Core SDK make file for building Pi-enabled library
RPIstdkconfig	Make config file for building Pi-enabled library
iot_bsp_wifi_rpi.c	SDK BSP wifi module for Raspberry Pi-based devices
iot_bsp_system_rpi.c	SDK BSP system module for Raspberry Pi-based devices

Runtime

softapstart	Start hostapd and dnsmasq services (used at runtime)
softapstop	Stop hostapd and dnsmasq services (used at runtime)

Docs

README.md	Repository overview
QuickstartGuide.pdf	Quick start guide for mastersetup script
ConfigGuide.pdf	Detailed setup and configuration guide (this document)

Python Subdirectory

setup	bash script to set up your Python project directory with all required files
requirements.txt	list of pre-req python modules
getprovfiles	bash script to get C example provisioning files for running Python example
iotcorebuild.py	API wrapper build script
STDevice.py	API wrapper class methods and constants import file
py_st-dev.h	SDK core API header include file for Python
pyexample.py	example python device application (simple switch)
libiotcore.a	pre-built SDK core library file (for convenience only; best to build yourself)
PyREADME.md	instructions for building and using the API wrapper

REFERENCE: Advanced Configuration Options

There is more than one way to configure a Raspberry Pi to allow you to provision and run a SmartThings direct connected device. There are two mandatory elements:

1) The ability to operate in Access Point (AP) mode during initial device onboarding
2) Normal client access to the internet; used post-onboarding

#2 is the simplest to address, so we'll cover that first…
On a Pi, one could use either wireless client or Ethernet - both standard capabilities. In some cases you may not have a hardwired Ethernet connection (e.g. Pi Zero W) and will rely solely on a wireless connection.

If you have both connections available, the way the system networking works is that Ethernet typically takes priority. This can be seen by issuing an `ip route` command in a terminal window. The output will show a list of your network devices, with something like "metric 202" or "metric 303" at the end of each line. This number is the priority the system uses, and the lower the number, the higher the priority. On a Buster OS system, eth0 is 202 and wlan0 is 303. So if you are connected simultaneously to both Ethernet and wireless, if an application is trying to communicate to the network, it will get routed via Ethernet. But if your Ethernet is down for whatever reason, then your wireless will be used.

Now let's address #1 above: AP mode…
There are multiple configurations possible to satisfy this requirement. For simplicity of getting up and running quickly, the previous instructions in this guide - along with all associated automation scripts - were written with the simplest setup in mind: using the default wlan0 device for both client and AP modes, and dynamic discovery of Ethernet existence. But here we'll lay out other possible scenarios in case the reader wants to explore them.

It's worth mentioning again that AP mode is a very fleeting requirement for our purposes, as it's only used during initial device provisioning. Once your device is onboarded, it will live the rest of it's existence as a standard network client application.

That said, as you come to understand what AP mode does and what the SoftAP applications (hostapd and dnsmasq) allow you to do on your Pi, you may become interested in exploring this capability further and perhaps want to have AP mode enabled more often (even full time) so you can turn your Pi into another wireless access point option in your home. You may have uses for this functionality beyond what is required to get your SmartThings devices onboarded. You could, for example, extend your AP configuration to provide internet access to the devices connected to your Pi's AP.

On the next page is a table of the various possible combinations of configuring your Pi. There are significant variations in complexity among these combinations, and your OS version may determine how simple or how complex some of these may be. More on that below.

Note that our discussion here is limited to the Raspberry Pi models 3, 4, and Zero W with built-in wireless capability. This of course could be augmented by adding a USB wifi dongle, which would provide even more configuration options, but that will not be discussed here.

Configuration Options to support SmartThings direct connect device applications

Ethernet	Wireless Devices		NOTES
	Managed	AP	
√			Only possible **after** provisioning has been completed via other options below
	√		wlan0 switched to AP mode as needed at runtime; no Ethernet
√	√		wlan0 switched to AP mode as needed at runtime; Ethernet used for internet
		√	Not supported - since no options for internet
√		√	Assume full-time AP mode started at boot; Ethernet available for internet
	√	√	Dual wifi devices but not run concurrently; keeps AP mode separate from wlan0
	√	√	Dual wifi devices run **concurrently**
√	√	√	Dual wifi devices **concurrent or not** plus Ethernet

Notes:

Ethernet means it is connected and operational; not just that you have an Ethernet jack :-)
Wireless Devices refers to the configured 'virtual' devices on your system (display with `iw dev` command)
Default devices on a standard Raspberry Pi are **eth0** (Ethernet) and **wlan0** (managed Wifi)
'Managed' for our purposes is synonymous with 'station' and 'client mode'
If no Ethernet, then 'Managed' device (e.g. wlan0) is used for internet / client mode communications

Regarding Ethernet - if it's connected it will be used for client communications. If not connected, wireless client will be needed/used.

As mentioned above, the simplest configuration for those just getting started is what is used by the mastersetup and SoftAPconfig scripts: assuming a single wifi device - wlan0 - and dynamically switching it to AP mode during device provisioning. Once provisioning is done, wlan0 is switched back to normal client mode. A minor downside to this configuration is that without also an Ethernet connection, your wireless client communications will be temporarily interrupted any time you are onboarding new SmartThings device applications.

Another fairly simple configuration option is changing your wlan0 device to be a permanent AP device. In this case you *must* also have Ethernet, which will provide the client mode communications, since you'll lose that ability over wireless. AP mode would be turned on at boot and left on indefinitely. The advantage here is you gain a new wireless access point to use for other purposes, like providing internet access to connecting devices. This configuration is actually quite commonly described in how-tos for configuring Pis for hostapd and dnsmasq, the two applications we use to implement the "SoftAP" capability.

There are, however, some interesting advantages to having a second and separate *virtual* wireless device defined that is dedicated to AP mode. The main idea being that you leave your wlan0 device alone and continue to use it for normal wireless client networking, while reserving the separately-defined 'virtual' device for AP operation. In theory, this could reduce potential issues with switching a single wlan0 device back and forth between station and AP modes. When Ethernet is not available, having dual wifi devices could mean that client communications would not be interrupted while onboarding devices - but only if the two virtual wifi devices can operate **concurrently**. This is a further option to the dual wifi device setup: whether they operate concurrently, or one mode at a time.

Some notes regarding concurrent mode

For many, concurrent wifi client and AP operation could be an ideal setup. It has many advantages and would be great for anyone that wants a full time Pi-based wireless access point that other devices in your home can connect to. On a Debian Jessie-based system this was fairly easy to accomplish. Unfortunately Debian Stretch and Buster versions introduced some problems, so achieving concurrent operation requires more challenging system configuration changes in those more recent operating systems.

To confirm that your wireless *hardware* (not necessarily your OS) can support concurrent operation, run `iw phy0 info`

from a terminal window and look almost to the end of the output where it says **valid interface combinations.** This part of the output shows what your wireless hardware can have running simultaneously. You should see a line that looks something like this:

```
* #{ managed } <= 1, #{ AP } <= 1, #{ P2P-client } <= 1, #{ P2P-device } <=1,
   total <= 4, #channels <= 1
```

The sample output above indicates the hardware can support up to 4 different types of virtual wifi devices running at the same time as long as they are all using the same channel. In this case you can have 0 or 1 managed, and 0 or 1 AP, and 0 or 1 P2P-client, and 0 or 1 P2P-device all defined and running at once.

The runtime code that comes with the current PI enablement package for SmartThings direct connect devices can support most of the scenarios shown above, however actually getting them to work on your system - especially the concurrent dual devices on a Buster-based Pi - can be challenging and is left to the intrepid user to explore.

Here are some links to various configuration discussion you can investigate, but proceed at your own risk:

Changing wlan0 to a full-time AP device: <==pretty safe to try
Setting up a Raspberry Pi as a routed wireless access point

Concurrent operation on Rasbian Stretch: <==not tested
Raspberry Pi 3 Wifi Station + AP

Concurrent operation on Rasbian Buster: <==*not for the faint of heart!*
Raspberry Pi 4 Wifi Station + AP

Adding a dedicated AP device on Rasbian Jessie: <==*confirmed to work*

1: **`iw phy phy0 interface add wlap0 type __ap`**
Where ‘wlap0’ is name of new virtual device (change to what you want)

2: **`sudo nano /etc/dhcpcd.conf`** <<*Add contents below to end of file*

```
interface wlap0                      << or whatever device name you used above
  static ip_address=192.168.2.1/24   << or whatever ip address you want
  nohook wpa_supplicant              <<not to be commented out
```

3: Modify hostapd.conf and dnsmasq.conf accordingly; concurrent operation with wlan0 should work with no further changes

Please post an issue to github if there are other configuration scenarios you have in mind so we can see if they can be supported.