How to Configure your Raspberry Pi for SmartThings Direct Connect

Author: Todd Austin toddaustin07@yahoo.com
Date: November 22, 2020

Introduction

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

Information is divided into the following sections

1. **Preparation**
2. **Install the required software**
3. **Register your device in SmartThings**
4. **Configure your Pi**
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 on a Raspberry Pi that are enabled in SmartThings is quite simple 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 bolded`. Some commands are purposely prefixed with 'sudo' where it is required, so mind those cases. I cannot predict all environments, so it's possible some of these commands may be slightly different for your OS version. Which brings me to my final point:

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. But we will refer to it and follow many 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 wireless 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 "AP mode" or Access Point mode. This 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 ST, so it's a rather fleeting requirement. Nevertheless it is required to successfully complete the add-device process from the SmartThings mobile app.

Even if your Pi supports wireless, it doesn't mean the wireless chip supports AP mode.

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 there, you are golden and can continue to the next step. If not, you'll have to get yourself a more recent Raspberri Pi.

Installing the Required Software

This part of the setup may be the most challenging, but I believe that my documentation here will reduce (maybe not eliminate!) some of the issues you could encounter.

Before you start this project, you might want to 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 version of Pi (Raspberry Pi 3b+ or 4) with Python 3.5 or later already installed.

There are 5 parts to the software configuration:

1. Python update to version 3.5 (if needed)
2. Installing additional system libraries
3. Downloading my Pi-enabling package
4. Cloning and building of SmartThings SDK
5. Installing & configuring the SoftAP software

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 may need to upgrade if you have an older Pi. You can find your Python version by issuing both of these commands in a terminal window:

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

You may have both version 2 and version 3 if you originally began with an older system. But confirm if your version 3 is at least 3.5.

I'm afraid 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. You can really muck up your system if you're not careful, which is why I am not providing those directions here. If you already have Python 3.5 or later, you can breathe a sigh of relief.

If upgrading your system's Python version proves to be problematic, I would recommend skipping the Python-related steps here and instead have someone else run those Python-based tool scripts for you and provide you the output since it is a one-time setup task per device application. You can contact me for help in doing that if you'd like. *You may be able to still build the SDK libraries without Python 3.5, but someone needs to confirm this.*

Once you have Python 3.5 or later installed, you may need to install a couple more libraries that will be required by the above-mentioned SDK tools.   As a precaution, I'd recommend you first confirm your version of pip (Python package installation program) by the following command:

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

Just make sure you are using the Python 3.x version of pip in the commands below (i.e use ‘pip3.x’ instead of just ‘pip’ if needed).

To install the needed Python libraries:

```
pip install pynacl
pip install qrcode
pip install pillow
```

If you encounter any errors regarding missing dependencies, then you may have additional modules to install first.   Just use the **pip install** <packagename> command with the name of the missing library.

Additional Software Dependencies

We’re now going to refer to that 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:

DON’T RUN THIS

```
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
```

You do need these modules, but you have to be very careful how you install them on a Pi. I would recommend updating the modules piece-meal rather than cutting and pasting and running the command as it is provided.   Again, I’m trying to prevent you from mucking up your Pi!   For the Python-related libraries, I recommend first trying **pip install** instead of sudo apt-get especially if you have Python version 2 also on your system.

Whatever you do, **DON’T do an *apt-get python3* or *apt-get python3-pip*.**

This is a bit tricky part of the setup on a Pi, so it may take some trial and error to get everything you need.   If in doubt or having problems, you can Google for some help, but put your trust in the **raspberrypi.org** recommendations rather than random websites.   Or just bypass installing the problem module and wait until you do the SDK build later to see which dependencies are really preventing you from a successful build.

Before you do any of these module installs, I recommend you first:

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

Here is a way to safely approach installing the needed modules:

```
sudo apt-get install gcc make
```

<< you probably already have these (c compiler & make)

```
sudo apt-get install cmake git gperf ninja-build ccache wget libncurses-dev flex bison gperf
```

^-- all these should be safe to install this way

```
sudo apt-get install libffi-dev libssl-dev
```

<< Python-related; installed by apt-get only as far as I know

*For the remaining python-related modules, try using **pip install** first just to be cautious. If they are not available through pip then apt-get will hopefully work ok for you:*

python3-serial, python3-cryptography, python3-future, python3-pyparsing, python3-pyelftools

I have found that I needed one more library that wasn't in the how-to instructions list, so you may need to do this one as well:

```
sudo apt-get install libpthread-stubs0-dev
```

If you get any errors regarding missing dependencies when trying to install these libraries, you may have to install those missing libraries first, then go back and retry the ones above.

If you have passed that hurdle, let's look at the next steps outlined in the how-to instructions…

Do NOT follow the how-to instructions where it says to make python 3 the default. Again, unless you know what you are doing, you could really mess up a Pi with multiple Python versions. Instead, if you have both version 2 and 3 of Python, always use 'python3' or 'python3.x' when running Python programs that require version 3, including pip (e.g. use pip3 or pip3.x instead of pip if you need to). If you have a recent Pi with only Python 3.5 or greater then things are much less complicated.

Also 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 Raspberry Pi Enabling Package

I have created this package for downloading from github. It contains the files you will need to build an Pi-enabled SDK core library, and configure your system for working as a SmartThings direct connected device. You can either clone the repository to your system thusly:

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

…or just download the individual files from the repository (there aren't that many). If you download manually, create a directory **~/rpi-st-device** and place all files downloaded there. *Don't deviate from this directory name, as some scripts depend on it.*

We'll refer back to this package once we get a bit further in the process.

The SmartThings SDK

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

Cloning the SDK

The SmartThings SDK is a rather large set of files that contain the source code to build the core 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

Before you start, just make sure you have sufficient free disk space on your Pi (~224 Mb required).

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**'. You certainly don't have to become an expert on this beast. You will replace a few of the files in this SDK with ones I am providing in my package, and then build the core module. If all goes well, this is a one-time task and subsequently you'll simply link the core shared object module you build with the device application.

Build the core device library

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

```
cd ~/rpi-st-device/     << ensure you are using this directory name; the bash file depends on it
bash ./sdkbuildsetup.sh
```

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. And if you really need to, there is a companion script in my package to put the original SDK files back: undo_sdkbuildsetup.sh

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
```

You may run into some errors if you still have some missing libraries. If so, use **sudo apt install** until you remove all dependency errors.

CONGRATS! You now have a compiled core SDK module with Raspberry Pi support!

Register Test Device in Developer Workspace

The next several steps will get a simple test switch device defined in SmartThings and create 2 **json** files that your device application will need. 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**.

Create Unique Device Serial Number & device_info.json

As described In section **2.6** of the how-to instructions, you must provide a device serial number and public key in the Developer Workspace as part of the device definition process. 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
```

The displayed output can be copy/pasted into the fields in the device identity fields within the Developer Workspace device setup screens as shown in the how-to instructions section **2.6**.

Copy device_info.json into device application directory

The **keygen** tool also creates a **device_info.json** file that needs to be copied to the example application directory (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.

```
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 example directory (you can replace the one that is there). This gets created once you finish registering your test device in the Developer Workspace, so once you completed that, there should be a download link on the configuration Overview page. See section **2.7** in the how-to instructions.

Do NOT follow section **2.8** in the how-to instructions. You can reference sections **3.1 – 3.2** for more information about the onboarding-config.json file and its contents.

Build the Example Application

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

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

Assuming you had successfully built the SDK core library previously and have not made any changes to the example.c code, you should have a successful make.   Note there are several *deprecated function* warnings – you can ignore those.

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

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

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 *run* the application that you have built.

Configuring Wireless Devices

What we need to do now is get your wireless devices set up on your Pi. You have some choices to make because there are multiple ways you could configure your system.

For simplicity, I've laid out 3 configuration options below. For the experts out there, you can come up with different combinations, but this document will focus on these three scenarios.

Before we delve into those options, you need a basic understanding of “AP mode” and why it's important for us. AP mode, or Access Point mode is a special temporary configuration 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 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 an 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* 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.

Wireless setup Options

Here are the three alternatives we’re going to consider for setting up your Pi’s wireless configuration:

1) Keep your wireless interface in AP mode all the time and use your Ethernet connection for all other internet and lan communication (including of course SmartThings MQTT server).

Pros:
- keeps things relatively simple
- leveraging wired connection is an option most IOT devices don't have
- Can use your Pi AP for other things (including internet access for clients in your home)
- Likely the safest option if you are provisioning/running **multiple concurrent** ST devices

Cons:
- could be a security concern having your Pi wireless access point always live
- Pi can't be a normal wireless client any more (until reconfigured)
- possible performance impact to other things running on your Pi
- requires Ethernet connection

2) Configure your wireless interface with **2 virtual devices**: one dedicated to 'managed' mode and another for ‘AP’ mode

Pros:
- No mucking about with your default 'wlan0' device
- Separate virtual device handles AP mode
- Can run both wireless modes simultaneously *(as long as channels are the same)*
- Can turn off AP mode when not needed
- Works also for situations where Ethernet is not available
- Can still use Ethernet if your home wireless AP is down

Cons:
- Some extra configuration steps required (but not much)

3) Switch dynamically between station and AP modes using single default wifi device (wlan0)

Pros:

- No additional virtual wireless device to create/manage
- Suitable for situations where Ethernet is not available
- Can still use Ethernet if home wireless AP is down

Cons:

- Can't run wireless station & AP simultaneously
- Will block other programs on your Pi needing wireless comms while in AP mode

Everyone has different needs, but if it works for you, I'd recommend option #2 above. It seems to be the cleanest in terms of configuration and provides the most flexibility. It's just a bit more work to set up than option #1 (which is probably the simplest operationally speaking), with the key advantage of not having the AP active all the time as in #1.

If you like the idea of having an extra AP you can connect to within your home network, you might like Option #1. Be aware that in order to use your Pi AP to actually get to the internet, there will be a couple additional configuration steps (they're not difficult). *Internet access via the Pi AP is not required for ST device provisioning, so those additional steps to set this up will be addressed as optional later on.*

If you ONLY have wirelesss connectivity to your Pi and you want to keep things as simple as possible, then Option #3 may be a good option, although Option #2 can also work well in a wireless-only config.

Again, the whole AP mode thing is only relevant for a relatively short amount of time when you first are adding your Pi-based device to ST. Once onboarding is complete, AP mode may never have to be invoked again unless you delete and re-add your device on the ST mobile app, or are running multiple separate devices (each of which has to go through the onboarding process). In normal post-provisioning operational mode, your device app is communicating with the SmartThings MQTT server as a run-of-the-mill internet-connected client – either through wireless or Ethernet. So the vast majority of the life of your Pi device will be spent in normal 'managed' (station/client) mode.

If all this talk about switching between AP mode and managed mode sounds like a major hassle, don't despair. The code I have written will do all that automatically for you if you configure it that way. But you do need to understand what's going on under the covers so you can make some choices on what configuration option you want to go with, and it will help if you need to do any troubleshooting.

For purposes of this guide, I'm going to assume configuration option #2, but will point out things as we go if you opt for option #1 or 3.

Wireless Configuration Steps

See 'Alternatives' below if you are not configuring for option 2.

Add a second virtual wireless device that will handle AP mode

First confirm what devices you currently have with the **iw dev** command in a terminal window.
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 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 there as well as mac address and other things. What we're going to do is add a second Interface, or 'virtual' device, with

the following command:

```
iw phy phy0 interface add wlmyap type __ap
```

Take note of the double underline in the last parameter. **'...type<sp> __ap'** indicates that it is an Access Point type device. '**wlmyap**' is the name you are giving to your new virtual device. You can make it anything you like but a standard convention to follow is for wireless device names to begin with 'wl'. Whatever you choose, keep the name relatively short.

Later when we are configuring the SoftAP application (hostapd) you can confirm your new device is working, but for now, do another **'iw dev'** and you should now see your new virtual device listed with **Type AP** and the same mac address as wlan0 (because they are the same physical device).

Take note of the default channel number; it's probably channel 1; we will likely change this later during our hostapd configuration.

Other Alternatives:
Option #1 or 3: the steps above are not required; you will always use your default wireless device wlan0.

Option #1 only - If you want to allow Internet access for clients connecting to your Pi Access Point follow the instructions below:

As mentioned previously, internet access is not required in AP mode by SmartThings. However if you are going to leave your AP mode on as in Option #1 you can optionally enable access to the internet from clients connecting to your Pi's AP. This of coures also requires an ethernet connection. Here are the additional steps:

1. Enable port forwarding

```
sudo nano /etc/sysctl.d/routed-ap.conf
```

Find the line containing the following and remove the comment char '#'

net.ipv4.ip_forward=1

2. Enable IP Masquerade (all traffic routed to internet will look like it's coming from your Pi)

```
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
sudo sh -c "iptables-save > /etc/iptables.ipv4.nat"
sudo nano /etc/rc.local
```

add new line **iptables-restore < /etc/iptables.ipv4.nat**

SoftAP Application Configuration

The last piece of the software puzzle is to set up an application that actually manages the AP mode of the Pi. This is accomplished with two service modules: hostapd and dnsmasq. There may be other options out there that perform similar functions, but these are what I have selected as they have a fairly good history of successfully running as-is on Raspberry Pis.

To install hostapd, issue this command in a terminal window:

```
sudo apt install hostapd
```

hostapd is designed to run as a service, although you can also run it from a command line thusly:

```
sudo /usr/sbin/hostapd /etc/hostapd/hostapd.conf
```

Once we get the hostapd config file set up you may want to try that command to see if it starts ok.

But now let's install the second service application we need:

```
sudo apt install dnsmasq
```

These two services are installed to start at boot time, but we want to prevent that assuming you are using option #2 configuration:

```
update-rc.d hostapd disable
update-rc.d dnsmasq disable
```

If you are going to be running in option #1configuration with AP mode running permanently, you'll want to leave these services starting at boot time. If you find they aren't running after your next boot, try issuing these commands to re-enable auto-start and reboot again:

```
sudo systemctl unmask hostapd
sudo systemctl enable hostapd
sudo systemctl unmask dnsmasq
sudo systemctl enable dnsmasq
```

The next step is to finalize some configuration files

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

```
cd ~/rpi-st-device
bash initsoftapconf.sh
```

This will copy some initial configuration files into the appropriate directories.

<u>hostapd Configuration File</u>

The file **hostapd.conf** should be located in /etc/hostapd/. This file will be dynamically modified by the core library during initial device provisioning.

There are three parameters in this file that you will need to modify, so edit the file now:

```
sudo nano hostapd.conf
```

country_code	make sure it's set for your 2-character country code (US, UK, etc)
interface	change to the wireless interface you decided to use for AP mode e.g. 'wlmyap' if configuring for Option 2 or 'wlan0' for Option 1 or 3.
channel	if using a second virtual interface as in Option 2, this value MUST be set to the same channel as wlan0. If you don't, you will have an unstable wireless radio as it would be forced to to constantly change frequencies back and forth. See also below Sidebar on Wifi Channels. For config options 1 or 3, channel number should be whatever channel number wlan0 is configured for (run **iw dev** to check)

Sidebar on Wifi Channels
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. Here's the thing: the 2.4GHz wireless spectrum overlaps that used by zigbee devices. So it's important to minimize this potential conflict by choosing a wireless channel frequency as far away as your zigbee frequency as 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. Some hubs may have this printed on the bottom also. Reference discussion 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. Typically it's going to be either channel 1 or channel 11. My hub uses zigbee channel 20 which is in the upper mid section of the 2.4Ghz range. I have two wireless routers – one configured for channel 1 which is relatively close to my ST hub, and one configured for channel 11 which is farther away from the hub. That gives room for zigbee to live between them frequency-wise and also keeps the two routers from conflicting with each other. Whatever wireless router your Pi is connected to by default (as a client), use that same channel for the AP mode device configuration. If your Pi 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.

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 set dynamically during runtime. Take note of the ssid that is defined now; you'll see that later when you bring up your AP. Save the file and exit the editor.

wlan0 Network Interface Definition

The info here is provided more for information rather than additional steps to take. If you have a normally functioning Pi with wireless, there should be no issues.

Look for the following file:

```
cat /etc/network/interfaces
```

Note that you may instead have a file in **/etc/network/interfaces/interfaces.d**, so look in that directory and if it looks like that's what your system uses, then examine the file you find there.

The contents you are looking for should look something like this;

```
auto lo
iface lo inet loopback
allow-hotplug wlan0
iface wlan0 inet manual
        wpa_conf /etc/wpa_supplicant/wpa_supplicant.conf
```

This config ensures that (1) your wlan0 device is brought up at boot time, and (2) your wifi security info in wpa_supplicant.conf is used when connecting wlan0 to an access point. I don't recommend any changes to this file. If your system is very different, proceed anyway and hopefully things will work ok.

<u>Set Static IP Address for your AP device</u>

In order for your AP device to work, it must be configured with a **static** IP address. You accomplish this with the following:

```
sudo nano /etc/dhcpcd.conf
```

Add the following lines to the end of the file:

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

Where **wlmyap** is your AP device name.

Substitute the fixed **IP address** you want to use. Be sure this address is <u>outside</u> 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 to our Pi Access Point.

<u>Configuring dnsmasq</u>

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

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

with the following contents:

```
interface=lo,wlmyap                   << interface=wlan0 for config options 1 & 3
no-dhcp-interface=lo,wlan0            << comment out for config options 1 & 3
dhcp-range=192.168.2.2,192.168.2.10,255.255.255.0,12h
domain=wlan
address=/gw.wlan/192.168.2.1  << use same static ip set previously in dhcpcd.conf
```

Change **wlmyap** to the AP device name you chose earlier.

Set the **dhcp-range** (ip address range) you want the Pi AP to assign out to its 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.

Modify the **address** parameter to match the static IP address set in /etc/dhcpcd.conf earlier.

Save the file and exit the editor.

Additional Info

It may be useful to know that your Pi's wireless connection options are stored in a file:

/etc/wpa_supplicant/supplicant.conf

Typically you want to make sure your passwords are stored here so you are not entering them each time you connect to the particular ssid.

This file should only be modified using the **wpa_cli** command. I won't go into details here.

At this point in the process, it would be advisable to reboot your Pi given all the interface configuration files we have modified.

Testing your Pi wireless Access Point

Once you have rebooted we can test that your new Pi Access Point is working. Manually start hostapd with this command:

```
sudo /usr/sbin/hostapd /etc/hostapd/hostapd.conf
```

Note that the use of 'sudo' is mandatory.
Note also that we don't need to start dnsmasq for this test.

If you have configured hostapd to automatically start at boot, it should already be running and you can do the following command to check status:

```
sudo service hostapd status
sudo service dnsmasq status
```

If you started hostapd manually, you may see errors like "failed to create interface mon.wlmyap" or 'Could not connect to kernel driver". These can be ignored. You have success if the last line of the output shows "**AP-ENABLED**". You can verify your AP is live by using your 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 with whatever ssid is currently specified in your hostapd.conf file.

If your wireless interface doesn't seem to be functioning, try issuing these commands:

```
ifdown wlmyap    (or ifdown wlan0)
ifup wlmyap      (or ifup wlan0)
```

Also ensure there is no “soft” block on your 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 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 you have to fix things, remember that reboots may be needed or you need to stop and restart hostapd.

When you are satisfied that everything seems to be in order, **Ctrl-c** out of hostapd if you launched it manually from a terminal window.

Getting ready to run Example app for the first time

We're getting close to the finish line! You've built the SDK core library, defined your test device in the ST Developer Workspace, and set up your Pi to enable AP mode operation. Now there are a couple final steps to take before we can try out the example device application.

Generate a QR Code for your device

This step uses the second Python tool included in the SDK called **qrgen**.

```
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** file that you earlier downloaded to the example directory. After running **qrgen**, you will have a new **.png** file in the qrgen directory that represents a unique QR code for your device app. You will need this soon when you use the ST mobile app to add your device.

Set up Pi config file

This is the last step!

This file is used to defined your unique Raspberry Pi configuration and is expected to be in the directory where you launch your device app from (in our case here, that is the example directory in the cloned SDK). This config file contains some key parameters to ensure successful wifi management.

Copy the sample file from my package into the example app directory:

```
cp ~/rpi-st-device/RPIConfig.conf ~/st-device-sdk-c/example/RPIConfig.conf
```

Bring the file up into an editor for modification:

```
nano RPIConfig.conf
```
<< note 'sudo' not required

There are description comment lines for each parameter you need to set, but I'll elaborate here:

ETHERNET = Pretty obvious, do you have an ethernet connection Y or N
MANAGEWIFI= Set this to Y if config option #2 &3, set to N for config option #1

DUALWIFIMODE= Set to Y if config option #2, otherwise set to N

HOSTAPDCONF= full path name; this shouldn't normally need changing

STATIONDEV, APDEV, ETHDEV= The names of each of your configured devices
For config options #1 & 3, APDEV and STATIONDEV would be the same, for config option #2 they would be unique; If no Ethernet, leave it blank after the =.

QRCODEDIR= directory name of qrcode-created image file (.png); typically either in ~/st-device-sdk-c/tools/qrgen/ or the directory of your device app if you copied it there (e.g. ~/st-device-sdk-c/example/)

Once you have made your modifications, save the file and exit the editor.

Add Device in SmartThings App (Onboarding process)

Now we're ready for the real action!   We will use the SmartThings mobile app to ‘add’ our test device. You can reference section **4.3** in the ST community how-to instructions for much of this process.

I recommend fully reading through everything here and in the how-to instructions before you take action.   It gets kind of exciting when you try this for the first time, and things will go smoother if you know what to expect.

As described in the how-to instructions back in section **4.1**, you must have '**Developer Mode**' enabled in the ST mobile app through the **settings** menu.   Otherwise the device definition that you created earlier in the Developer Workspace won't be listed as a select-able device type.

To ensure ‘Developer Mode’ is on: .

*On the main screen of the mobile app, tap the menu icon in the upper left, then the gray gear icon in the upper right.   Scroll all the way down in the options and make sure the **Developer Mode** switch is **enabled**.   If it's not, enable it, back out, close the app, and restart the app.*

You also need to make sure that the SmartThings mobile app has permission to use your camera, by going into the appropriate settings of your phone’s OS.

Back in the ST mobile app main screen, tap on the menu icon again in the upper left and then tap **Devices**.   Then on the device list screen, tap the **+** in the upper right to add a new device.   An icon for your test device should be shown somewhere at the end of the Devices and Sensors section.

If your test device doesn't show, check Developer Mode setting, exit and restart the app again. I've had this issue often, so it seems to be a bug in the app.   Hopefully you'll eventually see your test switch device listed in the icons.

Tap on your test switch device icon to get to the next screen.

Before you go further, bring up the **QR code** image on the Pi that you created earlier (you can use **gpicview** from command line 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.

You'll soon need to point the phone's camera at this image so make sure it’s fully visible on your screen.

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

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

You should see all sorts of messages flashing by as the software initializes.   It will see that you've never onboarded this device before and go into listen mode waiting for the mobile app to request a connection (AP mode!).   Once the messages settle and you haven't seen any horrible errors, continue back on your phone with the add-device procedure that we started earlier.

On the mobile app you should get to a screen that asks you what Room to put the device in, and then the next screen will turn on your camera and wait for you to point it at the QR code image.

Note that you will also see an option to manually enter the device serial number instead of using a QR code, however I have not been successful getting this to work, so use the QR code.

Once your QR is recognized, you're off to the races and you should see some activity from the Pi device app and within 15 seconds or less you should get a list of local access points presented to you on the mobile app. Choose the one you want the Pi to connect to as a client. (If you have Ethernet, this doesn't really matter that much). Make sure you have any wifi password handy since you'll need to enter that too.

Once you choose the AP in the mobile app, you should soon get a message that the device was successfully added. (And you may be startled to hear your Alexa announce a new device was added!).

You did it!! It worked!!

At this point you can go into the mobile app screens, find your new device in whatever Room you put it in, and try turning the switch on and off. You'll see corresponding messages pop out on your Pi terminal window where your device app is running.

If you'd like you can take a peek at the files that are created by the SDK to see the **provisioning data** that it stores. There is a utility program in my package you can run to do this called **STProv**. Open another terminal window and do this:

```
cd ~/rpi-st-device
./STProv <directory>
```

<< where 'directory' is where your device app is; in this case it is ~/st-device-sdk-c/example
If you leave directory blank, the current directory is assumed

Note you may have to change permissions on the file to make it executable:

```
sudo chmod +x 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).

Where to go from here

Once you have successfully onboarded your device, you are able to stop (Ctrl-C) and restart the Pi device application and it will simply reconnect to the ST 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 ST mobile app will show that device with an "Offline" status.

If you delete the device from the mobile app, ST 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.   At least that is how the example app is coded.

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

After that, try your hand at coding your own app for other device types. You will have to define those device types in the Developer Workspace and get the new json files that are unique to each instantiation.

Finally, announce your new devices on the Sametime community and raspberry pi communities if you want to share with others.   And if you really want to get professional, you could could pursue official device certification from SmartThings (I would hope they would do that for Pi-based devices!).

"I don't want to code in C; I want to use Python to write my device application!"

Right now only C is supported.   However my intention is to create an api shell for Python programs. Stay tuned!