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.

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 AP mode is required to successfully complete the device onboarding process from the SmartThings mobile app.

Even if your Pi has wireless, it doesn't necessarily mean the wireless 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 there you are good to proceed.

Installing the Required Software

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 Pi model (Raspberry Pi 3b+ or 4) with Raspberry Pi OS Version 10 ("Buster"), 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 command:

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

Just 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 either clone the repository to your system thusly:

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

…or download the individual files from the repository. 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 shortly.

Additional Software Dependencies

We're now going to reference to 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:

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: 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' or 'phython3-pip' unless you really know what you are doing; this could cause problems especially on older Pi's.

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

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

I'll present the 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 (listed further below).

1) These should already be installed on an up-to-date Pi:

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

2) These will probably need to be installed:

```
sudo apt-get cmake gperf ninja-build ccache libcurses-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 libpthread-stubs0-dev
pip3 install --user pynacl
pip3 install --user qrcode
```

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

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

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 get any 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 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 SDK is a 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**.

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, 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
./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. 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 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 defined in SmartThings and create 2 **json** files that your Pi-based device application will need. For this you will use the Samsung SmartThings Developer Workspace, so get signed in from your browser. 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**.

Additional info in support of these steps follow:

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 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.

```
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, you can replace the file that is already there). This file gets created when you finish registering your test device in the Developer Workspace, so once you completed that, there should be a download link on the upper right of the configuration 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 that you placed in 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. You will need this later when you use the SmartThings mobile app to add your device.

Build the Example Application

If you have the **device_info.json** and **onboarding_config.json** files stored in the SDK example 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 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 pretty much instantly have a successful make. Note there may be 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 APIs are used from a device application. This example implements a simple on/off switch.

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 *run* the device application that you have built.

Configuring your Pi to support Wireless AP mode

You will need an application that implements the SoftAP capability (AP mode) on the Pi. This is accomplished with two service modules called **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 proven history of successfully running as-is on Raspberry Pis and are fairly lightweight.

All SoftAP-related install and config tasks can be accomplish quite quickly by using a bash script included in the RPI package.

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

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
   e.g. wlan0, wlap0, wlmyap, etc.
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
   must 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.

But for those of you who just want to get on with it, 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.*

Once you 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, but the script will automate many of those final testing steps for you.

- Start of Manual Configuration Steps - *(Skip to page 14 if you ran the automated script)*

Configuring Wireless Devices

What we'll describe here is how to get your wireless devices set up on your Pi. 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 a **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* 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 without disrupting your normal wireless client connection.

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 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 use wlan0 as both our client and AP mode device. The mode will be changed dynamically during runtime.

Set Static IP Address for your AP device

In order for your AP device to work, it must be configured with a **static** IP address. We will do this by modifyig the dhcpcd.conf file:.

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

Add these lines to the very end of the file:

```
interface wlan0
   Static ip_address=192.168.2.1/24
#  nohook wpa_supplicant       << note the comment symbol (#); include it!
```

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.

The last line (`nohook wpa_supplicant`) although commented out by default, will be modified at runtime. Removing the # and restarting dhcpcd service - or rebooting - will change wlan0 to AP mode; conversely, commenting that line out and restarting dhcpcd will re-enable normal wireless client mode. For now we'll leave it for normal wireless client mode.

Save the dhcpcd.conf file and exit.

SoftAP Services

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

```
sudo apt install hostapd
sudo apt install dnsmasq
```

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

```
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 should only be used if Ethernet is also connected.

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
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 MUST be set to the same channel (1-14) as wlan0 is normally connected with as a wireless client. If you don't do this, you will have an unstable wireless radio as it would be forced to to constantly change frequencies back and forth whenever AP mode is enabled. 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. Reference 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; you'll see that later when you bring up your AP for testing.

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

dnsmasq Configuration File

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.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.

Let's first confirm that your Pi is now using it's new static IP address:

```
ip route
```

This will display the IP addresses being used by each of your devices (Ethernet & wireless). wlan0 should show the static IP address you configured in the dhcpcd.conf file earlier. If not, re-check dhcpcd.conf, fix and restart dhcpcd or reboot if needed.

Also 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.

There is a bash script that will manually start hostapd for testing, and return your wireless back to client mode when it is done.

Warning: this next step will turn off your wireless client operation. If you are connected as a remote console to your Pi, it needs to be via Ethernet; otherwise you'll need to 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 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.

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

Troubleshooting

If your wireless AP interface doesn't seem to be functioning, 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
```

(Substitute whatever name you gave your virtual wireless device)

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 servce.

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

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 inventory. You can reference section **4.3** in the how-to instructions for visuals of much of this process.

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 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, so take care of that now by going into the appropriate settings of your mobile phone's OS.

Back in the ST mobile, tap on the menu icon again in the upper left of the main screen 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 group of icons.

Tap on your test switch device icon to get to the next screen which will list your testing "products". Note that you may see duplicate listings.

Before you go further, bring up the **QR code** image on the Pi that you created earlier. 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. 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 launch 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 determine that you've never onboarded this device before and go into listen mode and wait for the mobile app to initiate onboarding. 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 the mobile app, tap your listed testing product then tap Start to proceed. 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 on this screen 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.

Point your camera at the QR code displayed on your screen, and once it is recognized, there will be a short pause and then you should see some activity from the Pi device app, and you should get a list of local access points presented to you on the mobile app. This is good: it means your Pi's AP mode is working, and your phone and the core SDK on your Pi are talking. Choose the SSID you want the Pi to (re-)connect to as a regular client* (not to be confused with AP mode). Make sure you have any required wifi password handy since you'll need to enter that too.

**Be sure to select an ssid on a router who's channel is the same as your AP device. Also, take note that if you have an operational Ethernet connection, it will normally take precedence over wireless for any client-mode communications anyway, so this AP selection is moot where Ethernet is available.*

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.

At this point you can use the mobile app to show your new test device in whatever Room you put it in, and you can turn the switch on and off. You'll see corresponding messages pop out on your Pi terminal window where your device app is running. *Sweet!*

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 messages will contain the characters **[rpi]**.

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 these files. Open another terminal window and 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).

What you'll see when you run this utility are the contents of several files that show the 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 by the mobile app during onboarding. 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.

If 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 your device, you are able to stop (Ctrl-C) and restart the Pi example 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 ST 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. At least that is how the example app is coded. You can re-add the device with the same serial number (and QR code), but if you want to try running multiple device apps, they will each need their own serial number.

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 to 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 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. Please note that at the current time, the API is for C language apps only. *I hope to create a shell API for Python in the not-to-distant future.*

Use the Developer Workspace to define your new projects and device(s) and remember that each 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 for each device, since they each require their own device_info.json files.

Reference the example app **Makefile** to see how your 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: **softapstart** and **softapstop**. Be sure these are present either in the ~/rpi-st-device directory or in your device's application folder. The need for these files will hopefully be removed in future updates, but are here for now in case a root password is required for the systemctl commands to start and stop the hostapd and dnsmasq services during provisioning.*

Multiple devices with the same profile

If you create multiple test devices under the same device profile as defined in the Developer Workspace, then those devices share the same onboarding_config.json file. However they will still each need their own device_info.json and QR code since they have unique serial numbers.

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!).

SDK Updating

You may want to occasionally update the SmartThings core SDK to make sure you have the latest changes. After you do that (e.g. using git fetch), be sure to re-run the script that modifies it with the required Pi files before you re-build the SDK library:

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

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)

Default Setup Files

RPIhostapd.conf	hostapd config
RPIdnsmasq.conf	dnsmasq config

Tools

STProv	program to display SmartThings provisioning data store files
testhostapd	bash script to put wlan0 into AP mode and test hostapd service
undo_sdkbuildsetup	bash script to return SDK to original state (after running sdkbuildsetup)

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

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

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 and will rely on a wireless connection to your home wireless router.

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 simple 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 and 4 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 with 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
√		√	Asssme full-time AP mode started at boot; Ethernet 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 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. This option also avoids having to define and manage a secondary virtual wireless device.

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 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 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 more recent Rasbian 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.