How to Configure your Raspberry Pi for SmartThings Direct Connect

Author: Todd Austin toddaustin07@yahoo.com
Date: December 2, 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 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, now look all the way 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. We want to run both station mode and AP mode at the same time, if possible. You should have a line that looks like this:

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

This indicates your 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. 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. If your hardware supports AP per the Supported interface modes, but does not support simultaneous operation, then you should still be ok; your applications should still work, but any normal wifi client operation will be temporarily interrupted during new device onboarding when AP mode is required.

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 4 parts to the software configuration:

1. Python update to version 3.5 or later (if needed)
2. Installing additional required software
3. Cloning and building of SmartThings SDK
4. 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 Pi model (pre-version 4). But confirm if your Python 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.*

As a precaution, I'd also recommend you 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 next section below (i.e use 'pip3.x' instead of just 'pip3' if needed).

Additional Software Dependencies

We're now going to refer 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, 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 - just in the spirit of preventing you from mucking up your Pi!

For the Python-related libraries, I recommend first trying **pip3 install** instead of sudo apt-get especially if you also have Python version 2 on your system.

Whatever you do, **DON'T do an *apt-get python3* or *apt-get python3-pip*** unless you know for certain it is safe to do on your system.

This is a 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 Linux or even Debian websites. If you are having trouble with a particular module, just bypass installing it for now 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 cautionary approach to 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 but definitely need installing by apt-get

*For the remaining python-related modules, try using **pip3 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 in addition to the modules specified above from the how-to instructions , I needed these additional ones on a Raspberry Pi 3b (you may have others):

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

If you get any errors regarding missing dependencies when trying to install any of these libraries, you may have to install those missing ones first, then go back and try again. It will be a bit of an iterative process, but you'll get through it!

Once you have passed that hurdle, 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 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/todd_austin/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 once we get a bit further in the process.

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**. You certainly don't have to become an expert on this beast. You will use a bash script to replace a few of the files in this SDK with ones I am providing in my package, before you build the core library module. If all goes well, this is a one-time task and subsequently you'll simply link the library module you build **(libiotcore.a**), with each of your device applications.

Build the core device library

Before we issue the make command to build the object module, 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/
bash ./sdkbuildsetup.sh
```

<< ensure you are using this directory name; the bash file depends on it

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 install** until you remove all dependency errors.

CONGRATS! You have now created the core SDK archive 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 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 (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 (**~/st-device-sdk-c/example**) You can replace the default onboarding_config.json 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 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 application that you have built.

Configuring your Pi to support Wireless AP mode

We 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.

In addition to installing and configuring these services, we must set up a new virtual wireless device to support the new AP mode.

All these tasks can be accomplish quite quickly by using a 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) New wifi device name
   e.g. wlap0, wlmyap, etc; it's up to you
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 completed running the script, you can skip to the section "**Testing your PI wireless Access Point**" in this document.

- Manual Configuration Steps - *(Skip 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.

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 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 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.

Add a new virtual 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. What we're going to do now is add a second interface, or 'virtual' device, with the following command:

```
iw phy phy0 interface add wlap0 type __ap
```

Take note of the double underline in the last parameter. '**...type<sp>__ap**' indicates that it is an Access Point type device. '**wlap0**' 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 (e.g. wlmyap, wlpiap, etc.).

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).

We will change your new device channel number later during our hostapd configuration.

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 two commands:

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

The next step is to finalize some configuration files

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

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

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

hostapd Configuration File

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 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=*wlap0* change to the virtual AP-type device name you created earlier
e.g. 'wlap0'

channel=*n* This value MUST be set to the same channel 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 set 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/hosapd.conf) and exit the editor.

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. You accomplish this with an entry in your /etc/dhcpcd.conf file. Before we make that change, it would be a good idea to save the current one in case anything goes wrong:

```
sudo cp /etc/dhcpcd.conf /etc/dhcpcd.ORIGconf

sudo nano /etc/dhcpcd.conf
```

Add the following lines to the end of the file:

```
interface wlap0
    static ip_address=192.168.2.1/24
    nohook wpa_supplicant
```

Where *wlap0* is whatever your AP device name is that you defined earlier.

Replace the **static ip_address** value with the one 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.

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=*wlap0*
no-dhcp-interface=lo,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*

Change *wlap0* to whatever 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 (without the '/24').

Save the dnsmasq.conf file and exit the editor.

Reboot!

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

```
sudo reboot
```

If you really don't want to reboot, you can try restarting your networking service:

```
sudo systemctl stop networking
sudo systemctl start networking
```

You can now 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.

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 automated setup script, or you have followed the manual instructions. Either way, by now you should have a fully configured system to enable SoftAP capability (wireless Access Point) with the hostapd and dnsmasq services.

Upon starting the hostapd service, 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 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 is set for initial testing in the hostapd.conf file.

Troubleshooting

If your wireless interface doesn't seem to be functioning…

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. `wlap0  <BROADCAST,MULTICAST,UP,LOWER_UP>`

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

```
ifdown wlap0
ifup wlap0
```

(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.

When you are satisfied that it working, **Ctrl-c** out of hostapd.

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 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 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 section.

If your test device doesn't show, double-check Developer Mode setting, exit and restart the app again.

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 Pi wifi-specific 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 asking for 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.

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. If you try to re-use the example app folder for a new app, you're likely to overwrite your example device_info.json file and once it's gone, it's gone and you won't be able to re-make that particular device without generating a new serial number and adding it into your Developer Workspace project.

If you try creating 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.

Reference the example **Makefile** to see how your application needs to be built.

Finally, announce your new devices on the SmartThings community and Raspberry Pi communities if you want to share with others. 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!).