**Preface**

This tutorial assumes some background knowledge of programming, but no prior experience with Raspbian (or any Linux distribution), GPIO microprocessors (Pi, Arduino, etc), or Python. References may be made to previous tutorials.

My main programming languages are Java, C#.NET, and VB.NET. You may notice some slight differences between my Python code and the standard conventions and formatting for Python. This should not affect your understanding of the code.

**Goal**

Send a cloud-to-device message using Microsoft Azure’s IoT Hub that alters the physical state of an LED attached to a Raspberry Pi 3.

**Getting Started**

You’ll need to set up the proper coding environment for this tutorial first. The instructions for doing so come with the Pi’s online documentation and are described in *Part 0* of the tutorial, so here are the basic steps:

1. Install Raspbian Jessie
2. Check for updates to your distribution
3. Physically prepare your device with a breadboard connected to the Pi’s pinouts or some other form of wiring.

Now that your Pi is up and running, we need to install the IoTHub library. There are two methods to do this; either you can access Github (<https://github.com/Azure-Samples/iot-hub-python-raspberrypi-client-app>), download the repository as a Zip, and extract the files, or; you can clone the repository via terminal.

sudo apt-get install git-core

git clone <https://github.com/Azure-Samples/iot-hub-python-raspberrypi-client-app.git>

To compile the Python library, run the following (adjusting the directory path as necessary):

cd ./iot-hub-python-raspberrypi-client-app

sudo chmod u+x setup.sh

sudo ./setup.sh

**Tips:**

* You can change directories using \*wildcards if you have no other similarly-named directories. In the above example: “cd ./iot-\*” will get you to the correct directory.
* The base directory “./” here refers to “/home/pi”. You’ll find the new folder and files here if you used git clone. Otherwise, replace it with your downloaded and extracted zip folder location.
* As the Github documentation states, you may find that iothub\_client.so gets stuck while compiling. To prevent this from happening, open a second terminal and prepare to run   
  “free -m” whenever you see the compiling slow down. It is possible that having an active VNC connection causes this problem; try viewing your Pi via HDMI or control it via SSH if the freezes persist. If the system does freeze completely, it is safe to forcefully reboot the Pi by disconnecting and reconnecting the power source.

For the purposes of this tutorial, either method will work, since we will be pulling a file from the example directory to use.

You’ll also need to set up an IoT Hub on Azure for this tutorial. The steps for that are extensively detailed in this Microsoft Helpdoc: <https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-raspberry-pi-kit-python-get-started#create-an-iot-hub> Stop when you reach the section “Set Up Raspberry Pi”.

If you can’t get to the webpage at the moment, follow these basic instructions. Create a New Resource, namely an IoT Hub. Set the proper subscription (a Free one will work, or a Pay-as-you-go) and the region closest to you. Continue with setting up your desired pricing and scale, and finish the creation. It will take some time to deploy. Afterwards, open your IoT Hub and select “IoT devices” in the menu, under Explorers. Add a new device, and take note of this new device’s connection string.

*Make sure you keep your* ***IoT Hub Device Details*** *page open*, either on a separate device or on your Pi. You’ll need to use some values to set up the proper authentication with your Pi later.

**The Library**

After compiling, you’ll find a file “iothub\_client.so” approximately 3.6 MB in size, described as a shared library. This is the Python-ready library compiled from the original C code. You’ll also find, nestled in the directories “device/iothub\_python”, the Python wrapper source code. It serves as a decent reference for all the methods and parameters you will need to write a IoTHub device.

This iothub\_client.so library (henceforth referred to as ‘the library’) contains the networking protocol and various classes that will shortcut us into immediately writing useful applications. Although there are a lot of functions and callback methods at your disposal, for this starter tutorial we’ll only be using a few of them. Of course, I strongly encourage you to explore the many facets of the library and to introduce new code wherever you fancy – but I won’t be, for now.

**Setting Up the Code**

**Important note:** Python 3 uses a different set of method calls than Python 2 for initializing stuff. This library and tutorial will *not* work with Python 3. Ensure you’re using Python 2. In Raspbian Jessie, both options will be available under the Programming menu item – ensure that you select “Python 2 (IDLE)”.

Once you’ve launched Python, you’ll need to create a new document via File>New File. Let’s set up the file with the important import statements:

from iothub\_client import IoTHubClient, IoTHubClientError,

IoTHubTransportProvider, IoTHubClientResult, IoTHubMessage,

IoTHubMessageDispositionResult, IoTHubError,

DeviceMethodReturnValue

import sys

import time

import Pi.GPIO as GPIO

**Tip:** Don’t simply copy-and-paste the code snippets here. Due to special formatting, they might not behave as you’d expect in the IDE.

The most important iothub\_client imports we’ll be using here are IoTHubClient, which is the client class that provides messaging capabilities, and IoTHubMessageDispositionResult, which is oddly important as we’ll see later on.

Let’s also create some variables to keep our code clean. We’ll store your device’s Connection String, as well as the state of the (not yet connected) LED.

# Replace the string with the string provided by Azure

CONNSTR = “HostName=hubname.azure-devices.net;...;...”

LIGHTON = True

CLIENT = 0

The connection string should have the format “HostName=…;DeviceId=…;SharedAccessKey=…”. If this is not correct, your client will fail to initialize.

**Tip: If your client fails to instantiate and throws an exception but the string format matches the above, ensure you have copied the key from your *Device Details* page and not the IoTHub page. The key formats are identical but serve different purposes.**

Alternatively, if you’d prefer, you can create three variables to store each part of the connection string. However, the client only accepts a connection string formatted as described above.

Now that we have our document set up, we can start creating the client that will communicate with our Hub. I recommend saving and Running the module periodically to ensure your code works. Go ahead and give it a try.

If you’ve followed the steps above, you may notice that Python doesn’t recognize your imports! This is because you haven’t created a copy of the library in the same directory as your Python file. Find where “iothub\_client.so” resides, and make a copy of it into the directory where you saved your code. Recall that this should be in the repository directory, and will not exist until the “setup.sh” script has been run.

Some files and folders may be hidden, and you may need to run an elevated File Manager to see them. If that’s the case, execute “sudo pcmanfm” in a terminal.

Try running the module again, if it failed at first. It should run fine, and stop execution immediately. If not, double check the error and resolve it as necessary.

**Creating a Client**

In order to keep our code clean and organized, we’ll be making functions for each task. The first task at hand is to initialize an IoTHubClient.

def setupClient():

global CONNSTR # We need to use this previously declared variable

# Create a client with the connection string and use MQTT

client = IoTHubClient(CONNSTR, IoTHubTransportProvider.MQTT)

# Optional key-value settings

client.set\_option(“product\_info”, “RPI-Python”)

return client

This function gives us an IoTHubClient object that can send and receive messages. Let’s create the main method to run this code as well. Finally, at the end of the file, have it run main(). Remember, Python is an indent-sensitive language. Your code should look like this:

def main():

global CLIENT

# Instantiate a client and store in global CLIENT variable.

# If CLIENT isn’t global, the object gets disposed immediately

# after exiting the method.

CLIENT = setupClient()

# Begin code execution

print (“Starting IoT Hub Client...”)

main()

That’s not very exciting, but give it a Run anyways. If your connection string is correct and your Pi has internet access, your code will run and exit without problem.

What we have now is an established connection to the cloud. All we need to do now is receive a message from the Hub, and to do something when that happens.

**Setting up the Callback methods**

The IoTHubClient utilizes callback methods to trigger events, and a received message will need its own callback method. Let’s set that up.

def gotMail(message, context):

# These arguments are required by IoTHubClient.

# ‘message’ contains the details of the message

# sent from the server.

# ‘context’ is the user-provided context of an undefined type

# Read the bytes and deserialize it into a string

message\_buffer = message.get\_bytearray()

size = len(message\_buffer)

msg = message\_buffer[:size].decode(“utf-8”)

print (“Server says: %s” % (msg))

**Tip:** Python’s string formatting is as follows: “%s abcdef %s …” % (arg1, arg2, …)

Now that we have this function, we need to inform the IoTHubClient. We’ll do that right as we instantiate a client in setupClient(). Modify your previously written setupClient method.

def setupClient():

global CONNSTR, gotMail

# Create a client with the connection string and use MQTT

client = IoTHubClient(CONNSTR, IoTHubTransportProvider.MQTT)

# Optional key-value settings

client.set\_option(“product\_info”, “RPI-Python”)

# Set callback method for messages

client.set\_message\_callback(gotMail, 0) #(context := 0)

return client

**Tip:** While Python is indent-sensitive, it is not line-sensitive. You may adjust spacing between code, comments, and functions as you see fit.   
**Note:** The global variables declared has been updated to include gotMail.

Give the code a Run. Again, it seems to stop execution almost immediately, even without throwing any errors. That’s because upon reaching the last line of code, Python doesn’t account for the fact that we’re waiting for the server to send us a message. Let’s create some code to keep the application running even if there’s nothing left to do.

...

# Begin code execution

print (“Starting IoT Hub Client...”)

main()

while True:

time.sleep(1)

**Tip:** time.sleep(n) takes *n* in seconds. It can take floats as well, to get fractions of a second.

Ah, the infamous “while True” loop. This will periodically (every one second) set the main thread to sleep, again and again. Unfortunately, this means your code won’t ever come to a natural stop. We can still halt code execution with Ctrl+C in the Python shell, however.

Run the code again. You’ll notice that the shell no longer immediately informs us that it has RESTARTed. This means our code is continually running, even if there’s no physical indication of it doing so. Let’s leave our code running for a bit and turn towards the Azure page for your IoT Hub.

**Sending a Message**

If you have your **Device Details** page open, read on. If not, you can find it via Home>Your Hub>IoT Devices>Your Device. Remember, this is the ***page with your device’s Connection String*** on it, not the Hub’s connection string.

On the upper menu of Device Details, you’ll see a button labelled “Message To Device”. Opening that will bring you to a simple form that supports sending a message and some additional information. Type something into the Message Body portion of the form, and click Send Message at the top.

**Note:** You cannot transmit “unidentified ASCII symbols” through the messaging portal.

If everything is set up properly, your Python shell should indicate that the server has sent a message. Congratulations, you’ve just sent data from your internet browser to your Raspberry Pi.

**Setting Up the LED**

Now that we’ve established that you can make a connection and transfer information from server to client, let’s make this a more visual experience. Prepare a standard LED and a resistor of about 100 ohms, and locate a free pin on your Pi3. For this tutorial, we’ll be using ***GPIO pin 6*** as labelled by the Pi3 pinout, but you may choose any number that supports GPIO.

Connect the LED’s positive terminal (the longer leg) to pin 6, and connect the negative terminal to the resistor. Finally, connect the resistor to ground (GND). This should be a circuit in series, not parallel.

The Pi3 GPIO pinouts all deliver 3V – if you see that the LED is extremely bright, remove it immediately and replace your resistor with a higher value resistor. If your LED is dim, reduce your resistor value. If your LED suddenly turns off, disconnect and test it with a 1.5 V source (AA battery or similar) – it may have burned out.

**Note:** If you’re unfamiliar with resistors or circuitry, follow all of the given values and pin selection exactly. This will help reduce the chance of malfunction.

Now that the circuit is complete, we need to program the pin in our code.

...

# Set pin numbering mode

GPIO.setmode(GPIO.BCM)

# Set pin 6 (or whatever number you’re using)

GPIO.setup(6, GPIO.OUT)

# Begin code execution

...

We’ve informed our Pi that we want to use GPIO 6 as an output pin. Now, we need a way to set the signal to our desired position. Let’s make some more methods. Remember that these method declarations must be written before they are called, so it helps to group all your declarations at the top of the document.

def turnLightOn(): # Regardless of state, turn the light on

global LIGHTON

# Remember the set state

LIGHTON = True

# Set pin 6 to HIGH voltage (3V)

GPIO.output(6, GPIO.HIGH)

print (“LED on!”)

def turnLightOff(): # Regardless of state, turn the light off

global LIGHTON

#Remember the set state

LIGHTON = False

# Set pin 6 to LOW voltage (0V)

GPIO.output(6, GPIO.LOW)

print (“LED off.”)

def toggleLight(): # Toggle the light state from one to the other

global LIGHTON

print (“Toggling...”)

if LIGHTON:

turnLightOff() # if ON then TURN OFF

else:

turnLightOn() # if OFF then TURN ON

To keep our code deterministic, let’s make sure we set a default state for our LED. Also, let’s make things interesting by having the LED blink.

...

# Begin code execution

turnLightOn() # Ensure start state is ON

print (“Starting IoT Hub Client...”)

main()

while True:

toggleLight() # Blink the LED every second

time.sleep(1)

Run your code - your LED should be blinking every second. If not, double check your wiring and pin number. Also keep track of your indents. If it helps, you can include ‘fake’ endings for your methods by using comments:

def func():

code code code

# end func

**Continuous Execution**

Rather than using Ctrl+C to stop code execution, let’s send a message from the Hub to your Pi and see what happens. You’ll notice that the blinking stops and your code stops as well, for some reason. As it turns out, the callback method for receiving messages needs to return a value in order for your thread to remain running. Let’s go back and edit that code.

def gotMail(message, context):

...

print (“Server says: %s” % (msg))

#Allow code execution to continue

return IoTHubMessageDispositionResult.ACCEPTED

Running this should now allow your LED to continue blinking even if multiple messages are received.

**Interactive LED**

Finally, let’s hook up the Hub messages and our LED. We want to be able to set the state of our LED through the server by sending a simple string.

First, we need to disable the blinking LED. You can do this by commenting it out with a pound (#) or deleting the line within the ‘while True’ loop.

Unfortunately, Python doesn’t have a Switch-case structure, so we can’t use that to set up our commands. For now, we’ll stick with the If-ElseIf structure to determine what the incoming message says.

def gotMail(message, context):

...

# print (“Server says: %s” % (msg))

# Message interpretation

if msg == “turn on” or msg == “on”:

turnLightOn()

elif msg == “turn off” or msg == “off”:

turnLightOff()

elif msg == “toggle”:

toggleLight()

else:

print (“I don’t know how to ‘%s’.” % (msg))

return ...

**Tip:** Python doesn’t use “elseif” or “else if”, but rather, “elif”. It also doesn’t use && or ||.  
 **Note:**  Notice that the unnecessary initial print statement has been commented out here.

Run the module, and start sending messages from your Hub webpage. You should be seeing your LED behave exactly as you tell it to; sending “turn on” should light the LED up, and “toggle” should switch between on and off. If you’re seeing the Python shell announce that it is turning the LED on and off but you don’t see it happening in reality, check your wiring and ensure your LED hasn’t burnt out.

**Conclusion**

Congratulations! You’ve set up an Azure Internet of Things Hub, connected your Raspberry Pi 3 as an IoT Device, received server messages sent from the Hub webpage, and altered the state of an LED via specifically written commands.

**What Can I Do Next?**

Explore the Python language and clean up the code you’ve written. Add detail into your code, such as the omitted ‘context’ fields. Attempt to connect a second LED to another GPIO pin, and create a command interpreter that can control a specific LED’s state given its number. Add some error handlers with Try-Except blocks to automatically handle exceptions.