# http://www.mcwane.com/upl/images/family-of-companies/logos/synapse-wireless-8cccdd3d.pngSoftware Guide for E20 Example 4 – Interfacing to Initial State

For information about “what this example does”, please refer to the corresponding Quick Start Guide.

The Quick Start also includes high-level instructions on how to install this example.

Full source code for this example is available on Github here: <https://github.com/synapse-wireless/demo-kits>

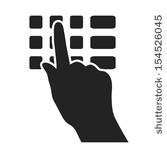
## What’s Going On Here?

The diagram on the next page should put everything in context. It shows the sequence of events that takes place between pushing the button on one of the SN171 boards and seeing the update on Initial State. Color is used to show the progression of the events: from the SN171 to the Bridge Node inside the E20, to the SNAP Connect instance running inside the E20, to the Initial State Cloud.

The sequence of events is numbered, the text boxes designate events (“what happened”), and the callout balloons provide extra commentary on “why those things happened”.

The *next* section of this document is a walk-through of the E20 source code.

**NOTE** – this example re-uses the SNAPpy script from “E20 Example 1”. Refer to the Example 1 documentation for a walk-through of that SNAPpy script.



1

2

HOOK\_GPIN event is created

The SNAP core firmware generates this event because the SNAPpy script previously called monitorPin()

3

Function pin\_event() gets called

Function pin\_event() was previously designated as the handler for HOOK\_GPIN events via @setHook()

4

Function send\_status() gets called by pin\_event()

5

A status() RPC call goes out over the airwaves, with a Time To Live (TTL) of 3 hops

The script made this happen by calling the mcastRpc() built-in

6

The Bridge Node *inside* the E20 receives the status() RPC via its radio, and forwards it to the containing E20’s SNAP Connect application (main.py) over an internal serial interface.

Because that’s what Bridge Nodes do

7

SNAP Connect receives the status() multicast RPC packet over that serial interface

Because main.py previously called API function connect\_serial()

8

Function \_on\_status() gets called

Function \_on\_status() in main.py was specified as the handler of ALL “status” RPC calls using a function dictionary passed to the \_\_init\_\_() function

9

Function \_on\_status() builds a JSON status report from the raw RPC parameters, builds a HTTP Request from that JSON, and then sends the HTTP Request to Initial State via TCP/IP

10

Initial State receives the HTTP Request over TCP/IP, and updates the website. It also sends a HTTP Response back to SNAP Connect to let it know the HTTP Request was received

11

Function \_handle\_request() in main.py gets invoked with the response from Initial State

## Source Code Walk-through (main.py)

The following code walk-through intersperses commentary (in this font and color) with source code (in this font and color).

First several standard Python libraries are imported.

import os

import sys

import json

import binascii

All SNAP Connect applications must import the SNAP Connect Python library.

from snapconnect import snap

This example uses the Tornado ioloop for scheduling. For more details, refer to the separate document **Using Tornado’s ioloop for SNAP Connect’s scheduler**. This example also uses the Tornado HTTP libraries.

from apy import ioloop\_scheduler

import tornado.ioloop

from tornado import httpclient

Here is where you are supposed to configure main.py with YOUR SNAP Addresses and YOUR Initial State keys.

Please refer back to the Instructions section of this document. I will go ahead and mention that binary addresses in Python are usually handled with a \x prefix. For example, a SNAP Address of 12.34.56 would be “\x12\x34\x56” in Python. **However**, here the code us going to *take care of that for us*, so 12.34.56 should actually be entered as “123456” (**no \x prefixes!**)

# TODO: Replace these with values from your own Initial State account and buckets

# We want to map Initial State buckets to nodes

INITIAL\_STATE\_BUCKETS = {

"xxxxxx": "enter unique Initial State bucket key here",

"yyyyyy": "another unique Initial State bucket key here"

}

ACCESS\_KEY = "enter unique access key here"

The constant below will be used to tell Tornado how often to give SNAP Connect a turn.

SNAPCONNECT\_POLL\_INTERVAL = 10 # milliseconds

The following “platform” check allows you to use this main.py file on an E20 Gateway or on a Windows PC.

if sys.platform == "linux2":

# E20 built-in bridge

serial\_conn = snap.SERIAL\_TYPE\_RS232

serial\_port = '/dev/snap1'

snap\_addr = None # Intrinsic address on Exx gateways

snap\_license = None

else:

# SS200 USB stick on Windows

serial\_conn = snap.SERIAL\_TYPE\_SNAPSTICK200

serial\_port = 0

snap\_addr = '\x00\x00\x20' # SNAP Connect address from included License.dat

cur\_path = os.path.normpath(os.path.dirname(\_\_file\_\_))

snap\_license = os.path.join(cur\_path, 'License.dat')

Wrapping functionality up into a Python class is a common encapsulation technique.

Please note that SNAP Connect can also be used with a purely functional programming style. SNAP Connect supports Object Oriented Programming (OOP) but does not require it.

The InitialStateExample class is very simple, and only has three methods: \_\_init\_\_(), \_on\_status(), and \_handle\_request(). Each will be discussed inline with the code.

class InitialStateExample (object):

All Python classes have a customizable “init” function named \_\_init\_\_() in which to perform any needed initialization.

def \_\_init\_\_(self):

"""

Initializes an instance of InitialStateExample

:return:

"""

You always have to tell SNAP Connect what functions you want to be invokable via Remote Procedure Call (RPC), and what you want them to be named. Tthe “RPC” names are not *required* to match the “real” function names, which comes in handy when using Python classes. Here we are filling in a function dictionary with a single entry. If your program had more functions that you wanted to be invokable via RPC, you would add them here.

snap\_rpc\_funcs = {'status': self.\_on\_status}

Importing the SNAP Connect library (up above) gives us access to the API, but it does not automatically create an actual SNAP Connect instance. We do that next.

# Create SNAP Connect instance. Note: we are using Tornado's scheduler.

self.snapconnect = snap.Snap(

license\_file=snap\_license,

addr=snap\_addr,

scheduler=ioloop\_scheduler.IOLoopScheduler(),

funcs=snap\_rpc\_funcs

)

Notice the use of an alternate scheduler (scheduler = …), and the use of the function dictionary created above.

SNAP Connect also has to be told which physical interfaces to use. For this example, we are connected *serially* the SM200 SNAP Module inside the E20, but be aware that SNAP Connect can also make connections over TCP/IP.

self.snapconnect.open\_serial(serial\_conn, serial\_port)

Here is where we use that constant defined up above. By telling Tornado to call SNAP Connect every so often, below we will be able to turn control of the CPU over to the Tornado HTTP library.

# Tell tornado to call SNAP connect internals periodically

tornado.ioloop.PeriodicCallback(self.snapconnect.poll\_internals, SNAPCONNECT\_POLL\_INTERVAL).start()

This completes the initialization of the InitialStateExample object, and the \_\_init\_\_() routine.

This next routine gets invoked whenever any node invokes “status” via RPC.

*Reminder – the “name mapping” from status() to \_on\_status() was done when we filled in that function dictionary.*

def \_on\_status(self, batt, button\_state, button\_count):

"""

Writes the various status values received from a node to Initial State

:return: None

"""

The calling node explicitly provided values for batt (battery voltage), button state (pressed or not), and “count of how many times the button has been pressed”.

Although not provided explicitly by the RPC caller, the SNAP Address of the calling node is provided implicitly, because it is a part of every SNAP Packet. This address is retrieved via function rpc\_source\_addr(), which the following line of code then converts from binary to HEX-ASCII. For example, “\x12\x34\x45” becomes “123456”.

remote\_addr = binascii.hexlify(self.snapconnect.rpc\_source\_addr())

The code prints the received status report just for debug purposes. This makes the values appear on the local console, but does nothing to get them to the Initial State website.

print batt, button\_state, button\_count

The variables we just printed get formatted into a JSON request by the following lines of code.

try:

headers = {

"X-IS-AccessKey": ACCESS\_KEY,

"X-IS-BucketKey": INITIAL\_STATE\_BUCKETS[remote\_addr],

"Content-Type": "application/json"

}

except KeyError:

print "Could not find SNAP address %s in INITIAL\_STATE\_BUCKETS" % remote\_addr

return

jsonreq = [

{"key": "batt", "value": int(batt)},

{"key": "state", "value": int(button\_state)},

{"key": "count", "value": button\_count}

]

This gets the raw data into JSON, but we still have to convert it into a HTTP request. The following line of code accomplishes this.

# Create a Tornado HTTPRequest

request = httpclient.HTTPRequest(url=INITIAL\_STATE\_URL,

method='POST',

headers=headers,

body=json.dumps(jsonreq))

Now that we have an actual HTTP request (with JSON-encoded data inside of it), we use the Tornado HTTP libraries to send it in a non-blocking (asynchronous) fashion. This allows us to continue listening for additional status() reports.

http\_client = httpclient.AsyncHTTPClient()

http\_client.fetch(request, self.\_handle\_request)

You will notice that the last parameter of the fetch() function call above was self.\_handle\_request.

This is a callback function (defined next) which later gets called with the result of the above “fetch request”.

@staticmethod

def \_handle\_request(response):

"""

Prints the response of a HTTPRequest

:param response: HTTPRequest

:return:

"""

This code lets us know how the “push” (reporting) of the data to Initial State turned out. For this simple example, we are simply printing the outcome, but we are not taking any additional actions.

if response.error:

print "Error:", response.error

The main() routine of this example program is very simple.

def main():

example = InitialStateExample()

1. An InitialStateExample() object gets created.
   1. Reminder – the \_\_init\_\_() function of that object gets invoked automatically
   2. THAT routine sets up SNAP Connect, etc. (refer back to the code up above)

tornado.ioloop.IOLoop.instance().start()

1. Tornado is given control of the CPU. It will continue running (without returning) until the program is forcibly shut down.

The following Python idiom is used to allow a single Python source file to serve as both a stand-along application AND as a reusable library.

if \_\_name\_\_ == '\_\_main\_\_':

main()

In other word, \*IF\* this file was invoked *by itself* (You typed **python main.py** on the command like)…

then invoke main().

Otherwise, main() does **not** get invoked.

As an example of NOT being invoked standalone, imagine if some OTHER Python source file had the line “import main” in it. For example, this would allow reuse of the “InitialStateExample” class without having to copy/paste the source code into that other file(s).

For more information about SNAP Connect Python programming, refer to the **SNAP Connect Python Package Manual**.

If you are unfamiliar with Python at all, one good reference is **Core Python Programming (2nd Edition)** by Wesley J. Chun (ISBN-10: 0-13-226993-7). A Google search for “Dive into Python” would also prove helpful.