Software Guide for E20 Example 1 – Gateway-Hosted Web Server

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

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

The Synapse Portal IDE will allow complete embedded module development, as well as wireless sniffer capability – download the latest version here: https://forums.synapse-wireless.com/showthread.php?t=9

The web application is a basic Python program built with high-performance libraries, Tornado and SNAP Connect. The Javascript/HTML is kept deliberately simple for ease of understanding, although it showcases a low-latency websockets technique. This can be easily extended to REST interfaces, and other web/backend approaches to fit application requirements.

See the readme.txt in the web_app directory for details and library dependencies.

Source Code Walk-throughs

In the following sections, we walk you through all of the source files that make up this example.

There are two broad organizing categories that apply:

1) "Location" - Where the source files apply / where they are used
2) "File Type" - Source file language

The possible "locations" and "types" are:

1) Inside a SNAP Node – this will usually be SNAPpy source files
2) Inside the E20 Gateway – this will usually be Python source files
3) Inside the web browser – multiple file types apply here:
    a. HTML files
    b. CSS files
    c. Javascript files

In the following sections, we will present the SNAPpy scripts first, then the Python source code, then the various web browser source files.

Source Code Walk-through (SNAPpy script demo_sn171.py)

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

Disclaimer – this script was created by modifying a copy of demo_sn173.py, and might have been written slightly differently had it had been created from scratch.

First up: a copyright notice, and a doc-string saying "what the file is".

Doc-strings are extra-handy in SNAPpy scripts, because Portal uses them to auto-generate tooltips.

```
# Copyright (C) 2014 Synapse Wireless, Inc.

"""Demo script for SN173 protoboard - status/control to E20 web interface"""
```

Here several other SNAPpy library files are imported. They will also have "walk-throughs", later in this same document.

```
from nv_settings import *

from batmon import *

from SN173 import *
```

Here a constant is defined, which will control how often a status() report will be sent, *even if nothing has changed*.

```
GRATUITOUS_STATUS_PERIOD = 5  # seconds
```

Here a variable is defined and initialized. It will be used as part of generating the "timed status reports".

```
second_count = 0
```

Here a counter variable is created for the purpose of simply tracking how many times the user has pushed the button.

```
button_count = 0
```

Here a constant is defined saying *which* button is being counted. Note that the definition of S1 (Switch 1) came from the SN173.py file imported up above.

```
BUTTON = S1
```

Here is the startup routine for this script. What *makes* this the startup code is not its name, but the fact that it is preceeded by the @setHook() call.

```
@setHook(HOOK_STARTUP)

def init():

    """Startup initialization"""
```

Here a subroutine (defined later in the script) is invoked. We'll discuss this routine later in this document.

For here, just know that "NV" stands for Non-Volatile and refers to the configuration parameters kept in a dedicated region of the SNAP Node's FLASH memory.

```
    # Set basic mesh parameters

    init_nv_settings(1, 1, True, True, False)
```

Here the digital output pins are initialized. Observant readers will notice that more LEDs are being initialized than actually exist. This is because this script reused code from a different Protoboard (the SN173, which *does* have more LEDs).

```
    # Init LEDs

    setPinDir(LED1, True)

    setPinDir(LED2, True)

    setPinDir(LED3, True)

    setPinDir(LED4, True)
```

Setting the "pin direction" to True makes each of them be **outputs**. Next the code goes ahead and does an initial "blip" on each LED.

NOTE – the standard form of the pulsePin() routine is **non-blocking**. These calls *initiate* the "blips" on each LED, but they do not wait for the blips (blinks) to complete. Each will occur in parallel, while other code runs.

```
    pulsePin(LED1, 500, True)

    pulsePin(LED2, 300, True)

    pulsePin(LED3, 200, True)

    pulsePin(LED4, 100, True)
```

Here the input switches are configured. The same comment about configuring more hardware than is actually present applies.

```
    # Init switches

    for s in SWITCH_TUPLE:

        setPinDir(s, False)

        setPinPullup(s, True)

        monitorPin(s, True)
```

It's worth mentioning that "for" statements were not supported in SNAPpy until version 2.6. If you are trying to use this example with an older version of firmware, you will need to substitute a manual "while loop" instead.

Here you can see setPinDir(…, False) being used to program input (versus output) pins. You will also notice that the internal pullup resistors within the chip are being enabled, preventing them from "floating" and giving spurious readings. The SNAPpy Virtual Machine is also told to monitor each button, which will later result in HOOK_GPIN events being generated. (More on HOOK_GPIN later in this walk-through).

Also note that SWITCH_TUPLE was defined in imported file SN173 – you won't find it defined in *this* file.

This next subroutine gets called automatically once every second, because of the @setHook() decorator placed immediately before it.

NOTE – SNAPpy also supports "timer hooks" for 100, 10, and 1 millisecond.

```
@setHook(HOOK_1S)

def tick1sec():

    """Tick event handler"""

    global second_count
```

The previous line is **very important!** If you don't tell Python you really mean the *global* variable, it defaults to creating a local variable of the same name *when you change its value*. This is often a point of confusion to new Python (and SNAPpy, which is a modified subset of Python) programmers.

We mentioned up above that this script would send reports every 5 seconds, even if nothing has changed. The following code is what does that.

```
    second_count += 1

    if second_count == GRATUITOUS_STATUS_PERIOD:

        second_count = 0

        send_status()
```

NOTE – subroutine send_status() is defined further below.

In addition to "time events", another asynchronous event that can occur in this SNAP Node is "button pushes" from the user. Because of our use of monitorPin() up above, the SNAPpy Virtual Machine will automatically generate HOOK_GPIN events when those buttons are pressed. The following routine is invoked when those HOOK_GPIN events are created, due to the use of a @setHook() generator right before the subroutine.

```
@setHook(HOOK_GPIN)

def pin_event(pin, is_set):

    """Button press event handler"""

    global button_count
```

Here the code is just making sure it's a button press. On a node with more than one button, different buttons could trigger different actions.

```
    if pin == BUTTON:
```

Because of how this board is wired up, pressing the button connects the digital input pin to GND, resulting in a reading of False. When the button is released, it is disconnected from GND and the internal pull-up resistors (enabled earlier) take it back True (HIGH).

In this particular script, we have *chosen* to increase the "button press count" on the push (versus the release).

Scripts could actually choose to do different actions on each (on both the True and the False value). It's also common to take the *duration* of the press into account (see for example MCastCounter.py) but that is not done here.

```
        if not is_set:
            button_count += 1
        send_status()
```

There's that send_status() routine again, and we still don't know anything about it.

Luckily, it is next in the source code.

```
def send_status():
```

Again, a reminder that doc-strings (like the following) automatically appear in the Portal User Interface.

Comment your code!

```
    """Broadcast a status RPC"""
```

First the code initiates a brief "blip" on one of the LEDs so you can visually tell it is generating packets.

By the way, the first parameter to pulsePin() is *which* pin to pulse, the second parameter is the *duration* of the pulse (in milliseconds), and the third parameter is the *polarity* of the pulse (True means leading edge is high, falling edge is low. False gives the opposite behavior). This is handy when you have LEDs that have been wired up so that "False == LIT", for example.

```
    pulsePin(LED4, 50, True)
```

Here is what generates the actual Remote Procedure Call (RPC) packet.

```
    mcastRpc(1, 3, 'status', batmon_mv(), not readPin(BUTTON), button_count)
```

The **1** here means **group 1**, which is the "broadcast" group in SNAP. The **3** specifies how many "hops" the status report should travel.

'status' is the name of the function being invoked, and the last three parameters to mcastRpc() become the parameters to status() *at the receiving nodes*. In other words, they will see this as

```
Status( batmon_mv(), not readPin(BUTTON), button_count )
```

Function batmon_mv() returns the power-supply level in millivolts (for example, 3300 corresponds to 3.3 volts). This routine is defined later.

Function readPin() does what it says (it READs the PIN). I mentioned earlier that on this hardware, “pressed” == False, so the extra **not** modifier is used to make this second parameter be “True means pressed”.

The third parameter is just the global variable button_count, which we already know is being updated in the HOOK_GPIN handler.

This next routine is not used locally. It gets invoked from the Web User Interface provided by this demo.

```
def lights(pattern):

    """RPC call-in to set our LEDs to given pattern.

       For SN173 we'll set 2 LEDs, but we could get a little fancier in the
future.

    """

    led_state = pattern & 1

    writePin(LED1, led_state)

    writePin(LED2, led_state)
```

Source Code Walk-through (SNAPpy script nv_settings.py)

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

This script won't do anything by itself – it is a *helper script*, intended to be imported and called by other SNAPpy scripts.

```
# Copyright (C) 2014 Synapse Wireless, Inc.

"""NV settings initialization"""
```

By now, Copyright notices and doc-string module comments should be common-place to you.

```
from synapse.nvparams import *
```

The above imported file is not unique to this demo. It is one of the *standard files* that ships with Portal.

Noteworthy is the use of "dirname<dot>filename" notation – source file nvparams.py actually lives in the synapse subdirectory underneath the snappyImages directory.

Anyway, this file defines constants for all of the NV_xxx parameters SNAP supports, making your code more readable by allowing you to say (for example) "NV_FEATURE_BITS" instead of "11".

This next routine is called by demo_sn171.py. It takes five parameters, which are the **desired** settings of five of the system NV Parameters.

What this routine does is make sure the **actual** values match the **desired** values, and correct any that are wrong.

(more commentary *after* the subroutine)

```
def init_nv_settings(mcast_proc, mcast_fwd, cs, ca, cd):

    """Set mcast processed groups, forwarding groups, CSMA settings, etc."""

    global _needs_reboot

    _needs_reboot = False



    # RPC CRC

    check_nv(NV_FEATURE_BITS_ID, 0x011F)
```

Noteworthy here – the line above enforces RPC_CRC == ON, regardless of the five passed in settings.

Setting RPC_CRC ON makes SNAP nodes more immune to packet corruption, by using an *additional* (software) CRC, in addition to the (hardware) CRC provided by the radio.

See also the PACKET_CRC feature added back in SNAP version 2.5, which provides even more robustness but is trickier to use (RPC_CRC tolerates some "mixed networks" – PACKET_CRC is 100% strict, and if you use it at all, you must use it everywhere.)

```
    check_nv(NV_GROUP_INTEREST_MASK_ID, mcast_proc)

    check_nv(NV_GROUP_FORWARDING_MASK_ID, mcast_fwd)

    check_nv(NV_CARRIER_SENSE_ID, cs)

    check_nv(NV_COLLISION_AVOIDANCE_ID, ca)

    check_nv(NV_COLLISION_DETECT_ID, cd)


    if _needs_reboot:

        reboot()
```

One thing you should notice when reading the above subroutine is the use of *another* subroutine (check_nv(), defined next) to make *this routine* shorter / have less duplicated code.

You will also see that the code is keeping track of any changes (variable needs_reboot), and if something has been changed, the reboot() function is called at the end. This makes the entire program "start over".

This is because most SNAP NV Parameter changes *only take effect at system startup*. Changes to those parameters are ignored by running code.

> Why doesn't the core firmware monitor those parameters for changes all of the time?
>
> Because we wanted to free up code space for more important functions.

```
def check_nv(param, val):

    global _needs_reboot

    if loadNvParam(param) != val:

        saveNvParam(param, val)

        _needs_reboot = True
```

The above subroutine simply checks to see if the specified *param* matches its desired *val*, and changes it if not.

NOTE – functions loadNvParam() and saveNvParam() are built-in SNAPpy functions that are always available to your scripts.

I will point out again the use of the explicit "global" specifier to let SNAPpy/Python know that it's the **global** _needs_reboot variable that check_nv() wants to change, **not** a dynamically created local variable with the same name.

SIDE NOTE – if you dislike global variables, the above code could be re-written such that check_nv() **returned** a "reboot is needed" value, which init_nv_settings() could keep track of itself. The trade-off would be longer, trickier code, but you will sometimes see this alternate approach used in other example scripts.

Source Code Walk-through (SNAPpy script batmon.py)

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

This script won't do anything by itself – it is a *helper script*, intended to be imported and called by other SNAPpy scripts.

```
# Copyright (C) 2014 Synapse Wireless, Inc.

"""ATmega128RFA1 internal battery monitor support

The Battery Monitor can be configured using the BATMON register. Register
subfield

BATMON_VTH sets the threshold voltage. It is configurable with a resolution
of 75 mV

in the upper voltage range (BATMON_HR = 1) and with a resolution of 50 mV in
the

lower voltage range (BATMON_HR = 0).

"""
```

Just want to note here that the doc-string up above is formatted that way to make the tooltip in Portal more readable while taking up less space.

Below some constants are defined. Their **values** and **names** were taken from the ATMEL datasheets for the ATmega128RFA1, this chip used inside the xx2xx series of SNAP Modules.

```
BATMON_REG = 0x151

BATMON_HR = 0x10    # select high/low range

BATMON_OK = 0x20    # set if batt voltage is above Vth

BATMON_SNAP_DEFAULT = 0x06
```

How to interpret such datasheets, and how to write SNAPpy scripts to take advantage of that info is beyond the scope of this document.

The actual chip can return 16 possible values (0-15), for two different range settings (low and high). The following constant tuple definitions allow us to convert these raw readings (0-15) to actual millivolt equivalents.

As one quick example, if the register value was 0, and the low range was in use, the chip would actually be reporting a voltage reading of 1700 millivolts (or 1.7 volts).

```
low_range = (1700, 1750, 1800, 1850, 1900, 1950, 2000, 2050, 2100, 2150,
2200, 2250, 2300, 2350, 2400, 2450)

high_range = (2550, 2625, 2700, 2775, 2850, 2925, 3000, 3075, 3150, 3225,
3300, 3375, 3450, 3525, 3600, 3675)
```

Again, this sort of information came straight out of the manufacturer's datasheets.

```
def batmon_mv():
```

The chip does not actually provide the current voltage. What it *can* do is tell you if the voltage is “OK” (or not), based on a threshold setting you place in another register.

By starting at the highest thresholds, and working downward, the chip can report a voltage as soon as it finds a threshold that “is OK”.

First the high range is checked.

```
    i = 16
```

There are 16 possible threshold settings within each range.

```
    while i > 0:
```

We want to try all of them.

```
        i = i - 1
```

1) This counts down the loop. 2) We need to write a value of 0-15 to the chip, not 1-16.

This next line sets the actual threshold into the chip.

```
        poke(BATMON_REG, BATMON_HR | i)
```

The code then reads back the “my battery is OK” status bit from the chip. If the battery reported “OK”, then we have found our answer. We just need to restore the “real” threshold and report the (converted) answer back.

```
        if peek(BATMON_REG) & BATMON_OK:
```

This next line if very important. SNAPmakes use of this same voltage monitor to tell if it is OK to be re-programming the node’s FLASH memory (for example, saveNvParam() or SNAPpy script uploads. We have to be sure it’s back to its normal setting.

```
            poke(BATMON_REG, BATMON_SNAP_DEFAULT)

            return high_range[i]
```

Earlier I mentioned that those tuples would be used for **table lookup**. This is why the code returns high_range[i], not just i.

If the above loop did not find a match (none of the HIGH RANGE settings were “OK”), then the loop below runs, checking all 16 possible LOW range thresholds.

```
    i = 16

    while i > 0:

        i = i - 1

        poke(BATMON_REG, i)

        if peek(BATMON_REG) & BATMON_OK:
            poke(BATMON_REG, BATMON_SNAP_DEFAULT)

            return low_range[i]

    return 0
```

NOTE – the hardware cannot tell if it is running from a battery or an external DC power supply. So, the name of the routine is a little inaccurate.

SIDE NOTE – the lower the voltage actually is, the longer it will take this routine to find a match and report back a value.

Source Code Walk-through (SNAPpy script SN173.py)

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

This script won't do anything by itself – it is a *helper script*, intended to be imported and called by other SNAPpy scripts.

As mentioned previously, the use of a SN173 helper script in a SN171 example script is just a side effect of the SN173 demo being created first, and then the SN171 demo created via a quick "clone and modify".

```
# Copyright (C) 2014 Synapse Wireless, Inc.

"""SN173 definitions"""
```

The following constants define the "SNAPpy IO to Switch" mappings for a SN173 demo board. The SN171 can reuse the "S1" definition because when we designed the SN173, we made its *first button* match the SN171's *only button* on purpose, knowing that scripts would often be shared between them.

```
S1 = 20

S2 = 0

S3 = 1

S4 = 9
```

Defining a tuple containing all of the switch definitions makes it easy to iterate over them, especially now that SNAPpy supports for loops (refer to the SNAP 2.6 Reference Manual).

```
SWITCH_TUPLE = (S1, S2, S3, S4)
```

The SN173 also has 4 LEDs (versus the two on a SN171). However, we followed the same practice of being as backwards compatible as we could. The *first two* LEDs on a SN173 match the *only two* LEDs on a SN171.

```
LED1 = 6

LED2 = 5

LED3 = 19

LED4 = 8

LED_TUPLE = (LED1, LED2, LED3, LED4)
```

The same comment about the advantage of tuples made up above applies here.

SIDE NOTE – you will see tuples used instead of byte-lists in many of our example scripts because *we have had tuples longer* – SNAPpy did not support **any sort** of lists until version 2.6.

In many cases, tuples and byte-lists can be used interchangeably.

Tuples have the advantage of being able to hold more than “just bytes”. SNAPpy byte-lists have the advantage of being **more compact**. They are just as small as character strings (and in fact, share the same RAM inside of the SNAP Node).

NOTE – this marks the end of the SNAPpy script walk-throughs. In the next section we will be discussing source code that runs on the E20 Gateway.

Source Code Walk-through (Python file app_server.py)

Reminder – the following code runs on the E20 Gateway (not on the SNAP Nodes, not in the web browsers).

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

There's not much to say about the Copyright notice and top-level doc-string.

```
# (c) Copyright 2015, Synapse Wireless, Inc.

"""Application Server for SNAP demo

   A web server based on Tornado, integrated with SNAP Connect.

"""
```

Several features are imported from the Tornado libraries.

```
import tornado.escape

import tornado.ioloop

import tornado.web

import tornado.websocket
```

We will also be leveraging the SNAP Connect Python Package…

```
import snapconnect

from snapconnect import snap

from apy import ioloop_scheduler
```

… and several standard Python libraries.

```
import asyncore

import os

import logging

import binascii

import sys

import time
```

We set up a log to write messages to. In Python, "__file__" evaluates to the actual base name of the source file, in this case it is "app_server" because this file is named "app_server.py".

```
log = logging.getLogger(__file__)
```

The correct serial port, SNAP address, and license file depends on whether or not we are running on an E20 Gateway or a PC.

```
# SNAP Connect settings

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
```

When SNAP Connect is running on an E20 (see above), the SNAP Address is determined by the last three bytes of the Ethernet MAC Address, and you do not need a license file. The Gateway hardware is your license).

```
    # Allow time for wifi AP to initialize (TODO: remove this and handle by
server exception)

    time.sleep(10.0)

else:

    # SS200 USB stick on Windows

    serial_conn = snap.SERIAL_TYPE_SNAPSTICK200

    serial_port = 0
```

When SNAP Connect is running on a PC, you **do** have to have a license file on your PC, and that file determines your *possible* SNAP Address. This *next* line of code is choosing one of the available addresses (a single SNAP Connect license file can enable more than one), not just arbitrarily making one up on the fly.

```
    snap_addr = '\xff\xb6\x06'
```

These next two lines are just specifying the name and location of that license file.

```
    cur_dir = os.path.dirname(__file__)

    snap_license = os.path.join(cur_dir, 'SrvLicense.dat')
```

All of the above was just "preparation" (imports and variable definitions). We haven't actually provided any significant amount of code yet.

Now we start getting into some of the "meatier" sections. In this next stretch of code, we extend the basic tornado.websocket.WebSocketHandler class by sub-classing it into our own custom WebSocketHandler.

Full explanation of Object Oriented Programming (OOP) is outside the scope of this walk-through, but a key takeaway here is that our derived class only has to provide the *differences* to the existing base class's functionality. It's also worth point out that WebSocketHandler communicates in two different directions:

1) Browser to server
2) Server to browser

First we say there is to be a class named WebSocketHandler…

```
class WebSocketHandler(tornado.websocket.WebSocketHandler):
```

… we provide a doc-string comment saying what it does...

```
    ''' Send and receive websocket messages between server and browser(s).

        Messages TO the browser are encoded "rpc-like" as

            {'kind' : 'funcName', 'args' : params}

        Messages FROM the browser are translated to RPC calls and invoked on

        our SNAP Connect instance.

    '''
```

… then we define all of the functions and data that make up that class.

```
    waiters = set()    # Browser connections
```

In the previous line of code, that's "waiters" as in "web browsers who are waiting for data". More importantly, notice we did **not** say "this.waiters = set()". This set is going to be shared among **all** instances of this class.

Side Note – the advantage of a **set** over a **list** is that using a set gives us "ignore any duplicates" capability for free.

(If you add something to a set more than once, it has no effect).

This next function is required in all derivatives of tornado.websocket.WebSocketHandler, and controls *backwards compatibility*. Here we chose to allow the older web browsers - returning False instead of True would have made this web server "more strict".

```
    def allow_draft76(self):

        '''Allow older browser websocket versions'''

        return True
```

By default, the Tornado implementation of WebSocketHandler takes no special action when web browsers connect (open) of disconnect(on_close()). What we are doing here is maintain a set of "known browser connections" (the "waiters" created up above) so that actions taken / data discovered in one browser session will be correctly reflected on all of them.

```
    def open(self):
        WebSocketHandler.waiters.add(self)
    def on_close(self):
        WebSocketHandler.waiters.remove(self)
```

The use of "self" here might be confusing, until you remember that the "waiters" variable is *shared* between all instances of this WebSocketHandler class. By adding/removing yourself, you are keeping everybody informed.

This next method is where we see the "set of waiters" actually made use of.

```
    @classmethod
    def send_updates(cls, message):
        #log.info("sending message to %d waiters", len(cls.waiters))
        for waiter in cls.waiters:
            try:
                waiter.write_message(message)
            except:
                log.error("Error sending message", exc_info=True)
```

The above code just goes through the entire set, and tries to write the specified message to each one in turn.

The "try/except" statement serves two purposes here:

1) It ensures that an issue with one browser session does not prevent other browser sessions from being updated
2) By use of the log.error() statement, it gives us a way to learn about any issues that occurred (assuming we actually bother to look in the log file)

NOTE – the above code was sending data *from* the web server (this Python program) *to* one or more web browsers. This next routine is how the web browsers can send incoming messages, which then get translated into SNAP Connect function calls.

```
    def on_message(self, message):
        '''Translate browser message into RPC function call (into local
SnapCom object)'''
        #log.info("got message %r", message)
```

This next line converts the message from a "string containing JSON-style text" into a "Python Dictionary". It is an example of how a powerful library of support routines can make developing a complex application easier.

```
        parsed = tornado.escape.json_decode(message)
```

Below you see another use of the try/except statement. The deal here is that the web browser may be asking for a Python function that does not exist.

NOTE – we will see the available set of Python functions defined later on in this walk-through. The important point here is that SNAP Connect only provides access to functions that you *explicitly* tell it to, it's not enough just to have the routine exist within your program.

```
        try:
            func = getattr(snapCom, parsed['funcname'])
        except AttributeError:
            log.exception('Browser called unknown function: %s' %
str(parsed))
        else:
            try:
```

Even if the web browser called a valid function, it may not be doing so *correctly*. Here the code tries to convert the arguments into a Python list, and then pass them to the actual function.

```
                args = [str(a) if isinstance(a,unicode) else a for a in
parsed['args']]
                func(*args)
```

That "*" in "*args" is worth mentioning. Python as the neat feature of being able to handle variable numbers of arguments easily. For example, if Python list args contains [1, 'hello', True] (3 arguments), then invoking foo(*args) makes function foo () think you called it as foo(1, 'hello', True). That's 3 parameters (like we wanted). If we instead called foo(args), then foo would receive only <u>one</u> parameter, the list of 3 items.

```
            except:
                log.exception('Error calling function: %s' % str(parsed))
```

That wraps up the WebSocketHandler class, and provides us one of several building blocks that we will need. Notice that at this point, this program hasn't actually created a web server yet. It's just getting prepared to do so.

Next up, the code defines (but does not actually create) another building block. The "SnapCom" object created below is what will actually be aware (actually make use of) the SNAP Connect library.

Trivia – "SnapCom" was the internal project name of what became the SNAP Connect 3.x series of software. So, you will often run into code that refers to a "SNAP Connect instance" as a "snapcom". Here the situation is slightly different, as this "SnapCom" object <u>has</u> a SNAP Connect instance, versus <u>is</u> a SNAP Connect instance.

(Just keep reading if this is not clear yet).

```
class SnapCom(object):
    """Snap Connect communication layer"""
```

All Python classes derive from some other existing class, even if it is just the built-in class “object”.

This next line of code is defining how often we want to let the “SNAP Connect code” (versus the “web socket code”) run. You will see this constant actually used further down.

```
    SNAPCONNECT_POLL_INTERVAL = 5 # ms
```

All Python classes need an __init__() function. In some other programming languages, this goes by the name “constructor”.

```
    def __init__(self):
        self.snapRpcFuncs = {'status' : self.status,
                             'send_ws' : self.send_ws
                            }
```

I mentioned up above that there would be an explicit set of “callable” functions defined. The previous line of code is what did that. A standard Python dictionary was filled in with pairs of “what I want to call the function by” (“name”) and “what function I actually want to invoke” (“function”) pairs. This “renaming” feature is handy here, because the calling code should not care if we used the OOP paradigm or not. I do want to point out that the names are often the same.

```
        # Create SNAP Connect instance. Note: we are using TornadoWeb's
scheduler.
        self.snapconnect = snap.Snap(license_file = snap_license,
                                     addr = snap_addr,

scheduler=ioloop_scheduler.IOLoopScheduler.instance(),
                                     funcs = self.snapRpcFuncs
                                    )
```

There are many parameters to the SNAP Connect constructor, but most of them take assume default values if you do not specify otherwise. The things that are specified here are:

1) License_file - Up towards the top of this source file, we specified which license file we wanted to use (or set this variable to None if we were running on an E20 Gateway)
2) Addr - We also chose our SNAP Address if we were running on a PC
3) Funcs - The “dictionary of functions others are allowed to call” was documented on the top of this page
4) Scheduler – looks scary, but see next paragraph

The “scheduler” parameter is just SNAP Connect’s way of cooperating more fully with other software libraries. *When* you provide an alternate scheduler (this is not required), SNAP Connect assumes you are going to set up some other code to "be in charge" of making sure all of the polling takes place. You will see that "other code" in a few more pages.

```
        # Configure SNAP Connect params

        self.snapconnect.save_nv_param(snap.NV_FEATURE_BITS_ID, 0x100)     #
Send with RPC CRC
```

There are many "knobs" that you can twist in SNAP Connect, either through save_nv_param() or other functions. The code above is keeping all but one at their default values. The RPC_CRC feature bit of NV #11 is being set so that SNAP Connect will both *provide* and *accept* Remote Procedure Call (RPC) packets with this extra CRC appended.

NOTE – As of version 2.5, SNAP also supports a PACKET_CRC that applies to **all** packets, not just **RPC** packets. The reason we don't usually enable this in our examples is two-fold:

1) PACKET_CRC is very strict – if you use it at all, you have to use it everywhere. Works great in a real deployment, can be a nuisance to get everything switched over if you are just trying to take a demo for a "test drive"
2) PACKET_CRC is much newer (2014), and users may not have firmware capable of doing it deployed throughout their networks yet. RPC_CRC is several years older, and by now is almost universally deployed.

Note that it is OK to enable RPC_CRC even if not all of your nodes support it, because it functions in a backwards compatible way (at the cost of not providing as much additional robustness).

I mentioned up above that there were other functions besides save_nv_param() that controlled SNAP Connect behavior. Here are two examples of that.

```
        # Connect to local SNAP wireless network

        self.snapconnect.open_serial(serial_conn, serial_port)

        #self.snapconnect.accept_tcp()
```

By default, SNAP Connect **does nothing**. If you want it to be talking over a serial port, and/or listening for incoming TCP/IP connections, you have to explicitly call open_serial() and accept_tcp().

NOTE – there is also a connect_tcp() function available for making *outbound* connections. For example, maybe you are running <u>another</u> SNAP Connect instance up in "the cloud".

These next two lines let us "hook" additional code to the occurrences of TCP/IP connections opening and closing. In retrospect, these probably should have been named "HOOK_TCPIP_OPENED" and "HOOK_TCPIP_CLOSED", but as I already mentioned, "SNAPCOM" was the original name for the codebase.

```
        self.snapconnect.set_hook(snap.hooks.HOOK_SNAPCOM_OPENED,
self.on_connected)

        self.snapconnect.set_hook(snap.hooks.HOOK_SNAPCOM_CLOSED,
self.on_disconnected)
```

More importantly, the above two lines of code mean that the on_connected() and on_disconnected() functions will get invoked at the appropriate times. You will see those routines in 2-3 more pages.

Here you can see where that alternate "scheduler" gets set up. We are telling the Tornado library to run the show, and how often to let SNAP Connect have a turn.

```
        # Tell the Tornado scheduler to call SNAP Connect's internal poll
function.
        tornado.ioloop.PeriodicCallback(asyncore.poll,
self.SNAPCONNECT_POLL_INTERVAL).start()
        tornado.ioloop.PeriodicCallback(self.snapconnect.poll_internals,
self.SNAPCONNECT_POLL_INTERVAL).start()
```

This completes the *initialization* of the SnapCom object. We still have to give it some member functions so that it can do useful stuff.

This next routine allows the other Python code to send messages to any connected web browsers.

```
    def send_ws(self, func, *args):
        '''SNAPpy call-in to invoke websocket functions'''
        message = {'funcname' : func, 'args' : args}
        WebSocketHandler.send_updates(message)
```

The only subtle point in the send_ws() (that's short for "send over websocket") routine up above is the use of a "class" method in the WebSocketHandler class (defined previously).

Because WebSocketHandler.send_updates() only used class *variables*, and was itself declared to be "class-wide", it can be invoked by outside code (like the routine above) without having an actual reference to a WebSocketHandler.

This next function is invoked when connected SNAP Nodes broadcast (multicast) status() reports. WHY this function gets invoked is because we specified so, when we defined that "function dictionary" up above. WHY the nodes are sending these reports is a function of their loaded SNAPpy scripts, refer the very first code walk-through in this document.

```
    def status(self, batt, pressed, count):
        """Status report call-in from SNAPpy remote nodes"""
        src_addr = self.snapconnect.rpc_source_addr()
        src_hex_addr = binascii.hexlify(src_addr)
        self.send_ws('report_status', src_hex_addr, batt, pressed, count)
```

Notice that status() makes use of the send_ws() routine defined right before it.

This next routine is something the web browsers can call to turn the light (LED) on/off on the specified SNAP Node. Be aware that other Python code *internal* to this server could also call this routine, but *external* code cannot – we did not put this routine into the dictionary of "allowed functions".

```
    def lights(self, hex_addr, pattern):
        """Browser call-in to control lights of addressed node"""
        addr = binascii.unhexlify(hex_addr)
        self.snapconnect.rpc(addr, 'lights', pattern)
```

Since the above routine was pretty short, I'll take this opportunity to mention that the binascii.unhexlify() function is being used to convert binary SNAP Addresses like "\xff\xee\xdd" (three bytes, and not readable when printed) to ASCII SNAP Addresses like "FFEEDD" (six characters, readable when printed).

This next routine appears to be leftover code, possibly predating the creation of the full WebSocketHandler class. (I could not find any other reference to it in this source file).

```
    def snap_method(self, func, *args):
        '''Browser call-in to directly invoke snapconnect methods'''
        func = getattr(self.snapconnect, func, None)
        if callable(func):
            func(*args)
```

Here is a simple "convenience" routine that lets you call do_log("message") instead of log.info("message). The issue here was providing a single place to change the "error level" of the logged messages (currently "info" but others, such as "error" and "debug" are also possible.

```
    def do_log(self, *args):
        log.info(*args)
```

Up above function set_hook() was used to specify some extra handlers for TCP/IP connect and disconnect events. Here are those actual handlers. They are basically just instrumenting the code.

```
    def on_connected(self, addr_pair, remote_snap_addr):
        self.connected = True
        log.debug("on_connected(%s)" % str(addr_pair))


    def on_disconnected(self, addr_pair, remote_snap_addr):
        """Called by SNAP Connect when a SNAP TCP connection has been
disconnected or failed to connect"""
        self.connected = False
        log.debug("on_disconnected(%s)" % str(addr_pair))
```

Next up, one more building block (class), and it <u>uses</u> the first class we defined (WebSocketHandler). Once again, this is an example of <u>extending</u> an existing Python class by subclassing it (deriving a new class from it).

```
class Application(tornado.web.Application):
    def __init__(self):
```

“Handlers” in the context of the following line means “web page request handlers”. These are checked in order, so any request for “/” will always serve up index.html, any request for “/wshub” will be processed by WebSocketHandler, and everything else will be searched for in the “www” directory.

```
        handlers = [
            (r"/", tornado.web.RedirectHandler, {"url": "/index.html"}),
            (r"/wshub", WebSocketHandler),
            (r"/(.*)",        tornado.web.StaticFileHandler,        {"path":
os.path.join(os.path.dirname(__file__), "www")}),
        ]
```

Similar to how we tweaked some SNAP Connect settings after we instantiated it, below we are fine-tuning the behavior of the Web Server provided by the Tornado library.

```
        settings = dict(
            cookie_secret="43oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o/Vo=",
            static_path=os.path.join(os.path.dirname(__file__), "www"),
            xsrf_cookies=True,
            autoescape=None,
            debug=True,
        )
```

“handlers” and “settings” variables have been defined, but we haven’t yet told tornado.web.Application to actually <u>use</u> them. See next line:

```
        tornado.web.Application.__init__(self, handlers, **settings)
```

We *still* don't actually have a running web server, just the building blocks from which to make on. This is where main() comes in.

```
def main():

    global snapCom
```

Key point – the above line does not actually create a "snapcom", it just says that *when we do* (coming up in a few lines), *we want it to be globally available to the entire program*.

Logging setup and usage is just basic Python stuff, refer to the Python docs.

```
    logging.basicConfig(level=logging.INFO,           format='%(asctime)-15s
%(levelname)-8s %(name)-8s %(message)s')

    log.info("***** Begin Console Log *****")
```

Finally we get to actually create a web server. The following line of code actually instantiates an instance of the Application class (which arguably could use a less generic name).

```
    app = Application()
```

That web server is told to start listening for web browsers to connect on the standard HTTP port, 80. You will often see "test" web servers set up on alternate ports like 8080 or 8888, the following like is where you would make that sort of change.

```
    app.listen(80)
```

Just like the web server object, the SnapCom object does not automatically get created just because we *defined* it. Here is where we rectify that.

```
    snapCom = SnapCom()
```

Remember, variable snapCom was defined up above to be **global**. The rest of this program can make use of it, refer back to the on_message() routine.

The "Tornado" library was previously given a "heads up" that he was expected to run the show. Here is where hs is given the actual "green light".

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

The above line of code is actually very important. Without it, the system **still** would not do anything useful.

Finally, as always the use of the following paradigm allows standalone programs like this one to *also* be reused as libraries themselves (for example, if you wanted to reuse the WebSocketHandler or SnapCom classes).

```
if __name__ == '__main__':
    main()
```

These last walk-throughs are all about files that *reside* on the E20 (in the www directory used by app_server.py), but they get *"served up"* to the web browsers, and *interpreted/executed/displayed* there. Please note that they are in a mix of HTML, Cascading Style Sheet (CSS), and Javascript files.

Source Code Walk-through (Web Page index.html)

The following code walk-through intersperses commentary (in this font and color) with source code (`in this font and color`). Also note that some whitespace has been removed relative to the original source file to better fit the printed page.

```
<!DOCTYPE    html    PUBLIC    "-//W3C//DTD    XHTML    1.0    Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
```

The above HTML is standard boilerplate…

```
  <head>

    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

    <title>SNAP Demo</title>

    <link rel="stylesheet" type="text/css" href="mainstyle.css"
media="screen" />

    <script type="text/javascript" src="jquery-1.11.3.min.js"></script>

    <script type="text/javascript" src="syn_websocket.js"></script>

    <script type="text/javascript" src="main.js"></script>

  </head>
```

The above HTML defines the header of the page (for example, the <title>), plus pulls in *four other files*.

NOTE – many of these other files have their own walk-through sections, later in this document.

Here I will just note that the file extensions give an indication of what each file contributes to the web page:

- Cascading Style Sheets (.css files) affect the look of the page, but not its content
- Javascript (.js) files actually add code to the web page

For an example of *just how much impact* a CSS file can have on a website without changing the actual HTML and CSS, please see http://www.csszengarden.com/

Next the definition of the body (main portion) of the web page **begins**.

NOTE – the body definition is everything between the <body> and </body> tags.

```
  <body style="background:white">
```

```
    <div id="banner">
       <img src="banner.png" width=100%>
    </div>
```

Portions of a web page are split up into divisions. The one above pulls in the spiffy banner.jpg that goes across the top of the page (“width=100%”).

NOTE – the main purpose of giving “div”s identifiers (“id”s such as id=”banner”) here is so that they can be referenced in the CSS files.

```
    <div class="main_page">
       <table style="width:100%" id="node_table">
          <tr>
             <th>Address</th>
             <th>Batt (mv)<th>
             <th>State</th>
             <th>Count</th>
             <th>Lights</th>
          </tr>
       </table>
    </div>
```

The main portion of the page is taken up by a dynamic table (dynamic in that the number of rows is not fixed, but instead depends on how many live SNAP Nodes you have reporting in, plus the contents of each column represent “live” data). Each *table row* (<tr>, </tr>) is made up of multiple columns. Each *table column* (<th>, </th>, where “th” stands for “t”able “h”eading) has a **header** defined here. The actual **values** get set by some Javascript code that is covered separately.

NOTE – if you relabel these columns, you must update the Javascript code too (or else your live data will **not** fill in).

```
  </body>
</html>
```

These are just the corresponding *ending tags* to the <body> and <html> tags up above.

Source Code Walk-through (Cascading Style Sheet mainstyle.css)

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

```
/* Styling for SNAP demo */
```

(File comment…)

```
/*=== Reset Styles ===*/

html, body, div, span, applet, object, iframe,

h1, h2, h3, h4, h5, h6, p, blockquote, pre,

a, abbr, acronym, address, big, cite, code,

del, dfn, em, font, img, ins, kbd, q, s, samp,

small, strike, strong, sub, sup, tt, var,

dl, dt, dd, ol, ul, li,

fieldset, form, label, legend,

table, caption, tbody, tfoot, thead, tr, th, td {

      margin:0;

      padding:0;

      border:0;

      outline:0;

      font-weight:inherit;

      font-style:inherit;

      font-size:100%;

      font-family:inherit;

      vertical-align:baseline;

}
```

The above set of definitions ensures the page looks the same across all browsers, by explicating setting every parameter. Otherwise, *default* values would have been used, which could vary from one web browser to another.

```
body {

      line-height:1;

      color:black;
      background:white;
}
```

The above set of definitions applies to the *body* of the web page. Note that since there is not much on the page (just a banner and a single table), there is not much styling that needs to be applied.

```
blockquote:before, blockquote:after,
q:before, q:after {
      content:"";
}
blockquote, q {
      quotes:"" "";
}
/* HTML5 tags */
header, section, footer,
aside, nav, article, figure {
      display: block;
}
```

(More ensuring consistency between browsers…)

```
/*=== Define Custom Styles ===*/
```

Now we get to the interesting stuff… heights, colors, fonts, etc. All specified so that the page has exactly the appearance we want.

NOTE – the "#" prefix indicates the use of an "id" from within the HTML file. So here we are styling the "banner" (id="banner" in the HTML file), followed by the title of the page.

Side Note – the sections *do not* have to be styled in the order of their occurrence within the HTML file. For example, the title actually occurs above the banner on the actual web page.

```
#banner {
    width:100%;
    height:100%;
    margin: 0px;
    padding: 0px;
}
```

```
#title {
    font:24px Arial Black, Gadget, sans-serif;
    color:#fff;
    position:relative;
    left:10px;
    top:8px;
}
```

Most of the above should be pretty readable. Worth mentioning is that "px" is short for pixel (other measurement units are possible), and colors are specified in a hexadecimal "R"ed "G"reen "B"lue (RGB) notation. For example, #F00 is pure red, #0F0 is pure green, and #00F is pure blue. That #FFF reference up above is pure white.

```
.main_page {
    float: left;
    margin: 0px;
    padding: 5px;
    width: 98%;
    height: 100%;
    /* border: 1px solid black; */
}


#node_table
{
     font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
     width:100%;
     border-collapse:collapse;
}
```

(Styling for the overall table…)

```
#node_table td, #log_table th
{
     font-size:1.5em;
```

```
      border:1px solid #06c;

      padding:3px 7px 2px 7px;

}
```

(Styling for the data within the table cells…)

```
#node_table th

{

      font-size:2em;

      text-align:left;

      padding-top:5px;

      padding-bottom:4px;

      background-color:#69f;

      color:#ffffff;

}
```

(Styling for the headers at the top of each column…)

```
#node_table tr.alt td

{

      color:#000000;

      background-color:#EAF2D3;

}
```

A little more styling for the table rows. Note that #000000 is “NO red, NO green, and NO blue”, AKA “black”.

Finally, the checkboxes in the rightmost column are styled. Note that most browsers ignore commands they do not understand, so here the same function is specified twice, once for the non-Apple browsers, once for Apple’s webkit (Safari) browser.

```
input[type=checkbox] {

  /* All browsers except webkit*/

  transform: scale(1.5);

  /* Webkit browsers*/

  -webkit-transform: scale(1.5);

}
```

Source Code Walk-through (Javascript File main.js)

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

NOTE – the bulk of the Web Socket specific code has been split out into a separate file “syn_websocket.js” – the walk-through for *that* file is after this one. Also, the Javascript library “JQuery” is used by this example code. You can learn more about the JQuery library at https://jquery.org .

```
// (c) Copyright 2015, Synapse Wireless, Inc.

// Main javascript file for SNAP Demo
```

(Copyright and file-level comment…)

```
// Document Loaded callback

$(document).ready(function() {

    // Initialize websocket connection from browser to E20

    wsHub.start();

});
```

Support for the $(document).ready() function comes from the JQuery library (JQuery takes care of calling this function at the correct time). The important point here is that this is where the Web Socket Server (“wsHub”) gets started. **The actual “wsHub” code is in a separate source file.**

This next function is the end of a chain of function calls that started at the SNAP Node. The **SNAPpy** script in the node sends out a status() Remote Procedure Call. **Python** code running on the Gateway implements a status() routine, which calls *this* routine (all the way up in the web browser).

```
// Call-in from server SNAP application

// Status update from addressed node. Update table to reflect status.

function report_status(addr, batt, pressed, count) {

    console.log("status: " + addr + ", count=" + count);
```

(Just printing part of the report for debugging purposes. The users will only see this if they look at the web browser’s console log.)

```
    // If there's not already a table row for this address, append one.
```

The following search bears explanation… when the rows are *created*, the code assigns them an “id” equal to the corresponding SNAP Address. When searching within the web page, “id”s are specified using a “#” prefix. So, the following line is trying to grab a table row that may or may not exist *yet*.

```
    var row = $('#' + addr)
    if (row.length == 0) {
        // Make a new row
```

Each row has a check box on the far right that allows you to control an LED on the SNAP Node. It's the one field that did not come from the node (so we have to create here).

```
        var inp_cell = '<td><input type="checkbox"/></td>';
```

Now we can construct the actual row. Notice that the columns are being given *names* here, not *values*.

```
        $("#node_table").append('<tr id="' + addr + '">
            <td>addr</td> <td>batt</td> <td>pressed</td> <td>count</td>' +
            inp_cell + '</tr>');
```

Although constructed, we want to make one further adjustments. So, we have to "find" the row *we just made*.

```
        row = $('#' + addr);
```

The newly constructed row has the checkbox already in it, but the web browser does not know what to do when it gets checked or unchecked. This next function call corrects that.

```
        // Hook the "Lights" checkbox
        row.find('td:eq(4)').change(
          function(ev) {
            // Send command to addressed node when changed
            set_lights(addr, ev.target.checked ? 1 : 0);
        });
    }
```

Notice that the code first had to *find* the correct column (4 since the numbering starts with 0), then it calls the change() function on that, and the code defines the function to be installed in-line.

Next the code fills in the *values* for each of the columns.

```
    // Update table with new status
    row.find('td:eq(0)').text(addr);
    row.find('td:eq(1)').text(batt);
    row.find('td:eq(2)').text(pressed);
    row.find('td:eq(3)').text(count);
}
```

Notice that row.find(criteria.text(string) is shorthand for temp = row.find(criteria); temp.text(string)

This final function gets invoked automatically by the web browser when any checkbox gets toggled. The reason WHY is that change() command made back in the report_status() routine.

```
// Call 'lights()' function on E20 -> addressed SNAP node

function set_lights(addr, pattern) {

    send_message('lights', [addr, pattern]);

}
```

NOTE – the send_message routine is defined in syn_websockets,js, described next.

Source Code Walk-through (Javascript File syn_websocket.js)

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

```
// (c) Copyright 2015, Synapse Wireless, Inc.
```

The other Javascript code can call the following function to send messages over the Web Socket interface to the app_server.py code running on the E20 Gateway.

```
// Send a message over our WebSocket to SNAP Connect server

function send_message(funcname, args)  {
```

First a JSON dictionary representing the message (function call) gets created. It may look like the code is stuttering but what it is really saying is "set the "funcname" entry *in the dictionary* to the value of the funcname parameter that was *passed into this function*", and then "set the "args" entry *in the dictionary* to the value of the args parameter that *was passed in*.

For example, if someone calls send_message("foo", [1,2,3]) then message will become:

{funcname: "foo", args: [1,2,3]}

```
    var message = {funcname:funcname, args:args};
```

That JSON dictionary is then converted into a literal string and sent over the Web Socket connection.

```
    wsHub.socket.send(JSON.stringify(message));

}
```

This next function allows the web page to ask the Gateway to send a multicast RPC call on its behalf.

Refer to the snap_method() function in app_server.py.

```
function mcastRpc(group, ttl, func, args) {

    send_message('snap_method',

    ['mcastRpc', group, ttl, func].concat(args));
```

In case it is not obvious, in Javascript all arrays have the ability to call their member functions (such as concat()), *even if they have not been placed into a named variable*.

To give another example, [3, 2, 1].sort() becomes [1, 2, 3].

```
}
```

```
// WebSocket Hub: Establish a socket between browser and SNAP Connect Web
server. Provide API to send/receive messages
```

Javascript doesn't really have "true" classes, but you can get the same effect by using a Javascript dictionary that *contains* the "member" variables and functions you would want such a class to have. So, you could read "var wsHub" as "class wsHub".

```
var wsHub = {

    socket: null,
```

Javascript syntax can be a little klunky. Here what the code is saying is "start is a function that does …".

```
    start: function() {

        var host = "ws://" + location.host + "/wshub"
```

All Web Socket connections have the URL prefix "ws:". "location.host" is provided to us by the web browser. The correct URL ending is "/wshub" because that is what we named it back in app_server.py.

```
        // Detect WebSocket support
```

(or report failure and give up…)

```
        if ("WebSocket" in window) {

            // Modern browsers

            wsHub.socket = new WebSocket(host);

        } else if ("MozWebSocket" in window) {

            // Firefox 6

            wsHub.socket = new MozWebSocket(host);

        }

        else  {

            $('body').html("<h1>Error</h1><p>Your browser does not support
HTML5 Web Sockets.</p>");

            return;

        }
```

Next we define a handler for any "onmessage" events *without actually giving the function a standalone name*.

```
        wsHub.socket.onmessage = function(event) {
```

The data comes in over the web socket as a raw string. We have to convert it back into JSON.

```
            message = JSON.parse(event.data);
```

You will notice we do **not** blindly trust the incoming request, but use a "try" block.

```
            try {
                  // Execute global function by stringname
                  window[message.funcname].apply(window, message.args);
```

NOTE - unlike the SNAP Connect “you must state what is to be callable” model, with the above code any Javascript function in your web interface is potentially callable over the web socket. If you wanted to institute some sort of restrictions, the above spot in the code is where you would do so.

```
            } catch (err) {
                  console.log("Error executing websocket message: " + 
err.message);
            }
         }
    },
};
```

This final routine was used to test the initial Web Socket server. I mentioned up above that any function would be callable. Below we have provided a function, just to have something to call. Because do_print() outputs to the web browser’s console log, you can easily verify that the data path between app_server.py and this file is working.

```
// wsHub Callback: Debug log function - TEST
function do_print(message)  {
    console.log(message);
}
```