# http://www.mcwane.com/upl/images/family-of-companies/logos/synapse-wireless-8cccdd3d.pngE20 Kit1 Demo

This demonstration kit showcases the following products:

* SNAP Connect E20
* SN171 Prototyping board, with RF200 module
* SN173 Prototyping board with SM220 module
* SS200 USB SNAP Stick
* SN132 USB SNAP Stick (loaded with SNAP Sniffer image)

The kit ships with a preloaded demonstration application which runs right out of the box. Simply power up the E20 and use a PC or mobile device to connect to its’ Wifi access point:

**SSID**: synapse-e20  
**Password**: synapse1

**Open a web browser, and point to the E20’s URL:** [**http://192.168.0.1**](http://192.168.0.1)

The web page will display a simple table of wireless SNAP nodes reporting “status”. The SN171 and SN173 boards are preloaded with SNAPpy scripts which report status every 5 seconds, or when a button is pressed.

**Connect the battery packs to the SN171 and SN173 boards, and verify each pack’s switch is ON.**

You should see a blinking LED on each prototyping board. Also, you’ll see both devices show up in the HTML table displayed in your web browser.

As you press button-1 on each board, you’ll see the press-count immediately updated in your browser. Also the current state of each button will be reflected in realtime. In addition, the boards report their current battery level.

## Exploring the Demo

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-through (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 is 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 (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 itelf – 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 (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 itelf – 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 (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 itelf – 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).