REFERENCE GUIDE

SNAP Connect® Python Package Manual

Reference Manual for Version 3.1

Wireless Technology to Control and Monitor Anything from Anywhere™

Doc# 600045-01B

Table of Contents

Overview

This manual covers programming with *SNAP® Connect*, which allows you to write applications for PCs and embedded gateways that participate in SNAP networks. SNAP is Synapse's mesh networking protocol stack. The purpose of SNAP is to provide seamless remote control and monitoring capability across a distributed mesh network. This includes the ability to develop and deploy new application software on remote SNAP instances, or "SNAP nodes." The nodes comprising a typical SNAP network range from tiny embedded devices with 8-bit processors to larger embedded Linux boxes, to desktop PCs and even cloud-servers. All SNAP nodes are programmed in the Python language, which combines ease-of-use with portable execution on a virtual machine architecture.

All SNAP Nodes:

- ***Route SNAP mesh traffic*** through the network using the available platform interfaces (TCP/IP, serial, USB)
- Execute ***Python scripts***, providing full programmer access to the host platform's capabilities
- Have a unique ***SNAP MAC address***, so they can be found by other SNAP nodes

SNAP Connect is an implementation of SNAP for your PC. It can be hosted on full Python-capable platforms such as Windows, Linux, and OSX.

Why run SNAP on a PC? Typical applications are:

- User Interface
  - Command-line
  - Standalone Windows GUI
  - Web Server back-end
- Data Logger
  - Collect remote data and store in a database or file-system
- Interface Adaptor
  - MODBUS TCP server, fronting for remote wireless devices
  - Legacy application protocols (serial or TCP/IP)
- Mesh Aggregator / Internet Bridge
  - Unify remote device networks into one network
  - Access remote networks from the Internet

It's a Standalone Application. It's a Software Library. It's both!

SNAP Connect can be used as a gateway "right out of the box," or it can be "baked in" to software applications that you develop in Python. Consider that *embedded* SNAP nodes can be used to route SNAP mesh traffic even when no user application script is loaded. With *SNAP Connect*, this capability is present in the form of a basic example application (Python script) included in the default installation. Look for "simpleSnapRouter.py", documented toward the back of this manual. This standalone SNAP instance listens for connections over TCP/IP, while command line options support opening additional serial connections (up to three).

How does my PC know what SNAP Address it should use?

Each instance of *SNAP Connect* requires a **License File**. This License File specifies what SNAP Addresses you are authorized to use.

If you are running *SNAP Connect* on a Synapse *SNAP Connect* E10, then you already have a license – just skip to the next section of this document.

If you are running SNAP on a desktop PC, then it comes with a default "license.dat" file that makes your PC be SNAP Address 00.00.20 on the SNAP network (similar to how Portal defaults to address 00.00.01).

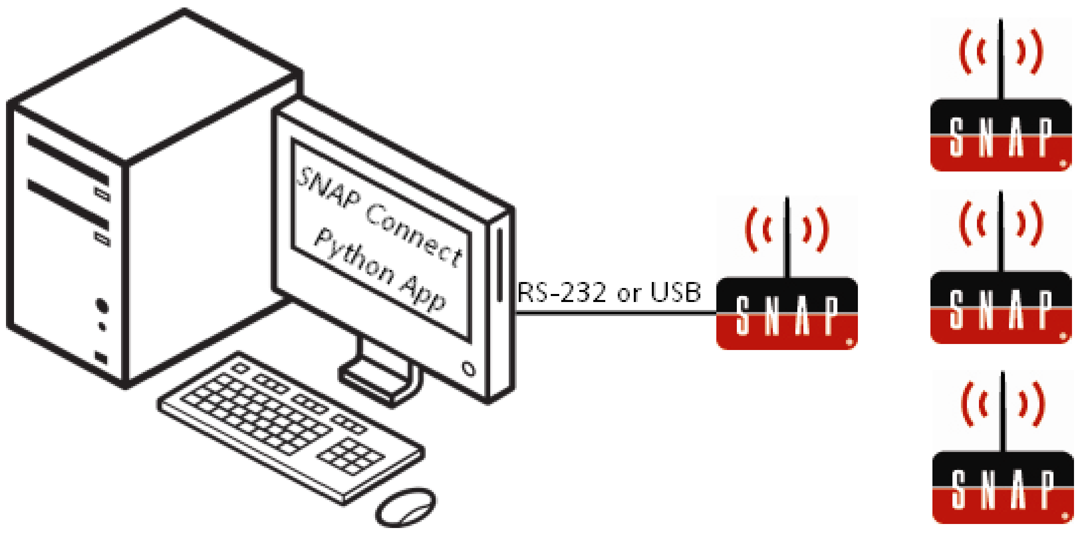
If you need your PC to have a different SNAP Address (for example, you are deploying multiple PCs in the same SNAP network), then you will need to purchase an alternate license.dat file.

Contact Synapse at 1-877-982-7888 or e-mail SNAPconnectlicense@synapse-wireless.com to order one or more *SNAP Connect* Licenses.

Once you have obtained your replacement license file, simply put the "license.dat" file in the same directory as your application software.

Introducing *SNAP Connect*

SNAP Connect is a pure Python package to interface your application to a SNAP network. This Python package is a full SNAP implementation, allowing you to create programs that natively interact with the SNAP network.



If you are not already familiar with SNAP networking and SNAPpy scripting please refer to Synapse's **SNAP Primer** ("SNAP Primer.pdf") and **SNAP Reference Manual** ("SNAP Reference Manual.pdf"). These documents (and others) can be found online at forums.synapse-wireless.com. They also come bundled with Synapse's *Portal* software. It is strongly recommended that you become familiar with the contents of this reference manual and have a copy of *Portal* installed before attempting to develop custom applications using *SNAP Connect*. This document also assumes that you have a good working knowledge of using the Python programming language.

SNAP Connect Licensing

There are several licensing options available when using *SNAP Connect*:

Option 1 - Evaluation License (10 nodes or less)

Cost – free!

This is a relatively new licensing option for *SNAP Connect*, introduced to make it easier for customers to quickly get started working on their SNAP applications. Previously, evaluators had to request a **time-limited** *SNAP Connect* evaluation license file, and could not run *SNAP Connect* at all until they had done so. Now a “fixed address, 10-node” license is bundled in with the *SNAP Connect* library.

Applications that only require a small number of SNAP nodes (up to 10 embedded SNAP Nodes, not counting the *SNAP Connect* application and optionally *Portal*) require no additional license.

Note that the address of the *SNAP Connect* node will be fixed at SNAP Address 00.00.20.

(Every node on a SNAP network must have a unique SNAP Address).

To move your *SNAP Connect* instance to a different (unique) address, you will have to license an additional SNAP MAC Address. Contact Synapse customer support for pricing and availability. This will be sent to you in the form of a replacement “license.dat” file.

Option 2 - Unrestricted License (more than 10 nodes)

Contact Synapse for pricing and availability.

Applications that require larger SNAP networks are still required to obtain a full *SNAP Connect* license, just like with previous versions.

Note that the unrestricted license will include a replacement SNAP Address, you will no longer be restricted to SNAP Address 00.00.20.

Installation

NOTE – if you are using a SNAP Connect E10, it comes with SNAP Connect preinstalled. You can skip ahead to the next section. Of course, these are the instructions you would follow if you later wanted to upgrade the version of SNAP Connect installed on your E10.

SNAP Connect requires that a copy of Python version 2.5, 2.6, or 2.7 is installed on the target computer. *SNAP Connect* does not currently support Python version 3.x.

If you don't already have Python installed on your development computer, please go to http://www.python.org to download and install a version of Python for your platform.

If AES-128 support is required, the third-party library PyCrypto (http://pycrypto.org/) is also needed. If you do not know whether you will need AES-128 support, you can go ahead and install *SNAP Connect* now and install the PyCrypto library later when it is needed.

Installation of *SNAP Connect* will be a familiar process to those using other add-on Python packages.

Step 1 – Obtain the latest SNAP Connect ZIP file

These can be downloaded from our user support forum at http://forums.synapse-wireless.com.

NOTE – be sure to grab the version that matches your installed version(s) of Python.

The various "installation ZIP" files all have filenames of the form:

snapconnect-x.y.z-python2.5.zip

snapconnect-x.y.z-python2.6.zip

snapconnect-x.y.z-python2.7.zip

where "x.y.z" will be replaced with the version number of the *SNAP Connect* release, in major.minor.build format, for example, snapconnect-3.1.0-python2.5.zip.

Step 2 – Unzip the ZIP file

Using the tool of your choice (7ZIP, PKZIP, etc.), unzip the file into a temporary directory.

Recent versions of Windows can "unzip" these files natively (no third-party software required.)

On many Linux systems (including the *SNAP Connect E10*) the command is simply:

```
unzip snapconnect-x.y.z-python2.6.zip
```

Step 3 – Go into the newly created directory

Using the "cd" or "chdir" commands as appropriate for your operating system.

Step 4 – Look in the README.txt file

Detailed instructions will be given in that text file, but basically what you will do is something like:

```
python setup.py install
```

Refer to the README.txt file for the latest installation instructions.

Step 5 (optional) – Unzip the EXAMPLES onto your computer too

Several example *SNAP Connect* applications are bundled into an “examples.zip” file (also available from https://forums.synapse-wireless.com).

Using the tool of your choice (7ZIP, PKZIP, etc.), unzip the file “examples.zip” into a working directory somewhere on your computer.

Recent versions of Windows can “unzip” these files natively (no third-party software required.)

On many Linux systems (including the *SNAP Connect E10*) the command is simply:

```
unzip examples.zip
```

NOTE - Be sure to specify “use directory paths” when you unzip the examples!

This will create an “examples” directory tree with a “manual” subdirectory (plus several others) underneath it.

As you might guess, the “manual” subdirectory contains all the examples from **this** User Manual.

You will also see subdirectories like:

DisplayInfo

EchoTest

pachube

SpyUpload

Etc.

Each of these contains another example program.

For more about the available example programs, refer to section Additional Examples of this document.

Sample Application: McastCounter

If you have previously used SNAP then you are probably familiar with the multi-cast counter example. In this example we will show how to write an equivalent Python application using *SNAP Connect*, based on the SNAPpy McastCounter.py script. Let’s take a look at the contents of our sample application:

```
import logging
from snapconnect import snap

class McastCounterClient(object):
    def __init__(self):
        # Create a SNAP instance
        self.snap = snap.Snap(funcs={'setButtonCount': self.set_button_count})
        # Open COM1 (port 0) connected to a serial SNAP bridge
        self.snap.open_serial(snap.SERIAL_TYPE_RS232, 0)
        # Create a logger
        self.log = logging.getLogger("McastCounterClient")

    def set_button_count(self, count):
        # This function is called by remote SNAP nodes
        # Log the current count received
        self.log.info("The button count = %i" % (count))

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO) # print logging messages to STDOUT
    client = McastCounterClient() # Instantiate a client instance
    client.snap.loop() # Loops waiting for SNAP messages
```

Details on all of the parameters that *SNAP Connect* methods take are described in the *SNAP Connect* API section of this manual.

Let’s walk through this simple multi-cast counter example to see how it works. The only two packages we import in this example are the standard python logging package and the *SNAP Connect* package. Once we define the McastCounterClient class, in the init function we instantiate the *SNAP Connect* instance we will use. (Look for self.snap in the sample code). After we instantiate the *SNAP Connect* instance we pass it a Python dictionary with the functions we would like to make publically available to be called by other SNAP nodes. In this example the only publicly available function we register is the function named “setButtonCount”, and when the function is called by other SNAP nodes it will execute our “set_button_count” method. Next we call the method “open_serial” on our *SNAP Connect* instance to have it open a serial connection to the SNAP bridge node. Lastly, we create a logging instance to log anything we receive from a SNAP node.

The “set_button_count” method accepts one parameter, which contains the current count as sent by the remote SNAP node. When this method is called all our function does is use the saved logger instance and log the current button count.

Finally, in the main section we first configure the default logger to print any logged messages to standard out. We then create an instance of the client class that we defined earlier in the example code. At the end we call the loop method of the *SNAP Connect* instance to start having *SNAP Connect* continuously send, receive, and process SNAP messages. Until we call the loop method, *SNAP Connect* will not be able to do anything we ask it to do.

To run this program, you should first change the open_serial() COM port number to that of your bridge device. If you're using a USB "SnapStick", you'll need to change the SERIAL_TYPE to match also (see the open_serial API docs later in this manual). Use Portal to verify the bridge device is working, and to upload the standard "McastCounter" example (installed with Portal) into a wireless test-device (e.g. Synapse SN171 protoboard). Now **disconnect** Portal from the bridge, since only one program at a time can have the serial connection open, and your SNAP Connect program will be connecting. At this point, you will be able to run the python program:

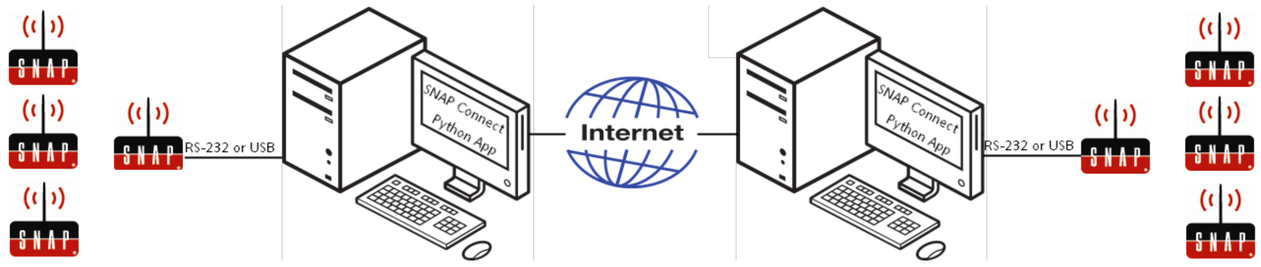
```
>python SimpleMcastCounter.py

Permanent license created on 2012-01-26 23:01:39.808000 for 000020
```

If all went well, you will see the above message, and the program will be waiting for SNAP messages. Now you can press the button on your wireless test-device and the "button count" log messages will appear on your computer console. After testing this program, press ctrl-C to terminate the test.

Remote Connection Capabilities

You can connect multiple *SNAP Connect* instances together over a local TCP/IP network, or the Internet. This capability allows SNAP networks to route their messages over radio interfaces as well as over existing TCP/IP networks.

For example, let's say you have an existing SNAP network located in Huntsville, Ala. Recently you opened a new location, with a SNAP network in San Jose, Calif., and would like to connect the two SNAP networks together. In your Huntsville location you could configure your *SNAP Connect* application to listen for remote connections. Then in your San Jose location you could configure your *SNAP Connect* application to connect to the IP address of the *SNAP Connect* application in Huntsville. This would enable SNAP messages from each location to be routed over the Internet allowing for SNAP nodes from each location to communicate with each other.



In general there are no configuration changes that need to be made on your existing SNAP nodes to enable this capability. However, it is important to note that each *SNAP Connect* instance counts as a mesh routing hop. Depending on how your SNAP nodes are currently configured the maximum number of hops mesh routing messages can take may need to be increased. Also, depending on how much latency there is on the TCP/IP connection between the two *SNAP Connect* instances, the mesh routing timeouts may need to be adjusted.

Remote SNAP TCP/IP connections require authentication between the two ends, and this is done without sending a password in clear text over the network. (*SNAP Connect* uses digest access authentication over the network to protect login credentials.) There are default authentication credentials (username = “public” and password = “public”) that are used if none are provided.

If you prefer, you can create your own system for supplying and validating credentials by adding one function on the client end to provide the username and password, and another function on the server end to validate that the specified username/password pair is correct. These functions can use hard-coded pairs, database or file lookups, or user prompts to generate their data, according to your needs. To replace the default authentication, specify the names of your new routines in your `connect_tcp()` function call on your client, and `accept_tcp()` function on your server. (See the API section for more details on these functions.)

NOTE – if you change this, Portal won’t be able to remotely access your SNAP Connect instance (Portal is currently hard-coded to the *default* authentication values).

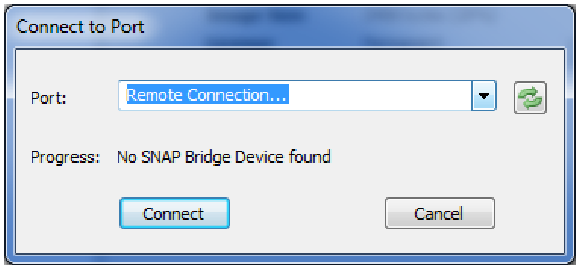
Sample Application: Remote TCP Access

In this sample application we will be building on our previous multi-cast counter example, by showing how to enable remote TCP connections. SNAP Connect programs can connect to each other over an IP network, forwarding routing and RPC traffic among all connected interfaces. A common use of this capability is to allow remote TCP/IP connection from Portal. A small modification of our previous example allows it to accept connections from other SNAP Connect instances, including Portal.

```
import logging
from snapconnect import snap

class McastCounterClient(object):
    def __init__(self):
        # Create a SNAP instance
        self.snap = snap.Snap(funcs={'setButtonCount': self.set_button_count})
        # Open COM1 (port 0) connected to a serial SNAP bridge
        self.snap.open_serial(snap.SERIAL_TYPE_RS232, 0)
        # Accept connections from other SNAP Connect instances and Portal
        self.snap.accept_tcp()

        # Create a logger
        self.log = logging.getLogger("McastCounterClient")

    def set_button_count(self, count):
        # This function is called by remote SNAP nodes
        # Log the current count received
        self.log.info("The button count = %i" % (count))

if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO) # print logging messages to STDOUT
    client = McastCounterClient() # Instantiate a client instance
    client.snap.loop() # Loops waiting for SNAP messages
```

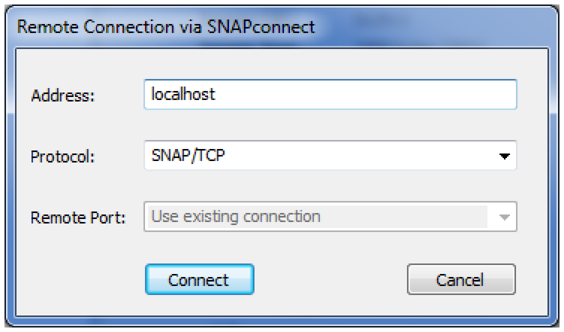
As you have probably noticed, this script is almost exactly the same as our first sample application. The main difference here is that in addition to connecting to a SNAP bridge node via serial port, we configure the SNAP instance to listen for and accept remote SNAP TCP/IP connections.

When you run this program, you can test it with your remote SNAP device as with the first example and see the button-count logged to the application console. But now you can simultaneously connect to this network with ***Portal***. In Portal's connect dialog, select "Remote Connection…" and then connect to *localhost*.


Connect to Port

Port: Remote Connection…

Progress: No SNAP Bridge Device found


Remote Connection via SNAPconnect

Address: localhost

Protocol: SNAP/TCP

Remote Port: Use existing connection

With Portal connected over TCP/IP you can do a broadcast-ping and see all of the SNAP nodes in your network, including the SNAP Connect example program above, as well as its bridge node and remote node(s). From Portal's node-info panel, invoke the 'incrementCount()' function on the remote SNAP device. You will see the count LEDs increment on the remote device itself, and the SNAP Connect console will log the event.

Recommended Practices

When writing your *SNAP Connect*-based application there are a few recommended practices that we would like to describe.

Hooks

The first recommend practice is to take advantage of the available hooks for controlling which code is executed when events occur in *SNAP Connect*. The available hooks are listed in the *SNAP Connect* API section, but the most important to remember are HOOK_RPC_SENT and HOOK_SNAPCOM_OPENED.

Just as HOOK_RPC_SENT is important in a SNAPpy script, it is important in *SNAP Connect* applications. This hook allows your program to know when the system has attempted to send a specific RPC (either unicast or multicast). Based on this knowledge it might be appropriate for your application to send another RPC or know that there is room in the queue to send another RPC. Also, just as in SNAPpy, the `rpc()` method returns immediately and does not mean your RPC was sent that instant. This hook does not trigger until *SNAP Connect* is done processing the specified packet (whether that processing was successful or not).

The HOOK_SNAPCOM_OPENED informs your application that you have successfully connected to a remote *SNAP Connect* instance or another *SNAP Connect* instance has connected to you. This is important to know if your application depends on having connectivity to another instance. If your application is not connected to another *SNAP Connect* instance or a SNAP bridge node, there is no one else to talk to so all your RPCs will fail.

Asynchronous Programming

SNAP Connect is designed to run in a single process within a single thread. As you have seen in our sample applications, once *SNAP Connect* and your application have been initialized, we call the *SNAP Connect* `loop()` method. This method is a convenience method that is a "while True" loop that continually calls the *SNAP Connect* `poll()` method:

```
def loop(self):
    while True:
        self.poll()
```

As mentioned in the *SNAP Connect* API, the poll method calls the necessary components that *SNAP Connect* uses. Since your program is essentially always running that loop, your application must be programmed in an asynchronous fashion.

To aid in writing asynchronous programs, access to the asynchronous event scheduler that *SNAP Connect* uses is available. The event scheduler that *SNAP Connect* uses is part of a Python package called "apy".

The apy Package

"apy" is short for "asynchronous Python" and is an open-source pure Python package that comes bundled with *SNAP Connect*. It is also available from http://sourceforge.net/projects/apy/.

The apy event scheduler is straightforward to use and allows your program to schedule functions to be called. Access to the event scheduler is available by calling methods on the "scheduler" property of your *SNAP Connect* instance. To schedule a function to be called later you would call the schedule method. For example, to schedule sending a multi-cast RPC call every second you could do:

```
def my_func(self):
    self.snap.mcast_rpc(1, 1, "hello")
    return True

def start_mcasting(self):
    self.snap.scheduler.schedule(1.0, self.my_func)
```

The signature for the schedule function looks like:

```
def schedule(self, delay, callable, *args, **kwargs)
```

where the parameters are:

`delay` : The delay, in seconds, to use before calling the callable argument. A delay of 0 indicates the callable should be called on the next poll interval

`callable` : The callable object to schedule

`*args, **kwargs` : Optional arguments and keyword argument to pass to the callable object

The schedule function returns an EventElement object that has a method called "Stop" that enables you to cancel the event (in other words, cancel having the callable be called).

The last thing to know about using the apy event scheduler is how to re-schedule a running function after it runs. When your function returns after being called by the event scheduler, its return value is checked. If the return value is `True`, then the event is automatically rescheduled using the original delay value. If you would like to reschedule with a different delay, your function can return a number (integer or float, indicating a delay duration in seconds) as a different valid delay value. Lastly, if you don't want your function to be rescheduled at all, your function can return `None` or `False`.

Multiple Instances

When attempting to run multiple instances of *SNAP Connect* it is important that each instance uses a unique SNAP address. Just as regular SNAP nodes need a unique address to differentiate each other over the air, *SNAP Connect* instances need unique addresses on the network as well. A single license file can contain more than one SNAP network address. When you instantiate your *SNAP Connect* instance you can provide as a keyword argument which SNAP network address out of the license you would like to use. See the `__init__()` function for more details.

SNAP Connect on the E10

Previous examples in this document have been somewhat “generic,” since we had no way of knowing exactly what platform you were running *SNAP Connect* on.

One of the platforms *SNAP Connect* can be run on is the **Synapse *SNAP Connect* E10**.

The *SNAP Connect* E10 is basically a small Linux computer that comes with *SNAP Connect* preloaded.

In this section, we go over some example code that is specific to the E10. The intent is that by showing something concrete, it will further clarify the material covered previously in this document.

For the purpose of these examples, all you need to know about the E10 is that it has both an internal SNAP Engine *and* a separate ARM processor that runs the Linux OS (on which is running the *SNAP Connect* application). The SNAP Engine connects to the ARM processor using its serial ports.

The SNAP Engine has control of the tricolor LED labeled “A,” and the Linux processor has control of the tricolor LED labeled “B” as well as direct access to the Push Button labeled “MODE.” (Tricolor LEDs can display green or red, or amber as a combination of green and red.)

For more about the E10, refer to the “*SNAP Connect* E10 User Guide.”

UserMain.py

The following is the example Python script that was included on the version 1.0 E10 units. Newer units may have an updated version of this script; take a look in the "/root" directory of your E10 to be sure.

First we list the entire file, then we list it again with inline commentary.

```
#
# A minimal example SNAP Connect 3.0 application. This example was
# deliberately kept simple, and so does not showcase the full power
# of Python or SNAP Connect 3.0. It enables RPC access via TCP/IP
# or via SNAP Engine (2.4 GHz radio, etc.), and gives you control
# of the Linux processor's "B" LED and Push Button.
#

from snapconnect import snap

import os # So we can gain access to the LEDs and Push Button

#
#
# Example routines to control the "B" LED on the E10
# (The "A" LED is connected to the internal SNAP Engine inside the E10)
#
#

#
# Individual LED pin control (used internally)
#
def setGreenLed(value):
    os.system("echo "+str(value)+" >> /sys/class/leds/greenled/brightness")

def setRedLed(value):
    os.system("echo "+str(value)+" >> /sys/class/leds/redled/brightness")

#
# It's actually a tri-color LED (this will be the public API)
#
def setLedBOff():
    setGreenLed(0)
    setRedLed(0)

def setLedBGreen():
    setGreenLed(1)
    setRedLed(0)

def setLedBRed():
    setGreenLed(0)
    setRedLed(1)

def setLedBYellow():
    setGreenLed(1)
    setRedLed(1)

#
# Example routine to read the MODE push button
# Normally a pullup resistor holds the pin high/True
# Pushing the button connects the physical pin to ground (low/False)
#
def readButton():
    result = os.system("/usr/bin/gpio9260 ?PB10")
    return result != 0 # Forcing boolean result

def server_auth(realm, username):
    """
    An example server authentication function

    Returns the password for the specified username in the realm

    realm : This server's realm
    username : The username specified by the remote server
    """
    if username == "public":
        return "public"


if __name__ == '__main__':
    import logging

    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s:%(msecs)03d
%(levelname)-8s %(name)-8s %(message)s', datefmt='%H:%M:%S')

    funcdir = { # You CHOOSE what you want to provide RPC access to...
               'setLedBOff'    : setLedBOff,
               'setLedBGreen'  : setLedBGreen,
               'setLedBRed'    : setLedBRed,
               'setLedBYellow' : setLedBYellow,
               'readButton'    : readButton
              }
    com = snap.Snap(funcs=funcdir)

    # Make us accessible over TCP/IP
    com.accept_tcp(server_auth)

    # Make us accessible over our internal SNAP Engine
    com.open_serial(1, '/dev/ttyS1')

    #
    # Configure some example settings
    #

    # No encryption
    com.save_nv_param(snap.NV_AES128_ENABLE_ID, False)

    # Lock down our routes (we are a stationary device)
    com.save_nv_param(snap.NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID, 0)

    # Don't allow others to change our NV Parameters
    com.save_nv_param(snap.NV_LOCKDOWN_FLAGS_ID, 0x2)

    # Run SNAP Connect until shutdown
    while True:
        com.poll()
    com.stop_accepting_tcp()
```

Let’s go through this file again, section by section.

```
#
# A minimal example SNAP Connect 3.0 application. This example was
# deliberately kept simple, and so does not showcase the full power
# of Python or SNAP Connect 3.0. It enables RPC access via TCP/IP
# or via SNAP Engine (2.4 GHz radio, etc.), and gives you control
# of the Linux processor's "B" LED and Push Button.
#

from snapconnect import snap
```

The above import provides access to the *SNAP Connect* library. The “snapconnect” in the above statement refers to the directory containing that library. If your directory has a different name, simply modify the import statement appropriately. (For example, early versions of the E10 had this package in the Snap directory instead of the snapconnect directory.)

```
import os # So we can gain access to the LEDs and Push Button
```

The “os” module is a standard Python library of routines that allow you to do things like invoke other programs. We are taking advantage of the fact that we already know how to control the LED and read the button from the Linux command line, and will use OS calls those command-line commands from within our script. (Refer to the E10 User Guide.)

We also wanted to showcase the idea that if your platform can do it, you can access that functionality from *SNAP Connect*. For features that are directly supported by Python, simply import the corresponding module. For features that do not have direct Python support, you can access the Linux command line.

```
#
#
# Example routines to control the "B" LED on the E10
# (The "A" LED is connected to the internal SNAP Engine inside the E10)
#
#

#
# Individual LED pin control (used internally)
#
def setGreenLed(value):
    os.system("echo "+str(value)+" >> /sys/class/leds/greenled/brightness")

def setRedLed(value):
    os.system("echo "+str(value)+" >> /sys/class/leds/redled/brightness")
```

The “system()” function in the standard Python “os” module allows you to invoke any function you could do directly from the command line.

On the E10’s version of Linux, commands like…

```
echo 0 >> /sys/class/leds/greenled/brightness
echo 1 >> /sys/class/leds/redled/brightness
```

…can be used to write directly to the LED hardware.

By wrapping these two primitive operations up in subroutines, we hide the complexity from the rest of the code.

```
#
# It's actually a tri-color LED (this will be the public API)
#
def setLedBOff():
    setGreenLed(0)
    setRedLed(0)

def setLedBGreen():
    setGreenLed(1)
    setRedLed(0)

def setLedBRed():
    setGreenLed(0)
    setRedLed(1)

def setLedBYellow():
    setGreenLed(1)
    setRedLed(1)
```

Above you can see that we use those primitive access routines to implement a nicer API. Note that just because the routines have been implemented, that does not mean they can be called via RPC. We specify that *later* in this same code.

```
#
# Example routine to read the MODE push button
# Normally a pullup resistor holds the pin high/True
# Pushing the button connects the physical pin to ground (low/False)
#
def readButton():
    result = os.system("/usr/bin/gpio9260 ?PB10")
    return result != 0 # Forcing boolean result
```

Here we again use the ability to run command line programs – in this case, the “gpio9260” program that comes preloaded on the E10, to read the MODE button on the front of the E10.

```
def server_auth(realm, username):
    """
    An example server authentication function

    Returns the password for the specified username in the realm

    realm : This server's realm
    username : The username specified by the remote server
    """
    if username == "public":
        return "public"
```

Here we provide an explicit “authorization” routine for the E10. By changing this routine, more complex authorization mechanisms could be implemented. For example, users and passwords might get looked up in a database, or at a minimum there might be separate hard-coded username/password pairs per realm.

```
if __name__ == '__main__':
    import logging
    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s:%(msecs)03d
%(levelname)-8s %(name)-8s %(message)s', datefmt='%H:%M:%S')
```

Here we are using standard Python logging. Nothing about the above three lines of code is specific to *SNAP Connect*.

```
    funcdir = { # You CHOOSE what you want to provide RPC access to...
               'setLedBOff'    : setLedBOff,
               'setLedBGreen'  : setLedBGreen,
               'setLedBRed'    : setLedBRed,
               'setLedBYellow' : setLedBYellow,
               'readButton'    : readButton
              }
    com = snap.Snap(funcs=funcdir)
```

We mentioned previously that just because a function has been defined does not automatically make it callable from other SNAP Nodes. Here the code fills in a Python dictionary (notice the {…}) with the set of Python functions that *are* to be callable via RPC, as well as the names we want to use. The code enables the four functions to set the LEDs to off or one of three colors, but does *not* include the more fundamental functions that specifically control the red and green LEDs.

In this example the mapping is one to one (nothing was renamed), but the renaming capability is there when you need it.

```
    # Make us accessible over TCP/IP
    com.accept_tcp(server_auth)
```

We tell *SNAP Connect* to listen for incoming TCP/IP connections.

```
    # Make us accessible over our internal SNAP Engine
    com.open_serial(1, '/dev/ttyS1')
```

There is not much mysterious here, once you know that "/dev/ttys0" is the command line console on the E10, and "/dev/ttyS1" is the UART that connects to the serial port on the internal SNAP Engine.

```
    #
    # Configure some example settings
    #

    # No encryption
    com.save_nv_param(snap.NV_AES128_ENABLE_ID, False)

    # Lock down our routes (we are a stationary device)
    com.save_nv_param(snap.NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID, 0)

    # Don't allow others to change our NV Parameters
    com.save_nv_param(snap.NV_LOCKDOWN_FLAGS_ID, 0x2)
```

If we *did* want to enable encryption, we would need to also specify the 16-byte encryption key.

Setting the "max timeout" for mesh routing to 0 reduces the number of "route discoveries" that have to take place. If your nodes are not stationary, setting this to 0 may not be a good idea.

Since we consider the E10 in our example to be the master, we want to maintain local control over its NV parameters. The E10 allows a value of 2 for the lockdown parameter to prevent changes to any NV parameters. (This is not a valid value in most SNAP nodes, where a value of 1 prevents over-the-air changes to the SNAPpy script in the node.)

Other settings for these parameters (as well as many other parameters) are possible. The above is only one example configuration.

```
    # Run SNAP Connect until shutdown
    while True:
        com.poll()
    com.stop_accepting_tcp()
```

We could have just called the `com.loop()` function and done away with the while loop. However, we wanted to show that your Python program can remain active, even with the *SNAP Connect* library active.

This particular example program is essentially “passive,” in that it mostly waits for other nodes to call on it (via RPC).

A more “active” program could perform additional functions after every call to `com.poll()`. For example, it could monitor the MODE pushbutton (we’ve already provided a function to do the actual “read”), and could invoke an RPC (or take some other action) whenever that button was pressed. As it stands, we’ve enabled monitoring of the button but perform no such monitoring locally.

With this script running on the E10, any node networked to the E10 can send an RPC to the E10 to control the device’s LED color. That function call can be originate from the SNAP Engine contained within the E10, from a SNAP node connected to that SNAP Engine over the air, or from a SNAP node connected to the E10 over the TCP/IP connection.

Sample Application – simpleSnapRouter.py

The following is one of the example Python scripts that are included with *SNAP Connect*. (look in the “examples\manual” subdirectory.) The simpleSnapRouter script performs exactly that function: it is a simple SNAP router. A PC or device running this script with *SNAP Connect* forms a nexus between any of several disparate networks.

For example, in addition to a TCP/IP connection, you could make a connection over a COM port to a network of 900 MHz SNAP nodes, make a connection over an SS200 to a network of 2.4 GHz SNAP nodes running at 2 Mbps, and make a connection over the USB port to a network of 2.4 GHz SNAP nodes running at 250 Kbps – and all three of these subnetworks would be able to communicate with each other.

Usage

Here are some examples of launching simpleSnapRouter.py:

```
python simpleSnapRouter.py –-help
```

Launching the application this way will display info on the various command line arguments, and then the program will exit. Note that there are two dashes before the word “help”.

```
python simpleSnapRouter.py –h
```

You will notice a pattern here. For most of the command line arguments, there is both a long form (a keyword) prefixed by two dashes, and a short form (single character) prefixed by one dash. So for example, --help and –h are equivalent.

```
python simpleSnapRouter.py
```

Running the application with no command line arguments will result in its default behavior – it will listen for incoming TCP/IP connections, but will not enable any other interfaces (even if they are present).

```
python simpleSnapRouter.py –-deaf
```

or

```
python simpleSnapRouter.py –d
```

Running the application with this command line argument (either form) will override the default “listening” behavior. Incoming TCP/IP connections will be rejected.

NOTE – using this option by itself is not very useful.

```
python simpleSnapRouter.py <IP Address>
python simpleSnapRouter.py 192.168.1.55 192.168.1.56
```

Running the application with one or more valid IP addresses as command line arguments will force *outbound* connections to be attempted to other *SNAP Connect* instances. This includes other instances of simpleSnapRouter.py.

```
python simpleSnapRouter.py –c <COM Port Number>
python simpleSnapRouter.py –-com <COM Port Number>
python simpleSnapRouter.py –-com 0
```

By specifying a “COM” port using the -c or --com options (0 = COM1, 1=COM2, etc.) you can enable the SNAP Packet Serial protocol on the specified serial port. Assuming there is a live bridge node connected and listening on that COM port, this will allow simpleSnapRouter.py to communicate over the air to other SNAP Nodes.

NOTE – COM ports include both “real” RS-232 ports as well as many common brands of “USB Serial” adapter.

```
python simpleSnapRouter.py –s <SS200 Number>
python simpleSnapRouter.py –-ss200 <SS200 Number>
python simpleSnapRouter.py –-ss200 0
```

By specifying a “SS200” device (the first one detected by your computer is number 0, the next one plugged in will become number 1, etc.) you can enable the SNAP Packet Serial protocol on the specified SNAP Stick 200.

```
python simpleSnapRouter.py –l <Synapse USB Device Number>
python simpleSnapRouter.py –-legacy <Synapse USB Device Number>
python simpleSnapRouter.py –-legacy 0
```

By specifying a “legacy” device (the first one detected by your computer is number 0, the next one plugged in will become number 1, etc.) you can enable the SNAP Packet Serial protocol on the specified USB interface. Examples of legacy devices include the SN132 SNAP Stick and the SN163 Bridge Demonstration Board.

Note that this application *as written* only supports one *of each type* of serial connection to be enabled. In other words, simpleSnapRouter.py can establish a maximum of three simultaneous serial connections, assuming your computer has a COM port, an SS200, and a legacy USB device connected.

Also note that the example program has been written to ignore any “serial” errors, on the assumption that the user would rather the program continue to run with any remaining interfaces still operational. (For example, if the SS200 SNAP Stick were unplugged from the computer, the serial and TCP/IP connections may still be sufficient.)

```
python simpleSnapRouter.py –u <user name>
python simpleSnapRouter.py –-user <username>
python simpleSnapRouter.py –-user public
python simpleSnapRouter.py –p <password>
python simpleSnapRouter.py –-password <password>
python simpleSnapRouter.py –-password public
```

By specifying the --user and --password options, you can override the default credentials used by *SNAP Connect*.

NOTE – If you are going to enable alternate credentials, you need to do so throughout your SNAP Network.

Source Code

First we list the entire simpleSnapRouter file. We then we provide additional commentary after the source listing.

```
"""
simpleSnapRouter.py - an example for the SNAP Connect User Manual

By default, enables listening for incoming TCP/IP connections.

Command line options allow:

1) NOT listening for incoming TCP/IP connections
2) Establishing a serial connection on a "COM" port
3) Establishing a serial connection on a legacy USB device like a Synapse SN163 or SN132
4) Establishing a serial connection on a USB device like a SS200
5) Connecting to other SNAP Connect instances over TCP/IP
6) Changing the user, and password settings from their default value of "public"
"""

import logging
from optparse import OptionParser
from snapconnect import snap

DEFAULT_USER = 'public'
DEFAULT_PASSWORD = 'public'

my_user = DEFAULT_USER
my_password = DEFAULT_PASSWORD

def my_client_auth(realm):
    """
    An example client authorization function

    Returns the previously set username and password to the remote server
    to use as it's authorization, *if* the specified realm is correct

    realm : The realm specified by the remote server. Note that this is
    intended for future expansion. Currently it is always 'SNAPcom'.
    """
    return (my_user, my_password)

def my_server_auth(realm, username):
    """
    An example server authentication function

    Returns the password for the specified username in the realm

    realm : This server's realm (currently unused, future expansion)
    username : The username specified by the remote server
    """
    if username == my_user:
        return my_password

def echo(obj):
    """Just a function that can be called remotely"""
    print str(obj)

def main():
    global comm
    global my_user, my_password

    usage = "usage: %prog [options] [other-node-IP-addresses]"
    parser = OptionParser(usage)

    parser.add_option("-d", "--deaf", action="store_true", dest="no_listener", default=False,
help="DON'T listen for incoming TCP/IP connections (default is DO listen)")
```

```
    parser.add_option("-c", "--com", dest="comport", help="open a connection on the specified COM
port (0 = COM1)")
    parser.add_option("-s", "--ss200", dest="ss200", help="open a connection on the specified
SS200 device (0 = SNAPstick0)")
    parser.add_option("-l", "--legacy", dest="usb", help="open a connection on the specified USB
device (0 = USB0)")

    parser.add_option("-u", "--user", dest="user", default=DEFAULT_USER, help='specify an
alternative SNAP Realm user name(default is "public")')
    parser.add_option("-p", "--password", dest="password", default=DEFAULT_PASSWORD,
help='specify an alternative SNAP Realm password (default is "public")')

    (options, args) = parser.parse_args()

    print options
    print args

    funcdir = {"echo": echo}
    comm = snap.Snap(funcs=funcdir)

    my_user = options.user
    my_password = options.password

    if options.no_listener:
        print "Listening for incoming TCP/IP connections has been disallowed by user"
    else:
        print "Listening for incoming TCP/IP connections"
        comm.accept_tcp(auth_info=my_server_auth)

    if len(args) != 0:
        # Establish requested outbound TCP/IP connections
        for arg in args:
            print "Establishing a TCP/IP link to "+arg
            comm.connect_tcp(arg, auth_info=my_client_auth)

    # Did the user ask us to open a normal serial port connection (COM1-COMn)?
    if options.comport != None:
        port = int(options.comport)
        print "Opening a serial connection on COM%d" % (port+1)
        try:
            comm.open_serial(snap.SERIAL_TYPE_RS232, port)
        except Exception, e:
            print "Invalid COM port specified"

    # Did the user ask us to open a connection to a SS200?
    if options.ss200 != None:
        port = int(options.ss200)
        print "Opening a serial connection to SNAPstick%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK200, port)
        except Exception, e:
            print "Invalid SS200 device specified"

    # Did the user ask us to open a connection to a legacy Synapse USB device?
    if options.usb != None:
        port = int(options.usb)
        print "Opening a serial connection on USB%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK100, port)
        except Exception, e:
            print "Invalid USB port specified"

    comm.loop()

if __name__ == '__main__':
    logging.basicConfig(filename='simpleSnapRouter.log', level=logging.DEBUG, format='%(asctime)s
%(levelname)s: %(message)s', datefmt='%Y-%m-%d %H:%M:%S')
    main()
```

Source Code with Commentary

```
"""
simpleSnapRouter.py - an example for the SNAP Connect User Manual

By default, enables listening for incoming TCP/IP connections.

Command line options allow:

1) NOT listening for incoming TCP/IP connections
2) Establishing a serial connection on a "COM" port
3) Establishing a serial connection on a legacy USB device like a Synapse SN163 or SN132
4) Establishing a serial connection on a USB device like a SS200
5) Connecting to other SNAP Connect instances over TCP/IP
6) Changing the user, and password settings from their default value of "public"
"""

import logging

from optparse import OptionParser
```

So far, this is pretty standard stuff. The only thing new here is the import of optparse, a standard Python library for working with command line arguments.

```
from snapconnect import snap

DEFAULT_USER = 'public'
DEFAULT_PASSWORD = 'public'

my_user = DEFAULT_USER
my_password = DEFAULT_PASSWORD

def my_client_auth(realm):
    """
    An example client authorization function

    Returns the previously set username and password to the remote server
    to use as it's authorization, *if* the specified realm is correct

    realm : The realm specified by the remote server. Note that this is
    intended for future expansion. Currently it is always 'SNAPcom'.
    """
    return (my_user, my_password)

def my_server_auth(realm, username):
    """
    An example server authentication function

    Returns the password for the specified username in the realm

    realm : This server's realm (currently unused, future expansion)
    username : The username specified by the remote server
    """
    if username == my_user:
        return my_password
```

Here you can see some variables and routines defined such that the default SNAP Authentication Scheme *can* be overridden. Notice that unless an alternate username and/or password is provided (either by changing the source code or using command line arguments), the resulting behavior is the same.

```
def echo(obj):
    """Just a function that can be called remotely"""
    print str(obj)
```

Here is a classic example of a test function. Once you launch simpleSnapRouter.py, how do you know it is really listening for incoming RPC calls? By invoking the “echo()” function from some other SNAP Node, you can tell from the console output that the RPC exchange took place.

```
def main():
    global comm
    global my_user, my_password
```

Here begins the “`main()`” of this program. When you want to change a global variable from within the scope of a local routine, you have to tell Python that is what you intend, by means of global statements.

```
    usage = "usage: %prog [options] [other-node-IP-addresses]"
    parser = OptionParser(usage)
```

Here we have created an “Option Parser” and specified what the first line of the “—help” output will be.

```
    parser.add_option("-d", "--deaf", action="store_true", dest="no_listener", default=False,
help="DON'T listen for incoming TCP/IP connections (default is DO listen)")

    parser.add_option("-c", "--com", dest="comport", help="open a connection on the specified COM
port (0 = COM1)")
    parser.add_option("-s", "--ss200", dest="ss200", help="open a connection on the specified
SS200 device (0 = SNAPstick0)")
    parser.add_option("-l", "--legacy", dest="usb", help="open a connection on the specified USB
device (0 = USB0)")

    parser.add_option("-u", "--user", dest="user", default=DEFAULT_USER, help='specify an
alternative SNAP Realm user name(default is "public")')
    parser.add_option("-p", "--password", dest="password", default=DEFAULT_PASSWORD,
help='specify an alternative SNAP Realm password (default is "public")')
```

Here we have told the OptionParser what each of the allowable command line options is. Note that the code has not actually *parsed* anything yet.

```
    (options, args) = parser.parse_args()

    print options
    print args
```

The call to parser.parse_args() actually does the command line parsing, based on the previously specified rules. Note that the two print statements are just left over debugging code. It is not necessary to show the command line arguments in order to act on them.

```
    funcdir = {"echo": echo}
    comm = snap.Snap(funcs=funcdir)
```

If you have read through the other examples in this manual, this should be pretty standard stuff. We are stating that the "`echo()`" function is allowed to be externally callable, and we are creating a SNAP instance so that can happen.

```
    my_user = options.user
    my_password = options.password
```

Here we are just transferring the command line options into the corresponding global variables.

```
    if options.no_listener:
        print "Listening for incoming TCP/IP connections has been disallowed by user"
    else:
        print "Listening for incoming TCP/IP connections"
        comm.accept_tcp(auth_info=my_server_auth)
```

Unless the user has told us not to (using the --deaf command line option), start listening for incoming TCP/IP connections.

```
    if len(args) != 0:
        # Establish requested outbound TCP/IP connections
        for arg in args:
            print "Establishing a TCP/IP link to "+arg
            comm.connect_tcp(arg, auth_info=my_client_auth)
```

If any IP addresses were specified on the command line, try and connect to other *SNAP Connect* instances running at those IP addresses.

```
    # Did the user ask us to open a normal serial port connection (COM1-COMn)?
    if options.comport != None:
        port = int(options.comport)
        print "Opening a serial connection on COM%d" % (port+1)
        try:
            comm.open_serial(snap.SERIAL_TYPE_RS232, port)
        except Exception, e:
            print "Invalid COM port specified"

    # Did the user ask us to open a connection to a SS200?
    if options.ss200 != None:
        port = int(options.ss200)
        print "Opening a serial connection to SNAPstick%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK200, port)
        except Exception, e:
            print "Invalid SS200 device specified"

    # Did the user ask us to open a connection to a legacy Synapse USB device?
    if options.usb != None:
        port = int(options.usb)
        print "Opening a serial connection on USB%d" % (port)
        try:
            comm.open_serial(snap.SERIAL_TYPE_SNAPSTICK100, port)
        except Exception, e:
            print "Invalid USB port specified"
```

Notice that each of the above serial connections had to specify the correct type of connection. Also notice the use of a try/except block to allow the program to continue execution, regardless of the outcome of any particular `open_serial()` call.

```
    comm.loop()
```

As pointed out in numerous places throughout this manual, most of the *SNAP Connect* functionality will not take place unless you tell the library code to actually run. Here we are handing over control of the program to the library by using the `loop()` function. If we needed to do other ongoing processing, we would have combined that processing with a call to `comm.poll()` inside of a “while 1” loop.

```
if __name__ == '__main__':
    logging.basicConfig(filename='simpleSnapRouter.log', level=logging.DEBUG, format='%(asctime)s
%(levelname)s: %(message)s', datefmt='%Y-%m-%d %H:%M:%S')
    main()
```

Nothing here that has not been shown in the other examples… a logging file is created, and then `main()` is executed.

Additional Examples

Throughout this manual various “tutorial” examples have been presented, including detailed walk-throughs of how the source code worked.

Additional examples of SNAP Connect usage are provided in file “examples.zip”.

Time and space constraints prevent providing a detailed code walk-thru of every example application. Each example is in its own subdirectory, and each one comes with a README.txt file that you should refer to.

The following example programs are provided as-is. You are welcome to use them as starting points for your own custom applications, but many of them incorporate third party components or services not under Synapse control. As such, we are unable to provide individual support for them. If you are unable to use these programs as a starting point for your application, you will need to contact our Custom Solutions Group to obtain help and guidance on a contract basis.

These examples are listed here roughly in increasing order of complexity.

DisplayInfo – Sanity Check and Version Numbers Display

This example was almost considered “too simple to include”, *but* we have had repeated requests from users for exactly this sort of thing, so here it is. At a minimum, you can use this example to confirm that *SNAP Connect* is correctly installed on your system, since it requires **no** bridge hardware or serial ports of any kind.

What It Does

Example program DisplayInfo.py just creates a *SNAP Connect* instance and displays the Python version number, the *SNAP Connect* version number, the SNAP Address, and the current encryption setting.

How To Run This Example

python DisplayInfo.py

```
Python version is 2.5.4 (r254:67916, Dec 23 2008, 15:10:54) [MSC v.1310 32 bit (Intel)]
Permanent license created on 2012-02-14 14:14:45.343000 for 000020
SNAP Connect version number 3.1.0
My SNAP Address is 00.00.20
Encryption is set to None
```

Example Files

The following files can be found in the DisplayInfo directory.

README.txt – describes the example in more detail

DisplayInfo.py – the sole source file.

EchoTest – Simple Benchmark

Refer to this example if you want to see how to maximize throughput of your SNAP network.

It demonstrates use of the HOOK_RPC_SENT event, and shows one way to implement receive timeouts.

What It Does

EchoTest.py polls a predefined node a predefined number of times. It counts the number of responses received, times the entire test, and displays the results when finished.

```
    10 queries, 10 responses in 812 milliseconds
```

How To Run This Example

python EchoTest.py

There are no command line arguments - to change the characteristics of the test (for example, *what* node to poll), you must edit the source code.

For an example of command line argument processing refer to simpleSnapRouter.py, covered earlier in this manual.

Example Files

The following files can be found in the EchoTest directory.

README.txt – describes the example in more detail

EchoTest.py – the sole source file to this example

SPY File Upload

The source code form of a SNAPpy script is a PY (".py") file. The compiled (binary) form of a SNAPpy script is a SPY (".spy") file. Just like Portal, *SNAP Connect* applications can upload a replacement SPY file into an embedded SNAP Node.

What It Does

SpyUploader.py uploads a SPY file into a SNAP Node.

This example code serves two purposes:

1) It demonstrates how a *SNAP Connect* application can upload a new .SPY file into an embedded SNAP Node.

2) It provides an easy to use API that "wraps" that same functionality, so that you can simply re-use the example code in your own application.

How To Run This Example

python SpyUploader.py

SpyUploader will upload (send) the pre-specified SPY file to the pre-specified SNAP Node.

There are no command line arguments - to change the characteristics of the demo (for example, *what node* to upload, or *what SPY file* to send it), you must edit the source code.

Example Files

The following files can be found in the SpyUpload directory.

README.txt – describes the example in more detail

SpyUploader.py – the sole source file.

Note that this is both a stand-alone demo, as well as an importable library that you can re-use in your own SNAP Connect applications.

shortBlinkRF100_2.4.9.spy – just an example SPY file to upload

As you might guess from the name, this file was compiled for a Synapse RF100 module, running firmware version 2.4.9.

If you have different hardware or a different version of SNAP firmware, just use Portal to create a different SPY file.

For more information on creating SPY files (and on Portal in general) please refer to the **Portal User Guide**.

TcpRawLink – Extending SNAP Data Mode over raw sockets

Embedded SNAP Nodes can exchange DATA MODE (AKA TRANSPARENT MODE) packets over radio and serial links, and with the help of one or more SNAP Connect nodes, their reach can even extend over the Internet.

But what if you have a non-SNAP TCP/IP application, such as MODBUS TCP?

What It Does

This example implements a "TCP server" so that non-SNAP devices that have TCP/IP "socket" access can send and receive SNAP TRANSPARENT DATA (DATA MODE) packets over SNAP networks.

How To Run This Example

```
python tcpRawLink.py
```

This program does not take any command line parameters. It creates a "listener" on port 3000 that you can then establish socket connections to.

Once connected, any data you send over that socket will be multicast over the SNAP network. Any DATA MODE packets received from that same SNAP network will be sent over that same socket connection to your application.

For a more detailed explanation, refer to the README.txt file.

Example Files

The following files can be found in the TcpRawLink directory.

README.txt – describes the example in more detail

tcpRawLink.py – the actual application source code - implements a TCP server on port 3000.

e10Bridge.py – This is a SNAPpy script that you can upload into your E10 Bridge node if you want the LED on the front of the E10 to "blink" with radio traffic (optional).

You could reuse this technique with other applications - this script does not care what the *source* of the radio traffic is.

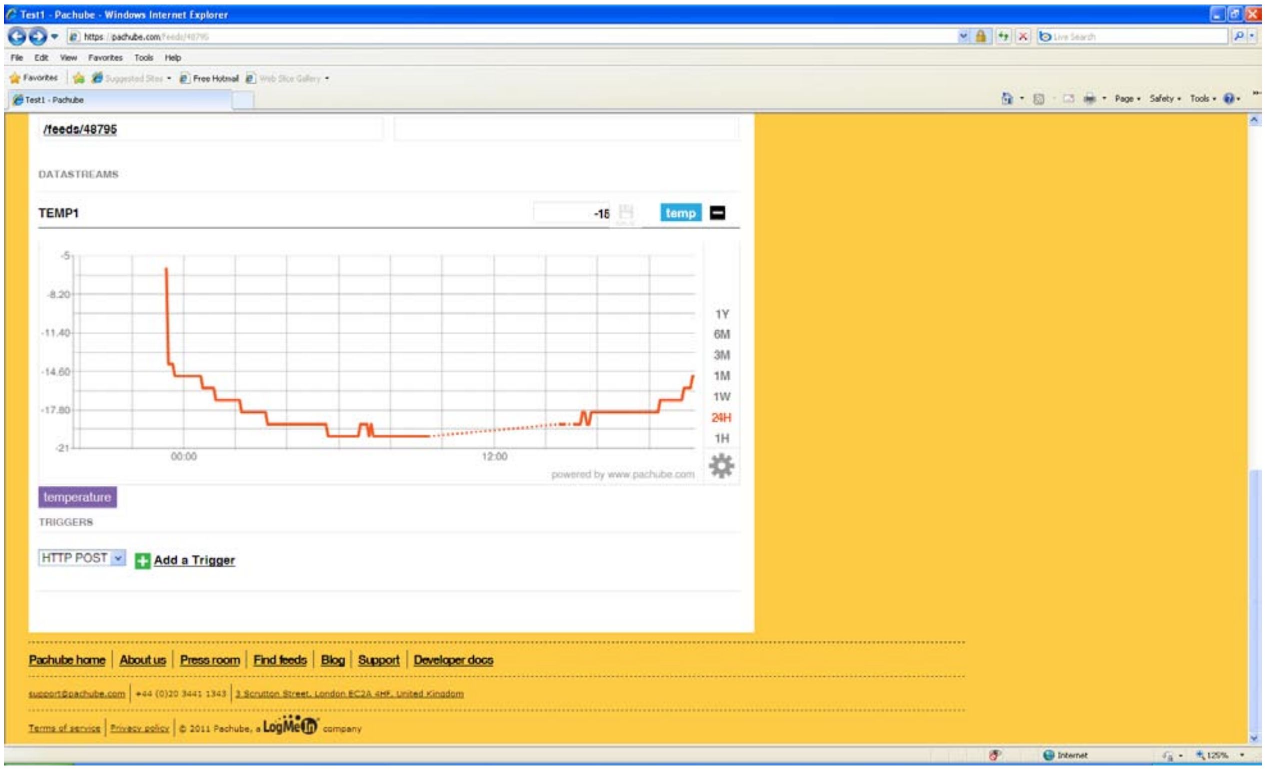
Extra Example Files

S999snap – this is meant for the /etc/init.d directory on a SNAP Connect E10. It replaces the existing file, and launches the tcpRawLink demo instead of the default E10 demo (UserMain.py).

remote_deamon.py – Linux systems can use this to run tcpRawLink.py as a background process (deamon). This file is used by S999snap.

Pachube – Pushing data to a web server

Pachube is a web service that accepts incoming data streams and makes them available via a web browser.


Test1 - Pachube - Windows Internet Explorer
/feeds/48795
DATASTREAMS
TEMP1
-15
temp
temperature
TRIGGERS
HTTP POST
Add a Trigger
1Y
6M
3M
1M
1W
24H
1H
00:00
12:00
powered by www.pachube.com
Pachube home | About us | Press room | Find feeds | Blog | Support | Developer docs

NOTE – Synapse Wireless and Pachube are not affiliated with one another.

What It Does

This example SNAP Connect application receives multicast data reports from SNAP nodes, and aggregates them into periodic reports that it then forwards to the Pachube server located at:

https://pachube.com.

How To Run This Example

python pachSnapE10.py

There are no command line arguments - to change the characteristics of this example (for example, *what* Pachube datastream to update), you must edit the source code.

For an example of command line argument processing refer to simpleSnapRouter.py, covered earlier in this manual.

Refer to the README.txt file for more details

Example Files

The following files can be found in the pachube directory.

README.txt – describes the example in more detail

pachSnapE10.py – the main source file to this demo.

Although as-written it expects to be run on a Synapse SNAP Connect E10, you can easily modify it to run on a Windows PC or Linux system.

HINT – change the serial port

pachUpdate.py – this import file provides the code to actually send data to the pachube server.

snapConversions.py – This file is intended to contain conversion routines from “raw” sensor data to “cooked” results. As-written it contains code to convert from raw ADC counts to resistance readings, which it then converts to temperature readings with some help from thermistor.py.

thermistor.py – this import file is used by snapConversions.py. It contains code to convert resistance readings to temperature readings.

pachube_Test1.py – This is **not** a SNAP Connect source file. This is a SNAPpy script, and as-written is meant to be loaded onto a Synapse SN111 End Device Demonstration Board.

It takes an analog sensor reading every second, and broadcasts it to any listening devices (like pachSnapE10.py).

For more information about SNAPpy scripts and embedded SNAP Nodes, please refer to the SNAP Reference Manual.

Extra Example Files

S999snap – this is meant for the /etc/init.d directory on a SNAP Connect E10. It replaces the existing file, and launches the pachube demo instead of the default E10 demo (UserMain.py).

remote_deamon.py – Linux systems can use this to run pachSnapE10.py as a background process (deamon). This file is used by S999snap.

NOTE – the techniques demonstrated by remote_daemon.py have nothing to do with pachube, and could easily be applied to other Linux projects.

RobotArm – Example SNAP Connect Web Server

A common customer request is "How can I create an application that is controllable through my **Web Browser**?"

What It Does

This example application combines *SNAP Connect* with the Tornado web server library, resulting in a program (robotServer.py) that interacts with web browsers on one side (via Web Sockets), and interacts with SNAP devices on the other (via SNAP of course).

Commands made by the user in the web browser are sent from the web browser to robotServer.py, which converts them into SNAP RPC calls over the wireless network.

Remote data reported from the SNAP nodes is received by robotServer.py, which updates the web page.

How To Run This Example

To run the full demo, you will need a robot arm to control. Look in file RobotArmHardware.pdf for details on how to modify an off-the-shelf robot arm to be controlled by a Synapse SN171 Proto Board. You then need to load the supplied SNAPpy script "robotArm.py" onto that Proto Board.

If you don't have a robot arm, you can modify the script to blink LEDs or control something else of your choosing.

How To Run The Web Server Portion Of This Example

Edit file local_overrides.py and replace the line

```
serial_port = 'COM5'
```

with the <u>correct</u> serial_port for your computer. Then do

```
python robotServer.py
```

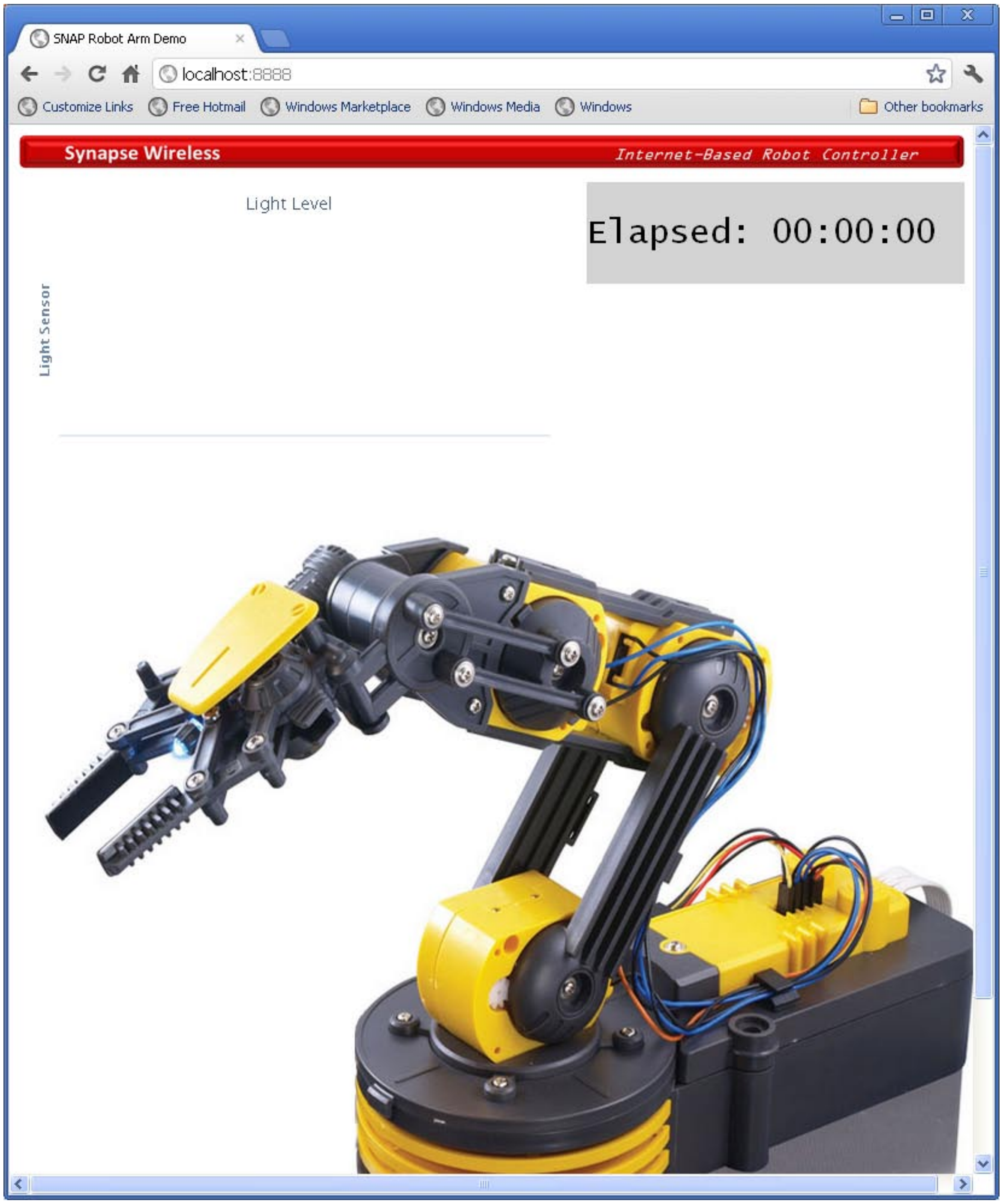
With robotServer.py running on your computer, you should be able to direct your web browser to http://localhost:8888 and see the "SNAP Robot Arm Demo" web page in your web browser.

NOTE – the demo requires a modern web browser that supports HTML5 Web Sockets.

If your browser is too old, you will get a message like:

Error

Your browser does not support HTML5 Web Sockets



If you move your mouse cursor over the image of the robot arm, you will see various graphical arrows appear. You can click and drag on these arrows to move the nearby portion of the arm.

The arm can be moved at the base (clockwise/counterclockwise), shoulder (up/down), elbow (up/down), and gripper (open/close).

You can also click to turn the light (located inside the gripper) on and off.

Example Files

The following files can be found in the RobotArm directory.

README.txt – describes the example in more detail

templates\index.html – defines the “structure” of the web page

static*.png – these files define all the images of the web page

static\main.js – this file implements the interactivity of the web page

static\HighCharts\*.* - these files implement the charting capability

static\jquery-1.6.2.min.js – the JQuery support library

static\ jquery-ui-1.8.16.custom.min.js – the JQuery UI library

These last three are third-party packages, and are used unmodified. For more information see README.txt.

The above files comprise the “web page” portion of the demo. This next file implements the actual web <u>server</u> and application.

robotServer.py – the actual web server

This program (robotServer.py) interacts with main.js over a Web Sockets interface. It interacts with the SNAP network (multicast RPC calls) over *SNAP Connect*.

tornado*.* - this is the Tornado web server library (www.tornadoweb.org).

This third-party library is used unmodified, and is included as a convenience.

Hints and Tips

The following are some "lessons learned" in creating the example applications for this User Manual.

Make sure you know the <u>correct</u> SNAP Address for your SNAP Connect Application

The range of *possible* SNAP Addresses for your application to use is determined by your SNAP License.

In the case of the *SNAP Connect* E10, the "hardware is your license." You can find the correct SNAP Addresses by looking at the sticker on the bottom of the unit.

When running *SNAP Connect* on a PC, the license is in the form of a license file.

- Default filename of 'License.dat' (although you can override this, see __init__()).
- Default location is "where your application resides" (in other words, in the current working directory for your program).

A single License.dat file can contain multiple addresses to choose from. (This is determined when you purchase the license).

When the *SNAP Connect* library reads the license file, it will output a message similar to:

```
Permanent license created on 2011-01-25 21:59:12.640000 for 4B425A
```

The address shown (4B425A in the example) is the one you will need to use *from other nodes* when making unicast RPC calls into your running *SNAP Connect* application.

For example, if your *SNAP Connect* application provides a `logTemperature()` function, and when it launched it displayed the "Permanent license" message shown above, then other nodes would use:

```
rpc('\x4B\x42\x5A', 'logTemperature', value)
```

If by mistake you do something like…

```
rpc('some-other-address-here', 'logTemperature', value)
```

…then it will <u>look</u> like your *SNAP Connect* application isn't working (no temperature readings will be getting logged) when in reality your other nodes just *aren't talking to it*.

Make sure each SNAP Connect application has a <u>unique</u> SNAP Address

If you want to run more than one *SNAP Connect* application at the same time, you will need to license multiple SNAP addresses. Attempting to assign the same address to more than one *SNAP Connect* application at a time will result in communications failures.

Multiple applications can share the same *multi-address* License.dat file (or multiple copies of the same License.dat file), but be sure to specify *which* of the licensed addresses from within that license file you want each application to use. If you don't specify an address, the library will use the *first* address from within the license file.

Make sure you know the <u>correct</u> serial port for your external devices

For example, it does no good to open a serial connection to COM1 when your external device is attached to COM2.

It's important to specify the correct *type* of serial port too – don't open a connection to a SERIAL_TYPE_RS232 when you have a SERIAL_TYPE_SNAPSTICK200 plugged in – but this mistake is less common.

Make sure that external bridge device really is available

For example, are you already connected to that device using Portal or some other application, such as a terminal program? (Or for advanced users, do you already have *another SNAP Connect* application connected to that device?)

Don't forget to call loop() or at least poll()

The library code does not act on queued up requests until you give it a chance to perform its background processing. You can call `poll()` repeatedly (intermingled with other processing), or you can just call `loop()` and turn over control to the library completely; but calling *neither* is a mistake.

Don't call poll() when you mean to call loop()

Function `poll()` only runs "one cycle" of the background code. It then returns control to the caller. If you mean for processing to continue, and you don't want to call `poll()` repeatedly (for example, from within your own "while loop"), then call `loop()` instead.

Adjust the logging levels to meet your needs

Many of the provided examples set the logging "level" (verbosity) to the maximum level. This provides the most information about what SNAP Connect is doing, but it also slows down performance. Once your application is working as you intend, you might consider changing the logging level. Refer to section String Constants used with Logging later in this manual.

Be very careful if you use Threads!

The *SNAP Connect* libraries are intended for single-thread usage only. You *can* use *SNAP* Connect from a multi-threaded application, but all of the accesses to the library (all API calls, etc.) **will have to be done from a single thread**.

Assuming the code “snippets” shown below are running in separate threads:

WRONG!

(snippet from thread 1)

```
snap_connect.rpc(nodeAddr, ‘foo’)
```

(snippet from thread 2)

```
snap_connect.rpc(nodeAddr, ‘bar’)
```

Code like the above will not work! The use of multiple threads will corrupt data internal to *SNAP Connect*, and interfere with correct serial port operation!

RIGHT!

(snippet from thread 1)

```
snap_connect .scheduler.schedule(0, snap_connect.rpc, nodeAddr, ‘foo’)
```

(snippet from thread2)

```
snap_connect .scheduler.schedule(0, snap_connect.rpc, nodeAddr, ‘bar’)
```

Code like the above will protect *SNAP Connect*’s internal data structures.

Don’t forget the other SNAP tools

Need to know what your *SNAP Connect* application(s) and other SNAP Nodes are saying to each other? Fire up a SNAP Sniffer!

Need a way to invoke commands manually (on demand)? Use the Portal command line! Running Portal with its own bridge node can provide a window into your complete network.

SNAP Connect API Reference – The Functions

Using *SNAP Connect* in your Python application is designed to be as easy as writing a SNAPpy script on your SNAP nodes. As you look through the API you will notice how similar the API for a SNAPpy script is compared to using *SNAP Connect*. The API was designed to follow the SNAPpy API as closely as possible, only deviating where it was absolutely necessary. Some of the API function names are formatted differently in *SNAP Connect* than in SNAPpy to follow Python's recommended style guidelines. There are, however, alias functions, shown below after a "|" character, that are formatted the same as SNAPpy's API for convenience. So, you can use either style in your *SNAP Connect* scripts.

__init__

`__init__(self, license_file=None, nvparams_file=None, funcs=None, scheduler=None, addr=None, rpc_handler=None)`

Initialize a *SNAP Connect* instance.

`license_file` : The full path to a *SNAP Connect* license file (default License.dat)

`nvparams_file` : The full path to a *SNAP Connect* NV parameters file (default nvparams.dat)

`funcs` : The callable functions for this instance to expose to the SNAP network (default None)

`scheduler` : Internally used by *SNAP Connect*

`addr` : The SNAP address to use from the license file. If nothing is specified, *SNAP Connect* uses the first address in the license file. The format of this parameter is a three-character byte string, e.g., '\x86\xa2\x5c' represents SNAP Address 86.A2.5C.

`rpc_handler` : Internally used by *SNAP Connect*

This function returns the instance of the object that has been created.

This function might raise the following exceptions:

RuntimeError("Non-licensed address provided") – For example, the License.dat file is for address 11.22.33 but you have asked to "be" address 22.44.66.

RuntimeError("Invalid license found") – The license file specified is invalid. If you don't specify a license file name, the default of "License.dat" is assumed.

IOError("Unable to load NV params file: <filename>") – the expected NV parameters file could not be loaded. If you specify a file, it is required to be present and valid. If you do not specify a file, then the default of "nvparams.dat" is assumed.

RuntimeError("Unable to determine callable functions") – you must provide a dictionary of callable functions, even if it is an empty one.

ImportError("Unable to find PyCrypto library required for AES support") – you have enabled AES-128 encryption, but you have not provided the required PyCrypto library (refer back to the Installation section of this manual).

ValueError("Unknown encryption type specified: <encryption type>" – valid choices are NONE, AES128, and BASIC.

accept_tcp

`accept_tcp(self, auth_info=server_auth, ip='', port=snaptcp.SNAP_IP_PORT, tcp_keepalives=False, forward_groups=None)`

Start listening for and accepting remote IP connections.

`auth_info` : The function to call when authenticating remote credentials (defaults to public credentials). If providing a custom function to call, it should have the signature "`server_auth(realm, username)`" where the two string arguments are supplied by the connecting instance and the function returns the appropriate password.

`ip` : The IP address to listen on for remote connections (default all addresses)

`port` : The IP port number to listen on for remote connections (default 48625)

`tcp_keepalives` : Enable or disable TCP keepalives (default False)

`forward_groups` : Multi-cast groups to forward onward through this interface; defaults to using the value specified in NV Parameter 6

This function returns None.

The forward_groups parameter can be important if you have a radio network that uses multicast traffic, but you do not want those packets to propagate over the Internet.

This function might raise the following exceptions:

ValueError("auth_info is not callable") – Parameter auth_info can be unspecified (in which case the default server_auth() routine is used), but if you provide a value for this parameter it has to in fact be a function that *SNAP Connect* can invoke.

Establishment of an actual connection can trigger a HOOK_SNAPCOM_OPENED event.

If the connection later goes down it can trigger a HOOK_SNAPCOM_CLOSED event.

See also functions connect_tcp(), disconnect_tcp(), and stop_accepting_tcp().

add_rpc_func

`add_rpc_func(self, rpc_func_name, rpc_func)`

Adds a function to the existing "RPC dictionary".

`rpc_func_name` : This is the name the new function will be callable by via RPC. It does not have to match the actual function's name.

`rpc_func` : This must be a Python callable, and represents the actual function to be invoked.

When the *SNAP Connect* instance is first instantiated by your application code, you pass to it a Python dictionary containing all of the function names that you want to be able to invoke from other SNAP nodes.

This function lets you add *additional* functions to that dictionary after-the-fact.

This function returns True if the function was successfully added to the RPC dictionary. It returns False if the function could not be added because one with the same name already exists in the dictionary (you can use this function to **add** a new function but you cannot use it to **replace** an existing function).

close_serial

`close_serial(self, serial_type, port)`
Close the specified serial port if open.

`serial_type` : The type of serial interface to close (RS-232=1, USB=2)

`port` : The port **of that particular type** to close, as appropriate for your operating system. On Windows, port is a zero-based list. (Specify 0 for COM1 for example.). On Linux, port will typically be a string, for example "/dev/ttys1".

This function returns None.

This function can trigger a HOOK_SERIAL_CLOSE event.

connect_tcp

```
connect_tcp(self, host, auth_info=client_auth, port=None, retry_timeout=60,
secure=False, forward_groups=None, tcp_keepalives=False, cache_dns_lookup=False)
```
Connect to another SNAP node over IP.

`host` : The IP address or hostname of the other SNAP node

`auth_info` : A local function to call to retrieve the authorization information to use (defaults to public credentials). If providing a custom function to call, it should have the signature "`client_auth(realm)`" and return a tuple contain the username and password to authenticate with. The `realm` parameter will be a string provided by *SNAP Connect*.

`port` : The IP port number to connect to

`retry_timeout` : A timeout in seconds to wait before retrying to connect (default 60)

`secure` : A boolean value specifying whether SSL encryption is enabled for this connection (default False)

`forward_groups` : Multi-cast groups to forward onward through this interface; defaults to using the value specified in NV Parameter 6

`tcp_keepalives` : Enable TCP layer keepalives (default False)

`cache_dns_lookup` : Only perform a DNS lookup once for the host being connected to (default False). This option was added because some DNS servers are extremely slow.

The forward_groups parameter can be important if you have a radio network that uses multicast traffic, but you do not want those packets to propagate over the Internet.

NOTE – enabling tcp_keepalives can increase the amount of TCP/IP traffic your SNAP network generates. This might be an issue if you are using a cellular modem for your network connectivity.

This function returns None.

Establishment of the actual connection can trigger a HOOK_SNAPCOM_OPENED event.

If the connection later goes down it can trigger a HOOK_SNAPCOM_CLOSED event.

See also functions accept_tcp(), disconnect_tcp(), and stop_accepting_tcp().

data_mode

`data_mode(self, dst_addr, data)`

Sends a transparent (aka data) mode packet to the specified SNAP network address.

`dst_addr` : The SNAP network address of the remote node
`data` : The data to send

This function returns an identifier for the packet sent. This identifier can later be used in conjunction with the HOOK_RPC_SENT handler.

This function can result in a HOOK_STDIN event at the node specified by dst_addr.

See also mcast_data_mode().

disconnect_tcp

`disconnect_tcp(self, host, port=snaptcp.SNAP_IP_PORT, all=False, retry=False)`

Disconnect from the specified instance.

`host` : The IP address or hostname of the other instance
`port` : The IP port number (default 48625)
`all` : Disconnect all connections matching the criteria (default False)
`retry` : Disconnect, but then retry connecting to the same host and port (default False)

This function returns True if the specified connection was found and closed, otherwise False.

This function can result in one or more HOOK_SNAP_CLOSED events being generated.

See also functions accept_tcp(), connect_tcp(), and stop_accepting_tcp().

get_info | getInfo

`get_info(which_info)`

Get the specified system information.

`which_info` : Specifies the type of information to be retrieved (0-15 but with some gaps – not all of the “info” types from the original (embedded) SNAP Nodes apply to a PC-based application like *SNAP Connect*). For more information on the get_info() function, refer to the **SNAP Reference Manual**. This function returns the requested information, or None if parameter which_info is invalid.

The possible values for which_info, and their meanings/return values are:

which_info	**Meaning**	**Return Value**
0	Vendor	Always returns 0 indicating “Synapse”
1	Type of radio	Always returns 1 indicating “no radio”
2	Type of CPU	Always returns 8 indicating “unknown”
3	Hardware platform	Always returns 7 indicating “*SNAP Connect*”
4	Build Type (debug or release)	Returns 0 if running under a debugger, Returns 1 if running standalone
5	Software MAJOR version	Example: if version is 1.2.3 get_info(5) returns 1
6	Software MINOR version	Example: if version is 1.2.3 get_info(6) returns 2
7	Software BUILD version	Example: if version is 1.2.3 get_info(7) returns 3
8	Encryption capability	Returns 1 if AES-128 is available (not necessarily enabled, just available for use) Otherwise returns 2, indicating that only “SNAP Basic” encryption is available
9	SNAP Sequence Number of most recently enqueued packet	For SNAP Connect applications you should use the value returned by the rpc() and mcast_rpc() functions. This enumeration is only implemented to loosely match the embedded nodes
10	Multicast flag	Returns 1 if the RPC currently being processed came in via multicast, returns 0 if the packet came in via unicast
11	TTL Remaining	Returns the TTL (“the “hops remaining”) field of the packet currently being processed. For this to be of any use, you would have to know how many hops were originally specified
12	Tiny strings remaining	N/A to *SNAP Connect*, always returns None
13	Medium strings remaining	N/A to *SNAP Connect*, always returns None
14	Size (capacity) of Route Table	N/A to *SNAP Connect*, always returns None
15	Routes stored in Route Table	Returns the number of active routes

load_nv_param | loadNvParam

`load_nv_param(self, nv_param_id)`

Return the indexed parameter from storage.

`nv_param_id` : Specifies which "key" to retrieve from storage. Some NV parameters may have no effect on the *SNAP Connect* instance.

For more details about the individual NV Parameters, refer to section **SNAP Connect API Reference – NV Parameters** later in this document.

This function returns the requested NV Parameter. Note that on some platforms the MAC Address parameter is set by the hardware (for example, on a *SNAP Connect E10*). In such cases, a request for that parameter will return the “hardware” value rather than any artificially stored value.

See also function save_nv_param().

local_addr | localAddr

`local_addr(self)`

This function returns the three byte binary string representing the address of the *SNAP Connect* instance. For example, a *SNAP Connect* instance running at the default address of 00.00.20 would return a Python string containing “\x00\x00\x20”.

loop

`loop(self)`

This function does not return. You should only call it if your *SNAP Connect* application is “purely reactive” – for example, an application that only responds to RPC calls from other nodes.

If your application needs to take action on its own behalf (for example, if it needs to be polling other nodes based), then you probably should be using the poll() function instead.

See also function poll() and function poll_internals().

mcast_data_mode

`mcast_data_mode(self, group, ttl, data)`

Sends a multicast transparent (aka data) mode packet.

`group` : specifies which nodes should respond to the request

`ttl` : specifies the Time To Live (TTL) for the request

`data` : The data to send

This function returns an identifier for the packet sent. This identifier can later be used in conjunction with the HOOK_RPC_SENT handler.

This function can result in a HOOK_STDIN event in one or more <u>other</u> nodes.

See also function data_mode().

mcast_rpc | mcastRpc

`mcast_rpc(self, group, ttl, func_name, *args)`

Makes a Remote Procedure Call, or RPC, using multicast messaging. This means the message could be acted upon by multiple nodes.

`group` : specifies which nodes should respond to the request

`ttl` : specifies the Time To Live (TTL) for the request

`func_name` : The function name to be invoked

`args` : Any arguments for the function specified by `func_name`. Note that this should be given individually (separated by commas), not bundled into a tuple. For example, if foo() is a function taking two parameters, use something like

```
mcast_rpc(1, 2, 'foo', 1, 2) # <- correct!
```

instead of using something like

```
mcast_rpc(1, 2, 'foo', (1,2)) # <- wrong!
```

That this is different from the way multiple parameters were handled in the original XML-RPC based *SNAP Connect*, which is why we mention it.

This function returns an identifier for the packet sent. This identifier can later be used in conjunction with the HOOK_RPC_SENT handler.

This function can trigger a HOOK_RPC_SENT event.

See also function rpc().

open_serial

```
open_serial(self, serial_type, port, reconnect=False, dll_path=MODULE_PATH,
forward_groups=None)
```

Open the specified serial port for sending and receiving SNAP packets.

`serial_type` : The type of serial interface to open. Available options are:

- SERIAL_TYPE_SNAPSTICK100: The original USB SNAP Stick SNAP Engine carrier
- SERIAL_TYPE_SNAPSTICK200: The SNAP Stick 200
- SERIAL_TYPE_RS232: An RS-232 COM port

`port` : The port **of that particular type** to open, as appropriate for your operating system. On Windows, port is a zero-based list. (Specify 0 for COM1 for example.). On Linux, port will typically be a string, for example "/dev/ttys1".

`reconnect` : Close the connection and re-open (default False)

`dll_path` : Path to where the serial port driver exists on your system `forward_groups` : Multi-cast groups to forward onward through this interface; defaults to using the value specified in NV Parameter 6

This function normally returns False. If you specified reconnect=True and the reconnect was successful, then True will be returned.

This function can raise the following exceptions:

ValueError("Serial interface type must be an integer")

ValueError("Unsupported serial interface type <type>")

poll

`poll(self)`

Performs one cycle of background operations, including network (asyncore) and scheduling processing, then returns. By calling this function repeatedly, you can keep *SNAP Connect* communications going while still performing other processing (for example, maintaining a GUI).

If your *SNAP Connect* application has processing of its own to do (it is active rather than reactive), then function `poll()` or function `poll_internals()` is the function to use.

If your application will only be responding to incoming RPC calls from other nodes, then you might consider using the convenience function `loop()` (which just calls `poll()` repeatedly).

This function returns None.

NOTE – this function performs not only the *SNAP Connect*-specific processing, but also invokes asyncore.poll() and the scheduler.poll() function. For many stand-alone *SNAP Connect* applications, this is optimal.

However, some applications (such as a web server that has *SNAP Connect* embedded inside it) may need to maintain control of asyncore and scheduling. In those situations, function poll() will be trying to “do too much”, and you should use function poll_internals() instead.

See also function loop() and poll_internals().

poll_internals

`poll_internals(self)`

Performs one cycle of background operations, *SNAP Connect*-specific processing only (no asyncore or scheduling), then returns. By calling this function repeatedly, you can keep *SNAP Connect* communications going while still performing other processing (for example, maintaining a GUI).

If your *SNAP Connect* application has processing of its own to do (it is active rather than reactive), then `poll()` is the function to use. If that additional processing involves asyncore and scheduling, you may need to use function poll_internals() instead of function poll().

For an example of a web server application that demonstrates the use of poll_internals() instead of function poll(), refer to the “Robot Arm” example application.

This function returns None.

rpc

`rpc(self, dst_addr, func_name, *args)`

Sends a unicast RPC command.

Call a remote procedure (make a Remote Procedure Call, or RPC), using unicast (directly addressed) messaging. A packet will be sent to the node specified by the `dst_addr` parameter, asking that remote node to execute the function specified by the `func_name` parameter. The specified function will be invoked with the parameters specified by args, if any args are present.

`dst_addr` : The SNAP network address of the remote node

`func_name` : The function name to be invoked on the remote node

`args` : Any arguments for the function specified by `func_name`. Note that this should be given individually (separated by commas), not bundled into a tuple. For example, if foo() is a function taking two parameters, use something like

```
rpc('\x12\x34\x56', 'foo', 1, 2) # <- correct!
```

instead of using something like

```
rpc('\x12\x34\x56', 'foo', (1,2)) # <- wrong!
```

That this is different from the way multiple parameters were handled in the original XML-RPC based *SNAP Connect*, which is why we mention it.

This function returns an identifier for the packet sent. This identifier can later be used in conjunction with the HOOK_RPC_SENT handler.

This function can trigger a HOOK_RPC_SENT event.

See also mcast_rpc().

rpc_source_addr | *rpcSourceAddr*

`rpc_source_addr(self)`

Originating address of the current RPC context (or None if called outside RPC).

This function returns the SNAP network address of the remote node that initiated the RPC call.

If no RPC call is currently in progress, then this function returns None.

See also function rpc_source_interface().

rpc_source_interface

`rpc_source_interface(self)`

Originating interface of the current RPC context (or None if called outside RPC).

This function returns an Interface object representing the interface type that the incoming RPC call was received on. The Interface object can be queried for its type, if necessary.

That type is contained in the intf_type field of the Interface object, and will be one of the following values:

INTF_TYPE_UNKNOWN = 0

INTF_TYPE_802154 = 1

INTF_TYPE_SERIAL = 2

INTF_TYPE_SILABS_USB = 3

INTF_TYPE_ETH = 4

INTF_TYPE_PROXY = 5

INTF_TYPE_SNAPSTICK = 6

If no RPC call is currently in progress, then this function returns None.

See also function rpc_source_addr().

save_nv_param | saveNvParam

`save_nv_param(self, nv_param_id, obj)`

Store individual objects for later access by id.

`nv_param_id` : Specifies which "key" to store the obj parameter under. Integers 0-127 are pre-assigned system IDs. Integers 128-254 are available for user-defined purposes on all SNAP nodes, but *SNAP Connect* instances allow you to use any value as a key.

`obj` : The object you would like to store for later use. Other SNAP nodes are capable of storing strings, integers, and Boolean values; but *SNAP Connect* instances can store any object that Python is capable of pickling. Refer to the Python documentation for details about the pickle functions.

This function returns True if the save operation was successful, otherwise it returns False.

See also function load_nv_param().

set_hook | setHook

`set_hook(self, hook, callback=None)`

Set the specified SNAP hook to call the provided function.

`hook` : The SNAP event hook identifier. Available identifiers are:

HOOK NAME	**EVENT**
hooks.HOOK_SNAPCOM_OPENED	Network communications established
hooks.HOOK_SNAPCOM_CLOSED	Network communications shut down
hooks.HOOK_SERIAL_CLOSE	Serial port closed
hooks.HOOK_RPC_SENT	RPC sent (or unicast retries exhausted)
hooks.HOOK_STDIN	Data received from another SNAP node
hooks.HOOK_TRACEROUTE	Trace Route results have been received

`callback` : The function to invoke when the specified event occurs

The required signature for a given callback handler depends on the particular hook. Refer to section *SNAP Connect* API Reference – HOOKS towards the end of this document.

This function returns None.

This function can raise the following exceptions:

TypeError("Unknown hook type") – you can only specify hook values from the above list.

TypeError("Invalid callback") – the object you specify to be the handler for the specified hook must in fact be a callable function.

stop_accepting_tcp

`stop_accepting_tcp(self, close_existing=False)`

This function cancels any previous `accept_tcp()` calls.

`close_existing` : In addition to not accepting any new incoming connections, this parameter tells SNAP Connect to automatically shutdown any connections that have already been established (default False).

If close_existing = True, this function can result in one or more HOOK_SNAP_CLOSED events being generated.

This function returns None.

See also functions accept_tcp(), connect_tcp(), and disconnect_tcp().

traceroute

`traceroute(self, dst_addr)`

Sends a traceroute request.

`dst_addr` : The SNAP network address of the remote node to perform a Trace Route to.

A Trace Route determines/reports a current route to the node that has the network address specified as specified `dst_addr`. This does not mean that the node cannot also be reached through other paths.

This function returns None. To actually get the "trace route report" you will need to use `set_hook(snap.hooks.HOOK_TRACEROUTE, …)` to specify a function that will receive and process the trace route list.

This function can result in a HOOK_TRACEROUTE event.

Refer to section HOOK_TRACEROUTE – Trace Route data received for more details.

SNAP Connect API Reference – Constants and Enumerations

Numerous constants are defined by the *SNAP Connect* libraries for readability. In this section we list and describe all of them.

Most of these constants are defined by the snap module, and so need a “snap.” prefix on them. For readability within this document, the leading “snap.” prefix is not usually shown, however within your code you will need to specify it. For example:

```
save_nv_param( snap.NV_AES128_ENABLE_ID, snap.ENCRYPTION_TYPE_NONE )
```

Some constants have an additional prefix, for example the various “HOOK_xxx” constants have been moved into a “hooks” group. Those prefixes will be shown in this section, but the leading “snap.” portion must be added too. For example:

```
set_hook(snap.hooks.HOOK_TRACEROUTE, trace_route_handler)
```

Constants used with encryption

ENCRYPTION_TYPE_NONE – used to turn off encryption.

ENCRYPTION_TYPE_AES128 – used to enable AES-128 encryption.

ENCRYPTION_TYPE_BASIC – used to enable basic SNAP encryption.

Constants used with close_serial(), open_serial(), and HOOK_SERIAL_CLOSE

You will use the following constants as the serial_type parameter to the close_serial(), open_serial(), and HOOK_SERIAL_CLOSE handler functions.

SERIAL_TYPE_SNAPSTICK100 – The SN132 SNAPstick, sometimes referred to as a “paddle board”. These are easily recognized, since they have no case (cover), and you can swap out the SNAP Engine on it for a different model. These have to be plugged into a USB port.

SERIAL_TYPE_SNAPSTICK200 – The SS200 SNAP Stick, which is much smaller than the SN132, does have a plastic case, and does not accept plug-in SNAP Engines (it is completely self-contained). These have to be plugged into a USB port.

SERIAL_TYPE_RS232 – “true” COM port, or a USB-serial cable.

Constants used with set_hook()

You will use the following constants as the hook parameter to the set_hook() function. They are described in more detail in the section *SNAP Connect* API Reference – HOOKS of this document, and so are merely listed here.

hooks. HOOK_SNAPCOM_OPENED

hooks. HOOK_SNAPCOM_CLOSED

hooks. HOOK_SERIAL_CLOSE

hooks. HOOK_RPC_SENT

hooks. HOOK_STDIN

hooks.HOOK_TRACEROUTE

Constants used with rpc_source_interface()

INTF_TYPE_UNKNOWN (you should never see this)

INTF_TYPE_802154 – for future use, *SNAP Connect* currently relies on a "bridge" node to provide the radio

INTF_TYPE_SERIAL – RPC call came in over RS-232

INTF_TYPE_SILABS_USB – RPC call came in over a Silicon Labs USB interface chip

INTF_TYPE_ETH – RPC call came in over TCP/IP

INTF_TYPE_SNAPSTICK – RPC call came in from a SN132 SNAPstick

Constants used with SPY Uploading

SNAPPY_PROGRESS_ERASE – previous script has been erased

SNAPPY_PROGRESS_UPLOAD – "chunk" of script accepted

SNAPPY_PROGRESS_COMPLETE – upload completed successfully

SNAPPY_PROGRESS_ERROR – upload failed

SNAPPY_PROGRESS_TIMEOUT – node failed to respond

SNAPPY_PROGRESS_WRITE_ERROR – FLASH write failure

SNAPPY_PROGRESS_WRITE_REFUSED – power too low to attempt

SNAPPY_PROGRESS_UNSUPPORTED – node does not support script upload

For example, it is a *SNAP Connect* instance, not an embedded node at all.

String Constants used with Logging

Python logging supports fine grained control of level (verbosity).

The levels that can be applied are DEBUG, INFO, WARNING, ERROR, and FATAL, where DEBUG is the most verbose, and FATAL is the least.

To change the log level globally, you would do something like:

```
log = logging.getLogger()
log.setLevel(logging.DEBUG)
```

To change the level on a per-module basis, you use the name of the module: "apy", "SerialWrapper", "snap", or "snaplib". For example:

```
snaplib_log = logging.getLogger('snaplib')
snaplib_log.setLevel(logging.ERROR)
```

Even finer grained control is possible, but you have to know the name (label) of the loggers you want to control. That is the purpose of this next list.

"SerialWrapper"

    "SerialWrapper.pySerialSocket"

"snap"

    "snap.AutoSaver"

“snap.Deferred”

“snap.PacketSink”

“snap.dispatchers”

“snap.listeners”

“snap.mesh”

“snap.SNAPtcpConnection”

“snap.SNAPtcpServer”

“snaplib”

“snaplib.ComUtils”

“snaplib.EventCallbacks”

“snaplib.PacketQueue”

“snaplib.PacketSerialProtocol”

“snaplib.PySerialDriver”

“snaplib.RpcCodec”

“snaplib.TraceRouteCodec”

“snaplib.ScriptsManager”

“snaplib.SerialConnectionManager”

“snaplib.SnappyUploader”

For example:

```
snaplib_log = logging.getLogger('snaplib.RpcCodec')
snaplib_log.setLevel(logging.INFO)
```

Constants used with load_nv_param() and save_nv_param()

These have been broken out into a separate section, which starts on the next page.

SNAP Connect API Reference – NV Parameters

Embedded SNAP Nodes keep configuration parameters in physical Non-Volatile (NV) memory.

SNAP Connect emulates this type of configuration repository using a standard Python **pickle** file named “nvparams.dat”.

The following non-volatile parameters are available through the save_nv_param() and load_nv_param() API functions.

NOTE – unlike in embedded SNAP nodes, *SNAP Connect* NV Parameter changes take effect immediately (no reboot required).

Here are all of the System (Reserved) NV Parameters (sorted by numeric ID) that apply to *SNAP Connect*, and what they do.

NOTE – Embedded SNAP Nodes use these same NV parameters, plus **many more** – refer to the **SNAP Reference Manual** for more information.

NOTE – You can also define your own NV Parameters (in the range 128-254) which your script can access and modify, just like the system NV Parameters.

ID 0-4 – Embedded SNAP only, not used by *SNAP Connect*

ID 5 – Multi-cast Processed Groups

Enumeration = snap.NV_GROUP_INTEREST_MASK_ID

This is a 16-bit field controlling which multi-cast groups the node will respond to. It is a bit mask, with each bit representing one of 16 possible multi-cast groups. For example, the 0x0001 bit represents the default group, or “broadcast group.”

One way to think of groups is as “logical sub-channels” or as “subnets.” By assigning different nodes to different groups, you can further subdivide your network.

For example, Portal could multi-cast a “sleep” command to group 0x0002, and ***only*** nodes *with that bit set* in their **Multi-cast Processed Groups** field would go to sleep. (This means nodes with their group values set to 0x0002, 0x0003, 0x0006, 0x0007, 0x000A, 0x000B, 0x000E, 0x000F, 0x0012, etc., would respond.) Note that a single node can belong to any (or even all) of the 16 groups.

Group membership does not affect how a node responds to a direct RPC call. It only affects multi-cast requests.

ID 6 – Multi-cast Forwarded Groups

Enumeration = snap. NV_GROUP_FORWARDING_MASK_ID

This is a separate 16-bit field controlling which multi-cast groups will be *re-transmitted* (forwarded) by the node. It is a bit mask, with each bit representing one of 16 possible multi-cast groups. For example, the 0x0001 bit represents the default group, or “broadcast group.”

By default, all nodes process and forward group 1 (*broadcast*) packets.

Please note that the Multi-cast Processed Groups and Multi-cast Forwarded Groups fields are independent of each other. A node could be configured to forward a group, process a group, or both. It can process groups it does not forward, or vice versa.

NOTE – If you set your bridge node to not forward multi-cast commands, any *Portal* or *SNAP Connect* attached to that bridge will not be able to multi-cast to the rest of your network.

ID 7-10 – Embedded SNAP only, not used by *SNAP Connect*

ID 11 – Feature Bits

Enumeration = snap.NV_FEATURE_BITS_ID

These control some miscellaneous hardware settings on embedded SNAP Nodes. The only Feature Bit that applies to a *SNAP Connect* instance is:

Bit 8 (0b0000,0001,0000,0000 0x0100) – Enable second RPC CRC

The second CRC bit (0x100) enables a second CRC packet integrity check on platforms that support it. Setting this bit tells the SNAP node to send a second cyclical redundancy check (using a different CRC algorithm) on each RPC or multicast packet, and require this second CRC on any such packet it receives. This reduces the available data payload by two bytes (to 106 bytes for an RPC message, or 109 bytes for a multicast message), but provides an additional level of protection against receiving (and potentially acting upon) a corrupted packet. The CRC that has always been a part of SNAP packets means that there is a one in 65,536 chance that a corrupted packet might get interpreted as valid. The second CRC should reduce this to a less than a one in four billion chance.

If you set this bit for the second CRC, you should set it in all nodes in your network, and enable the feature in your Portal preferences or as a feature bit in your *SNAP Connect* NV parameters.

A node that does not have this parameter set will be able to hear and act on messages from a node that does have it set, but will not be able to communicate back to that node. Not all platforms support this second CRC. Refer to each platform’s details to determine whether this capability is available.

Binary notations are provided here for clarity. You should specify the parameter value using the appropriate hexadecimal notation. For example, 0x001F corresponds to 0b0000,0000,0001,1111.

ID 12-18 – Embedded SNAP only, not used by *SNAP Connect*

ID 19 – Unicast Retries

Enumeration = snap.NV_SNAP_MAX_RETRIES_ID

This lets you control the number of unicast transmit attempts. This parameter defaults to 8.

This parameter refers to the total number of attempts that will be made to get an acknowledgement back on a unicast transmission to another node.

In some applications, there are time constraints on the “useful lifetime” of a packet. In other words, if the packet has not been successfully transferred by a certain point in time, it is no longer useful. In these situations, the extra retries are not helpful – the application will have already “given up” by the time the packet finally gets through.

By lowering this value from its default value of 8, you can tell SNAP to “give up” sooner. A value of 0 is treated the same as a value of 1 – a packet gets <u>at least one chance</u> to be delivered no matter what.

If your connection link quality is low and it is important that every packet get through, a higher value here may help. However it may be appropriate to reevaluate your network setup to determine if it would be better to change the number of nodes in your network to either add more nodes to the mesh to forward requests, or reduce the number of nodes broadcasting to cut down on packet collisions.

ID 20 – Mesh Routing Maximum Timeout

Enumeration = snap.NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID

This indicates the maximum time (in milliseconds) a route can “live.” This defaults to 0xEA60, or one minute.

Discovered mesh routes timeout after a configurable period of inactivity (see #23), but this timeout sets an upper limit on how long a route will be used, even if it is being used heavily. By forcing routes to be rediscovered periodically, the nodes will use the shortest routes possible.

Note that you *can* set this timeout to zero (which will disable it) if you know for certain that your nodes are stationary, or have some other reason for needing to avoid periodic route re-discovery.

You can use get_info(15) to determine the number of currently active (not timed out) routes.

ID 21 – Mesh Routing Minimum Timeout

Enumeration = snap.NV_MESH_ROUTE_AGE_MIN_TIMEOUT_ID

This is the minimum time (in milliseconds) a route will be kept. This defaults to 1000, or one second.

ID 22 – Mesh Routing New Timeout

Enumeration = snap.NV_MESH_ROUTE_NEW_TIMEOUT_ID

This is the grace period (in milliseconds) that a newly discovered route will be kept, even if it is never actually used. This defaults to 5000, or five seconds.

ID 23 – Mesh Routing Used Timeout

Enumeration = snap.NV_MESH_ROUTE_USED_TIMEOUT_ID

This is how many additional milliseconds of "life" a route gets whenever it is used. This defaults to 5000, or five seconds.

Every time a known route gets used, its timeout gets reset to this parameter. This prevents active routes from timing out as often, but allows inactive routes to go away sooner. See also Parameter #20, which takes precedence over this timeout.

ID 24 – Mesh Routing Delete Timeout

Enumeration = snap.NV_MESH_ROUTE_DELETE_TIMEOUT_ID

This timeout (in milliseconds) controls how long "expired" routes are kept around for bookkeeping purposes. This defaults to 10000, or 10 seconds.

ID 25 – Mesh Routing RREQ Retries

Enumeration = snap.NV_MESH_RREQ_TRIES_ID

This parameter controls the total number of retries that will be made when attempting to "discover" a route (a multi-hop path) over the mesh. This defaults to 3.

ID 26 – Mesh Routing RREQ Wait Time

Enumeration = snap.NV_MESH_RREQ_WAIT_TIME_ID

This parameter (in milliseconds) controls how long a node will wait for a response to a **Route Request (RREQ)** before trying again. This defaults to 500, or a half second.

Not that subsequent retries use longer and longer timeouts (the timeout is doubled each time). This allows nodes from further and further away time to respond to the RREQ packet.

ID 27 – Mesh Routing Initial Hop Limit

Enumeration = snap.NV_MESH_INITIAL_HOPLIMIT_ID

This parameter controls how far the initial “discovery broadcast” message is propagated across the mesh.

If your nodes are geographically distributed such that they are always more than 1 hop away from their logical peers, then you can increase this parameter. Consequently, if most of your nodes are within direct radio range of each other, having this parameter at the default setting of 1 will use less radio bandwidth.

If you set this parameter to zero, SNAP will make an initial attempt to talk directly to the destination node, on the assumption it is within direct radio range. (It will not attempt to communicate over any serial connection.) If the destination node does not acknowledge the message, and your Radio Unicast Retries and Mesh Routing Maximum Hop Limit are not set to zero, normal mesh discovery attempts will occur (including attempting routes over the serial connection).

This means you can eliminate the overhead and latency required of mesh routing in environments where all your nodes are within direct radio range of each other. However it also means that if the Mesh Routing Initial Hop Limit is set to zero and there are times when mesh routing is necessary, those messages will suffer an additional latency penalty as the initial broadcast times out unacknowledged before route requests happen.

*This parameter should remain less than or equal to the next parameter, **Mesh Routing Maximum Hop Limit**.*

ID 28 – Mesh Routing Maximum Hop Limit

Enumeration = snap.NV_MESH_MAX_HOPLIMIT_ID

To cut down on needless broadcast traffic during mesh networking operation (thus saving both power and bandwidth), you can choose to lower this value to the maximum number of physical hops across your network. The default value is 5.

NOTE – if your network is <u>larger</u> than 5 hops, you will need to <u>raise</u> this parameter.

ID 29 – Mesh Sequence Number

Enumeration = snap.NV_MESH_SEQUENCE_NUMBER_ID

Reserved for future use.

ID 30 – Mesh Override

Enumeration = snap.NV_MESH_OVERRIDE_ID

This is used to limit a node’s level of participation within the mesh network.

When set to the default value of 0, the node will fully participate in the mesh networking. This means that not only will it make use of mesh routing, but it will also “volunteer” to route packets for other nodes.

Setting this value to 1 will cause the node to stop volunteering to route packets for other nodes. It will still freely use the entire mesh for its own purposes.

This feature was added to better supports nodes that spend most of their time “sleeping.” If a node is going to be asleep, there may be no point in it becoming part of routes *for other nodes* while it is (briefly) awake.

This can also be useful if some nodes are externally powered, while others are battery-powered. Assuming sufficient radio coverage (all the externally powered nodes can “hear” all of the other nodes), then the **Mesh Override** can be set to 1 in the battery powered nodes, extending their battery life at the expense of reducing the “redundancy” in the overall mesh network.

NOTE – Enabling this feature on your bridge node means Portal will no longer be able to communicate with the rest of your network, regardless of how everything else is configured. No nodes in your network (except for your bridge node) will be able to receive commands or information from Portal or send commands or information to Portal.

ID 31-33 – Embedded SNAP only, not used by *SNAP Connect*

ID 34-38 – Reserved for Future Use

ID 39-41 – Embedded SNAP only, not used by *SNAP Connect*

ID 42-49 – Reserved for Future Use

ID 50 – Enable Encryption

Enumeration = snap.NV_AES128_ENABLE_ID

Control whether encryption is enabled, and what type of encryption is in use for firmware that supports multiple forms. The options for this field are:

0 =	Use no encryption. (This is the default setting.)
1 =	Use AES-128 encryption if you support it.
2 =	Use Basic encryption.

If you set this to a value that indicates encryption should be used, but either an invalid encryption key is specified (in NV Parameter #51), or your firmware does not support the encryption mode specified, your transmissions will not be encrypted.

SNAP versions before 2.4 did not include the option for Basic encryption, and nodes upgraded from those firmware versions may contain False or True for this parameter. Those values correspond to 0 and 1 and will continue to function correctly. Basic encryption is not as secure as AES-128 encryption, but it is available in all nodes.

If encryption is enabled and a valid encryption key is specified, all communication from the node will be encrypted, whether it is sent over the air or over a serial connection. Likewise, the node will expect that all communication to it is encrypted, and will be unable to respond to unencrypted requests from other nodes. If you have a node that you cannot contact because of a forgotten encryption key, you will have to reset the factory parameters on the node to reestablish contact with it.

ID 51 – Encryption Key

Enumeration = snap.NV_AES128_KEY_ID

The encryption key used by either AES-128 encryption or Basic encryption, if enabled. This NV Parameter is a string with default value of “”. If you are enabling encryption, you must specify an encryption key. Your encryption key should be complex and difficult to guess, and it should avoid repeated characters when possible.

An encryption key must be exactly 16 bytes (128 bits) long to be valid. This parameter has no effect unless NV parameter #50 is also set to enable encryption.

Even if NV parameter #50 is set for AES-128 encryption and parameter 51 has a valid encryption key, communications will not be encrypted unless the node supports AES-128 encryption.

ID 52 – Lockdown

Enumeration = snap.NV_LOCKDOWN_FLAGS_ID

If this parameter is 0 (or never set at all), access is unrestricted. You can freely change NV Parameters (even remotely).

If you set this parameter to 1, then the system enters a “lockdown” mode where remote NV Parameter changes are disallowed.

Values other than 0 or 1 are reserved for future use, and should not be used.

While in “lockdown” mode, you also cannot write to NV parameter #52 over-the-air (in other words, you cannot bypass the lockdown by remotely turning it off).

ID 53 – Embedded SNAP only, not used by *SNAP Connect*

ID 54-59 – Reserved for Future Use

ID 60-61 – Embedded SNAP only, not used by *SNAP Connect*

ID 62 – Reserved for Future Use

ID 63-66 – Embedded SNAP only, not used by *SNAP Connect*

ID 67-69 – Reserved for Future Use

ID 70 – Embedded SNAP only, not used by *SNAP Connect*

ID 71-127 – Reserved for Future Use

ID 128-254 – Available for User Definition

These are user-defined NV Parameters, and can be used for whatever purpose you choose (just like in embedded SNAP Nodes).

ID 255 – Embedded SNAP only, not used by *SNAP Connect*

SNAP Connect API Reference – Exceptions

When invoking *SNAP Connect* functions, your program should be prepared to “catch” any Python exceptions that are “thrown” (raised).

The following lists the possible Python exceptions that can be thrown by the *SNAP Connect* libraries, and some possible causes (exception text messages).

NOTE – obvious exceptions like **Exception**, **KeyboardInterrupt**, and **SystemExit** are not shown.

Exceptions that can be thrown from the “apy” package

The following modules from the apy package can throw (raise) exceptions:

PostThread

monotime

Exceptions that can be thrown from the “monotime” module

OSError

Exceptions that can be thrown from the “PostThread” module

PostThreadTerminated

RuntimeError

Exceptions that can be thrown from the “serialwrapper” package

The following modules from the serialwrapper package can throw (raise) exceptions:

ComUtil

PyserialDriver

ftd2xxserialutil

ftd2xxserialwin32

usbxserialutil

usbxserialwin32

Exceptions that can be thrown from the “ComUtil” module

NoMorePortsError

“No more ports to scan”

Exception

“Did not receive expected response”

“Unknown probe version”

Exceptions that can be thrown from the “ftd2xxserialutil” module

Ftd2xxSerialException

“Cannot set number of devices”

portNotOpenError

ValueError

“Serial port MUST have enabled timeout for this function!”

“Not a valid port: …”

“Not a valid baudrate: …”

“Not a valid byte size: …”

“Not a valid parity: …”

“Not a valid stopbit size: …”

“Not a valid timeout: …”

Exceptions that can be thrown from the “ftd2xxserialwin32” module

Ftd2xxSerialException

“Port must be configured before it can be used”

“Could not find device to open: …”

“Could not find port: …”

“Could not open device: …”

“Could not set latency: …”

“Can only operate on an open port”

“Could not set timeouts: …”

“Could not set baudrate: …”

“Could not set break: …”

“Could not clear break: …”

“Could not get CTS state: …”

“Could not get DSR state: …”

“Could not get RI state: …”

“Could not get DCD state: …”

“Could not set stopbits, parity, and/or bits per word: …”

“Could not set flow control: …”

“Cannot configure port, some setting was wrong…”

“An error occurred while checking rx queue: …”

“An error occurred while reading: …”

“An error occurred while writing: …”

“An error occurred while flushing the input buffer: …”

“An error occurred while setting RTS: …”

“An error occurred while clearing RTS: …”

Exceptions that can be thrown from the “PyserialDriver” module

Serial.SerialException

“Write File failed…”

“write failed: …”

PortNotOpenError

TypeError

“Expected str, got …”

“Unknown serial driver type”

“Unsupported output type”

SerialOpenException

“SNAP USB devices are not currently supported on this platform”

“An error occurred while setting up the serial port: …”

Exceptions that can be thrown from the “usbxserialutil” module

ValueError

“Serial port MUST have enabled timeout for this function!”

“Not a valid port: …”

“Not a valid baudrate: …”

“Not a valid byte size: …”

“Not a valid parity: …”

“Not a valid stopbit size: …”

“Not a valid timeout: …”

Exceptions that can be thrown from the “usbxserialwin32” module

portNotOpenError

UsbxSerialException

“Port must be configured before it can be used”

“Unable to verify device PID: …”

“USB device was not found to be a SNAP USB device”

“Could not open device: …”

“Can only operate on an open port”

“Could not set timeout: …”

“Could not set baud rate: …”

“Could not set stop bits, parity, and/or bits per word: …”

“Could not set flow control: …”

“Cannot configure port, some setting was wrong: …”

“Could not get CTS state…”

“An error occurred while checking rx queue: …”

“An error occurred while reading: …”

“Write time out occurred and there are no USB devices”

“A system error occurred while writing: …”

“An error occurred while flushing the input buffer: …”

“An error occurred while flushing the output buffer: …”

Exceptions that can be thrown from the “snapconnect” package

The following modules from the snapconnect package can throw (raise) exceptions:

auth_digest

dispatchers

LicenseManager

listeners

snap

snaptcp

Exceptions that can be thrown from the “auth_digest” module

cherrypy.HTTPError

“Bad Request…”

“You are not authorized to access that resource”

ValueError

“Authorization scheme is not Digest”

“Unsupported value for algorithm…”

“Not all required parameters are present”

“Unsupported value for qop: …”

“If qop is sent then cnonce and nc MUST be present”

“If qop is not sent, neither cnonce nor nc can be present”

“Unrecognized value for qop: …”

Exceptions that can be thrown from the “dispatchers” module

TypeError

“Unknown descriptor type”

Exceptions that can be thrown from the “LicenseManager” module

LicenseError

“… license file has expired”

“Your license is invalid…”

“Unable to load license file…”

Exceptions that can be thrown from the “listeners” module

ValueError

“Unknown serial type”

Exceptions that can be thrown from the “snap” module

ImportError

“Unable to find PyCrypto library required for AES support”

IOError

“Unable to load NV params file: …”

RuntimeError

“Non-licensed address provided”

“Invalid license found”

“Unable to determine callable functions”

TypeError

“Unknown Hook Type”

“Invalid callback”

ValueError

“Unknown encryption type specified: …”

“auth_info is not callable”

“Serial interface type must be an integer”

“Unsupported serial interface type”

Exceptions that can be thrown from the “snaptcp” module

socket.error

“Did not receive a valid lookup result”

ssl.SSLERROR

TypeError

“Invalid conn type”

ValueError

“SSL support was not found”

Exceptions that can be thrown from the “snaplib” package

The following modules from the snaplib package can throw (raise) exceptions:

DataModeCodec

EventCallback

MeshCodec

RpcCodec

SnappyUploader

TraceRouteCodec

Exceptions that can be thrown from the “DataModeCodec” module

DataModeEncodeError

“Packet is greater than 255 bytes”

“The packet is too large to encode …”

Exceptions that can be thrown from the “EventCallback” module

TypeError

Exceptions that can be thrown from the “MeshCodec” module

MeshError

“encode function does not yet support message type …”

“Packet is greater than 255 bytes”

“Mesh Routing message too large to encode …”

Exceptions that can be thrown from the “RpcCodec” module

RpcError

“Did not receive a list of arguments”

“Source Address must be 3 bytes”

“Source Address must be between \xFF\xFF\xFF and \x00\x00\x00”

“No source address was set”

“No destination adderss/group was set”

“The multicast group must be 2 bytes”

“Destination multicast group cannot be all zeros”

“TTL for multicast must be greater than 0 and less than 256”

“The destination address must be 3 bytes”

“Int out of range”

“Function name cannot be None”

“Max string length is 255”

“Unsupported DataType: …”

“Packet is greater than 255 btyes”

“Function/Args too large to encode …”

TypeError

“Unsupported type …”

“String length is greater than 255”

“Integer is out of range”

WrongArgCount

Exceptions that can be thrown from the “SnappyUploader” module

AlreadyInProgressError

“There is currently a SNAPpy upload already in progress”

Exceptions that can be thrown from the “TraceRouteCodec” module

TraceRouteEncodeError

“The data is too large to encode”

“Packet is greater than 255 bytes”

SNAP Connect API Reference – HOOKS

SNAP Hooks are events that can occur at runtime. This section lists the supported hooks, their meanings, their triggers, and their (callback) parameters.

Refer also to the description of the set_hook() (AKA setHook()) function earlier in this manual.

Here the focus is on the individual hooks, and their handlers.

HOOK_SNAPCOM_OPENED – SNAP COMmunications established

Meaning: SNAP COMmunications have been established over a TCP/IP link.

Triggered by:

Establishment of an inbound or outbound TCP/IP connection with another instance of *SNAP Connect* (possibly embedded inside of another application like *Portal*).

This means the root cause of the HOOK_SNAPCOM_OPENED was either:

1) a call to accept_tcp() that allowed/enabled incoming TCP/IP connections
2) a call to connect_tcp() that created an outgoing TCP/IP connection.

Required callback signature:

some_function(*connection_info, snap_address*)

where the parameters are:

connection_info – this is a Python tuple containing **two pieces** of information:

connection_info[0] is the IP address of the other *SNAP Connect* instance.

connection_info[1] is the TCP/IP port of the connection.

snap_address – this is a Python string containing the 3-byte SNAP Address of the other node.

Hint – if you want to know your *own* SNAP Address, try the local_addr() function.

HOOK_SNAPCOM_CLOSED – SNAP COMmunications ended

Meaning: SNAP COMmunications have ended (or failed to ever get established) over a TCP/IP link.

Triggered by:

Shutdown of an established TCP/IP connection, or failure to establish a connection in the first place (for example, calling connect_tcp() with the IP address of a computer that is **not** running *SNAP Connect*).

Required callback signature:

some_function(*connection_info, snap_address*)

where the parameters vary slightly depending on the cause:

Scenario 1 – connection could not be established

connection_info – this is a Python tuple containing **two pieces** of information:

connection_info[0] is the IP address of the other computer.

connection_info[1] is the TCP/IP port of the initial connection attempt.

snap_address – in this scenario, *snap_address* is None because there **was no** SNAP Node at the other end of the attempted connection.

Scenario 2 – an established connection was later shut down

connection_info – this is a Python tuple containing **two pieces** of information:

connection_info[0] is the IP address of the other *SNAP Connect* instance.

connection_info[1] is the TCP/IP port of the connection.

snap_address – this is a Python string containing the 3-byte SNAP Address of the other node.

In other words, the parameters for a valid connection closing are the same as those when it was initially opened (see HOOK_SNAPCOM_OPENED).

HOOK_SERIAL_CLOSE – Serial Communications ended

Meaning: Communications over a specific serial (USB or RS-232) interface have ended.

Triggered by:

This event can only occur after a successful call to open_serial().

A HOOK_SERIAL_CLOSE event can be triggered by a call to close_serial(), or (in the case of some removable USB devices) by the physical removal of the device from the computer *SNAP Connect* is running on.

Required callback signature:

some_function(*serial_type*, *port*)

where the parameters are:

serial_type – The type of the serial port, one of:

SERIAL_TYPE_RS232

SERIAL_TYPE_SNAPSTICK100

SERIAL_TYPE_SNAPSTICK200

port : The port **of that particular type** that closed, as appropriate for your operating system. On Windows, port is a zero-based list. (Specify 0 for COM1 for example.). On Linux, port will typically be a string, for example "/dev/ttys1".

NOTE – these parameters are the same ones used in the initial open_serial() call, and in any subsequent close_serial() call.

HOOK_RPC_SENT – RPC packet sent (or retries exhausted)

Meaning: A packet created previously by a call to rpc() or mcast_rpc() has been sent, or given up on (all retries were exhausted while attempting to send the packet).

Triggered by:

This event fires after any RPC packet is sent, regardless of whether it was sent by your application code, or automatically by *SNAP Connect* behind the scenes.

Required callback signature:

some_function(*packet_identifier, reserved_for_future_use*)

where the parameters are:

packet_identifier – The identifier for which packet was sent.

NOTE – this is the same identifier that was originally returned by the call to rpc() or mcast_rpc().

reserved_for_future_use – ignore this parameter for now.

NOTE – Receipt of a HOOK_RPC_SENT does not mean the packet made it all the way to the other node. This **is not** an end-to-end acknowledgement!

SNAP has no provisions for end-to-end acknowledgement at the protocol layer. Any such functionality must be implemented in <u>your</u> application. For example, have a node send a return RPC in response to a request RPC. Only when the originating node receives the response RPC can it be certain that the original request made it through.

HOOK_STDIN – Data Mode data received

Meaning: A packet containing user data has been received. This could either be unicast data addressed specifically to this node, or it could be multicast data sent to any nodes within that multicast group.

Triggered by:

This event is ultimately triggered by some other node invoking data_mode() or mcast_data_mode(), if they are a *SNAP Connect* application.

Alternatively, an embedded SNAP Node (such as an *RF100*) could be forwarding data (from a serial port for example) by use of the ucastSerial() or mcastSerial() functions (refer to the **SNAP Reference Manual**).

Required callback signature:

some_function(*data*)

where the parameters are:

data – The actual data (payload) of the DATA MODE packet.

Hint – if you need to know who the data came from, try the rpc_source_addr() function.

HOOK_TRACEROUTE – Trace Route data received

Meaning: A Trace Route has been successfully completed.

Triggered by:

This event is by invoking the traceroute() function for a node that actually is reachable.

Required callback signature:

some_function(*node_addr, round_trip_time, hops*)

where the parameters are:

node_addr – A Python string containing the 3-byte SNAP Address of the node that was “traced”.

round_trip_time – the round trip time for the trace route packet, after it made it past the initial hop.

NOTE - The first hop is not counted in Trace Routes because when the feature was originally implemented we were only interested in the over-the-air timing. Bottom line – the *round_trip_time* to your “bridge” node will be reported as 0 milliseconds.

hops – This is a Python **array** of tuples, with one array entry for every “hop” that the trace route made on its journey to and from the target node.

For example, a Trace Route to your directly connected bridge node will have two hops – one for the hop from *SNAP Connect* to the bridge node, and a second hop back from the bridge node to *SNAP Connect*.

Each Python tuple within the *hops* array will be made up of two values:

hop[0] will be the SNAP Address that the Trace Route packet was received from.

hop[1] will be the Link Quality that the Trace Route was received at, ***if*** it was received over a radio link (serial links and TCP/IP links have no true “link quality” in SNAP, and will always be reported as having a LQ of 0).

For an example of how Trace Route results can be displayed, refer to the *Portal* user interface.

For further information on any Synapse product or service, contact us at:

Synapse Wireless, Inc.

500 Discovery Drive
Huntsville, Alabama 35806
256-852-7888
877-982-7888
256-852-7862 (fax)
www.synapse-wireless.com