Users Guide and API Reference

SNAPconnect

351 Electronics Blvd. SW // Huntsville, AL 35824 // (877) 982-7888 // synapsewireless.com

CONTENTS

The **SNAPconnect** Python package is a full **SNAP** implementation, allowing you to create programs that natively interact with the **SNAP** network. This reference guide provides package release notes as well as detailed information about the **SNAPconnect** Python API.

CHAPTER

ONE

API REFERENCE GUIDE

The following sections provide details about **SNAPconnect**'s Python interface:

1.1 Functions

1.1.1 __init__

class Snap(*license_file=None*, *nvparams_file=None*, *funcs=None*, *scheduler=None*, *addr=None*, *rpc_handler=None*)

Initializes a **SNAPconnect** instance.

Parameters

- `license_file` – The full path to a **SNAPconnect** license file. The default value of None indicates that the license file is named License.dat and is located in the present working directory, which will typically be the same directory that holds your Python code files. For the free evaluation license, no License.dat is needed. If **SNAPconnect** cannot find License.dat in the current directory, it will fall back to the free evaluation license.
- `nvparams_file` – The full path to a **SNAPconnect** NV parameters file. The default value of None indicates that the file is named nvparams.dat and is located in the present working directory, which will typically be the same directory that holds your Python code files.
- `funcs` – The callable functions for this instance to expose to the **SNAP** network. If you do not want to expose any callable functions to the **SNAP** network, set funcs to an empty dictionary `{}`. Any given function dictionary should only be passed to a single instance of **SNAPconnect**.
- `scheduler` – Internally used by **SNAPconnect**
- `addr` – The **SNAP** address to use from the license file. If nothing is specified, **SNAPconnect** uses the first address in the license file. The format of this parameter is a three-byte string, e.g., x76 xa2 x5c represents **SNAP** address 76.A2.5C.
- `rpc_handler` – Internally used by **SNAPconnect**

Raises

- *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, and an empty file with default values will be generated if the file is not found
- *RuntimeError("Unable to determine callable functions")* – You must provide a dictionary of callable functions, even if it is an empty one, or None to indicate that all functions in your application are callable.
- *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.
- *ValueError("Unknown encryption type specified* – <encryption type>") : Valid choices are NONE (0), AES128 (1), and BASIC (2).

1.1.2 accept_sniffer

`Snap.accept_sniffer()`

Start allowing remote sniffer connections over TCP.

Returns None

Raises *RuntimeError("You must be accepting TCP connections to call accept_sniffer()")* – You must make a call to *accept_tcp()* prior to accepting remote sniffer connections over TCP.

Note: Incoming remote sniffer connections will be using a different realm for the authentication function than regular TCP connections. See the *Authentication Realms* section for more details.

See also:

- *accept_tcp()*
- *connect_tcp()*
- *stop_accepting_sniffer()*
- *stop_accepting_tcp()*

1.1.3 accept_tcp

`Snap.accept_tcp(`*auth_info=<function server_auth>*, *ip=''*, *port=48625*, *tcp_keepalives=False*, *forward_groups=None*`)`

Start listening for and accepting remote IP connections.

Parameters

- `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. See the *Authentication Realms* section for more details on the realm parameter.
- `ip` – The IPv4 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; the default value of None indicates that the interface will use the value specified in *NV6 - Multicast Forward Groups*. 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.

Returns None

Raises *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 must be a function that **SNAPconnect** can invoke.

Note: Establishment of an actual connection can trigger a *hooks.HOOK_SNAPCOM_OPENED* event. If the connection later goes down it can trigger a *hooks.HOOK_SNAPCOM_CLOSED* event.

See also:

- *connect_tcp()*
- *disconnect_tcp()*
- *stop_accepting_tcp()*

1.1.4 add_rpc_func

`Snap.add_rpc_func`(*rpc_func_name*, *rpc_func*, *allow_alternate_key=False*)

Adds a function to the existing "RPC dictionary".

Parameters

- `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.
- `allow_alternate_key` – Defaults to False if unspecified, but if True adds the specified rpc_func_name to a "whitelist" of functions that can also be remotely invoked with alternate encryption (an alternate encryption key). See also function *rpc_alternate_key_used()*, which is how you can detect that this has actually taken place.

Returns 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.)

Note: When the **SNAPconnect** instance is first instantiated by your application code, you pass 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.

1.1.5 allow_serial_sharing

`static Snap.allow_serial_sharing(`*enabled=True*`)`
Enable or disable sharing of serial connections between **SNAPconnect** instances.

If serial connection sharing is enabled, all open serial connections will be shared when a new **SNAPconnect** instance is constructed. Serial connections opened after a **SNAPconnect** instance is constructed will not be shared. Shared serial connections will be able to receive broadcast messages and route for all connected **SNAPconnect** instances.

Example

This is a static function on the Snap class and should be invoked like this:

```
snap.Snap.allow_serial_sharing(True)
```

Parameters `enabled` – *True* if serial connection sharing should be enabled or *False* if it should be disabled. This parameter is shared across all **SNAPconnect** instances. This defaults to *True*.

Returns None

Note: **SNAPconnect** instances must be running in the same Python interpreter session in order to share serial connections. **SNAPconnect** instances running in different Python interpreter sessions will not be able to share serial connections.

See also:

- *open_serial()*

1.1.6 cancel_upgrade

`Snap.cancel_upgrade(`*addr*`)`
Cancel an over-the-air firmware upgrade.

Parameters `addr` – The three-byte network address of the remote node for which the firmware upgrade is to be cancelled (Ex. “x12x34x56”)

Returns This function returns *True* if the upgrade was canceled. This function returns *False* if no upgrade is currently running for the given address or the upgrade has already completed.

Note: This function can result in a *hooks.HOOK_OTA_UPGRADE_COMPLETE* event if an upgrade was successfully canceled.

See also:

- *upgrade_firmware()*

1.1.7 close_all_serial

static Snap.close_all_serial()
Close all opened serial connections for all **SNAPconnect** instances.

> **Warning:** This is a static function on the Snap class. If multiple instances of **SNAPconnect** are running in the same application, they will all have their serial connections closed.

This is equivalent to calling the *close_serial()* function for every serial connection created with *open_serial()*.

Returns None

Note: This function can trigger one or more *hooks.HOOK_SERIAL_CLOSE* events.

See also:

- *open_serial()*
- *close_serial()*

1.1.8 close_serial

Snap.close_serial(*serial_type*, *port*)
Close the specified serial port if open.

Parameters

- `serial_type` – The type of serial interface to close
- `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`.

Returns None

Note: This function can trigger a *hooks.HOOK_SERIAL_CLOSE* event.

See also:

- *open_serial()*
- *Serial port operations* for serial_type constants

1.1.9 connect_tcp

Snap.connect_tcp(*host*, *auth_info=<function client_auth>*, *port=None*, *retry_timeout=60*, *secure=False*, *forward_groups=None*, *tcp_keepalives=False*, *cache_dns_lookup=False*, *sniffer_callback=None*)
Connect to another **SNAP** node over TCP/IP.

Warning: **SNAPconnect** only supports connecting to IPv4 addresses. If a hostname is provided it must be able to be resolved to an IPv4 address.

Parameters

- `host` – The IPv4 address or hostname of the other **SNAP** node as a dotted string (e.g., "192.168.1.1")
- `auth_info` – The function to call when requesting the client's credentials (defaults to public credentials). If providing a custom function to call, it should have the signature "client_auth(realm)" where the `realm` argument is supplied by the remote server and the function returns a tuple that contains the client's username and password. See the *Authentication Realms* section for more details on the realm parameter.
- `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) **SNAPconnect** can initiate an SSL connection but cannot accept an SSL connection. Some type of proxy is necessary to decrypt and reroute received traffic in order to form an encrypted connection between **SNAPconnect** instances. For example, stunnel (www.stunnel.org) could be used to receive encrypted data on a port, decrypt the data, and forward it to the port **SNAPconnect** is using to accept_tcp. When secure is set to True, the port will default to 443.
- `forward_groups` – Multi-cast groups to forward onward through this interface; defaults to using the value specified in *NV6 - Multicast Forward Groups*. 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.
- `tcp_keepalives` – Enable TCP layer keepalives (default False). If enabled, you may experience conflicts with certain firewall configurations, which may close the connection. The keepalives can also increase the amount of TCP/IP traffic your **SNAP** network generates, which might be an issue if you are using a cellular modem for your network connectivity.
- `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.
- `sniffer_callback` – Callback used for making remote sniffer connections to **SNAPconnect** instances that have called both *accept_tcp()* and *accept_sniffer()*. When this callback is used, regular traffic will not pass across this TCP connection, and the accepting end of the connection will forward all received and transmitted packets to this interface. This function should have the signature "sniffer(descriptor)". See the *Sniffer Descriptors* section for more detailson the descriptor objects that are passed to local and remote sniffers. The sniffer is NOT sniffing over-the-air packets. This callback is only passed packets which are received and transmitted over **SNAPconnect** serial and TCP connections.

Returns None

Note: Establishment of the actual connection can trigger a *hooks.HOOK_SNAPCOM_OPENED* event. If the connection later goes down it can trigger a *hooks.HOOK_SNAPCOM_CLOSED* event.

See also:

- *accept_tcp()*
- *disconnect_tcp()*
- *accept_sniffer()*
- *stop_accepting_tcp()*

1.1.10 data_mode

Snap.data_mode(*dst_addr*, *data*)

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

Parameters

- dst_addr – The three-byte network address of the remote node (Ex. 'x12x34x56')
- data – The data to send

Returns A packet identifier which can be used in the *hooks.HOOK_RPC_SENT* handler.

Note: This function can result in a *hooks.HOOK_STDIN* event at the node specified by dst_addr.

See also:

- *mcast_data_mode()*

1.1.11 directed_mcast_rpc

Snap.directed_mcast_rpc(*conn*, *port*, *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. The multicast message will only be directed out one interface instead of broadcast across all open TCP and serial connections.

Despite the similarity of the names, the result of this function is very different from that of the *dmcast_rpc()* function. This function restricts which interface will be used to send a standard multicast message, while the *dmcast_rpc()* function sends a message across all interfaces (subject to their Multicast Forwarded Groups settings in :ref:'sc_nv6').

Parameters

- conn – For serial connections, this is the serial type passed to *open_serial()*. For TCP interfaces, this value should be a string for the IP address of the interface.
- port – The TCP or serial port number of the interface that should transmit the message.
- group – Specifies which nodes should respond to the request, based on the receiving node's Multicast Process Groups setting in *NV5 - Multicast Process Groups*.
- 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. See *mcast_rpc()* for more details.

Returns A packet identifier which can be used in the *hooks.HOOK_RPC_SENT* handler. This function will return False if the specified interface does not exist.

Note: This function can trigger a *hooks.HOOK_RPC_SENT* event.

See also:

- *mcast_rpc()*
- *dmcast_rpc()*

1.1.12 disconnect_tcp

Snap.disconnect_tcp(*host*, *port=48625*, *all=False*, *retry=False*)

Disconnect from the specified instance.

Parameters

- `host` – The IP address or hostname of the instance from which to disconnect
- `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)

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

Note: This function can result in one or more *hooks.HOOK_SNAPCOM_CLOSED* events being generated.

See also:

- *accept_tcp()*
- *connect_tcp()*
- *stop_accepting_tcp()*

1.1.13 dmcast_rpc

Snap.dmcast_rpc(*dst_addrs*, *groups*, *ttl*, *delay_factor*, *func_name*, **args*)

Makes a Directed Remote Procedure Call, or RPC, using multicast messaging. This means the message could be acted upon by multiple nodes. Unlike a standard multicast, however, the message will only be acted upon by nodes explicitly listed in `dst_addrs`. Unlike an addressed RPC call, this directed multicast does not make use of routing or packet acknowledgement. Other nodes in the mesh network will forward the message (subject to their Multicast Forwarded Groups settings in *NV6 - Multicast Forward Groups*) when there is sufficient TTL to do so. There is no route discovery performed, and there are no retries.

Note that though they have similar names, *dmcast_rpc()* functions very differently from *directed_mcast_rpc()*. The *directed_mcast_rpc()* function allows you to restrict the interface used to transmit your message, but the message sent is a "standard" multicast. This function forwards through all interfaces that would normally forward the message (subject to the multicast group

forwarding settings specified when the interface connection is opened), but will only be acted on by the node(s) specified, and then only if the multicast group specified for the call matches the node's Multicast Process Groups setting in *NV5 - Multicast Process Groups*.

Parameters

- `dst_addrs` – A string containing concatenated three-byte addresses for any nodes you wish to act on the directed multicast. For example, if you have nodes with addresses 01.02.03, 04.05.06, 07.08.09, and 0A.0B.0C, the parameter should contain:

```
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c
```

  If you provide a string that is not a multiple of three characters in length, the *dmcast_rpc()* call will fail and no message will be sent to any node. If you provide an empty string ("") for this parameter, all nodes that receive the message that would otherwise act on the message (subject to the groups parameter and the existence of the function in the node's script) will act on the request as though the call were a regular *mcast_rpc()* call. However in this case, added features available only for directed multicast (such as information available through several *get_info()* calls) are also available.

- `group` – Specifies which nodes should respond to the request, based on the receiving node's Multicast Process Groups setting in *NV5 - Multicast Process Groups*.
- `ttl` – Specifies the Time To Live (TTL) for the request.
- `delay_factor` – Provides a mechanism to allow receiving nodes to stagger their responses to the request. The parameter should be a one-byte integer specifying the amount of time, in milliseconds, that should pass between node responses among the nodes targeted by the request. For example, if the *dst_addrs* parameter contains:

```
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c
```

  and the `delay_factor` parameter contains 40, any radio traffic generated in response by node 01.02.03 would be released immediately, radio traffic generated by 04.05.06 would be queued and held for 40 ms, radio traffic generated by 07.08.09 would be delayed for 80 ms, and radio traffic generated by 0A.0B.0C would be held for 120 ms before release. Note that the function on the target node is executed without delay. The delay only applies to radio communications directly invoked by the called function. No delay is applied to serial communications from the receiving node. Thus, an instance of **SNAPconnect**, which has only serial interfaces, is not affected by this parameter. Setting this parameter to zero allows all receiving nodes with radio interfaces to respond immediately, which may cause packet loss due to interference. A check of `get_info(25)` in the called function on the receiving node returns the `delay_factor` value specified, but also tells the receiving node to ignore the transmission delay that would normally be enforced.

- `func_name` – The function name to be invoked.
- `args` – Any arguments for the function specified by func_name. See *mcast_rpc()* for more details.

Returns A packet identifier which can be used in the *hooks.HOOK_RPC_SENT* handler. This function will return False if the specified interface does not exist.

Note: This function can trigger a *hooks.HOOK_RPC_SENT* event.

See also:

- *mcast_rpc()*
- *directed_mcast_rpc()*
- *rpc()*

New in version 3.4.

1.1.14 get_info

`Snap.get_info(`*which_info*`)`
Get the specified system information.

Parameters `which_info` – Specifies the type of information to be retrieved (0-28 but with some gaps – not all of the "info" types from the original [embedded] **SNAP** nodes apply to a PC-based application like **SNAPconnect**). This function returns the requested information or None if the `which_info` parameter 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 “**SNAPconnect**”
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 3.2.1 `get_info(5)` returns 3
6	Software MINOR version	Example: if version is 3.2.1 `get_info(6)` returns 2
7	Software BUILD version	Example: if version is 3.2.1 `get_info(7)` returns 1
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	For **SNAPconnect** applications you should use the value returned by the *rpc()*, *mcast_rpc()*, and *dmcast_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
15	Routes stored in Route Table	Returns the number of active routes
24	Is Directed Multicast	Returns a 1 if the function running was invoked by a directed multicast using *dmcast_rpc()*. Returns a 0 if the function running was invoked by a hook or scheduled event, by an addressed RPC, or by a “normal” multicast (i.e., *directed_mcast_rpc()* or *mcast_rpc()*).
25	Read Delay Factor	Returns the delay factor specified for a message sent using *dmcast_rpc()*. This delay factor has no effect on a **SNAPconnect**-based node, because the **SNAPconnect** instance has only serial interfaces.
26	Address Index	Directed multicast messages sent using *dmcast_rpc()* can target multiple nodes by concatenating multiple **SNAP** addresses in the `dst_addrs` parameter. This option indicates where the address of the contextual node appears in that list, as a zero-based index.
27	Multicast Groups	Returns an integer indicating the multicast group mask specified for the packet when it was originally sent, if it was sent as a directed multicast using *dmcast_rpc()*.
28	Original TTL	Returns the TTL (the “hops”) field specified for the packet when it was originally sent, if it was sent as a directed multicast using *dmcast_rpc()*. Combined with

Note: The string `snap.VERSION` is equivalent to retrieving the major, minor and build revisions (getinfo 5, 6, and 7 respectively). For example:

```
>>> from snapconnect import snap
>>> snap.VERSION
'3.4.0'
```

1.1.15 load_nv_param

`Snap.load_nv_param(`*nv_param_id*`)`

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 **SNAPconnect** E12). In such cases, a request for that parameter will return the "hardware" value rather than any artificially stored value.

Parameters `nv_param_id` – Specifies which "key" to retrieve from storage. Some NV parameters may have no effect on the **SNAPconnect** instance.

Returns The indexed parameter from storage.

See also:

- *save_nv_param()*
- *NV Parameters* section for more details about the individual NV Parameters

1.1.16 local_addr

`Snap.local_addr()`

Returns the local network address of this instance.

Returns The three-byte network address of the **SNAPconnect** instance (Ex. "x12x34x56").

1.1.17 loop

`Snap.loop()`

Calling the loop function is essentially equivalent to using this in your code:

```
from time import sleep
while True:
    poll()
    sleep(0.001)
```

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

Example

The 1 ms sleep between polls prevents **SNAPconnect** from completely monopolizing your CPU. You can adjust this sleep duration by setting the `snap.SLEEP_TIME` global in your code to a value, in seconds. Longer sleep periods between checks consume less of your CPU, but also reduce the respon-

siveness of your network as it takes longer to get packets between **SNAPconnect** and your bridge node (or other interface):

```
snap.SLEEP_TIME = 0.1
loop()
```

Note: If your application needs to take action on its own behalf (for example, if it needs to be polling other nodes but needs to perform its own independent processing between received messages), then you probably should be using the *poll()* function instead.

See also:

- *poll()*
- *poll_internals()*

1.1.18 mcast_data_mode

Snap.mcast_data_mode(*group*, *ttl*, *data*)

Sends a multicast transparent (aka data) mode packet to the nodes specified by the group.

Parameters

- group – Specifies which nodes should respond to the request, based on the receiving node's Multicast Process Groups setting in *NV5 - Multicast Process Groups*
- ttl – Specifies the Time To Live (TTL) for the request
- data – The data to send

Returns A packet identifier which can be used in the *hooks.HOOK_RPC_SENT* handler.

Note: This function can result in a *hooks.HOOK_STDIN* event in one or more other nodes.

See also:

- *data_mode()*

1.1.19 mcast_rpc

Snap.mcast_rpc(*group*, *ttl*, *func_name*, **args*)

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

Parameters

- group – Specifies which nodes should respond to the request, based on the receiving node's Multicast Process Groups setting in *NV5 - Multicast Process Groups*
- 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: Arguments 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!
```

Returns A packet identifier which can be used in the *hooks.HOOK_RPC_SENT* handler.

Note: This function can trigger a *hooks.HOOK_RPC_SENT* event.

See also:

- *rpc()*

1.1.20 open_serial

`Snap.open_serial(`*serial_type*, *port*, *reconnect=False*, *dll_path=MODULE_PATH*, *forward_groups=None*, *baudrate=None*, *io_loop=None*`)`

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

Parameters

- `serial_type` – The type of serial interface to open
- `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)
- `forward_groups` – Multi-cast groups to forward through; defaults to using NV param
- `baudrate` – The baudrate for the serial connection
- `io_loop` – A Tornado IOLoop instance

See also:

- *Serial port operations* for serial_type constants

1.1.21 poll

`Snap.poll()`

Polls all needed components packages.

1.1.22 poll_internals

Snap.poll_internals()
Polls only internal components for this instance.

1.1.23 replace_rpc_func

Snap.replace_rpc_func(*rpc_func_name*, *rpc_func*)
Replace an existing function to be called remotely via RPC.

Parameters

- rpc_func_name – An existing public RPC function name
- rpc_func – The callable to be executed via RPC

Returns The original function that was replaced None if the function name had not been previously added

1.1.24 rpc

Snap.rpc(*dst_addr*, *func_name*, **args*)
Sends a unicast RPC.

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

Parameters

- dst_addr – The three-byte network address of the remote node (Ex. “x12x34x56”)
- func_name – The function name to be invoked on the remote node
- args – Any arguments for the function specified by func_name

1.1.25 rpc_alternate_key_used

Snap.rpc_alternate_key_used()

Returns True if the packet came in with the alternate encryption key and the function was on the whitelist.

1.1.26 rpc_source_addr

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

Returns The three-byte network address of the remote node which initiated the RPC call (Ex. “x12x34x56”).

1.1.27 rpc_source_interface

`Snap.rpc_source_interface()`
Originating interface of the current RPC context (None if called outside RPC).

Returns The interface of the remote node which initiated the RPC call.

1.1.28 rpc_use_alternate_key

`Snap.rpc_use_alternate_key(`*enable_flag*`)`
Enable and disable use of the alternate key. When enabled, this setting persists until you disable it, and it affects ALL multicast RPC calls you send.

Parameters `enable_flag` – If True, use the alternate key. If False, revert to the normal key.

1.1.29 save_nv_param

`Snap.save_nv_param(`*nv_param_id*, *obj*, *optional_bitmask=None*`)`
Store individual objects for later access by id.

Parameters

- `nv_param_id` – Specifies which "key" to store the obj parameter under. 0-127 are pre-assigned system IDs.
- `obj` – The object you would like to store for later use.
- `optional_bitmask` – For integers only, you can specify a bitmask of "bits of obj to be applied". Any bits not set in optional_bitmask remain intact.

See also:

- *load_nv_param()*
- *NV Parameters* section for more details about the individual NV Parameters

1.1.30 set_hook

`Snap.set_hook(`*hook*, *callback=None*`)`
Set the specified **SNAP** hook to call the provided function.

Parameters

- `hook` – The **SNAP** event hook identifier
- `callback` – The function to invoke when the specified event occurs

1.1.31 start_sniffer

`Snap.start_sniffer(`*callback*`)`
Start a local sniffer that passes descriptors to the given callback.

Parameters `callback` – Called when packets are sent or received by this **SNAPconnect** instance.

Returns True if a sniffer could be started. False if one was already running.

1.1.32 stop_accepting_sniffer

`Snap.stop_accepting_sniffer(`*close_existing=False*`)`
Stop accepting remote sniffer connections over TCP.

Parameters `close_existing` – If True, close all existing sniffer connections

1.1.33 stop_accepting_tcp

`Snap.stop_accepting_tcp(`*close_existing=False*`)`
Stop listening for and accepting remote IP connections.

Parameters `close_existing` – If True, close all existing TCP connections

1.1.34 stop_sniffer

`Snap.stop_sniffer()`
Stop a running local sniffer.

Returns True if the sniffer could be stopped. False if one was not already running.

1.1.35 traceroute

`Snap.traceroute(`*dst_addr*`)`
Sends a traceroute request.

Parameters `dst_addr` – The three-byte network address of the remote node (Ex. "x12x34x56")

1.1.36 upgrade_firmware

`Snap.upgrade_firmware(`*addr*, *file*`)`
Start an over the air firmware upgrade.

Programming may take up to 2 minutes after the transfer completes. Do not remove power to the device during this time.

Parameters

- `addr` – The three-byte network address of the node to upgrade (Ex. "x12x34x56")
- `file` – The path to the firmware image

Returns Returns True if an upgrade can be started or False if the upgrade can't be started for some reason. Will result in a call to the OTA upgrade complete hook.

Note: This function will trigger a *`hooks.HOOK_OTA_UPGRADE_COMPLETE`* event when an upgrade completes successfully or an error occurs that halts the upgrade process. A `hooks.HOOK_OTA_UPGRAGE_STATUS` event will be triggered every time progress is made in an over the air upgrade.

1.2 Constants and Enumerations

This section lists and describes the numerous constants defined by the **SNAPconnect** library.

Note: The constants defined here are in the `snap` Python module. For readability, we have removed the module prefix, but you will need to prefix them with `snap.` in your code. For example:

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

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

1.2.1 Encryption

Constant	Description
`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.

1.2.2 Serial port operations

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

Constant	Description
`SERIAL_TYPE_SN132_SNAP`	The SN132 SNAPstick, sometimes referred to as a “paddle board”, or the SN163 demonstration board, sometimes referred to as a “bridge 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_SS200_SNAP`	The SS200 SNAPstick has a plastic case and does not accept plug-in **SNAP** Engines. It is completely self-contained and has to be plugged into a USB port.
`SERIAL_TYPE_RS232`	“True” RS232 COM port, or a USB-serial cable.

1.2.3 Rpc_source_interface()

Constant	Description
`INTF_TYPE_UNKNOWN`	You should never see this
`INTF_TYPE_802154`	For future use, as **SNAPconnect** 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 (i.e., an SN132 or SN163)
`INTF_TYPE_ETH`	RPC call came in over TCP/IP
`INTF_TYPE_SNAPSTICK`	RPC call came in from an SS200 SNAPstick

1.2.4 SPY Uploading

Constant	Description
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 **SNAPconnect** instance rather than an embedded node.

1.2.5 Firmware upgrades

Constant	Description
OTA_PROGRESS_COMPLETE	Upgrade completed successfully
OTA_PROGRESS_ERROR	Upgrade failed
OTA_PROGRESS_CANCELED	Upgrade canceled
OTA_PROGRESS_TIMEOUT	Node failed to respond
OTA_PROGRESS_WRITE_ERROR	FLASH write failure
OTA_PROGRESS_WRITE_REFUSED	Power too low to attempt
OTA_PROGRESS_UNSUPPORTED	Node does not support over the air firmware upgrade

1.2.6 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)
```

1.3 NV Parameters

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

SNAPconnect 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, **SNAPconnect** NV Parameter changes take effect immediately (no reboot required).

Below is a list and description of the System (Reserved) NV Parameters that apply to **SNAPconnect**. You can use these same constants when accessing NV parameters on an embedded node from a **SNAPconnect** script, even if the parameter has no meaning in the **SNAPconnect** context (such as querying for a node's radio link quality threshold).

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.

1.3.1 NV0-4 - Reserved

Reserved for Synapse use.

1.3.2 NV5 - Multicast Process Groups

`snap.NV_GROUP_INTEREST_MASK_ID = 5`

This is a 16-bit field controlling which multicast groups the node will respond to. It is a bit mask, with each bit representing one of 16 possible multicast groups. For example, the 0x0001 bit represents the default group, or "broadcast group." Removing a node from that group will make the node unable to respond to **Portal**'s multicasts, such as global pings.

Default Value = `0x0001`, which is the broadcast group

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

For example, **Portal** could multicast a "sleep" command to group 0x0002, and only nodes with that bit set in their **Multicast Process 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 - or none) of the 16 groups.

Group membership does not affect how a node responds to a direct RPC call. It only affects multicast requests. However, many of the infrastructure calls made "behind the scenes" in a network, such as route requests, are performed using multicasts on group 1.

See also:

- User Guide section on Multicast Groups
- *NV6 - Multicast Forward Groups*

1.3.3 NV6 - Multicast Forward Groups

`snap.NV_GROUP_FORWARDING_MASK_ID = 6`

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

Default Value = `0x0001`, so all nodes process and forward only broadcast group packets

The *NV5 - Multicast Process Groups* and *NV6 - Multicast Forward Groups* parameters 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. It can forward one set of groups over its radio interface and a different set of groups, with or without overlap, over its serial interface. As with processing groups, a node can be set to forward any combination of the 16 available groups, including none of them.

> **Warning:** If you set your bridge node to not forward multicast commands, **Portal** will not be able to multicast to the rest of your network.

Note: Every interface opened in a **SNAPconnect** application has a `forward_groups=None` parameter available when you open (or listen for) the interface. This parameter restricts which multicast messages will be forwarded through that interface from the **SNAPconnect** instance, with the default value of None indicating

that the interface will only allow multicast packets for groups that the **SNAPconnect** instance itself would forward. The hazard here is that if you do not explicitly set the `forward_groups` parameter when you open a connection, **SNAPconnect** will only be able to make multicast requests of nodes in groups that it would normally forward. In other words, if you want **SNAPconnect** to be able to initiate multicast requests to a group that it would not normally forward, you must explicitly set that group as a forwarding group when you open the interface.

See also:

- User Guide section on Multicast Groups
- *NV5 - Multicast Process Groups*

1.3.4 NV7 - Reserved

Reserved for Synapse use.

1.3.5 NV8 – Device Name

`snap.NV_DEVICE_NAME_ID = 8`

Allows you to assign a name for the node, although you do not have to give your nodes explicit names. If this parameter is set to `None`, then the node name will default to "SNAPcom". You do not have to give your nodes explicit names.

Default Value = `None`

Note: Spaces are not allowed in your Device Name. `"My Node"` is not a legal name, while `"My_Node"` is.

1.3.6 NV9-10 - Reserved

Reserved for future Synapse use.

1.3.7 NV11 - Feature Bits

`snap.NV_FEATURE_BITS_ID = 11`

These control some miscellaneous hardware settings on embedded **SNAP** nodes.

Feature Bit Name	Hex	Binary
Second CRC	0x0100	b0000.0001.0000.0000
Packet CRC	0x0400	b0000.0100.0000.0000

Second CRC The Second CRC bit (0x0100) enables a second CRC packet integrity check on platforms that support it. It does not apply to data mode packets or to infrastructure packets such as message acknowledgements. While this feature bit is still supported, the CRC provided by the *Packet CRC* bit is recommended.

Default Value = `0`

> **Warning:** If you set this bit for the second CRC, you must set it in all nodes in your network, including **Portal** and any **SNAPconnect** applications. 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.

Packet CRC The Packet CRC bit (0x0400) adds an additional CRC validation to the complete packet for every packet sent out over the air. This reduces the available packet payload, 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. This additional CRC should reduce the chance to less than one in four billion.

Default Value = `0`

This is different from the CRC controlled by the *Second CRC* bit in that it includes packet (payload and header) information for RPC, data, routing and acknowledgement packets rather than just covering the RPC payload.

Enabling this CRC reduces your maximum packet payload by two bytes each:

Packet CRC Bit (Bit 10)	Max Unicast Payload	Max Multicast Payload
0	108 bytes	111 bytes
1	106 bytes	109 bytes

> **Warning:** If you set this bit for packet-level CRC, you must set it in all nodes in your network. It is also recommended to configure **Portal** to match your **SNAPconnect** application to prevent generating packets that exceed the new maximum payload in your network.

1.3.8 NV12-18 - Reserved

Reserved for Synapse use.

1.3.9 NV19 - Radio Unicast Retries

`snap.NV_SNAP_MAX_RETRIES_ID = 19`

This lets you control the number of unicast transmit attempts.

Default Value = `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 at least one chance 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.

1.3.10 NV20 - Mesh Maximum Timeout

`snap.NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID = 20`

This indicates the maximum time (in milliseconds) a route can "live."

Default Value = `60000`, which is one minute

Discovered mesh routes timeout after a configurable period of inactivity (see *NV21 - Mesh Minimum Timeout*), but this timeout sets an upper limit on how long a route will be kept, even if it is being used heavily. By forcing routes to be rediscovered periodically, the nodes will use the shortest routes possible, at the expense of some time spent rediscovering routes when the routes expire.

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 `getInfo(14)` to determine the size of a node's route table (typically 10 entries, but that can vary on some platforms), and `getInfo(15)` to monitor its use.

1.3.11 NV21 - Mesh Minimum Timeout

`snap.NV_MESH_ROUTE_AGE_MIN_TIMEOUT_ID = 21`

This is the minimum time (in milliseconds) a route will be kept, subject to the route table becoming full.

Default Value = `1000`, which is one second

Note: In a busy mesh environment, routes may fall out of the route table before this time period has elapsed if the route table becomes full and another route is discovered.

1.3.12 NV22 - Mesh New Timeout

`snap.NV_MESH_ROUTE_NEW_TIMEOUT_ID = 22`

This is the grace period (in milliseconds) that a newly discovered route will be kept, even if it is never actually used, subject to the route table becoming full.

Default Value = `5000`, which is five seconds

Note: In a busy mesh environment, routes may fall out of the route table before this time period has elapsed if the route table becomes full and another route is discovered.

1.3.13 NV23 - Mesh Used Timeout

`snap.NV_MESH_ROUTE_USED_TIMEOUT_ID = 23`

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. *NV21 - Mesh Minimum Timeout* takes precedence over this timeout.

Default Value = 5000, which is five seconds

1.3.14 NV24 - Mesh Delete Timeout

```
snap.NV_MESH_ROUTE_DELETE_TIMEOUT_ID = 24
```

This timeout (in milliseconds) controls how long "expired" routes are kept around for bookkeeping purposes.

Default Value = 10000, which is ten seconds

1.3.15 NV25 - Mesh RREQ Retries

```
snap.NV_MESH_RREQ_TRIES_ID = 25
```

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.

Default Value = 3

1.3.16 NV26 - Mesh RREQ Wait Time

```
snap.NV_MESH_RREQ_WAIT_TIME_ID = 26
```

This parameter (in milliseconds) controls how long a node will wait for a response to a Route Request (RREQ) before trying again.

Default Value = 500, which is a half second

Note that subsequent retries use longer and longer timeouts (the timeout is doubled each time). This allows nodes from farther and farther away time to respond to the RREQ packet.

1.3.17 NV27 - Mesh Initial Hop Limit

```
snap.NV_MESH_INITIAL_HOPLIMIT_ID = 27
```

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

Default Value = 2

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 2 will use less radio bandwidth.

SNAPconnect does not support disabling mesh discovery by setting this value to 0. A value of 0 will be treated as if you had set the initial hop limit to the default value of 2.

This parameter should remain less than or equal to *NV28 - Mesh Maximum Hop Limit*.

1.3.18 NV28 - Mesh Maximum Hop Limit

```
snap.NV_MESH_MAX_HOPLIMIT_ID = 28
```

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.

Default Value = 5

Tip: If your network is larger than 5 hops, you will need to raise this parameter.

1.3.19 NV29 - Reserved

Reserved for Synapse use.

1.3.20 NV30 - Mesh Override

`snap.NV_MESH_OVERRIDE_ID = 30`

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

Default Value = `0`

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 (subject to other nodes' willingness to route packets for it).

This feature was added to better support nodes that spend most of their time "sleeping." If a node is likely to spend most of its time 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**.

1.3.21 NV31-49 - Reserved

Reserved for Synapse use.

1.3.22 NV50 - Enable Encryption

`snap.NV_AES128_ENABLE_ID = 50`

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:

ENCRYPTION_TYPE_NONE =	Use no encryption. (This is the default setting.)
ENCRYPTION_TYPE_AES128 =	Use AES-128 encryption if supported.
ENCRYPTION_TYPE_BASIC =	Use Basic encryption.

See also:

- *Encryption* enumeration

Default Value = `ENCRYPTION_TYPE_NONE`

If you set this to a value that indicates encryption should be used, but either an invalid encryption key is specified (in *NV51 - Encryption Key*), or your Python environment 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 of the encryption enumeration and will continue to function correctly. Basic encryption is not as secure as AES-128 encryption, but it is available in all nodes starting with release 2.4.

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.

Enabling AES-128 encryption in **SNAPconnect** requires that you have the third-party PyCrypto Python library installed. You can acquire the library from http://pycrypto.org/.

1.3.23 NV51 - Encryption Key

`snap.NV_CRYPTO_KEY = 51`

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.

Default Value = ““

An encryption key must be exactly 16 bytes (128 bits) long to be valid. This parameter has no effect unless *NV50 - Enable Encryption* is also set to enable encryption.

Even if *NV50 - Enable Encryption* is set for AES-128 encryption and *NV51 - Encryption Key* has a valid encryption key, **SNAPconnect** will fail with an exception if you do not have the PyCrypto Python library installed.

1.3.24 NV52 - Lockdown

`snap.NV_LOCKDOWN_FLAGS_ID = 52`

If this parameter is set to 2, the system is put into a “lockdown” mode, where NV Parameters may not be changed remotely. If it is set to 0 (or never set at all), NV Parameters may be changed (even remotely).

Default Value = `0`

Values other than 0 or 2 are reserved for future use and should not be used on a **SNAPconnect** node.

1.3.25 NV53-54 - Reserved

Reserved for Synapse use.

1.3.26 NV55 - Alternate Encryption Key

`snap.NV_ALTERNATE_KEY_ID = 55`

This NV parameter is only in **SNAPconnect**. It is like *NV51 - Encryption Key* in form and function. It represents a second ("alternate") encryption key that can be used in special circumstances.

Default Value = ""

For example, you could be running a fully encrypted network using key "encryption_key_1" when a new node is added with encryption key "KEY2(ENCRYPTION)". By saving the second key in *NV55 - Alternate Encryption Key* and proper use of the `add_rpc_func()`, `rpc_use_alternate_key()`, and `rpc_alternate_key_used()` functions, you can perform multicast interactions with the new node while still continuing to interact fully (both multicast and unicast) with the old nodes. This gives you a way to reprogram the new node to switch over to using the first key without having to pause communications to the rest of the network.

1.3.27 NV56-127 - Reserved

Reserved for Synapse use.

1.3.28 NV128-254 - User Defined

These are user-defined NV Parameters and can be used for whatever purpose you choose, just as with embedded **SNAP** nodes. Unlike embedded **SNAP** nodes, however, **SNAPconnect** lets you store any pickleable object in an NV parameter, rather than just a string, Boolean, integer, or None.

1.3.29 NV255 - Reserved

Reserved for future Synapse use.

1.4 Authentication Realms

The realm parameter is used by **SNAPconnect** when forming TCP connections. The value is passed to the authentication functions for accepting and connecting TCP. In versions of **SNAPconnect** prior to 3.2, this field would always take on the value "SNAPcom." With the addition of remote sniffers in **SNAPconnect** 3.2, the realm value can now be used to distinguish between regular TCP connections and remote sniffer TCP connections.

If desired, the realm parameter may be ignored in your client and server authentication functions and the login information for all connection types will be shared.

See also:

- *accept_tcp()*
- *accept_sniffer()*
- *connect_tcp()*

1.4.1 REALM_SNAP

snap.REALM_SNAP

Standard SNAP over TCP Connections This realm will be passed to authentication functions when *connect_tcp()* is used to create a connection between **SNAPconnect** instances. The parameter sniffer_callback should be set to None when creating a standard TCP connection.

1.4.2 REALM_SNIFFER

snap.REALM_SNIFFER

Remote Sniffer connection over TCP This realm will be passed to authentication functions when *connect_tcp()* is used to create a remote sniffer connection between **SNAPconnect** instances. The parameter sniffer_callback should be set to a callable function when creating a remote sniffer TCP connection.

This realm will only be used if *accept_sniffer()* has been called in addition to *accept_tcp()*.

1.5 Hooks

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

See also:

- *set_hook()*

1.5.1 HOOK_SNAPCOM_OPENED

hooks.HOOK_SNAPCOM_OPENED

SNAP communications have been established over a TCP/IP link.

Parameters

- connection_info (*tuple*) – The IP address and TCP/IP port of the other **SNAPconnect** instance being connected to.
- snap_address (*string*) – The three-byte **SNAP** address of the other node (Ex. "\x12\x34\x56").

Triggered By

This event is triggered by the establishment of an inbound or outbound TCP/IP connection with another instance of **SNAPconnect**, meaning that the root cause of the event was either:

- a call to *accept_tcp()* that allowed/enabled incoming TCP/IP connections; or
- a call to *connect_tcp()* that created an outgoing TCP/IP connection.

Example

```
def on_sc_open(connection_info, snap_address):
    pass
```

Note: If you want to know your own **SNAP** address, use the *local_addr()* function.

1.5.2 HOOK_SNAPCOM_CLOSED

hooks.HOOK_SNAPCOM_CLOSED

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

Scenario 1 – connection could not be established:

Parameters

- connection_info (*tuple*) – The IP address and TCP port of the other **SNAPconnect** instance for the failed connection attempt.
- snap_address (*string*) – None.

Scenario 2 – an established connection was shut down (parameters should match those provided with the SNAPCOM_OPENED hook):

Parameters

- connection_info (*tuple*) – The IP address and TCP port of the other **SNAPconnect** instance.
- snap_address (*string*) – The three-byte **SNAP** address of the other node (Ex. "\x12\x34\x56").

Triggered By

This event is triggered by the 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 **SNAPconnect**).

Example

```
def on_sc_close(connection_info, snap_address):
    pass
```

1.5.3 HOOK_SERIAL_OPEN

hooks.HOOK_SERIAL_OPEN

Communications over a specific serial (USB or RS-232) interface have started.

Parameters

- serial_type – The type of the serial port, one of: SERIAL_TYPE_RS232, SERIAL_TYPE_SNAPSTICK100, SERIAL_TYPE_SNAPSTICK200.

- `port` – The port of specified *serial_type* that opened, as appropriate for your operating system. On Windows, port is a zero-based list (for example, specify 0 for COM1). On Linux, port will typically be a string (for example, `/dev/ttys1`).
- `addr` (*string*) – The three-byte address of the **SNAP** device attached to this serial port (Ex. "\x12\x34\x56"), or None if the address could not be retrieved for any reason.

Triggered By

This event will occur after every successful call to *open_serial()*. After *open_serial()* is called, **SNAPconnect** will attempt to retrieve the address of the device attached to the serial port. This event will trigger after the address has been retrieved or the retrieval attempt has timed out.

Because the serial connection is open immediately after the call to *open_serial()*, it is possible that packets may be received on the interface before the node address can be determined and the hook function is called.

Example

```
def on_serial_open(serial_type, port, addr):
    pass
```

1.5.4 HOOK_SERIAL_CLOSE

`hooks.HOOK_SERIAL_CLOSE`

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

Parameters

- `serial_type` – The type of the serial port, one of: `SERIAL_TYPE_RS232`, `SERIAL_TYPE_SNAPSTICK100`, `SERIAL_TYPE_SNAPSTICK200`.
- `port` – The port of specified serial_type that closed, as appropriate for your operating system. On Windows, port is a zero-based list (for example, specify 0 for COM1). On Linux, port will typically be a string (for example, `/dev/ttys1`).

Triggered By

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 **SNAPconnect** is running on. This event can only occur after a successful call to *open_serial()*.

Example

```
def on_serial_close(serial_type, port):
    pass
```

1.5.5 HOOK_RPC_SENT

hooks.HOOK_RPC_SENT

A packet created previously by a call to *rpc()*, *mcast_rpc()*, or *dmcast_rpc()* has been sent or all retries were exhausted while attempting to send the packet.

Parameters

- packet_identifier – The identifier for which packet was sent.

  Note: packet_identifier is the same identifier that was originally returned by the call to *rpc()*, *mcast_rpc()*, or *dmcast_rpc()*. For compatibility with embedded **SNAP** nodes, you can also recover this identifier immediately after sending an RPC using get_info(9).

- transmit_success – True if **SNAPconnect** believes the packet has been transmitted successfully or False if **SNAPconnect** was unable to send the packet. A successful transmit does not indicate that the packet was able to reach the destination.

Receipt of a *hooks.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. If transmit_success is False, the packet has not been transmitted. But if transmit_success is True, that is no guarantee the destination node actually received the packet.

SNAP has no provisions for end-to-end acknowledgement at the protocol layer. Any such functionality must be implemented in your 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.

Triggered By

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

Example

```
def on_rpc_sent(packet_identifier, transmit_success):
    pass
```

1.5.6 HOOK_STDIN

hooks.HOOK_STDIN

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 specified multicast group[s].

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

Triggered By

This event is ultimately triggered by some other node invoking *data_mode()* or *mcast_data_mode()*, if it is a **SNAPconnect** application. Alternatively, an embedded **SNAP** node (such as an RF220) could be forwarding data from a serial port by use of the ucastSerial() or mcastSerial() functions.

Example

```
def on_stdin_rx(data):
    pass
```

Hint: If you need to know who the data came from, use the *rpc_source_addr()* function.

1.5.7 HOOK_TRACEROUTE

hooks.HOOK_TRACEROUTE

A traceroute has been successfully completed.

Parameters

- node_addr (*string*) – The three-byte **SNAP** address of the node that was "traced" (Ex. "\x12\x34\x56").
- round_trip_time – The round trip time for the traceroute packet, after it made it past the initial hop.

  Note: The first hop is not counted in traceroutes, because when the feature was originally implemented it was decided to only track the over-the-air timing. Bottom line – the bridge node component of your round_trip_time will be reported as 0 milliseconds. For TCP connections, this could significantly under-report the speed of that first hop.

- hops – A Python list of tuples, with one list entry for every "hop" that the traceroute made on its journey to and from the target node.

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

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

  - hop[0] will be the **SNAP** address that the traceroute packet was received from.
  - hop[1] will be the link quality at which the traceroute was received, *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.) The link quality is reported in (negative) dBm, so lower values represent stronger signals. The theoretical range for the link quality is 0-127.

Triggered By

This event is triggered by invoking the *traceroute()* function for a node that is actually reachable and can respond.

Example

```
def on_traceroute_done(node_addr, round_trip_time, hops):
    pass
```

1.5.8 HOOK_OTA_UPGRADE_COMPLETE

hooks.HOOK_OTA_UPGRADE_COMPLETE
An over-the-air firmware upgrade has completed.

Parameters

- upgrade_addr (*string*) – The three-byte **SNAP** address of the node for which the upgrade has completed (Ex. "\x12\x34\x56").
- status_code – The result of the upgrade. See the *Firmware upgrades* section for more details.
- message – A human-readable string describing the cause of an error, or None if the upgrade was successful.

Triggered By

This event follows a call to the *upgrade_firmware()* function. The event is triggered when an upgrade completes successfully or an error occurs that halts the upgrade progress.

Example

```
def on_ota_complete(upgrade_addr, status_code, message):
    pass
```

See also:

- *Firmware upgrades* enumeration

1.5.9 HOOK_OTA_UPGRADE_STATUS

hooks.HOOK_OTA_UPGRADE_STATUS
An over-the-air firmware upgrade status has advanced.

Parameters

- upgrade_addr (*string*) – The three-byte **SNAP** address of the node receiving the upgrade (Ex. "\x12\x34\x56").
- percent_complete – An estimate (as a Python float) of how much of an upgrade has been completed, as a percentage.

Triggered By

This event follows a call to the *upgrade_firmware()* function. The event is triggered every time progress is made in an over-the-air upgrade and will be called many times for every upgrade started and not immediately stopped.

Example

```
def on_ota_status(upgrade_addr, percent_complete):
    pass
```

1.6 Exceptions

When invoking **SNAPconnect** 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 **SNAPconnect** libraries and some possible causes (exception text messages).

Note: Obvious exceptions like `Exception`, `KeyboardInterrupt`, and `SystemExit` are not shown.

1.6.1 Thrown by `apy`

The monotime module in the apy package can throw the following exception:

- `OSError`

The PostThread module in the apy package can throw the following exception:

- `RuntimeError`

1.6.2 Thrown by `serialwrapper`

The `ComUtil` module in the `serialwrapper` package can throw the following exceptions:

- `NoMorePortsError`
  - "No more ports to scan"
- `Exception`
  - "Did not receive expected response"
  - "Unknown probe version"

The `ftd2xxserialutil` module in the `serialwrapper` package can throw the following exceptions:

- `Ftd2xxSerialException`
  - "Cannot set number of devices"
- `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: ..."

The `ftd2xxserialwin32` module in the `serialwrapper` package can throw the following exceptions:

- `Ftd2xxSerialException`
  - "Port must be configured before it can be used"

  - "Could not find devices 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 reset 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: ..."

The `PyserialDriver` module in the `serialwrapper` package can throw the following exceptions:

- `Exception`
  - "Buffers don't match"
- `Serial.SerialException`
  - "WriteFile failed..."
  - "write failed: ..."
- `PortNotOpenError`
- `TypeError`
  - "Expected str or bytearray, 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: ..."

The `usbxserialutil` **module in the** `serialwrapper` **package can throw the following exceptions:**

- `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: ..."
- `UsbxSerialException`
  - "Cannot set number of devices"

The `usbxserialwin32` **module in the** `serialwrapper` **package can throw the following exceptions:**

- `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 timeouts: ..."
  - "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: ..."

1.6.3 Thrown by `snapconnect`

The `auth_digest` **module in the** `snapconnect` **package can throw the following exceptions:**

- `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: ..."

The `dispatchers` module in the `snapconnect` package can throw the following exception:

- `TypeError`
    - "Unknown descriptor type"

The `LicenseManager` module in the `snapconnect` package can throw the following exception:

- `LicenseError`
    - "... license file has expired"
    - "Your license is invalid..."
    - "Unable to load license file..."

The `listeners` module in the `snapconnect` package can throw the following exception:

- `ValueError`
    - "Unknown serial type"

The `snap` module in the `snapconnect` package can throw the following exceptions:

- `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"
    - "Tornado ioloop not supported"
    - "You must be accepting TCP connections to call accept_sniffer()"
- `TypeError`
    - "Unknown Hook Type"
    - "Invalid callback"
    - "Invalid connection parameter provided"
- `ValueError`

  - "Unknown encryption type specified: ..."
  - "auth_info is not callable"
  - "Serial interface type must be an integer"
  - "Unsupported serial interface type"

The `snaptcp` module in the `snapconnect` package can throw the following exceptions:

- `socket.error`
  - "Did not receive a valid lookup result"
- `ValueError`
  - "SSL support was not found"

1.6.4 Thrown by `snaplib`

The `DataModeCodec` module in the `snaplib` package can throw the following exception:

- `DataModeEncodeError`
  - "Packet is greater than 255 bytes"
  - "The packet is too large to encode ..."

The `MeshCodec` module in the `snaplib` package can throw the following exception:

- `MeshError`
  - "encode function does not yet support message type ..."
  - "Packet is greater than 255 bytes"
  - "Mesh Routing message too large to encode ..."

The `RpcCodec` module in the `snaplib` package can throw the following exceptions:

- `RpcError`
  - "Do not receive a list of arguments"
  - "Source Address must be 3 bytes"
  - "Source Address must be between xFFxFFxFF and x00x00x00"
  - "No source address was set"
  - "No destination address/group was set"
  - "dmcast_dstAddrs length must be a multiple of 3 (including 0)"
  - "Original TTL for multicast must be greater than 0 and less than 256"
  - "Delay Factor for directed multicast must be between 0 and 255"
  - "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"
  - "Packet cannot be a Directed Multicast without being a Multicast Packet first"
  - "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 bytes"
  - "Function/Args too large to encode ..."
- `TypeError`
  - "Unsupported type ..."
  - "String length is greater than 255"
  - "Integer is out of range"
- `WrongArgCount`

The `SnappyUploader` **module in the** `snaplib` **package can throw the following exception:**

- `AlreadyInProgressError`
  - "There is currently a **SNAPpy** upload already in progress"

The `TraceRouteCodec` **module in the** `snaplib` **package can throw the following exception:**

- `TraceRouteEncodeError`
  - "The data is too large to encode"
  - "Packet is greater than 255 bytes"

1.7 Sniffer Descriptors

SNAP descriptors are objects that are passed to local and remote sniffer callbacks. These descriptors represent packets that are received and transmitted by a **SNAPconnect** instance. This section lists the supported descriptors, their meanings, and their attributes.

The descriptors passed to sniffer functions are NOT over-the-air packets. These descriptors only represent packets that are received and transmitted over **SNAPconnect** serial and TCP connections.

For more about sniffer operations, refer to the *accept_sniffer()*, *connect_tcp()*, and *start_sniffer()* functions.

Tip: Because not every descriptor type supports every attribute, you will want to use the Python builtin function `isinstance(descriptor, descriptor_type)` to verify the descriptor type prior to accessing its attributes.

1.7.1 Common Attributes

All packet descriptors share several attributes in common:

- `raw` : A string of decrypted bytes representing the undecoded packet.
- `rx` : True if the packet was received by the **SNAPconnect** instance running the sniffer.
- `tx` : True if the packet was transmitted by the **SNAPconnect** instance running the sniffer.

1.7.2 Data Mode Packets

`snap.DataModeDescriptor`

`DataModeDescriptor` represents received and transmitted (originated or forwarded) data mode packets.

Attributes

- `data` : The data being carried by the packet.
- `dstAddr` : A three-byte string representing the address of the immediate destination of the packet (subject to mesh routing) or a two-byte multicast group (for a multicast data mode packet).
- `final_dst_addr` : A three-byte string representing the address of the final destination of the packet (unicast only).
- `isMulticast` : True if the packet is multicast or False if it is unicast.
- `seq` : The sequence number of the packet.
- `srcAddr` : A three-byte string representing the **SNAP** address of the original source of the packet.
- `ttl` : The number of hops remaining for this packet (multicast only).

See also:

- *hooks.HOOK_STDIN* for how data mode packets are normally received.
- *data_mode()*
- *mcast_data_mode()*

1.7.3 Mesh Routing Packets

`snap.MeshDescriptor`

`MeshDescriptor` represents received and transmitted mesh routing packets.

Attributes

- `additionalAddresses` : An optional list of additional discovered/errored addresses.
- `hopLimit` : The number of hops remaining for this packet.
- `msgId` : A single character representing the type of mesh packet. `Q` is used for route requests, `P` is used for route replies, and `E` is used for route errors.
- `source` : A three-byte string representing the address of the immediate source of the packet.
- `originator` : A three-byte string representing the address of the node that created the packet.
- `originatorDistance` : The number of hops traveled from the originator.
- `target` : A three-byte string representing the address of the target for the routing packet.
- `targetDistance` : The number of hops until the target is reached.

1.7.4 Undecoded Packets

`snap.RawDescriptor`

`RawDescriptor` may show up in cases where a packet could not be decoded and broken into the correct components. A raw descriptor contains no additional information outside of the attributes common to all descriptors.

1.7.5 RPC Packets

`snap.RpcDescriptor`

`RpcDescriptor` represents received and transmitted RPC packets.

Attributes

- `args` : A tuple of arguments to be passed to the RPC function.
- `dstAddr` : A three-byte string representing the address of the immediate destination of the packet (subject to mesh routing) or a two-byte multicast group (for a multicast packet).
- `final_dst_addr` : A three-byte string representing the address of the final destination of the packet (unicast only).
- `funcName` : Name of the RPC function to be called.
- `isMulticast` : True if the packet is multicast or False if it is unicast.
- `last_hop_addr` : A three-byte string representing the address of the immediate source of the packet.
- `seq` : The sequence number of the packet.
- `srcAddr` : A three-byte string representing the address of the original source of the packet.
- `ttl` : The number of hops remaining for this packet (multicast only).

See also:

- *rpc()*
- *mcast_rpc()*
- *dmcast_rpc()*

1.7.6 Traceroute Packets

`TraceRouteDescriptor` represents received and transmitted traceroute packets.

Attributes

- `dstAddr` : A three-byte string representing the address of the immediate destination of the packet.
- `elapsed` : The elapsed time, in milliseconds, to reach the target node. (This time does not include the first hop in the path, which will typically be trivial for serial connections but can be significant when **SNAPconnect** performs a traceroute over a TCP bridge.)
- `final_dst_addr` : A three-byte string representing the address of the final destination of the packet.
- `isMulticast` : True if the packet is multicast or False if it is unicast.

- `msgId` : A single character representing the type of traceroute packet. `Q` is used for traceroute queries and `P` is used for traceroute replies.
- `orig_src_addr` : A three-byte string representing the address of the original source of the packet.
- `records` : A list of traceroute records with 'address' and 'linkQuality' attributes.
- `srcAddr` : A three-byte string representing the address of the immediate source of the packet.

See also:

- *traceroute()*
- *hooks.HOOK_TRACEROUTE* for information on how traceroute packets are received.

CHAPTER

TWO

RELEASE NOTES

2.1 Release 3.7.1

Released May 23rd, 2017

- Internal fixes

2.2 Release 3.7.0

Released February 9th, 2017

- Internal fixes

2.3 Release 3.6.1

Released October 21st, 2016

2.3.1 Bugs Fixed

- Mesh sequence numbers are properly randomized on startup.

2.4 Release 3.6.0

Released October 20th, 2016

As of this release, **SNAPconnect** is licensed under the Synapse SDK License.

2.4.1 Bugs Fixed

- vmStat now properly handles dynamic variable requirements based on status called.

2.4.2 New Features

- Added support for RF220SU-EU and SM220UF1-EU modules.

2.5 Release 3.5.4

Released September 23rd, 2015

Cumulative changes made to the **SNAPconnect** package since version 3.2.0 include bug fixes, new features, and changes and additions that allow **SNAPconnect** to use newer features of our **SNAPcore** product.

2.5.1 Bugs Fixed

- Fixed packet size when packet-CRC and extra-CRC were both enabled, script was not being uploaded.

2.5.2 New Features

- Added ability to send and receive Directed Multicast RPCs via `dmcastRpc()`
- Added new `getInfo()` options related to `dmcastRpc()`
- Added ability to perform topology polling via `vmStat(12)`
- Added ability to update NV parameters with an optional bitmask via:

```
``save_nv_param(id, integer_value, optional_integer_bitmask)``
```

2.5.3 Deprecated

- XML-RPC server
- Bluetooth Low Energy
- PyPy
- Python 2.5
- iOS

See also:

- gateways/snapconnect/index - installation, best practices and example projects

PYTHON MODULE INDEX

INDEX

S

T

U