Users Guide and API Reference

SNAP Reference

Copyright 2008-2024 Synapse Wireless, All Rights Reserved. All Synapse products are patent pending.
Synapse, the Synapse logo and SNAP are all registered trademarks of Synapse Wireless, Inc.

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

CONTENTS

1 Users Guide 3

2 API Reference 51

Index 129

This section includes a deep dive into SNAP networking and how to write SNAPpy scripts that run on the SNAP Modules and interact with your hardware.

CHAPTER

ONE

USERS GUIDE

This section covers **SNAP** networking and how write **SNAPpy** scripts that run on the **SNAP** Modules and interact with your hardware.

Topics

1.1 Release Notes

1.1.1 Release 2.8.2

Released April 4th, 2017

Bugs Fixed

- Modified topology `vmStat()` response so that it is sent using the same type of RPC that was received.

1.1.2 Release 2.8.1

Released October 6th, 2016

Bugs Fixed

- Corrected an issue which could cause an OTA upgrade to fail.
- Corrected an edge-case issue which could cause memory corruption.
- Corrected an edge-case issue in the `writeChunk()` logic.

New Features

- Added support for new module build for EU power levels: SM220UF1_EU. Calling `getInfo(3)` returns `33`.

Enhancements

- `tellVmStat()` response will reply using a directed multicast message if it was called via directed multicast.

1.1.3 Release 2.7.2

Released June 7th, 2016

Bugs Fixed

- Corrected an issue in random backoff where modulo bias distorted the frequency distribution of time slot selection.
- Corrected an issue where memory was being overwritten during packet storms.
- Corrected an issue where RF220SU turbo bit was not being set properly by vendor bits.
- Corrected an issue where NULL pointers were being de-referenced.
- Corrected an issue where sleep return values were returning incorrect values when short wakeup pulse occurred > 8s.
- Changed the **External Power Amp Feature Bit** (0x0010) to be OFF by default for the ATmega128RFA1 chip build.

New Features

- Added support for new module build for EU power levels: RF220SU_EU. Calling `getInfo(3)` returns `31`.

Enhancements

- Randomize initial mesh sequence number so that RREQs don't get dropped due to duplicates after subsequent reboots.

1.1.4 Release 2.7.1

Released March 25th, 2016

Cumulative changes made to **SNAPcore** since version 2.6.9 add a number of new features, as well as correcting issues encountered since the 2.6.9 release.

Bugs Fixed

- Corrected an issue where very short pulses to wake a sleeping node caused the return value from the sleep command to be incorrect.
- Corrected an issue where the byte list elements could not be incremented in place.
- Corrected an issue where deleting a byte list element using subscript -1 failed.
- Corrected an issue where oversized byte lists gave an incorrect error message.
- Corrected an issue where non-integer values produced troublesome results when a third "bitmask" parameter was included on *saveNvParam()* function calls.
- Corrected an issue where serial traffic to a node with no script could put the node into an unresponsive state.
- Corrected an issue where unresolved iterators could result in a leaked string buffer.

New Features

- Added support to accommodate calls to C functions from **SNAPpy** functions.
- Added getInfo(29) for the C compiler ID and getInfo(30) for the build ID hash, in order to support the new C functionality.
- Added *errno()* enumerations 22 and 23 for EXCEEDED_C_DATA_SIZE and INVALID_C_METADATA, both of which are possible if you compile a script file for one firmware version but are trying to use it with a different version. Additionally, the C functionality provides access to the errno() return values as a means to define your own error codes.
- Added new built-in function *dmCallout()*, which performs similar to the *callout()* built-in but uses directed multicasts rather than unicasts for the message delivery.
- Added support for new module types: **SN220 SNAPstick**, **RF220SU**.

Enhancements

- Timing for the *pulsePin()* function is more accurate.
- Moved the **Enlarged Route Tables Feature Bit** from NV 64 (platform-specific feature bits, which it shared with the ATmega128RFA1's "Turbo Mode") to NV 11 (feature bits, bit 0x800). Clearing this bit provides a route table with 10 entries. Setting it provides a table with 100 entries. Changes require a node reboot.
- Updated objRepr so that str(bytelist) matches the spacing of str(tuple), with no spaces between elements, for consistency and for more concise return values and printed values.
- Updated the Collision Avoidance algorithm to begin its "random backoff" timing based on the timing of the request to send a packet rather than the time when the last packet was received. (Note that for timing-sequenced replies to directed multicasts, the random backoff provided by collision avoidance is not applied.)
- The shifting window of relevance for multicast mesh sequence numbers that prevents multiple actions on a given message have been applied to the messaging used to establish unicast mesh routes.
- Adjusted the power output levels for the new **RF220SU** module to match restrictions required for FCC certification.

Deprecated

- With the release of version 2.6.2, **SNAP** only supports ATMEL-based modules and boards.

1.1.5 Release 2.6.2

July 6th, 2015

SNAP 2.6 adds a number of new features to the **SNAPcore**, as well as correcting issues encountered since the 2.5.6 release. With the release of version 2.6.2, SNAP no longer supports legacy modules and boards. Be aware that a change for a chip also affects any modules or boards that are based on that chip. So, ATmega128RFA1 also implies RF200, SS200, RF266, SM200, SM220 and RF220.

New Features

- Added support for "for" loops.
- Added support for string multiplication.
- Added support for "in"/"not in" for strings and tuples.
- Added getinfo() calls that report the bank and address of the SNAPpy script.
- Added NV parameters for default radio and serial port rates to support running them at non-standard rates.
- Added an NV parameter to tune collision avoidance and increase or decrease the window in which a node might respond.
- Added the ability to perform topology polling even with no SNAPpy script loaded. Note that this feature is built-in to the SNAP firmware, and is usable even if no SNAPpy script is loaded.
- Added support for directed multicast. This allows multicast instructions to be sent to a specified list of nodes, rather than broadcasting to all nodes by group.
- Added limited support for dynamic mutable lists, subject to RAM limitations of the underlying hardware. This includes support for methods to emulate Python's byte arrays.

Enhancements

- The reboot() function now accepts an optional parameter that delays a reboot for a number of milliseconds. This could, for example, allow a more graceful platform restart.
- Added optional parameter to setChannel() which allows changing of Network ID as well as channel.
- Added an optional third parameter to saveNvParam() to make it easier to manipulate individual bits in a single operation.
- Altered saveNvParam() to allow encryption keys to be changed without the need for a reboot.
- Expanded the getInfo() function to tell if a script is being run for the first time.
- The number of dynamic strings was increased at the expense of packet buffers. This required the addition/modification of getInfo () commands to retrieve counts of length 1, 16, 126, and 255 byte string buffers.
- Raised the number of global variables usable in SNAPpy scripts from 128 to 255.
- Added tiny and large string buffers, allowing dynamic strings to be as small as 1 byte, and as large as 255 bytes. This reorganized available strings into "tiny," "small," "medium," and "large."

Bugs Fixed

- Corrected a bug where a unit in "wildcard mode" replaced original network IDs with its own when repeating multicast packets.
- The increase in the number of public functions in SNAP Release 2.6 eliminated the issue of non-public functions causing hooks to invoke the wrong script at run time. In Release 2.6 you can safely hook a non-public function without concern about the incorrect function executing at run-time.
- SNAP 2.5.6 made SNAP nodes incompatible with Portal's "port scan" feature. SNAP 2.6 restores compatibility, and Portal can once again "detect" the nodes.

1.1.6 Release 2.5.6

Released February 17th, 2015

New Features

- CPU_IDLE – Behind the scenes, SNAP was constantly checking the radio and serial ports looking for incoming data to be processed. Starting in this version, if SNAP has checked all of the possible sources of incoming data and found nothing to be processed, it will use the CPU's built-in "idle" capability to wait for the next interrupt. This reduces the power consumption of SNAP nodes that are not processing a lot of traffic, which can increase battery life. Note that if your SNAP node is being kept busy (for example, your application sends a lot of radio and/or serial traffic) then you will not see much benefit from this enhancement.
- type() built-in added to SNAPpy. The ability to tell (at run-time) if a variable was (for example) a String versus an Integer was added to the SNAPpy Virtual Machine. As a quick example of where this can come in handy, the loadNvParam() function can reload a previously saved value, but until now there was no easy way to verify it's TYPE. You could do "is None" and "is not None" checks before (and you still can) but the type() function is much more versatile.
- SNAP now allows you to use different multicast "packet forwarding" settings on the serial port versus the radio. In previous versions of SNAP, NV Parameter 6 controlled which multicast groups were forwarded on both the radio and the serial port. You can still choose to do that, but now there is an additional NV Parameter 78 that when set gives the serial port its own settings, and means that the multicast group bitmask in NV Parameter 6 apply only to packets forwarded over the radio. This allows you to do things like "only forward group 0x0002" packets over the radio, and only forward group 0x0004 over the serial port". To have NV Parameter 6 control both serial and radio forwarding, let NV Parameter 78 to None.
- Includes a "SNIFFER Firmware" build for the ATmega1284RFR2.
- PACKET_CRC – similar to the previous RPC_CRC feature introduced in SNAP 2.4.19, the PACKET_CRC feature added an additional software CRC to the radio packets. This was added to address issues with "packet storms" seen out in the field. Like RPC_CRC, enabling PACKET_CRC costs you two bytes of packet space (the additional CRC takes up two bytes). You can enable both CRCs if you wish, but this will cost you 4 bytes of packet space total. Here is how PACKET_CRC differs from RPC_CRC:

Aspect	PACKET_CRC	RPC_CRC
Enabled by Feature Bit (look at NV #11)	0x0400	0x0100
Applies to	All packet types, including RPC packets	RPC packets only (both unicast and multicast)
Calculated from	The entire packet, including the header	The packet payload only
Applied to packets sent or received	Over the radio only	Radio and Serial (both)

- I2C_RESTART – Prior to version 2.4.37, SNAP could only work with devices that used the "I2C_START, I2C_STOP, I2C_START, I2C_STOP" hardware handshake sequence for back-to-back commands (for example, an i2cWrite() to specify the data to read, followed by an i2cRead() to capture that data). Some I2C devices instead use a "I2C_START, I2C_RESTART, I2C_STOP" hardware handshake sequence. SNAP version 2.4.37 introduces an optional trailing parameter to the i2cWrite() function. When the optional parameter is provided and its value is True, SNAP will end the i2cWrite() command such that the beginning of an I2C_RESTART is created. The following i2cRead() will complete the I2C_RESTART (instead of generating an I2C_START). When the optional parameter is omitted, or is provided but its value is False, then the normal I2C_STOP sequence is generated. This enhancement allows SNAP to work with a wider range of I2C device.
- SNAP has always had a "line mode" for HOOK_STDIN, but if you received too many characters before the receipt of a Carriage Return or Line Feed character, the system would print an error message and discard the data. Now even if you have specified "line mode" which technically means "don't send the data until you get a CR or LF" the system will push what it has received so far if the buffer fills up. This makes the feature more useful. To support this new behavior, an new getStat() option has been added, getStat(18). By calling this function, your SNAPpy script can check and see why the HOOK_STDIN handler has been called.
- The SM220 was the first SNAP Module to boast two onboard antennas – a "meandering F" and a "U.FL" connector. Support for software controlled antenna selection was added in this version (refer to NV Parameter 64).
- The hardware inside the ATMEL radios includes "trim capacitors" that can be selectively enabled. You can now specify an alternate radio trim setting via an NV Parameter (#63 – NV_ALT_RADIO_TRIM_ID) to take advantage of this hardware capability. Most customers will never need to use this, but if for some reason your units are running higher in frequency then you can change this NV Parameter from its default value of 0 to enable 1-15 steps of additional capacitance (which will lower the radio frequency). The hardware does not have the ability to adjust the frequency in the other direction. (You cannot use the internal trim to raise the frequency.)
- Sleep mode 2 (sleep(2, ticks)) added. This new sleep mode uses the "MAC Symbol Counter" inside the radio as a timebase, and provides finer-grained sleep durations.
- The "moveable I2C" feature was back-ported from the STM32W108xB version of SNAP. Now if you need to connect an I2C peripheral to a different pair of pins, just specify the alternate SCL and SDA pins in the i2cInit() function call. These two new parameters are optional, you do not have to change your existing scripts unless you want to leverage this new capability. Calling the i2cInit() function without the new optional parameters causes the original pin assignments to be used.

Enhancements

- The behavior of the "is" clause in SNAPpy was changed to more closely match what full "desktop" Python does.
- Now if you call initUart() on a UART that is being used by the Packet Serial feature, the Packet Serial state machine gets re-initialized too. (There were users who were changing their serial port baud rates on the fly and getting poor results.)
- Robustness of the NV Parameters storage area when performing "page swaps" in the presence of power outages or system resets was improved.
- Replaced the original software-based implementation of AES-128 with one that utilized the internal "crypto engine" of the ATmega128xxx processors. This enhancement applied to all of the ATMEL chips.
- The getLq() built-in was changed to return a "snapshotted" value taken at the time the radio packet was received, instead of returning a "live" reading. This brings the ATMEL platforms in line with the rest of the SNAP platforms.
- Overall current consumption was reduced slightly by removing the initialization of some unused hardware. The "go to sleep" and "wake back up" code paths were also optimized for speed.

Bugs Fixed

- Fixed an issue with calling SNAPpy functions with too many parameters that in certain situations could lead to string buffer leaks.
- Corrected issue with SNAPpy script upload with RF100 and MC1321x.
- Reduced txPwr() levels for "worldwide" from 4 to 2 in SM220.
- It was noticed that the chip was sometimes pausing for about 10 milliseconds at a 5 milliamp current draw before fully entering sleep mode. Since many SNAP applications are battery powered, this short period of higher power consumption was removed. A version of the "DMX" variant of SNAP for the SM220 module was created and added to the set of firmware images.
- Comparison of SNAPpy integers (signed 16-bit) was improved. Prior to this version, a comparison like "20000 > -12768" would return False instead of True due to 16-bit wrap-around. This has been corrected. Note that this might require changes to your existing SNAPpy scripts if you were relying on the previous (incorrect) behavior.
- The Manufacturing Date was not being preserved through a Factory Default (fixed)
- In an exhaustive review of the SNAPpy Virtual Machine, numerous "dynamic string leaks" were identified and corrected.
- The default Feature Bits for the RF266 were changed from "enable both UARTs" (this is the default used by all other ATMEL-based platforms) to "enable UART1 only". This was done because the first UART (UART0) is not brought out to any of the RF266 pins.
- It was discovered that the internal FLASH of the 'RFR2 chips was not 100% compatible with the FLASH of the original 'RFA1 chip. This required the FLASH "write" routines to be re-written, resulting in the 'RFR2 chips gaining their own unique Boot Loader. The 'RFR2 chips boasted an internal feature ATMEL dubbed "SRT – Smart Radio Technology." This was supposed to enable a 5 milliamp "radio receive" mode. A lot of effort went into this, but we were unable to get the chips to reliably enter and stay in this mode. (You will likely see some current savings on a 'RFR2 but not as much as we had hoped).
- It was discovered that the sleep() function could wake up early due to the internal MAC Symbol Counter rolling over, as well as the internal 1 millisecond clock interrupt occurring. Both of these issues were corrected, so that the unit would remain asleep for the requested duration. As part of the above sleep()

work, the sleep software was recalibrated for higher accuracy (the test case for this work was a 12 hour sleep duration).

1.2 SNAPpy Language

The **SNAPpy** language is a subset of Python, with a few extensions to better support embedded real-time programming.

Note

Legacy tools like **Portal** and **SNAPbuild** support a flavor of **SNAPpy** based on Python 2. Our current tools (including **SNAPcompiler**) are based on Python 3. Scripts written to work with **Portal** or **SNAPbuild** which rely on Python 2-only syntax will fail to compile w/ **SNAPcompiler**, and scripts written to work with **SNAPcompiler** which rely on Python 3-only syntax will fail to compile with **Portal** and **SNAPbuild**.

SNAPpy images (SPY files) generated with the legacy tools will continue to work. The **SNAPpy** VM in **SNAPcore** has not changed, just the tooling around compiling **SNAPpy** scripts into **SNAPpy** images.

Topics

1.2.1 Statements

Statements must end in a newline:

```
print("I am a statement")
```

The # character marks the beginning of a comment:

```
print("I am a statement with a comment")  # this is a comment
```

Indentation is used after statements that end with a colon (`:`):

```
if x == 1:
    print("Found number 1")
```

Indentation is significant. The amount of indentation is up to you (4 spaces is standard for Python), but you must be consistent. The indentation level is what distinguishes blocks of code, determining which code is part of a function or which code repeats in a while loop, for example:

```
print("I am a statement")
    print("I am a statement at a different indentation level")  # this is an error
```

Branching is supported via `if`, `elif`, and `else`:

```
if x == 1:
    print("Found number 1")
elif x == 2:
    print("Found number 2")
else:
    print("Did not find 1 or 2")
```

(continues on next page)

(continued from previous page)

```
y = 3 if x == 1 else 4  # Ternary form is acceptable
```

Looping is supported via `while`:

```
x = 10
while x > 0:
    print(x)
    x = x - 1
```

Looping is also supported via `for`:

```
myTuple = ("A", True, 3)

# for will step through tuples or byte lists
for element in myTuple:
    print(element)

# for will step through iterators returned by range
# the following prints "012" (note that SNAPpy does not insert spaces)
for number in range(3):
    print(number)

# for will step through strings
for letter in myStringVariableOrConstant:
    if letter == "Z":
        print("I found a Z")
```

1.2.2 Identifiers

Identifiers are case sensitive:

```
X = 1
x = 2
```

Here, `X` and `x` are two different variables.

Identifiers must start with a non-numeric character:

```
x123 = 99   # OK
123x = 99   # not OK
```

Identifiers may only contain alphanumeric characters and underscores:

```
x123_percent = 99   # OK
x123% = 99          # not OK
%^ = 99             # not OK
```

The following is a list of reserved keywords supported in the **SNAPpy** language which cannot be used as identifiers:

and	def	else	global	is	or	True
break	del	from	if	None	pass	return
continue	elif	False	import	not	print	while
for	in					

The following identifiers are reserved, but they are not yet supported in **SNAPpy**: `as`, `assert`, `class`, `except`, `exec`, `finally`, `lambda`, `raise`, `try`, `with`, `yield`.

1.2.3 Functions

You define functions using `def`:

```
def sayHello():
    print("hello")

sayHello()    # calls the function, which prints the word "hello"
```

Functions can take parameters:

```
def adder(a, b):
    print(a + b)
```

Note

Unlike Python, **SNAPpy** does not support optional/default arguments. If a function takes two parameters, you must provide two parameters. Providing more or fewer parameters gives an undefined result. There are a few built-in **SNAPpy** functions that do allow for optional parameters, but user-defined functions must always be called with the number of parameters defined in the function signature.

Functions can return values:

```
def adder(a, b):
    return(a + b)

print(adder(1, 2))  # would print out "3"
```

Functions can do nothing:

```
def placeHolder(a, b):
    pass
```

Functions cannot be empty:

```
def placeHolder(a, b):
    # ERROR! - you have to at least put a "pass" statement here
    # It is not sufficient to just have comments
```

This is also true for any code block, as might be found in a `while` loop or a conditional branch. Each code block must contain at least a `pass` statement.

Functions can change:

```
def sayHello(arg1, arg2, arg3):
    print("hello")

def sayHello():
    print("hello, world")

sayHello()   # calls the second function, which prints the word "hello, world"
```

If you have two function definitions that define functions with the same name, even with different parameter signatures, only the second function will be available. You cannot overload function names in **SNAPpy** based on the number or type of parameters expected.

1.2.4 Variables

There are several types of variables:

```
a = True                        # Boolean
b = False                       # Boolean

c = 123                         # Integer, range is -32768 to 32767
d = "hello"                     # String, size limits vary by platform
e = (None, True, 2, "Three")    # Tuple - usable only as a constant in SNAPpy

f = None                        # Python has a "None" data type
g = startup                     # Function
h = xrange(0, 10, 3)            # Iterator (introduced in SNAP 2.6)
i = [1, 1, 2, 3, 5, 8]          # Byte List (introduced in SNAP 2.6)
```

In the above example, invoking `g()` would be the same as directly calling `startup()`. You can use the *type()* function to determine the type of any variable in **SNAPpy**.

Variables can change their type on the fly:

```
x = 99              # variable x is currently an integer (int)
x = False           # variable x is now a Boolean value of False
x = "hello"         # variable x is now a string (str)
x = (x == "hello")  # variable x is now a Boolean value of True
```

String variables can contain binary data:

```
A = "\x00\xFF\xAA\x55"   # The "\x" prefix means hexadecimal character
B = "Pi\xe1"             # This creates a string of length 3
```

Byte lists allow for updates without rebuilding:

```
A = [7, 8, 9]
A[2] += 1
```

Variables at the top of your script are global:

```
x = 99   # this is a global variable

def sayHello():
    print("x=", x)
```

Variables within functions are usually local:

```
x = 99  # this is a global variable

def showNumber():
    x = 123  # this is a separate local variable
    print(x)  # prints 123
```

Unless you explicitly say you mean the global one:

```
x = 99  # this is a global variable

def showGlobal():
    print(x)  # this shows the current value of global variable x

def changeGlobal():
    global x # because of this statement
    x = 314  # this changes the global variable x

def changeLocal():
    x = 42   # this statement does not change the global variable x
    print(x)  # will print 42 but the global variable x is unchanged
```

Creating globals on the fly:

```
def newGlobal():
    global x   # this is a global variable, even without previous declaration
    x = x + 1  # ERROR! - variables must be initialized before use

    if x > 7:  # ERROR! - variables must be initialized before use
        pass
```

These two statements are not errors if some other function has previously initialized a value for global variable `x` before the `newGlobal()` function runs. Globals declared in this way have the same availability as globals explicitly initialized outside the scope of any function.

> **Note**
>
> On RAM-constrained devices, **SNAPpy** scripts limit the number of concurrent local variables and system global variables. See platform-specific for more information on limits.

1.2.5 Operators

The usual comparators are supported:

```
if 2 == 4:
    print("something is wrong!")

if 1 != 1:
    print("something is wrong!")
```

(continues on next page)

(continued from previous page)

```
if 1 < 2:
    print("that's what I thought")
```

Symbol	Meaning
==	Is equal to
!=	Is not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The usual math operators are supported:

```
y = m * x + b
z = 5 % 4          # z is now 1
result = 14 / 8   # result is now 1 -- integer math only
```

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

The usual bitwise operators are supported:

```
a = 0xAAAA & 0x5555   # a is now 0
o = 0xAAAA | 0x5555   # o is now 0xFFFF
x = 0xAFAF ^ 0xFAFA   # X is now 0x5555
n = ~ 0xAAAA          # n is now 0x5555
l = 0x0A0A << 3       # l is now 0x5050
rp = 0x50A0 >> 3      # rp is now 0x0A14
rn = 0xA050 >> 3      # rn is now 0xF40A
```

Symbol	Meaning
&	Boolean And
|	Boolean Or
^	Boolean Xor
<<	Shift Left
>>	Shift Right
~	Not (Complement)

Note

If the high bit is set when you shift right (indicating a negative number), then the high bit is imposed on the number after each shift occurs. This results in at least the number of bits you've shifted all being

set at the high end. If you need to shift without preserving the high (negative) bit, you must clear the bit yourself (e.g., `number &= 0x7FFF`) after the first bit shift.

The usual Boolean functions are supported:

```
Result = True and True              # Result is True
Result = True and False             # Result is False
Result = True and not False         # Result is True
Result = True and not True          # Result is False
Result = False and True             # Result is False
Result = False and False            # Result is False
Result = True or True               # Result is True
Result = True or False              # Result is True
Result = not True or not False      # Result is True
Result = not (True or not False)    # Result is False
Result = False or True              # Result is True
Result = False or False             # Result is False
```

Symbol	Meaning
and	Both must be True
or	Either can be True
not	Boolean inversion (not True == False)

Ternary `if` is supported:

```
x = "A is bigger" if a > b else "B is bigger"
```

Note

SNAPpy does not support the floor (`//`) and power (`**`) Python operators.

Slicing is supported for byte list, string and tuple data types. For example:

```
x = "ABCDE"
x[1:4]   # returns "BCD"
```

Concatenation is supported for string data types. For example:

```
x = "Hello"
y = ", world"
x + y   # "Hello, world"
```

String multiplication is supported:

```
3 * "Hello! "    # "Hello! Hello! Hello! "
```

Subscripting is supported for byte list, string, and tuple data types. For example:

```
x = ('A', 'B', 'C')
x[1]   # 'B'
```

1.2.6 Data Types

SNAPpy supports many of the standard Python data types.

You can use **SNAPpy**'s *type()* function to identify a variable's type.

NoneType

`None` is a valid value in Python, as the only entity of type NoneType. Comparisons of a `None` value as if it were a Boolean will return `False`:

```
n = None

if n:
    print("This will never print.")
```

Setting string or byte list variables to `None` will release that buffer for use elsewhere.

Integer

SNAPpy integers are 16-bit signed values ranging from -32768 to 32767. If you add 1 to 32767, you will get -32768. You can specify integers using decimal notation or hexadecimal notation:

```
i = 0x1c2c
```

Normal Python mathematical operations apply to integers:

```
a = 39 + 3          # a = 42
a += 5              # a = 47
s = 48 - 6          # s = 42
s -= 5              # s = 37
m = 6 * 7           # m = 42
m *= 3              # m = 126
d = 551 / 13        # d = 42
d /= 8              # d = 5
r = 757 % 15        # r = 42
r %= 10             # r = 2
e = 13482 & 20311   # e = 1026: 00110100,10101010 & 01001111,01010111 = 00000100,00000010
e &= -1286          # e = 2:    00000100,00000010 & 11111010,11111010 = 00000000,00000010
o = 10 | 7          # o = 15:   00000000,00001010 | 00000000,00000111 = 00000000,00001111
o |= 240            # o = 255:  00000000,00001111 | 00000000,11110000 = 00000000,11111111
l = 10 << 2         # l = 40
l = 16384 << 1      # l = -32768
h = 32767 >> 2      # h = 8191
h = -32768 >> 2     # h = -8192 !!! Might not be as expected !!!
```

SNAPpy does not generate an error if you divide by zero. The result of that division will be zero:

```
42 / 0 = 0
```

Note that the division is integer division, taking the *floor* value of the division:

```
999 / 1000 = 0
```

The Python *floor* (//) operator is not implemented. When negative numbers are involved as either the divisor or dividend, the value will be the quotient value closest to zero. For example

```
-20 / 3 = -6
-20 / -3 = 6
20 / 3 = 6
20 / -3 = -6
```

This is different from the implementation of the floor operator in pure Python, where `-20 // 3 = -7`, as it takes the next lowest integer rather than the integer with the lowest absolute value.

The *modulo* (%) operator returns the remainder after an integer division. Again, the implementation varies from the modulo implementation in pure Python. In **SNAPpy**, the values to expect are:

```
-20 % 3 = -2
-20 % -3 = -2
20 % 3 = 2
20 % -3 = 2.
```

Pure Python gives different results:

```
-20 % 3 = 1
-20 % -3 = -2
20 % 3 = 2
20 % -3 = -1.
```

Bitwise and (&) and *bitwise or* (|) operators function as expected, as does the *left-shift* (<<) operator. Beware when using the *right-shift* (>>) operator on an integer with the high bit set, though, as the high bit is what marks the number as negative, and after the shift that bit will be reapplied. The Python *power* (**) operator is not implemented.

String

Strings are not null-terminated in **SNAPpy**, so they can contain any of the 256 possible values for each character, including (`\x00`). **SNAPpy** treats static and dynamic strings differently:

Static string
An immutable constant in your code. Each static string has a maximum size of 255 bytes.

Dynamic string
Created when you assign a value to a string while a script is running or attempt to reassign a value to a string declared outside a function. The maximum size of dynamic strings (up to 255 bytes) is platform specific. See platform-specific for more information.

Strings in **SNAPpy** (as in Python) are immutable, so you cannot change any characters within the string after it has been created. In order to "modify" a string, you must perform slicing operations on the string, creating a new dynamic string in the process.

Note

To manage all these dynamic strings, **SNAPpy** uses a collection of string buffers. You will need to understand *Memory Management* to make the best use of these resources. See platform-specific for more information about resource limitations.

You can use the `in` operator to determine whether a string contains a substring and the `for` statement to iterate through characters in a string:

```
for c in myString:
    if isSpecial(c):
        print("Look what I found: ", c)
```

A non-standard feature of **SNAPpy** strings is that if a string variable contains the name of a valid **SNAPpy** function, you can invoke the variable name as if it were the function. In the following example, passing any string matching the name of a valid function loaded on the device (such as `"random"`, `"getLq"`, or even a user-defined one) would cause that named function to be invoked immediately:

```
def runArbitrary(function):
    return function()
```

Function

In **SNAPpy**, as in Python, a function name is essentially a variable that points to a function. As such, another variable can be assigned to point to that function, too:

```
@setHook(HOOK_STARTUP)
def onStartup():
    global myRandom
    myRandom = random

def odd():
    return random() & 4094  # Clear last bit

def even():
    return random() | 1  # Set last bit

def setRandomMode(newMode):
    global myRandom
    if newMode == 1:
        myRandom = odd
    elif newMode == 2:
        myRandom = even
    else:
        myRandom = random
```

After a call to `setRandomMode()` to specify which character of random numbers should be returned, any future calls to `myRandom()` will return either an odd random number, an even random number, or an unspecified random number.

Note that saying `myRandom = odd` is an assignment of the `odd()` function to the `myRandom` variable. This is very different from saying `myRandom = odd()`, which would assign the return value of a call to the `odd()` function to the `myRandom` variable.

Both user-defined functions and built-in functions can be assigned to your variables. You then call the function by invoking the variable name followed by parentheses (which should contain any arguments the function requires). The function can be invoked directly from another function on the same device or by Remote Procedure Call (direct or multicast) from another device, which establishes the ability to have one multicast call cause different devices to run different functions.

Both public and non-public functions are only limited by available flash space. Testing has confirmed that more than 500 functions can be available on a device.

Boolean

A Boolean has a value of either `True` or `False`, which is case-sensitive. Comparisons of Booleans can be direct:

```
b = True
if b:
    print("This will print.")

if b == True:
    print("This will also print.")
```

Tuple

A tuple is an ordered, **read-only** container of data elements. Not only is the tuple immutable, but its contents are, too. You cannot even change any of the bytes in a byte list that is contained within a tuple. There are restrictions on printing of nested tuples. See the section on *Printing* below for more details.

Elements in a tuple can be of almost[1] any type available in **SNAPpy**, including nested tuples:

```
myTuple = (None, True, 2, "Three", ("Four", "in", "this", "tuple"), [5, 10, 15, 20, 25])
```

You can access tuple elements by stepping through the tuple with a `for` loop or by selecting individual elements. In the sample tuple above, `myTuple[3]` would be `"Three"` and `myTuple[4][0]` would be `"Four"`.

You can use the `in` operator to determine whether a tuple contains an element and the `for` statement to iterate through elements in a tuple:

```
for element in myTuple:
    print(element)
```

Warning

You cannot pass a tuple as an argument in a Remote Procedure Call (RPC), though you can pass any tuple element.

[1] Tuples cannot contain iterators.

Iterator

Iterators are generated using the *xrange()* function. Iterators defined in the global space of **SNAPpy** scripts are not supported. Typically, an iterator is not assigned to a variable but is used in-line:

```
for a in xrange(3):
    print(a)
```

However, it is possible to assign iterators to variables and pass them as parameters within a device:

```
def makeIterator(top):
    a = xrange(top)
    return sum(a)

def sum(anIterator):
    count = 0
    for a in anIterator:
        count += a
        return count
```

Warning

You cannot pass an iterator as an argument in an RPC.

Byte List

Byte lists provide some limited Python list functionality. A byte list is an ordered list of unsigned, one-byte integers. While byte lists and strings work from the same pool of buffers, the processing that you can perform on the data types varies. While strings in **SNAPpy** (as in Python) are immutable, byte list elements can be changed in place:

```
myList = [1, 2, 3, 4, 5]
myList[2] = 42  # Now list is [1, 2, 42, 4, 5]
```

This ability to modify a byte (or a slice of bytes) without having to rebuild the list allows for much faster processing than trying to perform the same functions using strings, and it does not require available buffers for processing the slicing.

Beginning in **SNAP 2.7**, the += operator is supported on elements within a byte list:

```
myList = [1, 2, 3, 4, 5]
myList[4] += 1  # Now list is [1, 2, 3, 4, 6]
```

You can define byte lists specifying literals or variables in square brackets:

```
myList = [1, 2, 3]
myInt = 4
myList = myList + [myInt] + [myInt, myInt]  # Now list is [1, 2, 3, 4, 4, 4]
```

You can also build up lists using list multiplication:

```
myList = [0] * 10  # Now list is [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

It is easy to convert between byte lists and strings:

```
myList = ["Byte list"]  # Now list is [66, 121, 116, 101, 32, 108, 105, 115, 116]
myString = chr(myList)  # Now myString = "Byte list"
toStr = str(myList)     # Now toStr = "[66,121,116,101,32,108,105,115,116]"
```

You can step through byte lists using `while` or `for` loops, and you can also use the `in` operator to determine whether a number is in your list. The `del` operator can be used to delete an element or range of elements from a list (by position) or to delete the entire list:

```
myList = [1, 2, 3, 4, 5]
del myList[2]     # myList = [1, 2, 4, 5]
del myList[1:3]   # myList = [1, 5]
del myList[0:2]   # myList = [], an empty list
del myList  # myList is now an unknown variable. type(myList) returns 31
```

Beginning in **SNAP 2.7**, `del` works with negative index values:

```
myList = [1, 2, 3, 4, 5]
del myList[-1] # myList = [1, 2, 3, 4]
```

In order to preserve RAM, **SNAP** firmware makes some decisions about where it stores global variables, locating them in flash memory rather than RAM when a device boots. If you will be modifying individual entries in a globally defined byte list, you need to be working with the variable in RAM rather than flash, so you must force **SNAPpy** to make a copy of the variable (consuming a buffer) first. The easiest way to do that is to slice the list into a new list, and the most efficient way to ensure that this happens once (and only once) is to include it in your hooked startup code:

```
myList = [1, 2, 3, 4, 5]

@setHook(HOOK_STARTUP)
def onStartup():
    global myList
    myList = myList[:]
```

Byte lists that are defined at run-time are limited in size by the available stack size in **SNAP**, which can vary based on the current state of your call structure and how many parameters have been passed, etc. This means that if you are building a larger list on-the-fly as a local variable, you may have to break the list into several chunks and add them together. (The **SNAPpy** data stack is on the order of 64 variables deep.)

> **Warning**
>
> You cannot pass a byte list as an argument in an RPC call; however, you can use `chr()` to convert the byte list to a string which can be passed.

Unsupported

SNAPpy does not support user-defined classes or the following Python data types:

- `float` – A float is a floating-point number with a decimal part.
- `long` – A long is an integer with arbitrary length (potentially exceeding the range of an int).
- `complex` – A complex is a number with an imaginary component.
- `list` – A list is an ordered collection of elements, excepting byte lists as described above.
- `dict` – A dict is an unordered collection of pairs of keyed elements.
- `set` – A set is an unordered collection of unique elements.

Note

While unsupported types cannot be used in **SNAPpy** scripts, they can still be used in a **SNAPconnect** application.

1.2.7 Modules

SNAPpy supports the import of user-defined and standard predefined Python source library modules:

```
from module import *                  # Supported
from module import myFunction         # Supported
from module import _myPrivateFunction # Supported
import module                         # Not supported
```

Any non-public functions contained in an imported module will not be included when using `from module import *`, but you can explicitly import non-public functions using `from module import _myPrivateFunction`.

1.2.8 Printing

SNAPpy also supports a `print` statement:

```
print("line 1")
print("line 2")
print("value of x is ", x, " and y is ", y)
```

Printing multiple elements on a single line in **SNAPpy** produces a slightly different output from how the output appears when printed from Python. Python inserts a space between elements, where **SNAPpy** does not.

SNAPpy also imposes some restrictions on the printing of nested tuples. You may nest tuples; however, printing of nested tuples will be limited to three layers deep. The following tuple:

```
(1,'A',(2,'b',(3,'Gamma',(4,'Ansuz'))))
```

will print as:

```
(1,'A',(2,'b',(3,'Gamma',(...
```

SNAPpy also handles string representations of tuples in a slightly different way from Python. Python inserts a space after the comma between items in a tuple, while **SNAPpy** does not pad with spaces, in order to make better use of its limited string-processing space.

1.2.9 Docstrings

You can use a special type of comment called a docstring. At the top of a script and after the beginning of any function definition, you can put a specially formatted string to provide inline documentation about that script or function. These special strings are called docstrings.

Docstrings should be delimited with three single quote characters (`'''`) or three double quote characters (`"""`). (Use double quotes if your string will span more than one line.) Here are some examples:

```
"""This could be the docstring at the top of a source file,
explaining what the purpose of the file is"""
def printHello():
    """this function prints a short greeting"""
    print("hello")
```

1.3 SNAPpy Scripting

SNAPcore can run applications that are written in the *SNAPpy Language*. **SNAPcore** includes a **SNAPpy** Virtual Machine, which provides a layer of abstraction that separates the applications from the physical hardware. This means that your **SNAPpy** applications are extremely portable between our different module types. Start by understanding these concepts when developing your **SNAPpy** scripts.

Topics

1.3.1 Memory Management

Here is a high-level overview of the types of memory management that are going on “behind the scenes.”

SNAP Buffers

The **SNAP** Protocol Stack uses a pool of “packet buffers”, each 123 bytes long.

When you send a packet, receive a packet, print, etc., you are temporarily using up one of these packet buffers, which will later be returned to the global pool.

As a concrete example, when your script makes a call to the rpc() function, the actual RPC packet gets encoded into a buffer and enqueued to the radio code for transmission. Once the radio has actually sent the packet (possibly after a Mesh Route Discovery has first taken place), the packet will be returned to the pool.

Buffer Budgets

The "buffer pool" is shared among the various data sources, but no single source is allowed to use up all of the buffers. These "budget" numbers refer to how many buffers an individual data source is allowed to request.

Note

An individual budget number represents the maximum number of buffers that could get allocated to that particular function at one time. Because the buffer pool is "over-subscribed", there may be fewer than the "max budget" buffers available.

As a concrete example, on the RF100 there are only 2 buffers allocated to STDOUT ("print" statements). If your script printed more than 246 characters in one burst, then the excess characters would be dropped (not printed). If your device was busy processing a lot of inbound and outbound RPC calls, there might only be one buffer available to perform print statements, resulting in characters being dropped after the first 123.

Note

Use of the *HOOK_STDOUT* and *HOOK_RPC_SENT* events can help you make more efficient use of your packet buffers.

Dynamic Strings and Byte Lists

Four pools of buffers are used to service both the string and byte list operations. See platform-specific for the maximum size and quantity of the tiny, small, medium, and large pools.

As a concrete example, the following line of **SNAPpy** will use up one dynamic string buffer:

```
message = 'Hello, ' + nameStr
```

Note

The default value of *stdinMode()* is **line mode** which reserves one string buffer of the largest size available on the platform. To reclaim this string buffer for other use, set *stdinMode()* to **character mode** from within your **SNAPpy** script.

1.3.2 Event-Driven Programming

Applications in **SNAPpy** often have several activities going on concurrently. How is this possible, with only one CPU on the **SNAP** engine? In **SNAPpy**, the illusion of concurrency is achieved through event-driven programming. This means that most built-in **SNAPpy** functions run quickly to completion and almost never "block" or "loop" waiting for something. External events will trigger **SNAPpy** functions.

Warning

Notice the word *almost* in that last paragraph. As a quick counter-example, if you call the *pulsePin()* function with a negative duration, then by using that parameter you have requested a blocking pulse -

the call to *pulsePin()* will not return until the requested pulse has been generated. This means it is very important that *your* **SNAPpy** functions also run quickly to completion!

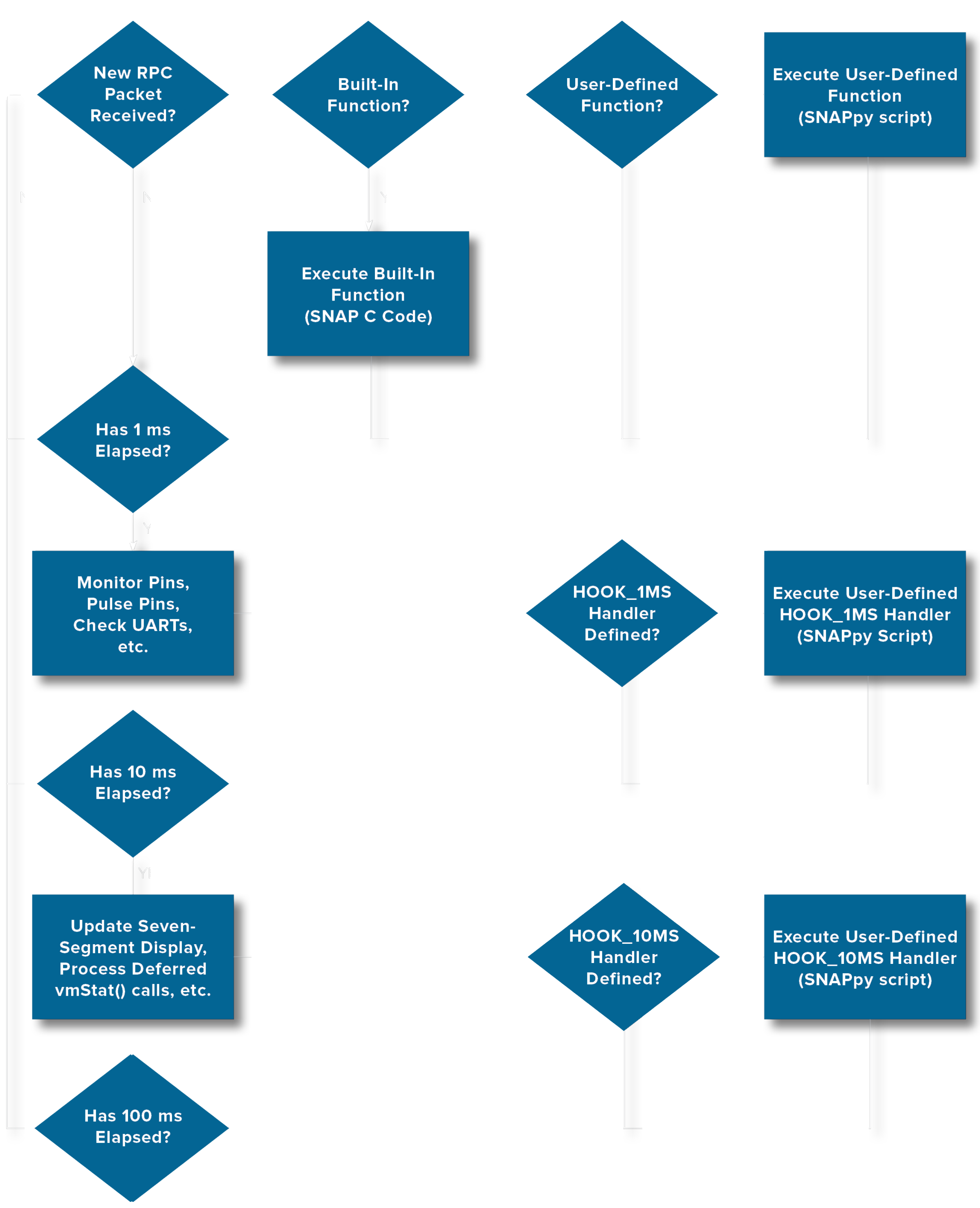
As an example of what not to do, consider the following code snippet:

```
while readPin(BUTTON_PIN) != BUTTON_PRESSED:
    pass

print("Button is now pressed")
```

Instead of monopolizing the CPU like this, your script should use **SNAPpy**'s *monitorPin()* and *HOOK_GPIN* functionality.

To understand why *hard loops* like the one shown above are so bad, take a look at this flowchart.



Note

The flowchart is not exhaustive and only shows high-level processing! The remainder of the flowchart is *chopped off* at this point. It was only being used to make a point, not to show all of **SNAP**'s internal

workflow.

If you focus your attention on the left side of the flowchart, you will recognize that **SNAP** itself uses a software architecture commonly referred to as *one big loop*. **SNAPcore** is written in C and is quickly able to monitor the radio, check for GPIO transitions, perform mesh routing, etc.

Now focus your attention on the highlighted blocks on the right side of the flowchart. These show some of the times when the **SNAPpy** virtual machine might be busy executing portions of your **SNAPpy** script (those associated with HOOK_xxx handlers, as well as user-defined RPC calls).

While the device is busy interpreting the **SNAPpy** script, the other functions (the ones not highlighted) are not getting a chance to run. **SNAPcore** cannot be checking timers or watching for input signals while it is busy running one of your **SNAPpy** functions.

To give a specific example, if one of your RPC handlers takes too long to run, then the `HOOK_1MS` handler will not be running at the correct time, because it had to wait. If your script hogs the CPU enough, you will not get the correct quantity of timer hooks – **SNAP** sets a flag indicating that a timer hook needs to be invoked, but it does not *queue them up*. So, if you have a function that takes several milliseconds to run to completion, upon completion **SNAP** will only see that the flag is set and will only advance its internal millisecond "tick" counter by one tick.

Note

Time-triggered event handlers must run quickly, finishing well before the next time period occurs. To ensure this, keep your timer handlers concise. There is no guarantee that a timing handler will run precisely on schedule. If a **SNAPpy** function is running when the time hook would otherwise occur, the running code will not be interrupted to run the timer hook code.

Be sure to "hook" the correct event. For example, `HOOK_STDIN` lets **SNAP** devices process incoming serial data. `HOOK_STDOUT` lets **SNAP** devices know when a previous "print" statement has been completed.

Also, be sure that the routine you are using for your event processing accepts the appropriate parameters, whether it actually uses them or not.

1.3.3 The Switchboard

The flow of data through a **SNAP** device is configured via the switchboard. This allows connections to be established between sources and sinks of data in the device.

Overview

The following data sources/sinks are defined in the file switchboard.py, which can be imported by other **SNAPpy** scripts. (You may also use the enumerations directly in your scripts, without importing the switchboard.py file.)

Value	Enumeration
0	DS_NULL
1	DS_UART0
2	DS_UART1
3	DS_TRANSPARENT
4	DS_STDIO
5	DS_ERROR
6	DS_PACKET_SERIAL
7	DS_AUDIO (ZIC2410 only)

The **SNAPpy** API for creating switchboard connections is:

```
crossConnect(dataSrc1, dataSrc2) # Cross-connect SNAP data source (bidirectional)
uniConnect(dst, src)             # Data from src goes to dst (unidirectional)
```

Two `uniConnect()` calls can be equal to one `crossConnect()` call. For example:

```
uniConnect(DS_UART0, DS_UART1)
uniConnect(DS_UART1, DS_UART0)
```

Has the same effect as:

```
crossConnect(DS_UART0, DS_UART1)
```

To configure UART1 for Transparent (Wireless Serial) mode, put the following statement in your **SNAPpy** startup handler:

```
crossConnect(DS_UART1, DS_TRANSPARENT)
```

The following table is a matrix of possible switchboard connections. Each cell label describes the "mode" enabled by a row-column cross-connect.

	DS_UART0	DS_UART1	DS_TRANSPARENT
DS_UART0	*Loopback*	*Crossover*	*Wireless Serial*
DS_UART1	*Crossover*	*Loopback*	*Wireless Serial*
DS_TRANSPA	*Wireless Serial*	*Wireless Serial*	*Loopback*
DS_STDIO	*Local Terminal*	*Local Terminal*	*Remote Terminal*
DS_PACKET_S	*Packet Serial*	*Packet Serial*	Remote **SNAPstack**

Any given data sink can be the destination for multiple data sources, but a data source can only be connected to a single destination. Therefore, if you cross-connect two elements you cannot direct serial data from either of those elements to additionally go anywhere else, but you can still direct other elements to be routed to one of the elements specified in the cross-connect.

The `DS_ERROR` element is a data source, but it cannot be a data sink. Uniconnecting `DS_ERROR` to a destination causes any error messages generated by your program to be routed to that sink. In this way, you can (for example) route error messages to **SNAPstack** while allowing other serial data to be directed to a UART.

Note

Most platforms have two UARTs available, so with most **SNAP** RF Modules, UART0 will connect to the USB port on an SN163 board and UART1 will connect to the RS-232 port on any appropriate Synapse demonstration board.

However, the RF300 **SNAP** RF Module has only one UART - UART0 - and it comes out where UART1 normally comes out (to the RS-232 port, via GPIO pins 7 through 10). If you are working with RF300 **SNAP** engines, be sure to adjust your code to reference UART0 rather than UART1 for your RS-232 serial connections.

Loopback

A command like `crossConnect(DS_UART0, DS_UART0)` will set up an automatic loopback. Incoming characters will automatically be sent back out the same interface.

Crossover

A command like `crossConnect(DS_UART0, DS_UART1)` will send characters received on UART0 out UART1 and characters received on UART1 out UART0.

Wireless Serial

SNAP supports efficient, reliable bridging of serial data across a wireless mesh. Data connections using the transparent mode can exist alongside RPC-based messaging.

A command like `crossConnect(DS_UART0, DS_TRANSPARENT)` will send characters received on UART0 over-the-air (OTA). Where the data will actually be sent is controlled by other **SNAPpy** built-ins.

See also

- *ucastSerial()*
- *mcastSerial()*

Local Terminal

SNAPpy scripts can also interact directly with the serial ports, allowing custom serial protocols to be implemented. The **SNAP** device can be either the consumer or the creator of the serial data.

A command like `crossConnect(DS_UART0, DS_STDIO)` will send characters received on UART0 to your **SNAPpy** script for processing. The characters will be reported to your script via your specified *HOOK_STDIN* handler. Any text "printed" (using the print statement) will be sent out that same serial port.

This makes it possible to implement applications like a Command Line Interface.

Remote Terminal

SNAP's transparent mode takes data from one interface and forwards it to another interface (possibly the radio), but the data is not altered (or even examined) in any way.

A command like `crossConnect(DS_TRANSPARENT, DS_STDIO)` will send characters received wirelessly to your **SNAPpy** script for processing. Characters "printed" by your **SNAPpy** script will be sent back out over-the-air.

This is often used in conjunction with a `crossConnect(DS_UARTx, DS_TRANSPARENT)` in some other **SNAP** device.

Packet Serial

A command like `crossConnect(DS_UART0, DS_PACKET_SERIAL)` will configure the unit to talk Synapse's Packet Serial protocol over UART0. This enables RS-232 connection to a PC running **SNAPstack**.

This also allows serial connection to another **SNAP** device (if the appropriate "cross-over" cable is used), which allows "bridging" of separate **SNAP** networks. Meaning networks that are on different channels and/or different Network IDs and/or different radio frequency ranges can communicate with each other.

A command like `crossConnect(DS_UART1, DS_PACKET_SERIAL)` will configure the unit to talk Synapse's Packet Serial protocol over UART1. On some **SNAP** demonstration boards, one UART will be a true RS-232 serial connection, and the other will be a USB serial connection.

See also

- *`crossConnect()`*
- *`uniConnect()`*

1.3.4 SNAPpy Scripting Tips

The following are some helpful tips (sometimes learned from painful lessons) for developing custom **SNAPpy** scripts:

Beware of Case Sensitivity

In **SNAPpy** (as with Python), identifiers are case sensitive - `foo` is not the same as `Foo`.

SNAPpy is a dynamically-typed language, so it is perfectly legal to create a new variable on-the-fly. In the following **SNAPpy** code snippet two variables are created, and `foo` still has the original value of `2`:

```
foo = 2
Foo = "The Larch"
```

Case sensitivity applies to function names as well as variable names:

```
linkQuality = getlq()   # ERROR! Unless you have defined your own function
linkQuality = getLq()   # Probably what you want
```

Beware of Accidental Local Variables

In **SNAPpy** (as with Python), all functions can read global variables, but you need to use the "global" keyword in your functions if you want to write to them:

```
count = 4  # create global count and set it to 4

def bumpCount():
    count = count + 1  # global count will still equal 4

def bumpCountTry2():
    global count       # needed to avoid creating a local version of count
    count = count + 1  # will actually increment global count
```

Don't Cut Yourself Off (Packet Serial)

SNAPstack talks to its "bridge" (directly connected) device using a packet serial protocol. **SNAPpy** scripts can change both the UART and Packet Serial settings.

This means you can be talking to a device via **SNAPstack** and then upload a script into that device that starts using that same serial port – or even just the same **SNAP** engine pins – for some other function (for example, for printing script text output or as an externally triggered sleep interrupt). **SNAPstack** will no longer be able to communicate with that node serially.

Serial Output Takes Time

In the following example, there is likely not enough time for the text to make it all the way out of the device (particularly at slower baud rates) before the *sleep()* command shuts off the device:

```
def goodNightMessage():
    print("imagine a very long and important message here")
    sleep(...)  # sleep parameters vary per platform
```

One possible solution would be to invoke the *sleep()* function from the *HOOK_100MS* hook event. First, create a new global by adding it to the top of your script:

```
goodNightCountDown = 0
```

Then, change the `goodNightMessage()` function to:

```
def goodNightMessage():
    global goodNightCountDown

    print("imagine a very long and important message here")
    goodNightCountDown = 500 # actual number of milliseconds may vary
```

Finally, add this logic to the handler for *HOOK_100MS*

```
@setHook(HOOK_100MS)
def callEvery100ms(tick):
    global goodNightCountDown

    if goodNightCountDown != 0:
```

(continues on next page)

(continued from previous page)

```
        if goodNightCountDown <= 100: # timebase is 100 ms
            goodNightCountDown = 0
            sleep(...)                # sleep parameters vary per platform
        else:
            goodNightCountDown -= 100
```

SNAP Engines Do Not Have a Lot of RAM

SNAPpy scripts should avoid generating a flood of text output all at once, because there is nowhere to buffer the output and the excess text will be truncated. Instead, generate the composite output in small pieces (for example, one line at a time), triggering the next step of the process using the *HOOK_STDOUT* event.

SNAPpy Numbers Are Integers

`2/3 = 0` in **SNAPpy**. As in all fixed-point systems, you can work around this by *scaling* your internal calculations up by a factor of 10, 100, etc. You then scale your final result down before presenting it to the user.

SNAPpy integers are 16-bit numbers and have a numeric range of -32768 to +32767. Remember that `32767 + 1 = -32768`, and be careful that any intermediate math computations do not exceed this range, as the resulting overflow value will be incorrect.

A side-effect of **SNAPpy** integers being signed is that negative numbers shifted right are still negative, because the sign bit is preserved. You might expect `0x8000 >> 1 = 0x4000`, but it is `0xC000`. You can use a *bitwise and* operator (`&`) to clear the sign bit after a shift:

```
myInt = myInt >> 1
myInt = myInt & 0x7FFF
```

Pay Attention to Script Output

Any **SNAPpy** script errors that occur can be printed to the previously configured STDOUT destination, such as serial port 1. If your script is not behaving as expected, be sure and check the output for any errors that may be reported.

Don't Define Functions Twice

In **SNAPpy** (as with Python), defining a function that already exists counts as a re-definition of that function. Any script code that used to invoke the old function will now be invoking the replacement function instead. Using meaningful function names will help alleviate this.

SNAPpy Has Limited Dynamic Memory

Functions that manipulate strings (concatenation, slicing, subscripting, chr()) all pull from a small pool of dynamic (reusable) string buffers.

You still do not have unlimited string space and can run out if you try to keep too many strings. See each platform's section in the SNAP Reference Manual for a breakdown of how many string buffers are available and what size those buffers are.

Use the Supported Form of Import

In **SNAPpy** scripts you should use the form:

```
from moduleName import *
from synapse.moduleName import *
from moduleName import specificFunction
```

Be Careful Using Multicast RPC

If all devices hear the question at the same time, they will all answer at the same time. If you have more than a few devices, you will need to coordinate their responses if you poll them via a multicast RPC call. Here are some possible solutions:

1. *NV18 - Collision Avoidance* inserts some random delay (up to 20 ms) when responding to multicast requests, to assist in overcoming this.
2. *NV16 - Carrier Sense* and *NV17 - Collision Detect* help ensure you do not have too many devices talking at the same time.
3. A directed multicast message has a built-in delay factor. By providing an empty string for the destination addresses, it will behave the same as a multicast message; however, you will still be able to take advantage of the new built-in delay feature.
4. Application-level control of when your device responds to a request.

Recovering an Unresponsive Node

As with any programming language, there are going to be ways you can put your devices into a state where they do not respond. Setting a device to spend all of its time asleep, having an endless loop in a script, enabling encryption with a mistyped key, or turning off the radio and disconnecting the UARTs are all very effective ways to make your **SNAP** devices unresponsive.

1.4 SNAP Networking

SNAPcore allows your device to participate in a **SNAP** network, unifying communications and control across disparate physical layers and across different platforms.

Topics

1.4.1 SNAP Routing

Reviewing the Basics

Consider a situation where there is a distribution of your devices across a large geographic area, such that some devices cannot communicate directly with other devices without **SNAP**'s automatic mesh networking assisting, by routing and delivering the messages through other devices. If the Alice device can consistently communicate with the Bob device, and the Bob device can consistently communicate with the Carol device, messages from the Alice device to the Carol device will be forwarded automatically by the Bob device if the Alice device and the Carol device cannot communicate directly.

However, the acknowledgement messages in this arrangement are incomplete, each confirming only part of the whole path. When the Alice device has a message for the Carol device, it begins by sending out a route request, essentially "Is device Carol in range, or does any nearby device know where I can find the Carol device?" Any non-Carol devices that hear the route request will forward the request: "Hey, out there! The Alice device is looking for the Carol device!", and this will continue (within limits) until the Carol device is found. The Carol device will then reply to the device that it heard asking, which then replies to the device it heard asking, all the way back to Alice.

So, the Alice device asks for a route to the Carol device, the Bob device hears the request and asks for a route to the Carol device, device Carol responds to the Bob device, which now knows it can talk to Carol, and which then responds to the Alice device, indicating that it has a path to the Carol device.

Now the Alice device knows that if it has a message for the Carol device, it must ask the Bob device to forward the message to device Carol. When the Alice device sends a message for the Carol device, the Bob device hears the request and sends the Alice device an acknowledgement. The Bob device then forwards the message to the Carol device and waits for the Carol device to send an acknowledgement.

For either of these transmissions, if the receiving device does not send an acknowledgement packet within a configurable timeout period, the sending device resends the message up to a configurable number of times before realizing that it cannot get through.

All of this route seeking and acknowledgement protocol occurs automatically with **SNAP**. There is nothing the user must "turn on" in order to make it work (though much of the functionality can be fine-tuned through the use of NV parameters).

In the above configuration, imagine a situation where the Alice device has discovered a route through the Bob device to device Carol and sends the Carol device a message. The Bob device hears the message and sends the Alice device an acknowledgement, so the Alice device goes on about its business confident that its message is delivered. However, at the point that the Alice device hears the acknowledgement, the Carol device has not yet received the message from the Bob device. If the Carol device is sleeping or is otherwise unable to hear the message from the Bob device, the Bob device will attempt the configured number of retries but will eventually give up if the Carol device cannot be found – and this failure to forward the message will not be reported back to the Alice device (other than a route failure message that goes out, indicating to the Alice device that the next time it tries to communicate with the Carol device, it should first perform a new route request).

If you need to be sure that your target device has received a message, whether the message were sent by unicast or multicast, it is best to write your application to explicitly send a confirmation that the message has been received and to explicitly retry sending the message if no such confirmation comes.

Preserving Unicast Routes

When one device communicates directly to another, it must know how to reach the other device. If the devices are in direct radio range of each other, the route of communication between them is very simple. But if it is necessary for a message to hop one or more times between devices, the transmitting device must know how to direct the message in order for it to be properly delivered.

Consider a network of six devices, where device A is within radio range of devices B and C, device B is in range of devices A, C, and D, device C is in range of devices A, B, and E, device D is in range of devices B, E, and F, device E is in range of devices C, D, and F, and device F is in range of devices D and E.

In this network, if device A needs to send a message to device F, the message can be routed through devices B and D, through devices C and E, or even by path A-B-C-E-D-F. But when the network first comes online, device A has no way of knowing these routes.

When device A needs to send that message, the first thing it does is send a Route Request message, asking "Does anybody know where device F is?" Devices B and C hear the request, but they do not yet know how to reach device F (though they now know they can hear device A). So, devices B and C send Route Request messages asking for device F. Device D hears the request from device B, and device E hears the request from device C. Devices D and E do not yet know how to reach device F, so they send Route Request messages of their own.

Device F will hear the Route Request messages from devices E and D and respond with a "Here I am!" message. Devices D and E now have routes to device F. Additionally, device D knows device B and knows that device A can be reached through device B; device E knows device C and knows that device A can be reached through device C. Device D then replies to device B, saying "You can reach device F through me." Device E sends device C the same message. Then, devices B and C both send device A messages saying "You can reach device F through me."

Device A picks one of those routes to keep (based on timing and signal strength) and now knows that if it needs to send device F a message, it can send device B a message saying, "Pass this message on to device F." Device B would send an acknowledgement to device A and then send device D a message saying, "Pass this message from device A on to device F." Device D acknowledges device B's message and then sends device F the message saying "Device A sent you this message."

Typically, devices can maintain up to 10 such routes. But in networks where devices are not stationary, it may be problematic for a device to continue to attempt to use a route that is no longer stable or available. So by default, these routes time out.

See also

The following NV parameters control how these routes are found and how long they survive before timing out:

- *NV20 - Mesh Maximum Timeout*
- *NV21 - Mesh Minimum Timeout*
- *NV22 - Mesh New Timeout*
- *NV23 - Mesh Used Timeout*
- *NV24 - Mesh Delete Timeout*
- *NV25 - Mesh RREQ Retries*
- *NV26 - Mesh RREQ Wait Time*
- *NV27 - Mesh Initial Hop Limit*

- *NV28 - Mesh Maximum Hop Limit*
- *NV29 - Mesh Sequence Number*
- *NV30 - Mesh Override*
- *NV31 - Mesh LQ Threshold*
- *NV32 - Mesh Rejection LQ Threshold*

1.4.2 SNAP Addresses

Each **SNAP** device has a unique **SNAP** address, defined by the last three bytes of the device's MAC address. Thus, a **SNAP** device with the MAC address `001C2C1E8600669B` would have a **SNAP** address of 00669B. Typically, people will add "dots" to the address when printing it to make it easier to read: 00.66.9B. **SNAP** devices reference each other (to send procedure calls) using these addresses, both for directed multicasts and for unicast RPC calls.

In such calls, the address is specified as a three-character string. The above address would be specified as `\x00\x66\x9b`. Of these three characters, only the `\x66` is directly printable (it displays as an `f`).

This can make it difficult to present a human-readable indication of a device's address if you have some function indicating from where a message was received. A function like the following will *decode* the address to human-readable form:

```
def decodeSnapAddress(address):
    key = "01234567889ABCDEF"
    returnValue = ""

    for character in address:
        byte = ord(character)
        returnValue += key[byte / 16]
        returnValue += key[byte % 16]
        returnValue += "."
        returnValue = returnValue[:-1]

    return returnValue
```

When using the directed Multicast functions (*dmcastRpc()* and *dmCallout()*), you will often want to address more than one device. Simply concatenate multiple **SNAP** addresses into a longer string. A value of `\x00\x66\x9b\x05\x47\x56\x5f\xe6\x23` would be acted on by devices 00.66.9B, 05.47.56, and 5F.E6.23, assuming all three of those devices are reachable by the network, within the specified TTL, and belong to the execution groups that match the multicast groups specified in the call.

1.4.3 Multicast Groups

By default, all devices belong to the "broadcast" group `0x0001`. You can configure your devices to belong to different or additional groups.

In the following example, all devices within 5 hops and belonging to group 1 or group 2 are asked to run the *reboot()* function:

```
mcastRpc(3, 5, 'reboot')
```

While this example will ask all devices within 2 hops and belonging to group 2 or group 4 to run the *reboot()* function:

```
mcastRpc(6, 2,'reboot')
```

Remember that the `group` parameter is a bit mask and is not specific group numbers. To calculate the groups for the two examples above, we need to represent the `group` value as a binary number to see which bits are set:

Value	Hex Value	Binary Value	Groups Included
3	0x0003	0000000000000011b	1 and 2
6	0x0006	0000000000000110b	2 and 3

If you are broadcasting to multiple groups concurrently, you may find it easier to use hexadecimal notation for your `group` parameter. The following two commands are equivalent:

```
mcastRpc(19753, 5, 'reboot')   # decimal 19753  = 0100110100101001b
mcastRpc(0x4d29, 5, 'reboot')  # hex     0x4d29 = 0100110100101001b
```

Value	Hex Value	Binary Value	Groups Included
19753	0x4d29	0100110100101001b	1, 4, 6, 9, 11, 12 and 15

See also

- *NV5 - Multicast Process Groups*
- *NV6 - Multicast Forward Groups*

1.4.4 Remote Procedure Calls

Remote Procedure Call (RPC)-related functionality is a fundamental part of **SNAP** and **SNAPpy**, and there are many types of **SNAPpy** built-in functions that can be used to invoke a function on another device.

Introduction

All public functions, including the **SNAPpy** built-in functions, are remotely callable using the **SNAP** RPC protocol. Non-public functions (prefixed with underscore) are not remotely callable, but they can be called by other functions within the same script.

These are non-blocking functions. They only enqueue the packet for transmission and do not wait for the packet to be sent (let alone wait for it to be processed by the receiving device(s)). Each of these functions can be used to invoke any public function on another **SNAP** device (either a user-defined function or a built-in function), but each has additional strengths, weaknesses, and capabilities.

It is important to provide the correct number of arguments when calling a remote function. Issuing a command with the wrong number of arguments will not work, because the receiving device will not be able to find a function signature that matches.

This also applies to **SNAPstack**-related programming. Make sure that any RPC handlers defined in **SNAPstack** applications accept the same number and type of arguments that the remote callers are providing. For example, if your script takes two arguments:

```
def displayStatus(msg1, msg2):
    print(msg1 + msg2)
```

But the **SNAPpy** script makes RPC calls with three parameters:

```
rpc(SNAPSTACK_ADDR, "displayStatus", 1, 2, 3)  # <- too many parameters provided
```

Or just one parameter:

```
rpc(SNAPSTACK_ADDR, "displayStatus", 1)  # <- too few parameters provided
```

Then the `displayStatus()` function will not be invoked function at all.

Calling By Name

You cannot invoke a function that a device does not have loaded. **SNAP** devices without scripts only support the "call by number" method to call RPC functions. The name lookup table that allow devices to use "call by name" is sent to the node when a script is loaded. This means that if you want to call a **SNAPpy** built-in function by name, a remote device needs a script loaded, even if the script is empty.

> **Note**
>
> In the case of *`mcastRpc()`*, the device will silently ignore any function that it does not know how to call. If sent via one of the unicast mechanisms (*`rpc()`*, *`callback()`* or *`callout()`*) the packet will be acknowledged but then ignored.

Realize that if you multicast an RPC call to function "foo", all devices in that multicast group that have a `foo()` function will execute it, even if their `foo()` function does something different from what your target device's `foo()` function is expected to do. Giving your **SNAPpy** functions distinct and meaningful names is recommended.

Selecting the Type of RPC

The reliability of message delivery is an important consideration in selecting the best way to send the message. Multicast messages generate no explicit confirmation of receipt from any other device in your network. If you want to be sure that a particular device has heard a multicast RPC call, you must provide that confirmation as part of your own application, generally in the form of a message sent back to the originating device.

Unicast RPC calls are addressed to a single target device, rather than being open to all devices that can hear the request. The reliability of these calls is bolstered by a series of acknowledgement messages sent back along the path, but even that is no guarantee of a final receipt of the message, especially in a dynamic environment.

Multicast RPC

The built-in function *mcastRpc()* is best at invoking the same function on multiple devices from a single invocation. The trade-off is that the actual packet is usually only sent once[1]. How many devices actually perform the requested function call is a function of three factors:

1. Number of hops
   a. How many hops (forwards or retransmissions of the message) away are the other devices?
   b. How many hops did you specify in the *mcastRpc()* call? (TTL parameter)
2. Script existence
   a. Is there a function with that name in the device's currently loaded script?
   b. Are you providing the correct number of arguments?
   c. What function name and arguments did you specify in the *mcastRpc()* call?
3. Group membership
   a. What *Multicast Groups* do the other devices belong to?
   b. What candidate groups did you specify in the *mcastRpc()* call?
   c. If the destination device is more than one hop away, do intermediate devices forward the specified group?

The following is an example of using multicast RPC to increase the synchronization of the devices. Not all of the devices will necessarily be running the `startDataCapture()` function at the same moment, because the closer devices will be hearing the command sooner than the devices that require more mesh network hops. All devices in group 2 that can be reached within 4 or fewer hops and that have a function named `startDataCapture()` in their script will invoke that function:

```
mcast_groups = 2
hop_count = 4

mcastRpc(mcast_groups, hop_count, "startDataCapture")
```

This next example is a little exaggerated, but sometimes you will need to send an RPC to all of the devices because of the importance of the message:

```
reactor_temperature = get_reactor_temperature()

if reactor_temperature > CRITICAL_TEMPERATURE:
    mcastRpc(1, 255, "runForYourLives")
```

It is important to note that the function being called might make an RPC call of its own. For the sake of the next example, imagine a network of exactly two **SNAP** devices, devices A and B, and that device B contains the following script snippet:

```
def askSensorReading():
    value = readSensor()
    mcastRpc(1, 1, "tellSensorReading", value)
```

Now imagine that device A contains the following script snippet:

[1] The exception to that rule is if you have enabled the optional collision detect feature of **SNAP**, in which case the packet might be sent more than once if a packet collision was detected.

```
def tellSensorReading(value):
    print("I heard the sensor reading was ", value)
```

If device A executes the following command, this would result in two-way communication:

```
mcastRpc(1, 1, "askSensorReading")
```

First, device A will send `askSensorReading()` to device B, and then device B will send `tellSensorReading()` back to node A.

You should also be aware that even though an RPC call is made via multicast, it is still possible to have only a single device completely process that call. Imagine the above two-node network is expanded by adding devices C-Z and that we upload the following script to devices B-Z:

```
def askSensorReading(who):
    if who == localAddr(): # is this command meant for ME?
        value = readSensor()
        mcastRpc(1, 1, "tellSensorReading", value)
```

Now, if device A executes the following command:

```
mcastRpc(1, 1, "askSensorReading", address_of_node_B)
```

Even though all devices within a one-hop radius will invoke function `askSensorReading()`, only device B will actually take a reading and report it back.

Directed Multicast RPC

The built-in function *dmcastRpc()* functions like *mcastRpc()* but with two additional features:

1. You may target one or more specific *remote* device(s) on which your function should execute.

   If a device does not find its own address in the incoming message, it will still forward the message to other devices (subject to remaining TTL and the restrictions placed by the specified multicast groups), but it will not execute the function, even if the specified multicast groups would otherwise instruct the device to do so.

2. You can define a delay (in milliseconds) so that the multiple *remote* devices can respond in a sequential manner rather than all at once.

   The delay only applies to *radio* communications directly invoked by the *remote* devices. There is no delay applied to *serial* communications directly invoked by the *remote* device. A check of `getInfo(25)` in the called function on the *remote* device returns the delay value specified, but it also tells the *remote* device to ignore the transmission delay that would have normally been enforced.

For example, if you wanted to target devices with addresses 01.02.03, 04.05.06, 07.08.09, and AA.BB.CC, you would need to pass these addresses as a concatenated string:

```
dmcastRpc('\x01\x02\x03\x04\x05\x06\x07\x08\x09\xaa\xbb\xcc', 0x0001, 5, 40, 'readAdc' ,
→0)
```

In the example above, the delay is set to `40` ms, so any radio traffic generated by device 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 AA.BB.CC would be held for 120 ms before release.

Unicast RPC

The built-in function *rpc()* is like *mcastRpc()* but with two differences:

1. Instead of specifying a group and a number of hops (TTL), with the rpc() function you specify the actual **SNAP** address of the intended target device.

   a. The **SNAP** mesh routing protocol will take care of "finding" the device (if it can be found).

   b. Other devices (with different **SNAP** addresses) will not perform the *rpc()* call, even if their currently loaded **SNAPpy** script also contains the requested function. However, they will (by default) assist in delivering the *rpc()* call to the addressed device.

2. Instead of only sending the RPC call a single time (blindly) as *mcastRpc()* does, the *rpc()* function expects a special ACK (acknowledgement) packet in return.

   a. When the target device hears the *rpc()* call, the ACK packet is sent automatically (by the **SNAP** firmware – you do not send the ACK from your script).

   b. If the target device does not hear the *rpc()* call, then it does not know to send the ACK packet. This means the source device will not hear an ACK, and so it will timeout and retry a configurable number of times.

Going back two examples, instead of modifying the `askSensorReading()` function in device B's script to take an additional `who` parameter and calling:

```
mcastRpc(1, 1, "askSensorReading", address_of_node_B)
```

Node A could simply call:

```
rpc(address_of_node_B, "askSensorReading")
```

Devices C-Z would ignore the function call, although they may be helping route the function call to device B without any additional configuration.

The `askSensorReading()` function could also benefit from the use of *rpc()* instead of *mcastRpc()*. Instead of telling the sensor reading to all devices in group 1 within 1 hop away via:

```
mcastRpc(1, 1, "tellSensorReading", value)
```

The script could instead only send the results back to the original requester via:

```
rpc(rpcSourceAddr(), "tellSensorReading", value)
```

Function *rpcSourceAddr()* is another built-in function that, when called from a function that was invoked remotely, returns the **SNAP** address of the calling device.

Note

If you call *rpcSourceAddr()* locally at some arbitrary point in time, such as within the *HOOK_STARTUP* or `HOOK_GPIO` handler, then it simply returns `None`.

Callback

The previous examples allowed one device to ask another device to perform a function and then send the result of that function back to the first device. In each case, the first device called the `askSensorReading()` function, whose only purpose was to call a separate function `readSensor()` and then send the value back. It turns out that **SNAP** has a built-in function to do just that, the *snappy.BuiltIn.callback()* function.

Expanding a little on the previous example, device B's `readSensor()` function is pulling its own weight – it is encapsulating some of the sensor complexity, thus hiding it from the rest of the system:

```
def readSensor():
    return readAdc(0) * SENSOR_GAIN + SENSOR_OFFSET

def askSensorReading():
    value = readSensor()
    mcastRpc(1, 1, "tellSensorReading", value)
```

Sometimes, the raw sensor readings are sufficient or possibly the calculations are so complex that they need to be offloaded to a bigger processor. In that case, we could change the code to this:

```
def readSensor():
    return readAdc(0)

def askSensorReading():
    value = readSensor()
    rpc(rpcSourceAddr(), "tellSensorReading", value)
```

Which can be simplified even further to:

```
def askSensorReading():
    value = readAdc(0)
    rpc(rpcSourceAddr(), "tellSensorReading", value)
```

You might wonder why device A could not skip `tellSensorReading()` all together and just remotely call `readAdc(0)`:

```
rpc(address_of_node_B, "readAdc", 0)
```

Although this will result in device B calling `readAdc(0)`, it will not cause any results to be sent back to node A. This is where the *callback()* function comes in. Let's replace `readAdc` with `callback`:

```
rpc(address_of_node_B, "callback", "tellSensorReading", "readAdc", 0)
```

This will result in device B calling `readAdc(0)`, and the results will be automatically reported back to device A via the `tellSensorReading()` function. The *callback()* function requires you to provide the final function to be invoked ("called back"), in addition to the remote function to be called and its parameters. Notice that we only had to add code to device A's script – we did not have to create an `askSensorReading()` function on device B at all.

It's also important to note that *callback()* is not limited to invoking built-in functions. For example, if we had retained the original `readSensor()` routine, it could be remotely invoked and the result automatically returned via:

```
rpc(address_of_node_B, "callback", "tellSensorReading", "readSensor")
```

Callout

The function *callout()* just takes the *callback()* concept one step further. Instead of asking a device to invoke a function and then call you back with the result, *callout()* is used to ask a device to call a function and then report the result to a *third device*.

For example, device A could ask device B to read analog input channel 0 and tell device C what the answer is. Imagine that device C has a graphical LCD display that the other devices lack:

```
rpc(node_b_address, "callout", device_c_address, "tellSensorReading", "readAdc", 0)
```

In a more complex example, device A could ask device B to find out how well all the devices one hop away could hear node B, and then ask them to send the answers to device A:

```
rpc(node_b_address, "mcastRpc", 1, 1, "callout", device_a_address, "tellLinkQuality",
↪"getLq")
```

Node A is asking device B to send a multicast. All devices within one hop of device B will receive a *callout()* instructing them to call their *getLq()* function. Each device will take the result of the *getLq()* and send it to device A as a parameter of the `tellLinkQuality()`, via an *rpc()* function. Node A would need to have the function for node B's neighbors to send their results:

```
def tellLinkQuality(lq):
    who = rpcSourceAddr()
    # do something with the address and the reported link quality
```

Depending on the network, device A could expect to receive many messages which will answer the question "how well did node B's neighbors hear the multicast transmission?"

Directed Multicast Callout

The *dmCallout()* function was added in **SNAP 2.7**. This function works very similarly to the *callout()* function, except that when the remote device invokes a function, it sends the return value to the list of devices as a *dmcastRpc()* function. As with *callout()*, this allows you to have the target device ask a remote device to do something and then tell other device(s) how it turned out. The difference being that the remote device will use a *dmcastRpc()* function instead of an *rpc()* function to report the result.

1.5 Interfacing Peripherals

SNAPpy allows you to easily interface with some common peripheral interfaces.

1.5.1 I^{2}C

Technically, the correct name for this two-wire serial bus is Inter-IC bus or I^{2}C, though it is sometimes written as I2C. I^{2}C uses two pins:

- **SCL** – Serial Clock Line
- **SDA** – Serial Data line (bidirectional)

Because both the value and direction (input versus output) of the **SCL** and **SDA** pins must be rapidly and precisely controlled, dedicated I^{2}C support functions have been added to **SNAPpy**. Using the following functions, your **SNAPpy** script can operate as an I^{2}C bus master and can interact with I^{2}C slave devices:

- *i2cInit()* – Prepare IO for I^{2}C operations
- *i2cWrite()* – Send data over I^{2}C to another device
- *i2cRead()* – Read data from I^{2}C device
- *getI2cResult()* – Check the result of the other I^{2}C functions

To allow these functions to be as fast as possible, the IO pins used for I^{2}C **SDA** and **SCL** are fixed. The IO pins that are reserved for I^{2}C varies by platform, so refer to platform-specific. These pins are not dedicated for I^{2}C, so if you are not using the I^{2}C functions, you may use the reserved pins for other functions.

Unlike CBUS and SPI, I^{2}C does not use separate Chip Select pins. The initial data bytes of each I^{2}C transaction specify an I^{2}C address. Only the addressed device will respond, so no additional GPIO pins are needed.

The specifics of which bytes to send to a given I^{2}C slave device (and what the response will look like) depend on the I^{2}C device itself. You will have to refer to the manufacturer's data sheet for any given device to which you wish to interface.

I^{2}C Restart

Usually, I^{2}C read and write commands begin with a hardware handshake sequence referred to as an "I^{2}C Start" and end with a mirror-image hardware handshake sequence referred to as an "I^{2}C Stop". The original implementations of *i2cRead()* and *i2cWrite()* automatically generate these handshake sequences for you.

A typical use case looks something like this:

```
bytes_to_write   = <sets up for the next i2cRead()>
addressing_bytes = <address that begins the actual read>

i2cWrite(bytes_to_write, retries, ignoreFirstAck)
info = i2cRead(addressing_bytes, retries, ignoreFirstAck)
```

This use case works with a wide variety of I^{2}C chips. However, various manufacturers have introduced new parts that won't work with the above code snippet. This is because the "I^{2}C Stop" sequence generated at the end of the *i2cWrite()* function resets the I^{2}C chip's internal address registers, thus undoing the setup that was accomplished by writing `bytes_to_write`. Instead of an "I^{2}C Start, I^{2}C Stop, I^{2}C Start, I^{2}C Stop" sequence, these devices require an "I^{2}C Start, I^{2}C Restart, I^{2}C Stop" sequence.

Note

To determine if your particular I^{2}C device requires an "I^{2}C Restart", refer to the manufacturer's datasheet for the I^{2}C part. Some manufacturer data sheets refer to the "I^{2}C Restart" as a "Repeated Start."

Beginning in **SNAP 2.5**, an optional `endWithRestart` argument was introduced to the *i2cWrite()* function which allows you to interact with these devices natively. Just change the above example to include the new parameter:

```
i2cWrite(bytes_to_write, retries, ignoreFirstAck, True)   # Passing True for endWithRestart
info = i2cRead(addressing_bytes, retries, ignoreFirstAck)
```

Prior to **SNAP 2.5**, the only way to interface to devices having this "I^{2}C Restart" requirement was to implement a custom version of the *i2cWrite()* functionality in your own **SNAPpy** script, using the lower-level **SNAPpy** *setPinDir()*, *readPin()*, and *writePin()* functions.

See also

- http://www.i2c-bus.org
- http://www.mcc-us.com/i2chowtouseit1995.pdf - The I2C-bus and how to use it (including specifications)

1.5.2 SPI

SPI is another type of clocked serial bus. It typically requires at least four pins[1] :

- **CLK** – Master timing reference for all SPI transfers
- **MOSI** – Master Out Slave In – data line FROM the master TO the slaves
- **MISO** – Master In Slave Out – data line FROM the slaves TO the master
- **CS** – At least one Chip Select

Numerous options complicate the use of SPI:

- Clock Polarity – The clock signal may or may not need to be inverted
- Clock Phase – The edge of the clock actually used varies between SPI devices
- Data Order – Some devices expect/require Most Significant Bit (MSB) first, others only work with Least Significant Bit (LSB) first
- Data Width – Some SPI devices are 8-bit, some are 12-bit, some are 16-bit, etc.

The SPI support routines in **SNAPpy** can deal with all these variations, but you will have to make sure the options you specify in your **SNAPpy** scripts match the settings required by your external devices. Dedicated SPI support (master emulation only) has been added to the set of **SNAPpy** built-in functions. Four functions (callable from **SNAPpy** but implemented in optimized C code) support reading and writing SPI data:

- *spiInit()* – Setup for SPI (supporting many options!)
- *spiWrite()* – Send data out SPI
- *spiRead()* – Receive data in from SPI (3 wire only)
- *spiXfer()* – Bidirectional SPI transfer (4 wire only)

Three-wire SPI interfaces omit the **MISO** pin. Some three-wire devices are read-only, and you must use the *spiRead()* function. Even if the slave does send data in a three-wire SPI interface, it will do so over the **MOSI** pin. Four-wire SPI interfaces transfer data in both directions simultaneously and should use the *spiXfer()* function. Some SPI devices are write-only, and you should use *spiWrite()* function to send data to them for both three-wire and four-wire hookup.

The data width for SPI devices is not standardized. Devices that use a data width that is a multiple of 8 are trivial (for example, send 2 bytes to make 16 bits total). However, device widths such as 12 bits are common. To support these “non-multiples-of-8” device widths, you can specify how much of the last byte to actually send or receive. For example:

```
spiWrite("\x12\x34", 4)
```

This will send a total of 12 bits: all 8 bits of the first byte (0x12), and 4 bits of the second byte (0x34). The “send LSB first” setting will determine which nibble of the second byte (the first or last) is sent. This setting is specified as part of the *spiInit()* function.

[1] SPI also exists in a three-wire variant, with the **MOSI** pin serving double-duty.

To allow these SPI functions to be as fast as possible, the IO pins used for **CLK**, **MOSI**, and **MISO** are fixed. The IO pins that are reserved for SPI varies by platform, so refer to platform-specific. These pins are not dedicated for SPI, so if you are not using the SPI functions, you may use the reserved pins for other functions.

You will also need as many Chip Select (**CS**) pins as you have external SPI devices. You can choose any available GPIO pin(s) to be your SPI **CS** pins. The basic program flow becomes:

```
# Select the desired SPI device, assuming the chip select is active-low
writePin(your_cs_pin, False)

# Transfer data to the selected SPI device
spiWrite("\x12\x34\x56")

# Deselect the SPI device, assuming the chip select is active-low
writePin(your_cs_pin, True)
```

SPI reads are handled in a similar fashion.

The specifics of which bytes to send to a given SPI slave device (and what the response will look like) depend on the SPI device itself. You will have to refer to the manufacturer's data sheet for any given device to which you wish to interface.

1.5.3 RS-485

Several of the **SNAP** demonstration boards include an RS-232 serial port. The board provides the actual connector (a DE-9, sometimes referred to as a DB-9) and the actual RS-232 line driver. **SNAP** engine UARTS only provide a logic level serial interface (3-volt logic).

RS-422 and RS-485 are alternate hardware standards that can be interfaced to by using the appropriate line driver chips. In general, the **SNAP** engine does not care what kind of serial hardware it is communicating over.

Some types of multi-drop serial hardware are an exception. For these, multiple devices are able to share a single serial connection by providing a special hardware signal called TXENA (transmit enable). Normally, none of the connected devices are asserting their TXENA signals. When a device wants to transmit, it first asserts TXENA. After all of the characters have been shifted out the serial port, the transmitting device deasserts TXENA so that another device can use the connection.

The following example of three devices sharing a multi-drop RS-485 bus may make this clearer. You will also notice that the TXENA signal is active low:

```
Device #1 TXENA --_____----------_____-----------------------------------
Device #1 TX    ---CMD------------CMD------------------------------------
Device #2 TXENA ----------_____------------------------------------------
Device #2 TX    -----------RSP-------------------------------------------
Device #3 TXENA ------------------------_____-----------------------------
Device #3 TX    -------------------------RSP-----------------------------
```

SNAP can interface to this type of hardware (**SNAP** can provide the needed TXENA signal). The TXENA signal is output on the pin normally used for Clear To Send (CTS).

The functionality of the CTS pin is controlled by the **SNAPpy** built-in function *flowControl()*.

1.5.4 CBUS

CBUS is a clocked serial bus and is similar to *SPI*. It requires at least four pins:

- **CLK** – Master timing reference for all CBUS transfers
- **CDATA** – Data from the CBUS master to the CBUS slave
- **RDATA** – Data from the CBUS slave to the CBUS master
- **CS** – At least one Chip Select

Using the *readPin()* and *writePin()* functions, virtually any type of device can be interacted with via a **SNAPpy** script, including external CBUS slaves. Arbitrarily chosen GPIO pins could be configured as inputs or outputs by using the *setPinDir()* function. The **CLK**, **CDATA**, and **CS** pins would be controlled using the *writePin()* function. The RDATA pin would be read using the *readPin()* function.

The problem with a strictly **SNAPpy**-based approach is speed – CBUS devices tend to be things like voice chips, with strict timing requirements. Optimized native code may be preferred over the **SNAPpy** virtual machine in such cases. To solve this problem, dedicated CBUS support (master emulation only) has been added to the set of **SNAPpy** built-in functions. Two functions (callable from **SNAPpy** but implemented in optimized C code) support reading and writing CBUS data:

- `cbusRd()` – "Shifts in" the specified number of bytes
- `cbusWr()` – "Shifts out" the specified bytes

To allow these functions to be as fast as possible, the IO pins used for CBUS **CLK**, **CDATA**, and **RDATA** are fixed. The IO pins that are reserved for CBUS varies by platform, so refer to platform-specific. These pins are not dedicated for CBUS, so if you are not using the CBUS functions, you may use the reserved pins for other functions.

You will also need as many Chip Select (**CS**) pins as you have external CBUS devices. You can choose any available GPIO pin(s) to be your CBUS **CS** pins. The basic program flow becomes:

```
# Select the desired CBUS device, assuming the chip select is active-low
writePin(your_cs_pin, False)

# Read 10 bytes from the selected CBUS
deviceResponse = cbusRd(10)

# Deselect the CBUS device, assuming the chip select is active-low
writePin(your_cs_pin, True)
```

CBUS writes are handled in a similar fashion. If you are already familiar with CBUS devices, you should have no trouble using these functions to interface to external CBUS chips.

Warning

Not all platforms support CBUS, refer to platform-specific.

1.6 Encryption

Communications between **SNAP** devices are normally unencrypted. Using the **SNAP** Sniffer (or some other means of monitoring radio traffic), you can clearly see the traffic passed between devices. This can be very useful when establishing or troubleshooting a network, but it provides no protection for your data from prying eyes. Encrypting your network traffic provides a solution for this. By encrypting all your communications, you reduce the chances that someone can intercept your data.

SNAP devices offer two forms of encryption: AES-128 and Basic encryption. If you have a compatible firmware version loaded into your devices, you can configure them to use AES-128 encryption for all their communications. You must have a firmware version that enables AES-128 to be able to do this. You can determine which firmware is loaded into a device by using **SNAPtoolbelt**. Firmware that supports AES-128 encryption will include "AES-128" in the firmware name.

> **Warning**
>
> Basic "encryption" is **not** strong encryption and should not be used. It is not supported by **SNAPstack** or **SNAPtoolbelt**.

Enabling encryption requires two steps. First, you must indicate that you would like to encrypt your traffic and specify which form of encryption you wish to use. Then, you must specify what your encryption key is. After rebooting the node, all communications from the device (both over the air and over the UARTs) are encrypted, and the device will expect all incoming communications to be encrypted. It will no longer be able to participate in unencrypted networks.

NV50 - Enable Encryption is where you indicate which form of encryption should be used. The valid values are:

- 0 = Use no encryption
- 1 = Use AES-128 encryption
- 2 = Use Basic "encryption"

NV51 - Encryption Key is where you specify the encryption key for your encrypted network. The key must be exactly 16 bytes long. You can specify the key as a simple string (e.g., ThEeNcRyPtIoNkEy), as a series of hex values (e.g., x2ax14x3bx44xd7x3cx70xd2x61x96x71x91xf5x8fx69xb9), or as some combination of the two (e.g. xfbOFx06xe4xf0Forty-Two!). Standard security practices suggest you should use a complicated encryption key that would be difficult to guess.

No encryption will be used if:

- *NV50 - Enable Encryption* is set to a value other than 1 or 2.
- *NV50 - Enable Encryption* is set to 1 in a device that does not have AES-128 encryption available in its firmware.
- The encryption key in *NV51 - Encryption Key* is invalid.

As with all NV parameter configuration, the changes you make will only take effect after the device reboots.

CHAPTER

TWO

API REFERENCE

2.1 Functions

2.1.1 call

`call`(*rawOpcodes*, **functionArgs*)

Call embedded C code.

This function is for advanced users only. There is a separate Application Note that covers how to use this advanced feature.

Parameters

- `rawOpcodes` (*str*) – Machine code that implements the function to be called.
- `*functionArgs` (*arbitrary argument list*) – Parameters depend on the actual function implemented by `rawOpcodes`.

2.1.2 callback

`callback`(*callback*, *remoteFunction*, **remoteFunctionArgs*)

It is easy to invoke functions on another node using the *rpc()* built-in function; however, to get data back from that node, you either need to put a script in that node to explicitly send the value back, or use the *callback()* function.

Parameters

- `callback` (*str*) – Specifies which function to invoke on the **originating** node, passing in the return value of the remote function.
- `remoteFunction` (*str*) – Specifies which function to invoke on the remote node.
- `*remoteFunctionArgs` (*arbitrary argument list*) – Used if the remote function takes any parameters.

Returns

Normally returns True, but it does not mean your RPC request was successfully sent and received

Returns False only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory).

Return type

bool

Examples

Imagine having a function like the following in **SNAP node A**:

```
def showResult(obj):
    print str(obj)
```

Invoking `callback('showResult', 'functionOnB')` on **node B** will cause function `showResult()` to get called on **node A** with the result of function `functionOnB()` on remote **node B**.

The `remoteFunction` parameter can take parameters, so **node A** could also invoke the following on **node B** by passing `0` to the `readAdc()` function:

```
callback('showResult', 'readAdc', 0)
```

Node B would invoke `readAdc(0)` and then remotely invoke `showResult(the-actual-ADC-reading-from-readAdc(0))` on **node A**.

The *callback()* function is most commonly used with the *rpc()* function:

```
rpc(nodeB, 'callback', 'showResult', 'readAdc', 0)
```

Essentially, *callback()* allows you to ask one node to do something and then tell you how it turned out.

Note

Even if you do not have a script that explicitly sends the return value back, you still must have some script in a node before you can call any function, including any of the built-in functions by name.

See also

- User Guide on *Remote Procedure Calls*
- *rpc()*

2.1.3 callout

`callout`(*nodeAddress*, *callback*, *remoteFunction*, **remoteFunctionArgs*)

This function is similar to *callback()*, but instead of the final result being reported back to the **originating** node, you explicitly provide the address of the **target** node.

Parameters

- `nodeAddress` (*str*) – Specifies the **SNAP** address of the target node that is to receive the callback.
- `callback` (*str*) – Specifies which function to invoke on the target node passing in the return value of the remote function.
- `remoteFunction` (*str*) – Specifies which function to invoke on the remote node.
- `*remoteFunctionArgs` (*arbitrary argument list*) – Used if the remote function takes any parameters.

Returns

Normally returns `True`, but it does not mean your RPC request was successfully sent and received.

Returns `False` only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory).

Return type
bool

Examples

Node A could invoke the following on **node B**, which would automatically invoke **node C**:

```
callout(nodeC, 'showResult', 'readAdc', 0)
```

Node B would invoke `readAdc(0)` and then remotely invoke `showResult(the-actual-ADC-reading-from-readAdc(0))` on **node C**.

The *callout()* function is most commonly used with the *rpc()* function:

```
rpc(nodeB, 'callout', nodeC, 'showResult', 'readAdc' ,0)
```

Essentially, *callout()* allows you to have one node ask another node to do something, and then tell a third node how it turned out.

Note

To understand this function, you should first be comfortable with using the *rpc()* and *callback()* built-ins.

See also

- User Guide on *Remote Procedure Calls*
- *rpc()*
- *callback()*

2.1.4 chr

`chr`(*number*)

Translates an ASCII code into a single-character string.

Parameters
`number` (*int*) – ASCII code as an integer.

Returns
A single-character string based on the number given.

Return type
str

Example

Ways to obtain the string `'A'`:

```
chr(0x41) # returns the string 'A'
chr(65)   # so does this
```

2.1.5 crossConnect

`crossConnect`(*dataSrc1*, *dataSrc2*)

Cross-connect **SNAP** data sources.

Parameters

- `dataSrc1` (*int*) – **SNAP** data source.
- `dataSrc2` (*int*) – **SNAP** data source.

Returns

None

See also

- User Guide on *The Switchboard* for details
- *uniConnect()*

2.1.6 dmCallout

`dmCallout`(*dstAddrs*, *dstGroups*, *ttl*, *delayFactor*, *callback*, *remoteFunction*, **remoteFunctionArgs*)

New in version **SNAP 2.7**.

Warning

If you provide a `dstAddrs` that is not a multiple of three characters in length, the call will fail and no message will be sent to any node.

Parameters

- `dstAddrs` (*str*) – A string containing concatenated addresses of the target nodes that are to receive the final (result) function call.
- `dstGroups` (*str*) – Specifies which nodes should receive the outgoing message, based on their multicast processed groups. By default, all nodes belong to the broadcast group 0x0001.
- `ttl` (*int*) – Specifies the Time To Live (TTL) for the request. This specifies how many hops the message is allowed to make before being discarded.
- `delayFactor` (*int*) – Provides a mechanism to allow remote nodes to stagger their responses to the message. 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. Setting this parameter to zero allows all

remote nodes to respond immediately, which may cause packet loss due to interference.

- `callback` (*str*) – Specifies which function to invoke on the originating node, passing in the return value of the remote function.
- `remoteFunction` (*str*) – Specifies which function to invoke on the remote node.
- `*remoteFunctionArgs` (*arbitrary argument list*) – Used if the remote function takes any parameters.

Returns

This function normally returns `True`; however, it does not mean your RPC request was successfully sent and received.

It returns `False` only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory or the RPC was too large to send).

Return type

bool

Note

As with a *dmcastRpc()*, any node that finds its address when it receives the multicast message will act on the message (assuming that the multicast processed groups setting is compatible). Also as with *dmcastRpc()*, if you pass an empty string as the `dstAddrs` parameter, the outgoing message will behave as a regular *mcastRpc()* call.

See also

- User Guide on *Remote Procedure Calls*
- *callout()*
- *dmcastRpc()*

2.1.7 dmcastRpc

`dmcastRpc`(*dstAddrs*, *dstGroups*, *ttl*, *delayFactor*, *remoteFunction*, **remoteFunctionArgs*)

New in version **SNAP 2.6**.

Directed multicast provides a means to send an RPC call to a list of remote nodes without incurring the route discovery overhead necessary for addressed RPC calls.

Parameters

- `dstAddrs` (*str*) – A string containing concatenated addresses for any nodes you wish to act on the directed multicast.

  If you provide an empty string (`""`), all remote nodes that receive the message that would otherwise act on the message (subject to the `dstGroups` parameter and the existence of the function in the remote node's script) will act on the request as though the call were a regular *mcastRpc()* call; however, in this case added features available only for directed multicast (such as information available through several *getInfo()* calls) are also available.

- `dstGroups` (*str*) – Specifies which nodes should respond to the request. By default, all nodes belong to the broadcast group 0x0001.
- `ttl` (*int*) – Specifies the Time To Live (TTL) for the request. This specifies how many hops the message is allowed to make before being discarded.
- `delayFactor` (*int*) – Provides a mechanism to allow remote 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. Setting this parameter to zero allows all remote nodes to respond immediately, which may cause packet loss due to interference.
- `remoteFunction` (*str*) – Specifies which function to invoke on the remote node.
- `*remoteFunctionArgs` (*arbitrary argument list*) – Used if the remote function takes any parameters.

Returns

This function normally returns `True`; however, it does not mean your RPC request was successfully sent and received.

It returns `False` only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory or the RPC was too large to send).

Return type
bool

Warning

If you provide a `dstAddrs` that is not a multiple of three characters in length, the call will fail and no message will be sent to any node.

See also

- User Guide on *Remote Procedure Calls*
- *mcastRpc()* for more information about groups and TTL settings
- *dmCallout()*
- *getInfo()*

2.1.8 eraseImage

`eraseImage()`

This function is used by **Portal** and **SNAPconnect** as part of the script upload process and is not normally used by user scripts. Calling this function automatically invokes the *resetVm()* function before erasing the image (otherwise the **SNAPpy** VM would still be running the script, as you erased the image out from under it).

Returns
None

2.1.9 errno

`errno()`

Returns the most recent error code from the **SNAPpy** Virtual Machine (VM), clearing it out as it does so.

If you receive an `UNSUPPORTED_OPCODE`, `UNRESOLVED_DEPENDENCY`, `BAD_GLOBAL_INDEX`, or `BAD_CONST_INDEX` error, please contact Synapse Wireless support and provide your **SNAPpy** script. These errors should not be reachable with a **SNAPpy** script, unless you are manipulating memory behind-the-scenes with pokes or with callable C strings.

The `EXCEEDED_MAX_BLOCK_STACK` and `EXCEEDED_MAX_OBJ_STACK` errors have been retired from **SNAPpy** code and can no longer be generated.

Returns
The most recent error code.

Return type
int

The possible error codes are:

Valu	Enu-mera-tion	Possible Cause
0	NO_ERRC	
1	OP_NOT_	Are you trying to add or compare things where that is not an option, such as adding two functions, or two Nones?
2	UN-SUP-PORTED_	
3	UNRE-SOLVED_	
4	IN-COM-PATI-BLE_TYP	Are you trying to add a number to a string?
5	TAR-GET_NOT	Are you trying to invoke `foo()`, but foo = 123?
6	UN-BOUND_L	Are you trying to use a variable before you put something in it?
7	BAD_GLC	
8	EX-CEEDED_	Are you calling a function recursively to too great a depth, or having too many nested layers of function calls?
9	EX-CEEDED_	
10	EX-CEEDED_	
11	IN-VALID_FU	Are you passing the wrong type of parameters to a function? Are you passing the wrong quantity of parameters?
12	UN-SUB-SCRIPT-ABLE_OB	Are you trying to slice or index something other than a string, tuple, or byte list?
13	IN-VALID_SU	Are you trying to access str[3] when str = 'ABC'? (When str = 'ABC', str[0] is 'A', str[1] is 'B', and str[2] is 'C'.)
14	EX-CEEDED_	Do you have too many local variables? Have you passed a large number of variables to nested levels of function calls?
15	BAD_CON	
16	AL-LOC_REF	Do you have more than 255 variables referencing the same buffer, such as a single string buffer? If so, as **SNAP** creates references to them with its one-byte reference counter, it will wrap past zero (causing an ALLOC_REF_OVERFLOW error), and as it releases references to them, it will wrap down past zero generating this error. (Debug builds only.)
17	AL-LOC_REF	Do you have more than 255 variables referencing the same buffer, such as a single string buffer? If so, as **SNAP** creates references to them with its one-byte reference counter, it will wrap past zero, generating this error, and as it releases references to them, it will wrap down past zero (generating an ALLOC_REF_UNDERFLOW error.) (Debug builds only.)
18	AL-LOC_FAIL	Are you trying to keep too many string results? As of version 2.2, you are no longer limited to a single buffer for each type of string operation, but they are still limited in number.
19	UN-SUP-PORTED_	Have you tried to save an unsupported type, such as a tuple or iterator, to an NV parameter, or are you trying to pass an unsupported type (tuple, iterator, or byte list) in a **SNAP** packet?
20	MAX_PAC	Are you passing too large of a string value?
21	MAX_STI	Have you created a dynamic string too large for your platform?
22	EX-CEEDED_	Your C code is trying to use more space for data than the firmware allows. Normally, this would be caught at compile time. However, if you get this error, the **SNAP** firmware has ignored your script and will not run it to prevent it from causing problems. To fix this, build your script using the firmware version currently running on

> **Note**
>
> If you are including C code in your **SNAPpy** script, your C code can use the pyerr macro to return one-byte error codes of your own definition. You can also use this as a way to return a second value from your C functions.

2.1.10 flowControl

`flowControl(`*uart*, *isEnabled*, *isTxEnable*`)`

Allows you to enable/disable flow control. Without flow control, there is a greater chance that characters will be dropped during communication; however, disabling flow control frees up two more pins (per UART) for use as other I/O. The initial state of flow control is set by bits 0x0080 (UART 1) and 0x0020 (UART 0) in *NV11 - Feature Bits*. By default, flow control is enabled. However, **SNAP** serial communications between nodes make no use of flow control.

When flow control is enabled, the **SNAP** node monitors the RTS pin from the attached serial device. As long as the **SNAP** node sees the RTS pin low, the node will continue sending characters to the attached serial device (assuming it has any characters to send). If the **SNAP** node sees the RTS pin go high, then it will stop sending characters to the attached serial device. By default, the **SNAP** node uses the CTS pin as a Clear To Send indication when flow control is enabled. The CTS pin indicates whether the **SNAP** node can accept more data. The CTS pin goes low if the **SNAP** node can accept more characters. The CTS pin goes high (temporarily) if the **SNAP** node is "full" and cannot accept any more characters. (The connected serial device can keep sending characters, but they will likely be dropped.)

It is important to realize that UART handshake lines are active-low. A low voltage level on the CTS pin is a boolean `False` but actually means that it is "Clear To Send." A high voltage level on the CTS pins is a boolean `True` but actually means it is not "Clear To Send." RTS behaves similarly.

The CTS pin can optionally act as a Transmit Enable (TXENA) indication using the `isTxEnable` parameter. In this mode, the CTS pin is normally high and transitions low before any characters are transmitted, remaining low until they have been completely sent.

When flow control is disabled, both the RTS and CTS pins are ignored.

Parameters

- `uartNum` (*int*) – Specifies the UART (0 or 1).
- `isEnabled` (*bool*) – Enables/disables hardware flow control.
- `isTxEnable` (*Optional [bool]*) – Controls CTS behavior when flow control is enabled. If `True`, CTS acts as a TXENA pin. If `False`, CTS acts as Clear To Send. (Default is `False`.)

Returns

None

> **Note**
>
> Remember that some **SNAP** nodes may have only one UART available. The RF266, based on the AT128RFA1, has two UARTS, but only UART1 comes out to pins on the module.

2.1.11 getChannel

`getChannel()`

Get the **SNAP** channel (0-15) for **SNAP** devices operating in the 2.4 GHz range, which corresponds to the 802.15.4 channel.

Returns
SNAP channel that the node is currently on.

Return type
int

Possible return values are:

SNAP Channel	802.15.4 Channel
0	11
1	12
2	13
3	14
4	15
5	16
6	17
7	18
8	19
9	20
10	21
11	22
12	23
13	24
14	25
15	26

2.1.12 getEnergy

`getEnergy()`

Returns a number indicating the result of a brief radio Energy Detection scan on the currently selected channel, providing an indication of the noise floor for radio energy at that frequency.

Returns
Energy detected on the current channel in (-) dBm.

Return type
int

See also

- *getLq()* returns the same units
- *scanEnergy()*

2.1.13 getI2cResult

`getI2cResult()`

Returns the status code from most recent I²C operation, clearing the value in the process.

Returns
Result of the most recently attempted I²C operation.

Return type
int

The possible return values are:

Value	Enumeration	Meaning
0	I2C_OFF	I²C was never initialized
1	I2C_SUCCESS	The most recent I²C operation succeeded
2	I2C_BUS_BUSY	I²C bus was in use by some other device
3	I2C_BUS_LOST	Some other device stole the I²C bus
4	I2C_BUS_STUCK	There is a hardware or configuration problem
5	I2C_NO_ACK	The slave device did not respond properly

Note

This function can only be used after function *i2cInit()* has been called.

See also

- User Guide on *I2C*

2.1.14 getInfo

`getInfo(`*whichInfo*`)`

Provides information about the RPC command currently being processed.

Parameters
`whichInfo` (*int*) – Specifies the type of information to be retrieved.

Value	Information Returned
0	*Vendor*
1	*Radio*
2	*CPU*
3	*Module Family*
4	*Build*
5	*Version* (Major)
6	*Version* (Minor)
7	*Version* (Build)
8	*Encryption*

continues on next page

Table 1 – continued from previous page

Value	Information Returned
9	*RPC Packet Buffer*
10	*Is Multicast*
11	*Remaining TTL*
12	*Remaining Small Strings*
13	*Remaining Medium Strings*
14	*Route Table Size*
15	*Routes in Route Table*
16	*Bank Free Space*
17	Reserved
18	*STDIN Hook Trigger*
19	*Remaining Tiny Strings*
20	*Remaining Large Strings*
21	*Script First Run*
22	*Script Base Address*
23	*Script Base Bank*
24	*Is Directed Multicast*
25	*Read and Reset Delay Factor* (Directed Multicast Only)
26	*Address Index* (Directed Multicast Only)
27	*Multicast Groups* (Multicast & Directed Multicast Only)
28	*Original TTL* (Directed Multicast Only)
29	*C Compiler ID*
30	*Build ID*

Returns
Varies based on the value of whichInfo.

Return type
int

Vendor
Indicates the manufacturer of the radio module on which **SNAP** is running. Possible return values for `getInfo(0)`:

Value	Manufacturer
0	Synapse
2	Freescale
3	CEL
4	ATMEL
5	Silicon Labs
7	PC
9	STMicrosystems

Radio
Indicates the method the node uses to connect to the rest of the network. Possible return values for `getInfo(1)`:

Value	Method
0	802.15.4-based 2.4 GHz
1	None (Serial only)
3	868 MHz
4	Powerline
5	900 MHz Frequency-Hopping
6	802.15.4-based 900 MHz

CPU

Indicates the processor paired with the radio in the **SNAP** module. Possible return values for `getInfo(2)`:

Value	Processor
0	Freescale MC9S08GT60A
1	ZIC 8051
2	MC9S08QE
3	Coldfire
4	ARM7
5	ATmega
6	Si100x 8051
7	X86
8	UNKNOWN
10	ARM CORTEX M3

Module Family

Indicates the module family. Possible return values for `getInfo(3)`:

Value	Module Family
0	Synapse RF100 SNAP Engine
3	CEL ZIC2410
5	MC1321x
6	ATmega128RFA1
7	SNAPcom
8	Si100x
9	MC1322x
11	Si100x KADEX
13	Synapse RF300 SNAP Engine
14	Synapse RF200 SNAP Engine or SNAPstick 200
15	Synapse SM300 Surface Mount Module
16	Synapse SM301 Surface Mount Module
17	Synapse SM200 Surface Mount Module
19	Synapse RF266
20	STM32W108xB
27	Synapse SM220 Surface Mount Module, RF220UF1 SNAP Engine, or SNAPstick 220
30	Synapse RF220SU SNAP Engine
31	Synapse RF220SU-EU SNAP Engine

Build

Indicates whether the firmware in your **SNAP** module is a *debug* or *release* build. Possible return values for `getInfo(4)`:

Value	Type	Trade-offs
0	debug	More error checking, slower speed, less **SNAPpy** room
1	release	Less error checking, faster speed, more **SNAPpy** room

Version

Indicates the Major, Minor and Build components of the firmware version. By using `getInfo(5)`, `getInfo(6)`, and `getInfo(7)`, you can retrieve all three digits of the firmware version number.

Encryption

Indicates the type of encryption that is **available** in the module; however, it does not indicate what encryption (if any) is enabled for the module. Possible return values for `getInfo(8)`:

Value	Encryption Support
0	Deprecated
1	AES-128
2	Basic encryption

RPC Packet Buffer

After you make an RPC call, a call to `getInfo(9)` returns an integer indicator of the packet buffer number used for the RPC call. That integer can be used in a function hooked to the *HOOK_RPC_SENT* event to determine that the processing of the packet buffer is complete. See the *HOOK_RPC_SENT* details for more information.

Is Multicast

Indicates how the RPC command currently being processed was invoked. Possible return values for `getInfo(10)`:

Value	Meaning
0	Received via an addressed RPC command or was triggered by a system hook
1	Received via a multicast or directed multicast

Remaining TTL

Indicates how many "hops" the RPC command currently being processed had left before its end-of-life. You can use this information to tune your TTL values for your network to reduce broadcast chatter. This value is valid for both multicasts and directed multicasts, but not for addressed RPC commands.

Remaining Small Strings

Indicates how many "small" string buffers remain unused in your node. The size and number of small strings available on your node will vary depending on the underlying hardware and firmware.

Remaining Medium Strings

Indicates how many "medium" string buffers remain unused in your node. The size and number

of medium strings available on your node will vary depending on the underlying hardware and firmware.

See also

Refer to platform-specific

Route Table Size

Indicates how many other nodes your node can keep track of in its address book. When a node needs to talk to another node, it must ask where that node is. It will find one of three things: that it can talk to the node directly, that it must communicate through another node, or that it cannot find the node at all. In these first two cases, **SNAPpy** keeps track of the path used to contact the node in a route table, so the next time it talks to the same node it does not have to query how to find the node first. How long the path to a node is kept depends on the mesh routing timeouts defined in the following NV parameters:

- *NV20 - Mesh Maximum Timeout*
- *NV21 - Mesh Minimum Timeout*
- *NV22 - Mesh New Timeout*
- *NV23 - Mesh Used Timeout*
- *NV24 - Mesh Delete Timeout*

See also

User Guide on *Preserving Unicast Routes*

Routes in Route Table

Indicates how many of the routes in the node's route table are in use, meaning how many other nodes the current node knows how to access without having to first perform a route request.

Bank Free Space

This query is only supported on platforms that support OTA Firmware Upload (it returns a value of 0 on the other platforms). This is how **Portal** knows if a given firmware image (.sfi file) will "fit" into the node.

STDIN Hook Trigger

Indicates what kind of data input event occurred that would have triggered *HOOK_STDIN*. Possible return values for `getInfo(18)`:

Valu	Meaning
0	In line mode, an end-of-line character (either `\x0a` or `\x0d`) was received, so the receive buffer contains a complete data transmission (less the EOL character, which is truncated) from the data source.
1	In line mode, the receive buffer received more data than it could hold, so the data received is not (necessarily) a complete transmission from the data source.
2	In character mode, the receive buffer received one character (or, depending on the incoming data rate, possibly more than one character).

See the *stdinMode()* function definition for a clarification of the difference between line mode and character mode.

Warning

Calling `getInfo(18)` from someplace other than a function hooked to *HOOK_STDIN* provides an undefined return value.

Remaining Tiny Strings

New in version **SNAP 2.6**.

Indicates how many "tiny" string buffers remain unused in your node. The size and number of tiny strings available on your node will vary depending on the underlying hardware and firmware.

Remaining Large Strings

New in version **SNAP 2.6**.

Indicates how many "large" string buffers remain unused in your node. The size and number of large strings available on your node will vary depending on the underlying hardware and firmware.

Script First Run

New in version **SNAP 2.6**.

Indicates whether the node has been rebooted since the last time a script was loaded onto it. Possible return values for `getInfo(21)`:

Valu	Meaning
0	There is no script on the node (either because it was empty when booted or because the script has been erased), or the node was rebooted since the script was loaded onto it.
1	The script has not been rebooted since the script was loaded onto it.

Script Base Address

New in version **SNAP 2.6**.

Indicates the base address for where the script resides in Flash.

Script Base Bank

New in version **SNAP 2.6**.

Indicates the base bank (zero-based) for where the script resides in Flash. For example, ATmega128RFA1's Flash has two banks of 64K each and the **SNAPpy** script, by default, is located in the lower 64K bank. Therefore, `getInfo(23)` will return zero when called.

Is Directed Multicast

New in version **SNAP 2.6**.

Similar to `getInfo(10)`, this indicates how the RPC command currently being processed was invoked and distinguishes between multicast and directed multicast. Possible return values for `getInfo(24)`:

Value	Meaning
0	Received via an addressed RPC command, a multicast command, or was triggered by a system hook
1	Received via a directed multicast

Read and Reset Delay Factor

New in version **SNAP 2.6**.

Indicates the staggered delay (in milliseconds) that should be applied before targeted nodes make any reply to the directed multicast, giving nodes an opportunity to respond in sequence, without interfering with each other's communications. See the description of the `delayFactor` parameter in the *dmcastRpc()* description for more information.

Invoking `getInfo(25)` clears the value, but also removes the staggered delay, putting the responsibility for avoiding communication collisions back in your hands. (There would be little reason to check this value without an intent to implement your own scheme to control communication timing.)

Warning

If called outside the context of a directed multicast, the return value is undefined.

Address Index

New in version **SNAP 2.6**.

Directed multicasts can target zero or more individual nodes by concatenating multiple addresses into the `dstAddrs` parameter. This indicates the (zero-based) position of this node's address in a directed multicast address list.

Warning

The value is only meaningful in the context of a directed multicast call that specifically targets multiple nodes. In any other context (an addressed RPC, a regular multicast, or a directed multicast with an empty string for the target list) the return value is undefined.

Multicast Groups

New in version **SNAP 2.6**.

Indicates the multicast group mask specified for the call to the function being run. This is valid for both multicasts and directed multicasts. If the function is invoked with an addressed RPC call, this will return zero.

See *mcastRpc()* for more information about multicast groups.

Original TTL

New in version **SNAP 2.6**.

Indicates the value set for the `ttl` parameter on the *dmcastRpc()* call that invoked the running function. Note that this value is different from the *Remaining TTL* value, which provides the TTL remaining when the receiving node gets the message.

> **Warning**
>
> If called outside the context of a directed multicast, the return value is undefined.

C Compiler ID

New in version **SNAP 2.7**.

Indicates a unique identifier for the compiler version used to create any C code that has been compiled into the current **SNAPpy** script image. This has little practical value for the user at runtime, but it may be valuable in troubleshooting failures in scripts that include C code.

Build ID

New in version **SNAP 2.7**.

Indicates a unique identifier for the firmware build in your node, combining the firmware compilation date, *Module Family*, *Build*, *Version* (major, minor, and build), and *Encryption* into one hashed value. This has little practical value for the user at runtime, but it is used by **SNAP** to ensure that a script containing compiled C code was appropriately created for the firmware on which it is being asked to run.

2.1.15 getLq

`getLq()`

Get the most recent link quality (received signal strength) of the most recently received packet, regardless of which node that packet came from. Remember that the last packet could have come from a node that is close by or one that is far away.

Returns
A number in the range 0-127 (theoretical), representing the link quality in (-) dBm.

Return type
int

Note

Because this value represents (-) dBm, lower values represent stronger signals, and higher values represent weaker signals.

2.1.16 getMs

`getMs()`

Get the value of a free-running counter within the **SNAP** Engine representing elapsed milliseconds since startup. Since the counter is only 16 bits, it rolls over every 65.536 seconds. Because all **SNAPpy** integers are signed, the counter's full cycle is:

```
0, 1, 2, ..., 32766, 32767, -32768, -32767, -32766, ..., -3, -2, -1, 0, 1, ...
```

Some scripts use this function to measure elapsed (relative) times. The value for this function is only updated between script invocations (events), meaning that you will get the same value no matter how many times you call `getMs()` during the same event.

Returns
16 bit count of milliseconds since startup.

Return type
int

2.1.17 getNetId

`getNetId()`

Get the current Network Identifier (ID).

The node will only accept packets containing this ID, or a special *wildcard* value of `0xffff` which is used during the "find nodes" process.

The Network ID and the Channel are what determine which radios can communicate with each other in a wireless network. Radios must be set to the same Channel and Network ID in order to communicate over the air. Nodes communicating over a serial link pay no attention to the Channel and Network ID.

Returns
The node's current Network ID.

Return type
int

See also

- *setNetId()*

2.1.18 getStat

`getStat(`*whichStat*`)`

This function returns details about how busy the node has been with processing packets. Each return value ranges from 0 to 255. The values "peg" at 255 (i.e., once reaching 255 they stay there until cleared). Reading a value resets the counter to 0.

Parameters
`whichStat` (*int*) – Specifies the counter to be retrieved.

Value	Counter Name	What is being counted?
0	Null Transmit Buffers	Buffers transmitted via a null serial port
1	UART0 Receive Buffers	Buffers received via UART0
2	UART0 Transmit Buffers	Buffers transmitted via UART0
3	UART1 Receive Buffers	Buffers received via UART1
4	UART1 Transmit Buffers	Buffers transmitted via UART1
5	Transparent Receive Buffers	Buffers received via transparent serial mode
6	Transparent Transmit Buffers	Buffers transmitted via transparent serial mode
7	Packet Serial Receive Buffers	Buffers received via packet serial mode
8	Packet Serial Transmit Buffers	Buffers transmitted via packet serial mode
9	Radio Receive Buffers	Buffers received via the radio
10	Radio Transmit Buffers	Buffers transmitted via the radio
11	Radio Forwarded Unicasts	Unicasts forwarded to nodes via the radio
12	Packet Serial Forwarded Unicasts	Unicasts forwarded to nodes via packet serial
13	Radio Forwarded Multicasts	Multicasts forwarded to nodes via the radio
14	Packet Serial Forwarded Multicasts	Multicasts forwarded to nodes via packet serial

Returns
Varies based on the value of `whichStat`.

Return type
int

2.1.19 i2cInit

`i2cInit(`*enablePullups*, *SCL_pin*, *SDA_pin*`)`

Performs the necessary setup for the I²C bus, including enabling internal pull-up resistors and optionally re-assigning the SCL and SDA I²C pins to another pair of **SNAPpy** IO pins.

The I²C clock and data lines require pull-ups. You can either choose to use your own external hardware or rely on the built-in internal ones. The internal pull-up resistors can come in handy when you are doing quick prototyping by dangling I²C devices directly off the **SNAP** Engine.

Parameters

- `enablePullups` (*bool*) – Internal pull-up resistors are enabled when `True`; disabled when `False`.
- `SCL_pin` (*Optional [int]*) – Specifies which **SNAPpy** IO pin to be used for SCL.
- `SDA_pin` (*Optional [int]*) – Specifies which **SNAPpy** IO pin to be used for SDA.

Returns
None

Warning

Be careful not to “double pull-up” the I²C bus!

See also

- Refer to platform-specific for **SNAPpy** IO numbers
- User Guide on *I2C*

2.1.20 i2cRead

`i2cRead`(*byteStr*, *numToRead*, *retries*, *ignoreFirstAck*)

Since I^{2}C devices must be addressed before data can be read out of them, this function performs a write followed by a read.

Parameters

- `byteStr` (*str*) – Specifies whatever "addressing" bytes must be sent to the device to get it to respond.
- `numToRead` (*int*) – Specifies how many bytes to read back from the external I^{2}C device.
- `retries` (*int*) – Controls a spin-lock count used to give slow devices extra time to respond. Try an initial value of 1 and increase if needed.
- `ignoreFirstAck` (*bool*) – `True` for devices that do not send an initial "ack" response, which will prevent an I^{2}C error based on lack of an initial acknowledgement; `False` otherwise.

Returns
Bytes read from the external I^{2}C device.

Return type
str

Note

This function can only be used after function *i2cInit()* has been called.

See also

- User Guide on *I2C*

2.1.21 i2cWrite

`i2cWrite`(*byteStr*, *retries*, *ignoreFirstAck*, *endWithRestart=False*)

Writes a byte string to the I^{2}C bus.

Parameters

- `byteStr` (*str*) – Specifies the data to be sent to the external I^{2}C device, including whatever "addressing" bytes must be sent to the device to get it to pay attention.

- retries (*int*) – Controls a spin-lock count used to give slow devices extra time to respond. Try an initial value of 1 and increase if needed.
- ignoreFirstAck (*bool*) – True for devices that do not send an initial “ack” response, which will prevent an I²C error based on lack of an initial acknowledgement; False otherwise.
- endWithRestart (*Optional [bool]*) – True for devices that expect an I²C restart between I²C commands; False for devices that expect an I²C Start - Stop handshake sequence. Default is False. This argument was added in **SNAP 2.5**.

Returns
Number of bytes actually written.

Return type
int

This function can only be used after function *i2cInit()* has been called.

- User Guide on *I2C*

2.1.22 imageName

imageName()

A **SNAPpy** script (.py file) gets compiled into a byte-code **SNAPpy** image (.spy file) using either **SNAPbuild** or **Portal**. The **SNAPpy** image name is assigned during this process from the underlying **SNAPpy** script. For example, image foo.spy would be generated from a script named foo.py. In this case, *imageName()* would return the string foo, even if someone had renamed the foo.spy file to a different name before loading it.

Returns
Name of currently loaded **SNAPpy** image.

Return type
str

This returns the **SNAPpy** image that gets downloaded into the node, not the original (textual) source code.

2.1.23 initUart

`initUart(`*uart*, *bps*, *dataBits=8*, *parity='N'*, *stopBits=1*`)`

Programs the specified UART to the specified bits per second (bps), or baud rate. Optionally, the data bits, parity, and stop bits can also be set.

Flow control defaults to the setting specified in *NV11 - Feature Bits*, typically it's 'On'. Use the *flowControl()* function to specify a setting.

Parameters

- `uartNum` (*int*) – The specified UART, 0 or 1.
- `bps` (*int*) – Usually set to the desired bits per second (1200, 2400, 9600, etc.); however, a value of 1 selects 115,200 bps (this large number would not fit into a **SNAPpy** integer and was treated as a special case). A value of 0 disables the UART.
- `dataBits` (*Optional [int]*) – 7 or 8. (Default is 8.)
- `parity` (*Optional [str]*) – 'E', 'O' or 'N', representing EVEN, ODD, or NO parity. (Default is 'N'.)
- `stopBits` (*Optional [int]*) – Number of stop bits. (Default is 1.)

Returns
None

Note

You are not limited to "standard" baud rates. If you need 1234 bps, you are allowed to do that.

See also

- *flowControl()*
- *NV11 - Feature Bits*
- Refer to platform-specific

2.1.24 initVm

`initVm()`

Calling this function restarts the **SNAPpy** virtual machine. It will clear all globals you may have changed in your **SNAPpy** code back to their initially declared values, and it will cause all static or global variables declared in C code in **SNAP 2.7** or newer to be reinitialized. It does not, however, reinvoke the **SNAPpy** script's "startup" handler (the function hooked to the `HOOK_START` event).

This function is normally only used by **Portal** and **SNAPconnect** at the end of the script upload process.

Returns
None

2.1.25 int

`int(obj)`

Converts the specified object (usually a string) into numeric form.

Parameters
`obj` – Object to transform into a number.

Returns
Numeric representation of `obj`.

Return type
int

Examples

```
int('123')       # Returns 123
int(True)    # Returns 1
int(False)   # Returns 0
```

> **Note**
>
> Unlike regular Python, the **SNAPpy** *int()* function does not take an optional second parameter indicating the numeric base to be used. The `obj` to be converted to a numeric value is required to be in base 10 (decimal).

2.1.26 len

`len(sequence)`

This function returns the size of parameter `sequence`. This will be an element count for a tuple or the number of characters in a string.

Parameters
`sequence` – Must be a string or a tuple.

Returns
Number of items in `sequence`.

Return type
int

Examples

```
len('ABC') # Returns 3
```

2.1.27 loadNvParam

`loadNvParam(`*id*`)`

This function reads a single parameter from the **SNAP** node's NV storage and returns it to the caller. For a full list of all the system (reserved) id values, refer to section 3. User parameters should have id values in the range 128-254.

Parameters

`id` (*int*) – Specifies which NV parameter to read.

Returns

Depends on the NV parameter.

See also

- *saveNvParam()*

2.1.28 localAddr

`localAddr()`

Get the node's local network address on the **SNAP** network.

Tracking what is essentially a six-digit (hexadecimal) number as a three-character string means that representations of the **SNAP** address may not appear meaningful when displayed. For example, if you were to use **Portal** to invoke *localAddr()* on a node with a a **SNAP** address of `5D.E3.AB`, **Portal**'s Event Log would display the return value as `]ã«`. If you were to display `ord("]")` or `ord(localAddr()[0])` you would get `93`, which is the decimal equivalent of hex `0x5D`. Similarly, `ord("ã")` yields `227`, which is the decimal equivalent of `0xE3`, and `ord("«")` yields `171`, which is the decimal equivalent of `0xAB`. You could build this string directly as `\x5d\xe3\xab`.

This string representation of numbers as their separate character strings is a fundamental part of specifying addresses of nodes for **SNAP** communications.

Returns

String representation of the node's 3-byte address on the **SNAP** network.

Return type

str

2.1.29 mcastRpc

`mcastRpc(`*dstGroups*, *ttl*, *remoteFunction*, **remoteFunctionArgs*`)`

Make a remote procedure call (RPC) using multicast messaging, meaning this message could be acted upon by multiple **SNAP** nodes.

Parameters

- `dstGroups` (*str*) – Specifies which nodes should respond to the request.
- `ttl` (*int*) – Specifies the Time To Live (TTL) for the request, which is basically how many hops the message is allowed to make before being discarded.
- `remoteFunction` (*str*) – Specifies which function to invoke on the remote node.

- `*remoteFunctionArgs` (*arbitrary argument list*) – Used if the remote function takes any parameters.

Returns

This function normally returns `True`; however, it does not mean your RPC request was successfully sent and received.

It returns `False` only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory or the RPC was too large to send).

Return type

bool

See also

- User Guide on *Remote Procedure Calls*

2.1.30 mcastSerial

`mcastSerial`(*dstGroups*, *ttl*)

Set serial transparent mode to multicast.

Parameters

- `dstGroups` (*int*) – Specifies the multicast groups that are eligible to receive this data.
- `ttl` (*int*) – Specifies the maximum number of hops the data will be re-transmitted.

Returns

None

Warning

Multicast serial transparent mode is less reliable than unicast serial transparent mode, because the received serial characters will be sent using unacknowledged multicast messages.

See also

- User Guide on *Wireless Serial*
- *ucastSerial()* - transmit to a single node

2.1.31 monitorPin

`monitorPin`(*pin*, *isMonitored*)

Enable background monitoring of a digital input pin's level.

When a **SNAPpy** IO pin changes state, a *HOOK_GPIN* event is sent to the **SNAPpy** virtual machine. If you have assigned a *HOOK_GPIN* handler, it will be invoked with the number of the **SNAPpy** IO pin that changed state and the pin's new value.

Parameters

- `pin` (*int*) – Specifies which **SNAPpy** IO pin to monitor.
- `isMonitored` (*bool*) – Enables monitoring when `True`, and disables monitoring when `False`.

Returns
None

Note

This **SNAPpy** IO pin must first be configured as a digital input pin using *setPinDir()*.

See also

- Refer to platform-specific for **SNAPpy** IO pin numbers
- User Guide on *Event-Driven Programming*
- *setRate()* - controls the sampling rate of the background pin monitoring

2.1.32 ord

`ord`(*str*)

Given a one-character string, this returns an integer of the ASCII for that character.

Parameters
`str` (*str*) – Specifies a single-character string to be converted.

Returns
Integer value of the ASCII character `str`.

Return type
int

Examples

::

The result of ord('A') is 65 (0x41). The result of ord('2') is 50 (0x32).

2.1.33 peek

`peek(addr)`

Read a byte from a specific memory location.

Parameters
addr (*int*) – Specifies which memory location to read (0-0x7FFF). A negative address will read a byte from internal EEPROM (-1 to -4096)

Returns
This returns the memory contents of the specified memory address

Return type
int

See also

- *poke()* for writing the contents of a specific memory address

2.1.34 poke

`poke(addr, byteVal)`

Write the value of a specific memory location.

Parameters

- addr (*int*) – Specifies which memory location to write (0-0x7FFF). A negative address will write a byte to internal EEPROM (-1 to -4096)
- byteVal (*int*) – Specifies the data value which will be written to the specified memory address

Returns
None

See also

- *peek()* for reading the contents of a specific memory address

2.1.35 pulsePin

`pulsePin`(*pin*, *msWidth*, *isPositive*)

Apply a pulse with a specified duration to a digital output pin.

The *pulsePin()* function behaves differently based on the value of `msWidth`:

- Specifying a positive value for `msWidth` will generate a non-blocking pulse, meaning your **SNAPpy** script continues to run in parallel. This also allows you to have multiple pulses in progress at the same time. In this case, `msWidth` specifies the desired pulse width in milliseconds (1-32767).
- Specifying a negative value for `msWidth` makes the pulse generation blocking, meaning the pulse runs to completion, and then your **SNAPpy** script resumes execution at the next line of code. In this case, the value of `msWidth` is platform-specific, but the desired pulse width is typically in microseconds. As a quick example, a value of `-10000` on a Synapse RF200 would result in a pulse that is 10 milliseconds wide.

If you have multiple pulses on the same pin with the same `isPositive` setting, the first pulse to complete will set the pin to its final state, and the second pulse to complete will effectively have ended.

Parameters

- `pin` (*int*) – Specifies which **SNAPpy** IO pin to pulse.
- `msWidth` (*int*) – Specifies the desired pulse width and determines if the function will return immediately or wait until the pulse is complete. Specifying a value of 0 will result in no pulse at all.
- `isPositive` (*bool*) – Controls the polarity of the pulse.

Returns

None

Note

This **SNAPpy** IO pin must first be configured as a digital output pin using *setPinDir()*.

Warning

The timing of the *pulsePin()* function is not guaranteed to be precise. Pulses begin when the function invokes and end at a tick of the node's internal clock. If you were to initiate a 1 ms pulse very shortly before the node's next millisecond tick, the pulse would would be notably shorter than the 1 ms specified.

See also

- Refer to platform-specific for **SNAPpy** IO pin numbers

2.1.36 random

`random()`

Returns a pseudo-random number.

Returns
A number between 0-4095.

Return type
int

2.1.37 readAdc

`readAdc(`*channel*`)`

Sample ADC on the specified analog input channel.

Some channels correspond to external analog input pins, the internal low voltage reference, or the internal high voltage reference. On some platforms, these can also return specialty readings such as processor temperature or voltage differences.

Parameters
`channel` (*int*) – Specifies which analog input channel to read.

Returns
Value of the specified analog input channel.

Return type
int

See also

- Refer to platform-specific for channel numbers

2.1.38 readPin

`readPin(`*pin*`)`

Reads the current level of either a digital input or digital output pin.

Parameters
`pin` (*int*) – Specifies which **SNAPpy** IO pin to read.

Returns
The current logic level of the specified pin. It returns a "live value" for an input pin or the last value written for an output pin.

Return type
bool

Note

This **SNAPpy** IO pin must first be configured as a digital input or output pin using *`setPinDir()`*.

See also

- Refer to platform-specific for **SNAPpy** IO pin numbers

2.1.39 reboot

`reboot(`*delay=200*`)`

Reboot the node after an optional delay.

Providing for a delay allows the node time to acknowledge the *reboot()* request (in case it came in over-the-air). The delay parameter is treated as an unsigned integer. Sending 0xEA60 provides 60,000 ms, or one minute, of delay, as negative values (-5,536 in this case) are not meaningful.

The delay parameter is available beginning in **SNAP 2.6**.

Parameters

`delay` (*Optional [int]*) – Specifies when the reboot is to occur, in milliseconds. (Default is 200.) New in **SNAP 2.6**.

Returns

None

Note

Once a reboot is issued, scheduling a reboot later than an already running countdown will have no effect. However, issuing a reboot with a shorter duration than the current countdown will work.

2.1.40 resetVm

`resetVm()`

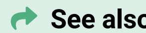
Reset the **SNAPpy** virtual machine ,which halts the currently running script. Even though the script is halted, it remains loaded in the node.

Returns

None

Note

Synapse tools like **Portal** and **SNAPconnect** use this in the script upload process.



See also

- *initVm()*

2.1.41 rpc

`rpc`(*nodeAddress*, *remoteFunction*, **remoteFunctionArgs*)

Request that another **SNAP** node execute a function.

Parameters

- `nodeAddress` (*str*) – Specifies the **SNAP** address of the target node.
- `remoteFunction` (*str*) – Specifies which function to invoke on the remote node.
- `*remoteFunctionArgs` (*arbitrary argument list*) – Used if the remote function takes any parameters.

Returns

This function normally returns `True`; however, it does not mean your RPC request was successfully sent and received.

It returns `False` only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory or the RPC was too large to send).

Return type

bool

See also

- User Guide on *Remote Procedure Calls*

2.1.42 rpcSourceAddr

`rpcSourceAddr()`

If a function on a node is invoked remotely (via RPC), then the called function can invoke function *rpcSourceAddr()* to find out the network address of the node which initiated the call. (If you call this function when an RPC is not in progress, it just returns None.)

This function allows a node to respond (answer back) directly to other nodes. An example will make this clearer. Imagine node "A" is loaded with a script containing the following function definition:

```
def pong():
    print 'got a response!'
```

Now imagine node "B" is loaded with a script containing the following function:

```
def ping():
    rpc(rpcSourceAddr(),'pong')
```

Node A can invoke function "ping" on node B. It can do this with a direct RPC, but it has to know node B's address to do so:

```
rpc(node_B_address_goes_here, 'ping')
```

Or, it can do it with a multicast RPC, assuming that node B's group membership is set appropriately:

```
mcastRpc(1, 1, 'ping')
```

When node B receives the RPC request packet, it will invoke local function "ping", which will generate the remote "pong" request. Notice that node B can respond to a "ping" request from any node.

All **SNAP** network addresses are three-byte strings. Please see the *localAddr()* function for a description of how **SNAP** addresses are specified and notated.

Returns:

See also

- User Guide on *Remote Procedure Calls*

2.1.43 rx

`rx`(*isEnabled*)

This function allows you to power down the radio, extending battery life in applications that do not actually need the radio (or only need it intermittently).

The radio defaults to *ON* in **SNAP** nodes. If you invoke `rx(False)`, the radio will be powered down. Invoking `rx(True)`, or sending any data over the radio, will power the radio back up.

To be clear, a node can wake up its own radio by attempting to transmit. A node's radio will not be woken up by transmissions from other nodes.

Parameters

`isEnabled` (*bool*) – Controls whether or not the radio is on.

Returns

None

Note

If you turn the radio off (using `rx(False)`), then you will not receive any more radio traffic!

2.1.44 saveNvParam

`saveNvParam`(*id*, *obj*, *bitmask*)

Save object to indexed non-volatile storage location.

Parameters

- `id` (*int*) – Specifies which NV parameter to modify.
- `obj` – The data to be saved. Depending on the NV parameter, the data type could be boolean, integer, string, byte list, or None.
- `bitmask` (*Optional [int]*) – Specifies which bits in `obj` will update the NV parameter's value. Omitting this argument will overwrite the previous value with `obj`. This argument should only be used with integer `obj` values. New in version **SNAP 2.6**.

Returns

A result code.

Return type

int

The possible return values are:

Value	Enumeration
0	NV_SUCCESS
1	NV_NOT_FOUND
2	NV_DEST_TOO_SMALL
3	NV_FULL
4	NV_BAD_LENGTH
5	NV_FAILURE
6	NV_BAD_TYPE
7	NV_LOW_POWER

Examples

Assume that *NV11 - Feature Bits* contains the value `0x001F`, which indicates that there is a power amplifier and that both UARTs are enabled. The following command will enable the Packet CRC bit (0x0400):

```
saveNvParam(NV_FEATURE_BITS_ID, 0x0400)          # NV11 = 0x0400
```

However, it will also overwrite all the other feature bits, disabling the power amplifier and both UARTs in the process. Instead, you need to combine the bit values:

```
saveNvParam(NV_FEATURE_BITS_ID, 0x041F)          # NV11 = 0x041F
```

Beginning in **SNAP 2.6**, you can use the optional `bitmask` argument:

```
saveNvParam(NV_FEATURE_BITS_ID, 0x0400, 0x0400) # NV11 = 0x041F
```

Note

NV parameters 128-254 are user-defined, so your script can use these parameters however you want. All values except `None` consume flash space. If you plan to no longer use a user-defined NV parameter, setting it to `None` will regain the flash space.

See also

- *loadNvParam()*
- API Reference for *NV Parameters*

2.1.45 scanEnergy

`scanEnergy()`

The *getEnergy()* function returns the result of a brief radio energy detection scan, as an integer. Function *scanEnergy()* is an extension of *getEnergy()*. It essentially calls *getEnergy()* N times in a row, changing the frequency before each *getEnergy()* scan. Here, 'N' refers to the number of frequencies supported by the radio.

For 2.4 GHz radios, 16 frequencies are supported by the radios, each corresponding to one channel. For 900 MHz radios running FHSS (frequency hopping) firmware, the 16 channels cover 66 radio frequencies, with each channel making use of 25 of those frequencies. For 868 MHz radios, there are three frequencies used, regardless of the channel selected. See the *getChannel()* function explanation for more details about how each radio platform uses the various frequencies available to it.

The *scanEnergy()* function returns an N-byte string, where the first character corresponds to the "detected energy level" on frequency 0, the next character corresponds to channel 1, and so on. (For 900 MHz FHSS radios, **SNAP** does not make use of the first and last frequencies but returns them as part of the string for completeness.) Thus, `ord(scanEnergy()[4])` would return the same integer that calling *getEnergy()* from channel 4 would return.

The units for the "detected energy level" are the same as that returned by *getLq()*. Refer to the documentation on that function for more info.

Returns:

2.1.46 setChannel

`setChannel`(*channel*, *network_id*)

Set the **SNAP** Channel, and optionally the **SNAP** Network ID, of the node.

Parameters

- `channel` (*int*) – Specify which frequency (or range of frequencies) the device should use for its communications. Values are platform specific.
- `network_id` (*Optional [int]*) – Specify the **SNAP** Network ID (0-0xFFFF); if excluded, the node's **SNAP** Network ID will be unchanged. New in version **SNAP 2.6**.

Returns

None

Example

```
setChannel(7 ,0x0C70)  # set the SNAP Channel to 7 and the SNAP Network ID to 0x0C70
```

Warning

This function only changes the "live" settings, so the effect lasts until the next reboot or power cycle. Use *saveNvParam()* to persist these settings into the correct NV Parameter.

Note

`channel` values 0-15 correspond to 802.15.4 channel 11-26 on 802.15.4/2.4 GHz devices.

See also

- Refer to platform-specific
- *getChannel()*
- *setNetId()*
- *NV3 - Network ID*
- *NV4 - Channel*

2.1.47 setNetId

`setNetId(`*netId*`)`

Set the SNAP Network ID of the node.

The combination of network ID and channel are what determine which radios can communicate with each other in a wireless network. Radios must be set to the same channel and network ID in order to communicate over the air. (Of course, they also must be transmitting in the same frequency band. 900 MHz radios set to a given channel cannot communicate over the air with 2.4 GHz radios set to the same channel.) Nodes communicating over a serial link pay no attention to the channel and network ID.

Parameters

`netId` (*int*) – Specify the **SNAP** Network ID (0-0xFFFF).

Returns

None

Note

- 0xFFFF is considered a "wildcard" network ID (matches all nodes), and you normally should only use network IDs of 0-0xFFFE.
- This function changes the "live" network ID setting, and the effect only lasts until the next reboot or power cycle, or until *setNetId()* is called again. You should also use *saveNvParam()* to save the "persisted" network ID setting in *NV3 - Network ID*, if you want the node to stay on that network ID after its next reboot.

See also

- *NV11 - Feature Bits*
- *getNetId()*
- *setChannel()*

- *saveNvParam()*

2.1.48 setPinDir

`setPinDir`(*pin*, *isOutput*)

Configures a **SNAPpy** IO pin as either a digital input or digital output.

Parameters

- `pin` (*int*) – Specifies which **SNAPpy** IO pin to configure.
- `isOutput` (*bool*) – Pin is output when `True`; input when `False`.

Returns

None

Note

- Calling setPinDir with a pin that is in use by another peripheral (eg UART or ADC) will automatically disable it.

See also

- Refer to platform-specific for **SNAPpy** IO pin numbers

2.1.49 setPinPullup

`setPinPullup`(*pin*, *isEnabled*)

Enables the internal pull-up resistor for an IO pin.

Parameters

- `pin` (*int*) – Specifies which **SNAPpy** IO pin to configure.
- `isEnabled` – Enables the **SNAPpy** IO pin's internal pull-up when `True`; disables when `False`.

Returns

None

Note

- A **SNAPpy** IO pin's internal pull-up is disabled by default.
- This **SNAPpy** IO pin must first be configured as a digital input pin using *setPinDir()*.

See also

- Refer to platform-specific for **SNAPpy** IO pin numbers

2.1.50 setPinSlew

`setPinSlew`(*pin*, *isRateControl*)

Enable slew rate-control (~30ns) for a digital output pin.

Parameters

- `pin` (*int*) – Specifies which **SNAPpy** IO pin to configure.
- `isRateControl` – Enables the **SNAPpy** IO pin's slew rate-control when `True`; disables when `False`.

Returns

None

Note

- A **SNAPpy** IO pin's slew rate-control is disabled by default.
- This **SNAPpy** IO pin must first be configured as a digital output pin using *`setPinDir()`*.

See also

- Refer to platform-specific for **SNAPpy** IO pin numbers

2.1.51 setRadioRate

`setRadioRate`(*rate*)

Set the data rate of the radio. Only nodes set to the same rate can talk to each other over the air! All radios on the same frequency range and set to rate 0 will be interoperable.

Parameters

`rate` (*int*) – 0 specifies the standard data rate for the platform. Other rate values are platform-specific.

Returns

None

Warning

The "encoding" for non-standard data rates may differ between radio manufacturers. This means that different radio hardware may not be interoperable, even if set to the same (non-standard) rate.

See also

- Refer to platform-specific for `rate` values

2.1.52 setRate

`setRate`(*rateNum*)

Adjusts the background sampling rate of digital IO pins.

Parameters
`rateNum` (*int*) – Specifies the background sampling rate.

Returns
None

The possible values for `rateNum`:

Value	Rate
0	OFF
1	Every 100 ms
2	Every 10 ms
3	Every 1 ms

Note

- The background sampling rate is every 100 milliseconds by default.
- This function has no effect unless/until you are using the *monitorPin()* function.

See also

- *monitorPin()*

2.1.53 sleep

`sleep`(*mode*, *ticks*)

This function puts the radio and CPU on the **SNAP** node into a low-power mode for a specified number of ticks. This is used to extend battery life.

A `ticks` parameter of 0 can be used to sleep until an IO pin interrupt occurs (see script pinWakeup.py), but **SNAPpy** is smart enough to know if you have not enabled a wakeup pin and will ignore a `sleep(mode, 0)` if there is no wakeup possible. Note that on some platforms not all processor pins come out to module pins; it might be possible to specify a wake pin that is not accessible, thus locking your node into a sleeping state. If this happens, make a serial connection to the node and use **Portal** 's Erase SNAPpy Image… menu option to clear the script and start over.

Starting with version **SNAP** 2.2, a negative `ticks` parameter can be used to access alternate sleep timings.

Starting with version **SNAP** 2.4.24, for most platforms the sleep function returns the remaining number of ticks if the node has been woken by a pin interrupt rather than the sleep call timing out. If the complete timed sleep has occurred, or if the sleep was untimed, the function returns zero. For example if you were to invoke `sleep(1, 60)` on a node but woke the node with an interrupt after 20 seconds, the function would return 40.

Parameters

- `mode` – Chooses from the available sleep modes. The number of modes available and their characteristics, such as resolution and accuracy, is platform-specific.
- `ticks`

Returns:

See also

- Refer to platform-specific

2.1.54 spiInit

`spiInit`(*clockPolarity*, *clockPhase*, *isMsbFirst*, *isFourWire*)

Initializes the Serial Peripheral Interface (SPI) Bus for the **SNAP** node.

Parameters

- `clockPolarity` (*bool*) – Specifies the level of the CLK pin between SPI exchanges. `True` specifies that the clock is high when idle, and `False` specifies that the clock is low when idle.
- `clockPhase` (*bool*) – Specifies which clock edge the incoming data will be latched in. If you number clock edges from 1, then `True` specifies the even clock edges for incoming data, and `False` specifies the odd clock edges for incoming data.
- `isMsbFirst` (*bool*) – Controls the order in which individual bits within each byte will be shifted out. Setting this parameter to `True` will make the 0x80 bit go out first, and setting this parameter to `False` will make the 0x01 bit go out first.
- `isFourWire` (*bool*) – Select the variant of SPI you are connecting to: `True` is four-wire, and `False` is three-wire.

Returns

None

Note

These settings depends on the device to which you are interfacing.

See also

- User Guide on *SPI*
- *spiInit()*

- *spiWrite()*
- *spiRead()*
- *spiXfer()*

2.1.55 spiRead

`spiRead`(*byteCount*, *bitsInLastByte=8*)

Reads data from a three-wire SPI device.

Parameters

- `byteCount` (*int*) – Specifies how many bytes to read.
- `bitsInLastByte` (*int*) – Makes it possible to accommodate devices with data widths that are not multiples of 8 bits. Value should be equal to the data width of device modulo 8. (Default is 8.)

Returns

Bytes received from SPI.

Return type

str

Note

This function can only be used after function *spiInit()* has been called.

See also

- User Guide on *SPI*
- *spiXfer()* - if using four-wire SPI

2.1.56 spiWrite

`spiWrite`(*byteStr*, *bitsInLastByte=8*)

Writes data to a three or four-wire SPI device.

Parameters

- `byteStr` (*str*) – Specifies the actual bytes to be shifted out.
- `bitsInLastByte` (*int*) – Makes it possible to accommodate devices with data widths that are not multiples of 8 bits. Value should be equal to the data width of device modulo 8. (Default is 8.)

Returns

None

Note

This function can only be used after function *spiInit()* has been called.

See also

- User Guide on *SPI*
- *spiXfer()* - to write and read data simultaneously (four-wire SPI only)

2.1.57 spiXfer

`spiXfer`(*byteStr*, *bitsInLastByte=8*)

Bidirectional data transfer over a four-wire SPI device. As bits are being shifted out to the slave device on the MOSI pin, bits from the slave device on the MISO pin are simultaneously being shifted in.

Parameters

- `byteStr` (*str*) – Specifies the actual bytes to be shifted out.
- `bitsInLastByte` (*int*) – Makes it possible to accommodate devices with data widths that are not multiples of 8 bits. Value should be equal to the data width of device modulo 8. (Default is 8.)

Returns

Byte string consisting of the bits that were shifted in (as the bits specified by parameter `byteStr` **were**
shifted out).

Return type

str

Note

This function can only be used after function *spiInit()* has been called.

See also

- User Guide on *SPI*
- *spiWrite()* - if using a write-only device
- *spiRead()* - if using a read-only device

2.1.58 stdinMode

`stdinMode`(*mode*, *echo*)

This function controls whether or not received characters are echoed back to the user and how serial data is presented to your **SNAPpy** script via the *HOOK_STDIN* handler.

You can choose between **line mode** or **character mode**:

Line Mode
Characters are buffered up until either a Carriage Return (CR) or Line Feed (LF) is received. The complete string is then passed to your **SNAPpy** script via the *HOOK_STDIN* handler. Either the CR or LF character can trigger the hand-off, so if your terminal (or terminal emulator) is automatically adding extra CR or LF characters, you will see additional empty strings ("") passed to your script. For example, the character sequence `A B C CR LF` (or `\x41\x42\x43\x0d\x0a`) looks like two lines of input to **SNAPpy**.

Character Mode
Characters are passed to your **SNAPpy** script as soon as they become available via the *HOOK_STDIN* handler. If characters are being received fast enough, it still is possible for your script to receive more than one character at a time; they are just not buffered waiting for a CR or LF.

Parameters

- `mode` (*int*) – `0` for line mode; `1` for character mode.
- `echo` (*bool*) – `True` will retransmit any received characters to the sender.

Returns
None

Note

While your node is in **line mode**, **SNAP** reserves one "medium" string buffer to accept incoming data from STDIN. If your script is heavy on string usage but does not make use of *HOOK_STDIN*, you can recover use of the medium string used by **line mode** by changing to **character mode**.

See also

- *HOOK_STDIN*

2.1.59 str

`str`(*obj*)

Given an element, returns a string representation of the element.

Parameters
`obj` – Element to be transformed into a string.

Returns
String representation of the element.

Return type
str

Examples

```
str(True)    # returns 'True'
str(123)     # returns '123'
str('hello') # returns 'hello'
```

2.1.60 txPwr

`txPwr(power)`

The radio on the **SNAP** node defaults to the maximum power allowed. Function *txPwr()* lets you reduce the power level from this default maximum.

Parameter `power` specifies a transmit power level from 0-17, with 0 being the lowest power setting and 17 being the highest power setting. On some platforms, government regulations prevent the hardware from being usable above certain levels. On one of those platforms, setting the transmit power higher than allowed pegs the power output at the highest allowed amount, so there may be no practical difference between setting the power to 16 and setting it to 17, for example.

Parameters
`power` (*int*) – The TX power level, in the range 0-17.

Returns
None

See also

- Refer to platform-specific

2.1.61 type

`type(arg)`

Given a single argument, it returns an integer indicating the data type of the argument.

This function does not distinguish between constant or dynamic variables, nor between user functions or **SNAPpy** built-in functions.

Parameters
`arg` – Object to inspect.

Returns
Integer indicating the type of the object passed to it.

Return type
int

The expected return values (different from that returned in Python) are:

0	None
1	Integer
2	String
3	Function
5	Boolean
6	Tuple
7	Iterator
8	Byte List
31	Unknown

Note

Typically, you will not be able to create variables of an unknown type. However, if you have a Byte List variable and you use the `del` keyword to delete the entire list, the variable will be left with contents of an unknown type. Removing all elements from a list does not set the variable to an unknown type. Only deleting the list itself will do that.

2.1.62 ucastSerial

`ucastSerial(`*dstAddr*`)`

Set serial transparent mode to unicast.

Parameters
`dstAddr` – Specifies the **SNAP** address of the destination node.

Returns
None

See also

- User Guide on *Wireless Serial*
- *mcastSerial()* - transmit to multiple nodes

2.1.63 uniConnect

`uniConnect(`*dst*, *src*`)`

Establishes a one-way connection between two **SNAP** data-sources.

Parameters

- `dst` – **SNAP** data destination.
- `src` – **SNAP** data source.

Returns
None

See also

- User Guide on *The Switchboard* for details
- *crossConnect()*

2.1.64 vmStat

`vmStat`(*statusCode*, *args*)

This function is specialized for management applications like **Portal** and provides a range of control/callback functionality.

Parameters

- `statusCode` (*int*) – Controls what actions will be taken and what data will be returned via `tellVmStat()`.
- `args` – Varies depending on the `statusCode`. See below for details.

Returns

This function does not return a value, but it causes a `tellVmStat()` call to be made to the node that requested the *vmStat()*.

The currently supported `statusCode` values are:

0-3	RESERVED	Internal Use Only
4	*VM_NVRE*	Read and return the specified NV parameter
5	*VM_NAME*	Returns NODE NAME if set, else IMAGE NAME, plus Link Quality
6	*VM_VERSI*	Returns software version number
7	*VM_NET*	Returns Network ID and Channel
8	*VM_SPACI*	Returns Total Image (script) Space Available
9	*VM_SCAN*	Scans all RF channels for energy
10	*VM_INFO*	Returns Image Name (script name) and Link Quality
11	*VM_DATA_*	Returns the amount of space available for a SNAPpy image in auxiliary memory (valid only for the Si1000-based SNAP modules RF300, RF301, SM300, SM301)
12	*VM_TOPO*	Returns a string providing a list of nodes with which the target node can directly communicate

After the "varying" `args` parameter comes a final optional argument that specifies a "reply window" (in seconds) to randomly reply within. This helps prevent communication interference if you send a *vmStat()* command to multiple nodes concurrently.

If you do not specify a "reply window" parameter (or specify zero), the nodes will respond as rapidly as they can (subject to network traffic, communication difficulties, etc.).

Some of these commands are multicast by **Portal**; the "reply window" provides a way to keep all of the nodes from trying to respond at once. Specifying a non-zero "reply window" tells the node to pick a random time within the next "reply window" seconds and wait until then to reply.

Return Value Format:

When a node receives a *vmStat()* call, it responds to the source node with a call to `tellVmStat(word, data)`.

The least significant byte of `word` will be the originally requested `statusCode`. The most significant byte will vary depending on the `statusCode` and is the "hiByte" described below. The `data` value is the main return value and is also dependant on the `statusCode`. For example, if a `vmStat(4, 15, 3)` were called, the called node would, within 3 seconds, reply with `tellVmStat(3844, {contents of NV parameter 15 on the node})`. Note that the second parameter could be an integer, a string, a Boolean, or None.

The 3844 value that was returned as `word` equates to 0x0F04. The "hiByte" of this value, 0x0F, indicates which NV parameter was read. The "lowByte" of this value, 0x04, indicates that the returned value comes as a result of a VM_NVREAD call. If the value of the first returned parameter is greater than 10, the "lowByte" will always contain the calling code, while the significance of the "hiByte" will vary depending on what call was made.

VM_NVREAD

For VM_NVREAD, the `args` parameter is the ID of the NV parameter you want to read. (These are the same IDs used in the *saveNvParam()* and *loadNvParam()* functions. The "system" NV parameter IDs are given in section 3.) You can also optionally specify a "reply window."

The reported values will be a "hiByte" of the NV parameter ID and a "data" of the actual NV parameter value.

VM_NAME

For VM_NAME, the only parameter is the optional reply window.

The reported "data" value will be a string name and a Link Quality reading.

VM_VERSION

For VM_VERSION, the only parameter is the optional reply window.

The reported "data" value will be a version number string.

VM_NET

For VM_NET, the only parameter is the optional reply window.

The reported values will be a "hiByte" containing the currently active channel, and a "data" value of the current Network ID.

VM_SPACE

For VM_SPACE, the only parameter is the optional reply window.

The reported "data" value will be the Total Image (script) Space Available.

VM_SCAN

For VM_SCAN, the only parameter is the optional reply window.

The reported "data" value will be a string containing the detected energy levels on all channels. This value is identical to the value returned by the *scanEnergy()* function. Note that each scan just represents one point in time. You will probably have to initiate multiple scans to determine which channels actually have **SNAP** nodes on them.

You can see this VM_SCAN function put to use in the Channel Analyzer feature of **Portal**.

VM_INFO

For VM_INFO, the only parameter is the optional reply window.

The reported values will be a "hiByte" of the current Link Quality and a "data" of the currently loaded script name (a string).

VM_DATA_SPACE
For VM_DATA_SPACE, the only parameter is the optional reply window.

The reported values will be a "hiByte" of the current Link Quality and a "data" of the space available in auxiliary memory for a **SNAPpy** script (an integer).

VM_TOPOLOGY
For VM_TOPOLOGY, the only parameter is the optional reply window.

The reported values will be a "hiByte" of the current Link Quality and a "data" of details about nodes with which the reporting node can directly communicate (a string). Each node with which the reporting node can directly communicate will be represented by a block of five characters: The first three will be the node's address, the fourth will be the link quality with which the neighboring node heard the reporting node, and the fifth will be the link quality with which the reporting node heard the neighboring node reply. Thus data[0:3] would be the address of one node and data[5:8] would be the address of another node (assuming that the reporting node could communicate with two neighbors).

It is possible that one call to `vmStat(12)()` may result in multiple calls back to `tellVmStat()` from a given node if that node has many neighbors with which it can directly communicate. In dense networks, it may be necessary to include the "reply window" parameter in order to provide enough time for all nodes to respond.

This option for *vmStat()* was added in **SNAP 2.6**, and the reporting node must have version **SNAP 2.6** (or later) firmware installed. However, neighboring nodes can have earlier firmware versions on them and still report correctly.

> **Note**
>
> Because of the *callback()* function, some of the *vmStat()* capabilities are redundant.

2.1.65 writePin

`writePin`(*pin*, *isHigh*)

Sets the current level of a digital output pin.

Parameters

- `pin` (*int*) – Specifies which **SNAPpy** IO pin to configure.
- `isHigh` (*bool*) – Sets the **SNAPpy** IO pin high when `True`; low when `False`.

Returns
None

> **Note**
>
> This **SNAPpy** IO pin must first be configured as a digital output pin using *setPinDir()*.

See also

- Refer to platform-specific for **SNAPpy** IO pin numbers
- *setPinSlew()* - controls how quickly the pin will transition to a new value

2.1.66 xrange

xrange(*args*)

New in version **SNAP 2.6**.

Creates a sequence iterator.

Parameters

args – Given a single integer as an argument, returns an iterator that yields integers beginning with zero and stepping up to one less than the passed argument. Given two integers, it returns an iterator that yields integers beginning with the first argument and steps up to one less than the second argument. Given three integers, it returns an iterator that yields integers beginning with the first argument and steps toward (but stops just before reaching or passing) the second argument, stepping in increments of the third argument.

Returns

An iterable that can be stepped through using a `for` loop.

Examples

There are three ways the function can be called:

```
for i in xrange(3):
    print i, # Prints 012

for i in xrange(3, 6):
    print i, # Prints 345

for i in xrange(3, 9, 2):
    print i, # prints 357
```

Note

Remember that **SNAPpy** integers are 16-bit signed numbers. Unlike a `while` loop that continually increments a counter until the stop value is reached, *xrange()* will generate an iterable that yields no numbers if the start value is greater than the stop value, unless the step value is negative.

2.2 Function Decorators

2.2.1 @setHook

`@setHook(hook)`

This decorator should precede the definition of any function to be triggered by a "hooked" event in the **SNAP** environment, such as when a node reboots or when a pin changes state.

Parameters
`hook` – Value of one of the *SNAP Hooks*

See also

- User Guide on *Event-Driven Programming*
- List of *SNAP Hooks*

2.3 NV Parameters

2.3.1 NV0-1 – Reserved

Reserved for Synapse use.

2.3.2 NV2 - MAC Address

`NV_MAC_ADDR_ID = 2`

The eight byte address of the SNAP node.

Not Modified on Factory Default

2.3.3 NV3 - Network ID

`NV_NETWORK_ID = 3`

The 16-bit Network Identifier of the SNAP node. The Network ID and *NV4 - Channel* are what determine which radios can communicate with each other in a wireless network. Radios must be set to the same channel and network ID in order to communicate over the air. Nodes communicating over a serial link pay no attention to the channel and network ID.

Network IDs can be set to any value from 0x0000 through 0xFFFF. However, 0xFFFF should generally be avoided, because it is a wildcard value which responds to all nodes and to which all nodes respond.

Factory Default Value = `0x1C2C`

Note

Changes to this parameter do not take effect until the node has been rebooted. For an immediate change of the Network ID, use **SNAPpy**'s *setNetId()* function, which changes the "live" value but does not persist through a reboot.

See also

- *NV4 - Channel*
- *setNetId()*

2.3.4 NV4 - Channel

`NV_CHANNEL_ID = 4`

The channel on which the SNAP node broadcasts.

The channel can be set to any value from 0 to 15. The Channel Analyzer in **Portal** can help you determine which channel has the least traffic on it in your environment. Some hardware platforms may restrict the broadcast power on certain channels.

Factory Default Value = `4`

Note

Changes to this parameter do not take effect until the node has been rebooted. For an immediate change of the channel, use **SNAPpy**'s *setChannel()* function, which changes the "live" value but does not persist through a reboot.

See also

- Refer to platform-specific
- *NV3 - Network ID*
- *setChannel()*

2.3.5 NV5 - Multicast Process Groups

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".

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 **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 - 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.

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

Warning

Removing a node from group 0x0001 (the default/broadcast group) will make the node unable to respond to **Portal**'s multicasts, such as global pings.

See also

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

2.3.6 NV6 - Multicast Forward Groups

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." By default, all nodes process and forward only group 0x0001 (broadcast) 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.

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

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

This parameter is ignored if *NV78 - Multicast Serial Forwarded Groups* is changed from its default of `None`. In that case, *NV78 - Multicast Serial Forwarded Groups* will control which packets will be forwarded over both the serial and radio interfaces for the node.

See also

- User Guide section on *Multicast Groups*
- *NV5 - Multicast Process Groups*
- *NV78 - Multicast Serial Forwarded Groups*

2.3.7 NV7 - Reserved

Reserved for Synapse use.

2.3.8 NV8 – Device Name

`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 first detected script name will determine the node name. If this parameter is blank and the node has no script loaded, it will have "Node" as its name.

Factory Default Value = `None`

Note

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

2.3.9 NV9 – Reserved

Reserved for Synapse use.

2.3.10 NV10 - Device Type

`NV_DEVICE_TYPE_ID = 10`

This is a user-definable string that can be read by scripts. This allows a single script to fill multiple roles, by giving it a way to determine what type of node it is running on. Like *NV8 – Device Name*, this parameter is one way to categorize your nodes.

Factory Default Value = `None`

2.3.11 NV11 - Feature Bits

`NV_FEATURE_BITS_ID = 11`

Settings to control miscellaneous hardware.

Feature Bit Name	Hex	Binary
UART0	0x0001	b0000.0000.0000.0001
Flow Control UART0	0x0002	b0000.0000.0000.0010
UART1	0x0004	b0000.0000.0000.0100
Flow Control UART1	0x0008	b0000.0000.0000.1000
Power Amplifier	0x0010	b0000.0000.0001.0000
External Power-down Output	0x0020	b0000.0000.0010.0000
Alternate Clock Source	0x0040	b0000.0000.0100.0000
DS_AUDIO	0x0080	b0000.0000.1000.0000
Second CRC	0x0100	b0000.0001.0000.0000
TX Power Levels	0x0200	b0000.0010.0000.0000
Packet CRC	0x0400	b0000.0100.0000.0000
Large Route Table	0x0800	b0000.1000.0000.0000

UART0

The UART0 bit (0x0001) enables Serial Port 0.

Flow Control UART0

The Flow Control UART0 bit (0x0002) enables hardware flow control on Serial Port 0.

UART1

The UART1 bit (0x0004) enables Serial Port 1.

Flow Control UART1

The Flow Control UART1 bit (0x0008) enables hardware flow control on Serial Port 1.

Power Amplifier

Synapse RF100 **SNAP** engines with PA hardware can be identified by the “RFET” on their labels. Units without PA hardware say “RFE” instead of “RFET.”

For RF100 **SNAP** engines, the Power Amplifier bit (0x0010) should only be set on “RFET” units. Setting this bit on an “RFE” board will not harm the **SNAP** engine, but it will actually result in lower transmit power levels (a 20-40% reduction). The bit should be set for **RF200 Series SNAP** engines, as well.

In custom hardware, you may need to set this bit according to your specific hardware configuration.

Factory Default Value = Not Modified on Factory Default

External Power-down Output

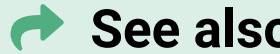
The External Power-down Output bit (0x0020) should be set on units that need to power down external hardware before going to sleep and power it back up after they awake.

Factory Default Value = Not Modified on Factory Default

Alternate Clock Source

The Alternate Clock Source bit (0x0040) modifies which timer is used on **SNAP** modules that have multiple timers available, for increased PWM flexibility. Not available on all platforms.

Factory Default Value = Not Modified on Factory Default



See also

- Refer to platform-specific

DS_AUDIO

The DS_AUDIO bit (0x0080) enables I^{2}S audio communications over the **SNAP** network on platforms that support it. Not available on all platforms.

Factory Default Value = 0

See also

- Refer to platform-specific

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.

Factory 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.

See also

- Refer to platform-specific

TX Power Levels

The TX Power Levels bit (0x0200) enables ETSI, instead of FCC (US), transmit power restrictions. Not available on all platforms.

Factory Default Value = 0

See also

Refer to platform-specific

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 it 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.

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** or your **SNAPconnect** application with this

setting to prevent generating packets that exceed the new maximum payload in your network.

- Enabling this feature will increase the processor load of the node.

Factory Default Value = 0

Large Route Table

New in version SNAP 2.7.

The Large Route Table bit (0x0800) changes the size of the route table from 10 entries to 100 entries, allowing your node to maintain route information for more nodes so it can more easily communicate over your mesh network.

Factory Default Value = 0

Note

Between the release of SNAP 2.4.34 and **SNAP 2.6**, route table size was controlled by a platform-specific feature bit on nodes based on the **Atmel AT128RFA1**.

See also

- Refer to platform-specific
- User Guide for *Preserving Unicast Routes*
- *getInfo(14)*
- *getInfo(15)*

2.3.12 NV12 - Default UART

`NV_DEFAULT_UART_ID = 12`

This controls which UART will be pre-configured for Packet Serial Mode.

Normally, the UART-related settings would be specified by the **SNAPpy** scripts uploaded into the node. This default setting has been implemented to handle nodes that have no scripts loaded yet, or for scripts that do not explicitly set which UART will be used for Packet Serial Mode.

Warning

These defaults are overridden when needed!

Although you can request that one or both UARTs be disabled (via the Feature Bits), and you can request that there be no Packet Serial Mode UART (by setting the Default UART parameter to 255), both of these user requests will be ignored unless there is also a valid **SNAPpy** script loaded into the unit. If

the parameter is set to a value outside the range of UARTs on your module (other than 255), UART1 (or UART0 on modules with only one UART) will be the default.

If there is no **SNAPpy** script loaded, a fail-safe mechanism kicks in and forces an active Packet Serial port to be initialized on UART1 (or UART0, if so specified in this parameter), regardless of the other configuration settings. This was done to help prevent users from "locking themselves out."

If there is a **SNAPpy** script loaded, then the assumption is that the script will take care of any configuration overrides needed, and the Feature Bits and Default UART setting will be honored.

Factory Default Value = `1` **on platforms with two UARTs;** `0` **on platforms with only one UART**

2.3.13 Serial Data Forwarding

The next three NV parameters (13-15) affect the "forwarding" of buffered-up serial data (data that has been received over one of the serial ports).

The *NV13 - Buffering Timeout* and *NV15 - Inter-character Timeout* are (as you might expect) time-related. They affect time-driven triggers that can cause serial data to be "pushed" to other parts of the system. In contrast, *NV14 - Buffering Threshold* is completely timing-independent and is driven solely by the quantity of data that has been received. All three of these parameters can be tuned to control when data that a **SNAP** node receives over a serial connection gets forwarded to other nodes. To control where the data gets forwarded to, refer to the `crossConnect()` and `uniConnect()` functions. For more information, refer the to the **SNAP** Users Guide.

2.3.14 NV13 - Buffering Timeout

`NV_UART_DM_TIMEOUT_ID = 13`

This lets you tune the overall serial receive data timeout. This value is in milliseconds and defaults to 5. This value controls the typical maximum amount of time between an initial character being received over the serial port and a packet of buffered serial data being enqueued for processing. Regardless of the number of characters buffered or the rate at which they are being buffered, each time this timeout passes, any buffered data will be queued.

Note that other factors can also trigger the queuing of the buffered serial data. In particular, see *NV14 - Buffering Threshold* and *NV15 - Inter-character Timeout*.

The larger this value is, the more buffering will take place. In transparent mode, every packet has 12-15 bytes of overhead, so sending more serial characters per packet is more efficient. Also, when using multicast transparent mode, keeping the characters together (in the same packet) improves overall reliability. The trade-off is that the larger this value is, the greater the maximum latency can be through the overall system, especially at lower baud rates.

Setting this value to zero means your system will never trigger packet processing based on this timeout. Packet processing would then only occur based on the limits set by *NV14 - Buffering Threshold* and *NV15 - Inter-character Timeout*.

If you enable both the Buffering Timeout and *NV15 - Inter-character Timeout* and your node transmits data as a result of the Buffering Timeout being reached, it will send any available data again on the next buffering timeout period, even if an Inter-Character Timeout period triggers a send in the meantime. In other words, when the Inter-Character Timeout triggers, it does not reset the clock for the Buffering Timeout.

Factory Default Value = `5`

2.3.15 NV14 - Buffering Threshold

`NV_UART_DM_THRESHOLD_ID = 14`

This value indicates the total packet size threshold used when sending packets of data. The size defaults to 75 bytes. If no timeout limit is reached first, this parameter will cause buffered data to be enqueued when there is sufficient data to cause the packet, including header, to be at least this many bytes long. At higher serial rates, this size can be overshot between **SNAP** checks of the packet size. There is no guarantee that packets will necessarily be precisely this size.

Each packet of data sent includes a header, which comprises 12 bytes for multicast packets and 15 bytes for unicast packets. So the actual amount of serial data sent in each packet will be reduced by either 12 or 15 bytes, depending on whether the data is sent by multicast or unicast. Additionally, if the feature bit in *NV11 - Feature Bits* indicates that **SNAP** should be using its second CRC to prevent data corruption, the data payload will be reduced by an additional two bytes. If you want to send N bytes of data per packet, this parameter should be set to N + 12 for multicasting or N + 15 for unicasting.

The maximum **SNAP** packet size is 123 bytes, if the packet-level CRC isn't enabled using the **Packet CRC Bit** in *NV11 - Feature Bits*. If the **Packet CRC Bit** is set, the effective maximum length is 121 bytes. If you set this parameter to a value greater than the maximum (up to 255), the system will simply substitute the maximum. If you set this parameter less than or equal to the packet header size, **SNAP** will construct packets with a complete header and one byte of data. If you set this parameter to a value higher than 255, the parameter will be reset to the default value of 75.

Like *NV13 - Buffering Timeout* and *NV15 - Inter-character Timeout*, larger values can result in larger (more efficient) packets, at the expense of greater latency. Also, at higher baud rates, setting this value too high can result in dropped characters if the packet buffer gets over-filled between **SNAP** checks.

Factory Default Value = `75`

2.3.16 NV15 - Inter-character Timeout

`NV_UART_DM_INTERCHAR_ID = 15`

This lets you tune the inter-character serial receive data timeout. This value is in milliseconds and defaults to 0 (in other words, disabled).

This timeout is similar to *NV13 - Buffering Timeout*, but this one refers to the time between individual characters. Essentially, this timeout restarts with every received character, while the Buffering Timeout always runs to completion (as long as the *NV14 - Buffering Threshold* value is not exceeded). Larger inter-character timeouts can give better multicast transparent mode reliability, at the expense of greater latency.

Note that either *NV13 - Buffering Timeout* or *NV15 - Inter-character Timeout* (if enabled) can trigger the transmission of the buffered data before the *NV14 - Buffering Threshold* is reached. Conversely, if the timeouts are high (or disabled), to the extent that enough data is buffered to reach the Buffering Threshold before the timeouts are reached, that threshold will trigger the transmission of the buffered data before either of the timeouts are reached.

Factory Default Value = `0`

2.3.17 NV16 - Carrier Sense

`NV_CARRIER_SENSE_ID = 16`

This instructs the radio to "listen before you transmit."

Setting this value to True will cause the node to perform a Clear Channel Assessment. Basically, this means that the node will briefly listen before transmitting anything and will postpone sending the packet if some other node is already talking. This results in fewer collisions (which means more packets make it through), but the "listening" step adds a delay to the time it takes to send each packet. In an especially noisy environment, this setting could substantially delay or prevent your packet's transmission, even if the radio noise comes from an RF source other than other **SNAP** nodes.

Carrier Sense applies to all packets transmitted over the air.

If the probability of collisions is low in your network(you don't have much traffic), and you need the maximum throughput possible, then leave this value at its default setting of False. If the probability of collisions is high in your network (you have a lot of nodes talking a lot of the time), then you can try setting this parameter to True and see if it helps your particular application.

Factory Default Value = `False`

See also

- *NV33 - Noise Floor*

2.3.18 NV17 - Collision Detect

`NV_COLLISION_DETECT_ID = 17`

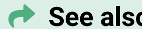
This instructs the radio to "listen after you transmit."

Setting this value to True will cause the node to perform a Clear Channel Assessment after sending a multicast packet, in an effort to determine whether some other node has "stepped on" its transmission. This will catch some (but not all) collisions. If the node detects that some other node was transmitting at the same time, or if there was a sufficiently high noise floor from another RF source, then the node will resend the multicast packet. This results in more multicast packets making it through, but there is a throughput penalty.

Collision Detect applies only to multicast (and directed multicast) packets. For unicast packets, **SNAP** relies on the acknowledgements and retries to account for noisy environments.

The same criteria given for *NV16 - Carrier Sense* apply to this one. You can try setting this parameter to True and see if it helps your application. If not, set it back to False.

Factory Default Value = `False`



See also

- *NV33 - Noise Floor*

2.3.19 NV18 - Collision Avoidance

`NV_COLLISION_AVOIDANCE_ID = 18`

This lets you control use of "random jitter" to try and reduce collisions. This setting defaults to True. The **SNAP** protocol uses a "random jitter" technique to reduce the number of collisions.

Before responding to a multicast packet, **SNAP** does a small random delay. This random delay, either 0, 4, 8, 12 or 16 milliseconds by default, reduces the number of collisions but increases packet latency. You can tune the delay characteristics using *NV91 - CSMA Timeslot Settings*.

If you set this parameter to False, then this initial delay will not be used. This reduces latency (some extremely time critical applications need this option) but increases the chances of an over-the-air collision. You should only change this parameter from its default setting of True if there is something else about your application that reduces the chances of collision. For example, some applications operate in a "command/response" fashion, where only one node at a time will be trying to respond anyway.

This parameter does not affect response time for directed multicasts or for unicasts.

Factory Default Value = `True`

2.3.20 NV19 - Radio Unicast Retries

`NV_SNAP_MAX_RETRIES_ID = 19`

This lets you control the number of unicast transmit attempts.

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 re-evaluate your network setup to determine if it would be better to either add more nodes to the mesh to forward requests or reduce the number of nodes broadcasting to cut down on packet collisions.

Factory Default Value = `8`

2.3.21 NV20 - Mesh Maximum Timeout

`NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID = 20`

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

Discovered mesh routes timeout after a configurable period of inactivity (see *NV23 - Mesh Used 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.

Factory Default Value = `0xEA60`**, which is one minute**

2.3.22 NV21 - Mesh Minimum Timeout

`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.

Factory Default Value = `1000`**, which is one second**

2.3.23 NV22 - Mesh New Timeout

`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.

Factory Default Value = `5000`**, which is five seconds**

2.3.24 NV23 - Mesh Used Timeout

`NV_MESH_ROUTE_USED_TIMEOUT_ID = 23`

This is how many additional milliseconds of "life" a route gets whenever it is used.

Every time a known route gets used, its timeout gets reset to this parameter. This prevents active routes from timing out as often, but it allows inactive routes to go away sooner.

Factory Default Value = 5000**, which is five seconds**

Note

- *NV20 - Mesh Maximum Timeout* takes precedence over *NV23 - Mesh Used Timeout*

2.3.25 NV24 - Mesh Delete Timeout

`NV_MESH_ROUTE_DELETE_TIMEOUT_ID = 24`

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

Factory Default Value = 10000**, which is ten seconds**

2.3.26 NV25 - Mesh RREQ Retries

`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.

Factory Default Value = 3

2.3.27 NV26 - Mesh RREQ Wait Time

`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 a second time.

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.

Factory Default Value = 500**, which is a half second**

2.3.28 NV27 - Mesh Initial Hop Limit

`NV_MESH_INITIAL_HOPLIMIT_ID = 27`

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 0, **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 must time out before route requests happen.

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

Also, although **Portal** is "one hop farther away" than all other **SNAP** nodes on your network (they are on the other side of a "bridge" node), the **SNAP** code knows this and will automatically give a "bonus hop" to this parameter's value when using it to find nodes with addresses in the reserved **Portal** address range of 00.00.01 - 00.00.0F. So, you can leave this parameter at its default setting of 1 (one hop) even if you use **Portal**.

Factory Default Value = `1`

2.3.29 NV28 - Mesh Maximum Hop Limit

`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.

Factory Default Value = `5`

2.3.30 NV29 - Mesh Sequence Number

`NV_MESH_SEQUENCE_NUMBER_ID = 29`

Reserved for future use.

2.3.31 NV30 - Mesh Override

`NV_MESH_OVERRIDE_ID = 30`

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 (subject to the mesh's willingness to be used).

This feature was added to better support 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 (as they broadcast fewer route requests and packets destined for other nodes) at the expense of reducing the "redundancy" in the overall mesh network.

Factory Default Value = 0

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**.

2.3.32 NV31 - Mesh LQ Threshold

`NV_MESH_PENALTY_LQ_THRESHOLD_ID = 31`

This allows for penalizing hops with poor link quality when establishing a mesh route. Hops that have a link quality worse than (i.e. a higher value than) the specified threshold will be counted as two hops instead of one. This allows the nodes to choose (for example) a three-hop route with good link quality over a two-hop route with poor link quality. The default threshold setting of 127 is the highest valid value, so that no "one hop penalty" will ever be applied.

Factory Default Value = 127

See also

- *NV27 - Mesh Initial Hop Limit*
- *NV32 - Mesh Rejection LQ Threshold*
- *NV39 - Radio LQ Threshold*

2.3.33 NV32 - Mesh Rejection LQ Threshold

`NV_MESH_REJECTION_LQ_THRESHOLD_ID = 32`

This allows for rejecting hops with poor link quality when establishing a mesh route. Hops that have a link quality worse than (i.e. a higher value than) the specified threshold will be rejected as the node performs route requests. The default threshold setting of 127 is the highest valid value, so that all routes will be considered for mesh routing.

Factory Default Value = 127

See also

- *NV27 - Mesh Initial Hop Limit*
- *NV31 - Mesh LQ Threshold*
- *NV39 - Radio LQ Threshold*

2.3.34 NV33 - Noise Floor

`NV_CS_CD_LEVEL_ID = 33`

The Carrier Sense and Collision Detect features work by checking the current ambient signal level before broadcasting (for Carrier Sense) and immediately after broadcasting (for Collision Detect), to determine whether some other node is broadcasting. In an environment with a lot of background noise, the noise floor can trigger false positives for these features, preventing the node from broadcasting or causing it to endlessly rebroadcast packets. On platforms that do not allow pokes (or radioPokes) to adjust the noise floor level, *NV33 - Noise Floor* can be used to define the signal strength that must be encountered to trigger the Carrier Sense and Collision Detect features. The parameter is in negative dBm, with a range from 0 to 127. If this parameter is not discussed in the section relating to your platform, refer to your platform's processor data sheet to determine the pokes (or radioPokes) appropriate to adjust the noise floor level.

Factory Default Value = Platform Specific or `None` if not used by the platform

See also

- Refer to platform-specific

2.3.35 NV34-38 – Reserved

Reserved for Synapse use.

2.3.36 NV39 - Radio LQ Threshold

`NV_SNAP_LQ_THRESHOLD_ID = 39`

This allows for ignoring packets with poor link quality. Link quality values range from a theoretical 0 (perfect signal, 0 attenuation) to a theoretical 127 (127 dBm "down").

If you lower this parameter from its default value of 127, you are in effect defining an "acceptance criteria" on all received packets. If a packet comes in with a link quality worse (higher) than the specified threshold, then the packet will be completely ignored. This gives you the option to ignore other nodes that are "on the edge" of radio range. The idea is that you want other (closer) nodes to take care of communicating to that node.

Warning

If you set this parameter too low, your node may not accept any packets.

Factory Default Value = `127`

2.3.37 NV40 - SNAPpy CRC

`NV_SNAPPY_CRC_ID = 40`

This is the 16-bit Cyclic Redundancy Check (CRC) of the currently loaded **SNAPpy** script. Most users will not need to write to this NV parameter. If you do change it from its automatically calculated value, you will make the **SNAP** node think its copy of the **SNAPpy** script is invalid, and it will not use it.

Not Modified on Factory Default

2.3.38 NV41 - Platform

`NV_SYS_PLATFORM_ID = 41`

This parameter makes it easier to write scripts that work on more than one type of **SNAP** node. Set this string parameter to some label that identifies your hardware platform.

New RF100 **SNAP** engines from Synapse will come with "RF100" in this parameter. Older RF100 engines may have had "RFEngine" here. If you are working with **SNAP**-compatible radios or engines from another source, the parameter might not be loaded with any meaningful value. Furthermore, like most other NV parameters, the value can be changed. To make use of this field, it is the responsibility of the user to ensure that the value in the parameter is meaningful and consistent across your collection of nodes.

To take advantage of the Platform value in your script, you must include the following line: 11 from synapse.snapsys import * When a script is loaded into a node, the script is compiled for the node. At compile time the platform variable is loaded with the contents of *NV41 - Platform*, which you can use to control which other **SNAPpy** modules get imported or what other code will be compiled. Because the variable is available at compile time (rather than only at run time), the compiler can optimize its code generation for the platform you are using, decreasing the code size and increasing the amount of space available for more complex scripts. The pinWakeup.py script, itself imported by the New-PinWakeupTest.py script, provides an example of this. See the "Cross-Platform Coding and Easy Pin Numbering" section in the SNAP Users Guide for examples of how to make use of the platform variable. If you do not import the synapse.snapsys module, the platform variable will not be defined.

Not Modified on Factory Default

2.3.39 NV42-49 – Reserved

Reserved for Synapse use.

2.3.40 NV50 - Enable Encryption

`NV_CRYPTO_TYPE = 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:

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

If you set this parameter to a value that indicates encryption should be used, but either an invalid encryption key is specified (in *NV51 - Encryption Key*) 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 respectively 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 or otherwise unknown encryption key, you will have to reset the factory parameters on the node to reestablish contact with it.

Even with a valid encryption key, encryption is not enabled until the node is rebooted. See the Encryption section in the **SNAP** Users Guide for more details.

Factory Default Value = `0`

2.3.41 NV51 - Encryption Key

`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 None. 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. You can specify your key as a series of keyboard-accessible characters, such as `My!Password? 123` (note that there is a space between the question mark and the "1"), as a series of escaped hexadecimal character representations , such as `\xD5\xAA\x96\x84\x94\x66\x97\x88\xF0\xAD\x10\x12\x91\x07\x86\xBA` (note that none of these characters is directly representable as a standard ASCII character), as a string containing "escaped" characters using the backslash, such as `\'\"\\\n\r\t\b\f\f\b\t\r\n\\\"\'`, or as any combination of the above.

When using escaped characters, it is possible that **Portal** will display them differently from how you entered them. For example, `\0` is shown as `\x00`, `\v` is shown as `\x0b`, `\"` will display as `"`, and `\'` will display as `'` if there is not also a `"` in your encryption key. Also, any characters you specify as escaped hexadecimal characters that fall into the range of "printable" ASCII characters (or simple escaped characters, such as `\t`) will be shown as those characters rather than the escaped hexadecimal value.

When escaping hexadecimal characters, the input is not case-sensitive. If you use a single backslash before a character that does not represent an escapable character, **Portal** will accept the two characters as two separate characters rather than one escaped character. Thus, `\h` would be two characters and would be rendered on the screen as `\h`, as **Portal** adds the backslash to escape the backslash you entered.

Changes to this parameter (as with most NV parameters) have no effect until the node is rebooted. However, beginning with release 2.6, you can specify -51 (negative 51) as the NV parameter in a *`saveNvParam()`* call, and your encryption key will be saved (in *NV51 - Encryption Key*) and will be applied immediately without requiring a reboot. (Calling loadNvParam(-51) remains invalid.) Setting this parameter to an invalid encryption key using -51 as the NV parameter will disable encryption.

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, communications will not be encrypted unless the node is loaded with a **SNAP** firmware image that

supports AES-128 encryption. Firmware images supporting AES-128 encryption will have "AES" in their filenames.

Refer to *getInfo()* for how to determine whether your firmware supports AES encryption.

Factory Default Value = `None`**, which provides no encryption**

2.3.42 NV52 - Lockdown

`NV_LOCKDOWN_FLAGS_ID = 52`

If this parameter is 0 (or never set at all), access to the node is unrestricted, and you can freely upload new scripts. If you set this parameter to 1 and the node is rebooted, then the system enters a "lockdown" mode where over-the-air script erasure or upload is not allowed.

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 *NV52 - Lockdown* over the air. (In other words, you cannot bypass the lockdown by remotely turning it off.)

Even in this mode, you can still perform all operations (including script upload or erasure) over the local Packet Serial link (assuming one is available). The lockdown only applies to over-the-air access. If you have disabled your UARTs and set this parameter, you will have to make a serial connection and use **Portal** to reset your factory parameters to regain control of your node.

Factory Default Value = `0`

2.3.43 NV53-62 – Reserved

Reserved for Synapse use.

2.3.44 NV63 - Alternate Radio Trim

`NV_ALT_RADIO_TRIM_ID = 63`

Usage is platform specific.

Not Modified on Factory Default

See also

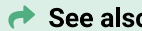
- Refer to platform-specific

2.3.45 NV64 - Vendor-Specific Settings

`NV_VENDOR_SETTINGS_ID = 64`

Similar in concept to *NV11 - Feature Bits*, this field is reserved for non-standard settings.

Not Modified on Factory Default



See also

- Refer to platform-specific

2.3.46 NV65 - Clock Regulator

`NV_CLOCK_REGULATOR_ID = 65`

In platforms that have sleep modes that do not use a crystal, this parameter allows you to adjust the regulation of the internal timer that controls sleep durations. The parameter does not apply to all platforms. See the platform-specific section for your platform to determine how to best adjust this value, if necessary. This value has no effect on sleep timings that are crystal-controlled.

See also

- Refer to platform-specific

2.3.47 NV66 - Radio Calibration Data

`NV_RADIO_CALIBRATION_ID = 66`

In platforms that require extra calibration data for proper radio operation, this parameter is used to store this calibration data. The parameter does not apply to all platforms. See the platform-specific section for your platform to determine how to best adjust this value, if necessary.

See also

- Refer to platform-specific

2.3.48 NV67-69 – Reserved

Reserved for Synapse use.

2.3.49 NV70 - Transmit Power Limit

`NV_TX_POWER_LIMIT_ID = 70`

The Transmit Power Limit is a string that specifies, channel by channel, the maximum power level that can be transmitted on each channel. The units for the setting match those for the *txPwr()* function, ranging from 0 through 17 (with 17 being the highest power). They represent a cap, or governor, limiting how high the output can be on the specified channel, possibly reducing the specified power if *txPwr()* is set higher than the channel setting specified here.

The value in the parameter is a string 16 bytes long, where the first byte represents the maximum power on channel 0, the second byte represents the maximum power on channel 1, and the 16th byte represents the maximum power on channel 15. For example, if you wanted to crank up the power to the maximum possible on all channels, you would use:

```
saveNvParam(70,'\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11')
```

This parameter is only implemented on MC1321x-based hardware. It does not override any system-set maximum power levels specified for government regulatory acceptance.

2.3.50 NV71-77 – Reserved

Reserved for Synapse use.

2.3.51 NV78 - Multicast Serial Forwarded Groups

`NV_GROUP_SERIAL_FORWARDING_MASK_ID = 78`

This is a 16-bit field controlling which multicast groups will be re-transmitted (forwarded) by the node over its serial connection. 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," while 0x0003 indicates that messages will be forwarded to groups 1 and 2.

If this field is set to its default value of None, SNAP will use *NV6 - Multicast Forward Groups* to determine how packets will be re-transmitted over both radio and serial interfaces. If this field is set to any integer value, the multicast group(s) represented by the integer bitmask will have associated packets forwarded over the serial interface, and *NV6 - Multicast Forward Groups* will apply only toward multicast packets forwarded over the node's radio interface.

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

Please note that, apart from the dependency when this parameter is set to None, *NV5 - Multicast Process Groups*, *NV6 - Multicast Forward Groups*, and *NV78 - Multicast Serial Forwarded Groups* 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 serially forward any combination of the 16 available groups, including none of them (by setting the field to zero, rather than None).

Factory Default Value = `None`

Note

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

2.3.52 NV79 – Reserved

Reserved for Synapse use.

2.3.53 NV80 - Default UART0 Rate

`NV_UART0_BAUDRATE_ID = 80`

New in version SNAP 2.6.

This parameter specifies the serial connection speed that will be applied to UART 0 for packet serial communications when the node boots. The valid values match those you would specify for a connection speed using the *initUart()* command. The default serial rate is 38,400 symbols per second.

Factory Default Value = `38400`

2.3.54 NV81 - Default UART1 Rate

`NV_UART1_BAUDRATE_ID = 81`

New in version SNAP 2.6.

This parameter specifies the serial connection speed that will be applied to UART 1 for packet serial communications when the node boots. The valid values match those you would specify for a connection speed using the *initUart()* command. The default serial rate is 38,400 symbols per second.

Factory Default Value = `38400`

2.3.55 NV82-89 – Reserved

Reserved for Synapse use.

2.3.56 NV90 - Default Radio Rate

`NV_RADIO_RATE_ID = 90`

New in version SNAP 2.6.

This parameter specifies which radio rate the node will use by default on reboot, in the absence of being set by the script. The valid values match those you would specify for the radio rate using the *setRadioRate()* command.

Factory Default Value = `0`**, which corresponds to 250 Kbps for 2.4 GHz devices**

2.3.57 NV91 - CSMA Timeslot Settings

`NV_CSMA_TIMESLOT_SETTINGS_ID = 91`

New in version SNAP 2.6.

The Carrier Sense Multiple Access Timeslot Settings parameter lets you fine-tune the nature of the delay that will be applied to responses to multicasts, based on the status of *NV18 - Collision Avoidance*.

This new feature, introduced in **SNAP** version 2.6, takes an integer as the setting. The high byte is the number of time slots from which receiving nodes can select for their responses. The low byte sets the width of each timeslot in milliseconds. Setting either the high byte or low byte too large can cause communication problems, as messages can time out before they can be sent. But making small adjustments to the parameters can be effective if you expect especially large return messages or if you have a large number of responding nodes in your network.

Factory Default Value = `0x0404`

2.3.58 NV92-127 – Reserved

Reserved for Synapse use.

2.3.59 NV128-254

These are user-defined NV parameters, and can be used for whatever purpose you choose. Factory defaulting a node's NV parameters resets all of these parameters to None.

2.3.60 NV255 – Reserved

Reserved for Synapse use.

2.4 SNAP Hooks

SNAP hooks help provide a method for *Event-Driven Programming* in **SNAPpy**. There are a number of events in the system that can invoke a **SNAPpy** function. Use the *`setHook()`* function decorator to associate your **SNAPpy** function with one the following events:

2.4.1 HOOK_STARTUP

`HOOK_STARTUP`

Called on device bootup.

Example

```
@setHook(HOOK_STARTUP)
def onBoot():
    pass
```

2.4.2 HOOK_GPIN

`HOOK_GPIN`

Called on transition of a monitored hardware pin.

Parameters

- `pinNum` (*int*) – The pin number of the pin that has transitioned.
- `isSet` (*bool*) – A Boolean value indicating whether the pin is set.

Example

```
@setHook(HOOK_GPIN)
def pinChg(pinNum, isSet):
    pass
```

See also

- *monitorPin()*

2.4.3 HOOK_1MS

`HOOK_1MS`

Called every millisecond.

Parameters

`tick` (*int*) – A rolling 16-bit integer incremented every millisecond, indicating the current count on the internal clock. The same counter is used for all four timing hooks.

Example

```
@setHook(HOOK_1MS)
def doEvery1ms(tick):
    pass
```

2.4.4 HOOK_10MS

HOOK_10MS

Called every 10 milliseconds.

Parameters

tick (*int*) – A rolling 16-bit integer incremented every millisecond, indicating the current count on the internal clock. The same counter is used for all four timing hooks.

Example

```
@setHook(HOOK_10MS)
def doEvery10ms(tick):
    pass
```

2.4.5 HOOK_100MS

HOOK_100MS

Called every 100 milliseconds.

Parameters

tick (*int*) – A rolling 16-bit integer incremented every millisecond, indicating the current count on the internal clock. The same counter is used for all four timing hooks.

Example

```
@setHook(HOOK_100MS)
def doEvery100ms(tick):
    pass
```

2.4.6 HOOK_1S

HOOK_1S

Called every second.

Parameters

tick (*int*) – A rolling 16-bit integer incremented every millisecond, indicating the current count on the internal clock. The same counter is used for all four timing hooks.

Example

```
@setHook(HOOK_1S)
def doEverySec(tick):
    pass
```

2.4.7 HOOK_STDIN

`HOOK_STDIN`

Called when "user input" data is received.

Parameters

`data` (*str*) – A data buffer containing one or more received characters.

Example

```
@setHook(HOOK_STDIN)
def getInput(data):
    pass
```

See also

- User Guide on *The Switchboard*

2.4.8 HOOK_STDOUT

`HOOK_STDOUT`

Called when "user output" data is sent.

Example

```
@setHook(HOOK_STDOUT)
def printed():
    pass
```

See also

- User Guide on *The Switchboard*

2.4.9 HOOK_RPC_SENT

`HOOK_RPC_SENT`

Called when the buffer for an outgoing RPC call is cleared.

Parameters

`bufRef` (*int*) – An integer reference to the packet that the RPC call attempted to send. This integer will correspond to the value returned from `getInfo(9)` when called immediately after an RPC call is made. The receipt of a value from *HOOK_RPC_SENT* does not necessarily indicate that the packet was sent and received successfully. It is an indication that **SNAP** has completed processing the packet.

Example

```
@setHook(HOOK_RPC_SENT)
def rpcDone(bufRef):
    pass
```

See also

- *getInfo()*

INDEX

B

built-in function
    setHook(), 100

C

call() (*in module snappy.BuiltIn*), 51
callback() (*in module snappy.BuiltIn*), 51
callout() (*in module snappy.BuiltIn*), 52
chr() (*in module snappy.BuiltIn*), 53
crossConnect() (*in module snappy.BuiltIn*), 54

D

dmCallout() (*in module snappy.BuiltIn*), 54
dmcastRpc() (*in module snappy.BuiltIn*), 55

E

eraseImage() (*in module snappy.BuiltIn*), 56
errno() (*in module snappy.BuiltIn*), 57

F

flowControl() (*in module snappy.BuiltIn*), 59

G

getChannel() (*in module snappy.BuiltIn*), 60
getEnergy() (*in module snappy.BuiltIn*), 60
getI2cResult() (*in module snappy.BuiltIn*), 61
getInfo() (*in module snappy.BuiltIn*), 61
getLq() (*in module snappy.BuiltIn*), 68
getMs() (*in module snappy.BuiltIn*), 68
getNetId() (*in module snappy.BuiltIn*), 69
getStat() (*in module snappy.BuiltIn*), 69

H

HOOK_100MS (*built-in variable*), 125
HOOK_10MS (*built-in variable*), 125
HOOK_1MS (*built-in variable*), 124
HOOK_1S (*built-in variable*), 125
HOOK_GPIN (*built-in variable*), 124
HOOK_RPC_SENT (*built-in variable*), 126
HOOK_STARTUP (*built-in variable*), 123
HOOK_STDIN (*built-in variable*), 126
HOOK_STDOUT (*built-in variable*), 126

I

i2cInit() (*in module snappy.BuiltIn*), 70
i2cRead() (*in module snappy.BuiltIn*), 71
i2cWrite() (*in module snappy.BuiltIn*), 71
imageName() (*in module snappy.BuiltIn*), 72
initUart() (*in module snappy.BuiltIn*), 73
initVm() (*in module snappy.BuiltIn*), 73
int() (*in module snappy.BuiltIn*), 74

L

len() (*in module snappy.BuiltIn*), 74
loadNvParam() (*in module snappy.BuiltIn*), 75
localAddr() (*in module snappy.BuiltIn*), 75

M

mcastRpc() (*in module snappy.BuiltIn*), 75
mcastSerial() (*in module snappy.BuiltIn*), 76
monitorPin() (*in module snappy.BuiltIn*), 77

N

NV_ALT_RADIO_TRIM_ID (*in module snappy.nvparams*), 119
NV_CARRIER_SENSE_ID (*in module snappy.nvparams*), 110
NV_CHANNEL_ID (*in module snappy.nvparams*), 101
NV_CLOCK_REGULATOR_ID (*in module snappy.nvparams*), 120
NV_COLLISION_AVOIDANCE_ID (*in module snappy.nvparams*), 111
NV_COLLISION_DETECT_ID (*in module snappy.nvparams*), 110
NV_CRYPTO_KEY (*in module snappy.nvparams*), 118
NV_CRYPTO_TYPE (*in module snappy.nvparams*), 117
NV_CS_CD_LEVEL_ID (*in module snappy.nvparams*), 116
NV_CSMA_TIMESLOT_SETTINGS_ID (*in module snappy.nvparams*), 123
NV_DEFAULT_UART_ID (*in module snappy.nvparams*), 107

NV_DEVICE_NAME_ID (*in module snappy.nvparams*), 103
NV_DEVICE_TYPE_ID (*in module snappy.nvparams*), 104
NV_FEATURE_BITS_ID (*in module snappy.nvparams*), 104
NV_GROUP_FORWARDING_MASK_ID (*in module snappy.nvparams*), 102
NV_GROUP_INTEREST_MASK_ID (*in module snappy.nvparams*), 102
NV_GROUP_SERIAL_FORWARDING_MASK_ID (*in module snappy.nvparams*), 121
NV_LOCKDOWN_FLAGS_ID (*in module snappy.nvparams*), 119
NV_MAC_ADDR_ID (*in module snappy.nvparams*), 100
NV_MESH_INITIAL_HOPLIMIT_ID (*in module snappy.nvparams*), 113
NV_MESH_MAX_HOPLIMIT_ID (*in module snappy.nvparams*), 114
NV_MESH_OVERRIDE_ID (*in module snappy.nvparams*), 114
NV_MESH_PENALTY_LQ_THRESHOLD_ID (*in module snappy.nvparams*), 115
NV_MESH_REJECTION_LQ_THRESHOLD_ID (*in module snappy.nvparams*), 115
NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID (*in module snappy.nvparams*), 112
NV_MESH_ROUTE_AGE_MIN_TIMEOUT_ID (*in module snappy.nvparams*), 112
NV_MESH_ROUTE_DELETE_TIMEOUT_ID (*in module snappy.nvparams*), 113
NV_MESH_ROUTE_NEW_TIMEOUT_ID (*in module snappy.nvparams*), 112
NV_MESH_ROUTE_USED_TIMEOUT_ID (*in module snappy.nvparams*), 112
NV_MESH_RREQ_TRIES_ID (*in module snappy.nvparams*), 113
NV_MESH_RREQ_WAIT_TIME_ID (*in module snappy.nvparams*), 113
NV_MESH_SEQUENCE_NUMBER_ID (*in module snappy.nvparams*), 114
NV_NETWORK_ID (*in module snappy.nvparams*), 100
NV_RADIO_CALIBRATION_ID (*in module snappy.nvparams*), 120
NV_RADIO_RATE_ID (*in module snappy.nvparams*), 122
NV_SNAP_LQ_THRESHOLD_ID (*in module snappy.nvparams*), 116
NV_SNAP_MAX_RETRIES_ID (*in module snappy.nvparams*), 111
NV_SNAPPY_CRC_ID (*in module snappy.nvparams*), 117
NV_SYS_PLATFORM_ID (*in module snappy.nvparams*), 117
NV_TX_POWER_LIMIT_ID (*in module snappy.nvparams*), 121
NV_UART0_BAUDRATE_ID (*in module snappy.nvparams*), 122
NV_UART1_BAUDRATE_ID (*in module snappy.nvparams*), 122
NV_UART_DM_INTERCHAR_ID (*in module snappy.nvparams*), 109
NV_UART_DM_THRESHOLD_ID (*in module snappy.nvparams*), 109
NV_UART_DM_TIMEOUT_ID (*in module snappy.nvparams*), 108
NV_VENDOR_SETTINGS_ID (*in module snappy.nvparams*), 120

O

ord() (*in module snappy.BuiltIn*), 77

P

peek() (*in module snappy.BuiltIn*), 78
poke() (*in module snappy.BuiltIn*), 78
pulsePin() (*in module snappy.BuiltIn*), 79

R

random() (*in module snappy.BuiltIn*), 80
readAdc() (*in module snappy.BuiltIn*), 80
readPin() (*in module snappy.BuiltIn*), 80
reboot() (*in module snappy.BuiltIn*), 81
resetVm() (*in module snappy.BuiltIn*), 81
rpc() (*in module snappy.BuiltIn*), 82
rpcSourceAddr() (*in module snappy.BuiltIn*), 82
rx() (*in module snappy.BuiltIn*), 83

S

saveNvParam() (*in module snappy.BuiltIn*), 83
scanEnergy() (*in module snappy.BuiltIn*), 85
setChannel() (*in module snappy.BuiltIn*), 85
setHook()
    built-in function, 100
setNetId() (*in module snappy.BuiltIn*), 86
setPinDir() (*in module snappy.BuiltIn*), 87
setPinPullup() (*in module snappy.BuiltIn*), 87
setPinSlew() (*in module snappy.BuiltIn*), 88
setRadioRate() (*in module snappy.BuiltIn*), 88
setRate() (*in module snappy.BuiltIn*), 89
sleep() (*in module snappy.BuiltIn*), 89
spiInit() (*in module snappy.BuiltIn*), 90
spiRead() (*in module snappy.BuiltIn*), 91
spiWrite() (*in module snappy.BuiltIn*), 91
spiXfer() (*in module snappy.BuiltIn*), 92
stdinMode() (*in module snappy.BuiltIn*), 93
str() (*in module snappy.BuiltIn*), 93

T

`txPwr()` (*in module snappy.BuiltIn*), 94
`type()` (*in module snappy.BuiltIn*), 94

U

`ucastSerial()` (*in module snappy.BuiltIn*), 95
`uniConnect()` (*in module snappy.BuiltIn*), 95

V

`vmStat()` (*in module snappy.BuiltIn*), 96

W

`writePin()` (*in module snappy.BuiltIn*), 98

X

`xrange()` (*in module snappy.BuiltIn*), 99