USER MANUAL

SNAP® Network Operating System

User Manual for Version 2.6

Doc# 116-061520-014-B000

6723 Odyssey Drive // Huntsville, AL 35806 // (877) 982-7888 // Synapse-Wireless.com

Disclaimers

Information contained in this Manual is provided in connection with Synapse products and services and is intended solely to assist its customers. Synapse reserves the right to make changes at any time and without notice. Synapse assumes no liability whatsoever for the contents of this Manual or the redistribution as permitted by the foregoing Limited License. The terms and conditions governing the sale or use of Synapse products is expressly contained in the Synapse's Terms and Condition for the sale of those respective products.

Synapse retains the right to make changes to any product specification at any time without notice or liability to prior users, contributors, or recipients of redistributed versions of this Manual. Errata should be checked on any product referenced.

Synapse and the Synapse logo are registered trademarks of Synapse. All other trademarks are the property of their owners. For further information on any Synapse product or service, contact us at:

Synapse Wireless, Inc.
6723 Odyssey Drive
Huntsville, Alabama 35806
256-852-7888
877-982-7888
256-924-7398 (fax)

www.synapse-wireless.com

License governing any code samples presented in this Manual

Table of Contents

1. Introduction

SNAP and SNAPpy

The Synapse SNAP product line provides an extremely powerful and flexible platform for developing and deploying embedded wireless applications.

The SNAP network operating system is the protocol spoken by all Synapse wireless nodes. The term SNAP has also evolved over time to refer generically to the entire product line. For example, we often speak of "SNAP Networks," "SNAP Nodes," and "SNAP Applications."

SNAP core software runs on each SNAP node. This core code handles wireless and serial communications, as well as implementing a Python virtual machine.

The subset of the Python programming language implemented by the core software is named SNAPpy. Scripts written in SNAPpy (also referred to as "Device Images", "SNAPpy images" or even "Snappy Images") can be uploaded into SNAP Nodes serially or over the air, and profoundly affect the node's capabilities and behavior.

Portal and SNAP Connect 3.x

Synapse Portal is a standalone software application that runs on a standard PC. Using a USB or RS232 interface it connects to any node in the SNAP wireless network, becoming a graphical user interface (GUI) for the entire network. Using Portal you can quickly and easily create, deploy, configure, and monitor SNAP-based network applications. Once connected, the Portal PC has its own unique network address and can participate in the SNAP network as a peer.

SNAP Connect 3.x is a Python **library** that allows your own Python applications to support the same "SNAP Connectivity" that Portal has. In addition to making USB or RS232 serial connections, SNAP Connect 3.x can also make TCP/IP connections to other running instances of SNAP Connect.

It is also possible for Portal to connect (via TCP/IP) to your SNAP network through a SNAP Connect application. This allows you to develop, configure, and deploy SNAP applications over the Internet.

The SNAP Wireless Sniffer

When you install Portal, you have the option of also installing a wireless "SNAP Sniffer" application. This program allows you to see SNAP messages that are broadcast over the air.

Navigating the SNAP Documentation

There are several main documents you need to be aware of:

Start with the "SNAP Primer"

The SNAP Primer introduces the SNAP products and ecosystem. If you are new to SNAP, be sure to read this document first.

Next look at an "Evaluation Kit Users Guide"

Each evaluation kit comes with its own Users Guide. For example, the EK2500 kit comes with the EK2500 Evaluation Kit Users Guide ("EK2500 Guide"), and the EK2100 kit comes with the EK2100 Evaluation Kit Users Guide ("EK2100 Guide").

Each of these guides walks you through the basics of unpacking your evaluation kit, setting up your wireless nodes, and installing Portal software on your PC. You may find it helpful to start with one of these manuals, even if you are not starting with an EK2500 or EK2100 kit. (Synapse SNAP nodes are also sold separately, as well as bundled into evaluation kits.)

About This Manual

This manual assumes you have read and understood the SNAP Primer and either the "EK2100 Users Guide" or the "EK2500 Users Guide." It assumes you have installed the Portal software, and are now familiar with the basics of discovering nodes, uploading SNAPpy scripts into them, and controlling and monitoring them from Portal.

The focus of this manual is general information about SNAP and SNAPpy. It covers topics like the SNAPpy language, and how to use it.

NOTE – This document is more "tutorial" in nature. If the information you are seeking is more "reference" in nature, for example a list of the built-in functions that are accessible from SNAPpy, or information about the different node configuration parameters that can be changed, then you should refer to the SNAP Reference Manual.

Other Important Documentation

Be sure to check out all of the SNAP documentation:

This document, the SNAP Users Guide, is only one of several. Be sure to also take a look at:

- The "SNAP Primer" (60037-01)
- The "Portal Reference Manual" (60024-01)
- The "SNAP Reference Manual" (600-0007)
- The "SNAP Hardware Technical Manual" (600-101.01)

Every switch, button, and jumper of every SNAP board is covered in this hardware reference document.

- The "End Device Quick Start Guide" (600-0001)
- The "SN171 Proto Board Quick Start Guide" (600-0011)

These two documents are subsets of the "SNAP Hardware Technical Manual" and come in handy because they focus on a single board type.

- The "SNAP Sniffer Users Guide" (600026-01)

Starting with Portal version 2.2.23, a "wireless sniffer" capability is included with Portal. If you follow the instructions in this standalone manual, you will be able to actually see the wireless exchanges that are taking place between your SNAP nodes.

- The "SNAP 2.2 Migration Guide" (600023-01)

There were enough changes between the 2.1 and 2.2 series of SNAP releases that we decided to provide an extra "transition" guide. This document also provides insights into the platform variable and the concept of GPIO pins to simplify cross-platform development. It may be worth a review, especially if you expect to be developing scripts for SNAP Engines based on more than one underlying architecture.

- The "SNAP Firmware Release Notes"
- The "Portal Release Notes"

Every Portal and SNAP Firmware release comes with a release notes document describing what has changed since the previous release.

All of these documents are in Portable Document Format (PDF) files for download from the Synapse website.

SNAP Documentation by Category

The table below may help you navigate the SNAP Documentation. It categorizes the existing documentation set across two dimensions.

Horizontally we categorize the various manuals as "Introductory" or "Reference". If you are new to SNAP, you should probably be starting with the "Introductory" documents first, and then move up to the "Reference" content.

Vertically we categorize the manuals along a spectrum between "Software" and "Hardware", with the "Kit" documentation considered to fall somewhere between the two extremes.

	Introductory Guides	Reference Guides
Software Guides	SNAP Primer SNAP Users Guide SNAP Sniffer Users Guide SNAP 2.2 Migration Guide	Portal Reference Manual SNAP Reference Manual SNAP Connect Python Package Manual Portal 2.4 Release Notes SNAP 2.4 Firmware Release Notes SNAP Connect 3.x Release Notes
Evaluation Kit Guides	EK2100 Evaluation Kit Users Guide EK2500 Evaluation Kit Users Guide DK-200 Evaluation Kit Users Guide	
Hardware Guides	SN171 Proto Board Quick Start Guide SN132 SNAP Stick Quick Start Guide End Device Quick Start Guide SNAP Connect E10 User Guide	SNAP Hardware Technical Manual

When the Manuals Are Not Enough

There is also a dedicated support forum at http://forums.synapse-wireless.com. In this forum, you can see questions and answers posted by other users, as well as post your own questions. The forum also has examples and Application Notes, waiting to be downloaded. Registering on the Synapse Wireless website to download Portal automatically registers you for the forum.

You can download the latest SNAP, Portal, and SNAP Connect software from the website. You can also download the latest documentation from the forum, including the EK2500 and EK2100 guides. (You might want to do this if you bought standalone modules instead of buying a kit.)

2. SNAP Overview

SNAP is a family of software technologies that together form an integrated, end-to-end solution for wireless monitoring and control. The latest version is 2.5, which this document covers.

Key features of SNAP

- All devices are peers – any device can be a bridge for Portal, do mesh routing, sleep, etc. There are no "coordinators" in SNAP.
- SNAP implements a full mesh topology. Any node can talk directly to any other node within radio range, and can talk indirectly to any node within the SNAP network.
- Communication among devices can be unicast (addressed to a specific destination, with acknowledged receipt, retrying as appropriate) or multicast (unacknowledged, only sent once, and potentially acted on by multiple recipients).
- Remote Procedure Call (RPC) among peers is the fundamental method of messaging.
- The PC based user interface (Portal) appears as a peer device on the SNAP network.
- By default, the SNAP operating system automatically forms a mesh network with other nodes immediately on receiving power. No further configuration is necessary. Multiple unrelated SNAP networks can exist within the same area through several configuration options outlined within this document.

RPC

Remote Procedure Calls are a hallmark of SNAP interoperability. With RPCs, any node can make a request to another node that it perform some task by running a function it knows. The calling node doesn't need to know anything about the task to be performed other than its name, and the task might or might not involve sending data back to the calling node. (Such a data transfer would happen via a separate RPC call, this from the second node back to the first.)

All SNAP devices implement a core set of built-in functions (procedures) to handle basic network configuration, system services, and device hardware control. Additional user-defined functions may be uploaded to devices as well. Functions are defined as SNAPpy scripts, in an embedded subset of the Python language called SNAPpy. This upload process can occur over directly connected serial interfaces, or over the air.

Once uploaded, both the built-in functions and the user-defined functions are callable locally (from within the user-defined script) or remotely by RPC from another node. These functions may themselves invoke local and remote functions.

For example, imagine an HVAC system where the compressor unit communicates with the thermostat using SNAP nodes. This could potentially allow a PC to become part of the network, to either query or control the system.

The PC could send an askCurrentTemp RPC request to the thermostat, which could respond with a tellTemp RPC back to the PC. The PC could send a setDesiredTemp RPC call to the thermostat, which (depending on how things are set and the ambient conditions) might then make RPC calls to the compressor unit to turn the unit on or to change its mode from heating to cooling, for example.

Each of these commands in a SNAP network is a separate, independent RPC call, acted on by the recipient of the call. In each case, the calling node does not need to know the implementation of the function at the called node. For example, the SNAPpy script in the PC node does not need to have an askCurrentTemp() function defined in order to be able to request that the thermostat node run its askCurrentTemp() function.

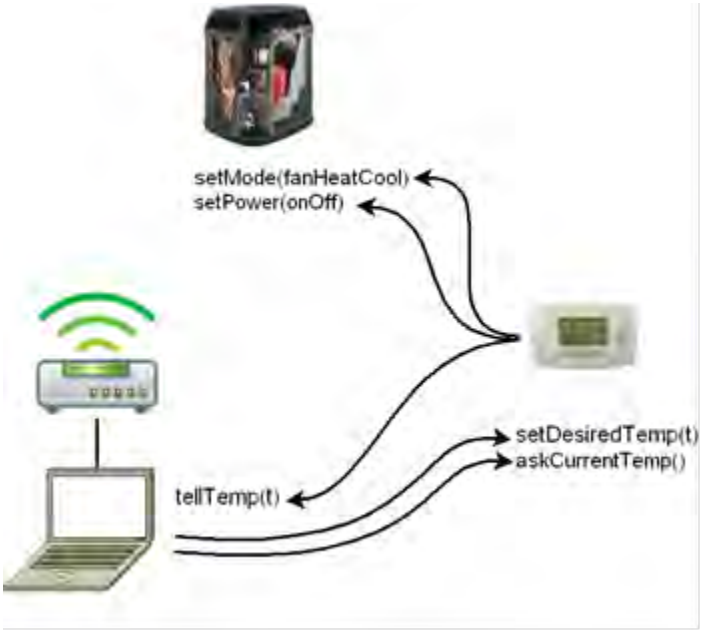


Figure 1 - Example HVAC System Showing RPC Call-flow (arrows)

RPCs are a fundamental part of SNAP's communication infrastructure, enough so that there is an entire section of this manual devoted to it, beginning on page 35.

SNAPpy Scripting

SNAPpy is a subset of the Python programming language, optimized for low-power embedded devices. A SNAPpy "script" is a collection of functions and data that is processed by Portal and uploaded to SNAP devices. All SNAP devices are capable of running SNAPpy – it is the native language of SNAP RPC calls.

SNAPpy Examples

On installation, Portal creates a folder under "My Documents" called "Portal\snappyImages".[1] Several sample script files are installed here by default. These scripts are plain text files that may be opened and edited with Portal's built-in editor. External text editors (such as Notepad) or even full-fledged Python Integrated Development Environments (IDEs) may also be used. Feel free to copy and modify the sample scripts (the installed copies are read-only), and create your own as you build custom network applications.

Be sure to make copies of the provided files rather than just changing the file attributes to be editable. Otherwise, if you ever update your version of Portal you will overwrite any changes you have made to the files.

Portal Scripting

Just as with the SNAP nodes, Portal can also be extended through scripting. By loading a Python script into Portal, you can add new functions that you (and the other SNAP nodes) can call.

Portal scripts are written in full Python. You are not limited to the embedded SNAPpy subset for Portal scripts, as you are for SNAPpy scripts that run on SNAP Engines. Python is a very powerful language, which finds use in a wide variety of application areas. Although the core of Python is not a large language, it is well beyond the scope of this document to cover it in any detail.

[1] The installation path shown is for Windows installations. File locations for Mac OS X and Ubuntu Linux installations will be appropriate for those file systems.

You won't have to search long to find an immense amount of information regarding Python on the Web. Besides your favorite search engine, a good place to start looking for further information is Python's home site:

http://python.org/

The Documentation page on Python's home site contains links to tutorials at various levels of programming experience, from beginner to expert.

As mentioned earlier, Portal acts as a peer in the SNAP network, and can send and receive RPC calls like any other node. Once you have loaded a Python script into Portal, any function available in the script can be invoked by any other node in your network. Thanks to this capability, any node in your network can send an e-mail or update a database in response to some monitored event, even if they do not have a direct connection to the appropriate servers.

3. SNAPpy – The Language

SNAPpy is basically a subset of Python, with a few extensions to better support embedded real-time programming. Here is a quick overview of the SNAPpy language.

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 be consistent. The indentation level is the thing that distinguishes blocks of code, determining what code is part of a function, or what 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"/"else"

```
if x == 1:
    print "Found number 1"
elif x == 2:
    print "Found number 2"
else:
    print "Did not find 1 or 2"
y = 3 if x == 1 else 4  # Ternary form is acceptable
```

Looping is supported via "while" and "for"[2]

```
x = 10
while x > 0:
    print x
    x = x - 1
# for will step through tuples or byte lists
myTuple = ("A", True, 3)
for element in myTuple:
    print element
# for will step through iterators returned by xrange
# the following prints "012" (note that SNAPpy does not insert spaces)
for number in xrange(3):
    print number,
# for will step through strings
for letter in myStringVariableOrConstant:
    if letter == "Z":
        print "I found a Z"
```

[2] SNAP added support for for loops in release 2.6.

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
```

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
i = [1, 1, 2, 3, 5, 8]  # Byte List
```

In the above example, invoking g() would be the same as directly calling startup(). You can use the `type(arg)` function (introduced in SNAP version 2.5) to determine the type of any variable in SNAPpy. See the SNAP Reference Manual for information on this built-in function. Iterators and byte lists were introduced in SNAP version 2.6.

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
```

You define new 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.

It is also important in your Portal and SNAP Connect related programming to make sure that any routines defined in Portal scripts (or SNAP Connect clients) accept the same number and type of parameters that the remote callers are providing. For example:

If in a Portal script you define a function like…

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

…but in your SNAPpy scripts you have RPC calls like…

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

…or…

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

…then you are going to see no output at all in Portal. Because the "signatures" do not match, Portal does not invoke the `displayStatus()` function at all.

You can change the calling SNAPpy script(s), or you can change the Portal script, but they must match.

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.

Variables at the top of your script (outside the scope of a function definition) 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
    # Note that these two statements are NOT errors if some other function
    # has previously initialized a value for global variable x before this
    # function runs. Globals declared in this way have the same availability
    # as globals explicitly initialized outside the scope of any function.
```

The usual comparators are supported

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

```
if 2 == 4:
    print "something is wrong!"
if 1 != 1:
    print "something is wrong!"
if 1 < 2:
    print "that's what I thought"
```

The usual math operators are supported

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication

/	Division
%	Modulo (remainder function)

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

SNAPpy does not support floating point, only integers.

SNAPpy integers are 16-bit signed values ranging from -32768 to 32767. If you add 1 to 32767, you will get -32768.

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

The usual Boolean functions are supported

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

```
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 = False or True  # Result is True
Result = False or False  # Result is False
```

Variables do have types, but they can change 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
```

Functions can change, too

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.

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"
```

These “docstrings” will appear as tool-tips in some portions of the Portal GUI. They are also considered a good practice in Python programming.

4. SNAPpy versus Python

Here are more details about SNAPpy, with emphasis on the differences between SNAPpy and Python.

Modules

SNAPpy supports import of user-defined as well as standard predefined Python source library modules.

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

Variables

Local and Global variables are supported. On RAM-constrained devices, SNAPpy images are typically limited to 64 system globals and 64 concurrent locals. Per-platform values are given in the SNAP Reference Manual.

Functions

Up to 255 "public" functions may be defined.[3] These are remotely callable using the SNAP RPC protocol, and include the SNAPpy built-in functions.

Non-public functions (prefixed with underscore) are limited only by the size of FLASH memory. These are not remotely callable, but can be called by other functions in the same script. (That is what it means to be non-public.)

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 _myFunction`.

It is not recommended that you use non-public functions for hooked events, as scripts with a large number of constants and/or functions (public and non-public) can cause the hooks to invoke the wrong script at run-time.[4]

Data Types

SNAPpy supports eight Python data types: None, integer, Boolean, string, function, tuple, byte list, and iterator.

None

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.

SNAPpy's `type()` function returns a 0 for variables of type None.

[3] SNAP Release 2.6 increased this limit to be constrained by the amount of flash space available. Testing has confirmed that more than 500 functions can be available on a node.

[4] The increase in the number of public functions in SNAP Release 2.6 eliminated this issue. In Release 2.6 you can safely hook a non-public function without concern about the incorrect function executing at run-time.

Integer

An integer in SNAPpy is a signed 16-bit integer, -32768 through 32767. 32767 + 1 = -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 !!!
```

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 inter 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 "or" (`|`) operators function as expected, as does the left-shift (`<<`) operator. Beware when right-shifting (`>>`) 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.

SNAPpy's `type()` function returns a 1 for variables of type Integer.

String

A static string (defined as a constant in your code and not modified by your code) has a maximum size of 255 bytes. However if 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, SNAPpy works from a collection of string buffers and may restrict the maximum dynamic string size to a smaller size.[5] The maximum length of the dynamic string and the number of string buffers available will be determined by the platform on which SNAP is running. See the platform-specific parameters in the SNAP Reference Manual for more details. (Note – built-in functions slice/concat/rpc enforce smaller limits on what they can do with strings.)

[5] SNAP release 2.6 introduced large string buffers, allowing dynamic strings to be as large as 255 bytes as well.

Strings are immutable in SNAPpy, as they are in Python; you cannot change characters within the string "in place." In order to modify a string, you must perform slicing operations on the string, creating a new string as you go. This means there must be sufficient string processing buffers available for your operation.

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:

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

In the above example, any string you passed to `runArbitrary(function)`, such as "random" or "getLq" or "reboot," would be invoked immediately, with the function's return value becoming the return value of the `runArbitrary` function. This passage of the *name of* a function is similar to, but different from, Python's ability to pass the function itself as a parameter to have the receiving variable be runnable.

SNAPpy strings can contain any of the 256 values storable in one byte for each character, including value `"\x00"`. (Strings are not null-terminated.)

Beginning in release 2.6, you can use `in` to determine whether a string contains a substring. You can also iterate through characters in a string with a `for` loop beginning in release 2.6.

SNAPpy's `type()` function returns a 2 for variables of type String.

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, 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 node, or by RPC (direct or

multicast) from another node, which establishes the ability to have one multicast call cause different nodes to run different functions, as configured.

SNAPpy's `type()` function returns a 3 for variables of type Function.

Boolean

A Boolean has a value of either True or False. Note that those are case-sensitive.

Comparisons of Booleans can be direct:

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

SNAPpy's `type()` function returns a 5 for variables of type Boolean.

Tuple

A tuple is an ordered read-only container of data elements, e.g., `myTuple = (None, True, 2, "Three", ("Four", "in", "this", "tuple"), [5, 10, 15, 20, 25])`.

Not only is the tuple immutable, but its contents are, too. You cannot change any of the bytes in a byte list contained within a tuple; it is treated as a tuple rather than as a list.

Elements in a tuple can be of any other type available in SNAPpy (including nested tuples) except for iterators. There are restrictions on printing of nested tuples. (See the details about printing later in this document.)

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 cannot pass a tuple as a parameter in an RPC call, though you can pass any element contained in a tuple, as long as it would otherwise be passable.

Beginning in release 2.6, you can use `in` to determine whether a tuple contains an element. You can also iterate through elements in a tuple with a `for` loop beginning in release 2.6.

SNAPpy's `type()` function returns a 6 for variables of type Tuple.

Iterator

Iterators were introduced in SNAP 2.6, and are generated using the `xrange()` function. 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 node:

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

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

Iterators defined in the global space in SNAPpy scripts do not function. You also cannot pass iterators as parameters in RPC calls to other nodes.

SNAPpy's `type()` function returns a 7 for variables of type Iterator.

Byte List

Byte lists were introduced in SNAP 2.6, and 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. Byte list elements can be changed in place, while SNAPpy strings (just as with Python strings) are immutable.

```
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 doesn't require that extra buffers be available for processing the slicing.[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` keyword to determine whether a number is in your list. The `del` keyword can be used to delete an element or range of elements from a list (by position)[7], 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.
```

[6] A known limitation of SNAP 2.6 is that while `list[position] = list[position] + 1` does work, `list[position] += 1` does not. This will be addressed in a future release of SNAP firmware.

[7] A known limitation of SNAP 2.6 is that del does not function when provided negative index values. This will be addressed in a future release of SNAP firmware.

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 node 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.)

You can use byte lists as parameters in functions calls within a node, but you cannot send them as parameters in RPC calls between nodes. You can use `chr(myByteList)` to convert the byte list to a string and pass that, though.

SNAPpy's `type()` function returns an 8 for variables of type Byte List.

Unsupported Data Types

SNAPpy currently *does not* support the following common Python types, so they cannot be used in SNAPpy scripts. They ***can*** still be used in Python scripts running in Portal or a SNAP Connect application.

- 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.
- User-defined objects (*class* types)

Keywords

The following Python reserved identifiers are supported in SNAPpy:

- and
- break
- continue
- def
- del
- elif
- else
- for[8]
- from
- global
- if
- import
- in[9]
- is
- not
- or
- pass
- print
- return
- while
- xrange[10]

The following identifiers are reserved, but not yet supported in SNAPpy:

[8] for loop support was added in release 2.6.

[9] Support for in, both as a test of inclusion and as support for iteration, was added in release 2.6.

[10] xrange support was added in release 2.6.

- as
- assert
- class
- except
- exec
- finally
- lambda
- raise
- try
- with
- yield

Operators

SNAPpy supports all Python operators, with the exception of floor (//) and power (**).

```
+       -       *       /       %

<<      >>      &       |       ^       ~

<       >       <=      >=      ==      !=      <>
```

This extends to operators that assign a changed value to a variable.

```
x = 1
x += 4  # x now equals 5
y = "1"
y += "4"  # y now equals "14"
```

Slicing

Slicing is supported for byte list[11], string and tuple data types. For example, if x is “ABCDE” then x[1:4] is “BCD”.

Concatenation

Concatenation is supported for string data types. For example, if x = “Hello” and y = “, world” then x + y is “Hello, world”. Starting in SNAP version 2.6 String multiplication is supported, so you can now use 3 * "Hello! " to get "Hello! Hello! Hello! " in SNAPpy as you can in Python.

Subscripting

Subscripting is supported for byte list[12], string, and tuple data types. For example, if x = ('A', 'B', 'C') then x[1] = 'B'.

NOTE – Prior to version 2.2, there was only a single “string buffer” for each type of string operation (slicing, concatenation, subscripting, etc.). Subsequent operations of that same type would overwrite previous results. Version 2.2 replaces the fixed string buffers with a small pool of string buffers, usable for any operation. This allows scripts like the following to now work correctly:

```
A = B + C  # for this example, all variables are strings
D = E + F
```

Scripts that do string manipulations that were written to work within the 2.0/2.1 restrictions will still work as-is. They just may be performing extra steps that are no longer needed with version 2.2 and above.

Expressions

SNAPpy supports all Python Boolean, binary bit-wise, shifting, arithmetic, and comparison expressions – including the ternary if form.

```
x = +1 if a > b else -1 # x will be +1 or -1 depending on the values of a and b
```

[11] byte list support was added in release 2.6.

[12] byte list support was added in release 2.6.

Python Built-ins

The following Python built-ins are supported in SNAPpy:

- chr – Given an integer, returns a one-character string whose ASCII is that number. The result of `chr(65)` is 'A '. If applied to a byte list, it returns a string of the same length, where the `ord()` of each character matches the value of the byte in the original list.
- int – Given a string, returns an integer representation of the string. The result of `int('5 ')` is 5.
- len – Returns the number of items in an object. This will be an element count for a tuple, or the number of characters in a string.
- ord – Given a one-character string, returns an integer of the ASCII for that character. The result of `ord('A')` is 65.
- str – Given an element, returns a string representation of the element. The result of `str(5)` is '5' for example. The result of `str(True)` is 'True'.
- type – Given a single argument, it returns an integer indicating the data type of the argument. Note that the format of the return value is different from that returned in Python. The `type()` function is new in SNAP 2.5.
- xrange – 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. Support for `xrange` is new in SNAP 2.6.

Additionally, many RF module-specific embedded network and control built-ins are supported.

Print

SNAPpy also supports a print statement. Normally each line of printed output appears on a separate line. If you do not want to automatically advance to the next line (if you do not want an automatic Carriage Return and Line Feed), end your print statement with a comma (“,”) character.

```
print "line 1"
print "line 2"
print "line 3 ",
print "and more of line 3"
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.

5. SNAPpy Application Development

This section outlines some of the basic issues to be considered when developing SNAP-based applications.

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.

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_GPIO functionality.

To understand why "hard loops" like the one shown above are so bad, take a look at the flowchart on the following page.

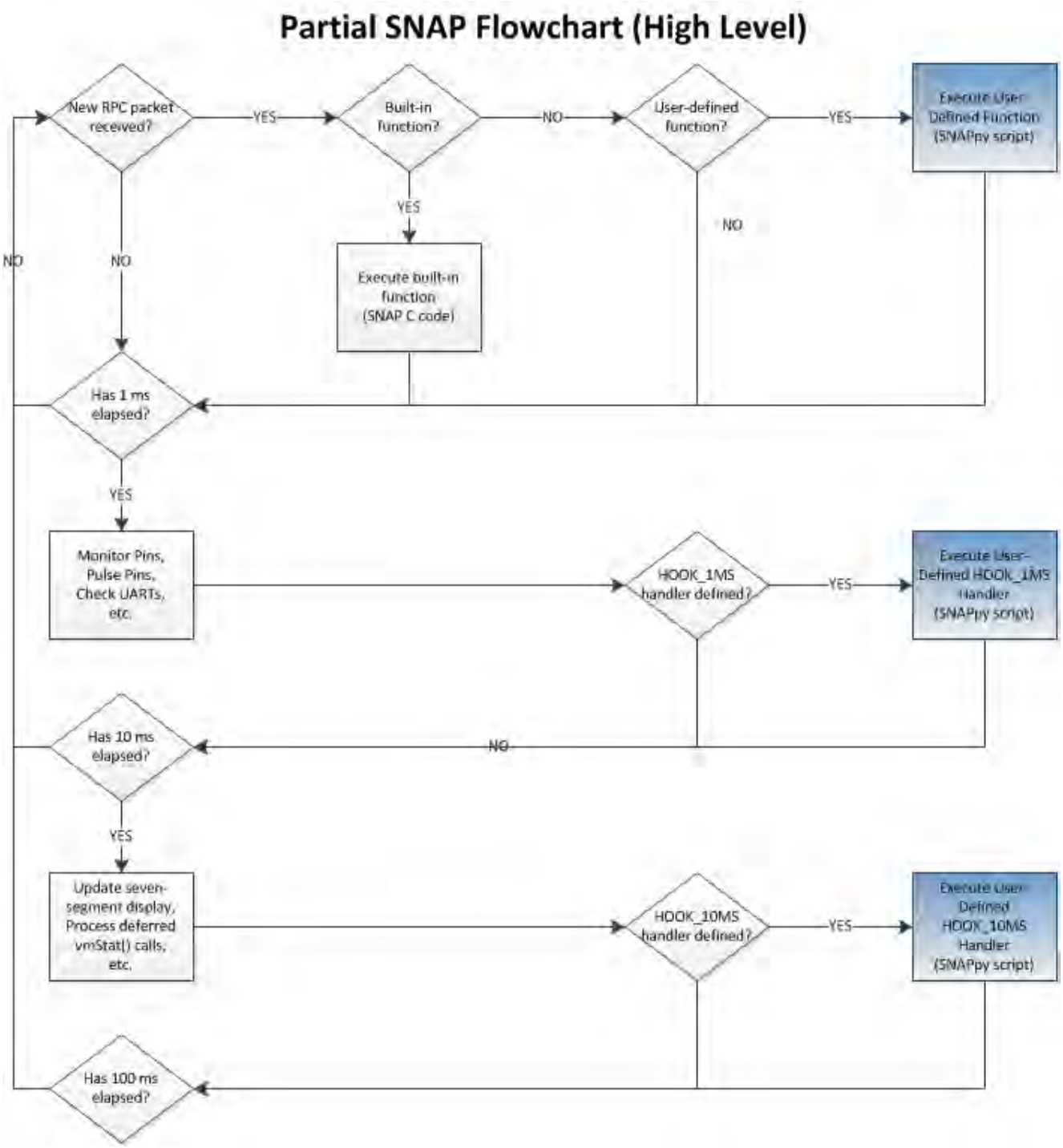
NOTE – the flowchart is not exhaustive, and only shows high-level processing!

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". The SNAP OS 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 node is busy interpreting the SNAPpy script, the other functions (the ones not highlighted) are not getting a chance to run. The SNAP OS 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 won't even get the correct quantity of timer hooks – SNAP sets a flag indicating that a timer hook needs to be invoked, 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 – 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 work-flow.

SNAP Hooks

There are a number of events in the system that you might like to trigger some SNAPpy function "handler." When defining your SNAPpy scripts, there is a way to associate functions with these external events. That is done by specifying a "HOOK" identifier for the function. The following HOOKs are defined:

Hook Name	When Invoked	Parameters	Sample Signature

HOOK_STARTUP	Called on device bootup	HOOK_STARTUP passes no parameters.	`@setHook(HOOK_STARTUP)`
			`def onBoot():`
			`  pass`

HOOK_GPIN	Called on transition of a monitored hardware pin	• pinNum – The pin number of the pin that has transitioned.
		• isSet – A Boolean value indicating whether the pin is set.
			`@setHook(HOOK_GPIN)`
			`def pinChg(pinNum, isSet):`
			`  pass`

HOOK_1MS	Called every millisecond	• tick – 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.	`@setHook(HOOK_1MS)`
			`def doEvery1ms(tick):`
			`  pass`

HOOK_10MS	Called every 10 milliseconds	• tick – 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.	`@setHook(HOOK_10MS)`
			`def doEvery10ms(tick):`
			`  pass`

HOOK_100MS	Called every 100 milliseconds	• tick – 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.	`@setHook(HOOK_100MS)`
			`def doEvery100ms(tick):`
			`  pass`

HOOK_1S	Called every second	• tick – A rolling 16-bit integer incremented every millisecond indicating the current count on the	`@setHook(HOOK_1S)`
			`def doEverySec(tick):`
			`  pass`

		internal clock. The same counter is used for all four timing hooks.	

HOOK_STDIN	Called when "user input" data is received	• data – A data buffer containing one or more received characters.	`@setHook(HOOK_STDIN)`
			`def getInput(data):`
			`  pass`

HOOK_STDOUT	Called when "user output" data is sent	HOOK_STDOUT passes no parameters.	`@setHook(HOOK_STDOUT)`
			`def printed():`
			`  pass`

HOOK_RPC_SENT	Called when the buffer for an outgoing RPC call is cleared	• bufRef – 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.	`@setHook(HOOK_RPC_SENT)`
			`def rpcDone(bufRef):`
			`  pass`

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.

Within a SNAPpy script, there are two methods for specifying the correct handler for a given HOOK event:

The modern way – @setHook()

Immediately before the routine that you want to be invoked, put a

`@setHook(HOOK_xxx)`

where HOOK_xxx is one of the predefined HOOK codes given previously. This method, known as using a "function decorator," is used in the samples provided above.

The old way (required for SNAP versions before version 2.2) –setHook()

Somewhere *after* the routine that you want to be invoked (typically these lines are put at the bottom of the SNAPpy source file), put a line like:

`setHook(HOOK_XXX)(eventHandlerXXX)`

where eventHandlerXXX should be replaced with the real name of your intended handling routine.

This method still works in the current version, but most people find the new way much easier to remember and use. The older way, however, allows you to define hooked functions in a module that is imported into your main file and then (from within your main file, after the import statement) specify that the hooks should be applied. (You cannot specify hooks in an imported file using the function decorator.)

Be sure to "hook" the correct event. For example, HOOK_STDIN lets SNAP Nodes process incoming serial data. HOOK_STDOUT lets SNAP Nodes 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.

Transparent Data (Wireless Serial Port)

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

Scripted Serial I/O (SNAPpy STDIO)

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.

SNAPpy scripts can also interact directly with the serial ports, allowing custom serial protocols to be implemented. For example, one of the included sample scripts shows how to interface serially to an external GPS unit. The SNAP node can be either the consumer or the creator of the serial data.

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. 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.)

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 sources (bidirectional)
uniConnect(dst, src) # Data from src goes to dst (unidirectional)
```

For example, 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 row-column cross-connect. Note that the "DS_" prefixes have been omitted to save space.

	UART0	UART1	TRANSPARENT	STDIO	PACKET_SERIAL
UART0	Loopback	Crossover	Wireless Serial	Local Terminal	Local SNAP Connect, Portal, or another SNAP Node
UART1	Crossover	Loopback	Wireless Serial	Local Terminal	Local SNAP Connect, Portal, or Another SNAP Node
TRANSPARENT	Wireless Serial	Wireless Serial	Loopback	Remote Terminal	Remote SNAP Connect

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 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 Portal while allowing other serial data to be directed to a UART.

You can configure Portal (using the preferences menu) to intercept just errors, non-error text, or both when you intercept STDIO to the application. Refer to the **Portal Reference Manual.**

Loopback

A command like `crossConnect(DS_UART0, DS_UART0)` will setup 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

As mentioned previously, a command of: `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. Refer to the API section of the **SNAP Reference Manual** regarding the ucastSerial() and mcastSerial() functions.

Local Terminal

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

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 Node.

Packet Serial

The last column of the table shows the effect of various combinations using DS_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 Portal or SNAP Connect.

It also allows serial connection to another SNAP Node, if the appropriate "cross-over" cable is used.

This allows "bridging" of separate SNAP Networks (networks that are on different channels and/or different Network IDs, or two nodes that work in different radio frequency ranges).

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.

Refer to the API documentation on crossConnect() in the **SNAP Reference Manual** for more details.

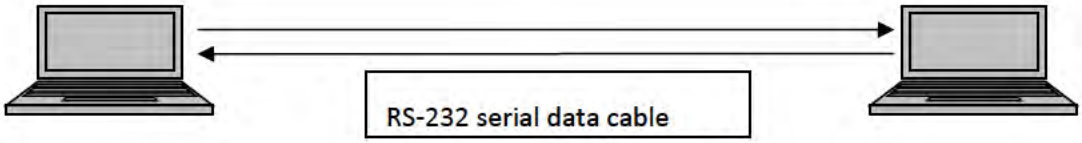
Debugging

Application development with SNAP offers an unprecedented level of interactivity in embedded programming. Using Portal you can quickly upload code, test, adjust, and try again. Some tips and techniques for debugging:

- Make use of the "print" statement to verify control flow and values. (Be sure to connect STDIO to a UART or Intercept STDOUT with Portal.)
- When using Portal's Intercept feature, you'll get source line-number information, and symbolic error-codes.
- Invoke "unit-test" script functions by executing them directly from the Snappy Modules Tree in Portal's Node Info panel.
- Use the included SNAP Sniffer to observe the RPC calls between devices.

Sample Application – Wireless UART

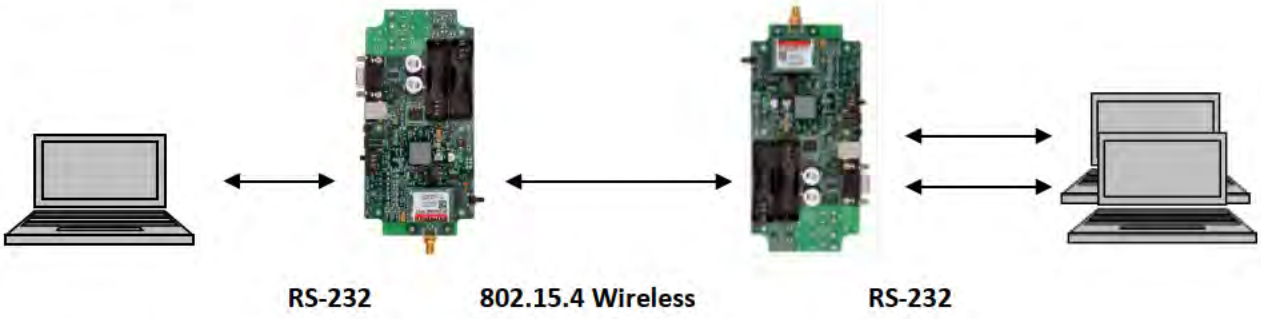
The following scenario is very common: two devices communicating over an RS-232 serial link.



The two devices might be two computers, or perhaps a computer and a slave peripheral. For the remainder of this section, we will refer to these devices as "end points."

In some cases, a direct physical connection between the two end points is either inconvenient (long distance) or even impossible (mobile end points).

You can use two SNAP nodes to wirelessly emulate the original hardwired connection. One SNAP node gets paired with each end point. Each SNAP node communicates with its local end point using an RS-232 port (such as the ones on the Synapse demonstration boards), and communicates wirelessly with the SNAP node connected to other end point.



To summarize the requirements of this application:

We want to go from RS-232 to wireless, and back to RS-232

We want to implement a point-to-point bidirectional link

We don't want to make any changes to the original endpoints (other than cabling)

This is clearly a good fit for the **Transparent Mode** feature of SNAPpy, but there are still choices to be made around "how will the nodes know *who* to talk to?"

Option 1 – Two Scripts, Hardcoded Addressing

A script named dataMode.py is included in the set of example scripts that ships with Portal. Because it is one of the demo scripts, it is write-protected. Using Portal's "Save As" feature, create two copies of this script (for example, dataModeA.py and dataModeB.py). You can then edit each script to specify the other node's address, before you upload both scripts into their respective nodes.

The full text of dataMode.py is shown here. Notice this script is only 19 lines long, and 8 of those lines are comments (and 3 are just whitespace).

```
"""
Example of using two SNAP wireless nodes to replace a RS-232 cable
Edit this script to specify the OTHER node's address, and load it into a node
Node addresses are the last three bytes of the MAC Address
MAC Addresses can be read off of the SNAP Engine sticker
For example, a node with MAC Address 001C2C1E 86001B67 is address 001B67
In SNAPpy format this would be address "\x00\x1B\x67"
"""
from synapse.switchboard import *
otherNodeAddr = "\x4B\x42\x35" # <= put the address of the OTHER node here
@setHook(HOOK_STARTUP)
def startupEvent():
    initUart(1, 9600) # <= put your desired baud rate here!
    flowControl(1, False) # <= set flow control to True or False as needed
    crossConnect(DS_UART1, DS_TRANSPARENT)
    ucastSerial(otherNodeAddr)
```

The script as shipped defaults to 9600 baud and no hardware flow control. Edit these settings as needed, too.

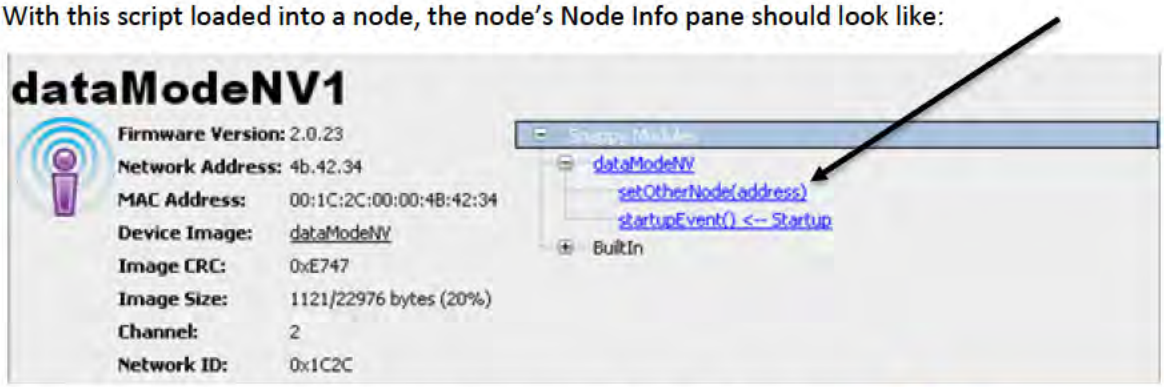
With these two edited scripts loaded into the correct nodes (remember, you are telling each node who the *other* node is; each node already knows its own address), you have just created a wireless serial link.

Option 2 – One Script, Manually Configurable Addressing

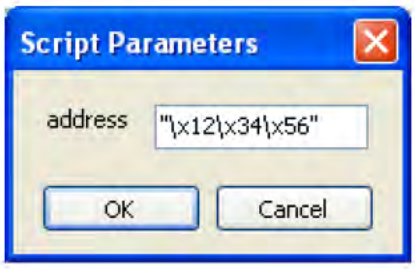
Instead of hard-coding the "other node" address within each script, you could have both nodes share a common script, and use SNAPpy's Non-Volatile Parameter (NV Param for short) support to specify the addressing after the script was loaded into the unit.

Look in your snappyImages directory for a script named dataModeNV.py. Since we won't be making any changes to this script, there is no need to make a copy of it. Simply load it into both nodes as-is.

With this script loaded into a node, the node's Node Info pane should look like:



Click on setOtherNode(address) in the Snappy Modules tree, and when prompted by Portal, enter the address of the *other* node **as a quoted string** (standard Python "binary hex" format).



For example, if the other node is at address 12.34.56, you would enter "\x12\x34\x56" in the Portal dialog box.

Do this for both nodes.

Here is the source code to SNAPpy script dataModeNV.py

```
"""
Example of using two SNAP wireless nodes to replace a RS-232 cable
After loading this script into a SNAP node, invoke the setOtherNode(address)
function (contained within this script) so that each node gets told "who his
counterpart node is." You only have to do this once (the value will be preserved
across power outages and reboots) but you DO have to tell BOTH nodes who their
counterparts are!
The otherNodeAddr value will be saved as NV Parameter 254, change this if needed.
Legal ID numbers for USER NV Params range from 128-254.
Node addresses are the last three bytes of the MAC Address
MAC Addresses can be read off of the SNAP Engine sticker
For example, a node with MAC Address 001C2C1E 86001B67 is address 001B67
In SNAPpy format this would be address "\x00\x1B\x67"
"""
from synapse.switchboard import *
OTHER_NODE_ADDR_ID = 254
@setHook(HOOK_STARTUP)
def startupEvent():
    """System startup code, invoked automatically (do not call this manually)"""
    global otherNodeAddr
    initUart(1, 9600) # <= put your desired baudrate here!
    flowControl(1, False) # <= set flow control to True or False as needed
    crossConnect(DS_UART1, DS_TRANSPARENT)
    otherNodeAddr = loadNvParam(OTHER_NODE_ADDR_ID)
    ucastSerial(otherNodeAddr)
def setOtherNode(address):
    """Call this at least once, and specify the OTHER node's address"""
    global otherNodeAddr
    otherNodeAddr = address
    saveNvParam(OTHER_NODE_ADDR_ID, otherNodeAddr)
    ucastSerial(otherNodeAddr)
```

This script shows how to use the saveNvParam() and loadNvParam() functions to have units remember important configuration settings. The script could be further enhanced to treat the baud rate and hardware handshaking options as User NV Parameters as well.

You can read more about NV Parameters in the **SNAP Reference Manual.**

Code Density

When you upload a SNAPpy script to a node, you are not sending the raw text of the SNAPpy script to the node. Instead the SNAPpy source code is compiled into byte-code for a custom Virtual Machine (the SNAPpy VM), and this byte-code and data (SNAPpy "Image") are sent instead.

We have not performed an exhaustive analysis, but a quick check of two typical example scripts (ZicCycle.py and Zic2410i2cTests.py) showed code densities of 11.625 and 13.138 bytes/line of code.

So, a conservative estimate of SNAPpy code density is 10-15 bytes per line of SNAPpy code.

Simple code will have a higher density; scripts that include a lot of data (for example, text) will be lower.

For example:

```
def hi():
    print "Hi"
```

takes 35 bytes, but

```
def hi():
    print "Hello everyone, I know my ABCs! - ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

takes 91 bytes.

Keeping text messages and function names short will help conserve SNAPpy script space.

Cross-Platform Coding and Easy Pin Numbering

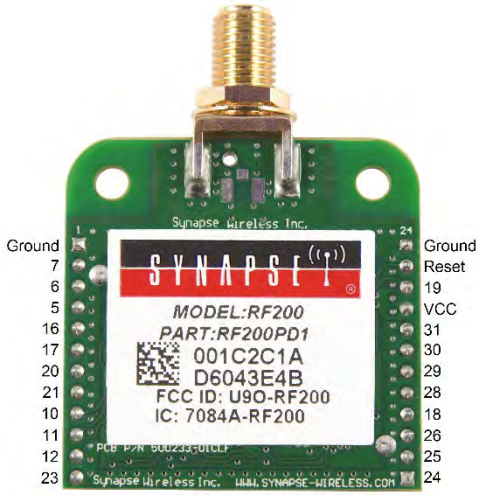
The first hardware platform to support SNAP was the RF100, based on Freescale's MC9S08 processor family. For the sake of simplicity, SNAP pin numbers were assigned sequentially to the pins available from the various ports on the microprocessor, and brought out to the SNAP Engine footprint sequentially, as shown in the image to the right.

If you have an LED connected to the third pin on the SNAP Engine (counting from the upper left) and you wish to blink it, your SNAPpy code could simply use code like this:

```
pinNum = 1
duration = 500  # Milliseconds
flashOn = True
pulsePin(pinNum, duration, flashOn)
```

As long as the code you write will always run on RF100 SNAP Engines, and your hardware always has its LED connected to that third pin, you could always pulse pin 1 and the LED would respond appropriately.

However, you might have a network that includes more than one hardware platform, and pulsing pin 1 on an RF200 would not give you the same results. As new types of SNAP Engines were based on different underlying hardware platforms, it was more important that pin functionality be preserved than that pin numbering compatibility be maintained. For example, the RF200 has two UARTs, just as the RF100 does, and those UARTs are available on the same pins on the SNAP Engine footprint. On both an RF100 and an RF200, UART1 (the default UART) comes out on pins 9 and 10 on the SNAP Engine footprint. On an RF100, those pins are referenced in SNAPpy code as pins 7 and 8. On an RF200, those pins are referenced in SNAPpy code as pins 10 and 11.



With an RF200, in order to flash an LED connected to the SNAP Engine's third pin, as we did above with the RF100, you would need to use a command like this (simplified from the above example):

```
pulsePin(6, 500, True)
```

Fortunately, SNAP provides an easy mechanism for simplifying cross-platform coding with the Platform value, stored in NV Parameter 41. SNAP nodes produced by Synapse Wireless will come with this field populated with the node type, e.g., "RF100" or "RF200" or "RF300". You can modify the value just as you could any other NV Parameter, but first let's take a look at why you might want to keep the default.

For starters, let's approach this problem the hard way. You could easily make use of the contents of the Platform parameter like this:

```
if loadNvParam(41) == 'RF100':
    pulsePin(1, 500, True)
elif loadNvParam(41) == 'RF200':
    pulsePin(6, 500, True)
```

But SNAP includes some things to improve on this. When you install Portal, a number of example scripts are added to your PC in the snappyImages directory. That directory contains a subdirectory named Synapse, which contains more scripts that help support the example scripts, and which you can use in your own scripts. The one we're most interested in at the moment is named platforms.py, and you add it to your script like this:

```
from synapse.platforms import *
```

Importing the platforms.py file like this tells SNAP to pay special attention to what's in the Platform field, and makes a "platform" variable available to you within your SNAPpy script. So now, you could do something like this:

```
from synapse.platforms import *
if platform == 'RF100':
    pulsePin(1, 500, True)
elif platform == 'RF200':
    pulsePin(6, 500, True)
```

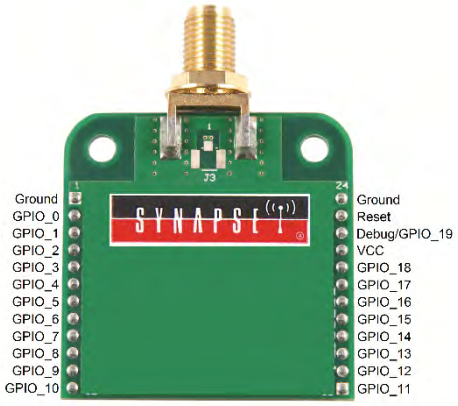
That isn't a lot better than the first code sample, though. You could just as easily assign the contents of NV Parameter 41 to a variable named platform yourself without importing the platforms.py file. But if you look into the contents of the platforms.py file, you'll see that it does more than that.

The first thing platforms.py does is import snapsys.py. This is the file that lets the system recognize the platforms variable. (The actual setting of the variable happens "behind the scenes" in SNAP through the inclusion of this file.) The platforms.py file then uses this platform variable to select another file to import, such as RF100.py for RF100s (or nodes that contain "RFEngine" in NV Parameter 41, for purposes of backwards compatibility), or RF200.py for RF200 SNAP Engines.

If you peruse those files, you'll see that each contains a collection of constant assignments with names like "GPIO_1". The "GPIO" in this constant name stands for "general-purpose input/output", and gives us the Rosetta Stone we need to translate from platform to platform. Notice that for the RF100, GPIO_1 is set to 1, while for the RF200, GPIO_1 is set to 6. (These numbers may look familiar, if you've been reading closely.)

Now you can simplify your previous code this way:

```
from synapse.platforms import *
pulsePin(GPIO_1, 500, True)
```



For an RF100, the GPIO_1 constant will already be defined as 1, and for an RF200 it will already be defined as 6. You don't need to make any conditional branch in order to affect the same pin on the SNAP-Engine-footprint. Just refer to the pin by its position on the SNAP Engine and the table of constants performs the translation for you.

The RF100.py and RF200.py files (and others like them) provide other useful conversions, too. Each contains a tuple constant, GPIO_TO_IO_LIST, that maps SNAP Engine pins to their corresponding SNAP constant values (i.e., for an RF200, position 1 in the tuple contains a 6), and another tuple constant, IO_TO_GPIO_LIST, that maps SNAP constants to their corresponding SNAP Engine pins (i.e., for an RF200, position 6 in the tuple contains a 1). This can be useful for code like this:

```
from synapse.platforms import *
@setHook(HOOK_GPIN)
def onSignal(whichPin, isSet):
    gPin = IO_TO_GPIO_LIST[whichPin]
    print 'GPIO_', gPin, ' is ', isSet
```

If this were set on the bottom right pin on a SNAP Engine, it would print "GPIO_11 is False" or "GPIO_11 is True" (depending on the isSet state) when the state of that pin changed (assuming you were monitoring that pin). This

translation to 11 would occur whether the whichPin variable received an 11 (as it would on an RF100) or a 24 (as it would on an RF200), making the output easy to interpret.

It's important to consider that these GPIO *numbers* are only meaningful in the context of a SNAP Engine, a SNAP node on Synapse's 24-pin through-hole module footprint. If you are working with one of the surface-mount SNAP modules, such as the SM200 or the SM220, the pin notation is not useful.

For surface-mount modules, then, the platforms file imports constants based on the eight-by-eight grid of pads on the module's underside. GPIO_D2 on an SM200 is equal to 6 when used in your SNAPpy code. That is the same pin, internally, that comes out as GPIO_1 on an RF200. But on the surface-mount module, it is the second pin up in the fourth row (row D) over. Similarly, GPIO_A4 on an SM200 is 24, which would be GPIO_11 on an RF200. (Any pad that does not have a meaningful value to SNAP, such as a power or ground pin, is set equal to -1 for the surface-mount modules.)

For either of these formats, you could use the GPIO_TO_IO_LIST tuple, for example, to drive all the output pins on a SNAP Engine high:

```
from synapse.platforms import *
def setAllHigh():
    whichPin = 0
    while whichPin < len(GPIO_TO_IO_LIST):
        pin = GPIO_TO_IO_LIST[whichPin]
        setPinDir(pin, True)
        writePin(pin, True)
        whichPin += 1
```

Simplifying the mapping of GPIO pins is only part of the power of the platforms.py file. By having the platform variable defined, you can take advantage of other special features of the various nodes:

```
from synapse.platforms import *
if platform == 'RF100':
    sleepMode = 0
elif platform == 'RF200':
    sleepMode = 1
sleep(sleepMode, 20)
```

On an RF200, sleep mode 0 results in a significantly higher sleep current than sleep mode 1 does. Mode 0 exists for AT128RFA1 hardware configurations that do not include an external timing crystal. But the RF200 does include that crystal, so sleep mode 1 is preferred. Using a script like the one above, sleeping on either platform will give the best efficiency.

The real "magic" part of this platform variable process is a little more subtle than has been made evident so far. The platform variable is available to Portal at compile time, rather than just being a run-time variable. So if your SNAPpy script includes code sections conditionally based on the platform variable, any parts that don't apply for the selected platform won't even be included in the script loaded onto your node. This prevents incorrect code from being run if the platform parameter is accidentally changed after the code is loaded, and more importantly, can significantly reduce the size of the image loaded into your node.

So far, the assumption has been that the platform would be used only to distinguish between nodes based on different underlying hardware. But you can set NV Parameter 41 to any value that suits your needs, and thus have the platform variable be anything you like. (If you were to do so, you could still explicitly include the GPIO mappings by importing synapse.RF200 or synapse.SM220 or the other appropriate mapping file to gain access to GPIO constants for easy pin referencing.)

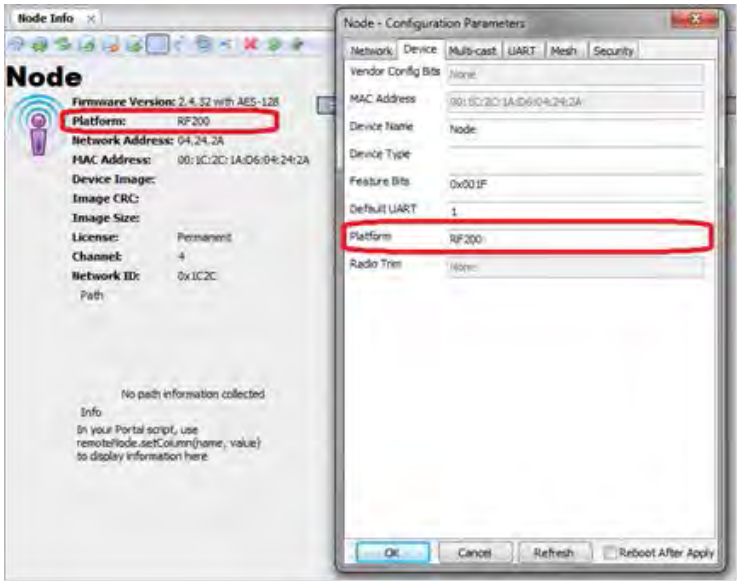
Defining your own platforms could be useful in a situation where you wanted to maintain a complete application's code base in one source file, even though different devices used in your installation required different functionality.

For example, you may have different revisions of hardware connecting to your SNAP Engines, with discrepancies in how things are connected, but wish to have only a single SNAPpy script code base to maintain. The difference could be a simple as connecting an LED to a different pin, or making a serial connection at a different baud rate. Or, it could be much more complicated, where a "collectData()" function for one board revision makes a simple serial connection, while on a later board revision it makes an I^2C connection to harvest the appropriate information.

Again, much of this could be accomplished with a user-defined NV Parameter as well. What NV Parameter 41 and the platform variable provide is access to this level of control at the point where your SNAPpy script is being compiled before loading it into your node, which allows selective control of which other SNAPpy modules are imported into your script, and which helps reduce the code size of your script, leaving more script space for more elaborate SNAPpy scripts.

Because the platform value's effects are felt at compile time rather than run time, it is important to make sure you specify the correct value for the target platform any time you export a SNAPpy script to a *.spy file from Portal. (The export window defaults the value to that of the selected node.)

Setting the platform variable is as easy to do as setting any other NV Parameter in SNAP. The Device tab of the Configuration Parameters window provides access to the Platform parameter, NV Parameter 41. The value specified for a node also displays in the Node Info pane within Portal.



6. Invoking Functions Remotely – Function Rpc() and Friends

Remote Procedure Call (RPC)-related functionality is such a fundamental part of SNAP and SNAPpy that it is worthy of a section by itself. Refer also to the SNAP Primer for even more basic coverage of this topic, and the SNAP Reference Manual for even more detailed coverage.

There are five SNAPpy built-in functions that can be used to invoke a function on another node:

- `mcastRpc()`
- `dmcastRpc()`
- `rpc()`
- `callback()`
- `callout()`

These are non-blocking functions. They only enqueue the packet for transmission, and do not wait for the packet to be sent (let alone waiting for it to be processed by the receiving node(s)).

Each can be used to invoke any public function on another SNAP Node (either a user-defined function, or a built-in function), but each has additional strengths, weaknesses, and capabilities.

The reliability of message delivery is an important consideration in selecting the best way to send the message. Multicast messages (those sent using mcastRpc()) generate no explicit confirmation of receipt from any other node in your network. If you want to be sure that a particular node 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 node.

Unicast RPC calls are addressed to a single target node rather than being open to all nodes 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.

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

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

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

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

For either of these transmissions, if the receiving node does not send an acknowledgement packet within a configurable timeout period, the sending node 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 node Alice has discovered a route through node Bob to node Carol, and sends node Carol a message. Node Bob hears the message and sends node Alice an acknowledgement, so node Alice goes on about its business confident that its message is delivered. However at the point that node Alice hears the acknowledgement, node Carol has not yet received the message from node Bob. If node Carol is sleeping or is otherwise unable to hear the message from node Bob, node Bob will attempt the configured number of retries, but will eventually give up if node Carol cannot be found – and (other than a route failure message that goes out, indicating to node Alice that the next time it tries to communicate with node Carol, it should first perform a new route request) this failure to forward the message will not be reported back to node Alice.

If you need to be sure that your target node 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.

Addressing SNAP Nodes

Each SNAP node has a unique SNAP address, defined by the last three bytes of the node's MAC address. Thus a SNAP node 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 nodes reference each other (to send procedure calls) using these addresses, both for directed multicasts and for addressed 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 node's address if you have some function indicating from where a message was received. A function like the following[13] will "decode" the address to human-readable form, as an eight-byte string that is no longer valid as a SNAP address for sending message requests:

```
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
```

McastRpc(*group, ttl, function, args…*)

Built-in function mcastRpc() is best at invoking the same function on multiple nodes from a single invocation. The tradeoff is that the actual packet is usually only sent once. (The exception to that rule is if you have enabled

[13] The provided function uses a "for" loop, introduced in SNAP 2.6. A "while" loop can be used in earlier SNAP versions.

the optional collision detect feature of SNAP, in which case the packet might be sent more than once if a packet collision were detected.)

How many nodes actually perform the requested function call is a function of three factors:

1) Number of hops
   a) How many hops away are the other nodes?
   b) How many hops did you specify in the mcastRpc() call? (TTL parameter)
2) Group membership
   a) What multicast groups do the other nodes belong to?
   b) What candidate groups did you specify in the mcastRpc() call?
   c) If the destination node is more than one hop away, do intermediate nodes forward the specified group?
3) Only nodes that actually have the requested function can execute it

Examples:

```
mcast_groups = 2
hop_count = 4
mcastRpc(mcast_groups, hop_count, "startDataCapture")
```

All nodes in the second group that can be reached within 4 or fewer "hops" (forwards, or retransmissions of the message) will invoke function "startDataCapture" (if they have a function with that name in their currently loaded script).

This is an example of using multicast RPC to increase the synchronization of the nodes. Note that all of the nodes will not necessarily be "startingDataCapture" at the same moment, because the closer nodes will be hearing the command sooner than the nodes that require more mesh network hops.

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

This is an exaggerated example, but the point is that sometimes you are sending to multiple nodes because of the importance of the command.

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

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

Now imagine node A contains the following script snippet:

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

If node A later does:

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

then two procedure calls can occur. First node A will invoke askSensorReading() on node B, but then that routine will invoke routine tellSensorReading() back on node A.

You should also be aware that even though a RPC call is made via multicast, it still is possible to have only a single node completely process that call.

Imagine the above "two node network" is expanded with nodes "C"-"Z", but that we also change to script to be:

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

Assume nodes C-Z are loaded with the same script as node B. If node A later does:

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

then even though all nodes within a one-hop radius will invoke function askSensorReading() only node B will actually take a reading and report it back.

Of course, wanting to invoke a function on a single node is actually a pretty common use case, which is where the following function comes in.

dmcastRpc(*dstAddrs, group, ttl, delayFactor, function, args…*)

The directed multicast function was added in SNAP release 2.6. It functions in much the same way that the "regular" multicast does as far as the group, ttl, function, and function arguments go. However it adds two additional parameters: dstAddrs, and delayFactor.

The dstAddrs parameter lets you target one or more specific node(s) on which your function should execute. If a node does not find its own address in the dstAddrs parameter of an incoming mcastRpc call, it will still forward the message to other nodes (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 node to do so.

You can specify one target node by sending that node's three-character SNAP address as the dstAddrs parameter. If you want to target more than one node, simply concatenate multiple SNAP addresses into a longer string for the parameter. A dstAddrs value of "\x00\x66\x9b\x05\x47\x56\x5f\xe6\x23" would be acted on by all three nodes (00.66.9B, 05.47.56, and 5F.E6.23) if all three of those nodes are reachable by the network in the specified TTL, and the execution groups for the nodes match the groups specified in the call.[14]

The delayFactor parameter allows you to define a time slice (in milliseconds) delay so that multiple target nodes can respond in a sequential manner rather than all at once. If you sent a directed multicast to the three nodes in the example above with a delayFactor of 40, invoking a function that sends a reply back to the calling node, node 00.66.9B would send its reply immediately, node 05.47.56 would send its reply after a 40 millisecond delay, and node 5F.E6.23 would send its reply after an 80 millisecond delay.

See the SNAP Reference Manual for further information about the dmcastRpc function.

Rpc(*address, function, args…*)

Built-in function rpc() is like mcastRpc(), but with two differences:

1) Instead of specifying a group and a number of hops (TTL – Time To Live), with the rpc() function you specify the actual SNAP Address of the intended target node
   a) The SNAP Mesh routing protocol will take care of "finding" the node (if it can be found).
   b) Other nodes (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 node, if it is not in direct communication range of the source node.

[14] If you specify an empty dstAddrs value, any node that hears the request (and matches multicast execution groups) will execute the function.

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 node does hear 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 node does not hear the rpc() call, then it does not know to send the ACK packet. This means the source node 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 node 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")
```

Nodes C-Z would ignore the function call (although they may be helping route the function call to node B without any additional configuration required by the user).

The askSensorReading() function could also benefit from the use of rpc() instead of mcastRpc().

Instead of telling all nodes in group 1 and 1 hop away what the sensor reading is via:

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

the script could instead send the results back to the original requestor only 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 node. (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.)

Note that the functions being invoked, by either an rpc() call or an mcastRpc() call, could be built-in (such as readPin(), readAdc(), random()) or could be user-defined (defined within the currently loaded SNAPpy script).

Callback(*callback, remoteFunction, remoteFunctionArgs…*)

Both the previous methods allowed one node to ask another node to perform a function and then send the result of that function back to the first node. In each case, the first node called the askSensorReading() function, whose only purpose was to call a separate function (readSensor()) and then send back the value from that.

It turns out that SNAP has a built-in function to do just that, the callback function.

Going back to a previous example, and expanding on it a little:

```
def readSensor():
    return readAdc(0) * SENSOR_GAIN + SENSOR_OFFSET
def askSensorReading():
    value = readSensor()
    mcastRpc(1, 1, "tellSensorReading", value)
```

In a scenario like this, routine readSensor() is pulling its own weight – it is encapsulating some of the sensor complexity, thus hiding it from the rest of the system.

Sometimes, however, the raw sensor readings are sufficient (or the calculations are so complex that they need to be offloaded to a bigger computer).

This results in the code being changed to something more like:

```
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 think that instead of calling

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

node A could simply call

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

Although this will result in node B calling his readAdc() function, it won't actually cause any results to be sent back to the caller.

This is where the callback() function comes in. Replacing

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

with

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

will do what you want – the readAdc() function will be invoked on node B, and the results automatically reported back to node A via the tellSensorReading() function.

Notice that in the actual callback() invocation, you must provide the final function to be invoked ("called back") in addition to still having to specify the initial function to be called, as well as any parameters it needs.

Notice also that in this case, we only had to add code to node A's script – we didn't have to create an "askSensorReading" function 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")
```

One common user of the callback() function is Portal itself. When you click on a function name in the Node Info pane of Portal, Portal is using callback() – not rpc() – behind the scenes to automatically get the result value so that it can print it in the Portal Event Log. For example, if you click on the random() function of a node, what Portal really sends is something equivalent to:

```
rpc(address_of_node, "callback", "printRtnVal", "random")
```

To summarize: callback() just lets you get the same data with less SNAPpy script.

Callout(*nodeAddress, callback, remoteFunction, remoteFunctionArgs…*)

Built-in function callout() just takes the callback() concept one step further.

Instead of asking a node to invoke a function and then call you back with the result, callout() is used to ask a node to call a function and then call a third node with the results.

For example, node A could ask node B to read his first analog input channel but tell node C what the answer was. (Imagine for instance that node C has a graphical LCD display that the other nodes lack.)

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

As another example, node A could ask node B to find out how well all the other nodes one hop away could hear node B, but send the answers to node A by doing something like:

```
rpc(b_addr, "mcastRpc", 1, 1, "callout", a_addr, "tell_link_quality", "getLq")
```

Translating the above line of SNAPpy script into English:

Node A is asking node B to send a 1-hop multicast that asks all nodes in multicast group number 1 that hear it to invoke their getLq() functions (this answering the question "how well did they hear node B's multicast transmission?") and send all of the results to node A.

Node A's script would have a snippet like:

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

Clever use of mcastRpc(), rpc(), callback(), and callout() can let you create applications like "Network Topology Explorers".

Additional Reminder

You cannot invoke a function that a node does not have.

In the case of mcastRpc(), the unit will silently ignore the function it does not know how to call.

If sent via one of the unicast mechanisms (rpc(), callback() or callout()) the RPC packet will be acknowledged, but then ignored.

7. Advanced SNAPpy Topics

This section describes how to use some of the more advanced features of SNAP. Topics covered include:

- Interfacing to external CBUS slave devices (emulating a CBUS master)
- Interfacing to external SPI slave devices (emulating a SPI master)
- Interfacing to external I^2C slave devices (emulating a I2C master)
- Interfacing to multi-drop RS-485 devices
- Encryption between SNAP nodes
- Recovering an unresponsive node

Interfacing to external CBUS slave devices

CBUS is a clocked serial bus, 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 (CS)

Using the existing 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(numToRead)` – "shifts in" the specified number of bytes
- `cbusWr(str)` – "shifts out" the bytes specified by str

To allow the cbusRd() and cbusWr() functions to be as fast as possible, the IO pins used for CBUS CLK, CDATA, and RDATA are fixed. On an RF100 SNAP Engine:

- GPIO 12 is always used as the CBUS CDATA pin
- GPIO 13 is always used as the CBUS CLK pin
- GPIO 14 is always used as the CBUS RDATA pin

For platforms other than the RF100, refer to the appropriate platform specific section in the back of the **SNAP Reference Manual**.

NOTE– These pins are only dedicated if you are actually using the CBUS functions. If not, they remain available for other functions.

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

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

```
# read bytes from the selected CBUS device
Response = cbusRd(10) # <- you specify how many bytes to read
# deselect the CBUS device
writePin(somePin, True) # assuming the chip select is active-low
```

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.

A detailed example of interfacing to an external CBUS voice chip may someday be the topic of an application note.

NOTE – Not all SNAP Engines support CBUS, refer to the **SNAP Reference Manual**.

Interfacing to external SPI slave devices

SPI is another clocked serial bus. It typically requires at least four pins:

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

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

Numerous options complicate 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 Least Significant Bit (LSB) first
- Data Width – some SPI devices are 8-bit, some are 12, some are 16, etc.

You can find more information on SPI at http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

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.

As with what was done for CBUS 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.

In order to support both three wire and four wire SPI, there are more spiXXX() functions than you might first expect.

- `spiInit(cpol, cpha, isMsbFirst, isFourWire)` – setup for SPI (supporting many options!)
- `spiWrite(byteStr, bitsInLastByte=8)` – send data out SPI
- `spiRead(byteCount, bitsInLastByte=8)` – receive data in from SPI (3 wire only)
- `spiXfer(byteStr, bitsInLastByte=8)` – bidirectional SPI transfer (4 wire only)

Four-wire SPI interfaces transfer data in both directions simultaneously, and should use the spiXfer() function.

Some SPI devices are write-only, and you can use spiWrite() to send data to them (three-wire or four-wire hookup).

Some three wire devices are read-only, and you must use function spiRead().

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

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

…will send a total of 12 bits: all of the first byte (0x12), and the first (or last) nibble of the second byte. Which 4 bits out of the total 8 get sent are specified by the “send LSB first” setting, which is specified as part of the spiInit() call.

To allow these functions to be as fast as possible, the IO pins used for CLK, MOSI, and MISO are fixed. For example, on a Synapse RF100 SNAP Engine, the following pins are used:

- GPIO 12 is always used as the MOSI pin
- GPIO 13 is always used as the CLK pin
- GPIO 14 is always used as the MISO pin, unless running in three wire mode

(The chip select pin is what raises the total number of pins from 3 to 4)

NOTE – These pins are only dedicated if you are actually using the SPI functions. If not, they remain available for other functions. Also, if using three wire SPI, GPIO 14 remains available.

For platforms other than the RF100, refer to the appropriate platform specific section in the back of the SNAP Reference Manual.

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

```
1. # select the desired SPI device
2. writePin(somePin, False) # assuming the chip select is active-low
3. # Transfer data to the selected SPI device
4. spiWrite(“\x12\x34\x56”)
5. # deselect the SPI device
6. writePin(somePin, True) # assuming the chip select is active-low
```

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.

For examples of using the SNAPpy SPI functions to interface to external devices, see the following scripts that are bundled with Portal:

- spiTests.py – This is the overall SPI demo script
- LTC2412.py – Example of interfacing to a 24-bit Analog To Digital convertor

Script spiTests.py imports the other script, and exercises some of the functions within it.

- ZIC2410spiTests.py – like spiTests.py but specifically for ZIC2410 evaluation board
- AT25FS010.py – Example of interfacing to an ATMEL Flash memory

Script ZIC2410spiTests.py imports the other script, and exercises some of the functions within it.

Interfacing to external I²C slave devices

Technically, the correct name for this two-wire serial bus is Inter-IC bus or I²C, though it is sometimes written as I2C. Information on this popular two-wire hardware interface is readily available on the Web; http:/www.i2c-bus.org/ is one starting point you could use. In particular look for a document called "The I2C-bus and how to use it (including specifications)."

I²C 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 I²C support functions have been added to SNAPpy.

NOTE– Parameters shown below in red are only available on selected hardware platforms and/or software versions, beginning with release 2.5. See the SNAP Reference Manual for more details.

- i2cInit(enablePullups, SCL_pin, SDA_pin) – Prepare for I²C operations (call this to set up for I²C)
- i2cWrite(byteStr, retries, ignoreFirstAck, endWithRestart) – Send data over I²C to another device
- i2cRead(byteStr, numToRead, retries, ignoreFirstAck) – Read data from device
- getI2cResult() – Check the result of the other functions

These routines are covered in more detail in the SNAP Reference Manual.

Using these routines, your SNAPpy script can operate as an I²C bus master, and can interact with I²C slave devices.

When performing I²C interactions, fixed IO pin assignments are usually used. For example, on an RF100 the following IO pins are used:

- GPIO 17 is always used as the I²C SDA (data) line
- GPIO 18 is always used as the I²C SCL (clock) line

Exceptions are the STM32W108xB platform, and (as of SNAP version 2.5) the ATmega128RFxx platforms. On these platforms, the i2cInit() function takes two additional platforms that specify which IO pins to use for I²C clock (SCL) and data (SDA). (For backwards compatibility, you do not have to specify values for the SCL_pin and SDA_pin parameters on ATmega128RFxx platforms. If you do not, SNAP will default the pins to GPIO_18 and GPIO_17, respectively.)

NOTE – These pins are only dedicated if you are actually using the I²C functions. If not, they remain available for other functions.

Refer to the platform-specific section for your hardware (located at the back of the **SNAP Reference Manual**) for the pin assignments for your platform.

Unlike CBUS and SPI, I²C does not use separate "chip select" lines. The initial data bytes of each I²C transaction specify an "I²C address." Only the addressed device will respond. So, no additional GPIO pins are needed.

The specifics of which bytes to send to a given I²C slave device (and what the response will look like) depend on the I²C device itself. You will have to refer to the manufacturer's data sheet for any given device to which you wish to interface.

For examples of using the new SNAPpy I²C functions to interface to external devices, look at the following scripts that are bundled with Portal:

- i2cTests.py – This is the overall I^{2}C demo script
- synapse.M41T81.py – Example of interfacing to a clock/calendar chip
- synapse.M41T00CAP.py – Example of interfacing to a real-time clock chip
- synapse.CAT24C128.py – Example of interfacing to an external EEPROM

Script i2cTests.py imports two of the other scripts, and exercises some of the functions within them.

Interfacing to multi-drop RS-485 devices

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 nodes 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-----------------------------
```

As of version 2.2, 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 (meaning) of the CTS pin is controlled by the SNAPpy built-in function flowControl(). Refer to the description of that function in the “SNAPpy – The API” section of the **SNAP Reference Manual.**

Encryption between SNAP nodes

Communications between SNAP nodes are normally unencrypted. Using the SNAP Sniffer (or some other means of monitoring radio traffic) you can clearly see the traffic passed between nodes. This can be very useful when establishing or troubleshooting a network, but 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 nodes offer two forms of encryption. If you have a compatible firmware version loaded into your nodes, 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 node by

checking the Node Info pane for the node in Portal. Firmware that supports AES-128 encryption will include "AES-128" in the firmware name.

Nodes that support AES-128 encryption are not available in all jurisdictions. Also, the Si100x platform does not have an AES-128 build available, due to space constraints. (The RF300/RF301 builds, based on the Si100x platform, have external memory available and therefore do have AES-128 builds available.) Users who would like some protection for their data but do not have AES-128 encryption available can use Basic encryption instead. Basic encryption is not strong encryption, and should not be relied on for high-security applications, but it does provide a level of protection to keep your data away from curious onlookers. Basic encryption is available in all SNAP nodes running firmware version 2.4 or newer.

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 node (both over the air and over the UARTs) is encrypted, and the node will expect all incoming communications to be encrypted. It will no longer be able to participate in unencrypted networks.

NV parameter #50 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

NV parameter #51 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., \x2a\x14\x3b\x44\xd7\x3c\x70\xd2\x61\x96\x71\x91\xf5\x8f\x69\xb9) or as some combination of the two (e.g. \xfbOF\x06\xe4\xf0Forty-Two!). Standard security practices suggest you should use a complicated encryption key that would be difficult to guess.

No encryption will be used if:

- NV parameter #50 is set to a value other than 1 or 2.
- NV parameter #50 is set to 1 in a node that does not have AES-128 encryption available in its firmware.
- The encryption key in NV parameter #51 is invalid.

When you are establishing encryption for a network of nodes, it is important that you work "from the outside in." In other words, begin by setting up encryption in the nodes farthest from Portal and work your way in toward your nearer nodes. This is necessary because once you have configured a node for encrypted communications, it is unable to communicate with an unencrypted node.

Consider a network where you have Portal talking through a bridge node (named Alice), and Alice is communicating with nodes Betty, Carla, and Daphne. If you configure Alice for encryption before you configure Betty, Carla and Daphne, Alice will no longer be able to reach the other three nodes to set up encryption in them. They will still be able to communicate with each other, but will not be able to talk to (or through) Alice (and therefore will not be able to report back to Portal).

In the same environment, if you are relying on multiple hops in order to get messages to all your nodes, if any intermediate node is encrypted all the unencrypted nodes beyond that one are essentially orphaned. If Daphne relays messages through Carla, and Carla relays messages through Betty to Alice (and thus to Portal), and you configure Carla for encryption before you configure Daphne, you will not be able to reach Daphne to set the encryption type or encryption key.

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

If you lose contact with a node as a result of a mistyped or forgotten encryption key, you will have to use Portal to reset the node back to factory parameters. This will set NV Parameter #50 back to 0 and NV Parameter #51 back to "" to disable encryption. Simply making a serial connection to the node to reset the encryption key will not be sufficient, as serial communications as well as over-the-air communications are encrypted.

Recovering an Unresponsive Node

As with any programming language, there are going to be ways you can put your nodes into a state where they do not respond. Setting a node 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 nodes unresponsive.

How to best recover an unresponsive node depends on the cause of the problem. If the problem is the result of runaway code (sleeping, looping, disabling UARTS, or turning off the radio) and "Erase SNAPpy Image" from the Node Info pane will not work because it requires communication with the node. In these cases, you can usually use Portal's "Erase SNAPpy Image..." feature from the Options menu to regain access to your node. This Portal feature interrupts the node before the script has a chance to start running – before it can put the node to sleep, fall into its stuck loop, or otherwise make the node responsive.

In the case of a lost encryption key or an unknown channel/network ID, or one of many other combinations that may arise, Portal's "Factory Default NV Params..." feature, also from the Options menu, resets the node back to the state it was in when delivered. (If you have intentionally changed any parameters, such as node name, channel, network ID, various timeouts, etc., you will need to reset them once you have access to the node.) Reconfiguring your node after resetting the default NV parameters can be more involved than simply correcting a script and reloading it, so if you are not sure why your node is unresponsive it may be best to try clearing its SNAPpy image first.

If these fail to recover access to your node, the "big hammer" approach is to reload the node's firmware, which you can also do from Portal's Options menu. Note that if the NV parameters are mis-set, reloading the firmware will not recover access to the node as it does not explicitly reset the parameters in the process. All three of these options require that you have a direct serial connection to the node.

For more on the use of these functions, refer to the **Portal Reference Manual**.

SNAPpy Scripting Hints

The following are some helpful hints (sometimes learned from painful lessons) for developing custom SNAPpy scripts for your nodes. These are not in any particular order.

Beware of Case SensitiViTy

Like “desktop” Python, SNAPpy scripts are case sensitive – “foo” is not the same as “Foo”.

Also, because SNAPpy is a dynamically typed language, it is perfectly legal to invent a new variable on-the-fly, and assign a value to it. So, the following SNAPpy code snippet:

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

…results in two variables being created, and “foo” still has the original value of 2.

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

```
linkQuality = getlq()
```

…is a script error unless you have defined a function getlq(). The built-in function is not named “getlq”.

```
linkQuality = getLq()
```

…is probably what you want.

Beware of Accidental Local Variables

All SNAPpy functions can read global variables, but (as with Python) you need to use the “global” keyword in your functions if you want to write to them.

```
count = 4
def bumpCount():
    count = count + 1
```

…is not going to do what you want (the global count will still equal 4). Instead, write something like:

```
count = 4
def bumpCount():
    global count
    count = count + 1
```

Don’t Cut Yourself Off (Packet Serial)

Portal talks to its “bridge” (directly connected) node using a packet serial protocol.

SNAPpy scripts can change both the UART and Packet Serial settings.

This means you can be talking to a node from Portal, and then upload a script into that node 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). Portal will no longer be able to communicate with that node serially.

Remember: Serial output Takes Time

A script that does:

```
print "imagine a very long and important message here"
sleep(mode, duration)
```

…is not likely to be allowing enough time for the text to make it all the way out of the node (particularly at slower baud rates) before the sleep() command shuts the node off.

One possible solution would be to invoke the sleep() function from the timer hook. This example hooks into the HOOK_100MS event.

In the script startup code:

```
sleepCountDown = 0
```

In the code that used to do the "print + sleep"

```
global sleepCountDown
print "imagine a very long and important message here"
sleepCountDown = 500  # actual number of milliseconds TBD
```

In the handler for HOOK_100MS:

```
global sleepCountDown
if sleepCountDown != 0:
    if sleepCountDown <= 100:  # timebase is 100 ms
        sleepCountDown = 0
        sleep(mode, duration)
    else:
        sleepCountDown -= 100
```

Remember: SNAP Engines do not have a lot of RAM

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

If a script generates too much output at once, the excess text will be truncated.

Remember: 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.

Remember: SNAPpy Integers are Signed

SNAPpy integers are 16-bit numbers, and have a numeric range of -32768 to +32767. Adding 1 to 32767 gives you -32768.

Be careful that any intermediate math computations do not exceed this range, as the resulting overflow value will be incorrect.

Remember: SNAPpy Integers have a Sign Bit

Another side-effect of SNAPpy integers being signed – negative numbers shifted right are still negative (the sign bit is preserved).

You might expect 0x8000 >> 1 to be 0x4000 but really it is 0xC000. You can use a bitwise "and" to get the desired effect if you need it.

```
X = X >> 1
X = X & 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.

If the node having the errors is a remote one (you cannot see its script output), remember that you can invoke the "Intercept STDOUT" action from the Node Info tab for that node. The error messages will then appear in the Portal event log, depending on the preferences specified in Portal.

Don't Define Functions Twice

In SNAPpy (as in Python), defining a function that already exists counts as a re-definition of that function.

Other 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.

There is limited dynamic memory in SNAPpy

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

> **NOTE** – this is different from prior versions, which only had a single fixed buffer for each type of string operation.

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
```

Remember Portal Speaks Python Too

SNAPpy scripts are a very powerful tool, but the SNAPpy language is only a small modified subset of full-blown Python.

In some cases, you may be able to take advantage of Portal's more powerful capabilities by having SNAPpy scripts (running on remote nodes) invoke routines contained within Portal scripts. This applies not only to the scripting language differences, but also to the additional hardware a desktop platform adds.

As an example, even though a node has no graphics display, it can still generate a plot of link quality over time, by using a code snippet like the following:

```
rpc("\x00\x00\x01", "plotlq", localAddr(), getLq())
```

For this to do anything useful, Portal must also have loaded a script containing the following definition:

```
def plotlq(who, lq):
  logData(who, lq, 256)
```

The node will report the data, and Portal will plot the data on its Data Logger pane. It wouldn't take much additional code to instead save the data to a text file or a database, or even to include other GUI libraries for your own custom visualizations.

Remember you can invoke functions remotely

Writing modular code is always a good idea. As an added bonus, if you are able to break your overall script into multiple function definitions, you can remotely invoke the individual routines to assist with unit testing them. This can help in determining where any problem lies.

Be careful using multicast RPC invocation

Realize that if you multicast an RPC call to function “foo”, all nodes in that multicast group that have a foo() function will execute theirs, even if their foo() function does something different from what your target node’s foo() function is expected to do. To put it another way, give your SNAPpy functions distinct and meaningful names.

If all nodes hear the question at the same time, they will all answer at the same time

If you have more than a few nodes, you will need to coordinate their responses (using a SNAPpy script) if you poll them via a multicast RPC call.

SNAP includes a Collision Avoidance feature (controlled by NV parameter 18) that inserts some random delay (up to 20 ms) when responding to multicast requests to assist in overcoming this. You can also enable Carrier Sense (NV 16) and Collision Detection (NV 17) to help ensure you do not have too many nodes talking at the same time. But none of these will be as reliable as an application-level control of when your node responds to a request.

If you want to call a built-in function by name, the called node needs a script loaded, even if the script is empty

SNAP Nodes without scripts loaded only support function calls by number. The “name lookup table” that lets nodes support “call by name” is part of what gets sent with each SNAPpy Image.

When Portal invokes built-in functions for you (from the GUI), it automatically converts function names to function numbers. Standalone SNAP nodes don’t know how to do this conversion.

So, if you don’t have any real script to put into a node that you want to control from something besides Portal, upload an empty one.

8. Example SNAPpy Scripts

The fastest way to get familiar with the SNAPpy scripting language is to see it in use. Portal comes with several example scripts pre-installed in the snappyImages directory.

Here is a list of the scripts preinstalled with Portal 2.4, along with a short description of what each script does. Take a look at these to get ideas for your own custom scripts. You can also copy these scripts, and use them as starting points.

NOTE – some of these scripts are meant to be imported into other scripts. Also, some of these scripts are found in a "synapse" subdirectory inside the "snappyImages" directory.

NOTE - Some example scripts predate newer SNAP Engines (such as the SM220) and may not function correctly without user modification.

General Purpose Scripts

Script Name	What it does
BatteryMonitor.py	Demonstrates interfacing to an external voltage reference in order to determine battery power.
BuiltIn.py (synapse.BuiltIn.py)	Portal uses this script to provide doc strings and parameter assistance for all the built-in functions. Several features in Portal will not work if you edit, move, or remove this file.
buzzer.py	Generates a short beep when the button is pressed. Also provides a "buzzer service" to other nodes. The example script DarkDetector.py shows one example of using this script. (This requires connection of a piezo buzzer to your node, hardware included in the EK2100 kit.)
CommandLine.py	An example of implementing a command line on the UART that is normally available on SNAP Engine pins GPIO9 through GPIO12. Provides commands for LED, relay, and seven-segment display control on supported Synapse evaluation boards.
DarkDetector.py	Monitors a photocell via an analog input, and displays a "darkness level" value on the seven-segment display on supported Synapse evaluation boards. Also requests a short beep *from a node running the buzzer.py script* when a threshold value is crossed.
DarkroomTimer.py	Operates a dark room enlarger light under user control.
datamode.py	An example of using two nodes to replace a serial cable.
dataModeNV.py	A more sophisticated example of implementing a wireless UART.

Script Name	What it does
evalBase.py (synapse.evalBase.py)	An importable script that adds a library of helpful routines for use with the Synapse evaluation boards. Board detection, GPIO programming, and relay control are just a few examples.
EvalHeartBeat.py	Example of displaying multiple networking parameters about a node on a single seven-segment display.
gpsNmea.py	Example decoding of data from a serial GPS. (There is an application note available from the Synapse Wireless website that expands on this functionality.)
hardTime.py (synapse.hardTime.py)	Helper script useful for SNAPpy benchmarking. **NOTE –** as of version 2.4, this script just imports the appropriate "platform-specific" helper script.
hexSupport.py (synapse.hexSupport.py)	Helper script that can generate hexadecimal output.
i2cTests.py	Demonstrates interacting with I^2C devices.
ledCycling.py	An example of using PWM.py. Varies the brightness of the LED on the Demonstration Boards.
ledToggle.py	Simple example of toggling an LED based on a switch input.
LinkQualityRanger.py	Radio range testing helper.
McastCounter.py	Maintains and displays a two-digit count, incremented by button presses. Resets the count when the button is held down. Broadcasts "count changes" to any listening units, and also acts on "count changes" from other units.
NewPinWakeupTest.py	Demonstrates using the functions in pinWakeup.py.
nvparams.py (synapse.nvparams.py)	Provides named enumerations for referencing NV parameters.
pinWakeup.py (synapse.pinWakeup.py)	An importable script that adds "wake up on pin change" functionality. **NOTE –** as of version 2.2, this script mainly just imports the appropriate "platform-specific" helper script.
platforms.py (synapse.platforms.py)	Import this to automatically enable the import of needed "platform dependent" scripts. These scripts enable you to code based on SNAP Engine GPIO pin numbers rather than tracking pin outputs on different SNAP Engine platforms.

Script Name	What it does
protoFlasher.py	Just blinks some LEDs on the SN171 Proto Board.
protoSleepcaster.py	Like McastCounter.py but this script is only for the SN171 Proto Board. Additionally, it demonstrates putting the node to sleep between button presses.
PWM.py (synapse.PWM.py)	An importable script that adds support for Pulse Width Modulation (PWM) on pin GPIO 0 on platforms based on the MC9S08GB60A chip from Freescale.
servoControl.py	A second example of using PWM.py. Controls the position of a standard hobby servo motor.
sevenSegment.py	Script providing support for the seven-segment display on the Synapse SN163 Bridge demonstration board for platforms that do not include the setSegments() built-in function.
snapsys.py (synapse.snapsys.py)	Required by Portal, do not move, edit, or delete. Import this script to enable compile-time population of the platform and version variables.
spiTests.py	Demonstrates interacting with SPI devices.
switchboard.py (synapse.switchboard.py)	An importable script that defines some constants (for readability) for switchboard-related enumerations.
sysInfo.py	An importable script that defines some constants (for readability) for the getInfo() function's enumerations.
Throughput.py	Can be used to benchmark packet transfer between two units.

Scripts Specific to I^2C

Script Name	What it does
M41T81.py (synapse.M41T81.py)	Demonstrates interfacing to a Clock Calendar chip via I^2C
M41T00CAP.py (synapse.M41T00CAP.py)	Demonstrates interfacing to a Clock Calendar chip via I^2C
CAT24C128.py (synapse.CAT24C128.py)	Demonstrates interfacing to a serial EEPROM chip via I^2C

Script Name	What it does
LIS302DL.py (synapse.LIS302DL.py)	Demonstrates interfacing to an Accelerometer chip via I^2C

Scripts Specific to SPI

Script Name	What it does
LTC2412.py (synapse.LTC2412.py)	Demonstrates interfacing to an Analog to Digital Converter chip via SPI
AT25FS010.py (synapse.AT25FS010.py)	Demonstrates interfacing to a 128K FLASH Memory chip via SPI

Scripts specific to the EK2100 Kit

Refer to the EK2100 Users Guide for more information about these example scripts.

Script Name	What it does
HolidayBlink.py	Demonstration for the SN171 Proto Board in the EK2100 kit.
HolidayLightShow.py	Demonstration for the USB SN132 in the EK2100 kit.
ManyMeter.py	Another Proto Board demo from the EK2100 kit showing how the SNAP node can gather information and report it back to Portal for display, tracking, or processing.
TemperatureAlarm.py	Script for use on the SN171 Proto Board to demonstrate a temperature-sensing alarm system.
TemperatureAlarmBridge.py	Script for use by the bridge node in conjunction with the TemperatureAlarm.py script.

Platform-Specific Scripts

Scripts specific to the RF100 Platform

These scripts are meant to be run on RF100 SNAP Engines (formerly known as RF Engines).

Script Name	What it does
pinWakeupRF100.py (synapse.pinWakeupRF100.py)	Pin Wakeup functionality specifically for the RF100 Engine. (Imported automatically by pinWakeup.py)
RF100.py (synapse.RF100.py)	Platform specific defines and enumerations for RF100 Engines. (Imported automatically by platforms.py)

Script Name	What it does
rf100HardTime.py (synapse.rf100HardTime.py)	How to reference the clock on the RF100 SNAP Engines. (Imported automatically by hardTime.py)

Scripts specific to the RF200, RF220, SM200, and SM220 Platforms

These scripts (all located in the synapse subdirectory of the snappyImages directory) are meant to be run on SNAP Engines based on the ATMEL Atmega128RFA1 chip.

Script Name	What it does
pinWakeupATmega128RFA1.py	Pin Wakeup functionality specifically for nodes based on the Atmega128RFA1 chip (which includes the RF200, RF220, SM200, and SM220). (Imported automatically by pinWakeup.py)
RF200.py, RF220.py, SM200.py, SM220.py	Platform-specific definitions and enumerations for RF200, RF220, SM200, and SM220 Engines. (Imported automatically by platforms.py) Note that the RF2x0 files import constants such as “GPIO_3”, to accommodate the logical structure of the 24-pin through-hole SNAP Engines, while the SM2x0 files import constants such as “GPIO_C3”, to accommodate the 64-pad surface-mount SNAP Engines.
rf200HardTime.py	How to reference the clock on the RF200 SNAP Engines. (Imported automatically by hardTime.py)

Scripts specific to the RF266 Platform

None at this time. You might want to look at some of the ATmega128RFA1 and RF200-specific scripts, as they are built on the same architecture. Also, consider checking on www.opensnap.org.

Scripts specific to the RF300/RF301 Platform

These scripts (all located in the synapse subdirectory of the snappyImages directory) are meant to be run on RF300 and RF301 SNAP Engines (based on the Silicon Labs Si1000 chip).

Script Name	What it does
pinWakeupRF300.py	Pin Wakeup functionality specifically for the RF300 Engine. (Imported automatically by pinWakeup.py)
RF300.py	Platform-specific definitions and enumerations for RF300 Engines. (Imported automatically by platforms.py)
rf300HardTime.py	How to reference the clock on the RF300 SNAP Engines. (Imported automatically by hardTime.py)

Scripts specific to the Panasonic Platforms

These scripts are meant to be run on the corresponding Panasonic hardware platforms.

Script Name	What it does
PAN4555.py	Defines initialization routine to drive unavailable IO pins as low outputs or pull them as high inputs. These IO pins on the chip are unavailable on the module, but must be configured for efficient sleep.
PAN4555_ledCycling.py	Demonstrates extra PWMs on PAN4555
PAN4555_PWM.py	Controls the additional PWMs on a PAN4555
PAN4555_SE.py	Defines the GPIO pins on a PAN4555 SNAP Engine
pinWakeupPAN4555_SE.py	Configures the “wakeup” pins on a PAN4555 SNAP Engine. (Imported automatically by pinWakeup.py)
PAN4561_ledCycling.py	Demonstrates extra PWMs on PAN4561
PAN4561_PWM.py	Controls the additional PWMs on a PAN4561
PAN4561_SE.py	Defines the GPIO pins on a PAN4561 SNAP Engine
pinWakeupPAN4561_SE.py	Configures the “wakeup” pins on a PAN4561 SNAP Engine. (Imported automatically by pinWakeup.py)

Scripts specific to the California Eastern Labs Platforms

These scripts are meant to be run on the corresponding CEL hardware platform.

Script Name	What it does
pinWakeupZIC2410.py	Configures the “wakeup” pins on a ZIC2410. (Imported automatically by pinWakeup.py)
ZIC2410_PWM.py	Support routines for accessing the two pulse-width modulation pins on a ZIC2410.
ZIC2410_SE.py	Platform-specific definitions and enumerations for SNAP Engines based on the ZICM2410 modules. (Imported automatically by platforms.py)
ZIC2410EVB3.py	Definitions for some of the hardware on the CEL EVB3 Evaluation Board
ZIC2410ledCycling.py	Demonstrates the PWMs on the ZIC2410

Script Name	What it does
ZIC2410spiTests.py	Demonstrates accessing the AT25FS010 chip built-in to the EVB1/2/3 Evaluation Boards, using SPI
ZicCycle.py	Blinks all the LEDs on the EVB3 board
ZicDoodle.py	Draws on the EVB1 LCD display, based on commands from another node running ZicDoodleCtrl.py. This script used functions deprecated as of release 2.4.
ZicDoodleCtrl.py	Uses the potentiometers on the EVB2 board to control an LCD display on an EVB1 board. This script used functions deprecated as of release 2.4.
ZicDoodlePad.py	Demonstrates the lcdPlot() built-in on a ZIC2410-LCD demonstration board. This script (along with that function) is deprecated.
zicHardTime.py	How to reference the clock on the nodes based on the ZIC2410 chips. (Imported automatically by hardTime.py)
ZicLinkQuality.py	A ZIC2410 counterpart to the original LinkQualityRanger.py
ZicMcastCtr.py	A ZIC2410 counterpart to the original MCastCounter.py script
ZicMonitor.py	Reads some ADCs on a CEL EVB1 Evaluation Board, plots the data in real-time on the EVB1 LCD display. This script used functions deprecated as of release 2.4.

Scripts specific to the ATMEL ATmega128RFA1 Platforms

These scripts are meant to be run on the corresponding ATMEL hardware platform. See also the scripts specific to the RF200, which is based on the ATmega128RFA1 chip.

Demonstrations written for the ATMEL STK600 board will also run on a Dresden “RCB” board, but then any references to “LED color” are wrong. (All the LEDs are red on the “RCB” board, compared to the red/yellow/green set on the STK600.)

Script Name	What it does
atFlasher.py	Demonstrates light flashing on a Dresden RCB test board with an ATmega128RFA1 node. This sample script is deprecated and may not be included in future releases.
atMcast.py	Demonstrates participation of an Atmega128RFA1 on a Dresden RCB test board in a group of nodes running McastCounter.py. This sample script is deprecated and may not be included in future releases.

Script Name	What it does
pinWakeupATmega128RFA1.py	Pin Wakeup functionality specifically for the ATmega128RFA1-based modules. (imported automatically by pinWakeup.py)
STK600.py	Defines and LED control routines for the STK600 board. This is imported by STK600demo.py.
STK600demo.py	Implements an up/down binary timer on the STK600 demo board. Push the button to reverse the direction.

Scripts specific to the SM700/MC13224 Platforms

These scripts are meant to be run on the Synapse SM700 surface-mount module, or the Freescale MC13224 chip on which it is based, or on a compatible board that uses one of the two.

Script Name	What it does
MC13224_PWM.py	Demonstrates Pulse Width Modulation on the TMR0/TMR1/TMR2 pins (GPIO8-10). For an example of using this script, see MC13224_ledCycling.py.
MC13224_ledCycling.py	Uses the PWM support routines in MC13224_PWM.py to vary the brightness of an LED attached to the TMR0(GPIO8) pin. By changing variable “TMR” within the script, the LED can be moved to either TMR1 (GPIO9) or TMR2 (GPIO10).
McastCounterSM700evb.py	The classic MCastCounter example, this one uses the LEDs and the SW1 push button of a CEL Freestar Pro Evaluation board (EVB)

Scripts specific to the STM32W108xB Platforms

These scripts are meant to be run on the DiZiC MB851 evaluation board, or on a compatible hardware design based on the underlying STM32W108CB and STM32W108HB chips.

Script Name	What it does
STM32W108xB_Example1.py	Simple example of how to blink LEDs and read a push button input from SNAPpy. The LED and button definitions assume the script is running on a DiZiC MB851 evaluation board.
STM32W108xB_GPIO.py	Some helper definitions and routines for working with the peripherals built into the ST Microelectronics STM32W108xB chips. For an example of using this script, see STM32W108xB_PWM.py

Script Name	What it does
STM32W108xB_PWM.py	An example of using the 8 PWM channels (2 sets of 4) available on this part. For an example of using this script, see STM32W108xB_ledCycling.py
STM32W108xB_ledCycling.py	Uses the PWM support routines in STM32W108xB_PWM.py to vary the brightness of an LED attached to the PB6 (IO14) pin. By changing the script, the PWM functionality can be demonstrated on the other 11 PWM capable pins
pinWakeupSTM32W108xB.py	Shows how to implement advanced hardware features from SNAPpy scripts, in this case how to access the "wake up" functionality of the chip
STM32W108xB_sleepTests.py	An example of using the "wake up" capabilities implemented in pinWakeupSTM32W108xB.py
STM32W108xB_HardTime.py	Demonstrates how to access a free-running hardware timer (for example, for benchmarking purposes)
LIS302DL.py	Demonstrates how to access the accelerometer readings from a STMicroelectronics LIS302DL chip
i2cTestsSTM32W108.py	Demonstrates how to access various I^2C devices, including the LIS302DL chip on the MB851 board
McastCounterMB851evb.py	The classic MCastCounter example, this one uses the two LEDs and the S1 push button of a DiZiC MB851 Evaluation board

Here is a second table listing some of the included scripts, this time organizing them by the techniques they demonstrate. This should make it easier to know which scripts to look at first.

Technique	Example scripts that demonstrate this technique
Importing evalBase.py and using the helper functions within it	CommandLine.py DarkDetector.py DarkRoomTimer.py EvalHeartBeat.py gpsNmea.py ledToggle.py LinkQualityRanger.py McastCounter.py protoSleepCaster.py

Technique	Example scripts that demonstrate this technique
Performing actions at startup, including using the @setHook() function to associate a user-defined function with the HOOK_STARTUP event	CommandLine.py DarkDetector.py DarkRoomTimer.py EvalHeartBeat.py gpsNmea.py ledToggle.py LinkQualityRanger.py McastCounter.py protoFlasher.py protoSleepCaster
Performing actions when a button is pressed, including the use of monitorPin() to enable the generation of HOOK_GPIN events, and the use of @setHook() to associate a user-defined routine with those events	buzzer.py DarkRoomTimer.py gpsNmea.py ledToggle.py McastCounter.py protoSleepCaster.py
Sending multicast commands	LinkQualityRanger.py McastCounter.py protoSleepCaster.py
Sending unicast commands	DarkDetector.py
Using global variables to maintain state between events	DarkDetector.py DarkRoomTimer.py EvalHeartBeat.py gpsNmea.py ledToggle.py LinkQualityRanger.py McastCounter.py protoFlasher.py protoSleepCaster.py
Controlling a GPIO pin using writePin(), etc.	buzzer.py evalBase.py gpsNmea.py ledToggle.py protoFlasher.py protoSleepCaster.py
Generating a short pulse using pulsePin()	buzzer.py

Technique	Example scripts that demonstrate this technique
Reading an analog input using readAdc(), including auto-ranging at run-time	DarkDetector.py
Performing periodic actions, including the use of @setHook() to associate a user defined routine with the HOOK_100MS event or other timed events	buzzer.py DarkDetector.py DarkRoomTimer.py EvalHeartBeat.py gpsNmea.py LinkQualityRanger.py McastCounter.py protoFlasher.py protoSleepCaster.py
Using the seven-segment display	DarkRoomTimer.py evalBase.py EvalHeartBeat.py LinkQualityRanger.py McastCounter.py sevenSegment.py
Deriving longer time intervals from the 100 millisecond event	buzzer.py DarkRoomTimer.py EvalHeartBeat.py McastCounter.py
Thresholding, including periodic sampling and changing the threshold at run-time	DarkDetector.py
Discovering another node with a needed capability	DarkDetector.py
Advertising a service to other wireless nodes	buzzer.py
Adding new capabilities by writing directly to processor registers (peek() and poke())	pinWakeup.py PWM.py [Platform]HardTime.py
Writing parameters to Non-volatile (NV) storage	evalBase.py DatamodeNV.py

Technique	Example scripts that demonstrate this technique
The use of “device types” as generic addresses, or to make a single script behave differently on different nodes	buzzer.py DarkDetector.py evalBase.py hardTime.py sevenSegment.py
Sleeping and waking up on a button press, importing and using pinWakeup.py	protoSleepCaster.py
Knowing when a RPC call has been sent out, by using HOOK_RPC_SENT.	protoSleepCaster.py
Parsing received serial data in a SNAPpy script (contrast with Transparent Mode)	CommandLine.py gpsNmea.py
Monitoring link quality using the getLq() function	LinkQualityRanger.py
Distributing a single application across multiple nodes	DarkDetector.py + buzzer.py TemperatureAlarm.py + TemperatureAlarmBridge.py
Displaying hexadecimal data on the seven-segment display	EvalHeartBeat.py McastCounter.py sevenSegment.py
Displaying custom characters on the seven-segment display	DarkRoomTimer.py EvalHeartBeat.py
Configuring Transparent Mode AKA Data Mode	datamode.py dataModeNV.py
Varying LED brightness using Pulse Width Modulation	ledCycling.py PAN4555_ledCycling.py PAN4561_ledCycling.py ZIC2410ledCycling.py MC13224_ledCycling.py MC13224_PWM.py STM32W108xB_LedCycling.py
Controlling a servo motor using Pulse Width Modulation	servoControl.py

Technique	Example scripts that demonstrate this technique
Writing a script so that it can run on multiple hardware platforms	NewPinWakeup.py + pinWakeup.py + pinWakeupRFEngine.py + pinWakeupPAN4555_SE.py + pinWakeupPAN4561_SE.py + pinWakeupZIC2410.py pinWakeupSTM32W108xB.py
Using external memory with a SNAP Engine	i2cTests.py + CAT24C128.py ZIC2410spiTests.py + AT25FS010.py
Higher resolution ADC	spiTests.py + LTC2412.py