REFERENCE GUIDE

SNAP® Network Operating System

Reference Manual for Version 2.4

Wireless Technology to Control and Monitor Anything from Anywhere™

500 Discovery Drive
Huntsville, Alabama 35806
877-982-7888

Doc# 600-0007K

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 communications, as well as implementing a Python virtual machine.

The subset of Python 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 even over the air), and dramatically alter the node’s capabilities and behavior.

Portal and SNAPconnect

Synapse *Portal* is a standalone software application which 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.

Synapse *SNAPconnect* is a standalone server application, which also runs on a standard PC. It connects to SNAP nodes over USB or RS-232 (just as Portal), but instead of providing a GUI, it acts as an XML-RPC server, allowing *your own* client applications to invoke functions on SNAP nodes, even over the Internet. These client applications can be written in Python, C++, C#, etc.

It is also possible for Portal to connect to your SNAP network through the SNAPconnect application (instead of a direct USB or RS-232 connection). This allows you to develop, configure, and deploy SNAP applications over the Internet.

Through an instance of the *SNAPconnect* software, you can have a total of 15 simultaneous client connects, which can be a mix of Portals and your own custom client applications.

The SNAP Wireless Sniffer

When you install Portal, a wireless “SNAP Sniffer” application is also installed. This program allows you to see SNAP messages over the air.

Navigating the SNAP Documentation

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

Start with 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 should start with one of these manuals, even if you are not starting with an EK2500 or EK2100 kit (Synapse SNAP nodes and even their component *SNAP Engines* are also sold separately, as well as bundled into evaluation kits).

About This Manual

This manual assumes you have read and understood 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 information about *SNAP and SNAPpy*. It covers topics like the SNAPpy language, and the built-in functions that are accessible from it. You will also find information about the different node configuration parameters that can be changed.

NOTE – In previous versions of this manual, information about the Portal GUI was also included. Starting with version 2.2, information specific to Portal has been moved to a separate **Portal Reference Manual**.

Previous versions of this manual referred to Synapse RF100 SNAP Engines as RFEngines. As SNAP has been ported to multiple platforms, the **SNAP Reference Manual** has been updated to better distinguish between the various platforms. See section 10 for details specific to each SNAP platform.

Other Important Documentation

Be sure to check out **all** of the SNAP documentation:

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

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

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

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

  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" (600026-01A)

  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" (600023-01A)

  There were enough changes between the 2.1 and 2.2 series of SNAP releases that we decided to provide an extra "transition" guide. You should check this document out if you were already a user of SNAP 2.1 and Portal 2.1.

- The "SNAP Firmware Release Notes"

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

- The "Portal Release Notes"

All of these documents are in Portable Document Format (PDF) files for download from the Synapse support forum (see below).

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.

You can download the latest SNAP, Portal, and SNAPconnect software from the forum. 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

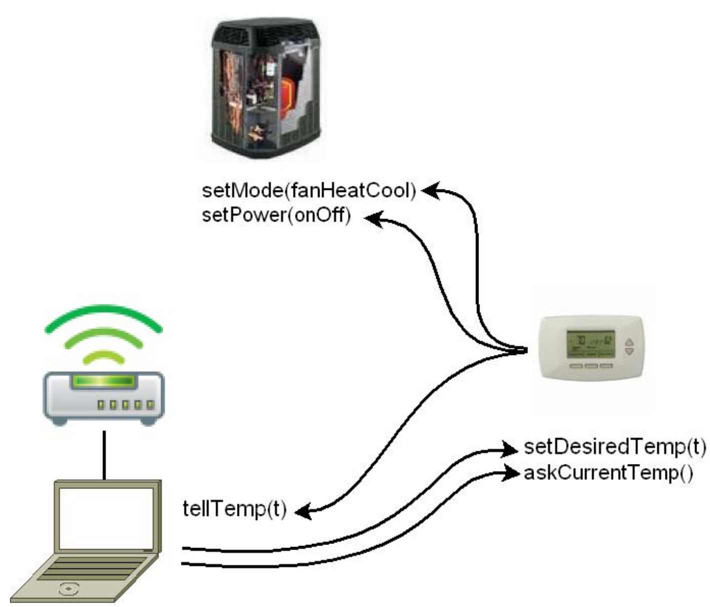
SNAP is a family of software technologies that together form an integrated, end-to-end solution for wireless monitoring and control. The latest version is 2.4, which this document covers.

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 (reliable) or multicast (unacknowledged).
- 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.

RPC

All SNAP devices implement a core set of built-in *functions* (procedures) to handle basic network configuration, system services, and device hardware control. These *functions* may be invoked remotely from Portal or from any other device on the SNAP network. Additional *user-defined functions* may be uploaded to devices as well. This upload process can be over directly connected serial interfaces, or over the air. Once uploaded, these functions are also callable locally or remotely, and may themselves invoke local and remote functions. Functions are defined in an embedded subset of the Python language, called SNAPpy.



Example HVAC System Showing RPC Call-flow (arrows)

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 which are processed by Portal and uploaded to SNAP devices. All SNAP devices are capable of running SNAPpy – it is the native language of RPC calls.

SNAPpy Examples

On installation, Portal creates a folder under "My Documents" called "Portal\snappyImages". Several sample script files are installed here by default. These scripts are plain text files, which may be opened and edited with Portal's built-in editor. External text editors 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.

Portal Scripting

Similar to the SNAP nodes, Portal can also be extended through scripting. By loading a script, you can add new functions to Portal, which you (and the other SNAP nodes) can call.

Python

Portal scripts are written in full Python (you are not limited to the embedded SNAPpy subset). 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.

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. Like other nodes, Portal has a Device Image (script) which defines the functions callable by incoming RPC messages. Since Portal runs on a PC, its script executes in a full Python environment with access to the many libraries, services, and capabilities available there.

> **SNAPpy RPC → Portal : Gateway to Full Python…**
> Thanks to this capability, it is quite simple for a low-power device on the network to (via an RPC call to Portal) send an email or update a database in response to some monitored event.

Portal Script Examples

On installation, Portal creates a folder under “My Documents” called “Portal”. Several sample script files are installed here by default. 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 read-only examples.

If you change the existing files to be writable, your changes to these examples will be overwritten when you install the next version of Portal.

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 significant

The amount of indentation is up to you (4 spaces is standard for Python) but *be consistent*.

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

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

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

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"
```

Looping is supported via "while"

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

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
$%^ = 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
e = (1, 2, 3) # Tuple
f = None # Python has a "None" data type
g = startup # Function
```

In the above example, invoking g() would be the same as directly calling startup().

String Variables can contain Binary Data

```
A = "\x00\xFF\xAA\x55" # The "\x" prefix means Hexadecimal
```

You define new functions using "def"

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

sayHello() # 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 always *provide two parameters*.

It is also important in your Portal and SNAPconnect related programming to make sure that any routines defined in Portal scripts (or SNAPconnect 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
```

Variables at the top of your script are global

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

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

Variables within functions are usually local…

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

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

…unless you explicitly say you mean the global one

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

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

def changeGlobal():
    global x # because of this statement…
    x = 99 # …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 conditionals 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 = 8 / 4
```

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 = 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 = 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 = "hello" # variable x is now a string (str)
x = True # variable x is now a Boolean (bool)
```

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.

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
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 back of this document.

Functions

Up to 255 "public" functions may be defined. These are remotely callable using the SNAP RPC protocol.

Non-public functions (prefixed with underscore) are limited only by the size of FLASH memory.

Data Types

SNAPpy supports the following fundamental Python data types:

- NoneType – None is a valid value
- int – An **int** is a signed 16-bit integer, -32768 through 32767. 32767 + 1 = -32768.
- bool – A bool has a value of either True or False.
- string – A static **string** has a maximum size of 255 bytes. If you assign a literal to a string when a function is created, this is the size limit that will be applied. 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 uses a different collection of string buffers, and the maximum length of the string will be determined by the platform on which SNAP is running. See the platform-specific parameters in Section 10 for more details. (**Note –** built-in functions slice/concat/rpc enforce smaller limits on what they can do with strings.)
- tuple – A **tuple** is a read-only container of other data types, e.g., (1,'A',True). SNAPpy tuples must be global in scope and cannot be used outside the scope of a local script. You cannot pass them as parameters in RPC calls to other nodes. You may nest tuples, however there are restrictions on the printing of nested tuples. (See the details about printing, below.)
- function – A **function** is a user-defined subroutine invoked from elsewhere in your script, or by RPC call from another node.

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

- 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.
- 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
- elif
- else
- from
- global
- if
- import
- is
- not
- or
- pass
- print
- return
- while

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

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

Operators

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

+	-	*	/	%		
<<	>>	&	|	^	~	
<	>	<=	>=	==	!=	<>

Slicing

Slicing is supported for **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”. String multiplication is not supported. You cannot use `3 * "Hello! "` to get "Hello! Hello! Hello! " in SNAPpy.

Subscripting

Subscripting is supported for **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
```

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.
- int – Given a string, returns an integer representation of the string. The 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.
- str – Given an element, returns a string representation of the element. The str(5) is '5' for example.

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:

('A',1,(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, concurrency is achieved through event-driven programming. This means that most SNAPpy functions run quickly to completion, and <u>never</u> "block" or "loop" waiting for something. External *events* will trigger SNAPpy functions.

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.[1]
		• 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`

[1] Note that the pin number refers to the numbering scheme relevant for the particular platform, and the number provided may not be the number that matches the pin placement on the SNAP Engine or module you are using. Refer to Section 10 for the hardware details for your particular platform for specifics. If you are working with a SNAP Engine (RF100, RF200, RF300, SM700, or ZICM2410-based engine), you can import the platforms file to provide mappings of platform-specific pins to GPIO numbers that match the footprint of the SNAP Engine.

Hook Name	When Invoked	Parameters	Sample Signature

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 internal clock. The same counter is used for all four timing hooks.	`@setHook(HOOK_1S)`
			`def doEverySec(tick):`
			`    pass`

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 occurs, 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 new 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 is used in the samples provided above.

The old way (before version 2.2) – snappyGen.setHook()

This method still works in the current version, but most people find the new way much easier to remember and use.

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**

```
snappyGen.setHook(SnapConstants.HOOK_XXX, eventHandlerXXX)
```

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

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 in any way (or even examined).

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 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:

0 = DS_NULL
1 = DS_UART0
2 = DS_UART1
3 = DS_TRANSPARENT
4 = DS_STDIO
5 = DS_ERROR
6 = DS_PACKET_SERIAL

The SNAPpy API for creating Switchboard connections is:

```
crossConnect(dataSrc1, dataSrc2) # Cross-connect SNAP data sources (bidirectional)
uniConnect(dst, src) # Connect src --> dst SNAP data sources (unidirectional)
```

For example, to configure UART1[2] 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 SNAPconnect, Portal, or another SNAP Node
UART1	Crossover	Loopback	Wireless Serial	Local Terminal	Local SNAPconnect, Portal, or another SNAP Node
TRANSPARENT	Wireless Serial	Wireless Serial	Loopback	Remote Terminal	Remote SNAPconnect

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) to intercept just errors, non-error text, or both when you intercept STDIO to the application.

[2] Most platforms have two UARTs available, so with most SNAP Engines UART0 will connect to the USB port on a SN163 board and UART1 will connect to the RS-232 port on any appropriate Synapse demonstration board. However the RF300 SNAP Engine has only one UART – UART0 – and it comes out where UART1 normally comes out (to the RS-232 port, via GPIO pins 7 through 10). If you are working with RF300 SNAP Engines, be sure to adjust your code to reference UART0 rather than UART1 for your RS-232 serial connections.

Loopback
A command like `crossConnect(DS_UART0, DS_UART0)` will 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 on 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 SNAPconnect.

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 Network IDs).

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 Nodes, 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 section 7 for more details.

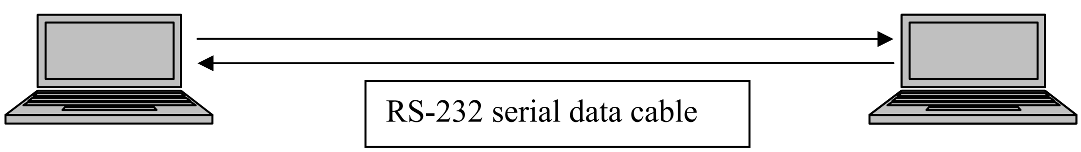
Debugging

Application development with SNAP offers an unprecedented level of interactivity in embedded programming. Using Portal you can quickly upload bits of 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

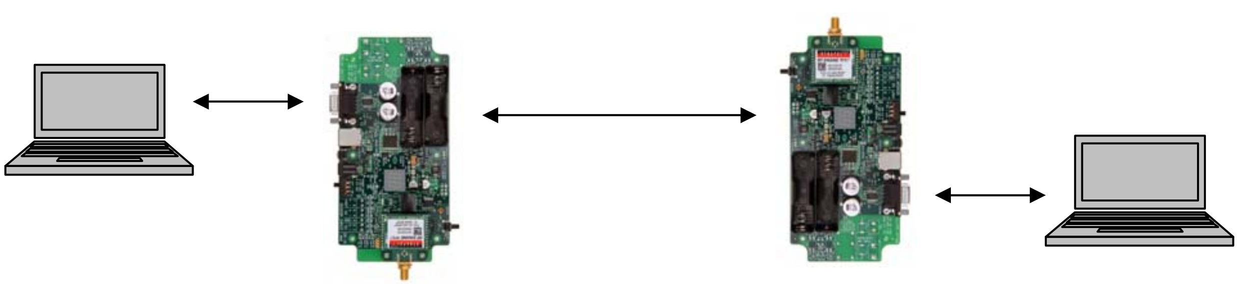
The following scenario is very common: two devices communicating over a RS-232 serial link.



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 its built-in RS-232 port, and communicates wirelessly with the other end point.



RS-232 **802.15.4 Wireless** **RS-232**

To summarize the requirements of this application:
We want to go from RS-232, to wireless, 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 below. 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.

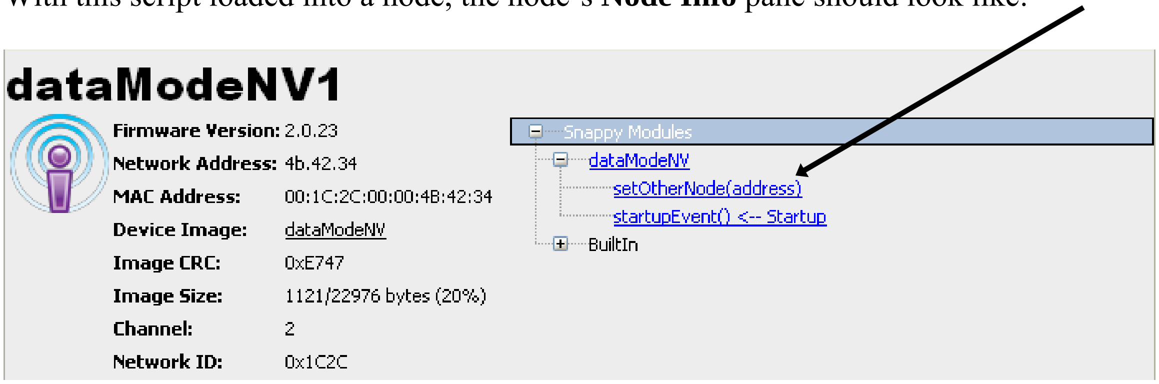
With these two edited scripts loaded into the correct nodes (remember, you are telling each node who the *other* node is, each node already knows its own address), you have just created a wireless serial link.

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.

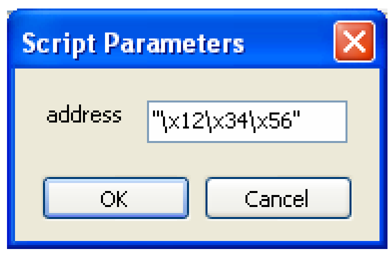
Look in your snappyImages directory for a script named dataModeNV.py. Since we won't be making any changes to this script, there is no need to make a copy of it. Simply load it into both nodes as-is.

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



Click on <u>setOtherNode(address)</u> 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.

On the following page 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 section 7 and section 8.

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") is 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.

6. 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 I2C 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 <u>fixed</u>. 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 this 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:

1. # select the desired CBUS device
2. writePin(somePin, False) # assuming the chip select is active-low
3. # read bytes from the selected CBUS device
4. Response = cbusRd(10) # <- you specify *how many* bytes to read
5. # deselect the CBUS device
6. 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 will be the topic of an upcoming application note.

NOTE – Not all SNAP Engines support CBUS.

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.

Like 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 (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 a function of 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 <u>fixed</u>. For example, on a Synapse RFE 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 to 3 or 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 this 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 you wish to interface to.

For examples of using the new 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 I^2C slave devices

Technically the correct name for this two-wire serial bus is Inter-IC bus or I^2C, 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 I^2C-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.

- i2cInit(*enablePullups*) – Prepare for I^2C operations (call this to setup for I^2C)
- i2cWrite(*byteStr*, *retries*, *ignoreFirstAck*) – Send data over I^2C to another device
- i2cRead(*byteStr*, *numToRead*, *retries*, *ignoreFirstAck*) – Read data from device
- getI2cResult() – used to check the result of the other functions

These routines are covered in more detail in section 7 of this document.

By using these routines, your SNAPpy script can operate as an I^2C **bus master**, and can interact with I^2C slave devices.

When performing I^2C interactions, fixed IO pin assignments are used. For example, on an RF100 the following IO pins are used:

- GPIO 17 is always used as the I^2C SDA (data) line
- GPIO 18 is always used as the I^2C SCL (clock) line

Note! – These pins are only dedicated if you are actually using the I^2C functions. If not, they remain available for other functions.

Refer to the platform-specific section for your hardware (located at the back of this manual) for the pin assignments for your platform.

Unlike CBUS and SPI, I^2C does not use separate "chip select" lines. The initial data bytes of each I^2C transaction specify an "I^2C address." Only the addressed device will respond. So, no additional GPIO pins are needed.

The specifics of *which* bytes to send to a given I^2C slave device (and what the response will look like) depend on the I^2C 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 I^2C functions to interface to external devices, look at the following scripts that are bundled with Portal:

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

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

Interfacing to multi-drop RS-485 devices

Many of the SNAP Demonstration Boards include an RS-232 serial port. The board provides the actual connector (typically 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).[3]

[3] Note that some naming conventions may expect the RTS pin to be considered the output pin, where the TXENA signal would occur. SNAP has standardized on the naming convention used by our earliest hardware platform, which has CTS as an output and RTS as an input.

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 this document.

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[4]
- 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.

---

[4] SNAP nodes running a firmware version before 2.4 may have False and True instead of 0 and 1, respectively, in this parameter. These boolean values are compatible with the numeric values used beginning with release 2.4. Basic encryption was not available before release 2.4.

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 network 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 NV parameters, the changes you make will only take effect after the node is rebooted.

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, or disabling UARTS) and "Erase SNAPpy Image" from the Node Info pane doesn't work, you can usually use Portal's "Erase SNAPpy Image..." feature from the Options menu to regain access to your node. 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.) 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. All three of these options require that you have a serial connection to the node.

7. SNAPpy – The API

This section details the "built-in" functions available to all SNAPpy scripts, as well as through RPC messaging. As of version 2.2, there are over 70 of these functions implemented by the SNAP "core" firmware.

These functions will first be presented in detail alphabetically. They will then be summarized, by category. Finally they will be categorized as immediate, blocking, or non-blocking.

Alphabetical SNAP API

bist() – Synapse internal use only

This function is for Synapse developer use only, and will likely be removed in a future release. User scripts should not bother calling this function.

call(*rawOpcodes, functionArgs…*) – Call embedded C code

This function is for advanced users only, and is outside the scope of this manual.

There is a separate Application Note that covers how to use this advanced feature.

Parameter *rawOpcodes* is a string containing actual machine code that implements the function.

The remaining *functionArgs* parameters depend on the actual function implemented by *rawOpcodes*.

callback(*callback, remoteFunction, remoteFunctionArgs…*)

Using the built-in function rpc() it is easy to invoke functions on another node. However, to get data back from that node, you either need to put a script in that node, or use the new callback() function.

Parameter *callback* specifies a function to invoke on the originating node *with the return value of the remote function*. For example, imagine having a function like the following in SNAP Node "A."

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

Invoking `callback('showResult', ...)` on Node B will cause function showResult() to get called on Node A with the live data from remote Node B.

Parameter *remoteFunction* specifies which function to invoke on the remote node, for example `readAdc`.

If the remote function takes any parameters, then the *remoteFunctionArgs* parameter of the callback() function is where you put them.

For example, node A could invoke the following on node B:

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

Node "B" would invoke readAdc(0), and then remotely invoke showResult(*the-actual-ADC-reading-goes-here*) on node A.

The callback() function is most commonly used with the rpc() function. For example:

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

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

This function normally returns True. It returns False only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory).

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

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

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

Parameter *nodeAddress* specifies the SNAP Address of the target node that is to receive the final (result) function call.

Parameter *callback* specifies a function to invoke on the target node *with the return value of the remote function*. For example, imagine having a function like the following in SNAP Node C.

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

Invoking `callout(nodeC, 'showResult', ...)` will cause function showResult() to get called with the live data from the remote node.

Parameter *remoteFunction* specifies a function to invoke on the remote node, for example readAdc.

If the remote function takes any parameters, then the *remoteFunctionArgs* parameter of the callout() function is where you put them.

For example, node A could invoke the following on node B, which would automatically invoke node C:

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

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

The callout() function is most commonly used with the rpc() function. For example:

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

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

This function normally returns True. It returns False only if it was unable to attempt the Remote Procedure Call (for example, if the node is low on memory).

cbusRd(*numToRead*) – Read bytes in from the CBUS

This function returns a string of bytes read in from the currently selected CBUS device. Parameter *numToRead* specifies how many bytes to read.

For more details on interfacing SNAP Nodes to external CBUS slave devices, refer to section 6.

cbusWr(*str*) – Write bytes out to the CBUS

This function writes the string of bytes specified by parameter *str* out to the currently selected CBUS slave device.

This function returns None.

For more details on interfacing SNAP Nodes to external CBUS slave devices, refer to section 6.

chr(*number*) – Generate a single-character-string

Returns a single-character string based on the number given. For example, chr(0x41) returns the string 'A'.

crossConnect(*endpoint1*, *endpoint2*) – Tie two endpoints together

The SNAPpy switchboard is covered in section 5. Refer to included script “switchboard.py" to see the possible values for *endpoint1* and *endpoint2*.[5]

See also function uniConnect() if what you really want is a one-sided data path.

---

[5] Most platforms have two UARTs available, so with most SNAP Engines UART0 will connect to the USB port on a SN163 board and UART1 will connect to the RS-232 port on any appropriate Synapse demonstration board. However the RF300 SNAP Engine has only one UART – UART0 – and it comes out where UART1 normally comes out (to the RS-232 port, via GPIO pins 7 through 10). If you are working with RF300 SNAP Engines, be sure to adjust your code to reference UART0 rather than UART1 for your RS-232 serial connections.

This function returns None.

eraseImage() – Erase any SNAPpy image from the node

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

This function takes no parameters, and returns None.

errno() – Read and reset latest error code

This function reads the most recent error code from the SNAPpy Virtual Machine (VM), clearing it out as it does so. The possible error codes are:

0 = NO_ERROR
1 = OP_NOT_DEFINED
2 = UNSUPPORTED_OPCODE
3 = UNRESOLVED_DEPENDENCY
4 = INCOMPATIBLE_TYPES
5 = TARGET_NOT_CALLABLE
6 = UNBOUND_LOCAL
7 = BAD_GLOBAL_INDEX
8 = EXCEEDED_MAX_BLOCK_STACK
9 = EXCEEDED_MAX_FRAME_STACK
10 = EXCEEDED_MAX_OBJ_STACK
11 = INVALID_FUNC_ARGS
12 = UNSUBSCRIPTABLE_OBJECT
13 = INVALID_SUBSCRIPT
14 = EXCEEDED_MAX_LOCAL_STACK
15 = BAD_CONST_INDEX
16 = ALLOC_REF_UNDERFLOW
17 = ALLOC_REF_OVERFLOW
18 = ALLOC_FAIL
19 = UNSUPPORTED_TYPE
20 = MAX_PACKET_SIZE_EXCEEDED
21 = MAX_STRING_SIZE_EXCEEDED

"Debug" firmware (look for "debug" in the file name) is required for complete error checking.

Some of these error codes are unlikely to occur from user-generated scripts, but a few would point directly to programming errors in the user's SNAPpy source code. For example:

INCOMPATIBLE_TYPES: Are you trying to add a number to a string?
TARGET_NOT_CALLABLE: Are you trying to invoke foo(), but foo = 123?

UNBOUND_LOCAL: Are you trying to *use* a variable before you put something in it?
INVALID_FUNC_ARGS: Are you passing the wrong *type* of parameters to a function?
Are you passing the wrong *quantity* of parameters?
INVALID_SUBSCRIPT: Are you trying to access str[3] when str = 'ABC'?
(When str = 'ABC', str[0] is 'A', str[1] is 'B', and str[2] is 'C')
EXCEEDED_MAX_LOCAL_STACK: Do you have too many local variables?
ALLOC_FAIL: Are you trying to keep too many string results? As of version 2.2 you are no longer limited to a *single* buffer for each type of string operation, but they are still limited in number.
MAX_PACKET_SIZE_EXCEEDED: Are you passing too large of a string value?
MAX_STRING_SIZE_EXCEEDED: Have you created a dynamic string too large for your platform?

Refer to the section for your specific platform (at the back of this document) for detailed buffer limits.

flowControl(*uart, isEnabled, isTxEnable*) – Enable/disable flow control

The flowControl() function allows you to disable or enable UART hardware handshaking.

Parameter *uart* is an integer that specifies which UART (0 or 1).
Parameter *isEnabled* is a boolean which turns hardware flow control on or off.
Parameter *isTxEnable* is an optional parameter that only matters if parameter *isEnabled* is True. If *isEnabled* is False, then *isTxEnable* has no effect.

When flow control is enabled for UART0, GPIO pins 5 and 6 become CTS and RTS pins for that UART. When flow control is enabled for UART1, GPIO pins 9 and 10 become CTS and RTS pins for that UART.

When flow control is ON, the RFE **monitors** the RTS pin from the attached serial device. As long as the RFE sees the RTS pin low, the RFE will continue sending characters to the attached serial device (assuming it has any characters to send). If the RFE sees the RTS pin go high, then it will stop sending characters to the attached serial device.

When flow control is OFF (*isEnabled* = False), the RTS and CTS pins are ignored.

The benefit of turning flow control off is that it frees up two more pins (per UART) for use as other I/O. The drawback of turning off flow control is that characters can be dropped.

Once enabled via the *isEnabled* parameter, the actual behavior of the CTS pin depends on the *isTxEnable* parameter. When *isTxEnable* is False, CTS acts as Clear To Send (described below). When *isTxEnable* is True, CTS instead acts as a TXENA pin.

NOTE – To maintain compatibility with scripts from version 2.1, the *isTxEnable* parameter does not have to be explicitly specified. If you leave it off, SNAP acts as if you explicitly specified False.

When flow control is ON (*isEnabled* = True) and *isTxEnable* is False, the SNAP Engine controls the CTS pin to indicate if it can accept more data. The CTS pin is low if the RFE can accept more characters. The CTS pin goes high (temporarily) if the RFE is "full" and cannot accept any more characters (you can keep sending characters, but they will likely be dropped).

When flow control is ON (*isEnabled* = True) and *isTxEnable* is also True, the CTS pin functions as a TXENA signal. The CTS pin is normally high. It transitions low before any characters are transmitted, and remains low until they have been completely sent. Only then does the CTS pin transition back high.

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

This function returns no value.

getChannel() – Get which channel the node is on

The getChannel() function returns a number representing which SNAP channel the node is currently on. For SNAP devices operating in the 2.4 GHz range, the number will be in the 0-15 range. For 900 MHz devices running the FHSS (frequency-hopping) firmware, the number *should* be in the 0-15 range, but could be in the 0-65 range. (See the getEnergy() function for more details about frequency distribution under FHSS firmware.)

For nodes operating in the 2.4 GHz range, SNAP channel 0 corresponds to 802.15.4 channel 11, 1 to 12, and so on.

SNAP Channel	802.15.4 Channel
0	11
1	12
2	13
3	14

SNAP Channel	802.15.4 Channel
4	15
5	16
6	17
7	18

SNAP Channel	802.15.4 Channel
8	19
9	20
10	21
11	22

SNAP Channel	802.15.4 Channel
12	23
13	24
14	25
15	26

For nodes operating in the 900 MHz range with FHSS (frequency-hopping) firmware, getChannel() returns an indication of which range of frequencies is in use by the node.

Radios with FHSS firmware "hop" between a minimum of 25 frequencies to avoid saturating any one frequency with too much transmission energy. Usable frequencies begin at 902 MHz and continue in 0.4 MHz increments through 928 MHz. SNAP selects 25 consecutive frequencies based on the "channel" specified by the user, skipping the first and last frequencies in the overall band. When the selected range would span past the last available frequency, SNAP wraps around to the beginning of the frequency range, so that if you have interference within part of the 900 MHz band in your environment you can select a channel that provides frequencies that avoid the interference.

The following table shows the frequency range in use by each SNAP channel with FHSS firmware.

SNAP Channel	900 MHz Chan. Range	Frequency Range (MHz)
0	1 – 25	902.4 – 912.0
1	5 – 29	904.0 – 913.6
2	9 – 33	905.6 – 915.2
3	13 – 37	907.2 – 917.0
4	17 – 41	908.8 – 918.4
5	21 – 45	910.4 – 920.0
6	25 – 49	912.0 – 921.6
7	29 – 53	913.6 – 923.2

SNAP Channel	900 MHz Chan. Range	Frequency Range (MHz)
8	33 – 57	915.2 – 924.8
9	37 – 61	916.8 – 926.4
10	41 – 64, 1	918.4 – 927.6, 902.4
11	45 – 64, 1 – 5	920.0 – 927.6, 902.4 – 904.0
12	49 – 64, 1 – 9	921.6 – 927.6, 902.4 – 905.6
13	53 – 64, 1 – 13	923.2 – 927.6, 902.4 – 907.2
14	57 – 64, 1 – 17	924.8 – 927.6, 902.4 – 908.8
15	61 – 64, 1 – 21	926.4 – 927.6, 902.4 – 910.4

For nodes operating in the 868 MHz range, the radio uses the same three frequencies (868.1 MHz, 868.3 MHz, and 868.5 MHz) for all communications, regardless of the channel specified. Radios on different channels cannot communicate with each other, but can interfere with each other.

This function takes no parameters.

getEnergy() – Get energy reading from current channel

The getEnergy() function returns the result of a brief radio Energy Detection scan on the currently selected channel. The result is in the same units as the getLq() function.

Using getEnergy() on radios working with frequency-hopping firmware is more cumbersome. This includes 900 MHz radios running FHSS firmware and 868 MHz radios.

Because each SNAP channel in sub-GHz firmware comprises a number of discrete frequencies, you must explicitly query the energy level at each frequency in use for your SNAP channel. For 900 MHz radios, this means you must step through each of the 25 900 MHz-band “channels” in your range, making a separate getEnergy() call for each. (See the explanation of 900 MHz FHSS frequency hopping in the getChannel() function description for more details.) For 868 MHz radios, you must check each of the three 868 MHz-band frequencies in use (channel 0, channel 1, and channel 2) and average the three values. (Regardless of which SNAP channel is specified, 868 MHz radios always use the same three frequencies for their communications.)

For purposes of getEnergy(), then, the range of valid values for the setChannel() function extends from 1 to 64, rather than the normal range you would use for setting a broadcast or receive channel. You should not use channel numbers outside the normal range of 0 to 15 except when attempting an energy scan. After performing an energy reading, be sure to set the channel back to the appropriate SNAP network channel to be able to communicate with your other nodes.

If you need an energy reading for a SNAP channel on a platform with 900 MHz frequency-hopping firmware, you could use a function like this to retrieve an average value of the appropriate frequencies:

```
def GetEnergy():
    if getInfo(2) == 6 and getInfo(1) == 5:
        # RF300, or compatible hardware, running FHSS
        incomingChannel = getChannel()
        energyLevel = 0
        channelLoop = 0
        while channelLoop < 25:
            setChannel((incomingChannel * 4 + channelLoop) % 64 + 1)
            energyLevel += getEnergy()
            channelLoop += 1
        setChannel(incomingChannel)
        return energyLevel / 25
    elif getInfo(2) == 6 and getInfo(1) == 3:
        # RF301, or compatible hardware, operating at 868 MHz
        incomingChannel = getChannel
        energyLevel = 0
        channelLoop = 0
        while channelLoop < 3:
            setChannel(channelLoop)
            energyLevel += getEnergy()
            channelLoop += 1
        setChannel(incomingChannel)
        return energyLevel / 3
    else:
        return getEnergy()
```

Alternately, you could use the scanEnergy() function and work with the appropriate three or 25 values returned by that function.

The getEnergy() function takes no parameters.

getI2cResult() – Get status code from most recent I^{2}C operation

This function takes no parameters. It returns the result of the most recently attempted I^{2}C operation. (It also resets the error code for next time). The possible return values and their meanings are:

0 = I2C_OFF means I^{2}C was never initialized (you need to call i2cInit()!)
1 = I2C_SUCCESS means the most recent I^{2}C read/write/etc. succeeded
2 = I2C_BUS_BUSY means the I^{2}C bus was in use by some other device
3 = I2C_BUS_LOST means some other device stole the I2C bus
4 = I2C_BUS_STUCK means there is some sort of hardware or configuration problem
5 = I2C_NO_ACK means the slave device did not respond properly

For more information on interfacing SNAP nodes to I^{2}C devices, refer to section 6.

getInfo(*whichInfo*) – Get specified system info

This function returns details about the platform and operating environment a script is running under.

Parameter *whichInfo* specifies the type of information to be retrieved:

0 = Vendor
1 = Radio
2 = CPU
3 = Platform/Broad Firmware Category
4 = Build
5 = Version (Major)
6 = Version (Minor)
7 = Version (Build)
8 = Encryption
9 = RPC Packet Buffer Reference
10 = Is Multicast
11 = Remaining TTL
12 = Remaining Tiny Strings
13 = Remaining Medium Strings
14 = Route Table Size
15 = Routes in Route Table

Based on the value of *whichInfo*, a value is returned. Many of the following "result codes" will be expanding in the future. Here are the currently established values:

getInfo(0) – Vendor
The vendor indicates the manufacturer of the radio module on which SNAP is running.

Possible Vendor result codes for getInfo(0):
0 = Synapse
1 = Reserved
2 = Freescale
3 = CEL
4 = ATMEL
5 = Silicon Labs
6 = Reserved
7 = PC
8 = Reserved
9 = STMicrosystems
(to be continued…)

getInfo(1) – Primary Communications Interface
The Network Interface indicates the means the node uses to connect to the rest of the network.

Possible Network Interface result codes for getInfo(1):

0 = 802.15.4
1 = None (Serial communications only)
2 = Reserved
3 = 868 MHz
4 = Powerline
5 = 900 MHz Frequency-Hopping

(other interfaces may be supported in the future)

getInfo(2) – CPU
The CPU indicates the processor paired with the radio in the SNAP module.

Possible CPU result codes for getInfo(2):

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

(other CPUs may be supported in the future)

getInfo(3) – Platform/Broad Firmware Category
The Platform indicates the model of module and firmware on which SNAP is running.

Possible Platform result codes for getInfo(3):

0 = Synapse RF100 SNAP Engine
1 = Reserved
2 = Reserved
3 = CEL ZIC2410
4 = Reserved
5 = MC1321x
6 = ATmega128RFA1
7 = SNAPcom
8 = Si100x
9 = MC1322x
10 = IT700
11 = Si100x KADEX
12 = Reserved
13 = Synapse RF300 SNAP Engine
14 = Synapse RF200 SNAP Engine
15 = Synapse SM300 Surface Mount Module
16 = Synapse SM301 Surface Mount Module
17 = Synapse SM200 Surface Mount Module
18 = Reserved
19 = Synapse RF266
20 = STM32W108xB

getInfo(4) – Build
The Build indicates whether the firmware in your SNAP module is a "debug" build or a "release" build.

Possible Build result codes for getInfo(4):

0 = "debug" build (more error checking, slower speed, less SNAPpy room)
1 = "release" build (less error checking, faster speed, more SNAPpy room)

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

getInfo(8) – Encryption
The Encryption setting specifies what type of encryption is available in the module. It does **not** indicate what encryption (if any) is enabled for the module.

Possible Encryption result codes for getInfo(8):

- 0 = None (no encryption support) **This is deprecated.** As of release 2.4, all nodes include support for the option of Basic encryption.
- 1 = AES-128
- 2 = Basic encryption

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

getInfo(10) – Is Multicast
Use the Is Multicast call to determine if the function running was invoked by a multicast call (meaning, other nodes on the network might also have heard the call and be running the function, too), or by a more direct means (such as a direct RPC call to the node invoking the function, or because the function was hooked to a timed event).

Possible Is Multicast result codes for getInfo(10):

- 0 = The RPC command currently being processed was received via an addressed RPC command or was triggered by a system hook.
- 1 = The RPC command currently being processed was received via a multicast rather than a direct RPC command.

getInfo(11) – Remaining TTL
The Remaining TTL value indicates how many “hops” a particular command had left before its end-of-life when it was processed by the node. You can use this information to tune your TTL values for your network to reduce broadcast chatter.

getInfo(12) – Remaining Tiny Strings
The Remaining Tiny Strings value indicates how many “tiny” string buffers remain unused in your node. The size and number of tiny strings available on your node will vary depending on the underlying hardware and firmware. See Section 10 for details specific to your platform.

getInfo(13) – Remaining Medium Strings
The Remaining Medium Strings value indicates how many “medium” string buffers remain unused in your node. The size and number of medium strings available on your node will vary depending on the underlying hardware and firmware. See Section 10 for details specific to your platform.

getInfo(14) – Route Table Size
The Route Table Size value indicates how many other nodes your node can keep track of in its address book. When a node needs to talk to another node, it must ask where that node is. It will find that it can talk to the node directly, that it must communicate through another node, or that it cannot find the node at all. In these first two cases, SNAPpy keeps track of the path used to contact the node in a route table so the next time it talks to the same node, it does not have to query how to find the node first. How long the path to a node is kept depends on the mesh routing timeouts defined in NV parameters Mesh Routing Maximum Timeout, Mesh Routing Minimum Timeout, Mesh Routing New Timeout, Mesh Routing Used Timeout, and Mesh Routing Delete Timeout (NV parameters 20 through 24).

getInfo(15) – Routes in Route Table
The Routes in Route Table value indicates how many of the routes in the node's route table are in use, meaning how many other nodes the current node knows how to access without having to first perform a route request. See the description of getInfo(14) for more details.

getLq() – Get the most recent Link Quality

The getLq() function returns a number 0-127 (theoretical) representing the link quality (received signal strength) of the most recently received packet, regardless of which node that packet came from. (It could be a near node, or it could be a far node.)

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

This function takes no parameters.

getMs() – Get system millisecond tick

The getMs() function returns the value of a free-running timer within the SNAP Engine. The value returned is in units of milliseconds. The timer is only 16 bits, and rolls back around to 0 every 65.535 seconds.

Because all SNAPpy integers are signed, the counter's full cycle is:
0, 1, 2,…,32766, 32767, -32768, -32767, -32766, …, -3, -2, -1, 0, 1,…

Some scripts use this function to measure elapsed (relative) times.

The value for this function is only updated *between* script invocations (events). If you do two back-to-back getMs() calls, you will get the same value.

This function takes no parameters.

getNetId() – Get the node's Network ID

The getNetId() function returns the 16-bit Network Identifier (ID) value the node is currently using. The node will only accept packets containing this ID, or a special "wildcard" value of 0xffff (the "wildcard" Network ID is used during the "find nodes" process).

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

See also, setNetId(). This function takes no parameters.

getStat() – Get Node Traffic Status

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

getStat(0) – Null Transmit Buffers
This provides the number of transmit buffers processed through a null serial port.

getStat(1) – UART0 Receive Buffers
This provides the number of received buffers processed through UART0.

getStat(2) – UART0 Transmit Buffers
This provides the number of transmit buffers processed through UART0.

getStat(3) – UART1 Receive Buffers
This provides the number of received buffers processed through UART1.

getStat(4) – UART1 Transmit Buffers
This provides the number of transmit buffers processed through UART1.

getStat(5) – Transparent Receive Buffers
This provides the number of received buffers processed through transparent serial mode.

getStat(6) – Transparent Transmit Buffers
This provides the number of transmit buffers processed through transparent serial mode.

getStat(7) – Packet Serial Receive Buffers
This provides the number of received buffers processed through packet serial mode.

getStat(8) – Packet Serial Transmit Buffers
This provides the number of transmit buffers processed through packet serial mode.

getStat(9) – Radio Receive Buffers
This provides the number of receive buffers processed through the radio.

getStat(10) – Radio Transmit Buffers
This provides the number of transmit buffers processed through the radio.

getStat(11) – Radio Forwarded Unicasts
This provides the number of "unicast" (directly addressed RPC) packets forwarded for nodes over the radio.

getStat(12) – Packet Serial Forwarded Unicasts
This provides the number of "unicast" (directly addressed RPC) packets forwarded for nodes over packet serial.

getStat(13) – Radio Forwarded Multicasts
This provides the number of multicast packets forwarded for nodes over the radio.

getStat(14) – Packet Serial Forwarded Multicasts
This provides the number of multicast packets forwarded for nodes over packet serial.

imageName() – Return name of currently loaded SNAPpy image

Prior to being downloaded into a SNAP node, the text form of a SNAPpy script gets compiled into a byte-code image. It is this executable image that gets downloaded into the node, not the original (textual) source code.

The generated image takes its base name from the underlying source script. For example, image "foo.spy" would be generated from a script named "foo.py".

Function imageName() returns the "base name" from the currently loaded image (if there is one). In the example given here, function imageName() would return the string "foo".

This function takes no parameters.

i2cInit(*enablePullups*) – Setup for I^2C

This function performs the necessary setup to allow subsequent i2cRead() and i2cWrite() calls to be made.

Parameter *enablePullups* causes <u>internal</u> pull-up resistors to be activated for the I^2C clock and data lines. These lines do require pull-ups, but often those pull-ups are part of your external hardware, and parameter *enablePullups* should be False. (Don't "double pull-up" the I^2C bus.)

Setting parameter *enablePullups* to True can come in handy when you don't have a real I^2C bus, but are doing quick prototyping by dangling I^2C devices directly off the SNAP Engine.

For more information about interfacing SNAP nodes to I^2C devices, refer to section 6.

i2cRead(*byteStr, numToRead, retries, ignoreFirstAck*) – I2C Read

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

I^2C devices must be addressed before data can be read out of them, so this function really does a write followed by a read.

Parameter *byteStr* specifies whatever “addressing” bytes must be sent to the device to get it to respond.

Parameter *numToRead* specifies how many bytes to read back from the external I^2C device.

Parameter *retries* can be used to give slow devices extra time to respond. Try an initial *retries* value of 1, and increase it if needed. This controls a spin-lock count.

Some devices do not send an initial “ack” response. For these devices, set parameter *ignoreFirstAck* to True. This will keep the lack of an initial acknowledgement from being counted as an I2C error.

This function returns the string of bytes read back from the external I^2C device.

For more information about interfacing SNAP nodes to I^2C devices, refer to section 6.

i2cWrite(*byteStr, retries, ignoreFirstAck*) – I^2C Write

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

Parameter *byteStr* specifies the data to be sent to the external I^2C device, including whatever “addressing” bytes must be sent to the device to get it to pay attention.

Parameter *retries* can be used to give slow devices extra time to respond. Try an initial *retries* value of 1, and increase it if needed.

Some devices do not send an initial “ack” response. For these devices, set parameter *ignoreFirstAck* to True. This will keep the lack of an initial acknowledgement from being counted as an I^2C error.

This function returns the number of bytes actually written.

For more information about interfacing SNAP nodes to I^2C devices, refer to section 6.

initUart(*uart, bps*) – Initialize a UART (short form)

This function programs the specified *uart* (0 or 1) to the specified bits per second (*bps*).

A *bps* value of 0 disables the UART.

A *bps* value of 1 selects 115,200 bps (This large number would not fit into a SNAPpy integer, and so was treated as a special case).

Usually you will set *bps* directly to the desired bits per second: 1200, 2400, 9600, etc.

NOTE – you are not limited to "standard" baud rates. If you need 1234 bps, do it.

Valid baud rates are platform-dependent. Refer to the back of this document.

This is the short form of the initUart() function. Data Bits defaults to 8, Parity defaults to None, and Stop Bits defaults to 1.

This function returns no value.

initUart(*uart, bps, dataBits, parity, stopBits*) – Initialize a UART

This is the long form of the initUart() function just described.

This function programs the specified *uart* (0 or 1) to the specified bits per second (*bps*).
In addition, this variant of the initUart() function also allows you to specify the *dataBits* (7 or 8), the *parity* ('E', 'O', or 'N' representing EVEN, ODD, or NO parity), and the number of stop bits.

This function returns no value.

initVm() – Initialize (restart) the SNAPpy Virtual Machine

This function takes no parameters, and returns no value.

Calling this function restarts the SNAPpy virtual machine. If a SNAPpy image is currently loaded in the node, the scripts "startup" handler will be invoked, and then normal SNAPpy script execution will begin (timer hooks, GPIO hooks, STDIN hooks, etc.)

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

This function does not return a value.

int(*obj*) – Convert an object to numeric form (if possible)

Converts the specified *obj* (usually a string) into numeric form. For example, int('123') = 123, int(True) = 1, and int(False) = 0.

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

lcdPlot() – LCD Support (Deprecated)

Currently this function only works on the CEL ZIC2410 firmware, on evaluation boards equipped with an LCD display. As CEL is no longer manufacturing the hardware, the function is deprecated and may not be supported in future releases. The lcdPlot() function performs different tasks, depending on the parameters passed to it:

Calling lcdPlot() with no parameters will trigger "LCD detection." If an LCD is present, it will be cleared, and the function will return True. If an LCD is not present, the function will return False.

Calling lcdPlot(x, y, isSet) will set (isSet=True) or clear (isSet=False) the pixel at the specified x, y coordinates. The pixel resolution of the LCD is 64 rows by 128 columns. Coordinate 0,0 is in the upper left hand corner.

Calling lcdPlot(str) will print the specified string at the current text cursor position.

Within the string, you can optionally include a "\x01" to turn inverse text on, and a "\x02" to turn inverse text off.

Calling lcdPlot(x, y, str) will print the specified string at the specified x (in pixels) and y (in text lines, **not** pixels).

Within the string, you can again turn inverse text on and off. In addition, you can specify a custom bitmap by specifying a binary string containing:

A literal \xF0 (this must be the very first character of the string)
A byte specifying the bitmaps width in pixels
A byte specifying the bitmaps height in pixels (must be a multiple of 8)
Bytes specifying the actual bitmap pixels, in column order.

For example, the following will draw a small triangle in the upper left hand corner of the display:

```
lcdPlot(0, 0, "\xF0\x08\x08\xFF\x81\x82\x84\x88\x90\xA0\xC0")
```

len(*sequence*) – Return the length of a sequence

This function returns the size of parameter *sequence*. Currently *sequence* must be a string or a tuple, but this may change in a future version of SNAPpy.

```
print len('123') # Returns 3
```

loadNvParam(*id*) – Read a Configuration Parameter from NV

This function reads a single parameter from the SNAP Node's NV storage, and returns it to the caller.

Parameter *id* specifies which parameter to read. For a full list of all the system (reserved) *id* values, refer to section 8. User parameters should have *id* values in the range 128-254.

See also function saveNvParam().

localAddr() – Get the node's SNAP address

The localAddr() function returns a string representation of the node's 3-byte address on the SNAP network.

This function takes no parameters.

mcastRpc(*group, ttl, function, args…*) – Multicast RPC

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

Parameter *group* specifies which nodes should respond to the request. By default, all nodes belong to the "broadcast" group, group 0x0001. You can configure your nodes to belong to different or additional groups, refer to NV parameter #5 and NV parameter #6 in section 8 on Node Configuration Parameters.

Parameter *ttl* specifies the Time To Live (TTL) for the request. This basically specifies how many hops the message is allowed to make before being discarded.

Parameter *function* specifies which remote function to be invoked. This could be a built-in SNAPpy function, or one defined by the SNAPpy script currently loaded into that node.

NOTE! Except for built-ins, *what* the function actually does depends on what script is loaded into each node.

The specified function will be invoked with the parameters specified by *args*, if any *args* are present.

For example, `mcastRpc(1, 3, 'writePin', 0, True)` will ask all nodes in the broadcast group and within 3 hops of the sender to do a writePin(0, True).

`mcastRpc(3, 5, 'reboot')` will ask all nodes within 5 hops and belonging to group 1 (0x0001 or 0000000000000001b) or group 2 (0x0002 or 0000000000000010b) to do a reboot().

Notice that groups are bits, not numbers. See the details on NV parameter 5 and NV parameter 6 in section 8 for more details.

`mcastRpc(6, 2, 'reboot')` will ask all nodes within 2 hops and belonging to group 2 (0x0002 or 0000000000000010b) or group 3 (0x0004 or 0000000000000100b) to do a reboot().

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

If this function returns True, it does not mean your RPC request was successfully received (SNAP multicast messages are unacknowledged).

If you need confirmation that the remote node executed your request, it needs to come from the remote node. Refer to the callback() function for one method of doing this.

mcastSerial(*destGroups, ttl*) – Setup TRANSPARENT MODE

SNAP TRANSPARENT MODE is covered in section 5.

When you want the outbound data to be sent to multiple nodes, use this function.

Parameter *destGroups* specifies the multicast groups that are eligible to receive this data. Parameter *ttl* specifies the maximum number of hops the data will be re-transmitted (in search of interested nodes).

Note that because the received serial characters will be sent using (unacknowledged) multicast messages, multicast TRANSPARENT MODE is less reliable than unicast TRANSPARENT MODE.

If you want the received serial characters to go to only one specific node, you should use function ucastSerial() instead.

This function does not return a value.

monitorPin(*pin, isMonitored*) – Enable/disable monitoring of a pin

This function can be used as an alternative to function readPin(), or in addition to it.

Parameter *pin* specifies which IO pin to monitor. Parameter *isMonitored* makes the pin be monitored when True, or ignored when False.

"Monitoring" in this context means "sampled periodically in the background," and only makes sense with pins previously configured as digital inputs via setPinDir().

You *could* monitor an output pin, but the changes you got notified about would be changes that you *caused* (via writePin()).

When monitored pins *change state*, a HOOK_GPIN event is sent to the SNAPpy virtual machine. If you have assigned a HOOK_GPIN handler (using the "set hook" capability described in section 5), then the previously specified handler function will be invoked with the number of the IO pin that changed state, and the pin's new value.

This function does not return a value.

See also function setRate(), which controls the *sampling rate* of the background pin monitoring that is *enabled/disabled* by this function.

ord(*str*) – Return the integer ASCII ordinal value of a character

Parameter *str* specifies a single-character string to be converted. For example, `ord('A')` = 65 (0x41), and `ord('2')` = 50 (0x32).

peek(*address*) or **peek(*addressHi, addressLow, word*)** – Read a memory location

The peek() function allows you to read a byte from a memory location. On 16-bit platforms, parameter *address* specifies which memory location to read (0-0xFFFF). On 32-bit platforms, the two address parameters are required to specify the target address.

On 16-bit platforms, this function returns an integer in the range 0-255. On 32-bit platforms, the size and character of the return value is determined by the word parameter:

- 0 = Return one byte (0x00 to 0xFF).
- 1 = Return one 16-bit integer (0x0000 to 0xFFFF).
- 2 = Return the high 16-bit integer of a 32-bit value, storing the low value for later retrieval.
- 3 = Return the low 16-bit integer of a 32-bit value, discarding the high value.
- 4 = Return the low 16-bit integer of a 32-bit value that has previously been stored using word = 2. You can use peek() with no parameters as a substitute for this. Performing this peek before storing a value using word = 2 provides an undefined result.

On 32-bit systems, you must have a properly aligned address when reading memory larger than one byte, i.e., when peeking a 16-bit value, your address must be even, and when peeking a 32-bit value, your address must be divisible by 4.

On platforms using the 8051 processor, special function registers (SFRs) are available in SNAPpy by specifying the peek address as -1 * ((SFR_page * 0x100) + SFR_address). For example, if you wanted to peek register 0x92 on page 0x0F, you would use:

```
Peek(-0x0F92)
```

If the register you wish to peek is in page 0, the leading byte can be omitted. To peek SFR register 0xF5 in SFR bank 0, you would use:

```
Peek (-0xF5)
```

or

```
Peek(-0x00F5)
```

This function is intended for advanced users only.

peekRadio(*address*) – Read an internal register of the radio

The peekRadio() function allows you to read any location within the radio's memory space (which on many SNAP Engines is separate from the processor's memory space). Parameter *address* specifies which memory location to read (0-65535).

This function returns an integer in the range 0-255.

This function is intended for advanced users only.

poke(*address, value*) or **poke(*addressHi, addressLow, word, data*)** or **poke(*addressHi, addressLow, word, dataHi, dataLow*)** – Write to a memory location

The poke() function allows you to write a byte to a memory location.

On 16-bit platforms, parameter *address* specifies which memory location to write to, and parameter *value* specifies the value (0x00 to 0xFF) to be written.

On 32-bit platforms, the addressLo and addressHigh parameters specify the target memory location. The word parameter specifies how large the data value is that will be written. For 8- or 16-bit data, a single data value is required. For 32-bit data, you must provide two data values, each 0x0000 to 0xFFFF:

- 0 = Poke a single byte (0x00 to 0xFF) into the target memory location.
- 1 = Poke a 16-bit value (0x0000 to 0xFFFF) into the target memory location.
- 2 = Poke a 32-bit value (0x00000000 to 0xFFFFFFFF) into the target memory location, with the data value broken into dataHi and dataLow, each with a range of 0x0000 to 0xFFFF.

On 32-bit systems, you must have a properly aligned address when writing memory larger than one byte, i.e., when poking a 16-bit value, your address must be even, and when poking a 32-bit value, your address must be divisible by 4.

This function does not return a value.

On platforms using the 8051 processor, special function registers (SFRs) are available in SNAPpy by specifying the poke address as -1 * ((SFR_page * 0x100) + SFR_address). For example, if you wanted to poke value 0x10 to register 0x92 on page 0x0F, you would use:

```
Poke(-0x0F92, 0x10)
```

This function is intended for advanced users only.

If you are not careful with this function, you could crash or even "unprogram" your device. Still, if you know what you are doing, peek() and poke() allow you to take advantage of additional hardware resources within the device itself (functions not supported by the "core" firmware).

See for example sample SNAPpy script PWM.py.

pokeRadio(*address, value*) – Write to an internal radio register

The pokeRadio() function allows you to write to any location within the radio's memory space.

On some SNAP Engines (including the Synapse RF100 SNAP Engine), the memory space for the 802.15.4 radio is not included within the memory space for the processor. The pokeRadio() function allows you to access the internal registers of the radio hardware, regardless of how the radio is physically accessed. (For example, on an RF100 SNAP Engine, the radio is actually connected to an internal SPI bus).

Parameter *address* specifies which radio register location to write to. Parameter *value* specifies the value to be written.

This function does not return a value.

This function is intended for advanced users only.

If you are not careful with this function, you could disable communications to your SNAP Engine. Still, if you know what you are doing, peekRadio() and pokeRadio() allow you to take advantage of additional features of the radio itself (features not supported by the "core" firmware).

print – Generate output from your script

The print capability of SNAPpy is not really a function (you don't put parentheses characters after it, for example), but it does let you send output from your SNAPpy scripts. Here are some examples:

```
print 'hello'
print 123
print xyz
```

You can also print multiple items from a single print statement, though unlike Python, SNAPpy does not insert a space between printed elements:

```
print 'this ', 'is ', 'a ', 'test'
```

The result of each individual print statement will usually go on a separate line. You must use a trailing comma (",") character to override this.

```
print 'line 1'
print 'line 2'
print 'line 3',
print ' and even more line 3'
```

Output from the print statement is enqueued to STDOUT, which can be connected to a serial port or transparent connection using the switchboard API (see the crossConnect() function).

Since a limited number of output RAM buffers can be enqueued to STDOUT, a script doing lots of 'print' output will need to HOOK_STDOUT. This allows your script to 'print' more output as space becomes available.

pulsePin(*pin, msWidth, isPositive*) – Generate a timed pulse

You *could* generate a pulse using an IO pin, and multiple writePin() commands. Function pulsePin() lets you initiate the pulse in a single step, gives you finer grained control of the pulse duration, and frees your script from having to "time" (countdown) the pulse.

Parameter *pin* is which IO pin (GPIO_0-GPIO_18 on an RF100 SNAP Engine) to generate the pulse on. Parameter *msWidth* specifies the desired pulse width in milliseconds (1-32767)

Specifying a pulse width of 0 will simply result in no pulse at all.

Specifying a pulse width > 0 will generate a non-blocking pulse – pulsePin() just initiates the pulse, but your SNAPpy script continues to run *in parallel*. Among other things, this means you can have multiple pulses in progress at the same time.

As of version 2.2, you can also specify a *negative* pulse width. Specifying a negative pulse width instead of a positive pulse width does two things:

1. It makes the pulse generation "blocking." When you specify a negative duration, the pulse runs to completion, and *then* your SNAPpy script resumes execution at the next line of code.
2. The time units are different for negative values. The resolution varies by platform, but is typically in the range of 1 microsecond per "tick." Refer to the "platform section" for your hardware in section 10 of this document for specifics. As a quick example, on a Synapse RF100 SNAP Engine a msWidth of -10000 would result in a pulse approximately 10 milliseconds wide.

Parameter *isPositive* controls the polarity of the pulse. It essentially specifies the logic level of the leading edge of the pulse, and the opposite of this value is used for the trailing edge of the pulse.

This function has no effect unless/until the specified IO pin is also configured as an output (via setPinDir()).

This function does not return a value.

random() – Generate a random number

This function returns a random number between 0-4095.

This function does not take any parameters.

readAdc(*channel*) – Read an Analog Input pin (or reference)

This function can be called to read one of the available analog input channels. Some channels correspond to external analog input pins, the internal low voltage reference, or the internal high voltage reference.

Parameter *channel* specifies which analog input channel (platform dependent) to read.

readPin(*pin*) – Read the logic level of a pin

This function can be called for IO pins that are configured as digital inputs or digital outputs (see also setPinDir()).

Parameter *pin* specifies which IO pin to read.

This function returns a boolean value representing the current logic level of the specified pin. For an input pin, this is a "live value." For an output pin, this is the last value written *to* the pin.

reboot() – Schedule a reboot

This function takes no parameters, and returns no value.

Approximately 200 milliseconds after this function is called, the SNAP node will reboot.

The 200 milliseconds delay is to allow the node time to acknowledge the reboot() request (in case it came in over-the-air).

resetVm() – Reset (shut down) the SNAPpy Virtual Machine

This function takes no parameters, and returns no value.

When this function is called, the SNAPpy Virtual Machine stops running any loaded script (but the script remains in the unit).

This function is used (by Portal and SNAPconnect) at the start of the script upload process, and would not normally be used by users. However, sometimes when testing a script it is useful to be able to halt it.

See also function initVm(), which restarts the SNAPpy VM.

rpc(*address, function, args…*) – Remote Procedure Call (RPC)

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

For example, `rpc('\x12\x34\x56', 'writePin', 0, True)` will ask the node at address 12.34.56 to do a writePin(0, True).

`rpc('\x56\x78\x9A', 'reboot')` will ask the node at address 56.78.9A to do a reboot().

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

If this function returns True, it does not mean your RPC request was successfully sent and received. (SNAP will give up after a programmable number of retries, which defaults to 8.) If you need confirmation that the remote node executed your request, it needs to come as an RPC call from that remote node. Refer to the callback() function for one method of doing this.

It is important that you provide the correct number of arguments for the function you are calling. The tooltip help that displays in Portal shows three arguments for this function (address, function, args), but the remote function you are calling might require more or fewer arguments than the one specified. A remote call of `rpc('\x56\x78\x9A', 'reboot')` will ask the node at address 56.78.9A to do a reboot(), while `rpc('\x56\x78\x9A', 'reboot', '')` will not work because the receiving node will not find a built-in function with a matching function signature. If the remote function you are calling does not require any arguments, you should omit the third “args” argument of the RPC function. If the remote function requires more arguments, you should include them, too.

Built-in functions in nodes can always be called interactively from Portal, but if you are invoking a built-in function in a node from a script in another node, the node on which the function is being called must also have a script loaded, even if it is just an empty script. In the act of loading the script file, Portal establishes a function table in the node that must be present before the remote node can find built-in functions to run from an RPC call.

rpcSourceAddr() – Who made this Remote Procedure Call?

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

This function allows a node to respond (answer back) directly to other nodes. An example will make this clearer.

Imagine node A is loaded with a script containing the following function definition:

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

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

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

Node A can invoke function "ping" on node B, but it has to know node B's address to do so:

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

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

All SNAP Network Addresses are three-byte strings.

This function takes no parameters.

rx(*isEnabled*) – Turn radio receiver on/off

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

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

The radio defaults to ON in SNAP Nodes. If you invoke rx(False), the radio will be powered down. Invoking rx(True) will power the radio back up.

In addition, *sending* any data over the radio will automatically wake the radio back up.

To be clear: a node can wake up its own radio by attempting to transmit. A node's radio will **not** be woken up by transmissions from other nodes. This function does not return a value.

saveNvParam(*id, obj*) – Save data into NV memory

This function lets you store individual pieces of data into the SNAP node's Non-Volatile (NV) memory.

Parameter *id* specifies which "key" to store the *obj* parameter under. NV parameter IDs 0, 255, and 1-127 all have pre-assigned meanings (refer to section 8). IDs 128-254 are user defined, your script can store whatever you want, under any ID (128-254) that you want. See section 8 for a listing of the available parameters.

The *obj* parameter should be the data you want to store, and can be a boolean, an integer, a string, or a None.

See also function loadNvParam().

This function returns a result code from the following list:

NV_SUCCESS = 0,

// Possible LOAD errors
NV_NOT_FOUND = 1,
NV_DEST_TOO_SMALL = 2,

// Possible SAVE errors
NV_FULL = 3, // no more room in NV (even after "compression")
NV_BAD_LENGTH = 4,
NV_FAILURE = 5, // literally unable to write to FLASH (should never happen)
NV_BAD_TYPE = 6, // invalid or unsupported data type
NV_LOW_POWER = 7, // we refuse to even try if power is bad (low voltage)

scanEnergy() – Get energy readings from all channels

The getEnergy() function returns the result of a brief radio Energy Detection scan.

Function scanEnergy() is an extension of getEnergy(). It essentially calls getEnergy() N times in a row, changing the frequency before each getEnergy() scan. Here, 'N' refers to the number of frequencies supported by the radio.

For 2.4 GHz radios, 16 frequencies are supported by the radios, each corresponding to one channel. For 900 MHz radios running FHSS (frequency hopping) firmware, the 16 channels cover 66 radio frequencies, with each channel making use of 25 of those frequencies. For 868 MHz radios, there are three frequencies used, regardless of the channel selected.

See the getChannel() function explanation for more details about how each radio platform uses the various frequencies available to it.

Function scanEnergy() returns an N-byte string, where the first character corresponds to the "detected energy level" on frequency 0, the next character goes with channel 1, and so on. (For 900 MHz FHSS radios, SNAP does not make use of the first and last frequencies, but returns them as part of the string for completeness.)

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

This function takes no parameters.

setChannel(*channel*) – Specify which channel the node is on

For all SNAP devices, the setChannel() function takes a number in the 0-15 range to specify which frequency (or range of frequencies) the device should use for its communications. Refer to the description for function getChannel() for more on this topic.

On 802.15.4/2.4 GHz devices, channels 0-15 correspond to 802.15.4 channel 11-26.

For 900 MHz devices running FHSS (frequency-hopping) firmware, the channel normally specifies which range of 25 frequencies SNAP should use to distribute its transmissions across the frequency range. However you can set the channel to any value up to 65, and need to do so to get meaningful data from the getEnergy() function. See the getEnergy() function for more details on this topic.

Note that this function changes the "live" channel setting, and the effect only lasts until the next reboot or power cycle. You can also use saveNvParam() to save the "persisted" channel setting into NV parameter 4 if you want the node to *stay* on that channel.

This function does not return a value.

setNetId(*networkId*) – Specify which Network ID the node is on

The setNetId() function takes a Network ID parameter 0-0xFFFF representing which SNAP Network ID the node should switch to. Note that Network ID 0xFFFF is considered a "wildcard" network ID (matches all nodes), and you normally should only use network IDs of 0-0xFFFE.

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

See also getNetId(). This function returns no value.

Note that this function changes the "live" network ID setting, and the effect only lasts until the next reboot or power cycle. You can also use saveNvParam() to save the "persisted" network ID setting in NV parameter 3 if you want the node to *stay* on that network ID after its next reboot.

setPinDir(*pin, isOutput*) – Set direction (input or output) for a pin

This function should be called for each IO pin you want to use as either a digital input or digital output in your application.

Parameter *pin* specifies which IO pin to configure. Parameter *isOutput* makes the pin be an output when True, or an input when False.

For a given IO pin, you should call this function once to initialize the pin before calling functions such as setPinPullup(), readPin() and monitorPin() (for input pins) or setPinSlew(), writePin() and pulsePin() (for output pins).

This function does not return a value.

setPinPullup(*pin, isEnabled*) – Control internal pull-up resistor

This function should be called for each IO pin you are using as a digital input if you want the internal pull-up resistor for that pin to be enabled. (The default pull-up setting is off, so you usually do not have to call this function unless you *want* the pull-up enabled), or you previously enabled it and now want to disable it.

Parameter *pin* specifies which IO pin to configure. Parameter *isEnabled* makes the internal pull-up for the pin be active when True, or inactive when False.

This function has no effect unless/until the pin is also configured as a digital input pin.

This function does not return a value.

setPinSlew(*pin, isRateControl*) – Enable/disable slew rate control

On every platform except the Si100x, this function should be called for each IO pin you are using as a digital output, if you want the internal slew rate control for that pin to also be enabled. (The default slew rate setting is off, so you usually do not have to call this function unless you *want* the slew rate control enabled).

Parameter *pin* specifies which IO pin to configure. Parameter *isRateControl* makes the slew rate control be active when True, or inactive when False.

On Si100x-based modules (such as the RF300), the `setPinSlew()` function controls the strength at which an output is driven. See the platform details in section 10 for more information.

This function has no effect unless/until the pin is also configured as a digital output pin. This function does not return a value.

setRadioRate(*rate*) – Set raw radio data rate

Parameter *rate* should be set to 0 to specify the standard (and default) data rate for the platform, e.g. 250 Kbps for 802.15.4 based devices.

The meaning of other *rate* values is platform specific; refer to section 10 of this document.

NOTE – Only units set to the same rate can talk to each other over the air!

NOTE – The "encoding" for non-standard data rates may differ between radio manufacturers. This means that different radio hardware may not be able to interoperate, even if set to the same (non-standard) rate. All radios on the same frequency range set to rate 0 will be able to interoperate.

setRate(*rate*) – Set monitorPin() sample rate

By default, the background pin sampling that is enabled/disabled by function monitorPin() takes place 10 times a second (every 100 milliseconds). This function allows you to vary that sampling rate.

Parameter *rate* specifies whether the sampling should be OFF (*rate* =0), every 100 ms (*rate* = 1), every 10 ms (*rate* = 2), or every 1 ms (*rate* = 3).

This function has no effect unless/until you are actually using pin monitoring.

This function does not return a value.

setSegments(*segments*) – Update seven-segment display

This function is only useful on Synapse boards that have a seven-segment (per digit) display on them. It assumes a particular type of "shift register controlled" display, that requires continuous refresh, with GPIO pins 13 and 14 controlling the display.

Parameter *segments* specifies a 16-bit binary pattern that controls which segments will be lit, and which will be dark. Because the displays currently used do not have decimal points, only 14 of the total 16 bits are meaningful.

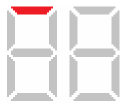
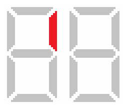
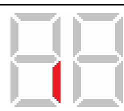
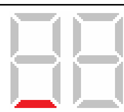
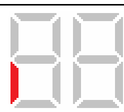
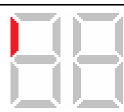
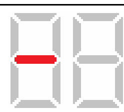
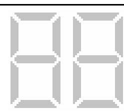
A *segments* value of 0x0000 turns off all segments. A value of 0x7F7F corresponds to "all segments on."

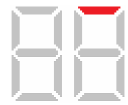
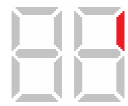
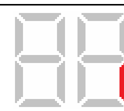
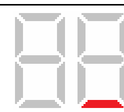
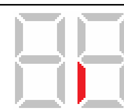
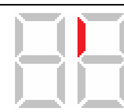
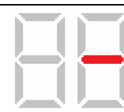
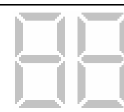
This interface gives you complete control of the display, but you can wrap the display in higher level access functions. See for example function display2digits() in supplied script evalBase.py.

Not all platforms have the `setSegments(segments)` built-in. (See the details for platform you are using in Section 10.) For broader compatibility, consider importing synapse.sevenSegment and using the `SetSegments(segments)` function instead. (Note the first letter is capitalized.)

The actual "bit to segment" assignments make sort of a "clockwise inward spiral" path around each digit.

Refer to the following table for more details.

Bit (in hexadecimal)	Bit Position Within Display
0x0100	
0x0200	
0x0400	
0x0800	
0x1000	
0x2000	
0x4000	
0x8000 (no effect)	

Bit (in hexadecimal)	Bit Position Within Display
0x0001	
0x0002	
0x0004	
0x0008	
0x0010	
0x0020	
0x0040	
0x0080 (no effect)	

sleep(*mode, ticks*) – Go to sleep (enter low-power mode)

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

Parameter *mode* chooses from the available sleep modes. The number of modes available, and their characteristics (such as resolution and accuracy), vary from platform to platform. Refer to section 10 at the end of this document.

A *ticks* parameter of 0 can be used to sleep until an IO pin interrupt occurs (see script pinWakeup.py), but SNAPpy is smart enough to know if you have *not* enabled a wakeup pin, and will **ignore** a sleep(*mode*, 0) if there is no wakeup possible.

Starting with version 2.2, a negative *ticks* parameter can be used to access alternate sleep timings. These also vary between platforms, refer to the section for your hardware.

This function does not return a value.

spiInit(*cpol, cpha, isMsbFirst, isFourWire*) – Setup SPI Bus

This function initializes the SNAP node to perform Serial Peripheral Interface (SPI) Bus interfacing.

The SPI standard supports multiple options, hence the large number of parameters in spiInit().

Parameter *cpol* refers to Clock Polarity, and can be either True or False. Basically it specifies the level of the CLK pin *between* SPI exchanges. To put it another way, *cpol* specifies the idle clock level.

Parameter *cpha* refers to the Clock Phase, and can be True or False. It specifies which clock edge the incoming data is to be "latched in." If you number clock edges from 1, then setting *cpha* = True specifies the **even** clock edges for incoming data, and setting *cpha* = False specifies the odd clock edges for incoming data.

> Because SPI mode is specified using both a *cpol* and *cpha* setting, there are **four possible combinations**. *Which* combination is correct depends on the device you are interfacing to; refer to the manufacturer's data sheets.

Parameter *isMsbFirst* controls the order in which individual bits within each byte will be shifted out. Setting this parameter to True will make the 0x80 bit go out first, setting this parameter to False will make the 0x01 bit go out first.

> Again, the *correct* setting for the *isMsbFirst* parameter depends on the device to which you are interfacing.

Parameter *isFourWire* lets you select the variant of SPI you are connecting to. Three wire SPI omits the MISO pin. In three-wire SPI, even if the slave does send data, it is over the MOSI pin.

This function does not return a value.

spiRead(*byteCount, bitsInLastByte*=8) – SPI Bus Read

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

This function reads data from a three wire SPI device (for four wire SPI, you should be using the bidirectional function spiXfer() instead).

Parameter *byteCount* specifies how many bytes to read.

Parameter *bitsInLastByte* makes it possible to accommodate devices with data widths that are not multiples of 8 (like 12 bits). The default value of *bitsInLastByte* is 8. For a device with a data width of 12 bits, *bitsInLastByte* would be set to 4. For a device with a data width of 31 bits, *bitsInLastByte* would be set to 7.

The order that bits get shifted in depends on the value of parameter *isMsbFirst* which was specified in the previous spiInit() call.

This function returns a string containing the actual bytes received.

More background information on using SPI is in section 6.

spiWrite(*byteStr*, *bitsInLastByte*=8) – SPI Bus Write

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

This function writes data to a three or four wire SPI device. If you want to write and read data simultaneously (four wire SPI only), then you should be using the bidirectional function spiXfer() instead of this one).

Parameter *byteStr* specifies the actual bytes to be shifted out.

Parameter *bitsInLastByte* makes it possible to accommodate devices with data widths like 12 bits. The default value of *bitsInLastByte* is 8. For a device with a data width of 12 bits, *bitsInLastByte* would be set to 4. For a device with a data width of 31 bits, *bitsInLastByte* would be set to 7.

The order that bits get shifted out depends on the value of parameter *isMsbFirst* which was specified in the previous spiInit() call.

This function does not return a value.

More background information on using SPI is in section 6.

spiXfer(*byteStr*, *bitsInLastByte*=8) – Bidirectional SPI Transfer

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

This function reads and writes data over a four wire SPI device. If your device is read-only or write-only, you should look at the spiRead() and spiWrite() routines.

Parameter *byteStr* specifies the actual bytes to be shifted out.

As these bits are being shifted out to the slave device (on the MOSI pin), bits from the slave device (on the MISO pin) are simultaneously shifted in.

Parameter *bitsInLastByte* makes it possible to accommodate devices with data widths like 12 bits. The default value of *bitsInLastByte* is 8. For a device with a data width of 12 bits, *bitsInLastByte* would be set to 4. For a device with a data width of 31 bits, *bitsInLastByte* would be set to 7.

The order that bits get shifted in and out depends on the value of parameter *isMsbFirst* which was specified in the previous spiInit() call.

This function returns a byte string consisting of the bits that were shifted in (as the bits specified by parameter *byteStr* were shifted out).

More background information on using SPI is in section 6.

stdinMode(*mode, echo*) – Set console input options

This function controls how serial data gets presented to your SNAPpy script (via the HOOK_STDIN), and how it appears to the user.

Parameter *mode* chooses between line-at-a-time (*mode* = 0) or character based (*mode* = 1).

In "line mode" (mode 0, the default), characters are buffered up until either a Carriage Return (CR) or Line Feed (LF) are received. The complete string is then given to your SNAPpy script in a single HOOK_STDIN invocation. Note that **either** character can trigger the handoff, so if your terminal (or terminal emulator) is automatically adding extra CR or LF characters, you will see additional empty strings ("") passed to your script.

> The character sequence *A B C CR LF* looks like two lines of input to SNAPpy.

In "character mode" (mode 1), characters are passed to your SNAPpy script as soon as they become available. If characters are being received fast enough, it still is possible for your script to receive more than one character at a time; they are just not buffered waiting for a CR or LF.

While your node is in line mode, SNAP reserves one "medium" string buffer to accept incoming data from standard-in. If your script is heavy on string usage but does not make use of the HOOK_STDIN hook, you can set the standard-in mode to character mode and recover the use of the medium string.

Parameter *echo* is a Boolean parameter that controls whether or not the SNAP firmware will "echo" (retransmit) the received characters back out (so that the user can see what they are typing).

This function does not return a value.

str(*object*) – Return the string representation of an object

Function str() returns a string based on the value of the object you give it.

For example, str(123) = '123', str(True) = 'True', and str('hello') = 'hello'.

txPwr(*power*) – Set Radio TX power level

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

Parameter *power* specifies a transmit power level from 0-17, with 0 being the lowest power setting and 17 being the highest power setting.

This function does not return a value.

ucastSerial(*destAddr*) – Setup outbound TRANSPARENT MODE

SNAP TRANSPARENT MODE is covered in section 5.

When you want the outbound data to be sent to a specific node, use this function.

Parameter *destAddr* specifies the Network Address of some other node to give the received serial characters to.

If you want the received serial characters to go to more than one node, you should use function mcastSerial() instead.

This function does not return a value.

uniConnect(*dest, src*) – Make a one-way switchboard connection

The SNAPpy switchboard is covered in section 5.

This function establishes a one-way connection between two points. See included script switchboard.py for the possible values of parameters *src* and *dest*.[6]

See also function crossConnect() if you need a <u>bi-directional</u> hookup. As an alternative, two uniConnect() calls can be equal to one crossConnect() call. For example:

```
uniConnect(UART0, UART1)
uniConnect(UART1, UART0)
```

has the same effect as:

```
crossConnect(UART0, UART1)
```

NOTE – A source can only be connected to a single destination. Multiple sources can feed into a single destination.

This function does not return a value.

[6] Most platforms have two UARTs available, so with most SNAP Engines UART0 will connect to the USB port on a SN163 board and UART1 will connect to the RS-232 port on any appropriate Synapse demonstration board. However the RF300 SNAP Engine has only one UART – UART0 – and it comes out where UART1 normally comes out (to the RS-232 port, via GPIO pins 7 through 10). If you are working with RF300 SNAP Engines, be sure to adjust your code to reference UART0 rather than UART1 for your RS-232 serial connections.

vmStat(*statusCode, args…*) – Invoke “status” callbacks

This function is specialized for management applications (like Portal), and provides a range of control/callback functionality.

Parameter *statusCode* controls what actions will be taken, and what data will be returned (via a tellVmStat() callback to the original node).

The currently supported *statusCode* values are:
0 = VM_RESET: stop the SNAPpy Virtual Machine (for script uploading)
1 = VM_IMG_ERASE: erase the current SNAPpy script
2 = VM_WR_BLK: used when uploading scripts – DO NOT CALL THIS!
3 = VM_INIT: restart the SNAPpy Virtual Machine (after script uploading)
4 = VM_NVREAD: read the specified NV Parameter
5 = VM_NAME: returns NODE NAME if set, else IMAGE NAME, plus Link Quality
6 = VM_VERSION: returns software version number
7 = VM_NET: returns Network ID and Channel
8 = VM_SPACE: returns Total Image (script) Space Available
9 = VM_SCAN: scans all 16 channels for energy, returns 16 character string
10 = VM_INFO: returns Image Name (script name) and Link Quality

You probably should not be invoking **vmStat**() with status codes of 0-3 (unless you are implementing your own downloader). Status codes 4-10 are safe to call.

After the statusCode, the next argument varies (depending on the statusCode).

After the “varying” argument comes a final optional argument that specifies a “time window” to randomly reply within. (More about this below.)

For VM_NVREAD, the second argument is the ID of the NV Parameter you want to read (these are the same IDs used in the saveNvParam() and loadNvParam() functions). You can also optionally specify a "reply window."

The “system” NV Parameter IDs are given in section 8.

The reported values will be a "hiByte" of the NV Parameter ID, and a "data" of the actual NV Parameter value.

For VM_NAME, the only parameter is the optional reply window.

The reported “data” value will be a string name and a Link Quality reading.

For VM_VERSION, the only parameter is the optional reply window.

The reported “data” value will be a version number string

For VM_NET, the only parameter is the optional reply window.

The reported values will be a “hiByte” containing the currently active channel (0-15), and a “data” value of the current Network ID

For VM_SPACE, the only parameter is the optional reply window.

The reported “data” value will be the Total Image (script) Space Available

For VM_SCAN, the only parameter is the optional reply window.

The reported “data” value will be a 16 character string containing the detected energy levels on all 16 channels. Note that each scan just represents one point in time, you will probably have to initiate multiple scans to determine which channels actually have SNAP nodes on them.

You can see this VM_SCAN function put to use in the *Channel Analyzer* feature of Portal.

See also function scanEnergy(), which returns data in an equivalent format.

For VM_INFO, the only parameter is the optional reply window.

The reported values will be a “hiByte” of the current Link Quality, and a “data” of the currently loaded script name (a string).

Return value format:

All of the VM_xxx functions invoke a callback named tellVmStat(*word*, *data*)

The least significant byte of *word* will be the originally requested *statusCode*. The most significant byte will vary depending on the *statusCode*, and is the “hiByte” described above. The *data* value is the main return value, and is also dependant on the *statusCode*.

Return value timing:

If you do not specify a “time window” parameter, the nodes will respond immediately.

Some of these commands are multicast by Portal, and we needed a way to keep all of the nodes from trying to respond at once.

Specifying a non-zero “time window” tells the node to pick a random time within the next “time window” seconds, and wait until then to reply.

This function does not return a value, but it causes a tellVmStat() call to be made to the node that requested the vmStat().

Note – because of the callback() function, some of the vmStat() capabilities are redundant.

writeChunk(*offset, data*) – Synapse Use Only

This function is used by Portal and SNAPconnect as part of the script uploading process.

There should be no reason for user scripts to call this function, and attempting to do so could erase or corrupt all of your SNAP firmware, requiring a firmware reload (Portal has the capability to do this).

This function does not return a value.

writePin(*pin, isHigh*) – Set output pin level

This function allows you to control digital output pins (IO pins configured as digital outputs).

Parameter *pin* specifies which IO pin to control. Parameter *isHigh* makes the pin go high (True) or low (False).

This function has no effect unless/until the pin is configured as a digital output pin via setPinDir(). See also related function setPinSlew(), which controls how quickly the pin will transition to a new value.

This function does not return a value.

Here are the functions again, but this time broken down by category.

ADC

readAdc(*channel*)	Sample ADC on specified input channel, returns raw reading

CBUS Master Emulation

cbusRd(*numToRead*)	Reads *numToRead* bytes from CBUS, returns string
cbusWr(*byteStr*)	Writes every byte in *byteStr* to the CBUS

These functions are discussed in section 6 of this document.

GPIO

setPinDir(*pin, isOutput*)	Set direction for parallel I/O pin
setPinPullup(*pin, isEnabled*)	Enable pull-up resistor for Input pin
setPinSlew(*pin, isRateControl*)	Enable slew rate-control for Output pin
monitorPin(*pin, isMonitored*)	Enable GPIN events on Input pin
pulsePin(*pin, msWidth, isPositive*)	Apply pulse to Output pin
readPin(*pin*)	Read current level of pin
writePin(*pin, isHigh*)	Set Output pin level
setRate(*rateCode*)	Set pin sampling rate to off (0), 100 ms (1), 10 ms (2), or 1 ms (3)

I^2C Master Emulation

getI2cResult()	Returns the result of the most recent I^2C operation
i2cInit(*enablePullups*)	Prepare for I^2C operations
i2cRead(*str, numBytes, retries, ignoreFirstAck*)	Write *str* out, then read *numBytes* back in from I^2C bus. Parameters *retries* and *ignoreFirstAck* are used with slow or special case devices
i2cWrite(*str, retries, ignoreFirstAck*)	Write *str* out over the I^2C bus. Parameters *retries* and *ignoreFirstAck* are used with slow or special case devices

These functions are discussed in section 6 of this document.

Misc

setSegments(*segments*)	Set eval-board LED segments (clockwise bitmask)
bist()	Built-in self test
eraseImage()	Erase user-application FLASH memory
resetVm()	Reset the embedded virtual machine (prep for upload)
initVm()	Initialize embedded virtual machine
vmStat(*statusCode, args...*)	Solicit a tellVmStat for system parameters

writeChunk(*ofs, str*)	Write string to user-application FLASH memory
chr(*number*)	Returns the character string representation of “number”
str(*obj*)	Returns the string representation of *obj*
int(*obj*)	Returns the integer representation of *obj* Notice that you cannot specify the base. **Decimal** is assumed
len(*str*)	Returns the length of string *str* (0-255)
random()	Returns a pseudo-random number 0-4095
stdinMode(*mode, echo*)	*mode* is 0 for line, 1 for character at a time *echo* is True or False

Network

getNetId()	Current Network ID
setNetId(*netId*)	Set Network ID (1-0xFFFE)
localAddr()	Local network address (3-byte binary string)
rpcSourceAddr()	Originating address of current RPC context (None if called outside RPC)
mcastSerial(*dstGroups, ttl*)	Set Serial transparent mode to multicast
ucastSerial(*dstAddr*)	Set Serial transparent mode to unicast
callback(*callbackFnObj, remoteFnObj, args...*)	Remote Procedure Call (back to the original invoker) of Remote Procedure Call results
callout(*addr, callbackFnObj, remoteFnObj, args...*)	Remote Procedure Call (to an arbitrary node address) of Remote Procedure Call results
rpc(*dstAddr, remoteFnObj, args...*)	Remote Procedure Call (unicast)
mcastRpc(*dstGroups, ttl, remoteFnObj, args...*)	Remote Procedure Call (multicast)

Non-Volatile (NV) Parameters

loadNvParam(*id*)	Load indexed parameter from NV storage
saveNvParam(*id, obj*)	Save object to indexed NV storage location

Radio

rx(*isEnabled*)	Enable/disable radio receiver
txPwr(*power*)	Adjust radio transmit level (0 is lowest, 17 is highest)
setChannel(*channel*)	Set radio channel
getChannel()	Radio channel
getLq()	Link Quality in (-) dBm
getEnergy()	Detected RF energy in (-) dBm (current channel)
scanEnergy()	Detected RF energy in (-) dBm (all 16 channels)
peekRadio(*addr*)	Read a memory location from inside the radio
pokeRadio(*addr, byteVal*)	Write a memory location inside the radio

SPI Master Emulation

spiInit(*cpol, cpha, isMsbFirst, isFourWire*)	setup for SPI, with specified Clock Polarity, Clock Phase, Bit Order, and Physical Interface
spiRead(*byteCount, bitsInLastByte*)	receive data in from SPI – returns response string (three wire SPI only)
spiWrite(*byteStr, bitsInLastByte*)	send data out SPI – *bitsInLastByte* defaults to 8, can be less
spiXfer(*byteStr, bitsInLastByte*)	bidirectional SPI transfer – returns response string (four wire SPI only)

These functions are discussed in section 6 of this document.

Switchboard

crossConnect(*dataSrc1, dataSrc2*)	Cross-connect SNAP data-sources
uniConnect(*dst, src*)	Connect src->dst SNAP data-sources

Use crossConnect() to setup bidirectional transfers. Use uniConnect() to setup a one-way connection. Note that multiple sources can be uni-connected to the same destination.

System

getMs()	System millisecond tick (16bit)
getInfo(*which*)	Get specified system info
getStat(*which*)	Get radio traffic status info
call()	Invoke a user-defined binary function
peek(*addr*)	Read a memory location
poke(*addr, byteVal*)	Write a memory location
errno()	Read and reset last error code
imageName()	Name of current SNAPpy image
random()	Returns a random number (0-4095)
reboot()	Reboot the device
sleep(*mode, ticks*)	Enter sleep mode for specified number of ticks Resolution, accuracy, and maximum duration vary between hardware platforms. For example, on an RF100 SNAP Engine: In mode 0, ticks are 1.024 seconds each, +/- 30% In mode 1, ticks are 1 second each, and can be 0-1073 On some platforms, negative values for ticks produce times shorter than one second.

UARTs

initUart(*uartNum, bps*)	Enable UART at specified rate (zero rate to disable)
initUart(*uartNum, bps, dataBits, parity, stop*)	Enable UART at specified rate (zero rate to disable), data bits, parity, and stop bits
flowControl(*uartNum, isEnabled*)	Enable RTS/CTS flow control. If enabled, the CTS pin functions as a "Clear To Send" indicator
flowControl(*uartNum, isEnabled, isTxEnable*)	Enable RTS/CTS flow control. If enabled and parameter *isTxEnable* is True, then the CTS pin functions as a TXENA (transmit enable) signal. If enabled and *isTxEnable* is False, then the CTS pin functions as a "Clear To Send" indicator

Note: uartNum is 0 or 1 on most platforms. Some platforms will have only UART0 available.

Immediate Functions

Most SNAPpy built-ins (when called) quickly do their job, then return. Script execution then continues with the next line of SNAPpy source code. Although technically they are *blocking* functions (they do not return until they have completed), because of their relatively short duration we classify them as *immediate* functions.

The following SNAPpy built-ins are classified as **immediate**:

chr(), errno(), flowControl(), getChannel(), getI2cResult(), getInfo(), getLq(), getMs(), getNetId(), getStat(), imageName(), i2cInit(), initUart(), int(), len(), localAddr(), mcastSerial(), ord(), peek(), peekRadio(), poke(), pokeRadio(), random(), readAdc(), readPin(), rpcSourceAddr(), rx(), setChannel(), setNetId(), setPinDir(), setPinPullup(), setPinSlew(), setRadioRate(), setRate(), spiInit(), stdinMode(), str(), txPwr(), ucastSerial(), uniConnect(), vmStat(), writePin()

Blocking Functions

Some SNAPpy built-ins take too long to complete to be classified as immediate. Execution of the next line of script does not occur until these functions complete.

The following functions are classified as **blocking**:

bist(), call(), cbusRd(), cbusWr(), eraseImage(), getEnergy(), i2cRead(), i2cWrite(), initVm(), lcdPlot(), loadNvParam(), pulsePin() *with a negative duration specified on hardware that supports it*, resetVm(), saveNvParam(), scanEnergy(), sleep(), spiRead(), spiWrite(), spiXfer(), writeChunk()

Non-blocking Functions

Some functions (once initiated) actually complete in the background. Script execution continues, while the requested function continues to take place behind the scenes.

The following functions are classified as **non-blocking**:

callback(), callout(), mcastRpc(), rpc() – the RPC packet is built immediately (if possible), but the actual *sending* occurs in the background. **Reminder** – these functions can return False (with no background processing) if there is insufficient RAM in which to enqueue the generated packet.

crossConnect() – the actual switchboard configuration takes place immediately, but a crossConnect() often results in ongoing data transfer afterwards.

monitorPin() – the tagging of the specified pin as “to be monitored” occurs immediately, but then the actual pin polling takes place in the background.

pulsePin() – the *leading* edge of the requested pulse is generated immediately, but the “countdown” to the trailing edge occurs in the background. Calling pulsePin() with a negative duration *is* blocking.

print – the textual output is *generated* in a blocking fashion, but the output is *sent* in the background.

reboot() – this function schedules a reboot in approximately 200 milliseconds, then allows script execution to continue until the reboot occurs.

setSegments() – the new pattern is defined immediately, but the actual periodic refresh occurs in the background.

Non-blocking Functions and SNAPpy Hooks

Some non-blocking functions result in "hook" (HOOK_xxx) callback functions being called, if they are defined in your script.

Use of function…	…can result in
rpc(), mcastRpc()	HOOK_RPC_SENT
monitorPin()	HOOK_GPIN
print	HOOK_STDOUT
crossConnect(), uniConnect()	HOOK_STDIN

SNAPpy Scripting Hints

The following are some helpful hints 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 in 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 (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 for some other function (for example, for script text output). 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)
```

…might not 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 timer code:

```
global sleepCountDown

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

Remember nodes do not have a lot of RAM
SNAPpy scripts should avoid generating a flood of text output all at once (there will be no where to buffer the output). Instead, generate the composite output in small pieces (for example, one line at a time), triggering 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.

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 bitwise and-ing to get the desired effect if you need it.

```
X = X >> 1
X = X & 0x7FFF
```

Pay Attention to Script Output
Any SNAPpy script errors (see section 4) 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.

Use the Supported Form of Import

In SNAPpy scripts you should use the form:

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

Remember Portal Speaks Python Too

SNAPpy scripts are a very powerful tool, but the SNAPpy language is a 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.

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. 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. 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.

If you want to call a built-in function *by name*, the <u>called</u> node needs a script loaded, even if it is empty
SNAP Nodes <u>without scripts loaded</u> 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. SNAP Node Configuration Parameters

You make your SNAP nodes do completely new things by loading SNAPpy scripts into them. You can often adjust the way they do the things that are already built-in by adjusting one or more *Configuration Parameters*.

These *Configuration Parameters* are stored in a section of Non-Volatile (NV) memory within the SNAP node. For this reason *Configuration Parameters* are also referred to as *NV Parameters*.

SNAPpy scripts can access these NV Parameters by using the loadNvParam(*id*) function. SNAPpy scripts can change these parameters by using the saveNvParam() function (and then rebooting so that the changes will take effect).

When using the loadNvParam() and saveNvParam() functions, you must specify *which* NV Parameter by *numeric ID*.

You can also easily view and edit these parameters using Portal. Refer to the **Portal Reference Manual**. When you view and edit these parameters from Portal, you do not need to know the NV Parameter ID. Portal takes care of that for you.

Some of the NV Parameters control the functionality of your nodes at a very fundamental level. Making careless changes to these parameters can cause you to lose access to your nodes, either over the air, over a serial connection, or both. If you find you are unable to make any connection to a node, you will be able to return the node to your control through the *Factory Default NV Params...* option in Portal. (It may first be necessary to use the *Erase SNAPpy Image...* option in Portal, if the image loaded into the node is setting NV Parameters.)

Here are all of the System (Reserved) NV Parameters (sorted by numeric ID), and what they do.

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

> Remember – you must reboot a node after changing any system NV Parameter for the change to actually take effect.

ID 0 – Reserved for Synapse Use

0 means “erased” inside the actual NV storage.

ID 1 – Reserved for Synapse Use

used to support a NV page-swapping scheme internally.

ID 2 – MAC Address

The eight byte address of the SNAP Node. This parameter is not modified when you reset parameters to factory defaults.

ID 3 – Network ID

The 16-bit Network Identifier of the SNAP Node. The Network ID and the Channel are what determine which radios can communicate with each other in a wireless network. Radios must be set to the same channel and Network ID in order to communicate over the air. Nodes communicating over a serial link pay no attention to the channel and Network ID.

Network IDs can be set to any value from 0x0000 through 0xFFFF. However 0xFFFF is a wildcard value to which all nodes respond and should generally be avoided. The default Network ID is 0x1C2C.

ID 4 – Channel

The channel on which the SNAP Node broadcasts. See also, Network ID.

The channel can be set to any value from 0 to 15. The Channel Analyzer in Portal can help you determine which channel has the least traffic on it in your environment. The default channel is 4.

ID 5 – Multi-cast Processed Groups

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

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

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

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

ID 6 – Multi-cast Forwarded Groups

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

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

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

NOTE – If you set your bridge node to not forward multi-cast commands, Portal will not be able to multi-cast to the rest of your network.

ID 7 – Manufacturing Date

Synapse use only. This parameter is not modified when you reset parameters to factory defaults.

ID 8 – Device Name

This NV Parameter lets you choose a name for the node, rather than letting it be determined by what *script* happened to be loaded in the node at the time Portal first detected it. If this parameter is set to None, then the first detected script name will determine the node name. If this parameter is blank and the node has no script loaded, it will have "Node" as its name. You do not *have* to give your nodes explicit names.

NOTE – It is invalid to put embedded spaces in your **Device Name**. "My Node" is not a legal name, while "My_Node" is.

ID 9 – Last System Error

If a SNAP Node reboots due to a system error, it stores the error code telling *why* here in this NV Parameter (Synapse internal use only).

ID 10 – Device Type

This is a user-definable string that can be read by scripts. This allows a single script to fill multiple roles, by giving it a way to determine what type of node it is running on. This NV Parameter is one way to "categorize" your nodes.

ID 11 – Feature Bits

These control some miscellaneous hardware settings. The individual bits are:

Bit 0 (0b0000,0000,0000,0001 0x0001) – Enable Serial Port 0 (USB port on a SN163 board)[7]
Bit 1 (0b0000,0000,0000,0010 0x0002) – Enable hardware flow control on Serial Port 0
Bit 2 (0b0000,0000,0000,0100 0x0004) – Enable Serial Port 1 (RS-232 port on a SN111 or SN171 board) (This is the *only* serial port on a SN171 Proto Board)
Bit 3 (0b0000,0000,0000,1000 0x0008) – Enable hardware flow control on Serial Port 1
Bit 4 (0b0000,0000,0001,0000 0x0010) – Enable the radio Power Amplifier (PA)
Bit 5 (0b0000,0000,0010,0000 0x0020) – Enable external power-down output
Bit 6 (0b0000,0000,0100,0000 0x0040) – Enable alternate clock source
Bit 7 (0b0000,0000,1000,0000 0x0080) – Enable DS_AUDIO on platforms that support it
Bit 8 (0b0000,0001,0000,0000 0x0100) – Enable second data CRC
Bit 9 (0b0000,0010,0000,0000 0x0200) – Reduce TX Power levels to "World Wide" settings

Synapse RF100 SNAP Engines with PA hardware can be identified by the "RFET" on their labels. Units without PA hardware say "RFE" instead of "RFET."

[7] Most platforms have two UARTs available, so with most SNAP Engines UART0 will connect to the USB port on a SN163 board and UART1 will connect to the RS-232 port on any appropriate Synapse demonstration board. However the RF300 SNAP Engine has only one UART – UART0 – and it comes out where UART1 normally comes out (to the RS-232 port, via GPIO pins 7 through 10). If you are working with RF300 SNAP Engines, be sure to adjust your code to reference UART0 rather than UART1 for your RS-232 serial connections.

For RF100 SNAP Engines, the PA feature bit (0x10) should only be set on “RFET” units. Setting this bit on a “RFE” board will not harm the SNAP Engine, but will actually result in lower transmit power levels (a 20-40% reduction). The bit should be set for RF200 SNAP Engines, as well.

The external power-down bit (0x20) should be set on units that need to power down external hardware before going to sleep, and power it back up after they awake. This bit is not modified when you reset parameters to factory defaults.

The alternate clock source bit (0x40) modifies which timer is used on SNAP modules that have multiple timers available, for increased PWM flexibility. See the details about the individual platform builds to determine if an alternate clock is available on your platform.

The DS_AUDIO enable bit (0x80) enables I^2S audio communications over the SNAP network on platforms that support it. Refer to each platform’s details to determine whether this capability is available.

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

If you set this bit for the second CRC, you should set it in all nodes in your network, and enable the feature in your Portal preferences or as a feature bit in your SNAP Connect NV parameters. A node that does not have this parameter set will be able to hear and act on messages from a node that does have it set, but will not be able to communicate back to that node. Not all platforms support this second CRC. Refer to each platform’s details to determine whether this capability is available. This feature was added in release 2.4.20.

The world-wide bit (0x200) enables an alternate set of power restrictions on platforms that support it. For example, an SM700 module will normally enforce FCC (US) Transmit Power Restrictions. If the 0x0200 Feature Bit is set, the SM700 will instead enforce ETSI restrictions.

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

ID 12 – Default UART

This controls which UART will be pre-configured for Packet Serial Mode.

Normally the UART related settings would be specified by the SNAPpy scripts uploaded into the node. This *default setting* has been implemented to handle nodes that *have no scripts loaded yet*.

These defaults are overridden when needed!

Although you can request that one or both UARTs are disabled (via the Feature Bits), and you can request that there is no Packet Serial mode UART (by setting the Default UART parameter to 255), both of these user requests will be ignored *unless there is also* a valid SNAPpy script loaded into the unit. If the parameter is set to a value outside the range of UARTs on your module (other than 255), UART1 (UART0 on modules with only one UART) will be the default.

If there is *no* SNAPpy script loaded, a fail-safe mechanism kicks in and **forces** an active Packet Serial port to be initialized on UART1 (or UART0, if so specified in this parameter), regardless of the other configuration settings. This was done to help prevent users from "locking themselves out."

If there *is* a SNAPpy script loaded, then the assumption is that the *script* will take care of any configuration overrides needed, and the **Feature Bits** and the **Default UART** setting will be honored.

ID 13 – Buffering Timeout

This lets you tune the overall serial data timeout. This value is in milliseconds, and defaults to 5. This value controls the maximum amount of time between an initial character being received over the serial port, and a packet of buffered serial data being enqueued for processing. Regardless of the number of characters buffered or the rate at which they are being buffered, each time this timeout passes any buffered data will be queued.

Note that other factors can also trigger the queuing of the buffered serial data. In particular see the next two NV Parameters.

The larger this value is the more buffering will take place. In TRANSPARENT MODE, every packet has 12-15 bytes of overhead, so sending more serial characters per packet is more efficient. Also, when using MULTICAST TRANSPARENT MODE, keeping the characters together (in the same packet) improves overall reliability.

The tradeoff is that the larger this value is, the greater the maximum latency can be through the overall system.

ID 14 – Buffering Threshold

This value indicates the total packet size threshold used when sending packets of data. The size defaults to 75 bytes. If no timeout limit is reached first, this parameter will cause buffered data to be enqueued when there is sufficient data to cause the packet, including header, to be at least this many bytes long. At higher serial rates, this size can be overshot between SNAP checks of the packet size. There is no guarantee that packets will necessarily be precisely this size.

Each packet of data sent includes a header, which comprises 12 bytes for multicast packets and 15 bytes for unicast packets. So the actual amount of serial data sent in each packet will be reduced by either 12 or 15 fewer bytes, depending on whether the data is being sent by multicast or unicast. Additionally, if the feature bit in NV Parameter 11 indicates that SNAP should be using its second CRC to prevent data corruption, the data payload will be reduced by an additional two bytes. If you want to send N bytes of data per packet, this parameter should be set to N + 12 for multicasting or N + 15 for unicasting, or N + 14 for multicasting with a secondary CRC or N + 17 for unicasting with a secondary CRC.

The maximum SNAP packet size is 123 bytes. If you set this parameter to a value greater than 123, the system will simply substitute a value of 123. If you set this parameter equal to or less than the packet header size, SNAP will construct packets with a complete header and one byte of data.

Like parameters #13 and #15, larger values can result in larger (more efficient) packets, at the expense of greater latency. Also, at higher baud rates, setting this value too high can result in dropped characters if the packet buffer gets overfilled between SNAP checks.

ID 15 – Inter-character Timeout

This lets you tune inter-character serial data timeout. This value is in milliseconds, and defaults to 0 (in other words, disabled).

This timeout is similar to NV Parameter #13, but this one refers to the time *between* individual characters. One way of thinking of it: this timeout restarts with every received character – the other timeout always runs to completion.

Larger inter-character timeouts can give better MULTICAST TRANSPARENT MODE reliability, at the expense of greater latency.

Note that either timeout #13 or #15 (if enabled) can trigger the transmission of the buffered data before the Buffering Threshold (#14) is reached. Conversely, if the timeouts are high (or disabled) to the extent that enough data is buffered to reach the Buffering Threshold before the timeouts are reached, that threshold will trigger the transmission of the buffered data before either of the timeouts are reached.

ID 16 – Carrier Sense

Basically, this instructs the radio to “listen before you transmit.”

This value defaults to False. Setting this value to True will cause the node to do what is called a Clear Channel Assessment (CCA). Basically this means that the node will briefly listen before transmitting anything, and will postpone sending the packet if some other node is already talking. This results in fewer collisions (which means more multicast packets make it through), but the “listening” step adds a small delay to the time it takes to send each packet.

If in your network the probability of collisions is low (you don’t have much traffic), and you need the maximum throughput possible, then leave this value at its default setting of False. If in your network the probability of collisions is high (you have a lot of nodes talking a lot of the time), then you can try setting this parameter to True, and see if it helps your particular application.

ID 17 – Collision Detect

Basically, this instructs the radio to “listen after you transmit.”

This value defaults to False. Setting this value to True will cause the node to do a CCA after sending a multicast packet. This will catch some (but not all) collisions. If the node detects that some other node was transmitting at the same time, then it will resend the multicast packet. This results in more multicast packets making it through, but again at a throughput penalty.

The same criteria given for NV Parameter #16 apply to this one as well. You can try setting this parameter to True, and see if it helps your application. If not, set it back to False.

ID 18 – Collision Avoidance

This lets you control use of “random jitter” to try and reduce collisions. This setting defaults to True.

The SNAP protocol uses a “random jitter” technique to reduce the number of collisions.

Before transmitting a packet, SNAP does a small random delay. This random delay reduces the number of collisions, but increases packet latency

If you set this parameter to False, then this initial delay will not be used. This reduces latency (some extremely time critical applications need this option) but increases the chances of an over-the-air collision.

You should only change this parameter from its default setting of True if there is something else about your application that reduces the chances of collision. For example, some applications operate in a “command/response” fashion, where only one node at a time will be trying to respond anyway.

ID 19 – Radio Unicast Retries

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

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

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

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

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

ID 20 – Mesh Routing Maximum Timeout

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

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

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

You can use getInfo(14) to determine the size of a node's route table, and getInfo(15) to monitor its use.

ID 21 – Mesh Routing Minimum Timeout

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

ID 22 – Mesh Routing New Timeout

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

ID 23 – Mesh Routing Used Timeout

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

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

ID 24 – Mesh Routing Delete Timeout

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

ID 25 – Mesh Routing RREQ Retries

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

ID 26 – Mesh Routing RREQ Wait Time

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

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

ID 27 – Mesh Routing Initial Hop Limit

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

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

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

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

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

Also, although Portal (or SNAPconnect) are "one hop further away" than all other SNAP nodes on your network (they are on the other side of a "bridge" node), the SNAP code knows this, and will automatically give a "bonus hop" to this parameter's value when using it to find nodes with addresses in the reserved Portal/SNAPconnect address range of 00.00.01 – 00.00.15. So, you can leave this parameter at its default setting of 1 (one hop) even if you use Portals and/or SNAPconnects.

ID 28 – Mesh Routing Maximum Hop Limit

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

ID 29 – Mesh Sequence Number

Reserved for future use.

ID 30 – Mesh Override

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

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

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

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

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

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

ID 31 – Mesh Routing LQ Threshold

This allows for penalizing hops with poor Link Quality.

Hops that have a link quality worse than (i.e. a higher value than) the specified threshold will be counted as two hops instead of one. This allows the nodes to choose (for example) a two-hop route with good link quality over a one-hop route with poor link quality.

The default threshold setting of 127 is the highest valid value, so that no “one hop penalty” will ever be applied.

See also NV Parameters #32, #39, and #27.

ID 32 – Mesh Rejection LQ Threshold

This allows for rejecting hops with poor link quality.

Hops that have a link quality worse than (i.e. a higher value than) the specified threshold will be rejected as the node performs route requests. The default threshold setting of 127 is the highest valid value, so that all routes will be considered for mesh routing.

See also NV Parameters #31, #39, and #27.

ID 33 – Noise Floor

The Carrier Sense and Collision Detect features work by checking the current ambient signal level before broadcasting (for Carrier Sense) and immediately after broadcasting (for Collision Detect) to determine whether some other node is broadcasting. In an environment with a lot of background noise, the noise floor can trigger false positives for these features, preventing the node from broadcasting, or causing it to endlessly rebroadcast packets.

On platforms that do not allow pokes (or radioPokes) to adjust the noise floor level, NV Parameter 33 can be used to define the signal strength that must be encountered to trigger the Carrier Sense and Collision Detect features. The parameter is in negative dBm, with a range from 0 to 127. Refer to your platform's section at the end of this manual to determine whether this parameter applies to your platform.

ID 34 through 38 – Reserved for Future Use

Reserved for future Synapse use.

ID 39 – Radio LQ Threshold

This allows for ignoring packets with poor Link Quality.

Link quality values range from a theoretical 0 (perfect signal, 0 attenuation) to a theoretical 127 (127 dBm "down"). This parameter defaults to a value of 127, which makes it have no effect (you cannot receive a packet with a link quality "worse" than 127).

If you lower this parameter from its default value of 127, you are in effect defining an "acceptance criteria" on all received packets. If a packet comes in with a link quality worse (higher) than the specified threshold, then the packet will be completely ignored.

This gives you the option to ignore other nodes that are "on the edge" of radio range. The idea is that you want other (closer) nodes to take care of communicating to that node.

Caution – if you set this parameter too low, your node may not accept **any** packets.

ID 40 – SNAPpy CRC

The 16-bit Cyclic Redundancy check (CRC) of the currently loaded SNAPpy script.

Most users will not need to write to this NV parameter. If you *do* change it from its automatically calculated value, you will make the SNAP node think its copy of the SNAPpy script is invalid, and it will not use it.

ID 41 – Platform

This System NV parameter makes it easier to write scripts that work on more than one type of SNAP Node. Set this string parameter to some label that identifies your hardware platform.

New RF100 SNAP Engines from Synapse will come with "RF100" in this parameter. Older RF100 engines may have had "RFEngine" here. If you are working with SNAP-compatible radios or engines from another source, the parameter might not be loaded with any meaningful value. Furthermore, like other NV parameters the value can be changed. To make use of this field, it is the responsibility of the user to ensure that the value in the parameter is meaningful and consistent across your collection of nodes.

In your script, you must include the following line:[8]

```
from synapse.snapsys import *
```

[8] The synapse.snapsys file must be imported, but may be imported indirectly. For example, if you import synapse.platforms to get a meaningful enumeration of GPIO pins for SNAP Engines, that script already imports synapse.snapsys, so you do not need to explicitly import it separately.

When a script is loaded into a node, the script is compiled for the node. At compile time the `platform` variable is loaded with the contents of NV parameter 41, which you can use to control which other SNAPpy modules get imported or what other code will be compiled.

Because the variable is available *at compile time* (rather than only at run time), the compiler can optimize its code generation for the platform you are using, decreasing the code size and increasing the amount of space available for more complex scripts. The pinWakeup.py script, itself imported by the NewPinWakeupTest.py script, provides an example of this.

If you do not import the synapse.snapsys module, the platform variable will not be defined.

This parameter is not modified when you reset parameters to factory defaults.

ID 42 through 49 – Reserved for Future Use

Reserved for future Synapse use.

ID 50 – Enable Encryption

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

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

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

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

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

Even with a valid encryption key, encryption is not enabled until the node is rebooted. See the Encryption section in Section 6 for more details.

ID 51 – Encryption Key

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

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

Even if NV parameter #50 is set for AES-128 encryption and parameter 51 has a valid encryption key, communications will not be encrypted unless the node is loaded with a SNAP firmware image that supports AES-128 encryption. Firmware images supporting AES-128 encryption will have “AES” in their filenames.

Refer also to function getInfo() in the SNAPpy API section.

ID 52 – Lockdown

If this parameter is 0 (or never set at all), access is unrestricted. You can freely upload new scripts.

If you set this parameter to 1, and then reboot the node (as you always have to do for any NV Parameter change to take effect), then the system enters a “lockdown” mode where over-the-air script erasure or upload is disallowed.

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

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

Even in this mode, you can still perform all operations (including script upload or erasure) over the local Packet Serial link (assuming one is available). The lockdown only applies to over the air access. If you have disabled your UARTs and set this parameter, you will have to use Portal to reset your factory parameters to regain control of your node script.

ID 53 – Maximum Loyalty

This parameter, expressed in milliseconds, is valid only for the FHSS (frequency-hopping) firmware for the Si100x (including the RF300). It will be ignored on all other platforms.

After transmitting or receiving on a particular frequency, a node will wait for a signal on the next expected frequency for the duration of the loyalty period before it begins scanning all frequencies for additional communications.

If a node has transmitted or received a message and its loyalty period has not expired before it transmits its next message, it will transmit with a shorter preamble, expecting that the receiving node is listening on the appropriate channel within its own loyalty period. The shorter preamble allows for faster communications, at the risk of packets being missed by nodes that are not currently “loyal” to an expected channel.

If you adjust the loyalty period, all nodes in the network should be set to the same value. Setting the value to 0 means that no node will ever expect a loyalty period: all broadcasts will begin with a full preamble, and all radios will scan all frequencies for transmissions rather than expecting a transmission on any particular channel. The default value is 185.

ID 54 through 59 – Reserved for Future Use

Reserved for future Synapse use.

ID 60 – Last Version Booted *(Deprecated)*

At one time the system tracked this to allow "test driving" a newer demo version of the firmware, even after using up the "reboots remaining" with a previous version. The "demo policy" has since been changed such that even a reload of the *same* version of firmware restarts the countdown.

ID 61 – Reboots Remaining

In a standard (signed, non-demo) build, this parameter is always 32767 (this is the closest we could come to "infinity" in a 16-bit signed integer).

In a standard build, this value does not "count down." Normally you have unlimited reboots.

In a demo build only, this value will "count down" with every reboot. When it reaches 0, the unit will go into a "crippled" mode, in which it **will not run (or even load) SNAPpy scripts**, but can still be "pinged." (No, you cannot write to this parameter yourself to give yourself "more reboots.")

ID 62 – Reserved for Future Use

Reserved for future Synapse use.

ID 63 – Alternate Radio Trim value

This is only used on some platforms. This parameter is not modified when you reset parameters to factory defaults.

ID 64 – Vendor-Specific Settings

Similar in concept to NV parameter #11, but this field is reserved for non-standard settings. For valid values, and their exact meaning, refer to the platform specific sections in section 10 of this manual. This parameter is not modified when you reset parameters to factory defaults.

ID 65 – Clock Regulator

In platforms that have sleep modes that do not use a crystal, NV parameter 65 allows you to adjust the regulation of the internal timer that controls sleep durations. The parameter does not apply to all platforms. See the platform-specific section for your platform to determine how to best adjust this value, if necessary. This value has no effect on sleep timings that are crystal-controlled.

ID 66 – Radio Calibration Data

In platforms that require extra calibration data for proper radio operation, NV parameter 66 is used to store this calibration data. The parameter does not apply to all platforms. See the platform-specific section for your platform to determine how to best adjust this value, if necessary.

ID 67 through 127 – Reserved for Future Use

Reserved for future Synapse use.

ID 70 – Transmit Power Limit

The Transmit Power Limit is a string that specifies, channel by channel, the maximum power level that can be transmitted on each channel. The units for the setting match those for the txPwr() function, ranging from 0 through 17 (with 17 being the highest power). They represent a cap, or governor, limiting how high the output can be on the specified channel, possibly reducing the specified power if txPwr() is set higher than the channel setting specified here.

The value in the parameter is a string 16 bytes long, where the first byte represents the maximum power on channel 0, the second byte represents the maximum power on channel 1, and the 16th byte represents the maximum power on channel 15. For example, if you wanted to crank up the power to the maximum possible on all channels, you would use:

```
saveNvParam(70,'\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11')
```

This parameter is only implemented on MC1321x-based hardware.

ID 128 through 254 – Available for User Definition

These are user-defined NV Parameters, and can be used for whatever purpose you choose.

ID 255 – Reserved for Synapse Use

Reserved for internal use. (0xFF means “blank” inside the actual NV storage.)

9. 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.

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. You must not 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.
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
DarkDetector.py	Monitors a photocell via an analog input, and displays a "percent darkness" value on the seven-segment display. 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
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

Script Name	What it does
hardTime.py (synapse.hardTime.py)	Helper script useful for SNAPpy benchmarking. **NOTE –** as of version 2.4, this script imports the correct “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 other Pin Wakeup related scripts
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 correct “platform-specific” helper script.
platforms.py (synapse.platforms.py)	Import this to automatically import 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.
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, plus it goes 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 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 switchboard related constants (for readability)

Script Name	What it does
sysInfo.py	Defines some constants for the getInfo() function
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
CAT24C128.py (synapse.CAT24C128.py)	Demonstrates interfacing to a serial EEPROM chip via I^2C
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	Demo for the SN171 Proto Board in the EK2100 kit
HolidayLightShow.py	Demo 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)
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 Platform

These scripts are meant to be run on RF200 SNAP Engines (based on the ATMEL Atmega128RFA1 chip).

Script Name	**What it does**
pinWakeupATmega128RFA1.py (synapse.pinWakeupATmega128RFA1.py)	Pin Wakeup functionality specifically for nodes based on the Atmega128RFA1 chip (which includes the RF200). (Imported automatically by pinWakeup.py)
RF200.py (synapse.RF200.py)	Platform specific defines and enumerations for RF200 Engines. (Imported automatically by platforms.py)
rf200HardTime.py (synapse.rf200HardTime.py)	How to reference the clock on the RF200 SNAP Engines. (Imported automatically by hardTime.py)

Scripts specific to the RF300/RF301 Platform

These scripts 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 (synapse.pinWakeupRF300.py)	Pin Wakeup functionality specifically for the RF300 Engine. (Imported automatically by pinWakeup.py)
RF300.py (synapse.RF300.py)	Platform specific defines and enumerations for RF300 Engines. (Imported automatically by platforms.py)
rf300HardTime.py (synapse.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 (synapse.ZIC2410_PWM.py)	Support routines for accessing the two pulse-width modulation pins on a ZIC2410.
ZIC2410_SE.py (synapse.ZIC2410_SE.py)	Platform specific defines 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
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 (synapse.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.

Demos 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.
pinWakeupATmega128RFA1.py (synapse.pinWakeupATmega128RFA1.py)	Pin Wakeup functionality specifically for the ATmega128RFA1-based modules. (imported automatically by pinWakeup.py)
STK600.py (synapse.STK600.py)	Defines and LED control routines for the STK600 board. This is imported by STK60demo.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
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" capabilitites 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
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

Technique	Example scripts that demonstrate this technique
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
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
Distributing a single application across multiple nodes	DarkDetector.py + buzzer.py TemperatureAlarm.py + TemperatureAlarmBridge.py
Monitoring link quality using the getLq() function	LinkQualityRanger.py
Parsing received serial data in a SNAPpy script (contrast with Transparent Mode)	CommandLine.py gpsNmea.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
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

10. Supported Platform Details

In the remainder of this document, we present some of the low-level details of each platform (physical environment) you might be writing SNAPpy scripts for.

Some variations are due to differences in physical I/O (both quantity and capability). Each platform specific section starts with information about the physical pins.

Other variations between platforms are due to the varying amounts of RAM available. Here is a high-level overview of the types of memory management that is going on “behind the scenes.”

SNAP Buffers: The SNAP Protocol Stack uses a pool of “packet buffers.”

Buffer Budgets: The “buffer pool” is shared between the various data sources, but no single source is allowed to use up *all* of the buffers. These numbers refer to how many buffers an individual data source is allowed to request.

Dynamic Strings: Two pools of “string buffers” are used to service all the string operations.

The last variations presented are focused around differences in the various SNAPpy built-ins.

These are the currently supported platforms:
- Synapse RF100 SNAP Engine
- Freescale MC1321x chip
- Panasonic PAN4555 module
- Panasonic PAN4561 module
- Panasonic PAN4555 SNAP Engine
- Panasonic PAN4561 SNAP Engine
- CEL ZIC2410 chip
- CEL ZIC2410 SNAP Engine
- ATMEL ATmega128RFA1 chip
- Synapse RF200 SNAP Engine
- Synapse SM200 Surface Mount module
- Synapse RF266 module
- Silicon Labs Si100x chip
- Synapse RF300 SNAP Engine
- Synapse SM300 Surface Mount module
- Synapse RF301 SNAP Engine
- Synapse SM301 Surface Mount module
- Freescale MC13224 chip
- Synapse SM700 Module
- ST STM32W108CB chip
- ST STM32W108HB chip

Each of these is detailed in the following pages. Information is provided on:

- Platform specs – performance/features
- Package/pinout options – modules/carriers

Synapse RF100

The original SNAP platform, formerly referred to as the RF Engine®.

Form factor

Currently only available in the SNAP Engine form factor, the RF100 supports 19 GPIO pins (GPIO_0 to GPIO_18), each with different special abilities.

GPIO pins

Any of the 19 GPIO can be a digital input, or digital output.

Wakeup pins

Six of the 19 GPIO support a hardware "wakeup" capability; see GPIO 1, 2, 5, 6, 9 and 10.

Analog inputs

Eight of the 19 GPIO can be used as analog inputs; see GPIO 11-18 (**but notice the order in the table on the following page**).

UART0

Four pins support UART 0; see GPIO 3-6. If you do not need RTS/CTS signals, then GPIO 5 and 6 are available for other usage.

UART1

Four pins support UART 1; see GPIO 7-10. If you do not need RTS/CTS signals, then GPIO 9 and 10 are available for other usage.

CBUS

Three pins can optionally be used for CBUS; see GPIO 12-14. You will also need one "CBUS Chip Select" pin *per external CBUS device*. Any available GPIO pin can be used for this purpose.

SPI

Three pins can optionally be used for SPI; see GPIO 12-14. You will also need one "SPI Chip Select" pin *per external SPI device*. Any available GPIO pin can be used for this purpose.

I^2C

Two pins can optionally be used for I^2C; see GPIO 17 and 18.

PWM

One pin can optionally be used as a Pulse Width Modulation (PWM) output, see GPIO 0.

Seven-segment displays

The seven-segment LED displays on the SN163 and SN111 demo boards connect to GPIO 13 and 14.

Sleep Modes

There are two sleep modes available on the RF100 SNAP Engines.

0 = the radio is put completely to sleep. (The processor measures time.)

- Parameter *ticks* is in units of 1.024 seconds
- The timing in this node is much less accurate (+/- 30%)

- This mode uses less power than mode 1
- You can sleep for up to 32767 *ticks* using this mode

1 = the radio stays awake just enough to “count down” the sleep interval

- Parameter ticks is in units of 1.0 seconds
- The timing in this mode is more accurate
- This mode uses more power than mode 0
- You can sleep for up to 1073 *ticks* using this mode

Timers

Normally the RF100 uses timer 2 in the hardware for its system clock. Feature bit 0x40 of NV Parameter 11 can be set to instruct SNAP to use timer 1 instead. This affects the availability of PWM on associated pins. Refer to the Freescale documentation for details on how to make use of this.

Synapse RF100 Pin Assignments

Pin No.	Name	Description
1	GND	Power Supply
2	GPIO0_TPM1CH2	GPI/O, or Timer1 Channel 2 (ex. PWM out)
3	GPIO1_KBI0	GPI/O, Keyboard Interrupt
4	GPIO2_KBI1	GPI/O, Keyboard Interrupt
5	GPIO3_RX_UART0	GPI/O, or UART0 Data In
6	GPIO4_TX_UART0	GPI/O, or UART0 Data Out
7	GPIO5_KBI4_CTS0	GPI/O, Keyboard Interrupt, or UART0 CTS output
8	GPIO6_KBI5_RTS0	GPI/O, Keyboard Interrupt, or UART0 RTS input
9	GPIO7_RX_UART1	GPI/O, or UART1 Data In
10	GPIO8_TX_UART1	GPI/O, or UART1 Data Out
11	GPIO9_KBI6_CTS1	GPI/O, Keyboard Interrupt, or UART1 CTS output
12	GPIO10_KBI7_RTS1	GPI/O, Keyboard Interrupt, or UART1 RTS input
13	GPIO11_AD7	GPI/O or Analog In
14	GPIO12_AD6	GPI/O, Analog In, CBUS CDATA, or SPI MOSI
15	GPIO13_AD5	GPI/O, Analog In, CBUS CLK, or SPI CLK
16	GPIO14_AD4	GPI/O, Analog In, CBUS RDATA, or SPI MISO
17	GPIO15_AD3	GPI/O, or Analog In
18	GPIO16_AD2	GPI/O, or Analog In
19	GPIO17_AD1	GPI/O, Analog In, or I^2C SDA
20	GPIO18_AD0	GPI/O, Analog In, or I^2C SCL
21	VCC	Power Supply
22	PTG0/BKDG	Background Debug Communications
23	RESET*	Module Reset, Active Low
24	GND	Power Supply

SNAP Protocol Memory Usage

Global Buffer Pool: 12
UART Budget: 4
Mesh Routing Budget: 4
RPC Budget: 4
Radio Budget: 4
STDOUT Budget: 2

SNAPpy Virtual Machine Memory Usage

Number of Tiny Strings:	7
Tiny String Size:	up to 8 characters
Number of Medium Strings:	6
Medium String Size:	up to 62 characters
Global Variables:	64
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

Platform-Specific SNAPpy Built-In Functionality

Built-in function lcdPlot():
On Synapse RF100 Engines, function lcdPlot() has no effect.

Built-in function pulsePin():
On Synapse RF100 Engines, *negative* durations are in units of approximately 1.1 microsecond.

Built-in function random():
On Synapse RF100 Engines, the pseudo-random number generation is done completely in software.

Built-in function readAdc()
On the Synapse RF100 Engine, channels 0-7 correspond to one of the eight external analog input pins. Channel 8 refers to the internal high voltage reference. Channel 9 refers to the internal low voltage reference.

The mapping of Analog Input Channels to GPIO pins is as follows:

Analog Input Channel	GPIO Pin
0	18
1	17
2	16
3	15
4	14
5	13
6	12
7	11

This function returns an integer value 0-1023 (these are 10-bit analog to digital converters). Since the full-scale voltage is 3.3 volts, each step represents about 3.2 millivolts.

Built-in function setRadioRate():
On Synapse RF100 Engines, setRadioRate() has no effect. Only the standard 250 Kbps rate is supported.

Built-in function sleep():
On Synapse RF100 Engines, there are two sleep modes supported.

Mode 0 uses the internal Real Time Interrupt (RTI) as a time base. It has the lowest current consumption, but the worst accuracy (+/- 30%).

For sleep mode 0, each "tick" is normally 1.024 seconds. The exceptions are when you specify a **negative** number of ticks, as shown in the following table:

"ticks" value	Actual sleep duration
-1	8 ms
-2	32 ms
-3	64 ms
-4	128 ms
-5	256 ms
-6	512 ms
-7	1024 ms

Specifying a negative value beyond -7 is treated as a -7.

Mode 1 uses the radio as the sleep timer. Each "tick" is 1 second. Specifying a negative sleep duration when sleeping using mode 1 has no effect.

UART Performance
The minimum *bps* value that can be used on the Synapse RF100 Engine is 20 bps.

The following are the only legal combinations of *data bits*, *stop bits*, and *parity* on the Synapse RF100 SNAP Engine:

```
initUart(uart, baud) # default to 8N1
initUart(uart, baud, 8, 'N', 1) # 8 data bits, no parity
initUart(uart, baud, 8, 'E', 1) # 8 data bits, even parity
initUart(uart, baud, 8, 'O', 1) # 8 data bits, odd parity
initUart(uart, baud, 7, 'E', 1) # 7 data bits, even parity
initUart(uart, baud, 7, 'O', 1) # 7 data bits, odd parity
```

In particular, notice that 7 data bits with NO parity is not supported (hardware limitation).

Vendor-specific settings:
None as of version 2.4

Performance Metrics

Time to awaken from sleep:
< 10 milliseconds

Time to startup from power-on:
250 milliseconds

SNAP has to wait for the radio to power up.

Maximum rate a SNAPpy script can toggle a GPIO pin:
1.9 kHz

Keep in mind that as a general rule, SNAPpy scripts **should not be looping**, the 1.9 kHz rate is only attainable if the node is doing *nothing else* (for example, no radio or serial port communication).

Maximum rate for readAdc() calls:
maximum 5000 samples/second

NOTE! – This measurement was taken using a script that did not actually do anything with the data. You will also have to take into consideration any numeric processing required (data thresholding, etc.), as well as the need to actually store the data someplace.

Propagation Delay Tests
We used a pair of scripts such that one node would monitor an input pin, and when that pin changed state the node would make an RPC call to a second node. The second node (upon receiving the RPC call) would then drive one of its output pins to the same state as the first node's input pin.

Unicast Propagation Delay: Due to the collision avoidance mechanism used by SNAP, the average propagation delay is 5 milliseconds.

Multicast Propagation Delay: With a TTL = 1, the pin-to-pin propagation delay was measured at 4 milliseconds. With a TTL > 1, the RPC message incurs a >20 millisecond delay for each hop, plus local retransmit delay.

I^2C Byte Transfer Time
The actual I^2C transfers are done using "bit banging" in software. This was measured using a logic analyzer at 264 µs per byte.

SPI Byte Transfer Time
The actual SPI transfers are done using "bit banging" in software. This was measured using a logic analyzer at 140 µs per byte.

Virtual Machine Performance
Instructions Per Second (IPS): 11400

Freescale MC1321x Chip

This section applies to you if you are running SNAP on a “raw” MC1321x chip.

If you are running SNAP on a Panasonic PAN4555 or PAN4561 <u>module</u> which is based on the MC1321x, please refer instead to one of the following sections.

The MC1321x port of SNAP implements 33 “IO” pins. (Refer to the SNAP 2.2 Migration Guide if you do not understand the difference between an “IO” and a “GPIO.”)

The mapping is as follows:

PTA0-PTA7 are mapped to IO 0-7. These pins also support “wakeup” capability.

PTB0-PTB7 are mapped to IO 8-15. These pins can also be used as analog inputs.

PTC0-PTC7 are mapped to IO 16-23. PTC0/IO 16 can be used as the UART 1 TX. PTC1/IO 17 can be used as the UART 1 RX. PTC4 is used as the PA_EN (Power Amplifier Enable) signal *when enabled by the corresponding Feature Bit* (look at NV Configuration Parameter 11).

PTD2 is mapped to IO 24. This pin can also be used for full PWM.

PTD4-PTD7 are mapped to IO 25-28. These four pins can do PWM too, but **can only vary the duty cycle** – the pulse frequency rate is fixed to 1 millisecond, because by default the underlying hardware timer is providing the “1 millisecond time base” to the rest of the SNAP firmware. Setting feature bit 0x40 in NV Parameter 11 causes the system to use a different internal clock, allowing these four pins to take full advantage of PWM.

PTE0-PTE1 are mapped to IO 29-30. PTE0/IO29 can also be used as the UART 0 TX. PTE1/IO 30 can be used as the UART 0 RX.

PTG1-PTG2 are mapped to IO 31-32.

The “missing” pins PTD0, PTD1, PTD3, and PTE2-PTE7 are used *inside* the MC1321x chip to interface to the built-in radio. Since they could *never* be used for external hardware, we did not give them “IO” numbers.

Some other hardware mappings:
PTA4/IO 4 can be the CTS output for UART 0.
PTA5/IO 5 can be the RTS input for UART 0.
PTA6/IO 6 can be the CTS output for UART 1.
PTA7/IO 7 can be the RTS input for UART 1.
The emulated I^2C signals are on pins PTB0/IO 8 (SCK) and PTB1/IO 9 (SDA).
The emulated SPI signals are on pins PTG2/IO 32 (SCLK), PTG1/IO 31(MOSI), and PTD6/IO 27 (MISO).

Timers
Normally the MC1321x uses timer 2 in the hardware for its system clock. Feature bit 0x40 of NV Parameter 11 can be set to instruct SNAP to use timer 1 instead. This affects the availability of PWM on associated pins. Refer to the Freescale documentation for details on how to make use of this.

MC1321x IO Mapping

Processor Port Pin	SNAPpy IO
PTA0/KBD0	0
PTA1/KBD1	1
PTA2/KBD2	2
PTA3/KBD3	3
PTA4/KBD4	4
PTA5/KBD5	5
PTA6/KBD6	6
PTA7/KBD7	7
PTB0/AD0	8
PTB1/AD1	9
PTB2/AD2	10
PTB3/AD3	11
PTB4/AD4	12
PTB5/AD5	13
PTB6/AD6	14
PTB7/AD7	15
PTC0/TxD2	16

Processor Port Pin	SNAPpy IO
PTC1/RxD2	17
PTC2	18
PTC3	19
PTC4	20
PTC5	21
PTC6	22
PTC7	23
PTD2/TPM1CH2	24
PTD4/TPM2CH1	25
PTD5/TPM2CH2	26
PTD6/TPM2CH3	27
PTD7/TPM2CH4	28
PTE0/TxD1	29
PTE1/RxD1	30
PTG1/XTAL	31
PTG2/EXTAL	32

NOTE – in the above table we are using the chip manufacturer’s naming scheme. Because of this, the first UART is designated with a 1 and the second UART is designated with a 2. Within SNAPpy, we refer to these as UARTs 0 and 1.

SNAP Protocol Memory Usage

Global Buffer Pool:	12
UART Budget:	4
Mesh Routing Budget:	4
RPC Budget:	4
Radio Budget:	4
STDOUT Budget:	2

SNAPpy Virtual Machine Memory Usage

Number of Tiny Strings:	7
Tiny String Size:	up to 8 characters
Number of Medium Strings:	6
Medium String Size:	up to 62 characters
Global Variables:	64
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

Platform Specific SNAPpy Built-In Functionality and Performance Metrics

Because the MC1321x chip contains essentially the same 9S08 based processor as the one used on the Synapse RF100 SNAP Engine, it matches the Synapse RF100 Engine in capability. Refer to the appropriate sections in the **RF100 Engine** portion of the document.

Panasonic PAN4555 SNAP Module

Because it is based on the Freescale MC1321x chip, the PAN4555 wireless module can also run SNAP.

NOTE – If you are using a <u>SNAP Engine</u> *based on* the PAN4555 module, skip ahead to the next section.

This section is for users putting the PAN4555 module directly down on their board. Because the hardware is not in the SNAP Engine form factor, there is no such concept as GPIO. You should write your script using plain "IO" numbering (Refer to the SNAP 2.2 Migration Guide if you don't know what this means).

It **is** important to note that not all of the MC1321x pins are brought out on the PAN4555 module. The table on the following page summarizes the correspondence between SNAPpy IO, module pin, and the underlying processor pin.

Timers

Normally the PAN4555 uses timer 2 in the hardware for its system clock. Feature bit 0x40 of NV Parameter 11 can be set to instruct SNAP to use timer 1 instead. This affects the availability of PWM on associated pins. Refer to the Freescale documentation for details on how to make use of this.

PAN4555 Module IO Mapping

PAN4555 Module Pin Number	PAN4555 Module Pin Name	SNAPpy IO Number
1, 9, 17, 25, 31	GND	N/A
2	PTB0	8
3	PTB1	9
4	PTB2	10
5	PTB7	15
6	VREF	N/A
7	PTA7	7
8	PTA5	5
10	PTA6	6
11	PTG0/BKGD	N/A
12	PTG1	31
13	PTG2	32
14	CLKO	N/A
15	PTC0	16
16	PTC1	17
18	PTC5	21
19	PTC3	19
20	PTC2	18
21	PTE0	29
22	PTE1	30
23	VDDA	N/A
24, 26	VCC	N/A
27	RESET	N/A
28	PTD6	27
29	PTD4	25
30	PTD2	24
32	EXTANT	N/A

Panasonic PAN4555 (SNAP Engine Form Factor)

In addition to the existing line of Synapse RF Engines, SNAP 2.2 is also available as a SNAP Engine based on Panasonic's PAN4555 module. Like the other SNAP Engines, this PAN4555 board has 24 pins, and supports 19 GPIO. These two types of modules are largely interchangeable. However, there are a few functional differences to be aware of:

Fewer "Wakeup" Pins

On a Synapse RF100 Engine, GPIO pins GPIO1, GPIO2, and GPIO5 can be used to wake the module from "sleep mode." On a Panasonic PAN4555 Wireless Module, GPIO pins 1, 2, and 5 <u>cannot</u> wake the processor. Note that GPIO pins 1, 2, and 5 *can* be used as inputs, and they *can* be monitored. Only the "wakeup" functionality is missing.

GPIO pins 6, 9, and 10 *can* be used to wake from sleep mode on both Synapse RFEs and PAN4555 wireless modules.

Fewer ADC Input Pins

On a Synapse RF100 Engine, GPIO pins GPIO11 through GPIO18 can all be used as Analog to Digital Converter (ADC) inputs. On the PAN4555 Wireless Module, only GPIO 11, 16, 17, and 18 support ADC. This means only ADC channels 0, 1, 2, and 7 provide live readings.

You cannot "cheat" and read/write 8 GPIO with a single poke()

On a Synapse RF100 Engine, GPIO pins GPIO11 through 18 are all mapped to the *same* I/O register on the microcontroller. This means these pins can be used to easily implement an 8-bit wide data bus (see for example script "lcd8bit.py"). On the Panasonic PAN4555, the 4 missing ADC pins (mentioned above) affect this "data bus" as well. You can still write to the 8-bit wide data register, but only 4 of the pins controlled by that register are actually brought out to the real world.

Two Additional PWM Output Pins

On a Synapse RF100 Engine, only GPIO 0 can perform Pulse Width Modulation (PWM). On the PAN4555 Wireless Module, GPIO pins 14 and 15 can also do <u>limited</u> PWM.

The PWM limitation on these two pins (GPIO 14 and 15) has to do with the frequency of the pulse that can be modulated. On these two pins, the pulses always occur every 1 millisecond. SNAPpy scripts can affect the width of the pulses, but not the rate at which they occur.

Refer to example script PAN4555_ledCycling.py for one example of using these additional PWM pins.

If you need a pulse rate different than 1 per millisecond (for example, you are doing servo motor control), you will have to use GPIO 0.

getInfo() Differences

On a getInfo(0) call, the parameter value of 0 requests a "vendor code."

On a Synapse RFE, getInfo(0) returns 0 (meaning "Synapse").

On a PAN4555, getInfo(0) returns 2 (meaning "Freescale").

On a getInfo(3) call, the parameter value of 3 requests a "platform code." The PAN4555 returns a value of 5, indicating MC1321x (the chipset the PAN4555 is based on).

SNAPpy scripts can use the getInfo() function to adapt themselves to the board they find themselves running on. **See also section 5 of this document, where an alternate method is explained.**

Sleep() considerations

For most efficient sleeping, it is important for all the chip's pins to be configured before sleeping – not just the pins brought out by the SNAP Engine. Refer to the `initUnusedIO()` function in the synapse.PAN4555.py sample script as an example of how to do this.

For Advanced Users Only

Here are the exact pin changes from a Synapse RF100 to a Panasonic PAN4555 when using the SNAPpy GPIO import scheme.

SNAPpy GPIO	Processor pin used on RFE	Processor pin used on PAN4555
1	Port A Bit 0	Port C Bit 3
2	Port A Bit 1	Port C Bit 2
5	Port A Bit 4	Port C Bit 5
12	Port B Bit 6	Port G Bit 1
13	Port B Bit 5	Port G Bit 2
14	Port B Bit 4	Port D Bit 6
15	Port B Bit 3	Port D Bit 4

Pin Configuration of a PAN4555 in SNAP Engine Format

Pins that differ from Synapse RF100 Engines are highlighted in **bold**.

Pin No.	Name	Description
1	GND	Power Supply
2	GPIO0_TPM1CH2	GPI/O or Timer1 Channel 2 (ex. PWM out)
3	**GPIO1**	**GPI/O**
4	**GPIO2**	**GPI/O**
5	GPIO3_RX_UART0	GPI/O or UART0 Data In
6	GPIO4_TX_UART0	GPI/O or UART0 Data Out
7	**GPIO5_CTS0**	**GPI/O or UART0 CTS output**
8	GPIO6_KBI5_RTS0	GPI/O, Keyboard Interrupt, or UART0 RTS input
9	GPIO7_RX_UART1	GPI/O or UART1 Data In
10	GPIO8_TX_UART1	GPI/O or UART1 Data Out
11	GPIO9_KBI6_CTS1	GPI/O, Keyboard Interrupt, or UART1 CTS output
12	GPIO10_KBI7_RTS1	GPI/O, Keyboard Interrupt, or UART1 RTS input
13	GPIO11_AD7	GPI/O or Analog In
14	**GPIO12**	**GPI/O, CBUS CDATA, or SPI MOSI**
15	**GPIO13**	**GPI/O, CBUS CLK, or SPI CLK**
16	**GPIO14_TPM2CH3**	**GPI/O, CBUS RDATA, SPI MISO or Timer2 Channel 3 (ex. limited PWM out)**
17	**GPIO15_TPM2CH1**	**GPI/O, or Timer2 Channel 1 (ex. limited PWM out)**
18	GPIO16_AD2	GPI/O, or Analog In
19	GPIO17_AD1	GPI/O, Analog In, or I^2C SDA
20	GPIO18_AD0	GPI/O, Analog In, or I^2C SCL
21	VCC	Power Supply
22	PTG0/BKDG	Background Debug Communications
23	RESET*	Module Reset, Active Low
24	GND	Power Supply

PAN4555 GPIO Assignments

(GPIO assignments defined in PAN4555_SE.py)

SNAPpy GPIO	Processor Port	PAN4555 Module Pin	SNAPpy IO Num
GPIO_0	PTD2/TPM1CH2	30	24
GPIO_1	PTC3/SCL	19	19
GPIO_2	PTC2/SDA	20	18
GPIO_3	PTE1/RxD1	22	30
GPIO_4	PTE0/TxD1	21	29
GPIO_5	PTC5	18	21
GPIO_6	PTA5/KBD5	8	5
GPIO_7	PTC1/RxD2	16	17
GPIO_8	PTC0/TxD2	15	16
GPIO_9	PTA6/KBD6	10	6
GPIO_10	PTA7/KBD7	7	7
GPIO_11	PTB7/AD7	5	15
GPIO_12	PTG1/XTAL	12	31
GPIO_13	PTG2/EXTAL	13	32
GPIO_14	PTD6/TPM2CH3	28	27
GPIO_15	PTD4/TPM2CH1	29	25
GPIO_16	PTB2/AD2	4	10
GPIO_17	PTB1/AD1	3	9
GPIO_18	PTB0/AD0	2	8

Vendor-specific settings:
None as of version 2.4.9

Performance Metrics

Because the PAN4555 module contains at its heart essentially the same 9S08 based processor as the one used on the Synapse RF100 Engine, you can use the performance metrics given in the **RF100 Engine** section of the document.

Panasonic PAN4561 (SNAP Engine Form Factor)

The PAN4561 utilizes the same core processor as the original RFE and PAN4555 (the Freescale HCS08). In fact, the PAN4561 utilizes the same MC13213 integrated IC (HCS08 plus radio front end) as the PAN4555.

As such, the PAN4561 will use a firmware version designed for all modules based on the Freescale MC1321x series of chipsets. This same firmware will run on the PAN4555, PAN4561, etc.

The version of SNAP software developed for the PAN4561 follows the same GPIO structure of the PAN4555 for the first 18 GPIO pins.

However, there are a few functional details to be aware of:

Increased Number of GPIO Pins

The PAN4561 has a total of 33 available GPIO pins. These include:

- 8 Analog-to-Digital (ADC) pins
- UARTs
- 8 Keyboard Interrupt pins (KBI)
- Pulse-Width Modulated (PWM) output pins

Platform Specific Settings

SNAPpy NV-Param #41 is used to store the platform name. For this module the name is ‘PAN4561’. This value can be modified by the user. However, this has the potential to affect SNAPpy script functionality. The platform name will normally be set during the initial programming of the module at the factory.

SNAPpy NV-Param #63 stores the trim value for the radio transceiver’s crystal oscillator (not to be confused with the separate trim value related to the HCS08 MCU). If this value is not set, then the radio will be configured to use the default value- typically 0x7E (see MC1321x reference manual for details). The trim value will be set during the initial programming of the module at the factory.

SNAP feature bits are used to control a number of things, including module specific settings. For the PAN4561 mounted onto a SNAPpy Engine carrier, a feature bit has be used to specify that a power amplifier exists and that this PA will be disabled before the module enters into a sleep state (and re-enabled upon waking). The 6^{th} bit (0x20) of Non-Volatile Parameter (NV-Param) #11 is used to store this particular setting. The appropriate feature bits will be set during initial programming at the factory. This setting has no effect unless the module is mounted on a SNAP Engine carrier board or Pin 52 and 45 are tied together (See following section for details).

None of these values will be reset when executing the ‘Factory Default NV Params’ menu option within the Portal software.

Platform Specific Hardware Configuration

Low-Noise Amplifier High-Gain Mode (HGM) => Radio Transceiver GPIO- 3:
The LNA located on the PAN4561 module supports a high-gain mode. This can be enabled or disabled by using GPIO-3 of the MC1321x's radio transceiver.

HGM State	GPIO-3
Enabled	High
Disabled	Low

This GPIO pin can be read and/or controlled by using SNAP's peekRadio() and pokeRadio() built-in functions. This pin is driven high during the startup process when SNAP Feature Bit 5 (0x20) is set.

Refer to the reference manual for the MC1321x and PAN4561 for details regarding memory locations and pin behavior.

Power Amplifier Enable/Disable (PA_EN)
The Power Amplifier located on the PAN 4561 module can be enabled or disabled by manipulating external pin 52. Setting this pin high will enable the Power Amplifier.

PA State	Pin 52
Enabled	High
Disabled	Low

NOTE: When the PAN4561 is mounted onto a SNAP Engine carrier board, pin 52 and pin 45 (SNAPpy Raw IO 20/ PTC4) are connected together. This allows the PA to be enabled/disabled by manipulating SNAPpy IO 20 (PTC4). This pin is driven high during the startup process when SNAP Feature Bit 5 (0x20) is set.

Timers
Normally the PAN4561 uses timer 2 in the hardware for its system clock. Feature bit 0x40 of NV Parameter 11 can be set to instruct SNAP to use timer 1 instead. This affects the availability of PWM on associated pins. Refer to the Freescale documentation for details on how to make use of this.

ADC Pins
8 ADC (Analog-to-Digital) pins are available for use on the PAN 4561. This is like the Synapse RF100 SNAP Engines, but dislike the PAN4555 that only brings out 4 ADC pins.

Low Power Settings (LNA/PA)
The PAN4561 module includes a Low-Noise Amplifier (for receiving) and Power Amplifier (for transmitting). This power amplifier needs to be placed in a disabled mode in order for the entire module to reach low power operation. The SNAP core will disable the PA during the sleep process and will re-enable once the system has emerged from sleep. The user does not need to take any action.

The LNA is set to a high-gain mode and the PA is enabled by default during startup when feature bit 5 (0x20) is set and module pin 45 is tied to pin 52.

Default UART remains UART1

The default UART is still designated as UART1. This is consistent with SNAP ports to other devices.

I^{2}C Emulation vs. Hardware pins

The hardware I^{2}C pins designated as SCL and SDA are assigned to GPIO pins 1 and 2 respectively. However, hardware I^{2}C is not currently enabled within the SNAP core. Instead, the pins associated with SNAPs I^{2}C software emulation (GPIO 17 and 18) remain unchanged from the Synapse RFE and PAN4555.

Additional PWM Output Pins

PWM on GPIO 0, 14, 15, 31, and 32

On a Synapse RF Engine, only GPIO 0 can perform Pulse Width Modulation (PWM). On the PAN4555 Wireless Module, GPIO pins 14 and 15 can also do limited PWM. On the PAN4561 GPIO pins 31 and 32 can also do limited PWM.

The PWM limitation on these four pins (GPIO 14, 15, 31, 32) has to do with the frequency of the pulse that can be modulated. On these pins, the pulses occur every 1 millisecond. SNAPpy scripts can affect the width of the pulses, but not the rate at which they occur. If you need a pulse rate different than 1 per millisecond (for example, you are doing servo motor control), you will have to use GPIO 0.

Refer to example script PAN4561_ledCycling.py for one example of using some of these additional PWM pins.

getInfo() Differences

A call to the SNAP function getInfo() with a parameter value of 0 will request a “vendor code.”

On a Synapse RFE, getInfo(0) returns 0 (meaning “Synapse”).

On a PAN4561, getInfo(0) returns 2 (meaning “Freescale”).

On a getInfo(3) call, the parameter value of 3 requests a “platform code.” The PAN4561 returns a value of 5, indicating MC1321x (the chipset the PAN4561 is based on).

SNAPpy scripts can use the getInfo() function to adapt themselves to the board they find themselves running on. **See also section 5 of this document, where an alternate method is explained.**

Please refer to the PAN 4561 Product Specification from Panasonic and the MC1321x Reference Manual from Freescale for more information regarding pin and module functionality.

PAN4561 GPIO Assignments

(GPIO assignments defined in PAN4561_SE.py)

Bold indicates the pins that are the same on the PAN 4555 and 4561

SNAPpy GPIO	Processor Port	PAN4561 Pin	SNAPpy IO Num
GPIO_0	**PTD2/TPM1CH2**	**6**	**24**
GPIO_1	**PTC3/SCL**	**11**	**19**
GPIO_2	**PTC2/SDA**	**10**	**18**
GPIO_3	**PTE1/RxD1**	**47**	**30**
GPIO_4	**PTE0/TxD1**	**46**	**29**
GPIO_5	**PTC5**	**44**	**21**
GPIO_6	**PTA5/KBD5**	**36**	**5**
GPIO_7	**PTC1/RxD2**	**9**	**17**
GPIO_8	**PTC0/TxD2**	**8**	**16**
GPIO_9	**PTA6/KBD6**	**35**	**6**
GPIO_10	**PTA7/KBD7**	**34**	**7**
GPIO_11	**PTB7/AD7**	**19**	**15**
GPIO_12	**PTG1/XTAL**	**25**	**31**
GPIO_13	**PTG2/EXTAL**	**26**	**32**
GPIO_14	**PTD6/TPM2CH3**	**4**	**27**
GPIO_15	**PTD4/TPM2CH1**	**2**	**25**
GPIO_16	**PTB2/AD2**	**14**	**10**
GPIO_17	**PTB1/AD1**	**13**	**9**
GPIO_18	**PTB0/AD0**	**12**	**8**
GPIO_19	PTA0/KBD0	41	0
GPIO_20	PTA1/KBD1	40	1
GPIO_21	PTA2/KBD2	39	2
GPIO_22	PTA3/KBD3	38	3
GPIO_23	PTA4/KBD4	37	4
GPIO_24	PTB3/AD3	15	11
GPIO_25	PTB4/AD4	16	12
GPIO_26	PTB5/AD5	17	13
GPIO_27	PTB6/AD6	18	14
GPIO_28	PTC4	45	20
GPIO_29	PTC6	43	22
GPIO_30	PTC7	42	23
GPIO_31	PTD5/TPM2CH2	3	26
GPIO_32	PTD7/TPM2CH4	5	28

Pin Functionality for the PAN4561 Module

Processor Port	PAN4561 Pin	SNAPpy IO Num	SNAPpy GPIO
PTA0/KBD0	41	0	GPIO_19
PTA1/KBD1	40	1	GPIO_20
PTA2/KBD2	39	2	GPIO_21
PTA3/KBD3	38	3	GPIO_22
PTA4/KBD4	37	4	GPIO_23
PTA5/KBD5	36	5	GPIO_6
PTA6/KBD6	35	6	GPIO_9
PTA7/KBD7	34	7	GPIO_10
PTB0/AD0	12	8	GPIO_18
PTB1/AD1	13	9	GPIO_17
PTB2/AD2	14	10	GPIO_16
PTB3/AD3	15	11	GPIO_24
PTB4/AD4	16	12	GPIO_25
PTB5/AD5	17	13	GPIO_26
PTB6/AD6	18	14	GPIO_27
PTB7/AD7	19	15	GPIO_11
PTC0/TxD2	8	16	GPIO_8
PTC1/RxD2	9	17	GPIO_7
PTC2/SDA	10	18	GPIO_2
PTC3/SCL	11	19	GPIO_1
PTC4	45	20	GPIO_28
PTC5	44	21	GPIO_5
PTC6	43	22	GPIO_29
PTC7	42	23	GPIO_30
PTD2/TPM1CH2	6	24	GPIO_0
PTD4/TPM2CH1	2	25	GPIO_15
PTD5/TPM2CH2	3	26	GPIO_31
PTD6/TPM2CH3	4	27	GPIO_14
PTD7/TPM2CH4	5	28	GPIO_32
PTE0/TxD1	46	29	GPIO_4
PTE1/RxD1	47	30	GPIO_3
PTG1/XTAL	25	31	GPIO_12
PTG2/EXTAL	26	32	GPIO_13

Another view of the same data (PAN4561)

PAN4561 Pin	Processor Port	SNAPpy IO Num	SNAPpy GPIO
2	PTD4/TPM2CH1	25	GPIO_15
3	PTD5/TPM2CH2	26	GPIO_31
4	PTD6/TPM2CH3	27	GPIO_14
5	PTD7/TPM2CH4	28	GPIO_32
6	PTD2/TPM1CH2	24	GPIO_0
8	PTC0/TxD2	16	GPIO_8
9	PTC1/RxD2	17	GPIO_7
10	PTC2/SDA	18	GPIO_2
11	PTC3/SCL	19	GPIO_1
12	PTB0/AD0	8	GPIO_18
13	PTB1/AD1	9	GPIO_17
14	PTB2/AD2	10	GPIO_16
15	PTB3/AD3	11	GPIO_24
16	PTB4/AD4	12	GPIO_25
17	PTB5/AD5	13	GPIO_26
18	PTB6/AD6	14	GPIO_27
19	PTB7/AD7	15	GPIO_11
25	PTG1/XTAL	31	GPIO_12
26	PTG2/EXTAL	32	GPIO_13
34	PTA7/KBD7	7	GPIO_10
35	PTA6/KBD6	6	GPIO_9
36	PTA5/KBD5	5	GPIO_6
37	PTA4/KBD4	4	GPIO_23
38	PTA3/KBD3	3	GPIO_22
39	PTA2/KBD2	2	GPIO_21
40	PTA1/KBD1	1	GPIO_20
41	PTA0/KBD0	0	GPIO_19
42	PTC7	23	GPIO_30
43	PTC6	22	GPIO_29
44	PTC5	21	GPIO_5
45	PTC4	20	GPIO_28
46	PTE0/TxD1	29	GPIO_4
47	PTE1/RxD1	30	GPIO_3

Pin Configuration of a PAN4561 in SNAP Engine Format

Pins that differ from Synapse RF Engines are highlighted in **bold**.

Pin No.	Name	Description
1	GND	Power Supply
2	GPIO0_TPM1CH2	GPI/O or Timer1 Channel 2 (ex. PWM out)
3	**GPIO1**	**GPI/O**
4	**GPIO2**	**GPI/O**
5	GPIO3_RX_UART0	GPI/O or UART0 Data In
6	GPIO4_TX_UART0	GPI/O or UART0 Data Out
7	**GPIO5_CTS0**	**GPI/O or UART0 CTS output**
8	GPIO6_KBI5_RTS0	GPI/O, Keyboard Interrupt, or UART0 RTS input
9	GPIO7_RX_UART1	GPI/O or UART1 Data In
10	GPIO8_TX_UART1	GPI/O or UART1 Data Out
11	GPIO9_KBI6_CTS1	GPI/O, Keyboard Interrupt, or UART1 CTS output
12	GPIO10_KBI7_RTS1	GPI/O, Keyboard Interrupt, or UART1 RTS input
13	GPIO11_AD7	GPI/O or Analog In
14	**GPIO12**	**GPI/O, CBUS CDATA, or SPI MOSI**
15	**GPIO13**	**GPI/O, CBUS CLK, or SPI CLK**
16	**GPIO14_TPM2CH3**	**GPI/O, CBUS RDATA, SPI MISO or Timer2 Channel 3 (ex. limited PWM out)**
17	**GPIO15_TPM2CH1**	**GPI/O, or Timer2 Channel 1 (ex. limited PWM out)**
18	GPIO16_AD2	GPI/O, or Analog In
19	GPIO17_AD1	GPI/O, Analog In, or I^2C SDA
20	GPIO18_AD0	GPI/O, Analog In, or I^2C SCL
21	VCC	Power Supply
22	PTG0/BKDG	Background Debug Communications
23	RESET*	Module Reset, Active Low
24	GND	Power Supply

Vendor-specific settings:
NV Parameter 64 is used on the PAN4561.

Bit 0x0001 – Indicates PAN4561 module is older (Rev B) hardware, and requires special handling for I/O differences.

Users should verify this bit is set to 1 for the older units, and set to 0 for the newer (Rev C) units.

Performance Metrics

Because the PAN4561 module contains at its heart essentially the same 9S08 based processor as the one used on the Synapse RF Engine, you can use the performance metrics given in the **RF Engine** section of the document.

California Eastern Labs ZIC2410 Chip and Module

In addition to modules built on a SNAP Engine footprint, you will find SNAP running on CEL *chips* (ZIC2410) and *modules* (ZICM2410P0, without a power amplifier, and ZICM2410P2, with a power amplifier). Versions with the power amplifier require SNAP firmware version 2.2.16 or higher. See the following section for details unique to the SNAP Engine form factor.

The default setting for NV parameters is to assume that the power amplifier is available. Users running on the original (non-PA) hardware will need to clear the "PA" Feature Bit (0x10). SNAP modules with the hardware amplifier will be labeled ZICM2410P2, while those without the amplifier will be labeled ZICM2410P0.

For example, if your feature bits are currently 0x1F, then a saveNvParam(11, 0x0F) will clear the "PA" bit.

The following table summarizes the IO mapping on the ZIC2410 chip.

ZIC2410 IO Mapping

Processor Port Pin (P0)	SNAPpy IO
P0.0	0
P0.1	1
P0.2	2
P0.3	3
P0.4	4
P0.5	5
P0.6	6
P0.7	7

Processor Port Pin (P1)	SNAPpy IO
P1.0/RXD1	8
P1.1/TXD1	9
P1.2	10
P1.3	11
P1.4	12
P1.5*	13
P1.6*	14
P1.7	15

Processor Port Pin (P3)	SNAPpy IO
P3.0/RXD0	16
P3.1/TXD0	17
P3.2/INT0	18
P3.3/INT1	19
P3.4/RTS0/ SPIDI	20
P3.5/CTS0/ SPIDO	21
P3.6/RTS1/ PWM2/SPICLK	22
P3.7/CTS1/ PWM3	23

The same IO numbering scheme applies to the ZICM2410P0 and ZICM2410P2 *modules*, but be aware that not all of the chip's pins are brought out of the module. Specifically, **P1.2 (IO 10)** and **P1.5 (IO 13)** are not brought out, and so cannot be used in your module scripts. (You *can* use these pins if you are running SNAP on the bare ZIC2410 chip.)

Additionally, **P1.6 (IO 14)** and **P1.7 (IO 15)** are available on the ZICM2410P0 modules, but are not connected on the ZICM2410P2 modules, as they are used internally to control the power amplifier.

Separate Analog Input Pins

Unlike many of the SNAP platforms, the analog input pins on the ZIC2410 do not overlap any of the digital IO or other peripherals. You still use the readAdc(*channel*) SNAPpy built-in to access the four ACHx channels on the ZIC2410.

I^2C Emulation

The ZIC2410 has no I2C hardware, but SNAP emulates I^2C (**I^2C master only**) in software, using the following two pins:

I^2C SDA is emulated using P1.3 / SNAPpy IO 11

I^2C CLK is emulated using P1.4 / SNAPpy IO 12

Please refer to the CEL ZIC2410 and ZICM2410 data sheets for more information on the pinouts and capabilities of these parts.

Memory Usage

In the current version of ZIC2410 code, AES-128 encryption support has an impact on the amount of RAM available.

Memory Usage without AES-128 Support

Here are the settings in ZIC2410 builds *without* AES-128 encryption support:

SNAP Protocol Memory Usage:

Global Buffer Pool:	20
UART Budget:	6
Mesh Routing Budget:	6
RPC Budget:	6
Radio Budget:	6
STDOUT Budget:	4

SNAPpy Virtual Machine Memory Usage:

Number of Tiny Strings:	14
Tiny String Size:	up to 16 characters
Number of Medium Strings:	8
Medium String Size:	up to 126 characters
Global Variables:	64
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

Memory Usage with AES-128 Support
Here are the settings in ZIC2410 builds *with* AES-128 encryption support:

SNAP Protocol Memory Usage:

Global Buffer Pool:	16
UART Budget:	6
Mesh Routing Budget:	6
RPC Budget:	6
Radio Budget:	6
STDOUT Budget:	4

SNAPpy Virtual Machine Memory Usage:

Number of Tiny Strings:	7
Tiny String Size:	8 characters
Number of Medium Strings:	6
Medium String Size:	62 characters
Global Variables:	64
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

Platform Specific SNAPpy Functionality

Audio Enable:
Feature bit 0x80 in NV Parameter 11 is available on the ZIC2410, allowing two ZIC2410-based SNAP nodes to transmit and receive audio data across a SNAP network. Each end of such a network must have this feature enabled, though any other nodes used to form a mesh network to forward these communications do not have to be enabled.

Carrier Sense:
The Carrier Sense function (NV Parameter 16) does not affect nodes based on the ZIC2410. This platform has this type of functionality built into the hardware.

Built-in functions cbusRd**() and** cbusWr**():**
These functions have no effect on the ZIC2410.

Built-in function lcdPlot**():**
Built-in function lcdPlot() is fully functional in the ZIC2410 builds. For it to be of any use, you must connect the same type of LCD as used on the CEL EVB1 Evaluation Board, and you must wire it up in the same way. Refer to the CEL EVB1 Data Sheets for more information.

Built-in functions peek**() and** poke**():**
Use negative address values to peek and poke special function registers. See the peek() and poke() description for more details.

Built-in functions peekRadio**() and** pokeRadio**():**
These functions are not necessary on the ZIC2410. The internal radio registers are in the main memory space of the ZIC2410, and can be read or written using the regular peek() and poke() functions.

Built-in function pulsePin**():**
On the ZIC2410, *negative* durations are in units of approximately 1.63 microsecond.

Built-in function random**():**
On the ZIC2410, the pseudo random number generation is done in hardware, not software.

Built-in function readAdc**()**
On the ZIC2410, channels 0-3 correspond to direct reads of one of the four external analog input pins. Channels 4 and 5 are differential versions of the channel 0/1 pair, and the channel 2/3 pair. Channel 6 is based on the internal temperature of the ZIC2410. Channel 7 is connected to an internal battery monitor circuit. Channel 8 refers to the internal high voltage reference. Channel 9 refers to the internal low voltage reference.

This function normally returns an integer value 0-255 (these are 8-bit analog to digital converters).

The ZIC2410 hardware also supports reading an uncalibrated 16-bit value. You can take these alternate readings by adding 10 to the channel number.

The following table summarizes the ADC related options on the ZIC2410:

Channel Meaning	**Channel Number for 8-bit calibrated read**	**Channel Number for 16-bit uncalibrated read**
ACH0	0	10
ACH1	1	11
ACH2	2	12
ACH3	3	13
Differential read from ACH0/ACH1	4	14
Differential reading from ACH2/ACH3	5	15
Internal Temperature	6	16
Battery Monitor	7	17
High Reference	8	18
Low Reference	9	19

Refer to the ZIC2410 datasheets for more information.

NOTE – The ADC on this chip is 1.5 volts. You should not scale it off VCC.

NOTE – SNAPpy only supports signed integers (-32768 to 32767), so you will need to account for that when performing calculations from raw sensor readings > 32767.

A raw reading of 0x8000 will look like -32768 to SNAPpy. 0x8001 will be interpreted as -32767, 0xFFFE will be -2, and 0xFFFF will be -1.

To compensate, you will likely need to adjust for this at the point you try to actually *use* the results of readAdc(10) – readAdc(19).

For example, in your Portal script you might need a function like the following:

```
def signedToUnsigned(value):
    if value > 0:
        return value # already a positive number, no adjustment needed

    unsignedValue = 65535 + value + 1
    # if you are on a processor with >= 32-bit integers, you can just say 65536 +
value

    return unsignedValue
```

In other programming languages, like C or C++, you can do what is called a cast:
`adjustedValue = (unsigned int) signedValue.`

Built-in function setRadioRate():
On the ZIC2410, setRadioRate() supports values of 0, 1, and 2. The resulting data rates are as follows:

setRadioRate() parameter	Radio Data Rate
0	250 Kbps
1	500 Kbps
2	1 Mbps

Radio rate 0 is compatible with any 802.15.4 SNAP radio using rate 0. Any other rate may not work with any non-ZIC2410 radio, and will only work with a ZIC2410 radio working at the same rate.

Built-in function sleep**():**
On ZIC2410, there are three sleep modes supported. Unlike on some of the other platforms, these sleep modes do not differ in accuracy or tick units. Instead they differ in how they treat I/O pins, how they recover from sleep, and how much current they draw while sleeping.

Mode 0 maintains all I/O pins while sleeping, and wakes up normally after the requested seconds have elapsed, or an external interrupt occurs.

Mode 1 *does not* maintain I/O while sleeping, and instead of waking up, it completely *reboots* after the requested seconds have elapsed, or an external interrupt occurs.

Mode 2 *does not* maintain I/O while sleeping, and also *reboots* instead of truly waking up. In addition, this sleep mode does not support timed sleep. The only way to exit a mode 2 sleep is via one of the external interrupt pins (EXT0 or EXT1).

NOTE – The ZICM2410P2, which includes a power amplifier, requires that you have P3.2 (pin 18) and P3.3 (pin 19) pulled high (CEL recommends a 10 KΩ pull-up resistor between VCC and the pins) in order to prevent internal interrupts from waking the module prematurely when sleep modes 1 and 2 are used.

For more information on the sleep modes of the ZIC2410, refer to the manufacturers data sheets.

In each mode, one tick is one second. The maximum sleep duration for timed sleep on the ZIC2410 is 256 seconds. Time values larger than 256 will be reduced by modulo 256 (e.g., values of 258 or 514 would both result in 2-second sleeps). No "negative" sleep durations are supported.

NOTE – The ZIC2410 data sheets refer to 4 "power modes," where 0 is awake, and "power modes" 1-3 correspond to SNAPpy sleep modes 0-2.

Performance Metrics

Here are the results of some performance measurements, which may help you gauge if SNAPpy can address your application's timing requirements.

These results are for the California Eastern Labs ZIC2410.

Time to awaken from sleep (mode 0):
< 200 microseconds

Time to startup from power-on:
< 60 milliseconds

Maximum rate a SNAPpy script can toggle a GPIO pin:
528.5 Hz

Keep in mind that as a general rule, SNAPpy scripts **should not be looping**, the above rate is only attainable if the node is doing *nothing else* (for example, no radio or serial port communication).

Maximum rate for readAdc() calls:
maximum 696 samples/second

NOTE! – This measurement was taken using a script that did not actually do anything with the data. You will also have to take into consideration any numeric processing required (data thresholding, etc.), as well as the need to actually store the data someplace.

I^{2}C Byte Transfer Time
The actual I^{2}C transfers are done using "bit banging" in software. This was measured using a logic analyzer at 710 µs per byte.

SPI Byte Transfer Time

The actual SPI transfers are done using “bit banging” in software. This was measured using a logic analyzer at 1600 µs (1.6 ms) per byte.

Virtual Machine Speed

SNAP 2.2 Instructions Per Second (IPS): 1770

California Eastern Labs ZIC2410 (SNAP Engine Form Factor)

SNAP Engines based on the ZICM2410P2 are available. All the details appropriate for the chip- and module-based SNAP Modules apply to module-based SNAP Engines, with the following additions.

SNAP Engines based on the ZIC2410 should not be used on the Synapse SN111 End Device board. The power-up state of the pins conflicts with the relay controls on that board.

Just as the module exposes fewer pins than the chip, the SNAP Engine based on the module is further reduced. Pin 10 and pins 12 through 15 are not brought out to GPIO pins on the SNAP Engine footprint.

Separate Analog Input Pins

Unlike many of the SNAP platforms, the analog input pins on the ZIC2410 do not overlap any of the digital IO or other peripherals. On SNAP Engines based on the ZICM2410, the analog pins are not available by default. You must enable each analog pin you want to use by jumpering one pair of pads and cutting the trace between another pair of pads on the underside of the SNAP Engine, which disables the associated pin as a digital I/O.

Analog Channel	SNAP Engine GPIO Pin	Short Pad	Cut Trace on Pad
ACH0	Pin 14	R1	R6
ACH1	Pin 13	R2	R7
ACH2	Pin 12	R3	R8
ACH3	Pin 11	R4	R5

Once enabled on the hardware, you can read the analog channel as an 8-bit calibrated value or a 16-bit uncalibrated value, just as you can with the modules when they are not on the SNAP Engine form factor.

Pin Configuration of a ZICM2410P2 in SNAP Engine Format

Engine Pin No.	SNAPpy IO No.	Name	Description
1		GND	Power Supply
2	11	GPIO0_P1.3	GPIO_0
3	18	GPIO1_INT0_P3.2	GPIO_1 or interrupt
4	19	GPIO2_INT1_P3.3	GPIO_2 or interrupt
5	16	GPIO3_RXD0_UART0_P3.0	GPIO_3 or UART0 Data In
6	17	GPIO4_TXD0_UART0_P3.1	GPIO_4 or UART0 Data Out
7	21	GPIO5_CTS0_SPIDO_P3.5	GPIO_5 or UART0 CTS Output or SPI MOSI
8	20	GPIO6_RTS0_SPIDI_P3.4	GPIO_6 or UART0 RTS Input or SPI MISO
9	8	GPIO7_RXD1_UART1_P1.0	GPIO_7 or UART1 Data In
10	9	GPIO8_TXD1_UART1_P1.1	GPIO_8 or UART1 Data Out
11	23	GPIO9_CTS1_PWM3_P3.7	GPIO_9 or UART1 CTS output or PWM3
12	22	GPIO10_RTS1_PWM2_SPICLK_P3.6	GPIO_10 or UART1 RTS input or PWM2 or SPI SCLK
13	7 (or 3)	GPIO11_P0.7 (or ACH3)	GPIO_11 (or Analog In if pads jumped and trace cut)
14	6 (or 2)	GPIO12_P0.6 (or ACH2)	GPIO_12 (or Analog In if pads jumped and trace cut)
15	5 (or 1)	GPIO13_P0.5 (or ACH1)	GPIO_13 (or Analog In if pads jumped and trace cut)
16	4 (or 0)	GPIO14_P0.4 (or ACH0)	GPIO_14 (or Analog In if pads jumped and trace cut)
17	3	GPIO15_P0.3	GPIO_15
18	2	GPIO16_P0.2	GPIO_16
19	1	GPIO17_P0.1	GPIO_17
20	0	GPIO18_P0.0	GPIO_18
21		VCC	Power Supply
22		ISP	In-System Programming Line
23		RESET*	Module Reset, Active Low
24		GND	Power Supply

ATMEL ATmega128RFA1

In addition to modules built on the SNAP Engine footprint, you will find SNAP running on ATMEL chips. See the following section for details unique to the SNAP Engine form factor.

IO pins

The ATmega128RFA1 supports 38 IO pins. Any of the 38 IO can be a digital input, or digital output.

Wakeup pins

Seventeen of the 38 IO support a hardware “wakeup” capability. See IO 0-7, 8-11, 16, and 20-23. Setting the unit to wake from an edge-triggered interrupt on INT4-INT7 (SNAPpy IOs 20-23) will work, but will result in higher power consumption during sleep. If you must have an edge-triggered wake signal, it is recommended you use a different pin.

Analog inputs

Eight of the 38 IO can be used as analog inputs. See IO 24-31.

UART0

Four pins support UART 0, see IO 16, 17, 20, and 21. If you do not need RTS/CTS signals, then IO 20 and 21 are available for other usage.

UART1

Four pins support UART 1, see IO 10, 11, 12, and 23. If you do not need RTS/CTS signals, then IO 12 and 23 are available for other usage.

SPI

Three pins can optionally be used for SPI. See IO 28-30.

NOTE – these are not the *hardware* SPI pins. **SNAPpy SPI is done via software emulation**. You will also need one “SPI Chip Select” pin *per external SPI device*. Any available IO pin can be used for this purpose.

I^{2}C

Two pins can optionally be used for I^{2}C, see IO 24 and 25.
NOTE – these are not the *hardware* I^{2}C pins. **SNAPpy I^{2}C is done via software emulation**.

PWM

Eight pins can optionally be used as Pulse Width Modulation (PWM) outputs, see IO 4-7, 19-21, and 37.

The table on the following page summarizes the IO mapping on the ATmega128RFA1 chip.

You will notice that the “IO” numbering scheme chosen simply steps through the available ports in alphabetical order: ports B, D, E, F, and G. (There is no port A or port C on an ATmega128RFA1).

ATmega128RFA1 Port mappings

Processor Port Pin	SNAPpy IO
PB0 PCINT0	0
PB1 PCINT1	1
PB2 PCINT2	2
PB3 PCINT3	3
PB4 PCINT4 OC2	4
PB5 PCINT5 OC1A	5
PB6 PCINT6 OC1B	6
PB7 PCINT7 OC1C OC0A	7
PD0 INT0	8
PD1 INT1	9
PD2 INT2 RXD1	10
PD3 INT3 TXD1	11
PD4 CTS1 ICP1	12
PD5 RTS1	13
PD6	14
PD7	15
PE0 RXD0 PCINT8	16
PE1 TXD0	17
PE2 CTS0	18

Processor Port Pin	SNAPpy IO
PE3 RTS0 OC3A AIN0	19
PE4 INT4 OC3B	20
PE5 INT5 OC3C	21
PE6 INT6	22
PE7 INT7 ICP3	23
PF0 ADC0 I^{2}C_SCL	24
PF1 ADC1 I^{2}C_SDA	25
PF2 ADC2	26
PF3 ADC3	27
PF4 ADC4	28
PF5 ADC5	29
PF6 ADC6	30
PF7 ADC7	31
PG0	32
PG1	33
PG2	34
PG3	35
PG4	36
PG5 OC0B	37

More “Wakeup” Pins

On an ATMEL ATmega128RFA1 Wireless Module, any of the following IOs can be used to wake the processor:

IO0 –IO7: Map to PCINT0 – PCINT7
IO8 –IO11: Map to INT0 – INT3
IO16: Maps to PCINT8
IO20 –IO23: Map to INT4 – INT7

Analog Input Pins

On the ATmega128RFA1 Wireless Module, the physical ADC inputs map to IO24 through IO31. These correspond to processor pins PF0-PF7.

Serial port 0

Four IO pins can optionally function as UART0. IO16 becomes RXD0 and IO17 becomes TXD0. If hardware flow control is required, IO20 becomes CTS0 and IO21 becomes RTS0.

Serial port 1

Four IO pins can optionally function as UART1. IO10 becomes RXD1 and IO11 becomes TXD1. If hardware flow control is required, IO12 becomes CTS1 and IO23 becomes RTS1.

PWM Output Pins

On the ATmega128RFA1 Wireless Module, you can use the following IOs for pulse width modulation PWM:

IO4: Maps to OC2
IO5: Maps to OC1A
IO6: Maps to OC1B
IO7: Maps to OC1C and OC0A
IO19: Maps to OC3A
IO20: Maps to OC3B
IO21: Maps to OC3C
IO37: Maps to OC0B

SPI

Three IO pins can optionally function as an SPI bus. IO29 becomes SCLK, IO30 becomes MOSI, and IO28 becomes MISO.

I^2C

Two IO pins can optionally function as an I^2C bus. IO24 becomes SCL, and IO25 becomes SDA.

Memory Usage

SNAP Protocol Memory Usage:

Global Buffer Pool:	20
UART Budget:	6
Mesh Routing Budget:	6
RPC Budget:	6
Radio Budget:	6
STDOUT Budget:	4

SNAPpy Virtual Machine Memory Usage:

Number of Tiny Strings:	14
Tiny String Size:	up to 16 characters
Number of Medium Strings:	8
Medium String Size:	up to 126 characters
Global Variables:	64
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

Platform Specific SNAPpy Built-In Functionality

Built-in function getInfo**():**
On a `getInfo()` call, a parameter value of 0 requests a "vendor code."

On a ATmega128RFA1, `getInfo(0)` returns 4 (meaning "Atmel").

On a `getInfo()` call, a parameter value of 3 requests a "platform code." These are "per vendor" (not global), so both RF Engines and ATmega128RFA1 nodes return a value of 0, indicating they are the first platform from their respective vendors.

Built-in functions cbusRd**() and** cbusWr**():**
These functions have no effect on the ATmega128RFA1.

Built-in functions peekRadio**() and** pokeRadio**():**
These functions are not necessary on the ATmega128RFA1. The internal radio registers are in the main memory space of the chip, and can be read or written using the regular `peek()` and `poke()` functions.

Built-in function lcdPlot**():**
Built-in function `lcdPlot()` has no effect in ATmega128RFA1 builds.

Built-in function setPinSlew**():**
Built-in function `setPinSlew()` has no effect in ATmega128RFA1 builds.

Built-in function pulsePin**():**
On the ATmega128RFA1, *negative* durations are in units of approximately 0.94 microsecond.

Here are some requested versus measured pulse timings, taken with a logic analyzer.

Requested duration	Measured Pulse Width (µs)
-1	0.94
-2	1.37
-3	1.82
-10	4.88
-20	9.25
-30	13.63

Built-in function random**():**
On the ATmega128RFA1, the pseudo random number generation is done in hardware, not software.

Built-in function readAdc**()**
On the ATmega128RFA1, channels 0-7 correspond to direct reads of one of the eight external analog input pins. Channels 8-29 return various "differential" readings, as shown in the following table.

Channel Number	Returns
0	ADC0
1	ADC1
2	ADC2
3	ADC3
4	ADC4
5	ADC5
6	ADC6
7	ADC7
8	10 * (ADC0 – ADC0)
9	10 * (ADC1 – ADC0)
10	200 * (ADC0 – ADC0)
11	200 * (ADC1 – ADC0)
12	10 * (ADC2 – ADC2)
13	10 * (ADC3 – ADC2)
14	200 * (ADC2 – ADC2)
15	200 * (ADC3 – ADC2)
16	ADC0 – ADC1
17	ADC1 – ADC1
18	ADC2 – ADC1
19	ADC3 – ADC1
20	ADC4 – ADC1
21	ADC5 – ADC1
22	ADC6 – ADC1

23	ADC7 – ADC1
24	ADC0 – ADC2
25	ADC1 – ADC2
26	ADC2 – ADC2
27	ADC3 – ADC2
28	ADC4 – ADC2
29	ADC5 – ADC2

This function returns an integer value 0-1023 (these are 10-bit analog to digital converters).

The reference voltage is 1.6 volts. Refer to the ATmega128RFA1 datasheets for more information.

Built-in function setRadioRate**():**
On the ATmega128RFA1, `setRadioRate()` supports values of 0, 1, 2, and 3. The resulting data rates are as follows:

setRadioRate() parameter	Radio Data Rate
0	250 Kbps
1	500 Kbps
2	1 Mbps
3	2 Mbps

Radio rate 0 is compatible with any 802.15.4 SNAP radio using rate 0. Any other rate may not work with any non-ATMEL ATmega128RFA1 radio, and will only work with an ATmega128RFA1 radio working at the same rate.

Built-in function setSegments**():**
On the ATmega128RFA1, `setSegments()` has no effect.

Built-in function sleep**():**
The ATmega128RFA1 supports two sleep modes. Mode 1 should always be used if a 32kHz crystal is installed on the target. Use mode 0 if no RTC is available.

In the original RF100 SNAP Engine, the ticks parameter could accept either a zero, a positive, or a negative number. Zero indicates “untimed sleep” (i.e. wake up on pin-interrupt only). Positive numbers indicated how long to sleep in mode-dependant “ticks,” and negative numbers indicated fractions of a second.

On the ATmega128RFA1, zero still indicates “untimed sleep.” Positive numbers indicate how long to sleep in 1.0s “ticks” (regardless of mode), and negative numbers indicate the following fractions of a second:

Ticks	Mode 0 Sleep Time	Mode 1 Sleep Time
-1	33 ms	82 ms
-2	33 ms	82 ms

-3	66 ms	82 ms
-4	132 ms	111 ms
-5	263 ms	236 ms
-6	508 ms	486 ms
-7	1000ms	986 ms

Maximum allowed sleep: 1023s

Note – Do not configure the ATmega128RFA1 to wake up from an edge-triggered interrupt on INT4-INT7. This will work, but the processor will not go into a very "deep" sleep, and it will continue to draw a substantial amount of current (~900µA).

Built-in function getLq**():**
The ATmega128RFA1 supports two `getLq()` modes:

Invoking `getLq()` or `getLq(0)` returns a signal strength in negative dBm. So a return value of 20 indicates that the last transmission received had a signal strength of -20dBm. This matches the behavior of the other SNAP platforms.

Invoking `getLq(1)` returns a link quality measurement from 0 – 255, where 255 indicates the best possible link quality, and 0 indicates the worst.

Relevant Feature Bits (NV #11):

Bit 0x0200 – Sets the radio to control transmit power based on European protocols and standards. If you are using nodes based on the ATmega128RFA1 in Europe, you should set this feature bit. Otherwise, you should leave the bit unset.

Vendor-specific settings:
NV Parameter 64 is used on the ATmega128RFA1.

Bit 0x0001 – Enables a "turbo" mode in the node, allowing for slightly faster radio communications between nodes by reducing the pauses between packets. You can set this bit (and reboot the node) to slightly increase throughput, but the node will only be able to communicate over the air with other ATmega128RFA1 nodes that also have the bit set. The node will no longer be able to communicate with other 2.4 GHz SNAP nodes based on other platforms, or based on this platform but with the bit not set.

Performance Metrics

Here are the results of some recent performance measurements, which may help you gauge if SNAPpy can address your application's timing requirements.

These results are for the ATMEL ATmega128RFA1.

Time to awaken from sleep (mode 0):
< 850 microseconds

Time to startup from power-on:
< 250 milliseconds

Maximum rate a SNAPpy script can toggle a GPIO pin:
9518 Hz

In other words, each True/False cycle took 105.06 microseconds.
(To change the state of a pin takes 52.56 microseconds, and each pulse requires two state changes).

Keep in mind that as a general rule, SNAPpy scripts **should not be looping**, the above rate is only attainable if the node is doing *nothing else* (for example, no radio or serial port communication).

Maximum rate for readAdc() calls:
maximum 19952 samples/second

In other words, each sample took 50.12 microseconds to gather.

NOTE! – This measurement was taken using a script that did not actually do anything with the data. You will also have to take into consideration any numeric processing required (data thresholding, etc.), as well as the need to actually store the data someplace.

Also note that the *very first* `readAdc()` call takes 174 microseconds (not 50.12 microseconds) because of some initial "first time only" hardware initialization that is required. You may want to make a "setup" `readAdc()` call in your script's startup event handler.

I^{2}C Byte Transfer Time
The actual I^{2}C transfers are done using "bit banging" in software. This was measured using a logic analyzer at 239 µs per byte.

SPI Byte Transfer Time
The actual SPI transfers are done using "bit banging" in software. This was measured using a logic analyzer at 177 µs per byte.

Virtual Machine Speed
SNAP 2.3 Instructions Per Second (IPS): 39240

Reserved Hardware

On the ATmega128RFA1, SNAP uses Timer 4 and Timer 5 internally.

Timer 4 provides the 1 millisecond “heartbeat” and Timer 5 is used in some sleep modes.

Do not peek()/poke() these timers!

The remaining timers are available for use by your scripts (for example, PWM).

Synapse RF200

SNAP Engines based on the ATmega128RFA1 are available. All the details appropriate for the chip-based SNAP Modules (see the previous section) apply to the SNAP Engine, with the following additions and exceptions. Pin numbers below refer to the pin on the SNAP Engine footprint. To reference the pins in your code, use the SNAPpy IO number from the table below. Note that the SPI and I^2C locations are different on the RF200 than they are on the ATmega128RFA1.

RF200 SNAP Engines should not be used on the Synapse SN111 End Device board. The power-up state of the pins conflicts with the relay controls on that board.

Form Factor
The RF200 SNAP Engine is the SNAP Engine based on the ATMEL ATmega128RFA1.

IO pins
The RF200 supports 20 GPIO pins, one more than the original RF100 SNAP Engine. GPIO_19 is available on pin 22 of the SNAP Engine.

Sleep
The RF200 includes a 32kHz crystal RTC, so for most efficient sleep you should use sleep mode 1.

Wakeup pins
Nine of the IO pins can be used to wake a sleeping engine: pins 2-5 (GPIO_0-GPIO_3), pins 7-10 (GPIO_5-GPIO_8), and pin 12 (GPIO_10). Using pins 7, 8, or 12 (GPIO_5, GPIO_6, or GPIO_10) as an edge-triggered wakeup signal will work, but will result in higher power consumption.

Analog inputs
Seven of the IO pins support Analog Input: pins 13-15 and pins 17-20.

UART0
Four pins support UART 0: pins 5-8 (GPIO_3-GPIO_6). If you do not need RTS/CTS signals, then 7 and 8 are available for other usage.

UART1
Four pins support UART 1: pins 9-11 (GPIO_7-GPIO_10). If you do not need RTS/CTS signals, then IO 10 and 11 are available for other usage.

SPI
Three pins can optionally be used for SPI: pins 14-16 (GPIO_12-GPIO_14) are MISO, SCLK and MOSI. These are not the *hardware* SPI pins. **SNAPpy SPI is done via software emulation**.
You will also need one "SPI Chip Select" pin *per external SPI device*. Any available IO pin can be used for this purpose.

I^2C
Two pins can optionally be used for I^2C: pins 19 and 20 (GPIO_17 and GPIO_18). These are not the *hardware* I^2C pins. **SNAPpy I^2C is done via software emulation**. These are *also* not the same pins used for I^2C on the ATmega128RFA1 build of the firmware.

PWM
Six pins can optionally be used as Pulse Width Modulation (PWM) outputs: pins 2-4 (GPIO_0-GPIO_2), pins 7 and 8 (GPIO_5 and GPIO_6), and pin 22 (GPIO_19).

Pin Configuration of an ATmega128RFA1 in SNAP Engine Format (RF200)

Engine Pin	SNAPpy IO	Name	Description
1		GND	Power Supply
2	7	GPIO0 OC0A OC1C PCINT7 PB7	GPIO_0 or PWM or Interrupt
3	6	GPIO1 OC1B PCINT6 PB6	GPIO_1 or PWM or Interrupt
4	5	GPIO2 OC1A PCINT5 PB5	GPIO_2 or PWM or Interrupt
5	16	GPIO3 RXD0 PCINT8 PE0	GPIO_3 or UART0 Data In or Interrupt
6	17	GPIO4 TXD0 PE1	GPIO_4 or UART0 Data Out
7	20	GPIO5 OC3B INT4 PE4	GPIO_5 or UART0 CTS Output or PWM or Interrupt
8	21	GPIO6 OC3C INT5 PE5	GPIO_6 or UART0 RTS Input or PWM or Interrupt
9	10	GPIO7 RXD1 INT2 PD2	GPIO_7 or UART1 Data In or Interrupt
10	11	GPIO8 TXD1 INT3 PD3	GPIO_8 or UART1 Data Out or Interrupt
11	12	GPIO9 CTS1 ICP1 PD4	GPIO_9 or UART1 CTS output or Input Capture
12	23	GPIO10 RTS1 ICP3 INT7 CLKO	GPIO_10 or UART1 RTS input or Clock Output Buffer or Interrupt
13	24	GPIO11 ADC0 PF0	GPIO_11 or Analog0
14	25	GPIO12 ADC1 MOSI PF1	GPIO_12 or Analog1 or SPI MOSI
15	26	GPIO13 ADC2 DIG2 SCLK PF2	GPIO_13 or Analog2 or SPI CLK or Antenna Diversity Control
16	18	GPIO14 XCK0 AIN0 MISO PE2	GPIO_14 or SPI MISO or Analog Comparator or External Clock[9]
17	28	GPIO15 ADC4 TCK PF4	GPIO_15 or Analog4 or JTAG Test Clock
18	29	GPIO16 ADC5 TMS PF5	GPIO_16 or Analog5 or JTAG Test Mode Select
19	30	GPIO17 ADC6 TDO SDA PF6	GPIO_17 or Analog6 or JTAG Test Data Out or I^2C SDA
20	31	GPIO18 ADC7 TDI SCL PF7	GPIO_18 or Analog7 or JTAG Test Data In or I^2C SCL
21		VCC	Power Supply
22	19	GPIO19 OC3A AIN1 PE3	"GPIO_19" or Analog Comparator or PWM or Output Compare Match[10]
23		RESET*	Module Reset, Active Low
24		GND	Power Supply

[9] On RF200 SNAP Engines with no power amplifier, pin 16 (GPIO_14) provides Analog3 (ADC3/PF3) instead of the characteristics of PE2. Analog3 is unavailable on RF200 SNAP Engines with a power amplifier.
[10] Other SNAP Engines have a debug connection on pin 22. The architecture of the RF200 requires multiple debug connections, which come out on other pins. Rather than leave pin 22 useless, it is available as an additional GPIO or Analog Comparator. This will not be directly accessible on Synapse development boards, but custom circuit designs have the pin available for specialized purposes.

Synapse SS200

Synapse also offers the SNAP Stick 200. This device, based on the ATMEL ATmega128RFA1 hardware, is a USB dongle - about the size of a thumb drive. It is designed to act as a bridge between Portal or SNAP Connect and your 802.15.4 2.4 GHz wireless network.

Because it is based on the ATmega128RFA1, the SS200 has the same capabilities as the underlying hardware, relating to sleep options and radio rates.

The USB dongle form factor means that only one UART is available on the SS200. UART1 connects through the USB port. If you change the default UART (NV Parameter 12) to 0, you will not be able to communicate directly with the device, and will have to either use Portal to Factory Default NV Params or use a different SNAP Device as a bridge and reset the default UART over the air.

Also because of the form factor, you do not have normal access to the GPIO pins on the SS200. The device was designed to primarily act as a bridge device. The only feedback available from the device comes in the form of a tri-color LED, controlled by pins 5 and 6:

LED State	Pin 5	Pin 6
Off	High (True)	High (True)
Red	Low (False)	High (True)
Green	High (True)	Low (False)
Amber	Low (False)	Low (False)

The SS200 includes an internal power amplifier. It also has a 32 kHz crystal, so for most efficient sleep you should use sleep mode 1. Note that there is no way to trigger an external wakeup signal to the device, so you should be careful to only use timed sleep.

Silicon Labs Si100x

In addition to RF300 and RF301 modules built on the SNAP Engine footprint, you will find SNAP running on Silicon Labs Si100x chips. (See the following section for details unique to the RF300/RF301 on the SNAP Engine form factor.)

There are two versions of the Si100x firmware: One version provides frequency hopping in the 900 MHz range, and the other provides service in the 868 MHz range. Details below refer to both versions of the firmware except where explicitly specified as different.

IO pins
The Si100x supports 19 output pins. 18 of these can be input pins.

Wakeup pins
Nine of the 19 IO support a hardware "wakeup" capability. see IO 1-8.

Analog inputs
Seventeen of the 19 IO can be used as analog inputs. See IO 0-15 and 17.

UART0
Four pins support UART 0, see IO 1-4. If you do not need RTS/CTS signals, then IO 1 and 2 are available for other usage. For serial connections, the UART is more restricted than on some other platforms. The only serial configurations available are 8N1 and 8N2.

UART1
There is only one UART (UART 0) available on the Si100x. Any call to `initUart()` that references UART 1 will be applied to UART 0 instead.

SPI
Three pins can optionally be used for SPI, see IO 6-8.

NOTE – these are not *hardware* SPI pins. **SNAPpy SPI is done via software emulation**.
You will also need one "SPI Chip Select" pin *per external SPI device*. Any available IO pin can be used for this purpose.

I^{2}C
Two pins can optionally be used for I^{2}C. see IO 10 and 11.
NOTE – these are not *hardware* I^{2}C pins. **SNAPpy I^{2}C is done via software emulation**.

PWM
Six pins can optionally be used as Pulse Width Modulation (PWM) outputs. The pins are user-configurable.

Serial rates
The lowest serial data transfer rate supported is 889 baud.

The table on the following page summarizes the IO mapping on the Si100x chip.

Si100x Port mappings

Processor Port Pin	SNAPpy IO
P0.0 ADC0 V_{REF}	0
P0.2 ADC2 RTS0	1
P0.3 ADC3 CTS0	2
P0.4 ADC4 TXD0	3
P0.5 ADC5 RXD0	4
P0.6 ADC6 CNVSTR	5
P1.5 ADC13 SPI_SPLK	6
P1.6 ADC14 SPI_MISO	7
P1.7 ADC15 SPI_MOSI	8
P2.0 ADC16	9

Processor Port Pin	SNAPpy IO
P2.1 ADC17	10
P2.2 ADC18	11
P2.3 ADC19	12
P2.4 ADC20	13
P2.5 ADC21	14
P2.6 ADC22	15
P2.7 C2D	16
GPIO_0	17
ANT_A	18[11]

"Wakeup" Pins

On an Si100x Wireless Module, any of the following IOs can be used to wake the processor:

IO0 Map to P0.0
IO1–IO5: Map to P0.2–P0.6
IO6–IO8: Map to P1.5–P1.7

Analog Input Pins

On the Si100x Wireless Module, 16 pins can be used as ADC inputs. P0.0-P0.6 (excepting P0.1) are available as ADC0-ADC6 (excepting ADC1), P1.5-P1.7 as ADC13-15, and P2.0-P2.6 as ADC16-22, are all available.

Serial port 0

Four IO pins can optionally function as UART0. IO4 becomes RXD0 and IO3 becomes TXD0. If hardware flow control is required, IO2 becomes CTS0 and IO1 becomes RTS0.

PWM Output Pins

On the Si100x Wireless Module, you can configure up to six PWM pins. These are user-assignable to any available IO (except IO16-IO18).

SPI

Three IO pins can optionally function as an SPI bus. IO6 becomes SCLK, IO7 becomes MISO, and IO8 becomes MOSI.

[11] IO18 is available for output only.

I^{2}C

Two IO pins can optionally function as an I^{2}C bus. IO11 becomes SCL, and IO10 becomes SDA. Use external pull-up resistors to VCC. Resistors on the order of 10 KΩ work well.

Memory Usage

SNAP Protocol Memory Usage:

Global Buffer Pool:	7
UART Budget:	4
Mesh Routing Budget:	4
RPC Budget:	4
Radio Budget:	4
STDOUT Budget:	2

SNAPpy Virtual Machine Memory Usage:

Number of Tiny Strings:	7
Tiny String Size:	up to 8 characters
Number of Medium Strings:	6
Medium String Size:	up to 62 characters
Global Variables:	64
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

Platform-Specific SNAPpy Functionality

900 MHz Channel usage:

FCC rules for FHSS radios in the 900 MHz range require that the transmitting radio hop across multiple frequencies rather than performing all its transmissions on a single frequency. FHSS SNAP modules based on the Si100x select a range of 25 frequencies from the 66 available, based on the channel specified for the node (in NV parameter 4), and hop amongst those 25 frequencies.

Adjacent SNAP channels will have overlap in the frequencies in use with FHSS firmware, but interference will be minimized by the frequency hopping. There will not be any cross-channel communications: Even though nodes on different channels may be sharing some subset of frequencies, they will not “hear” each others’ communications.

Consult the getChannel() function for details about which channels and frequencies are used.

If your operating environment has issues with noise and interference in the 900 MHz range, you may find that changing to a different channel (which provides a different pool of 25 frequencies) reduces or eliminates the interference.

868 MHz Channel usage:
Si100x radios operating in the 868 MHz range share the same pool of three frequencies regardless of the radio channel specified. Controls within the SNAP firmware prevent communications from bleeding across channels.

Carrier Sense (NV Parameter 16) and Collision Detect (NV Parameter 17):
The default values for Carrier Sense (NV Parameter 16) and Collision Detect (NV Parameter 17) are disabled. (This is true for all SNAP platforms.) Enabling Carrier Sense or Collision Detect can increase the reliability of multicast messaging, but may provide a performance hit. If you wish to enable these abilities, you may want to adjust Silicon Labs' default value for the RSSI Threshold for Clear Channel Indicator, which you can do with the `pokeRadio(0x27, value)` command, to meet the needs of your network and environment.

Setting the value too low will cause SNAP to interpret noise as interference and needlessly rebroadcast multicast packets. In a worst-case scenario, the node may get stuck rebroadcasting hundreds or thousands of times before it can receive or transmit any other radio traffic. Silicon Labs' default value for the register is 30 (0x1E), and higher values make the radio accept a higher noise floor before indicating that a transmission has been compromised. The SNAP firmware defaults the value to 65. Reference the manufacturer's documentation for more details.

Maximum Loyalty (NV Parameter 53):
The requirement for channel hopping in the 900 MHz FHSS firmware means that a listening radio will constantly be scanning the available frequencies to listen for transmissions from other nodes on the same channel. Because the frequency hopping order is defined, after sending or receiving a transmission a node will know where to expect the next transmission. It will wait, listening on that frequency, for the duration of the Maximum Loyalty period (in milliseconds) before resuming its scan of all frequencies.

If a node is broadcasting sequential packets, it will assume that the receiving radio will be expecting its next packet on a defined frequency and will broadcast its next packet with a shorter preamble, allowing for faster data throughput. If a node sends a broadcast and is expecting a reply, it will likewise wait on that next hop frequency for that reply for the duration of the loyalty period before resuming a scan of all channels.

The default value of 185 is an optimized value for most environments. If you modify this parameter, be sure to set it to the same value for all nodes in your network. Setting the value to 0 means that no node will ever delay before scanning all channels when listening for messages, and that no transmitting node will ever transmit packets with a shorter preamble.

Clock Regulator (NV Parameter 65):
For Si100x implementations that do not have an external crystal to control RTC and sleep timing, NV Parameter 65 allows you to regulate the internal clock somewhat to adjust for differences in your hardware and your environment. For most accurate timing, you may need to adjust this value depending on the ambient temperature and the operating voltage of your node. The range for the value is 0-15, and the default value is 8. The change takes effect only after rebooting.

This parameter has no effect if you have an external crystal controlling your sleep timing.

Built-in functions cbusRd**() and** cbusWr**():**
These functions have no effect on the Si100x.

Built-in functions getEnergy**() and** scanEnergy**():**
For FHSS firmware, the getEnergy() and scanEnergy() built-in functions return the energy levels heard on the frequency you specify. (Each channel uses 25 frequencies in the 900 MHz band. In the 868 MHz band, all channels use the same three frequencies.) See the getEnergy() and getChannel() functions for details about how the frequencies are distributed through the SNAP channels.

For 900 MHz firmware, the scanEnergy() function returns a 66-character string instead of the normal 16-character string returned on other platforms. For 868 MHz firmware, the scanEnergy() function returns a 3-character string.

Built-in function getInfo**():**
On a `getInfo()` call, a parameter value of 0 requests a "vendor code."
On an Si100x, `getInfo(0)` returns 5 (meaning "Silicon Labs").

On a `getInfo()` call, a parameter value of 3 requests a "platform code." On 900 MHz Si100x-based nodes (including the RF300), this function returns an 11 for FHSS firmware. On 868 MHz Si100x-based nodes (including the RF301), this function returns an 8.

Built-in function getStat**():**
The results of the `getStat(3)` and `getStat(4)` calls are undefined, as the Si100x-based nodes have only one UART.

Built-in function lcdPlot**():**
Built-in function `lcdPlot()` has no effect in Si100x builds.

Built-in functions peek**() and** poke**():**
Use negative address values to peek and poke special function registers. See the peek() and poke() description for more details.

Built-in function pulsePin**():**
On the Si100x, *negative* durations are in units of approximately 1.0 µs, once certain overhead minimum limits are exceeded. Here are some requested versus measured pulse timings, taken with a logic analyzer.

Requested duration	Measured Pulse Width (µs)	
	GPIO_0-GPIO_16	GPIO_17-GPIO_18
-1	15.917	29.813
-10	16.000	29.750
-20	20.000	29.813
-50	50.417	50.063
-100	100.167	99.875
-200	199.583	199.250

Built-in function readAdc**()**
The reference voltage is 3.3 volts. In addition to the 10-bit ADC channels listed in the Port Mappings table, `readAdc(27)` can provide a reading of an internal temperature sensor in the module, and `readAdc(28)` can provide an indication of voltage supply. (`readAdc(28)` requires additional pokes to enable the reading. Refer to the Silicon Labs documentation for more details.)

You can also choose to scale your ADC readings to 1.65 volts. To do so, poke(-0xE8, 0x80) to enable the ADC and then poke(-0xBC, 0x1B) to set the ADC to a 6.075 MHz sample clock that samples on demand on a 1.65 volt scale. Refer to the Si100x documentation for further details.

Built-in function setPinPullup()**:**
Built-in function `setPinPullup()` has no effect on any pin except pin 17 (radio GPIO_0). If you need to establish pin pull-ups, you can `poke(-0xE3, 0x40)` to set the pull-up on pins 0-16. Use `poke(-0xE3, 0xC0)` to disable the pull-ups. You cannot set a pull-up on an individual pin.

Built-in function setPinSlew**():**
Built-in function `setPinSlew()` has a different purpose on the Si100x builds. Rather than controlling the slew rate for a pin, it controls the strength at which the pin can drive an output signal. The default setting (False) provides a standard output signal strength. Calling `setPinSlew(pin, True)` allows the module to push more current through the pin, which may be necessary for some applications (such as driving a relay).

Built-in function setRadioRate**():**
On the Si100x, setRadioRate() has no effect. Only the standard 150 Kbps rate is supported.

Built-in function setSegments**():**
On the Si100x, `setSegments()` has no effect.

Built-in function sleep**():**
The Si100x supports four sleep modes. Modes 0 and 1 require the presence of an external 32kHz crystal to regulate your sleep timing. Using these modes with no crystal present will “hang” the node. Modes 2 and 3 should be used when there is no external crystal present, and will “hang” the node if the crystal is present. Modes 0 and 2 provide a slightly “deeper” sleep (lower power consumption) than modes 1 and 3. However modes 0 and 2 require an additional 300 ms to wake up, and do not maintain state for IO17 or IO18 while sleeping.

If a crystal is present, the first timed sleep executed after a node boots requires approximately 80 milliseconds of additional configuration time to initialize the clock. Subsequent timed sleeps will not have this overhead.

On the Si100x, zero indicates "untimed sleep." Positive numbers indicate how long to sleep in 1.0s "ticks" (regardless of mode), and negative numbers indicate the following fractions of a second:

Ticks	Sleep time
-1	16ms
-2	31ms
-3	63ms
-4	125ms
-5	250ms
-6	500ms
-7	1000ms

Maximum allowed sleep: 32767 seconds, or 9:06:07.

Encryption
The Si100x supports Basic encryption. AES-128 encryption is not available in the Si100x firmware. (It is available for RF300/RF301 firmware.)

Alternate Radio Trim settings:
NV Parameter 63 is used on the Si100x. The default value is 183 but you can override this by saving a new value in NV #63. For the definition of the trim value, refer to the Silicon Labs datasheets.

Vendor-specific settings:
NV Parameter 64 is used on the Si100x.

Bit 0x0001 – Indicates how the TX and RX registers are set in the hardware.

When this bit is clear (set to 0), register 0x0C is set to 0x15, the receive state, and register 0x0D is set to 0x12, the transmit state. This is the appropriate setting for the RF30x SNAP Engine and any hardware similarly connected.

When this bit is set (to 1), register 0x0C is set to 0x12 and register 0x0D is set to 0x15. This is the appropriate setting for the Silicon Labs EZRadio Pro 1000-TCB1 development board and any hardware similarly connected.

If this bit is incorrectly set radio signal strength will be diminished, as will the radio's ability to clearly receive signals. Users developing their own hardware according to the Silicon Labs demonstration board should be sure to set this feature bit.

Resetting factory parameters does not modify this parameter.

Performance Metrics

Here are the results of some recent performance measurements, which may help you gauge if SNAPpy can address your application's timing requirements.

Time to awaken from sleep (mode 0):
340 ms

Time to awaken from sleep (mode 1):
1.2 ms

Time to startup from power-on:
677 ms

Maximum rate a SNAPpy script can toggle a GPIO pin:
1149.6 Hz

In other words, each True/False cycle took 869.9 µs.
(To change the state of a pin takes 434.9 µs, and each pulse requires two state changes).

Keep in mind that as a general rule, SNAPpy scripts **should not be looping**, the above rate is only attainable if the node is doing *nothing else* (for example, no radio or serial port communication).

Maximum rate for readAdc() calls:
maximum 1107 samples/second. In other words, each sample took 903.1 µs to gather.

NOTE! – This measurement was taken using a script that did not actually do anything with the data. You will also have to take into consideration any numeric processing required (data thresholding, etc.), as well as the need to actually store the data someplace.

I^{2}C Byte Transfer Time
The actual I^{2}C transfers are done using "bit banging" in software. This was measured using a logic analyzer at 460 µs per byte.

SPI Byte Transfer Time
The actual SPI transfers are done using "bit banging" in software. This was measured using a logic analyzer at 390 µs per byte.

Virtual Machine Speed
SNAP 2.4 Instructions Per Second (IPS): 3600

Reserved Hardware

Timers

On the Si100x, SNAP uses Timer 1 and Timer 2 internally.

Timer 2 provides the 1 millisecond "heartbeat." You should not do anything that makes explicit use of that timer.

Timer 1 is used to set the UART baud rate. If you are not using a serial connection, you can use Timer 1 for your own purposes. (The real-time clock is also available to you. Initialization of the real-time clock is cumbersome, but can be automated by executing a brief timed sleep.)

Timers 0 and 3 are available for user interaction. See the manufacturer's documentation for details in using these.

Advanced SNAP Hardware Assumptions

If you are designing your own circuitry using the Si100x, keep these assumptions in mind:

- The radio shutdown pin is tied to P0.7.
- The radio interrupt pin is tied to P0.1.
- Tie GPIO_1 to receive on the RF multiplexer.
- Tie GPIO_2 to transmit on the RF multiplexer.

Synapse RF300/RF301

SNAP Engines based on the Si1000 are available. All the details appropriate for the chip-based SNAP Modules (see the previous section) apply to the SNAP Engine, with the following additions and exceptions. Pin numbers below refer to the pin on the SNAP Engine footprint. To reference the pins in your code, use the SNAPpy IO number from the table below, or import synapse.platforms into your script and refer to the pins by their GPIO number.

The RF300 operates in the 900 MHz range with frequency hopping. The RF301 operates in the 868 MHz range. They will collectively be referred to as the RF30x SNAP Engine in this section, with RF300 and RF301 being used where distinctions are necessary.

RF30x SNAP Engines should not be used on a Synapse SN111 End Device board. The power-up state of the pins conflicts with the relay controls on that board.

The RF30x SNAP Engines make use of an external memory chip to increase the available script space. This memory availability comes at a cost, though: GPIO pins 11 through 14 (pins 13 through 16 on the SNAP Engine footprint) are not available. If you have need of these pins and do not need the extra script space, you can load the RF30x with the frequency-appropriate Si100x firmware to recover the use of the pins.

Form Factor

The RF30x SNAP Engines are the SNAP Engines based on the Silicon Labs Si1000.

IO pins

The RF30x SNAP Engines support 14 GPIO pins, plus pin 20 (GPIO_18), which is an output-only pin. Pin 18 (GPIO_16) has limited drive strength, as it routes through a 1 KΩ resistor. Pins 13-16 (GPIOs 11-14) are not available for use when using the RF30x firmware. (If you load the RF30x SNAP Engine with the appropriate Si100x firmware you recover the use of these four pins, but lose script space.) Note that while using the RF30x firmware, attempts to use GPIO_14 as an input will give accurate readings but may conflict with the module's ability to access your script from the external memory, and could cause the SNAP Engine to hang. You should not connect any signals to GPIOs 11-14 when using the RF30x firmware.

Wakeup pins

Six of the IO pins can be used to wake a sleeping engine: Pins 8-12 (GPIOs 6-10), and pin 17 (GPIO 15). (If you load the RF30x SNAP Engine with the appropriate Si100x firmware, you recover the use of GPIOs 11-14 as interrupt pins, but lose script space.)

Analog inputs

Twelve of the IO pins support Analog Input: pins 2-12 (GPIOs 0-10) and pin 17 (GPIO 15). (If you load the RF30x SNAP Engine with the appropriate Si100x firmware, you recover the use of GPIOs 11-14 as analog inputs, but lose script space.)

UART0

Four pins support UART 0: pins 9-12 (GPIO_7-GPIO_10). If you do not need RTS/CTS signals, then pins 11 and 12 (GPIO_9 and GPIO_10) are available for other uses. Note that on other SNAP Engines, these pins are used for UART 1. However there is only one UART (UART 0) on any Si100x-based SNAP device.

UART1

There is only one UART on the RF30x. It is UART 0, though it comes out on the pins normally used for UART 1.

SPI

SPI uses pins GPIO_4, GPIO_5 and GPIO_6 as MOSI, SCLK, and MISO, respectively. Additionally, you will need to define a chip select pin for each SPI device. (If you load the RF30x SNAP Engine with the appropriate Si100x firmware, the SPI interface shifts back to GPIO_12, GPIO_13, and GPIO_14, as MOSI, SCLK, and MISO, consistent with other SNAP Engines. You will no longer have access to SPI on GPIOs 4-6 if you do this.)

I^2C

Two pins can optionally be used for I^2C: pins 2 and 3 (GPIO_0 and GPIO_1, for SDA and SCL, respectively). Use external pull-up resistors to VCC. Resistors on the order of 10 KΩ work well. **NOTE –** these are not the *hardware* I^2C pins. **SNAPpy I^2C is done via software emulation**.

PWM

You can configure up to six Pulse Width Modulation (PWM) outputs, assigned to the available GPIO pins of your choice, except GPIO_16 through GPIO_18. (If you load the RF30x SNAP Engine with the appropriate Si100x firmware, GPIOs 11-14 can be used for PWM.)

Virtual Machine Speed

RF30x firmware SNAP 2.4 Instructions Per Second (IPS): 3300

Sleep

If using the Si100x firmware on an RF30x SNAP Engine and you need to use a low-power sleep mode, you should make sure pin 13 (GPIO_11) is high before sleeping. Setting this pin low activates the external memory, causing it to draw current while the node sleeps. The RF300 includes a crystal to regulate sleep timings, so you must use only sleep modes 0 and 1. NV Parameter 65 has no effect.

Alternate Radio Trim settings:

NV Parameter 63 is used on the Si100x. The default value is 183 but you can override this by saving a new value in NV #63. For the definition of the trim value, refer to the Silicon Labs datasheets.

Vendor-specific settings:

NV Parameter 64 is used on the RF30x.

Bit 0x0001 – For the RF30x, the bit should be clear (set to 0). See the Si100x section for details about the setting.

Encryption

RF30x SNAP Engines do have AES-128 firmware available. The default firmware for the nodes does not include AES-128 capability, but you can load AES-128-capable firmware into the modules. All RF30x SNAP Engines have Basic encryption available.

Pin Configuration of an Si1000 in SNAP Engine Format (RF300/RF301)

Engine Pin	SNAPpy IO	Name	Description
1		GND	Power Supply
2	10	GPIO0 ADC17 P2.1	GPIO_0 or ADC17 or I^2C SDA
3	11	GPIO1 ADC18 P2.2	GPIO_1 or ADC18 or I^2C SCL
4	12	GPIO2 ADC19 P2.3	GPIO_2 or ADC19
5	13	GPIO3 ADC20 P2.4	GPIO_3 or ADC20
6	14	GPIO4 ADC21 P2.5	GPIO_4 or ADC21 or SPI MOSI
7	15	GPIO5 ADC22 P2.6	GPIO_5 or ADC22 or SPI SCLK
8	0	GPIO6 ADC0 P0.0 V_{REF}	GPIO_6 or ADC0 or INT or external voltage reference or SPI MISO
9	4	GPIO7 ADC5 P0.5 UARTRX	GPIO_7 or ADC5 or INT or UART0_RX
10	3	GPIO8 ADC4 P0.4 UARTTX	GPIO_8 or ADC4 or INT or UART0_TX
11	2	GPIO9 ADC3 P0.3 CTS	GPIO_9 or ADC3 or INT or UART0_CTS
12	1	GPIO10 ADC2 P0.2 RTS	GPIO_10 or ADC2 or INT or UART0_RTS
13	(9)	(GPIO11 ADC16 P2.0)	Not Available[12]
14	(8)	(GPIO12 ADC15 P1.7)	Not Available
15	(6)	(GPIO13 ADC13 P1.5)	Not Available
16	(7)	(GPIO14 ADC14 P1.6)	Not Available[13]
17	5	GPIO15 ADC6 P0.6 CNVSTR	GPIO_15 or ADC6 or INT or external convert start input for ADC0
18	16	GPIO16 P2.7	GPIO_16[14]
19	17	GPIO17 (GPIO_0) [15]	GPIO_17
20	18	ANT_A	GPIO_18 (output only)
21		VCC	Power Supply
22		C2D	Background Debug Communications
23		RESET*	Module Reset, Active Low
24		GND	Power Supply

[12] Pins 13-16 are not available for use on the RF30x, and should not be tied to any hardware on devices you design. You can load an RF30x with the Si100x firmware and have access to these pins, including SPI MOSI on GPIO_12, SPI SCLK on GPIO_13, and SPI MISO on GPIO_14, with interrupts on those three pins as well. However you lose access to the external memory on the RF30x, significantly reducing your available code space. If using the Si100x firmware, GPIO_4, GPIO_5, and GPIO_6 do not have SPI capabilities.
[13] The SNAP code will respond to attempts to read GPIO_14, and will even trigger GPIN-hooked events if monitored. However if you connect external hardware to this pin, you might end up disabling the external memory on the RF30x, rendering the chip inoperable. (Don't do that.)
[14] GPIO_16 has limited drive or sink strength, as it routes through a 1 KΩ resistor. The signal from (or to) GPIO_16 can also be read from (or driven into) Engine pin 22, the debug pin, to route around this resistor instead.
[15] This is GPIO_0 of the underlying *radio* hardware, and is unrelated to the GPIO_0 of the SNAP Engine. Refer to the EZRadioPRO documentation for details.

Freescale MC13224 chip

This section applies if you are running SNAP on a Freescale MC13224 chip in your own hardware. If you instead are running SNAP on a Synapse SM700 module, refer also to the section following this one.

IO pins

The MC13224 supports 64 GPIO input/output pins, referenced as GPIO_0 through CPIO_63.

Wakeup pins

Four pins, GPIO_26 through GPIO_29, can be configured to wake the module from sleep. Note that these pins *automatically* become inputs when entering sleep. Four other pins, GPIO_22 through GPIO_25 *automatically* become outputs when entering sleep (this behavior is not under software control).

Analog inputs

Eight pins support ADC usage. GPIO_30 through GPIO_37 can be read as ADC0 through ADC7, respectively. The ADCs can be read against VCC or either of two external reference voltages.

UART0

Four pins support UART 0. GPIO_14 and GPIO_15 perform as TX and RX, respectively, and GPIO_16 and GPIO_17 perform as CTS and RTS, respectively. If you do not need RTS/CTS signals, then those pins are available for other uses. The serial port requires 8 data bits, and ignores any other value set in the initUart() function. Each serial port supports 1 or 2 stop bits, and Odd, Even, or No parity.

UART1

Four pins support UART 1. GPIO_18 and GPIO_19 perform as TX and RX, respectively, and GPIO_20 and GPIO_21 perform as CTS and RTS, respectively. If you do not need RTS/CTS signals, then those pins are available for other uses. The serial port requires 8 data bits, and ignores any other value set in the initUart() function. Each serial port supports 1 or 2 stop bits, and Odd, Even, or No parity.

SPI

Three pins can optionally be used for SPI. GPIO_5, GPIO_6, and GPIO_7 are MISO, MOSI, and SCK, respectively.

NOTE – these are not *hardware* SPI pins. **SNAPpy SPI is done via software emulation**.
You will also need one "SPI Chip Select" pin *per external SPI device*. Any available IO pin can be used for this purpose.

I^{2}C

Two pins can optionally be used for I^{2}C. GPIO_12 and GPIO_13 are SCL and SDA, respectively.
NOTE – these are not *hardware* I^{2}C pins. **SNAPpy I^{2}C is done via software emulation**.

PWM
Three pins support hardware PWM, refer to GPIO_8, GPIO_9, and GPIO_10 (These pins are also referred to as TMR0, TMR1, and TMR2 in the MC1322x data sheets).

Serial rates
1200-65535 bps as well as 115.2Kbps (115.2Kbps is specified using a "baudrate" of 1).

Network IDs
The MC13224 hardware does not function properly with all network IDs. An MC13224 node set to a network ID that fits the pattern 0xn2nn or 0xnAnn will not be able to receive radio transmissions, though it can still send them. This is an issue with the underlying Freescale radio.

For example:
Network ID 0xFADE does not work. Network ID 0xFBDE does work.

Platform-Specific SNAPpy Functionality

Noise Floor (NV Parameter 33):
You can adjust NV Parameter 33 on MC13224 nodes to tune the node's sensitivity to background noise to prevent improper triggering of the Carrier Sense and Collision Detect features (if they have been enabled). The parameter is specified in negative dBm, with a range from 0 to 127. Setting the value too high (closer to 127) will cause the unit to interpret the background "noise floor" as real communications, and can prevent the node from broadcasting if Carrier Sense and/or Collision Detect are active. Setting it too low will render the Carrier Sense and Collision Detect features useless.

The default value is 57. Changes take effect only after rebooting.

Clock Regulator (NV Parameter 65):
MC13224 implementations offer sleep modes that use the external crystal (if available), and other sleep modes that do not rely on an external crystal. NV Parameter 65 allows you to regulate the internal clock (the one that does not use a crystal) to adjust for differences in your hardware and environment. For most accurate timing, you may need to adjust this value depending on the ambient temperature and the operating voltage of your node.

The range for the value is 0-0xFFFF, and the default value is 0x1F8F. The change takes effect only after the radio restarts, which can be caused by several things. For the greatest predictability, it is recommended that you reboot the node after setting this parameter.

This parameter has no effect on odd-numbered sleep modes.

Built-in function getInfo**():**

The values returned by the getInfo() function for the first four parameters vary by platform. For nodes based on the MC1322x, getInfo() will return the following enumerations:

Command	Value
`getInfo(0)`	2

Command	Value
`getInfo(1)`	0

Command	Value
`getInfo(2)`	4

Command	Value
`getInfo(3)`	9

Built-in function peek**():**

The MC1233x modules use a 32-bit architecture, so the signature for the `peek()` function has changed to accommodate the larger address space and the potential for return values of different sizes. The new function signature is:

```
peek(addrHi, addrLow, word)
```

The addrHi and addrLow parameters are 16-bit values. The word parameter determines the size and character of the return value, with the following options valid:

- word = 0: return value is one byte, in the range 0 to 255 (0x00 to 0xFF).
- word = 1: return value is one 16-bit signed integer, in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 2: peeks a 32-bit value, returning the high value and preserving the low value, which can be subsequently retrieved with a peek of the same address using word = 4, or with a simple peek() command with no parameters. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 3: peeks a 32-bit value, returning the low value and discarding the high value. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 4: performs no peek, but returns the value previously stored when a peek with word = 2 was performed. If no peek with word = 2 has been performed, the result of this function is undefined. Each new peek with word = 2 replaces the previous stored value; there is no stacking or queueing of results. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).

Using peek() with no parameters is a shortcut for performing the peek with word = 4. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).

Peeks that return values larger than one byte must be appropriately word-aligned. In other words, the addrLow value must be even for a peek of a 16-bit value, and must be a multiple of 4 for any of the 32-bit result peeks. If you are not appropriately word-aligned, the result of your peek is undefined.

Built-in function poke**():**
The MC1233x modules use a 32-bit architecture, so the signature for the `poke()` function has changed to accommodate the larger address space and the potential for different value sizes being poked. The new function signature is:
`peek(addrHi, addrLow, word, data)`
with word = 0 or word = 1, or
`peek(addrHi, addrLow, word, dataHi, dataLow)`
with word = 2.

The addrHi and addrLow parameters are 16-bit values. The word parameter determines the size and character data being poked:

- word = 0: data value is one byte, in the range 0 to 255 (0x00 to 0xFF).
- word = 1: data value is one 16-bit signed integer, in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 2: pokes a 32-bit value, specified as two data values in dataHi and dataLow.

Pokes that store values larger than one byte must be appropriately word-aligned. In other words, the addrLow value must be even for a poke of a 16-bit value, and must be a multiple of 4 for a 32-bit poke. If you are not appropriately word-aligned, the result of your poke is undefined, and may result in system issues with your module.

Built-in functions peekRadio**() and** pokeRadio**():**
The `peekRadio()` and `pokeRadio()` functions are not implemented on this platform.

Built-in functions readAdc**():**
The ADC on an MC13224 returns a 12-bit value (0x0000 to 0x0FFF, 0 to 4096), with 8 to 9 bits of precision. Most other SNAP platforms return 10-bit values from their ADCs. In order to scale these values to match, you can shift right two bits or divide by four, without any loss of precision.

On an MC13224, a call to `readAdc()` clears any state set on a pin by the `setPinDir()` function to define the pin as either a digital input or output. If your design requirements mandate that a pin be able to function as both an ADC and either a digital input or output, you must use `setPinDir()` to define the digital state of the pin between `readAdc()` calls.

The following table indicates which "virtual" channel to use, depending on which voltage reference you want to use, and which analog channel you want to read.

For example, if you wanted to take a reading from the ADC3 pin and use the external VREF1 pin as a reference voltage, you would call readAdc(12).

ADC signal	GPIO pin	VCC Channel	VREF1 Channel	VREF2 Channel
ADC0	30	0	9	18
ADC1	31	1	10	19
ADC2	32	2	11	20
ADC3	33	3	12	21
ADC4	34	4	13	22
ADC5	35	5	14	23
ADC6	36	6	15	24
ADC7	37	7	16	25
Battery	(internal)	8	17	N/A

Built-in functions setPinDir**():**
The state of a pin set using the `setPinDir()` function is lost on GPIO_30 through GPIO_37 if you perform a `readAdc()` function on the pin. If your design requirements mandate that a pin be able to function as both an ADC and either a digital input or output, you must use `setPinDir()` to define the digital state of the pin between `readAdc()` calls.

Similarly, using `setPinDir()` on GPIO_38 through GPIO_41 removes those pins' abilities to function as VREF pins for the ADCs. If your design requirements mandate that a pin be able to function as both an ADC VREF and either a digital input or output, you must use `setPinDir()` to define the digital state of the pin before reading from or writing to it, and then use pokes to reset keepers on the pins before making any use of the pin as a VREF. Please reference the Freescale MC1322x reference manual for details on the GPIO_PAD_KEEP1 register.

Built-in functions setPinPullup**():**
The `setPinPullup()` function does not apply a pull-up to GPIO_30 through GPIO_41. No internal pull-ups are available on these pins, and it is not recommended that external pull-ups be applied as they do not function well with the keepers that must be applied to those pins to allow them to function as digital inputs or outputs rather than ADCs or VREFs. If your design requirements mandate that an input have a pull-up or pull-down resistor, it would be best to use a different pin.

Built-in functions setPinSlew**():**
The `setPinSlew()` function configures the pin's hysteresis rather than setting a slew rate.

Built-in functions setRadioRate**():**
The `setRadioRate()` function does not do anything on the MC13224. The only radio rate available is the default rate of 250 Kbps, compatible with other 2.4 GHz-based SNAP nodes.

Built-in functions sleep**():**
There are four `sleep()` modes on the MC13224 module. Even-numbered sleep modes do not require that an external 32 kHz crystal be connected, and are less accurate with their timing. (The internal clock can be regulated on a node-by-node basis, if necessary, using NV Parameter 65.) Odd-numbered sleep modes provide very accurate timing, but require the presence of the external crystal.

Sleep Mode	Details
0, 1	• Fast recovery • GPIO states are maintained during sleep[16] • Highest current usage
2, 3	• Fast recovery • GPIO states are NOT maintained (though they are reset on waking)

Negative durations for sleep values are valid for these seven enumerations. The sleep durations listed here represent the time the node spends sleeping, and do not include the node's recovery time.

Ticks enumeration	Sleep duration
-1	15.625 ms
-2	31.25 ms
-3	62.5 ms
-4	125 ms
-5	250 ms
-6	500 ms
-7	1000 ms

The first odd-mode sleep (using the external crystal) after a reboot of the node requires additional time to enable the crystal.

[16] Pins GPIO_22, GPIO_23, GPIO_24, and GPIO_25 will always shift to being outputs while the node is sleeping in all sleep modes. Pins GPIO_26, GPIO_27, GPIO_28, and GPIO_29 will always shift to being inputs while the node is sleeping in all sleep modes.

Memory Usage

SNAP Protocol Memory Usage:

Global Buffer Pool:	20
UART Budget:	6
Mesh Routing Budget:	6
RPC Budget:	6
Radio Budget:	6
STDOUT Budget:	4

SNAPpy Virtual Machine Memory Usage:

Number of Tiny Strings:	32
Tiny String Size:	up to 16 characters
Number of Medium Strings:	16
Medium String Size:	up to 255 characters
Global Variables:	128
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

Virtual Machine Speed
SNAP 2.4 Instructions Per Second (IPS): 667914

Reserved Hardware

Internal timer TMR3 is used to generate the 1 millisecond time base for SNAP. Timers TMR0, TMR1, and TMR2 remain available for use.

GPIO pins GPIO_42, GPIO_44, and GPIO_45 are normally available for use. However, if you set the Power Amplifier (PA) Feature Bit (NV Parameter #11 bit 0x0100), then these pins will be used for PA control instead, as follows:

> GPIO_42 will enable power to the external amplifier. Basically, GPIO_42 will be high unless the chip enters sleep mode.
>
> GPIO_44 and GPIO_45 will be used as the TX_ON and RX_ON signals.

Synapse SM700 Surface-Mount Module

The SM700 surface-mount module is based on the Freescale MC13224 chip. All the details described for the MC13224 in the previous section apply to the SM700 as well, with these exceptions and elaborations.

getInfo() return values
A call to built-in function getInfo(0) returns 0 (zero, meaning Synapse) on an SM700 module.

Sleep
The SM700 includes an external 32kHz crystal, so you should use odd-numbered sleep modes for the most accurate sleep timing. You can still use sleep modes 0 and 2, if you prefer, but there is no power-consumption advantage, and the timing of your sleep will be adversely affected.

Feature Bits
The SM700 defaults to enforcing FCC TX Power restrictions. You can set the 0x200 bit of NV Parameter #11 (Feature Bits) to enforce ETSI restrictions instead.

When enforcing FCC restrictions, the SM700 reduces power to minimum on channels 0 and 1, and disables packet transmission on channel 15 entirely (you can call setChannel(15) on an SM700, but the end result is a "receive only" node).

When enforcing ETSI restrictions, the SM700 reduces power on all channels to minimum (as if you had invoked txPower(0) explicitly). This includes channel 15, which remains usable for packet transmission under ETSI rules.

IO pins
The SM700 supports 46 GPIO input/output pins.

GPIO_0 through GPIO_41 and GPIO_46 through GPIO_49 are accessible.

GPIO_42 through GPIO_45 are not accessible. GPIO_42, GPIO_44, and GPIO_45 are always used inside the module for Power Amplifier control.

GPIO_50 through GPIO_63 are not accessible.

The table on the following page summarizes the IO mapping on the SM700 module.

SM700 Port Pin mappings

Module Pin	Processor Port Pin	SNAPpy IO
1	Ground	
2	Ground	
3	Ground	
4	GPIO_39 ADC2_VREFL	39
5	GPIO_41 ADC1_VREFL	41
6	GPIO_40 ADC1_VREFH	40
7	GPIO_38 ADC2_VREFH	38
8	GPIO_30 ADC0	30
9	GPIO_31 ADC1	31
10	GPIO_32 ADC2	32
11	GPIO_33 ADC3	33
12	VCC	
13	GPIO_34 ADC4	34
14	GPIO_35 ADC5	35
15	GPIO_36 ADC6	36
16	GPIO_37 ADC7	37
17	GPIO_49	49
18	GPIO_48	48
19	GPIO_47	47
20	GPIO_46	46
21	GPIO_21 UART1_RTS	21
22	Ground	
23	GPIO_20 UART1_CTS	20
24	GPIO_19 UART1_RX	19
25	GPIO_18 UART1_TX	18
26	GPIO_17 UART0_RTS	17
27	GPIO_16 UART0_CTS	16
28	GPIO_13 I^{2}C_SDA	13
29	CPIO_12 I^{2}C_SCL	12
30	GPIO_11	11

Module Pin	Processor Port Pin	SNAPpy IO
31	VCC	
32	GPIO_10 [17]	10
33	GPIO_9 [18]	9
34	GPIO_8 [19]	8
35	GPIO_7 SPI_SCK	7
36	GPIO_14 UART0_TX	14
37	GPIO_15 UART0_RX	15
38	Ground	
39	GPIO_6 SPI_MOSI	6
40	GPIO_5 SPI_MISO	5
41	GPIO_4	4
42	GPIO_3	3
43	GPIO_2	2
44	GPIO_1	1
45	GPIO_0	0
46	GPIO_29 INT KBI_7	29
47	COIL_BK[20]	
48	GPIO_28 INT KBI_6	28
49	RESET*	
50	LREG_BK_FB	
51	Ground	
52	GPIO_27 INT KBI_5	27
53	GPIO_26 INT KBI_4	26
54	GPIO_25 KBI_3[21]	25
55	GPIO_24 KBI_2	24
56	GPIO_23 KBI_1	23
57	GPIO_22 KBI_0	22
58	Ground	
59	Ground	
60	Ground	

[17] GPIO_10 can be used for hardware PWM (TMR2).
[18] GPIO_9 can be used for hardware PWM (TMR1).
[19] GPIO_8 can be used for hardware PWM (TMR0).
[20] COIL_BK and LREG_BK_FB (pins 47 and 50) can be used to connect a buck regulator to reduce system power consumption when running on batteries. See the Freescale MC1322x documentation for details.
[21] KBI_3 through KBI_0 are considered outputs for the keyboard interface. They do not function as interrupts capable of waking a sleeping node (in spite of their name).

STMicroelectronics STM32W108xB chip

This section applies if you are running SNAP on a STMicroelectronics STM32W108CB or STM32W108HB chip. At the time of this writing there is no SNAP Engine or other Synapse module based on this chip.

IO pins

The STM32W108CB supports 24 input/output pins, referenced as IO 0 through IO 23.

Port Pin	SNAPpy IO	Port Pin	SNAPpy IO	Port Pin	SNAPpy IO
PA0	0	PB0	8	PC0	16
PA1	1	PB1	9	PC1	17
PA2	2	PB2	10	PC2	18
PA3	3	PB3	11	PC3	19
PA4	4	PB4	12	PC4	20
PA5	5	PB5	13	PC5	21
PA6	6	PB6	14	PC6	22
PA7	7	PB7	15	PC7	23

The STM32W108HB supports 24 IO, but only 18 of them are brought out to external pins due to the smaller 40-pin package. The following table shows the pins that *are not available for use* if you use the smaller package/lower pin count chip. For more details refer to the manufacturer's datasheet.

Pin Name	Functions	SNAPpy IO Number
PA6	PA6, TIM1C3	6
PA7	PA7, TIM1C4, REG_EN	7
PB0	PB0, VREF, IRQA, TRACECLK, TIM1CLK, TIM2MSK	8
PB5	PB5, ADC0, TIM2CLK, TIM1MSK	13
PC6	PC6, OSC32B, TX_ACTIVE*	22
PC7	PC7, OSC32A, OSC32EXT	23
VDD_PADS		N/A
VDD_SYNTH		N/A

The remainder of this document will present the chip-specific details in a "STM32W108CB-centric" fashion.

Wakeup pins

All 24 pins, IO 0 through IO 23, can be configured to wake the chip from sleep. Note that these pins *automatically* wake the chip on *any* transition, there is no notion of "configurable polarity" on this particular hardware.

To repeat – wakeup **polarity** on the STM32W108 is not under software control.

Analog inputs
Six pins support ADC usage.

ADC	STM32W108 Signal Name	SNAPpy IO
0	PB5	13
1	PB6	14
2	PB7	15
3	PC1	17
4	PA4	4
5	PA5	5

For more about using the ADC channels on this part, refer to the text on readAdc() later in this section of the document.

UART
Four pins support UART 0. IO 9 and IO 10 perform as TX and RX, respectively, and IO 12 and IO 21 perform as CTS and RTS, respectively. If you do not need RTS/CTS signals, then those two pins are available for other uses. (If you do not need a UART at all, then those two pins can be used for other purposes as well). The serial port supports 7 or 8 data bits, and ignores any other value set in the initUart() function. The serial port supports 1 or 2 stop bits, and Odd, Even, or No parity. Hardware limits the supported baud rates to to >= 300 and <= 921600 bps.

NOTE – There is only one UART on this platform.

SPI
Three pins can optionally be used for SPI. IO 1, IO 0, and IO 2 are MISO, MOSI, and SCK, respectively.

NOTE – these are *hardware* SPI pins on the chip, but **SNAPpy SPI is done via software emulation**. You will also need one "SPI Chip Select" pin *per external SPI device*. Any available IO pin can be used for this purpose.

I^{2}C
Two pins can optionally be used for I^{2}C. By default, IO 2 and IO 1 are used for SCL and SDA, respectively. These two pins were chosen to match the physical hardware.

NOTE – SNAPpy I^{2}C is done via software emulation. Because of this, on the STM32W108xB the I^{2}C pins can be dynamically relocated to other pins, refer to the writeup on i2cInit() later in this section.

PWM

Twelve pins support hardware PWM, refer to the following table. Note that there are really only 8 PWM generators on this chip, four of them have the ability to be routed to one of two different pins (only one of the two pins can be a PWM output at a time). So keep in mind there are 8 total PWM generators, not 12.

Timer	Channel	Output Pin	SNAPpy IO	Alternate Output Pin	Alternate SNAPpy IO
TIM 1	C1	PB6	14	N/A	N/A
TIM 1	C2	PB7	15	N/A	N/A
TIM 1	C3	PA6	6	N/A	N/A
TIM 1	C4	PA7	7	N/A	N/A
TIM 2	C1	PA0	0	PB1	9
TIM 2	C2	PA3	3	PB2	10
TIM 2	C3	PA1	1	PB3	11
TIM 2	C4	PA2	2	PB4	12

Serial rates

The initUart() built-in supports 300-65535 bps as well as 115.2Kbps (115.2Kbps is specified using a “baudrate” of 1). Other baud rates are attainable using poke() statements.

Platform-Specific SNAPpy Functionality

Radio Calibration Info (NV Parameter 66):
The system automatically maintains calibration data for 16 radio channels (4 bytes per channel). This calibration info is stored in NV Parameter 66.

To see how to *trigger* radio calibration, refer to the descriptions of pokeRadio(5) and pokeRadio(6) later in this section.

Built-in function getInfo**():**
The values returned by the getInfo() function for the first four parameters vary by platform. For nodes based on the STM32W108xB chips, getInfo() will return the following enumerations:

Command	Value
`getInfo(0)`	9

Command	Value
`getInfo(1)`	0

Command	Value
`getInfo(2)`	10

Command	Value
`getInfo(3)`	20

Calling getInfo(0) returns a vendor code. Vendor code 9 means "STMicroelectronics".
Calling getInfo(1) returns a radio code. Radio code 0 means "2.4 GHz 802.15.4".
Calling getInfo(2) returns a CPU code. CPU code 10 means "ARM Cortex-M3".
Calling getInfo(3) returns a platform code. Platform code 20 means "STM32W108xB".

Built-in function i2cInit**():**
On most SNAP platforms, the i2cInit() built-in only accepts a single parameter, controlling the use of internal pull-up resistors.

On the STM32W108xB, you can also call i2cInit() with **three** parameters:
i2cInit(*enablePullups*, *SCL_pin*, *SDA_pin*)

The second and third parameters allow you to "move" the I2C pins to another pair of IO pins on the chip. This opens up the possibility of using both I2C and SPI peripherals in the same design, while at the same time providing compatibility with existing hardware designs (on the physical chip, the hardware I2C and SPI pins *really do* overlap).

Note that calling i2cInit(xxx, 2, 1) is the same as calling i2cInit(xxx) – the pin assignments default to SCL = IO 2 and SDA = IO 1.

Built-in function lcdPlot**():**
Built-in function `lcdPlot()` has no effect in STM32W108xB builds.

Built-in function peek**():**
The STM32W108xB chips use a 32-bit architecture, so the signature for the `peek()` function has changed to accommodate the larger address space and the potential for return values of different sizes. The new function signature is:

```
peek(addrHi, addrLow, word)
```

The addrHi and addrLow parameters are 16-bit values. The word parameter determines the size and character of the return value, with the following options valid:

- word = 0: return value is one byte, in the range 0 to 255 (0x00 to 0xFF).
- word = 1: return value is one 16-bit signed integer, in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 2: peeks a 32-bit value, returning the high value and preserving the low value, which can be subsequently retrieved with a peek of the same address using word = 4, or with a simple peek() command with no parameters. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 3: peeks a 32-bit value, returning the low value and discarding the high value. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 4: performs no peek, but returns the value previously stored when a peek with word = 2 was performed. If no peek with word = 2 has been performed, the result of this function is undefined. Each new peek with word = 2 replaces the previous stored value; there is no stacking or queueing of results. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).

Using peek() with no parameters is a shortcut for performing the peek with word = 4. The return value is in the range -32768 to 32767 (0x0000 to 0xFFFF).

Peeks that return values larger than one byte must be appropriately word-aligned. In other words, the addrLow value must be even for a peek of a 16-bit value, and must be a multiple of 4 for any of the 32-bit result peeks. If you are not appropriately word-aligned, the result of your peek is undefined.

Built-in function **poke**():
The STM32W108xB chips use a 32-bit architecture, so the signature for the `poke()` function has changed to accommodate the larger address space and the potential for different value sizes being poked. The new function signature is:

```
peek(addrHi, addrLow, word, data)
```

with word = 0 or word = 1, or

```
peek(addrHi, addrLow, word, dataHi, dataLow)
```

with word = 2.

The addrHi and addrLow parameters are 16-bit values. The word parameter determines the size and character data being poked:

- word = 0: data value is one byte, in the range 0 to 255 (0x00 to 0xFF).
- word = 1: data value is one 16-bit signed integer, in the range -32768 to 32767 (0x0000 to 0xFFFF).
- word = 2: pokes a 32-bit value, specified as two data values in dataHi and dataLow.

Pokes that store values larger than one byte must be appropriately word-aligned. In other words, the addrLow value must be even for a poke of a 16-bit value, and must be a multiple of 4 for a 32-bit poke. If you are not appropriately word-aligned, the result of your poke is undefined, and may result in system issues with your module.

Built-in functions peekRadio**() and** pokeRadio**():**
The internal radio registers inside the STM32W108xB are not documented by ST. Instead, they supply a binary (no source code) library of routines with which to control the lowest level functionality of the internal radio.

For this reason, the peekRadio() and pokeRadio() functions as implemented in this version of SNAP are somewhat non-standard: they do not provide access to low-level radio *registers*. Instead they provide a means to invoke low-level API functions that normally would not otherwise be available to the user.

All of the "addresses" in the following two tables are a total fabrication (they can be considered "virtual addresses").

peekRadio(*address*)	**What it really does…**
0-4	Always returns 0
5	Returns the result of calling ST_RadioCheckRadio(): 0 if the radio does NOT need to be recalibrated 1 if the radio DOES need to be recalibrated (see pokeRadio(5, xx) in the next table)
Any other "address"	Always returns 0

pokeRadio(*address, value*)	**What it really does…**
0	Invokes ST_RadioSetPowerMode(*value*)
1	Invokes either ST_RadioStartTransmitTone() (when *value* == 1) or ST_RadioStopTransmitTone() (when value == 0)
2	Invokes either ST_RadioStartTransmitStream() (when *value* == 1) or ST_RadioStopTransmitStream() (when *value* == 0)
3	Invokes ST_RadioSetPower(*value*)
4	Invokes ST_RadioSetEdCcaThreshold(*value*)
5	Invokes ST_RadioCalibrateCurrentChannel() (regardless of *value*). You might choose to do this if peekRadio(5) returns 1
6	Invokes ST_RadioSetChannelAndForceCalibration(*value*) Where *value* should be a 802.15.4 channel number 11-26

Built-in function pulsePin**():**
On the STM32W108xB, *negative* durations are in units of approximately 1.4 µs. Here are some requested versus measured pulse timings, taken with a logic analyzer.

Requested Duration	**Measured Pulse Width**
-1	1.42 uS
-2	2.34 uS
-3	3.43 uS
-4	4.42 uS
-5	5.43 uS
-10	10.34 uS
-100	100.42 uS
-10000	10,000.4 mS
-20000	19,999.5 mS
-30000	29,999 mS

Built-in function readAdc**():**

The ADCs on a STM32W108xB return **signed** 16-bit values, but the data sheets state that only 12 bits are significant. So, SNAPpy returns signed 12-bit values in the range -4096 to 4095.

> Note that this is different than many versions of SNAP to date – usually the values returned by the SNAPpy readAdc() built-in are **unsigned** numbers.

There are also optional “divide by 4” (gain = 0.25) “buffers” that can be switched in.

A bigger difference compared to other SNAP ports is that the ADCs on the STM32W108 chips are *always* differential – ADC readings are *always* relative to another signal, even if that signal happens to be GND (analog ground). The possible voltage references are:

GND
VREF / 2 (0.6 volts)
VREF (1.2 volts)
VREG / 2 (0.9 volts)

(VREG on this chip is 1.8 volts, but is not directly selectable as an analog reference).

Between the 6 different ADC pins, the optional “buffers”, and the 4 voltage references, there are many possible sampling combinations. The desired combination is specified by choosing **two** nibbles from the following table, and combining them together in high nibble/low nibble format.

Hexadecimal Value	Normal Meaning (there are exceptions)
0x0	ADC 0 pin, NOT using the divide by 4
0x1	ADC 1 pin, NOT using the divide by 4
0x2	ADC 2 pin, NOT using the divide by 4
0x3	ADC 3 pin, NOT using the divide by 4
0x4	ADC 4 pin, NOT using the divide by 4
0x5	ADC 5 pin, NOT using the divide by 4
0x6	ADC 0 pin, USING the divide by 4
0x7	ADC 1 pin, USING the divide by 4
0x8	GND as a reference
0x9	VREF / 2 (0.6 volts) as a reference
0xA	VREF (1.2 volts) as a reference
0xB	VREG / 2 (0.9 volts) as a reference
0xC	ADC 2 pin, USING the divide by 4
0xD	ADC 3 pin, USING the divide by 4
0xE	ADC 4 pin, USING the divide by 4
0xF	ADC 5 pin, USING the divide by 4

For example, invoking readAdc(0x08) would give a comparision between ADC 0 (code 0x0) and GND (code 0x8). Invoking readAdc(0x80) would compare the same two voltages, but the sign of the result would be negated, since the voltage comparison itself would be reversed in polarity.

To give another example, invoking readAdc(0x12) would return a reading of the differential voltage between the ADC1 and ADC2 pins.

For a call to readAdc(0x23), the internal divide-by-4 "buffer" would not be switched inline. This would increase the resolution, but the voltage being presented to the analog input might be out of range.

> When the input voltages are out of range, the ADC channel will return a full scale value (either -4096 or +4095, depending on the actual voltage polarity).

A call to readAdc(0xCD) would be taking a differential reading between the same two pins, but with the internal divide-by-4-buffer switched in. This would scale the value down by a factor of 4.

Since not all 256 possible combinations (16 possible upper nibbles x 16 possible lower nibbles) are equally useful, a few special code combinations were re-assigned to make it possible to access some hardware combinations referenced in the STM32W108 datasheets.

Special Value	**What it does…**
0xAA	Read VREF relative to VREF_DIV_2 but with the "divide by 4" buffer switched in
0xBB	Read VREG relative to VREF_DIV_2 but with the "divide by 4" buffer switched in
0xCC	Read VREF_DIV_2 relative to VREF_DIV_2 but with the "divide by 4" buffer switched in

Refer to the manufacturer datasheets for more information on the ADC subsystem.

Built-in function setPinPullup**():**
The `setPinPullup()` function enables internal pullup resistors as expected. However, on this hardware you actually have the hardware capability to have a pull up **or** pull DOWN resistor. The "up or down-ness" is controlled by the pin's data output register.

This means for example that the following sequence will result in IO 0 being an input pin with an internal pull-DOWN resistor enabled.

setPinDir(0, False) # NOT an output, so it becomes an input
setPinPullup(0, True) # We DO want it "pulled", the default direction is pull-UP
writePin(0, False) # The pin is still an input, but now it is pulled DOWN instead of up

Built-in function setPinSlew**():**
There is no slew rate control capability in this hardware. Built-in `setPinSlew()` configures the pin as an open drain output (rather than having it do nothing at all).

Built-in function setRadioRate**():**
The `setRadioRate()` function does not do anything on the STM32W108xB. The only radio rate available is the default rate of 250 Kbps, compatible with other 2.4 GHz-based SNAP nodes.

Built-in function setSegments**():**
On the STM32W108xB, `setSegments()` has no effect.

Built-in function sleep**():**
There are three `sleep()` modes on the MC13224 module. Higher numbered sleep modes draw less current, but have increasing limitations. Refer to the following table:

Sleep Mode	Details
0	• Can be timed sleep • Can also be woken by a pin transition (if enabled) • Highest current usage of the three modes
1	• **Cannot** be timed sleep (pin wakeup **only**) • The hardware sleep timer is maintained (even though it cannot be used as a wakeup source)
2	• **Cannot** be timed sleep (pin wakeup **only**) • The sleep timer is not maintained *at all* (counter value is lost) • Lowest current usage of the three modes

Negative durations for sleep values are valid for the following three enumerations. The sleep durations listed here represent the time the node spends sleeping, and do not include the node's recovery time.

Ticks enumeration	Sleep duration
-1	250 ms
-2	500 ms
-3	750 ms
Any other negative value	Ignored (no sleep at all)

Note that these *tick* values only apply in sleep mode 0, the other two modes do not support "timed" sleep at all.

Current savings in sleep

The following measurements were taken with a DiZiC MB851 evaluation board, and so some of the current draw was from the board, not just the chip. However, the **delta** in current readings should be meaningful.

Mode	**Measured current draw**
Awake with radio on (rx(True))	29.1 mA
Awake with radio off (rx(False))	10.3 mA
sleep(0, 0)	1.4 uA
sleep(1, 0)	0.6 uA
sleep(2, 0)	0.3 uA

Built-in function txPwr():

The SNAPpy txPwr() function takes a parameter between 0 and 17, providing access to 18 total power settings. The following table shows how those 18 settings were mapped to the 25 distinct power settings supported by the STM32W108. Refer to the pokeRadio() built-in for a way to access the other 7 settings.

txPwr(*value*)	**ST_RadioSetPower(*value*)**
17	8
16	7
15	6
14	5
13	4
12	2
11	0
10	-2
9	-4
8	-6
7	-8
6	-11
5	-12
4	-14
3	-17
2	-20
1	-26
0	-43

STM32W108CB Port Pin mappings

Chip Pin	Processor Port Pin	SNAPpy IO
1	VDD_24MHZ	
2	VDD_VCO	
3	RF_P	
4	RF_N	
5	VDD_RF	
6	RF_TX_ALT_P	
7	RF_TX_ALT_N	
8	VDD_IF	
9	BIAS_R	
10	VDD_PADSA	
11	PC5, TXACTIVE	21
12	nRESET	
13	PC6, OSC32B, nTXACTIVE	22
14	PC7, OSC32A, OSC32_EXT	23
15	VREG_OUT	
16	VDD_PADS	
17	VDD_CORE	
18	PA7, TIM1_CH4, REG_EN	7
19	PB3, TIM2_CH3, UART_CTS, SC1SCLK	11
20	PB4, TIM2_CH4, UART_RTS, SC1nSSEL	12
21	PA0, TIM2_CH1, SC2MOSI	0
22	PA1, TIM2_CH3, SC2SDA, SC2MISO	1
23	VDD_PADS	
24	PA2, TIM2_CH4, SC2SCL, SC2SCLK	2

Chip Pin	Processor Port Pin	SNAPpy IO
25	PA3, SC2nSSEL, TRACECLK, TIM2_CH2	3
26	PA4, ADC4, PTI_EN, TRACEDATA2	4
27	PA5, ADC5, PTI_DATA, nBOOTMODE, TRACEDATA3	5
28	VDD_PADS	
29	PA6, TIM1_CH3	6
30	PB1, SC1MISO, SC1MOSI, SC1SDA, SC1TXD, TIM2_CH1	9
31	PB2, SC1MISO, SC1MOSI, SC1SCL, SC1RXD, TIM2_CH2	10
32	SWCLK, JTCK	
33	PC2, JTDO, SWO	18
34	PC3, JTDI	19
35	PC4, JTMS, SWDIO	20
36	PB0, VREF, IRQA, TRACECLK, TIM1CLK, TIM2MSK	8
37	VDD_PADS	
38	PC1, ADC3, SWO, TRACEDATA0	17
39	VDD_MEM	
40	PC0, JRST, IRQD, TRACEDATA1	16
41	PB7, ADC2, IRQC, TIM1_CH2	15
42	PB6, ADC1, IRQB, TIM1_CH1	14
43	PB5, ADC0, TIM2CLK, TIM1MSK	13
44	VDD_CORE	
45	VDD_PRE	
46	VDD_SYNTH	
47	OSCB	
48	OSCA	

STM32W108HB Port Pin mappings

Chip Pin	Processor Port Pin	SNAPpy IO
1	VDD_VCO	
2	RF_P	
3	RF_N	
4	VDD_RF	
5	RF_TX_ALT_P	
6	RF_TX_ALT_N	
7	VDD_IF	
8	BIAS_R	
9	VDD_PADSA	
10	PC5, TXACTIVE	21
11	nRESET	
12	VREG_OUT	
13	VDD_PADS	
14	VDD_CORE	
15	PB3, TIM2_CH3, UART_CTS, SC1SCLK	11
16	PB4, TIM2_CH4, UART_RTS, SC1nSSEL	12
17	PA0, TIM2_CH1, SC2MOSI	0
18	PA1, TIM2_CH3, SC2SDA, SC2MISO	1
19	VDD_PADS	
20	PA2, TIM2_CH4, SC2SCL, SC2SCLK	2

Chip Pin	Processor Port Pin	SNAPpy IO
21	PA3, SC2nSSEL, TRACECLK, TIM2_CH2	3
22	PA4, ADC4, PTI_EN, TRACEDATA2	4
23	PA5, ADC5, PTI_DATA, nBOOTMODE, TRACEDATA3	5
24	VDD_PADS	
25	PB1, SC1MISO, SC1MOSI, SC1SDA, SC1TXD, TIM2_CH1	9
26	PB2, SC1MISO, SC1MOSI, SC1SCL, SC1RXD, TIM2_CH2	10
27	SWCLK, JTCK	
28	PC2, JTDO, SWO	18
29	PC3, JTDI	19
30	PC4, JTMS, SWDIO	20
31	PC1, ADC3, SWO, TRACEDATA0	17
32	VDD_MEM	
33	PC0, JRST, IRQD, TRACEDATA1	16
34	PB7, ADC2, IRQC, TIM1_CH2	15
35	PB6, ADC1, IRQB, TIM1_CH1	14
36	VDD_CORE	
37	VDD_PRE	
38	OSCB	
39	OSCA	
40	VDD_24MHZ	

Memory Usage

SNAP Protocol Memory Usage:

Global Buffer Pool:	20
UART Budget:	6
Mesh Routing Budget:	6
RPC Budget:	6
Radio Budget:	6
STDOUT Budget:	4

SNAPpy Virtual Machine Memory Usage:

Number of Tiny Strings:	14
Tiny String Size:	up to 16 characters
Number of Medium Strings:	10
Medium String Size:	up to 128 characters
Global Variables:	128
Concurrent Local Variables:	64
Maximum Call Stack Depth:	8

SNAPpy Script Space: 60K

Performance Metrics

Here are the results of some recent performance measurements, which may help gauge if SNAPpy can address your application's timing requirements.

Time to awaken from sleep (mode 0):
< 51 ms

Time to awaken from sleep (mode 1):
< 51 ms

Time to awaken from sleep (mode 2):
< 51 ms

In other words, the wakeup time is not significantly affected by the sleep mode.

Time to startup from power-on:
< 88 ms

Maximum rate a SNAPpy script can toggle a GPIO pin:
11583 Hz

In other words, each True/False cycle took 86.33 µs.

Keep in mind that as a general rule, SNAPpy scripts **should not be looping**, the above rate is only attainable if the node is doing *nothing else* (for example, no radio or serial port communication).

Maximum rate for readAdc() calls:
Maximum 120 samples/second.

NOTE! – This measurement was taken using a script that did not actually do anything with the data. You will also have to take into consideration any numeric processing required (data thresholding, etc.), as well as the need to actually store the data someplace.

I^{2}C Byte Transfer Time
The actual I^{2}C transfers are done using "bit banging" in software. This was measured using a logic analyzer at 155 µs per byte.

SPI Byte Transfer Time
The actual SPI transfers are done using "bit banging" in software. This was measured using a logic analyzer at 76 µs per byte.

Virtual Machine Speed
SNAP 2.4 Instructions Per Second (IPS): 51630

Reserved Hardware

The internal 1 KHZ timer ("sysTick") inside the ARM processor is used to generate the 1 millisecond time base for SNAP. Hardware timers TIM1 and TIM2 remain available for use (for example, for generating PWM waveforms).

The PA5 pin if held low *at power-on reset* forces entry into a serial boot loader built into the chip.

This is a hardware feature of the chip, and cannot be overridden. This does not mean you cannot use this pin for other purposes, but you definitely should refer to the manufacturer's data sheets and reference designs before doing so.

License governing any code samples presented in this Manual

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.
500 Discovery Drive
Huntsville, Alabama 35806

256-852-7888
877-982-7888
256-852-7862 (fax)

www.synapse-wireless.com