SNAP based Wireless Mesh Networks

The world is currently seeing an exponential growth in the use of wireless networks for monitoring and control in consumer, commercial, industrial, and government markets. Uses range from building automation (lighting, heating, A/C…) to industrial control and machine communication, to medical monitoring, to security applications, to home automation.

Unfortunately, conventional wireless networks require extensive embedded programming skills (typically involving C/C++ and assembly language) to develop and deploy applications that run on the nodes forming the network. Also, conventional wireless networks require significant amounts of time and resources to configure and manage.

By comparison, a wireless network based on the SNAP® software stack is instant-on, self-forming, and self-healing (SNAP is the abbreviation for Synapse Network Application Protocol). The process of creating applications to run on the nodes forming a SNAP based network is fast, efficient, and does not require embedded programming skills. Furthermore, users can administer and manage a SNAP based network without actually having to know anything about wireless networks! This approach makes wireless technology accessible to a much broader range of users.

SNAP

A high-level view of the SNAP software stack is illustrated in Figure 1. This high-performance stack is designed to run efficiently on cost-effective 8-bit microprocessors.

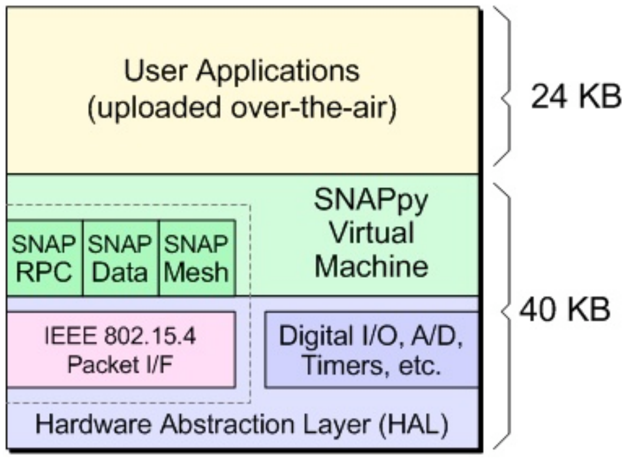
User Applications (uploaded over-the-air)
24 KB
SNAPpy Virtual Machine
SNAP RPC
SNAP Data
SNAP Mesh
IEEE 802.15.4 Packet I/F
Digital I/O, A/D, Timers, etc.
40 KB
Hardware Abstraction Layer (HAL)

Figure 1. The SNAP software stack.

Even though it provides full mesh capabilities, SNAP has a very small memory footprint of only 40 KB, thereby leaving more space for user applications. This is a real consideration in cost-conscious deployment scenarios, because conventional stacks typically consume 60 KB or more. This means that network nodes running these conventional stacks have to move from a 64 KB memory device to a more-expensive 128 KB part in order to provide enough space to store the users' applications.

The SNAP stack includes a Python virtual machine (the combination of SNAP and Python is referred to as SNAPpy). End-user wireless applications are compiled into processor-independent "byte code" that is run on the SNAPpy virtual machine. This means that the same application can be run on any processor without the need for recompilation. Furthermore, SNAPpy code is easy to tokenize, so a single byte can perform the work of 2 to 5 typical machine code bytes. Thus, assuming an average of 1 SNAPpy byte to 3.5 regular machine code bytes, and also assuming a 64 KB memory device, this means that the 24 KB of application code space is actually equivalent to 84 KB.

In addition to the IEEE 802.15.4 physical layer (PHY) hardware (which is not shown here for reasons of simplicity) SNAP also employs the industry-standard 802.15.4 packet interface.

Designed for portability

Written in optimized, portable, industry-standard ANSI C code, the SNAP software stack has been designed from the ground up for portability. This is important because – as an alternative to running on Synapse RF Engine modules as discussed later in this paper – some users wish to run SNAP on the same processor that is being used to drive their main product.

Two key aspects to this portability are the SNAPpy Virtual Machine (as discussed below) and the Hardware Abstraction Layer (HAL) illustrated in Figure 1. The HAL provides a layer of abstraction between the main body of the software stack and the physical world, including the specifics of accessing registers in the processor, reading/writing values to Input/Output (I/O) pins, controlling Analog-to-Digital (A/D) converters, and so forth. This means that the process of porting the stack to a new processor requires modifications only to the "thin slice" of the HAL.

The SNAPpy virtual machine

User applications that run on SNAP based nodes are created in the high-level Python scripting language using the Portal® application development environment (this process is discussed in more detail later in this paper). The combination of SNAP and Python is known as "SNAPpy".

SNAPpy applications are automatically translated into what is known as "byte code" and are then downloaded over-the-air into the wireless network nodes.

As illustrated in Figure 1, the SNAP software stack includes the SNAPpy (Python) Virtual Machine, which executes the byte codes forming the SNAPpy applications. This provides extreme portability, because the SNAPpy Virtual Machine provides a layer of abstraction that separates the applications from the physical hardware. This means that a SNAPpy application executable will immediately run on any processor without requiring any modification or re-compilation.

SNAP based wireless mesh networks

The SNAP protocol can theoretically support up to 16 million nodes in a single network. Since these are mesh networks, there is no single point of failure: any node can talk *directly* to any other node that is in range, and any node can talk *indirectly* to any other node via intermediate nodes.

One key point is that routes between nodes do not have to be pre-configured by the user. SNAP based networks are self-forming (the network establishes itself) – when a new node is powered-up it is automatically integrated into the network. SNAP based nodes are also instant-on – as soon as a node is powered-up, it becomes fully operational in a fraction of a second. Furthermore, SNAP based networks are self-healing – if a node fails for any reason, other nodes will automatically route signals around the failed node.

Python and SNAPpy – No Embedded Programming Required

Let's assume you wish to write a letter to your mother. You *could* write it using Egyptian Hieroglyphics, but – unless you happen to be one of the world's few experts in this area – this would be extremely time-consuming and prone to error. Alternatively, you could quickly draft your message in a modern language such as English.

Similarly, when it comes to creating applications to run on wireless networks, you *could* use the traditional approach of programming the embedded devices using C/C++ or assembly language and debugging your applications using special hardware and software tools, but this requires a high level of wireless network and embedded programming expertise and can be extremely time-consuming.

The alternative is to use the Python programming and scripting language. A modern, high-level language, Python can be learned in just a few days. Programmers using Python report substantial productivity gains and the language encourages the development of higher quality, more maintainable code. In fact, Python was declared *"The programming language of 2007 with the largest increase in user ratings over any other language,"* by the coding standards company TIOBE Software.

The term "SNAPpy" refers to the combination of the Synapse Network Application Protocol (SNAP) and Python. The smallest known implementation of Python, SNAPpy is a subset of the Python language that has been optimized for use in low-power embedded devices and that offers unprecedented productivity in developing embedded applications.

Developing and deploying SNAPpy applications

Synapse Portal is a software application that runs on your PC and that can be used to develop SNAPpy applications and deploy them "over-the-air" to SNAP based nodes. Portal can also be used to configure and manage the network as required, and it can provide additional functions such as data logging, event monitoring, and debugging.

Small, fast, and extremely efficient (it's written in Python), Portal is seen by the network as "just another node". The PC running Portal may be physically connected to any SNAP Node, which subsequently acts as a bridge into the network.

When a group of physical SNAP Nodes are powered-up, they will automatically self-form into a fully-functioning wireless mesh, irrespective of whether or not Portal is currently present on the network. When Portal is connected into the network, it can be used to configure the network and monitor and log network activity as required. If Portal is later disconnected from the network, the network will continue to run quite happily on its own.

SNAPpy applications are created in the form of easy-to-understand Python scripts, which are composed of one or more functions. Any function in any node on the network can be called from Portal or from any other node on the network. Similarly, any node on the network can invoke a script running on Portal.

Most networking environments require the use of multiple applications to perform tasks such as editing code, listing the nodes in the mesh, accessing and indicating the status of nodes, and so forth. By comparison, Synapse Portal provides all of these functions in a single, intuitive, easy-to-use interface. For example, clicking on a node in Portal displays the application script that is running on that node. The user can then call up the source for that script, make a modification, and upload the new code into the target node over-the-air – all in a matter of seconds.

Sleepy nodes and sleepy meshes – ultra-low-power mesh routing

In many environments it is necessary for the nodes forming the network to be powered by batteries, in which case power consumption can be a significant problem. If a node is constantly active (“talking” and “listening”), two AA batteries will be capable of powering it for only a couple of days. This is simply not acceptable in terms of resource requirements (someone changing the batteries) and expense (batteries aren’t cheap) for the vast majority of installations, especially in the case of networks involving hundreds or thousands of wireless nodes.

The solution is for the nodes to alternate between being "awake" for a short amount of time and then entering a "sleep" mode in which they consume dramatically less power. First consider a SNAP based network in its "wide-awake" mode (Figure 2).

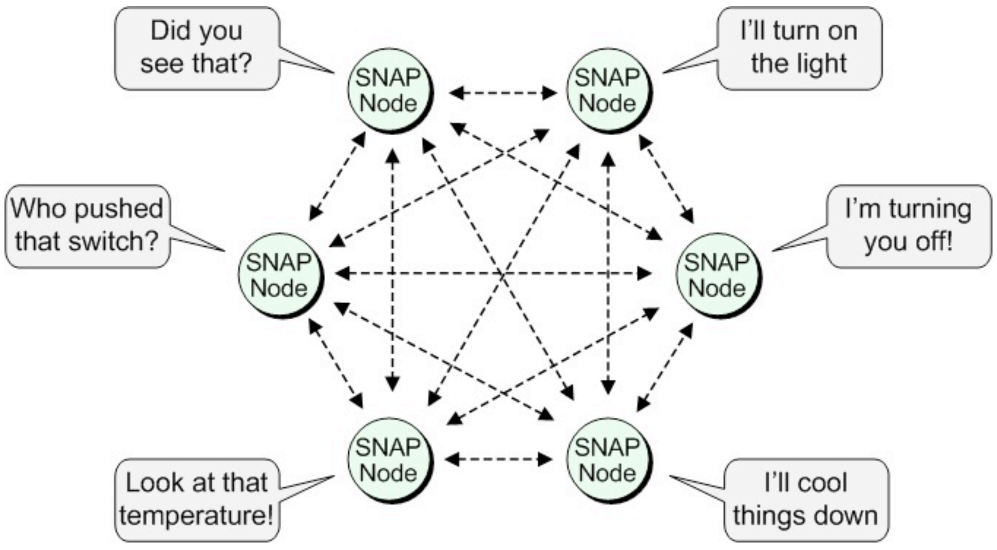

Figure 2. A SNAP based "sleepy mesh" in its "wide awake" mode.

Observe that SNAP Nodes support peer-to-peer communication, which means that they form the mesh network themselves without requiring special router or bridge nodes.

Now, traditional mesh networks find it difficult to fully-implement a network-wide sleep scenario. By comparison, the nodes forming a SNAP based network can easily be configured to implement a "sleepy mesh" in which all of the nodes go to sleep at the same time (Figure 3).

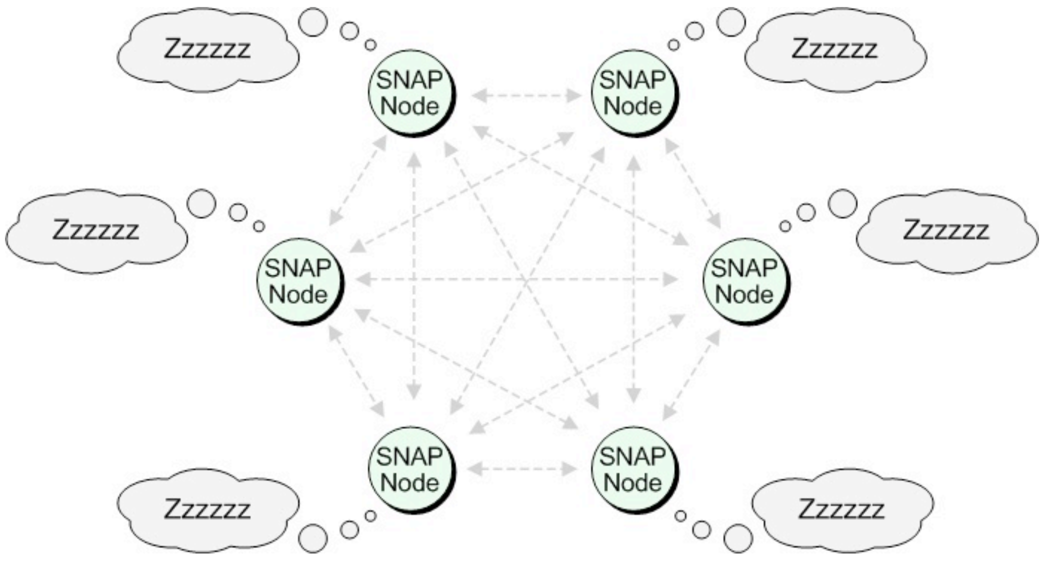

Figure 3. A SNAP based "sleepy mesh" in its sleep mode.

The power savings provided by sleepy nodes in a sleepy mesh can be truly amazing – in fact, depending on the requirements of the target application, the result can be to extend the battery life of each node from a little over two days to more than a year!

Synapse RF Engine Modules

Synapse has created a family of Synapse RF Engine modules with a range of up to three miles. At the time of writing, there are three such modules as follows:

- Un-amplified with an internal F antenna.
- Amplified with an internal F antenna.
- Amplified with an external antenna.

Note that the "amplified" qualifier refers to output (transmit) amplification – all of these modules include input (receive) amplification as standard.

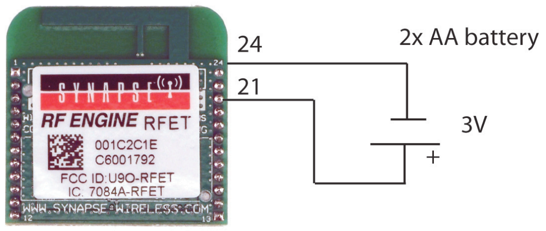

Figure 4. Synapse RF Engine Module.

Pre-loaded with the SNAP software stack, Synapse RF Engine modules are self-contained, instant-on units. As soon as power is applied to such a module (Figure 4), it is immediately integrated into the network and starts routing signals from other nodes as required.

Each Synapse RF Engine module supports nineteen available general purpose I/Os, including up to eight analog inputs with 10-bit ADC and two serial UART ports configurable for control or transparent data.

These modules support extremely low power modes, consuming as little as 2.5 µA with the internal timer running. Also, they provide 60 KB of Flash memory, of which 30 KB is available for user applications that can be downloaded and upgraded over-the-air.

FCC certified on all 16 channels, Synapse RF Engine modules can be used in conjunction with off-the-shelf SNAP Nodes from Synapse; alternatively, they can be used to power custom wireless nodes that have been developed by third-parties.

Custom SNAP based solutions

In addition to off-the-shelf SNAP based solutions, Synapse also offers full-custom design and implementation capabilities. For users who have special SNAP Node or RF Engine requirements, for example, Synapse can help realize their specifications in the shortest amount of time.

Consider a Heating, Ventilation, and Cooling (HVAC) scenario in which a company currently deploys a sophisticated thermostat as illustrated in Figure 5. Assume that this product includes a microcontroller that is used to monitor one or more sensors, accept user input, drive a display, and output signals via a wired connection (using RS232, for example).

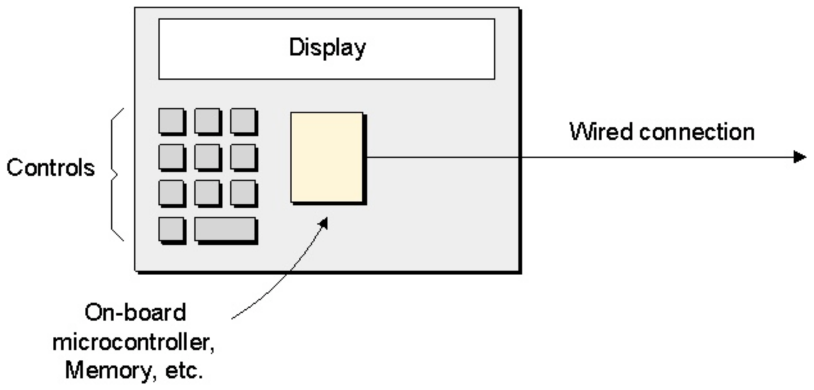

Figure 5. Conventional thermostat with a wired connection to the outside world.

Assuming the HVAC company wished to augment their product with wireless capabilities, one scenario would be to simply modify their existing board to accept a SNAP based Synapse RF Engine plug-in module as illustrated in Figure 6.

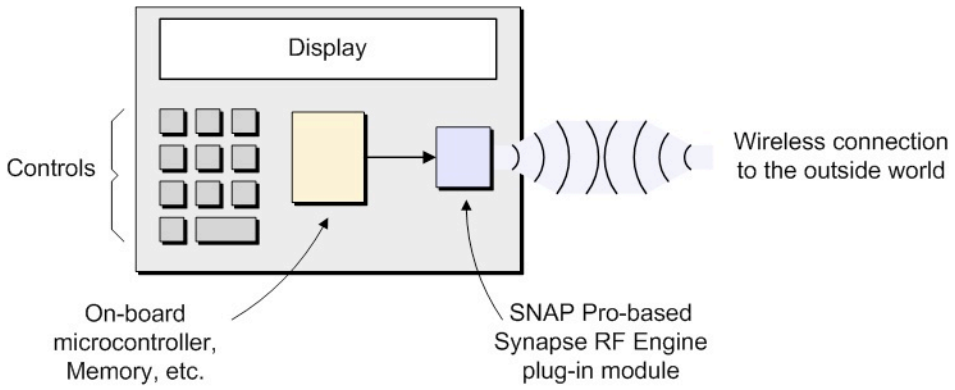
Display
Wireless connection to the outside world
Controls
On-board microcontroller, Memory, etc.
SNAP Pro-based Synapse RF Engine plug-in module

Figure 6. Adding a SNAP based Synapse RF Engine plug-in module to provide a wireless connection to the outside world.

In the case of high-volume and/or price-sensitive projects, however, a fully customized solution may entail implementing the SNAP based functionality directly onto the motherboard as illustrated in Figure 7.

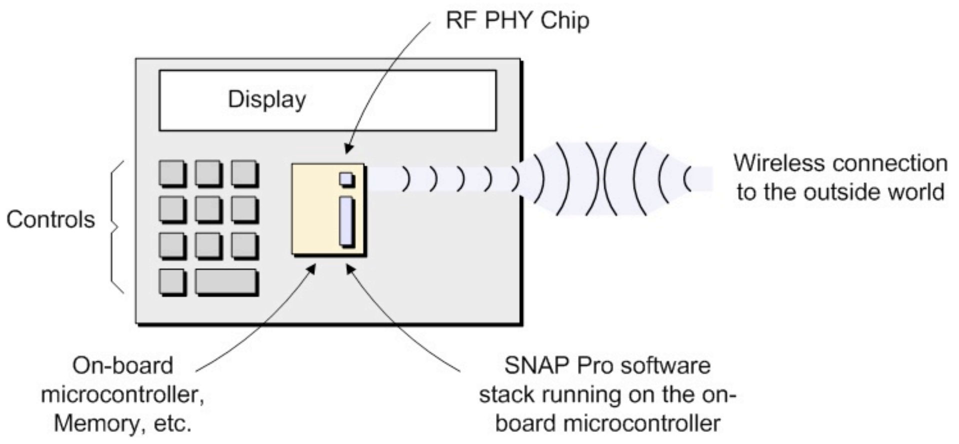
RF PHY Chip
Display
Controls
Wireless connection to the outside world
On-board microcontroller, Memory, etc.
SNAP Pro software stack running on the on-board microcontroller

Figure 7. Running the SNAP software stack on the on-board microcontroller.

In this case, rather than having two processors (the microcontroller for the thermostat and the microprocessor on the RF Engine Module), the PHY chip – and any related RF components – would be mounted on the main board, while the SNAP software stack would be executed on the same microcontroller used to control the thermostat.

Summary

The SNAP software stack re-defines wireless networking. Networks based on this high-performance, low-power, small-memory-footprint stack are self-forming, instant-on, and self-healing.

Using the Synapse Portal application development and network administration software, applications that run on the nodes forming the network can be created or modified quickly and efficiently without requiring embedded programming skills.