SensorNet

Wireless Home Automation and Sensor Network

Matthew Hengeveld
Computing & Computer Electronics
Directed Research Project
Nora Znotinas
Fall 2014/Winter 2015

Table of Contents

Overview

SensorNet is a wireless network designed to interface sensors and automation controls to a central control system, accessible from local networks and the World Wide Web. Using custom, open source code running on multiple microcontroller architectures, sensors and automation controls, called nodes, can interact with their environment, and many different electronic devices in the home. Monitoring and control of nodes is facilitated by the central control system, called the root, through which several interfaces are available. Inexpensive wireless transceivers allow for a low cost per node, as well as a low-latency network that utilizes the same frequency band as WiFi, but does not interfere.

1. Background

Currently, there are several systems on the market that offer similar features as SensorNet provides. Companies such as Belkin, D-Link, Insteon, Nest, Philips, and Skylink each have product lines designed to offer remote access to sensors and automation controls in the home, all in limited forms. Products like the Nest Learning Thermostat and the Philips Hue are both specialized systems, and are limited to one function. Other product lines are offered by companies such as Belkin, Insteon and SkylinkHome that offer a broader set of features. These product lines typically offer only controllable mains voltage plugs, light dimmers and thermostats, as well as a few other specialized products. As well, most of these products rely on each device having its own WiFi connection to the home network. This causes issues with access points not able to handle a high number of connections, excess network traffic, and wireless interference for the whole wireless network.

The 802.11 standard (WiFi) was designed to make high data-rate transfers between computers and network infrastructure. WiFi is expensive to implement, and adds significant cost to low data-rate devices without the requirements of the advanced features that the protocol provides. Integrating WiFi with microcontrollers is expensive – current solutions available include:

Arduino WiFi shield	$80
Arduino Yun	$70
XBee	$35
Intel Galileo	$75

Since the beginning of this project, a new WiFi module with serial communication has come on to the market, and currently can be found for close to $4. While this is a significant reduction in cost compared to the previously mentioned solutions, it is still close to twice the cost of the wireless transceiver used in SensorNet. As well, this module does not address the latency, interference and increased number of connections associated with WiFi.

WiFi was created for transfer of large amounts of data, and comes with advanced, complicated protocols and the significant overhead that accompanies them. Sensors and automation controls have inherently small data requirements. Sensors, such as temperature, humidity, light and hall sensors, typically output from a single bit (on/off) to 24 bits (RGB format colour) of data. Similarly, automation controls typically require from a single bit (on/off) to 24 bits (PWM control of a RGB light). With SensorNet using two bytes for addressing, and one byte for error detection – for a total of about 4-6 bytes –the 802.11 standard has a typical packet size 250 - 375 times greater than that used in SesnorNet.[1]

Bluetooth, and other RF transceivers each have their own disadvantages that make them unsuitable for home automation and sensors: signal range, lack of networking capabilities, complexity, interference and lack of noise rejection.

[1] Given an Ethernet packet size of 1500 octets. http://en.wikipedia.org/wiki/IEEE_802.11

2. Network

SensorNet uses two networks: Wifi, and a custom-designed network protocol for communicating with nodes. The network is based on the Nordic nRF24L01+ 2.4GHz transceiver.

The nRF24L01+ (nRF) operates on the same license-free ISM (Industrial, Scientific, Medical) wireless band as WiFi and Bluetooth, but uses a bandwidth of only 1MHz, compared to 20/40MHz of WiFi. WiFi uses orthogonal frequency-division multiplexing (OFDM), which reduces interference with the nRFs Gaussian frequency-shift keying modulation. Over the 2.4GHz to 2.5GHz frequency range, WiFi typically operates on channels 1, 6 and 11, so that the maximum number of WiFi channels can be utilized without overlap. This leaves 40MHz of bandwidth for use by SensorNet networks (Figure 1)[2] and means that multiple different SensorNet networks can co-exist with WiFi networks, without using the same channels, and reducing possible interference between them. Bluetooth, however, uses the same Gaussian frequency-shift keying as the nRF, but employs advanced frequency-hoping techniques to change its frequency up to 1600 times a second. This frequency-hoping allows Bluetooth to always find a clear channel.

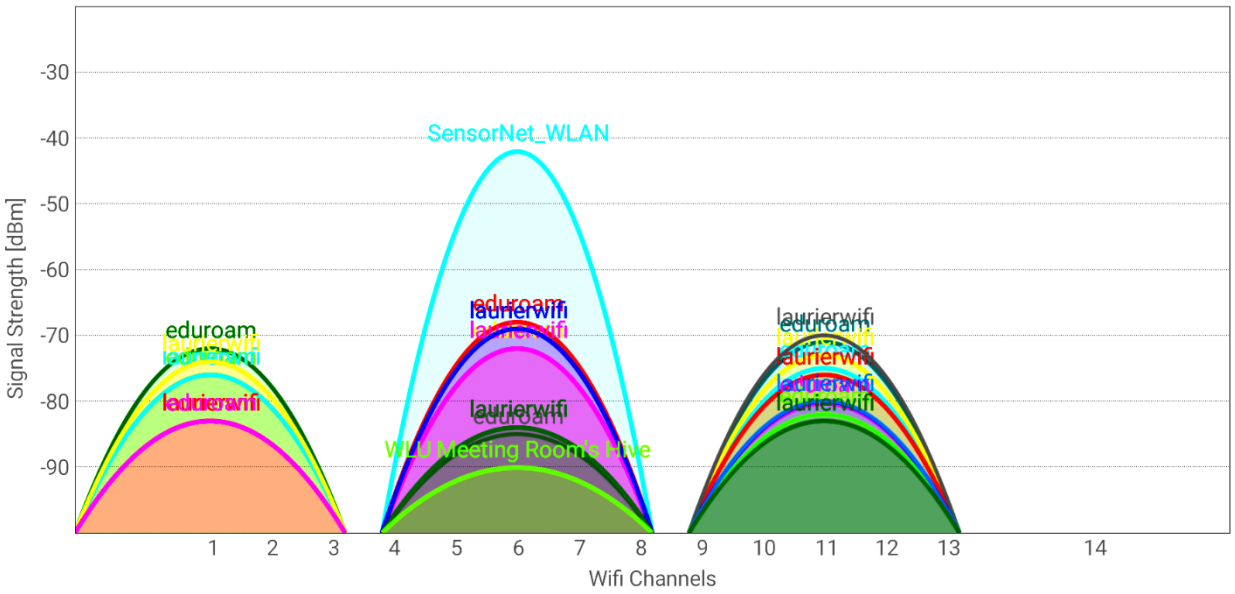

Figure 1. WiFi channels. Actual graph of WiFi networks on Laurier campus (Bricker Academic). Each channel below 13 has 5MHz spacing.

The network protocol was designed in a star topology, where the root connects to each node individually. The nRF transceiver can only operate in half-duplex. Therefore, the network protocol uses a request-reply pattern. The central control, the root, sends requests to nodes. This pattern is necessary

[2] http://en.wikipedia.org/wiki/List_of_WLAN_channels

so that interference does not occur between nodes. Nodes can only contact the root following a request.

The root and each node have a unique 4-byte address. The protocol also has some advanced features, provided by the Nordic nRF "Enhanced Shockburst" hardware on each transceiver. These features include automatic 1-byte CRC check for each packet, dynamic packet payload length, and automatic acknowledgement of received packets. The protocol for SesnorNet was designed to reduce the on-air time for each packet sent (and thus the chance of in-air packet collision and interference), as well as insuring that each packet is recieved.

Typically, a sensor or automation control needs less than the 32 byte maximum of one packet. The makeup of a packet is as follows:

1-byte Preamble	4-byte Address	9-bit Packet Control	0 to 32-byte Payload	1-byte CRC

This results in a packet length of just over 7 bytes to just over 39 bytes. The nRF is configurable for three different air data-rates: 250Kbps, 1Mbps, and 2Mbps. Each rate has their advantages and disadvantages. The higher the data-rate, the lower the on-air time and the less change of interference. However, the higher the data-rate, the lower the signal-to-noise ratio, which results in decreased range and object penetration. An air data-rate of 1Mbps was chosen, which results in a packet on-air time of

$$\frac{\left(8\ bits/_{byte} \times 7\ bytes\right) + 1\ bit}{1Mbps} = 0.000057s$$

$$\frac{\left(8\ bits/_{byte} \times 39\ bytes\right) + 1\ bit}{1Mbps} = 0.000313s$$

or $57\mu s\ to\ 313\mu s$. The dynamic payload length feature of the nRF is used to transmit variable packet lengths without the receiver needing to know the incoming packet length.

Each packet sent using the protocol is required to be acknowledged by the receiver. The auto acknowledgement feature of the nRF is used to implement this requirement, and helps to reduce lost packets, determine if a node is outside the range of the root, and ensure critical packets are received.

3. Nodes

Nodes are physical pieces of hardware, each with a microcontroller and a nRF transceiver. Each node has a unique address in the SensorNet network, as well as several properties, including status, location, and power.

The microcontroller on each node can accommodate several sensors and automation controls. Therefore, it was logical to group sensors and automation controls into modules. Each module is made up of sensors and controls that are associated with each other. For example, temperature, humidity and barometric pressure sensors can all be grouped into an environmental module. On the same node, it is possible to also have a module that includes a 2-channel relay for control of mains sockets. Each sensor and control has an associated 1-byte command, which uniquely identifies it locally on the node, and is sent by the root when requesting it. This allows for up to 32 different sensors and automation controls on a single node. Each module also has a unique address in the SensorNet network. This 5-byte address is a concatenation of the 4-byte node address and the *first, or lowest number,* sensor or control command byte. Figure 2 shows the associations between nodes, modules and sensors, as well as their attributes and the data size of each.

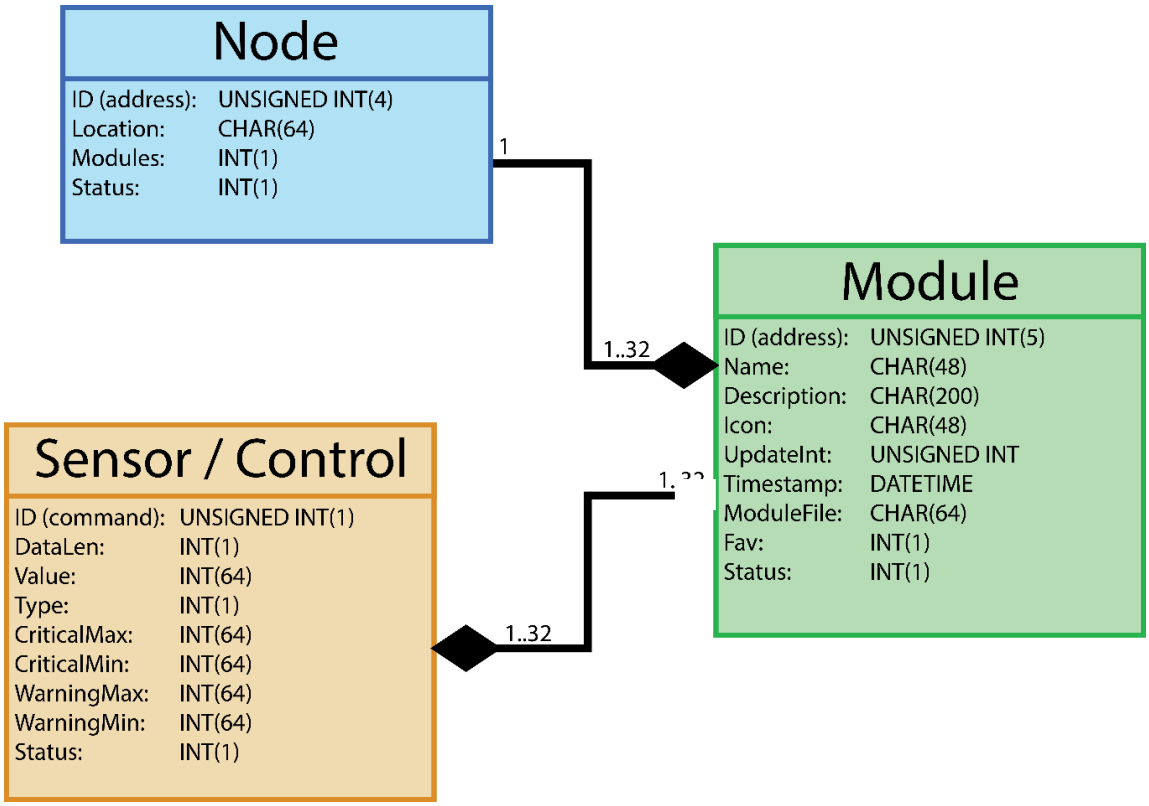

Figure 2. Node association diagram.

4. Control Software

The control software for SensorNet consists of two parts. The first part if the interface software, which consists of a webserver built around the Flask microframework. Flask is written in the Python language, and was chosen for its support on the Raspberry Pi. The second part of the control software is a separate Python process that interfaces with a database, implements timing functions, and acts as a bridge between the Web interface and the nRF network.

The webserver and database/timing/nRF interface process were separated to allow for multi-threading performance. Webserver functions and other unrelated functions can therefore operate synchronously.[3] Communication between both processes uses a fast inter-process messaging system called ZeroMQ, which uses sockets. The database is a lightweight SQL variant called SQLite3. It was chosen over a full SQL database to optimize performance on the Raspberry Pi.

The nRF uses the SPI communication protocol to communicate with the Raspberry Pi. There are several libraries for controlling the GPIO on the Raspberry Pi using Python and C. It was discovered through testing that libraries implemented in Python were very slow to manipulate the Raspberry Pi GPIO, including SPI. Therefore, the bcm2835 C library[4] was used for controlling the nRF. Interfacing C with Python required a specialized version of Python called Cython to wrap the C code. The result is a C library that is callable from Python, with the speed of C.

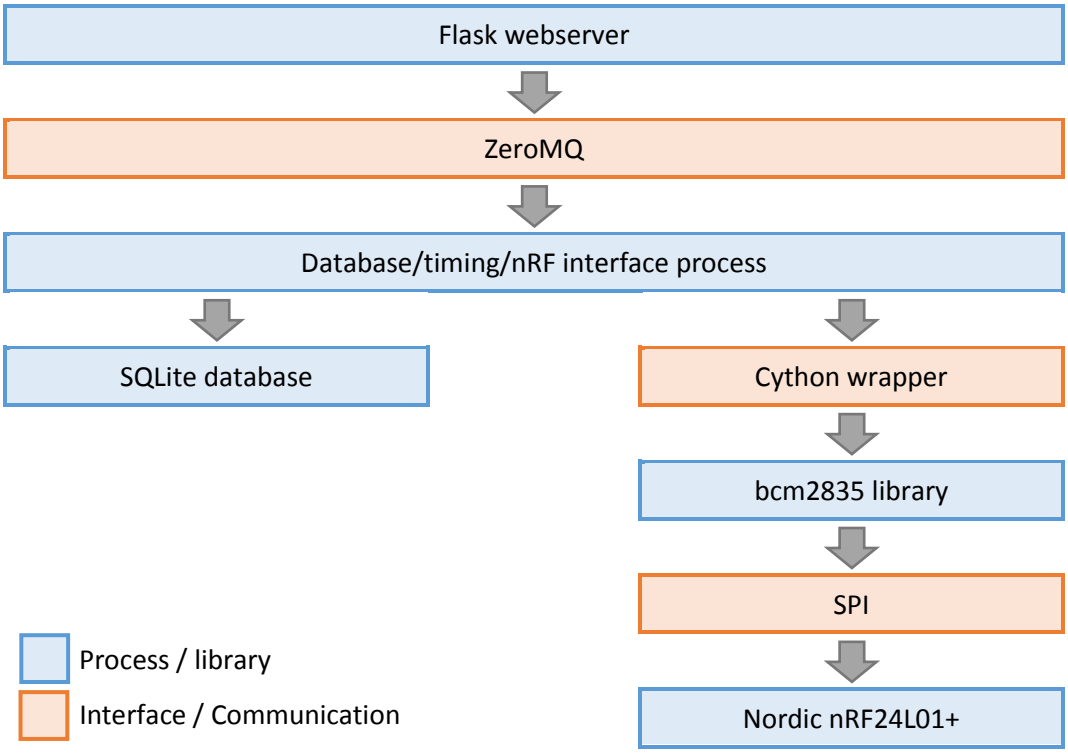

Figure 3. Design of control software.

[3] The Raspberry Pi has a single-core processor, however, the Linux OS uses a time-shared process scheduler.
[4] www.airspayce.com/mikem/bcm2835/

Flask Webserver & Web Interface

The Flask microframework was chosen for SensorNet for multiple reasons. Compared to the popular Apache webserver, Flasks built-in webserver is lightweight, and uses significantly less resources. Also, Flasks native language is Python, which is also very well supported by the Raspberry Pi, and has many libraries. *Note, Flasks webserver is meant for testing purposes, and is not production-ready. However, due to the typically few simultaneous users and low data usage of SensorNets web interface, Flasks built-in webserver showed no ill-effects of being used full-time in testing.*

SensorNets web interface is a liquid design, which allows the interface to adapt to different screen resolutions and orientations, including for mobile smartphones. The liquid design uses CSS and JavaScript to implement a sliding menu, as well as a favorite grid that transitions from one tile wide to three tiles wide upon resizing the web browser.

Using data stored in the database, data analysis can be done. Using the archive of previous values, a graph of changes over time can be constructed, showing trends and anomalies.

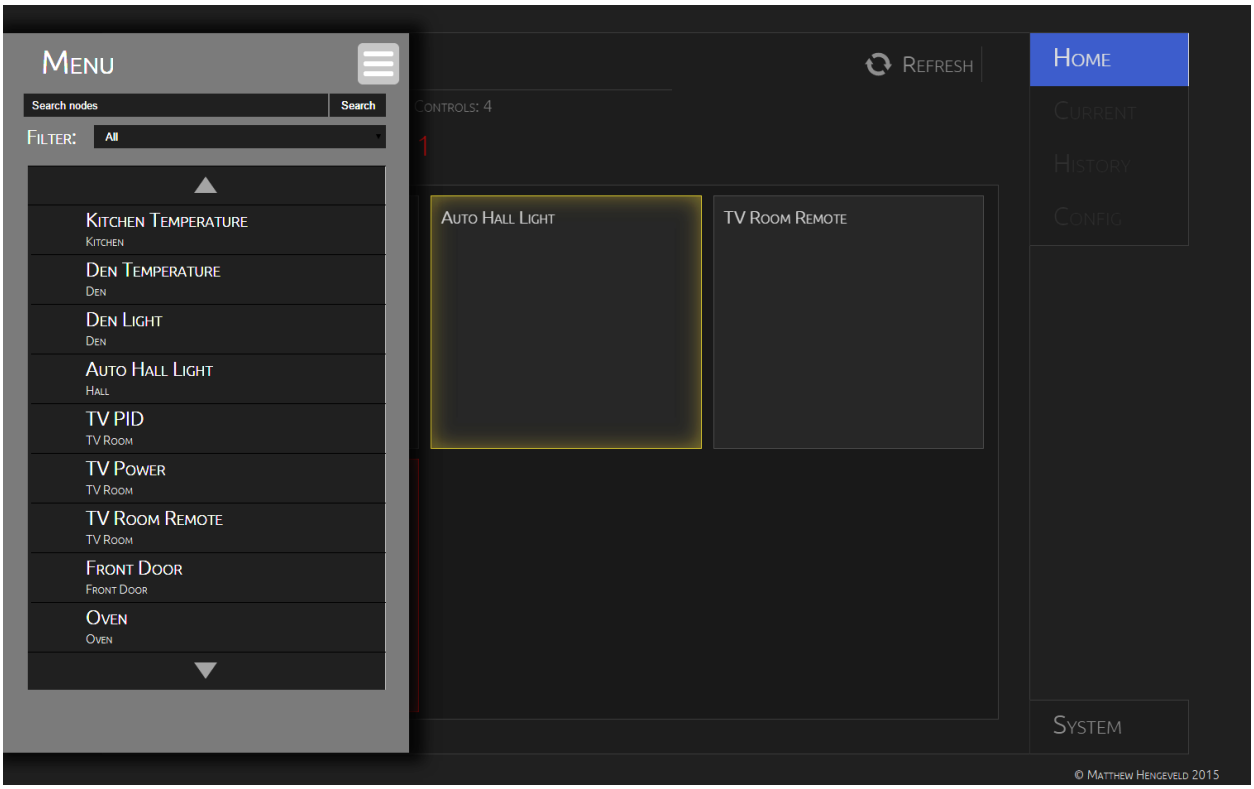

Figure 4. Web interface with slide menu and favorite grid.

Communication between the web interface client and the Flask server is handled using AJAX. The asynchronous nature of AJAX allows the web interface to be updated without the need to reload the page. For every module, there is an associated set of files (HTML, CSS and JavaScript) that creates a custom control interface. For example, an RGB light may have a colour slider, as well as swatches on its interface. The generic coding of the interface allows similar modules to use the same interface files,

speeding up design and implementation of new modules, as well as easy addition of new modules to a SensorNet network.

Database, Scheduling & nRF Interface Process

The second process in the SensorNet control software handles communication with the database, executes updates on modules at specified times and intervals, and contacts nodes via the nRF interface.

The database in SensorNet uses SQLite3 – a database system with very similar commands and rules to that of MYSQL, but which uses fewer resources, uses very portable database files, has tight integration with Python, and uses a dynamic typing system similar to that of Python.

Along with the data outlined in Figure 2, the database also stores system data and settings, as well as archived data for all sensors and controls. However, reading and writing a lot of data to the database creates a problem when combined with the Raspberry Pi. The Raspberry Pi uses SD media as its main storage, and excessive writes to SD media may cause irreversible damage quickly, which could occur with a sensor that is updated several times a minute. To prevent excessive writes to the SD media, SensorNet, at startup, creates a local copy of all often-used data in RAM. Not only does this prevent excessive writes, it also allows for quicker gathering of requested data, as well as custom sorting and searching algorithms to be implemented. The downside to this technique is the possibility of data loss in the event of a power failure or hard reboot of the Raspberry Pi. To mitigate this, all new and changed data is written to the SD card once per hour.

Scheduling functions of SensorNet are programmed to go down to five second intervals. Due to the Linux operating system and its time-sharing process scheduling, timing may not be accurate to within half a second or more.

GPIO pins and other hardware on the Raspberry Pi are controllable via many different libraries for languages including Python, C/C++, Perl, PHP and some others. C was chosen over Python for its speed. During testing, Python was observed to take up to three orders of magnitude longer to change a GPIO pin than the same C implementation. The library used is the bcm2835 library. The bcm2835 library directly programs fuses and registers in the bcm2835 CPU on the Raspberry Pi. This requires the library be run in sudo in Linux.

Using the bcm2835 library, a library was written for the nRF24L01+ transceiver. The library includes several functions for configuring settings, as well as receiving and transmitting. The nRF library for the Raspberry Pi also includes higher-level functions for combining commonly used groups of functions, reducing the number of Python calls to C functions. This allows the control software to use a single call to the library to update a node.

The significant difference in speed between libraries is somewhat mitigated by the wrapper code required to call C functions from Python code. Cython was used to wrap the C library. Cython is a C implementation of Python, which is written in Python code, and compiles to C code. Wrapping C code requires a Cython module that includes code to convert C data types to Python compatible data types (and vice versa), handle pointers, and implement faster C versions of Python functions. This wrapper code increases the time to call C functions, and results in a Python library taking up to two orders of magnitude longer to change a GPIO pin than wrapped C code.

5. nRF24L01+ Libraries

Libraries for the nRF are written for the Raspberry Pi, Arduino, and PIC microcontrollers, and the structure and code of the libraries can easily be changed to support other microcontrollers and platforms with C/C++ support. The library has several common functions, and they are outlined in the following table.

void init(uint8_t SPIDiv, uint8_t CEpin, uint8_t CSNpin, uint8_t IRQpin)		
		Initializes hardware and software variables and libraries needed for the nRF. Includes pin modes and directions, initial pin levels, SPI, global variables, interrupts, buffers and default nRF configuration registers.
SPIDiv		SPI frequency divider. Differs per development platform. Upper limit of the nRF24L01+ is 10MHz.
CEpin	*out*	Chip Enable pin. Selects RX/TX mode on nRF. See datasheet for details.
CSNpin	*out*	Chip Select Not pin. Enables SPI communication with nRF. Active low.
IRQpin	*in*	Interrupt pin.

void initSPI(uint8_t SPIDiv)	
	Initializes SPI hardware. *Note: PIC microcontrollers must use settings CKP = 0, CKE = 1, and SMP = 1. Arduino must use SPI_MODE0*
SPIDiv	SPI frequency divider. Differs per development platform. Upper limit of the nRF24L01+ is 10MHz.

void setTXMode(void)	
	Sets nRF to TX mode. This takes 140us to complete.

void setRXMode(void)	
	Sets nRF to RX mode. This takes 140us to complete.

uint8_t getMode(void)	
	Gets current mode of nRF. Returns (0) TX, (1) RX

void setPower(uint8_t pwrLvl)	
	Sets output power level of nRF antenna.
pwrLvl	Power level. (0) lowest, -18dBm to (3) highest, 0dBm

uint8_t getPower(void)

Gets current power level of nRF.
Returns (0) lowest, -18dBm to (3) highest, 0dBm

void setChannel(uint8_t ch)

Sets nRF frequency channel. Channel frequency = 2400MHz + channel. Ex. Channel 105 = 2400MHz + 105 = 2505MHz. Note: for an air data-rate of 2Mbps, channels must be separated by 1MHz, or one channel.

ch	Channel number - 0 to 125.

uint8_t getChannel(void)

Gets current channel number.
Returns 0 to 125

void setMaxRT(uint8_t numRT)

Sets max number of transmit retries. Only valid if auto acknowledgement feature is enabled.

numRT	Number of retries – 0 to 15.

uint8_t getMaxRT(void)

Gets current max number of retires.
Returns 0 to 15.

void setMaxRTdelay(uint8_t numRTdelay)

Sets delay between retries. Delay = (numRTdelay + 1) x 250us. Ex. numRTdelay = 2, therefore delay = (2 + 1) x 250us = 750us.

numRTdelay	Delay – 0 to 15.

uint8_t getMaxRTdelay(void)

Gets current retry delay.
Returns 0 to 15

void setTXAddr(uint8_t addr[], uint8_t len)

Sets TX address. Address is 4 bytes for SensorNet.

addr[]	Address array pointer.
len	Length of address array.

void setRXAddr(uint8_t pipe, uint8_t addr[], uint8_t len)	
	Sets RX address for pipe. Address is 4 bytes for SensorNet. *Important: only two pipes are used – pipe 0 set to RX address, and pipe 1 set to TX address. See section on addresses in this document for SensorNet-specific implementation details. For more information, see datasheet.*
pipe	Pipe number – 0 to 5.
addr[]	Address array pointer.
len	Length of address array.

uint8_t *getTXAddr(void)	
	Gets current TX address. Returns pointer to a byte array.

uint8_t *getRXAddr(uint8_t pipe)	
	Gets current TX address of pipe. Returns pointer to a byte array.
pipe	Pipe – 0 to 5.

void clearInt(uint8_t interrupt)	
	Clears interrupt(s). Interrupts can be combined to clear multiple at one time using the same function call.
interrupt	Interrupt(s) to clear. (0x10) MAX_RT, (0x20) TX_DS, and (0x40) RX_DR. 0x70 clears all interrupts.

uint8_t updateStatus(void)	
	Gets current status register of nRF, including interrupt flags. Returns one byte.

void setReg(uint8_t reg, uint8_t data)	
	Sets register in nRF. Used by set* functions that take one byte.
reg	Register to set.
data	Byte to set register to.

uint8_t getReg(uint8_t reg)	
	Gets current register value from nRF. Returns byte value.
reg	Register to get.

void transmit(uint8_t len)

Transfers data in output buffer to nRF and sends transmit command to transmit data. Transmission is started with a 12us high pulse of the CE pin. Transmission time depends on length of data, SPI frequency, and air data-rate, and may take up to 1.1ms (given 32 byte payload and 250Kbps air data-rate). If auto acknowledge activated, time will increase to include response from receiver, and includes a 140us delay to switch to RX mode. As well, if max retries > 0, transmitter may attempt to transmit data up to max number of retries, along with the retry delay time. It is highly recommended to wait for the TX_DS interrupt before ending nRF any further commands.

len	Number of bytes to transmit. Transmits bytes starting at byte 0.

void respond(uint8_t len)

High-level function that sets nRF to TX mode, transmits data, and returns to RX mode. Used to quickly respond to a request with minimum number of functions calls. Data to transmit must be in output buffer.

len	Number of bytes to transmit. Transmits bytes starting at byte 0.

uint8_t getPayloadSize(void)

Gets size of received payload following an RX_DR interrupt.
Returns 0 to 32. Not valid if > 32.

void getPayload(uint8_t len)

Gets received data from nRF following an RX_DR interrupt. Data is put in input buffer.

len	Number of bytes to retrieve from nRF. Should be set to value returned from ***getPayloadSize()***.

void putBufOut(uint8_t data[], uint8_t len)

Puts data into output buffer. Buffer is 32 bytes in size.

data[]	Pointer to data to place in output buffer. Will place data starting in byte 0.
len	Length of data to place in buffer. Attempting to place > 32 bytes in buffer will result in undefined behavior.

uint8_t *getBufIn(uint8_t len)

Gets current register value from nRF. Buffer is 32 bytes in size.
Returns pointer to input buffer. *Do not modify!*

len	Length of data to get from buffer. Attempting to read more than the returned data length will result in garbage data.

Additionally, the nRF uses several commands and memory-mapped registers. The values are summarized here, and are the same as the contents of nRF24L01+.h – a file that every library includes.

Commands	Word	Description
R_REGISTER	0x00	Read; Bits <5:0> = register map address (LSB first)
W_REGISTER	0x20	Write; Bits <5:0> = register map address (LSB first)
R_RX_PAYLOAD	0x61	Read RX payload 1-32 bytes (LSB first)
W_TX_PAYLOAD	0xA0	Write TX payload 1-32 bytes (LSB first)
FLUSH_TX	0xE1	Flush TX FIFO
FLUSH_RX	0xE2	Flush RX FIFO
REUSE_TX_PL	0xE3	TX; Reuse last transmitted payload; active until FLUSH_TX or W_TX_PAYLOAD
R_RX_PL_WID	0x60	Read RX payload width for top R_RX_PAYLOAD in RX FIFO
W_ACK_PAYLOAD	0xA8	RX; Write payload + ACK packet; <2:0> = write payload (LSB first)
W_TX_PAYLOAD_NO	0xB0	TX; Disable AUTOACK on this specific packet
NRF_NOP	0xFF	No operation; used as dummy data

Registers	Word	Description
CONFIG	0x00	Configuration register
EN_AA	0x01	Enable AUTOACK function
EN_RXADDR	0x02	Enable RX addresses
SETUP_AW	0x03	Setup address widths
SETUP_RETR	0x04	Setup auto retransmission
RF_CH	0x05	RF channel
RF_SETUP	0x06	RF setup register
STATUS	0x07	Status register
OBSERVE_TX	0x08	Transmit observe register
RPD	0x09	RPD (Carrier Detect)
RX_ADDR_P0	0x0A	Receive address data for pipes 0-5
RX_ADDR_P1	0x0B	
RX_ADDR_P2	0x0C	
RX_ADDR_P3	0x0D	
RX_ADDR_P4	0x0E	
RX_ADDR_P5	0x0F	
TX_ADDR	0x10	Transmit address
RX_PW_P0	0x11	Receive data width for pipes 0-5
RX_PW_P1	0x12	
RX_PW_P2	0x13	
RX_PW_P3	0x14	
RX_PW_P4	0x15	
RX_PW_P5	0x16	
FIFO_STATUS	0x17	FIFO status register
DYNPD	0x1C	Enable dynamic payload length
FEATURE	0x1D	Feature register

Interrupts	Word	Description
RX_DR	0x40	Data received interrupt
TX_DS	0x20	Data sent interrupt
MAX_RT	0x10	Max retransmit interrupt

For more information on implementation of nRF24L01+ libraries, please see the "Problems" section of this document.

6. Hardware

Hardware for SensorNet was designed to be three things: inexpensive, scalable and flexible. These considerations were meant to make SensorNet open-source, yet easy to build for the inexperienced do-it-yourself ("DIY") person. More experienced people have the option to build and expand the SensorNet system to fit their needs and design goals, while still using the underlying software.

Raspberry Pi Root

The Raspberry Pi Model B was chosen for the root for a number of reasons, mainly because it is inexpensive, and has great community support. Some downsides to the Raspberry Pi is that it has a single core CPU, and requires a heatsink and/or fan for continuous use, as the CPU can reach greater 50°C. Alternatives include the Raspberry Pi 2, the Banana Pi, and the BeagleBone Black, all of which have two or more cores, and are efficient enough to not need additional cooling. The additional CPU cores would help with simultaneous webserver and control software loads. Upgrading to one of these alternatives would only require a changing of the nRF library code, specifically SPI and GPIO.[5]

To help with cooling of the Raspberry Pi, heatsinks were added to the main CPU, the LAN/USB controller, and the main 3.3V voltage regulator. As well, a 50mm fan was placed above the Raspberry Pi, and using a custom-made circuit, was wired to be controlled by the Raspberry Pi's PWM pin. Control software was written to vary the fan speed based on the CPU temperature of the Raspberry Pi.[6]

Several online sources and literature makes reference to problems with nRF boards and its power regulation during transmitting and receiving. Transmitting and receiving results in a transient power spike, which can put too high a demand on the power supply source, especially if it is a source with low current capacity. The Raspberry Pi board uses a low drop-out regulator to supply several on-board components, as well as the 3.3V pin on the GPIO. A 47uF capacitor provides a reservoir for the output of the regulator. From the regulator datasheet[7], the regulator, with a 10uF output capacitor, has a load regulation of just 10mV max when current demands change from 10mA to 800mA. This makes it very unlikely that the 14mA max current the nRF demands at any one time would significantly affect the power supply enough to cause a malfunction.

[5] Different versions and distribution of the Linux OS may also require code changes.
[6] Software can be found in the "Extra" folder in the project folder.
[7] From the Raspberry Pi schematic (https://www.raspberrypi.org/wp-content/uploads/2012/04/Raspberry-Pi-Schematics-R1.0.pdf), the regulator is a NCP1117 (http://www.onsemi.com/pub_link/Collateral/NCP1117-D.PDF)

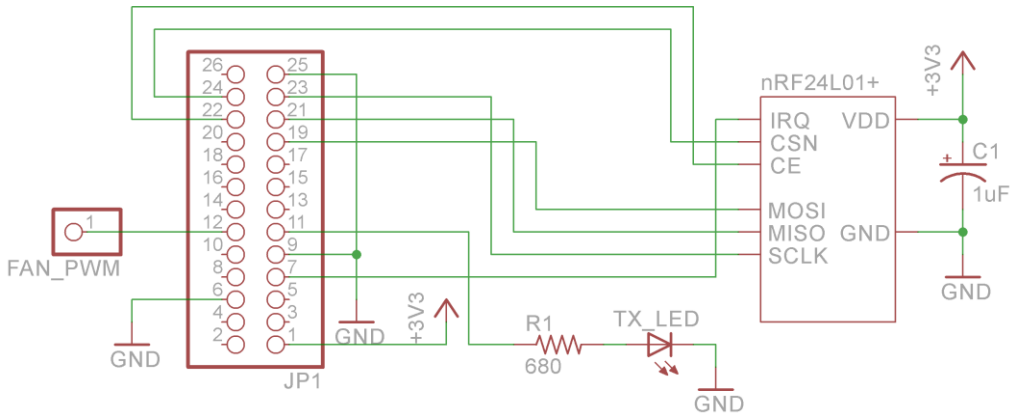

Figure 5. Raspberry Pi GPIO header schematic. Note: FAN_PWM goes to a separate MOSFET fan driver, and uses a separate 12V power supply. The GPIO pin cannot provide the necessary voltage or current to power a fan.

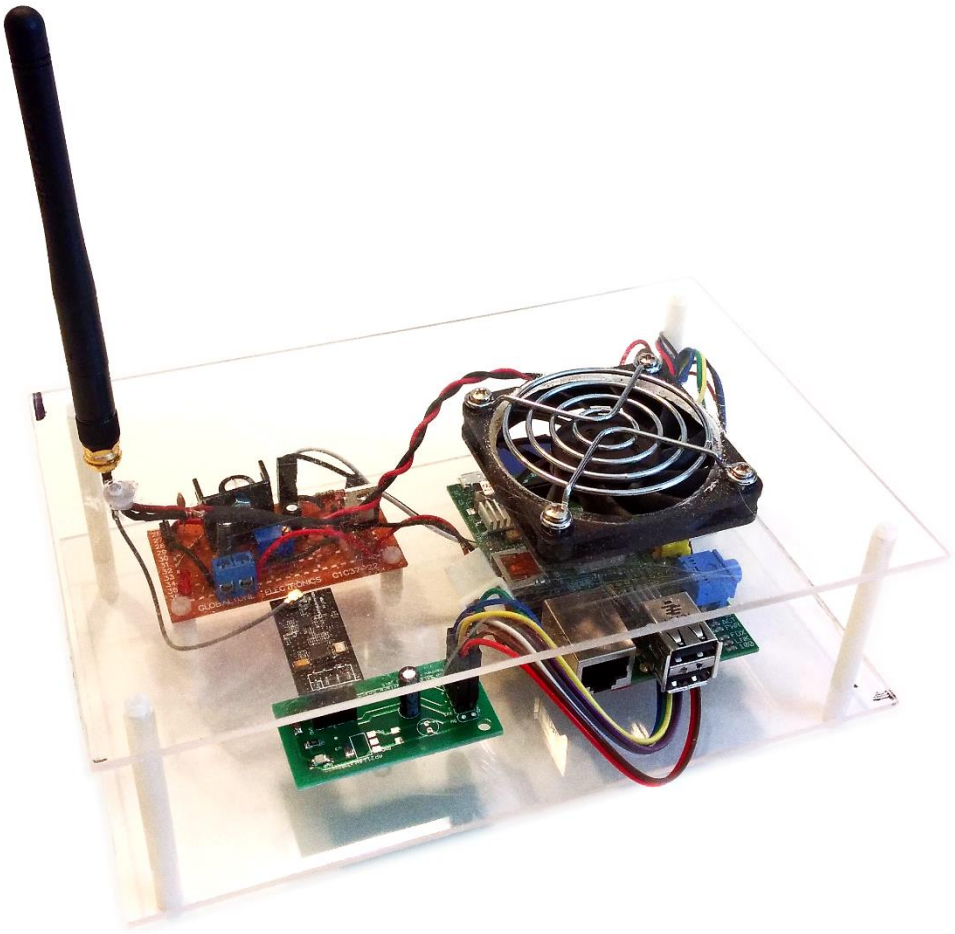

Figure 6. Raspberry Pi with nRF24L01+, cooling fan, antenna and power supply.

Microcontrollers

The heart of SensorNet's open-source and flexible design are the microcontrollers that power each node. The microcontroller used is transparent to the root, so that virtually any microcontroller, DSP, or platform with SPI capability can be seamlessly integrated[8].

Arduino microcontrollers are simple, easy to program, and in most cases, do not require code changes to support different boards. Designing a node involves very little interaction with the nRF library code – only setup code, request response code, and a function call inside the main loop. This allows designers to focus on interfacing with sensors and automation controls. A complete node can easily be designed in less than an hour. This is what makes the Arduino the main microprocessor used in nodes, and user-friendly to beginners in the embedded world.

Below is an example of temperature and humidity sensor. It uses a library for the DHT11 sensor to get the temperature and humidity, and uses two commands to relay the data back to the root. Using the Arduino Nano board, it is possible to run this from a battery for up to a week[9]. This example can easily be used as a template, and requires minimal changes in order to convert it to a 24-bit RGB LED lamp: Use one command to receive the 24-bit value, and use three of the Arduino's PWM channels to control three MOSFETs, one each for the three RGB colors.

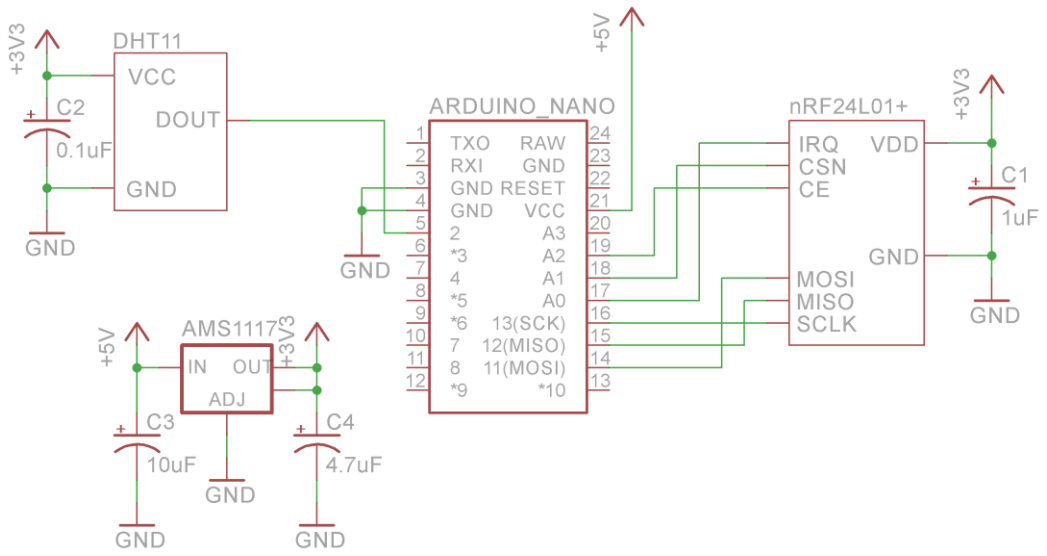

Figure 7. Schematic of Arduino demonstration node.

[8] As with the previous generation nRF24L01 and several Arduino libraries, the SPI protocol can be 'bit-banged' for control by microcontrollers that lack SPI hardware capability.

[9] Based on an average power consumption of 15mA and running off a 2200mAH USB battery. No power optimizations were used.

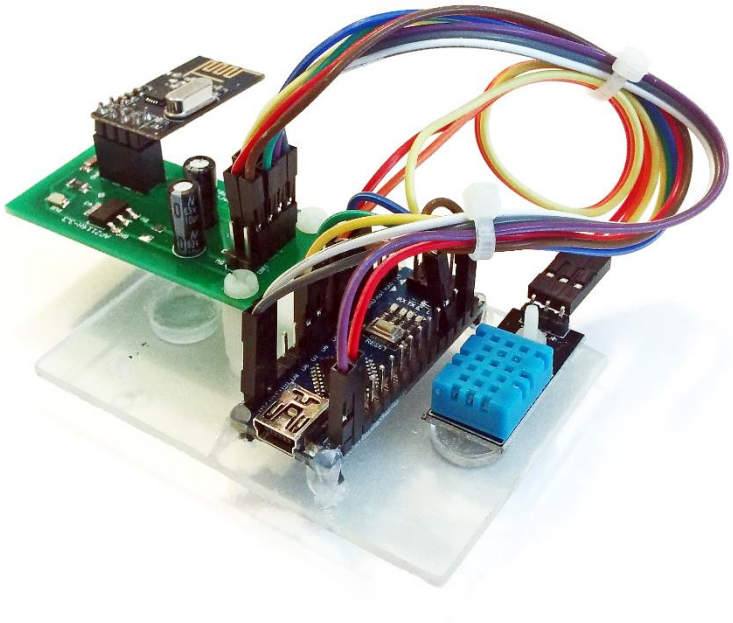

Figure 8. Demonstration node using an Arduino Nano, and a DHT11 temperature and humidity sensor.

PIC microcontrollers allow a designer to have a lot more control over the microcontroller compared to Arduino. As well, PIC microcontrollers with SPI are available in packages with as few as eight pins, and up to full 32-bit, 128-pin microcontrollers with DSP and other advanced features. With only minor code changes to the nRF library, this gives great flexibility to design small and powerful sensors and automation controls that integrate into a SensorNet network, as well as have independent functions.

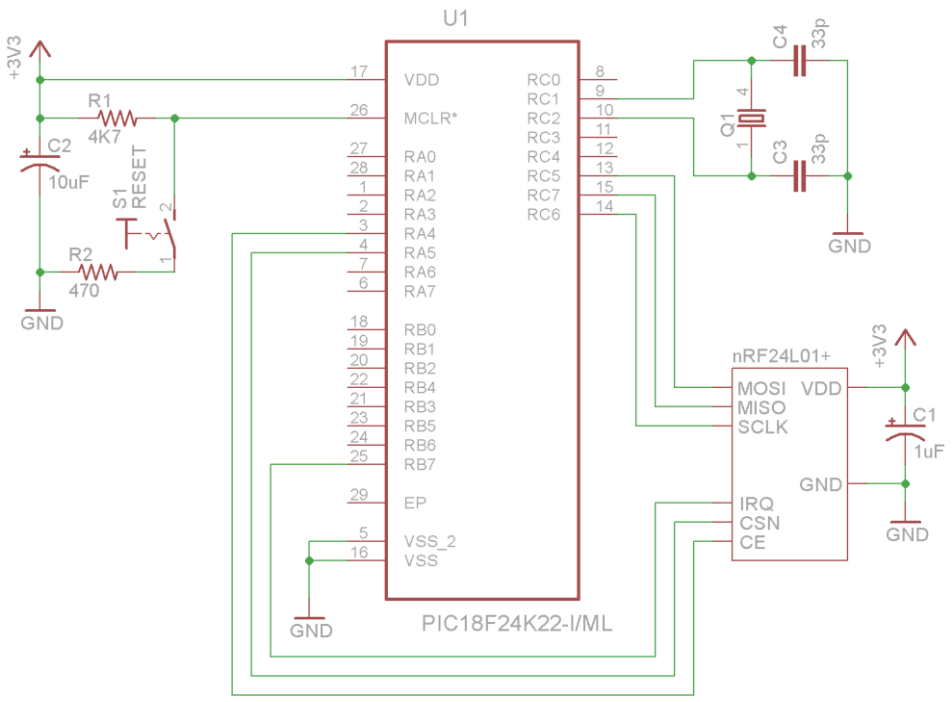

Figure 9. PIC18F24K22 microcontroller with external oscillator, reset switch, and nRF24L01+.

Returning to the power requirements for the nRF, we have seen that the Raspberry Pi's 3.3V regulator can handle the transient power that the nRF needs. However, small microcontrollers may not be able to handle the transient power spikes. The Arduino Nano board, used for both the temperature and humidity sensor, and the RGB light demo circuits, rely on the onboard FT232RL USB to serial interface chip[10] to supply the 3.3V supply pin. The datasheet gives a 50mA current limit on this pin. The Arduino Nano schematic[11] shows only a 0.1uF reservoir capacitor. As this pin powers a few other components, including LEDs, we can assume that the 14mA required by the nRF could easily push the 50mA limit, and cause the nRF to make errors.

Sensors and Automation

The diverse selection of sensors and automation control is where SensorNet differentiates itself from commercial products. Using microcontrollers, virtually any sensor and automation control can be realized. From led lights and thermostats, to complex power consumption monitoring and smart A/V entertainment systems, there are no limits.

Microcontrollers currently support a wide range of protocols that can interface with a number of different products. SPI, I^2C, I^2S, CAN, Ethernet, USB, UART, P/S2, WiFi, NFC, SMB, DMX, and IRDA are just some.

[10] https://www.soselectronic.com/a_info/resource/c/pdf/ft232r.pdf
[11] http://arduino.cc/en/uploads/Main/ArduinoNano30Schematic.pdf

All this gives way to some very interesting possibilities, including:

- Smart, programmable HVAC system
- A/V system remote
- Remote location home monitoring
- Household system notifications, for laundry, dishwashing, oven and pool temperature
- Automatic, GPS-based room lighting
- Time and weather-based room ambience settings, including lights, temperature and blinds
- Garage car occupancy monitor

Other Hardware

To make sure the nRF always gets enough power, an auxiliary power supply board was designed. This board includes filtering and reservoir capacitors for the 3.3V supply line, a 3.3V regulator with appropriate capacitors for use with a 5V supply, and a small green power-on LED. While errors were observed in initial testing of the nRF (unrelated to code), the errors could not directly be related to the nRF and insufficient supply current or voltage. Several different solutions resulted in elimination of the errors, and implementing the auxiliary power supply board for all nRFs made sure it was not a problem in the future.

The auxiliary power supply board was designed in CadSoft Eagle[12], and the PCBs were manufactured by Seeed Studio[13]. Each PCB included two boards. Parts were hand soldered[14].

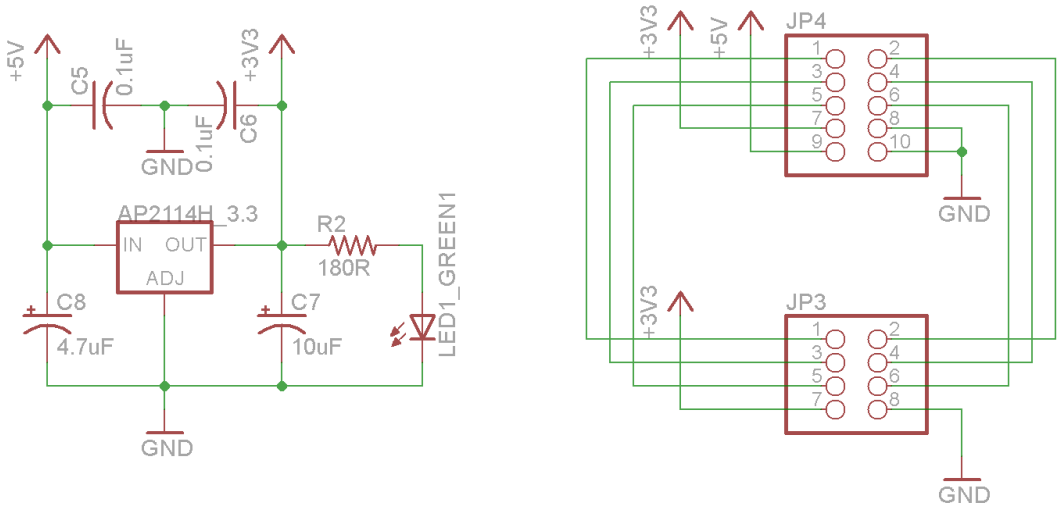

Figure 10. nRF auxiliary power supply board schematic.

[12] http://www.cadsoftusa.com/
[13] https://www.seeedstudio.com/service/index.php?r=pcb
[14] Board schematics and PCB files are available under Extra/EAGLE/projects in the project folder

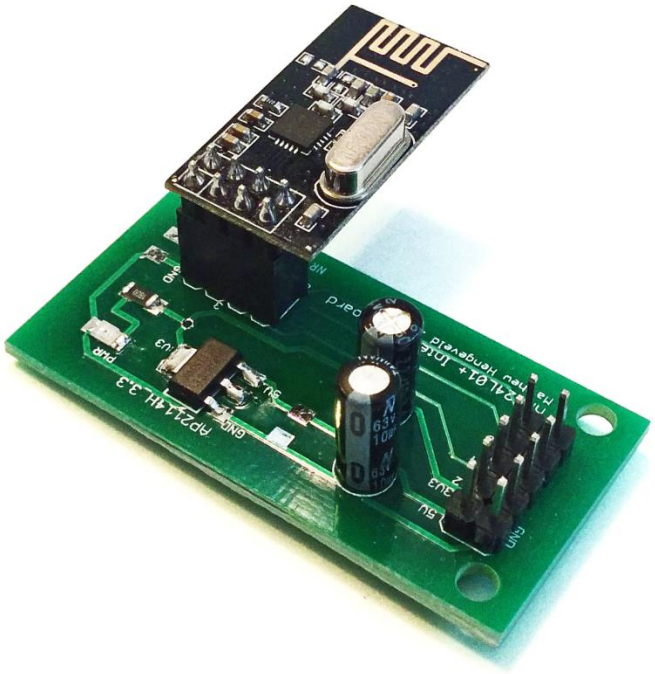

Figure 11. nRF auxiliary power supply board.

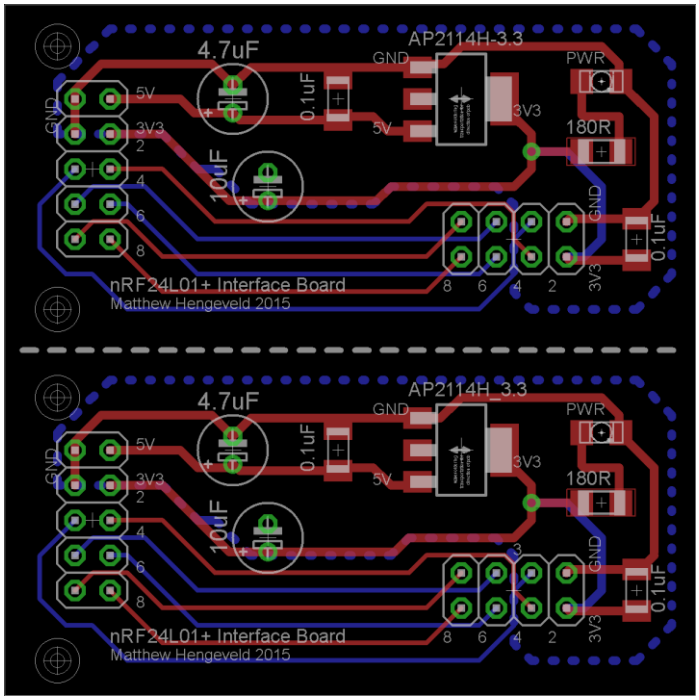

Figure 12. nRF auxiliary power supply board PCB.

7. Challenges

Several problems were encountered during the project.

The SPI interface for the nRF24L01+ has specific requirements. First, all consecutive data must be sent while the CSN pin is held low. If the CSN pin line is pushed high during the data sequence, then the nRF24L01+ will assume it is the end of the data. For example, to transmit data, the CSN pin must first be held low. A command is then sent to let the nRF2L01+ know there is data coming. The data is then sent. After the last byte is sent, then the CSN can be changed to high once more.

One of the features of the nRF24L01+ is the auto acknowledgement. When a packet is sent and receiver successfully, the receiver quickly changes to TX mode, while the transmitter quickly changes to RX mode. If the transmitter does not receive an acknowledgement, then the packet will be resent. In implementing this feature, it was not clear that the auto acknowledge feature used a different data pipe to send the acknowledgement packet. While the acknowledgement packet was being sent, it was being sent to the wrong address.

In testing the libraries, it was found that packets having payload sizes from 30 to the full 32 byte payload were not being received. This problem has not been resolved, therefore, SensorNet is limited to payloads of 28 bytes or less.

PIC microcontrollers can be complex. During the initial stages of programming the PIC to talk to the nRF24L01+, it was unknown that the PIC has options for changing the SPI sample position (start or middle of clock signal) and for changing the SPI clock polarity (low or high). These were, by default, not set up right for communication with the nRF24L01+. The following settings were needed to resolve the problem[15]:

Register.bit	Setting
Clock Polarity:	
`SSP1CON1bits.CKP`	`0`
Clock Edge Detect:	
`SSP1STATbits.CKE`	`1`
Sample Bit:	
`SSP1STATbits.SMP`	`1`

The root node for SensorNet is the Raspberry Pi. It was first released in 2012, and has since become very popular. Despite its popularity and community support, there are still challenges when it comes to the hardware aspect of programming. One difficulty with the Raspberry Pi was finding a suitable programming interface for the GPIO and SPI hardware. Several different libraries were tested. The majority of the problems arose from the libraries needing root (account with all rights and privledges) access to execute. This is a security risk, and could compromise a home network. Another

[15] For the PIC18F2xK2x series.

problem was finding a library that supported both GPIO and SPI functions. Lastly, because of the Linux operating system and its process scheduler, time-critical code was prevented from executing when it was intended to. The most severely affected function were interrupts, as it could take tens of milliseconds to register and execute the interrupt service routine. A solution was found that resolved all of these problems: a C library called bcm2835. Bcm2835 supports GPIO and SPI function. Also, because it is written and compiled in C, it runs significantly faster than other tested Python libraries. In an extreme case, bcm2835 executed a GPIO pin change in less than 40µs, whereas the Python library took more than 200ms. This large difference is because C is a compiled language, and Python is an interpreted language. Interrupt functions are not supported in bcm2835, however, because of the speed of C code, polling is used to detect pin changes. For the purposes of SensorNet, this means that an estimated 1500-2000 packets can be sent a second[16].

During the design and coding of the control software, it was determined that the Flask webserver was not able to be created using the multiprocessing library in Python – the result was two Flask processes. While some research as to why this occurred was done, it was deemed low priority, and was eventually stopped in favour of starting the two control processes separately.

The Cython wrapper is a unique and powerful tool for use with Python and C. However, a lack of documentation for new users, and tutorials that were too specific to an example makes beginning Cython a challenge. This was discovered during the coding of the C wrapper. Conversion between C and Python data types – especially arrays and pointers – as well as advanced features such as interfacing with C++ classes were some of the challenges during this project. C++ wrapping was abandoned because of this.

[16] Using 1 Mbps data rate, assuming a packet size of 29 - 32 bytes, and a time of 500-800µs per packet.

8. Conclusion

Status

The status of the project is working, but not complete. Timing functions in SensorNet do not work, but are at a stage that would require little work to complete. As well, the web interface has several uncompleted features, including graphed archive data, system options and information, and the refresh button.

Future

In the future, several features are planned, including:

- Two-way requests. This would allow a node to contact the root without the root first requesting data. This is complicated, as there is no reliable way to know if the root is currently in the middle of another request. Sending a request while a root or another node is on-would interfere with the communication in progress, and neither communication would get through. Implementing this feature would allow “control nodes” to be able to request information of other nodes, and to control automation control nodes. For example, a node that controls a thermostat could request temperature data from another node automatically.
- Signal relays. Signal relays were a planned feature at the start of the project, however, they were not implemented due to time constraints. Signal relays would allow nodes to be placed much farther away from the root, without changing antenna type or increasing output power. As well, nodes placed in locations with obstruction between it and the node could use a relay to avoid the obstruction.
- Android application. An iPhone application was developed for SensorNet by Quinton Black and Brian Sage. The web interface, despite working well in smartphone web browsers, lacks capability of more advanced features, such as notifications, calendar integration, and automatic SensorNet network identification.
- Security. The project, currently, has no security measures at all. This is not such a concern if it is not reachable from outside the LAN, and the wireless networks have passwords, so that only people with computers allowed on the network would have access to the web interface. For use outside the LAN, and from anywhere in the world with internet, security systems must be put in place so that only authorized people can have access.
- Webserver. The Flask webserver, as mentioned, is adequate for testing and small amounts of traffic. However, with increased traffic, multiple users and more features, the Flask webserver may not hold up. Using a dedicated, lightweight webserver such as Lighttpd or Nginx would ensure future reliability.

- Additional wireless technologies. Intergrating Bluetooth into the root would allow Bluetooth enabled devices to become nodes without any additional hardware. Integrating the ESP8266 serial WiFi transceiver could allow high-bandwidth transfers between nodes and the root.

Conclusion

Overall, the project was rewarding, and a lot was learned about wireless transceivers, designing multi-architecture libraries, Python, C/C++, and especially the Raspberry Pi and its Linux operating system. I intend to carry on in my own time with this project and to continue to share it in the open-source community for others to use, experiment with, learn from, and to be inspired by.