**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 | | 2 |
| 1 | Background | | 3 |
| 2 | Network | | 4 |
| 3 | Nodes | | 6 |
| 4 | Control Software | | 7 |
|  | | Flask Webserver and Web Interface | 8 |
|  | | Database, Scheduling & nRF Interface Process | 9 |
|  | |  |  |
| 5 | nRF24L01+ Libraries | | 10 |
| 6 | Hardware | | 15 |
|  | | Raspberry Pi |  |
|  | | Microcontrollers |  |
|  | | Sensors and Automation |  |
|  | |  |  |
| 7 | Problems | |  |
| 8 | Project Conclusion | |  |

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.

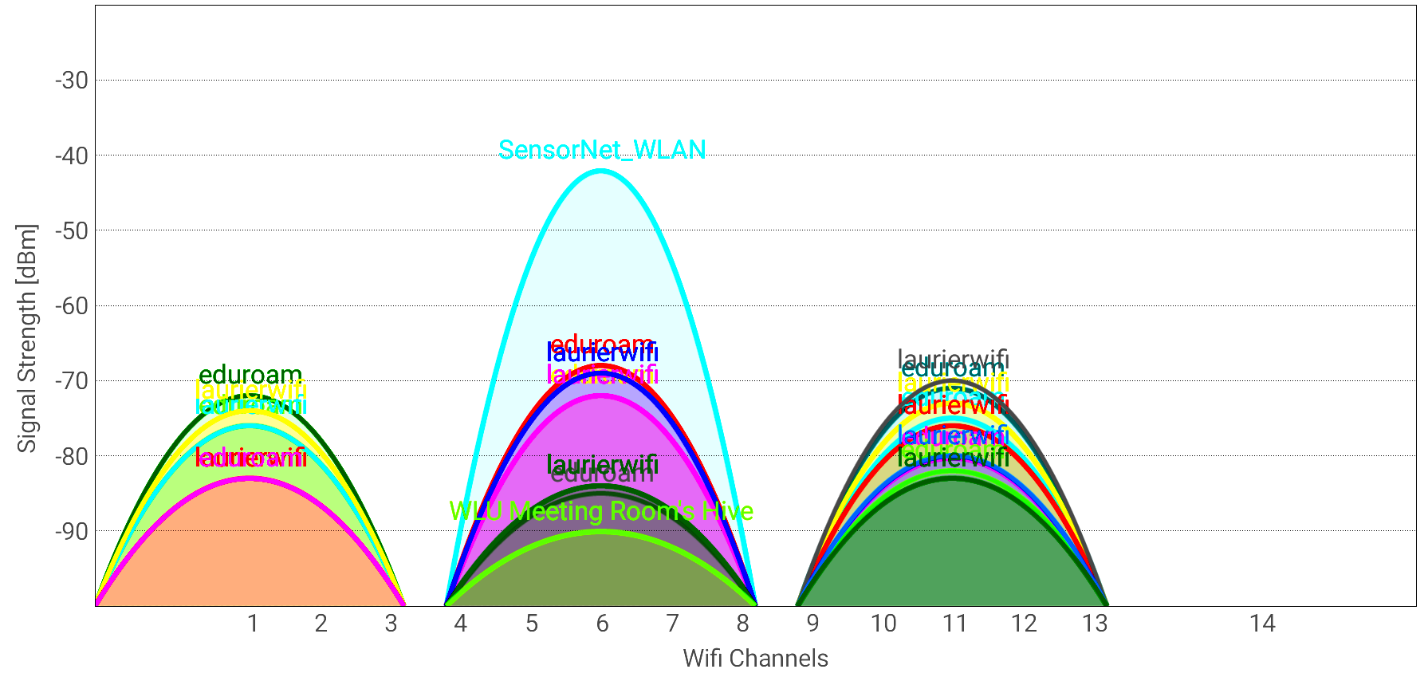
WiFi was created for transfer of large amounts of data, and comes with advanced, complicated protocols and the significant overhead that accompanies them. Sensors and automation controls have inherently small data requirements. Sensors, such as temperature, humidity, light and hall sensors, typically 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]](#footnote-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.

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]](#footnote-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 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

or . 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.

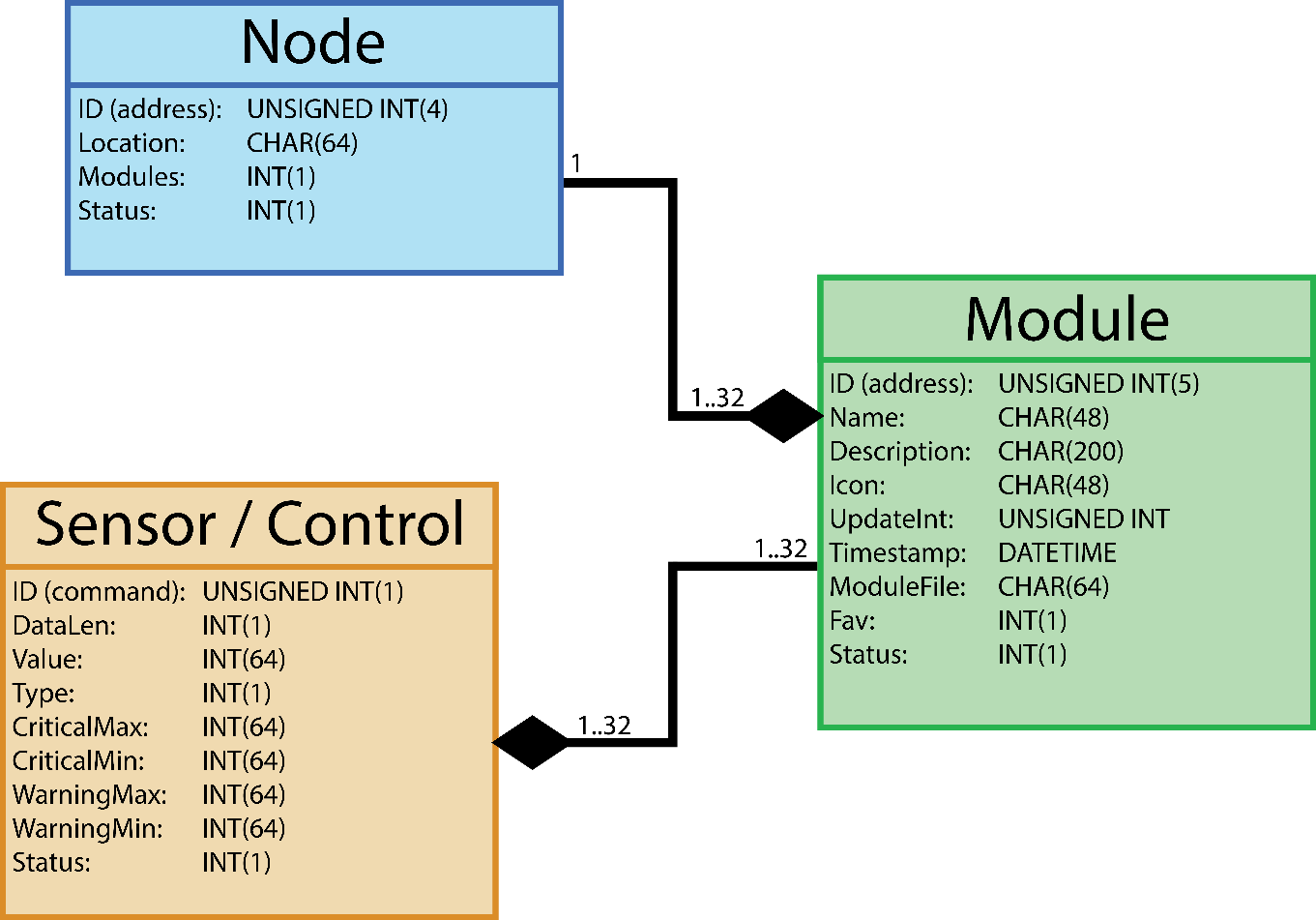
Each packet sent using the protocol is required to be acknowledged by the receiver. The auto acknowledgement feature of the nRF is used to implement this requirement, and helps to reduce lost packets, determine if a node is outside the range of the root, and ensure critical packets are received.

*For more information on the nRF24L01+ and its features, please see the accompanying document “Nordic nRF24L01+ Implementation Tips and Techniques”.*

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]](#footnote-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 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.

The control software for SensorNet can be visualized as layers:

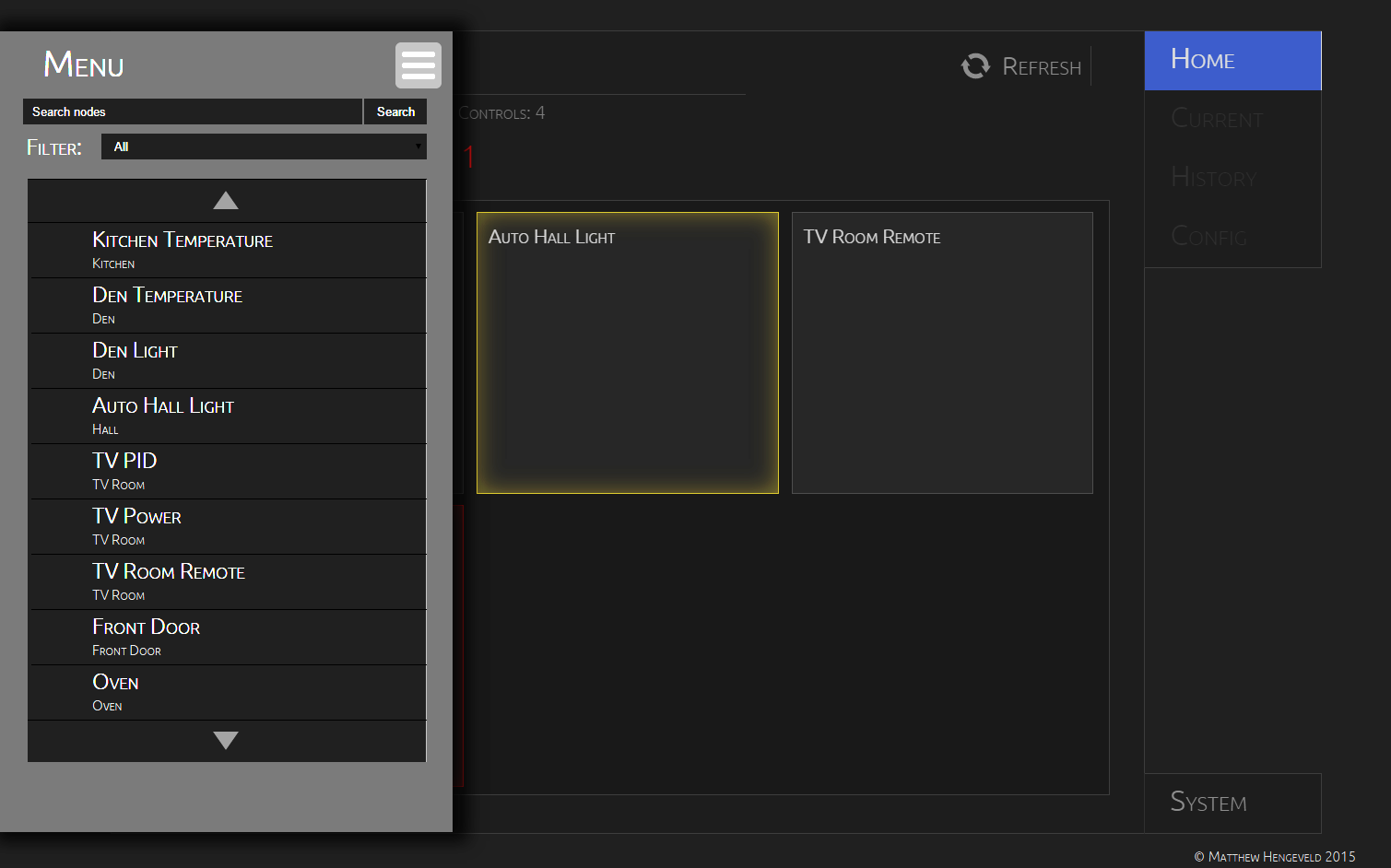
|  |  |  |
| --- | --- | --- |
| Flask webserver | | |
|  | | |
| ZeroMQ | | |
|  | | |
| Database/timing/nRF interface process | | |
|  |  |  |
| SQLite database |  | Cython wrapper |
|  |  |  |
|  |  | bcm2835 library |
|  |  |  |
|  |  | SPI |
| Process / library |  |  |
| Interface / Communication |  | Nordic nRF24L01+ |

*Figure 3. Design of control software.*

**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.



*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. | |
| *pwrLvl* |  | 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** |  |  |
| 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** |  |  |
| 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** |  |  |
| 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 accompanying document “Nordic nRF24L01+ Implementation Tips and Techniques”, as well as the “Problems” section of this document.*

6. Hardware

Hardware

1. Given an Ethernet packet size of 1500 octets. http://en.wikipedia.org/wiki/IEEE\_802.11 [↑](#footnote-ref-1)
2. http://en.wikipedia.org/wiki/List\_of\_WLAN\_channels [↑](#footnote-ref-2)
3. The Raspberry Pi has a single-core processor, however, the Linux OS uses a time-shared process scheduler. [↑](#footnote-ref-3)