UNIVERSITY OF CALIFORNIA

Santa Barbara

Design and Analysis of Arduino, Raspberry Pi, and Xbee Based Wireless Sensor Networks

A Thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in Electrical and Computer Engineering

by

Sean Alexander Cassero

Committee in charge:

Professor Joao Hespanha, Chair

Professor Katie Byl

Professor Jason Marden

September 2016

The thesis of Sean Alexander Cassero is approved.

Katie Byl

Jason Marden

Joao Hespanha, Committee Chair

September 2016

ACKNOWLEDGEMENTS

Thanks to

Professor Joao Hespanha for agreeing to act as advisor for the duration of the project

Professors Katie Byl and Jason Marden for consenting to be a part of the thesis committee

Mohamed “Rufus” Hafez and Stefan Waterstaat for their contributions to the *Rails* development and subsequent analysis section

and Sara Khatib for her support and encouragement throughout

ABSTRACT

Design and Analysis of Arduino, Raspberry Pi, and Xbee Based Wireless Sensor Networks

by

Sean Alexander Cassero

With a constantly changing technological landscape, the Engineering world is continually tasked with justifying the optimality of accepted standards and practices. The recent development of inexpensive simple microprocessors and linux computers has the potential to replace various methodologies for low energy, low-rate information transfer and control such as environmental monitoring, smart houses, smart lighting and others.

In the area of wireless sensor networks, a design standard is developing incorporating Xbee series 2 as a wireless bridge between Arduino or Raspberry Pi sensor and data aggregate nodes. In this thesis I construct an Xbee series 2 ZigBee wireless star topology network with an Arduino as a ZigBee End Device and Raspberry Pi as the ZigBee network coordinator.

The End Device uses an Arduino Uno v3 for local signal processing on a Parallax PMB-648 GPS and DS18B20 temperature sensor for periodic signal transmission via Xbee series 2. Xbee uses API mode 2 with escaping for package formation and transmission and is connected to the Arduino via the hardware serial port.

The Coordinator node consists of an Xbee Series 2 with Coordinator firmware communicating via the Raspberry Pi GPIO serial input ports. The Raspberry Pi uses specialized *Python* libraries to parse incoming API statements from active end devices.

The Raspberry Pi doubles as an internet gateway to an *SQLite* database run on a *Ruby on Rails* web application framework. The Raspberry Pi uses the *Python* `requests` library to transmit received End Device sensor measurements to the cloud server as URL parameters. The *Ruby on Rails* framework uses a Model View Controller architecture to pass data as URL parameters to an *SQLite* database, as well as display End Device sensor data on an interactive user interface upon a browser request. The user interface uses *Gmaps4Rails* to render an interactive map consisting of the GPS markers of reporting End Devices and their corresponding temperature measurements. The cloud server functions as a shared database linking multiple complete wireless sensor networks together under a single web app.

By testing End Device node lifetimes with various data transmission frequencies, an experimental relationship between Arduino/Xbee sleep duration and End Device lifetime is found. Using direct current measurements and information on the End Device hardware, a theoretical relationship between battery charge and End Device charge consumption during runtime is used to generate experimental equations relating End Device average current consumption during different phases in End Device lifetime. Multiple regression analysis is performed to derive an experimental value for the average current consumption of the End Device during all phases of operation, resulting in an experimental relationship between End Device average current and data transmission frequency of

$I_{avg} = \frac{(2.3\,mA * t_{sleep} + 131.3\,mC)}{(t_{sleep} + 2.6\text{s})} + 34.0\text{mA}$, where t_{sleep} is the End Device sleep cycle duration in seconds. The above relationship was able to predict the average current for all End Device trials to within 5% error.

I. Introduction

1. Related Work/Motivation

As wireless technology matured, Wireless Sensor Networks (WSN) began to emerge as an advantageous alternative to their wired counterparts due in part to easy deployment and scalability [1]-[2]. The 802.15.4 IEEE communication standard was developed for use specifically with low-rate wireless personal area networks (LR-WPANs) with a focus on wireless sensor networks [3]. In the early 2000s, the ZigBee alliance worked to construct the ZigBee protocols, communication protocols functioning on the 802.15.4 MAC and Physical layers. The main advantage of the ZigBee protocols over its competitor Bluetooth was ZigBees' highly efficient sleep mode; ZigBee devices use a basic master-slave configuration suited for low frequency data transmission star topologies, and can wake from sleep and transmit a packet in around 15 miliseconds. As a result, ZigBee devices can last for long periods on a single power supply [7].

In recent years, Digi incorporated the 802.15.4 standard and ZigBee protocols into a proprietary RF module known as the Xbee. Xbee devices have modular firmware capable of constructing various network topologies and have been utilized as end devices in wireless sensor network and monitoring applications [4]-[6].

However, Xbee does not contain large processors for signal processing or local data analysis at the End Device. The limited processing capabilities of an Xbee device can be addressed with the implementation of additional hardware for processing support. Current WSN designs utilize an Arduino, a low-cost, reliable microcontroller capable of functioning as a building block for data acquisition or control systems [8], to augment a sensor nodes processing capabilities [9]-[10].

In addition to the Arduino and Xbee, prototype WSN routinely incorporate a Raspberry Pi, a small inexpensive linux computer. The Raspberry Pi usually serves as a hardware platform for the ZigBee network Coordinator, and is used to direct network communication and control in wireless systems [12]-[13]. Additionally, the Raspberry Pi can be used to handle WSN data storage by functioning as a database server [11].

Raspberry Pi, Arduino, and Xbee based WSN posit two main questions. First, since ZigBee protocols were developed specifically for facilitating long node lifetimes, how does introducing additional processing hardware in the form of an Arduino impact overall node lifetime? And second, if one reason for the advance of WSN is its scalability, how do developers address the relatively limited storage capabilities of the IoT devices (Arduino, Raspberry Pi) and their potential inability to successfully scale with increasing WSN traffic?

2. Project Overview

This project designs and builds an Xbee, Arduino, and Raspberry Pi based wireless sensor network as a client to a cloud database server.

The project uses Xbee series 2 with ZigBee protocols, which are based on the IEEE 802.15.4 standard, as a wireless personal area network (WPAN) communication bridge. The Xbee Series 2 modules are programmed to communicate in API mode with escaping to form a star topography WPAN with one ZigBee Coordinator node and multiple ZigBee End Devices.

Each ZigBee End Device consists of an Xbee Series 2 radio frequency (RF) communication module loaded with End Device firmware via XCTU, mounted on an Arduino Uno v3 via a proprietary Xbee shield. The Arduino Uno serves as a signal processor for two local sensors: a DS18B20 temperature sensor and a Parallax PMB-648 SiRF GPS Module.

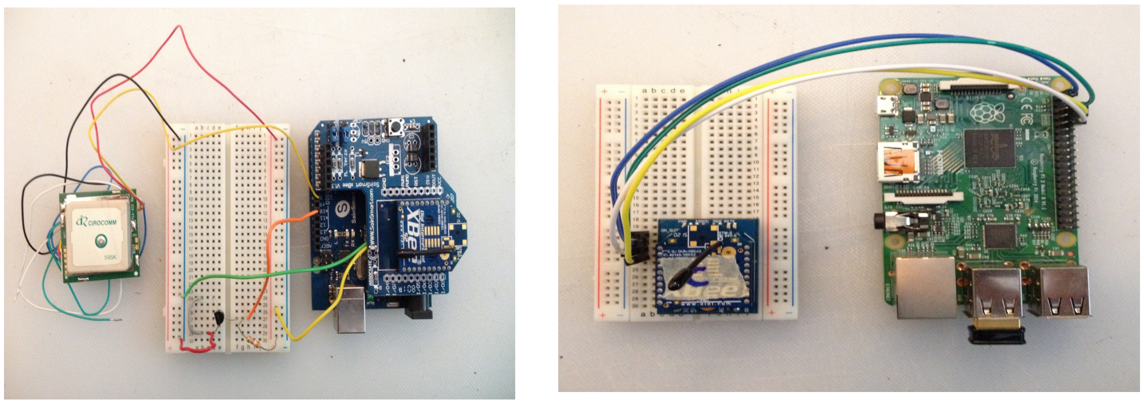


Figure 1. Initial prototype hardware for the ZigBee End Device (left)and Coordinator (right)

Both sensors require signal processing to convert their data into human readable format. The Arduino uses the *One Wire* and *Dallas Temperature* libraries to read temperature values from the DS18B20 sensor, and the *softSerial* and *TinyGPS* libraries to parse GPS data from the PMB-648 GPS module. The Arduino runs a single loop that manages reading temperature and GPS sensor data, and communicating data via Xbee to the ZigBee network Coordinator.

Both the Xbee and Arduino have sleep functions that minimize power consumption by periodically stopping unnecessary internal processes when those processes are not needed. The sleep functions were implemented inside the Arduino main code loop to halt superfluous processes while the node was neither gathering nor transmitting data.

In oder to address the impact of the Arduino on End Device lifetimes, End Device average power consumption was compared for a range of transmission frequencies to generate a graphical relationship between transmission frequency and power consumption of an Arduino-Xbee End Device.

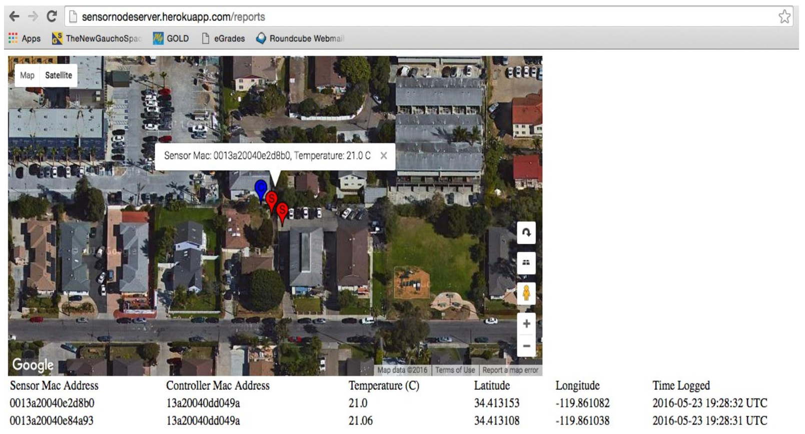
The ZigBee Coordinator node consists of a Raspberry Pi series 2 B running OS Raspbian Jessie and a single Xbee Series 2 loaded with Coordinator firmware via XCTU. The Xbee Coordinator transmission and reception lines are input to the Raspberry Pi via its GPIO pins as a serial communication device. Raspberry Pi uses the *Python serial* and *Xbee* libraries to parse incoming API statements from End Devices.

In order to address the limited local storage on the Raspberry Pi ZigBee Coordinator, the device is transformed into an *SQLite* cloud database client. The Raspberry Pi uses the *Python requests* library to transmit data packets as URL parameters to a cloud server. The cloud database server handles all WSN data storage, alleviating the responsibility from the Raspberry Pi.

Web Application Development uses Heroku as a Platform as a Service (PaaS). Heroku runs a *Linux* Operating System, a *Puma* Web Server, and an *SQLite* database as a framework for development. The project uses the Heroku foundation to run a Ruby on Rails Web Application.

The Web Application is in charge of managing wireless sensor network data storage in the *SQLite* database and rendering a useful human readable User Interface (UI) for data presentation with a browser request. The Web Application uses *Rails* Model, View, and Controller (MCV) architecture to pass incoming URL parameters to the the *SQLite* database via Object Relational Mapping (ORM).

User Interface uses the *Gmaps4Rails*, a *Ruby* gem to superimpose Sensor and Coordinator GPS data as markers on an interactive map using the google maps API. The markers display relevant sensor data when clicked by the user, such as MAC address for sensor and Coordinator node, and temperature in degrees celsius for sensor node. A full list of the latest received data for each unique Sensor node is displayed in table format underneath the map for easy viewing.



Sensor Mac: 0013a20040e2d8b0, Temperature: 21.0 C

Sensor Mac Address	Controller Mac Address	Temperature (C)	Latitude	Longitude	Time Logged
0013a20040e2d8b0	13a20040dd049a	21.0	34.413153	-119.861082	2016-05-23 19:28:32 UTC
0013a20040e84a93	13a20040dd049a	21.06	34.413108	-119.861038	2016-05-23 19:28:31 UTC

Figure 2. Example Web Application UI

Additionally, the cloud database server is designed to be a shared database for multiple wireless sensor networks. A collaborative wireless sensor network cloud database may be useful in monitoring large scale geographically separate areas of interest such as a nationwide average temperature census or large scale environmental monitoring. Examples are given showing the cloud server functioning as a shared database.

I. Wireless Communication

A. 802.15.4 Communication Protocol

There are four main wireless Personal Area Network (PAN) standards for data transmission: Wifi (based on IEEE 802.11/a/b/g), ZigBee (based on IEEE 802.15.4), Bluetooth (based on IEEE 802.15.1), and UWB (based on IEEE 802.14.3). While each have their intended use cases, the 802.15.4 IEEE protocol was developed specifically for low-rate, low power signal transmission [3] and is ideal for industrial level sensor network applications [2].

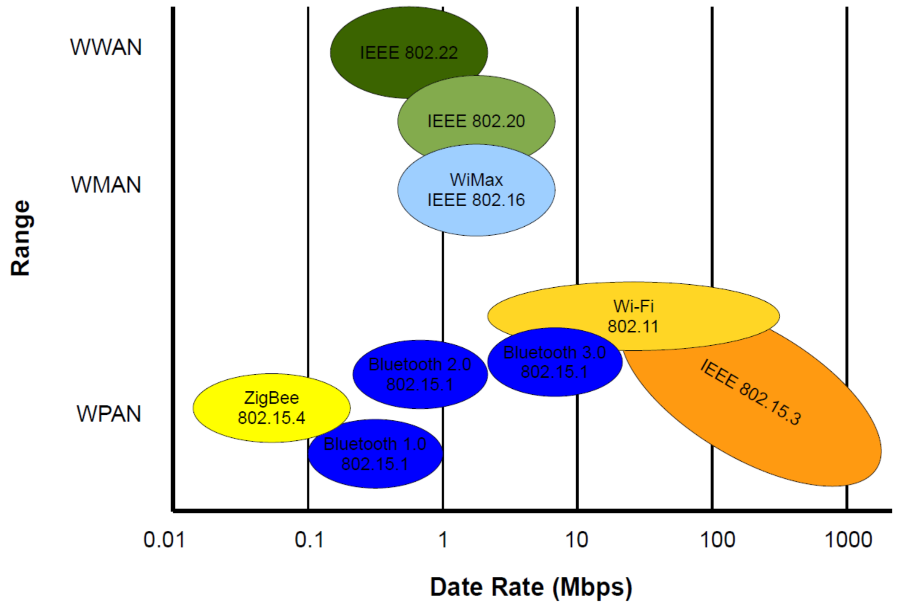


Figure 3. Range vs Data Rate for various IEEE standards. Image courtesy of Arthur Stachowicz

While the 802.15.4 protocol specify the MAC and physical layer, ZigBee protocols handle application layer and above which generally deal with network configuration, security, and application profile.

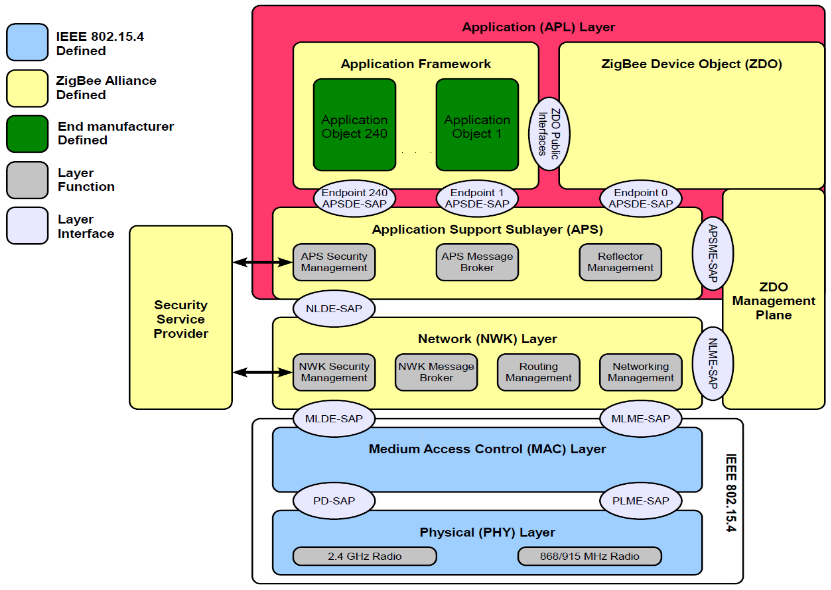


Figure 4. Protocol layers for ZigBee stack. Image courtesy of Arthur Stachowicz

Xbee Series 1 and Series 2 modules operate on the 802.15.4 and ZigBee standards respectively with small variations for specific uses. Due to its intended development for low rate sensor networks, Xbee series 2 with ZigBee protocols was used as a communication bridge.

B. ZigBee Network Topology

ZigBee is a global open standard for communication using the 802.15.4 protocol. Maintained by the ZigBee Alliance, transceivers communicate over ISM signal bands (2.4 GHz globally, 868MHz in Europe, and 915 MHz in US) with intended ranges of 10-100m.

ZigBee device firmware can have one of three functions which combine to form various network topologies. The three types are: ZigBee Coordinator (ZC), ZigBee Router (ZR), and ZigBee End Device (ZED).

ZigBee Coordinators (ZC) act as the central controller and parent node to both end devices and routers. They are in charge of network management functions such as storing security keys and network ids as well as handling network traffic. They are the most resource heavy nodes in terms of processing and local memory, and must be active for a network to exist.

ZigBee Routers (ZR) are capable of performing application layer tasks as well as acting as fully functional sensor nodes. Routers may function as network repeaters, extending network size by relaying information from end devices or other routers out of range of the Coordinator node. Routers are not necessary for a ZigBee network to exist, but are useful in forming sophisticated network structures or when network contains a large number of nodes.

ZigBee End Devices are not fully functional devices, they send information only to their parent node, which can be either a Router or Coordinator. Limited functionality makes End Devices the least power hungry network nodes. Generally, End Devices form the bulk of sensor nodes in a network structure.

In order to facilitate long battery life, ZigBee protocols allow for an easily programmable sleep mode for End Devices, periodically limiting computation and thus power consumption. Coordinator and Router nodes continually transmit beacon signals to alert child nodes of their presence. Upon waking, End Devices wait for a beacon signal and relay information before continuing their sleep cycle.

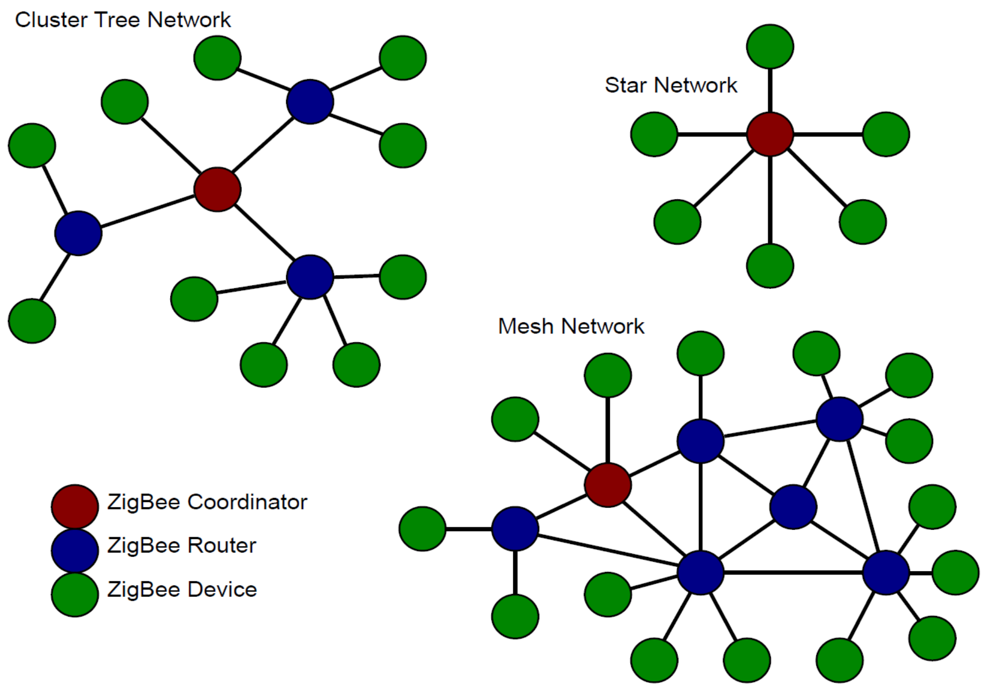


Figure 5. Potential ZigBee Network Topologies. Image courtesy of Arthur Stachowicz.

C. Xbee

Xbee module is a small programmable RF transmitter functioning on the IEEE 802.15.4 standard. The series 2 and series 2 PRO models operate with ZigBee protocols and have a

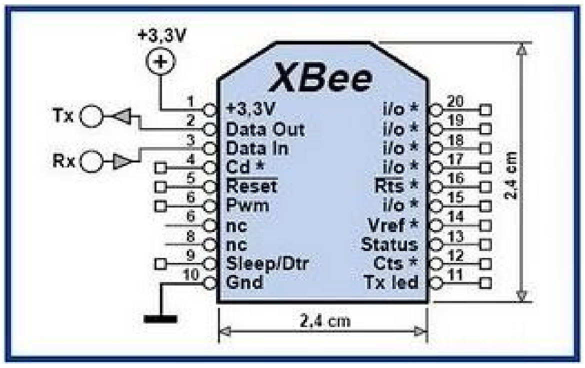


Figure 6. Xbee Series 2 module and Schematic. Images from Digi site.

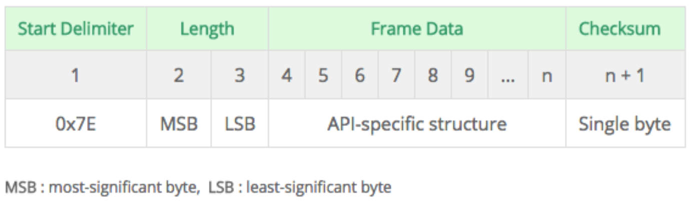
communication range of 30m indoor 90m outdoor and 90m outdoor 1600m outdoor respectively. They operate on 3.3V, transmit at up to 250,000bps and 115,200 baud, and are available with a variety of antennas for specific uses. They are relatively low power with a programmable sleep mode, and and are a recommended communication bridge between IoT devices in sensor networks due to a simple implementation and existing hardware integration [11].

1. AT vs API Mode

There are two types of UART schemes available for Xbee modules, AT and API mode.

AT mode is synonymous with transparent mode, each module has a single network destination and personal address. Configuration of network parameters must be done through command mode, either by the user or a micro controller. AT mode is useful for simple point-to-point communication or non-modular network topology.

API mode is recommended for larger networks with more complicated overall topologies. Messages contain a personal and destination address, which can be changed during run time depending on message type, making mesh topology simple to implement. Additionally, API mode contains packet delivery confirmation messages and has the option of escape parameters.



Start Delimiter	Length		Frame Data								Checksum
1	2	3	4	5	6	7	8	9	...	n	n + 1
0x7E	MSB	LSB	API-specific structure								Single byte

MSB : most-significant byte, LSB : least-significant byte

Figure 7. Standard API Frame Format. Image taken from Digi Website.

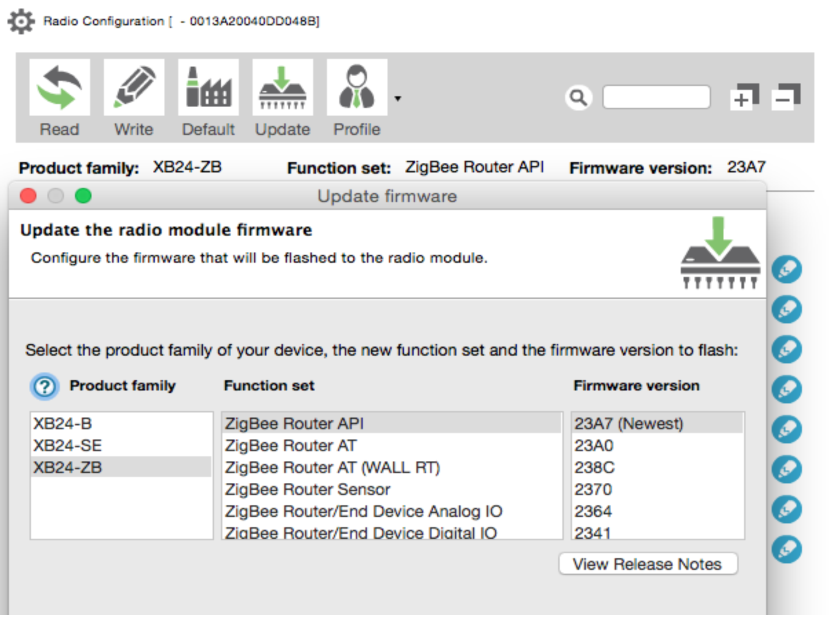
For this design I used API mode “2” with escaping for modular destination addresses, package delivery confirmation, and robust packet reception.

2. Programming

Xbee devices have a number of adjustable parameters. Each device is configured either through XCTU, an open source GUI interface maintained by Digi, or configured wirelessly using API commands via the local UART. For this project, a one Coordinator multiple End Device star topography is used to minimize energy expenditures of sensor nodes.

Xbee modules can be connected to a computer running XCTU via a micro usb to usb cable and an Xbee explorer board. Here, firmware can be updated and parameters configured.

First, update each unit with the latest version of its intended firmware (ZC, ZR, or ZED).



Product family: XB24-ZB Function set: ZigBee Router API Firmware version: 23A7

Update the radio module firmware

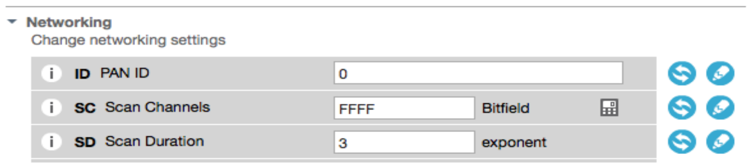
Configure the firmware that will be flashed to the radio module.

Select the product family of your device, the new function set and the firmware version to flash:

Product family	Function set	Firmware version
XB24-B	ZigBee Router API	23A7 (Newest)
XB24-SE	ZigBee Router AT	23A0
XB24-ZB	ZigBee Router AT (WALL RT)	238C
	ZigBee Router Sensor	2370
	ZigBee Router/End Device Analog IO	2364
	ZigBee Router/End Device Digital IO	2341

Figure 8. XCTU interface for Firmware Download

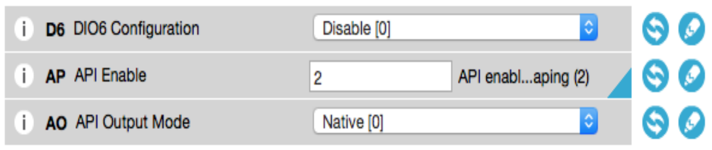
Each device must specify its Personal Area Network ID as a hexadecimal string on the range x0000 to xFFFF, all devices must operate on the same channel to communicate.


Networking
Change networking settings

ID PAN ID	0	
SC Scan Channels	FFFF	Bitfield
SD Scan Duration	3	exponent

Figure 9. XCTU Interface for Setting PAN ID

Lastly, API mode with escaping (API = 2) is used.



D6 DIO6 Configuration	Disable [0]	
AP API Enable	2	API enabl...aping (2)
AO API Output Mode	Native [0]	

Figure 10. XCTU Interface for Setting API mode 2

Additionally, End Devices have a configurable sleep mode that will be handled in later sections.

XCTU terminal application can be used to check network connectivity. If parameters have been assigned correctly, messages may be broadcast in the correct API format from one devices' terminal session to another.

Project network design is one ZigBee Coordinator and two ZigBee End Devices forming a simple star topology.

II. Arduino as Sensor Node

A. Overview

Arduino is a small, open source micro-controller initially intended for simple electronics prototyping applications, but has found a niche as low-cost reliable hardware. The design is open sourced, with programmable analog and digital ports to serve a variety of functions including local signal analysis and processing.

For this project I used and Arduino Uno v3 board. The Uno uses the Atmel ATmega328P processor chip with 32KB flash memory (.5KB used for bootloader), 2KB RAM, and 1KB EEPROM, is capable of 16MHz clock speed, and contains 6 Analog and 14 Digital I/O pins.

Arduino uses its own programming language (based on Wiring) and IDE (based on processing) with a large community of engineers contributing software libraries for interfacing relevant technologies.

The Arduino Uno serves as a signal processor for the data collector node, periodically reading and formatting GPS data and temperature for transmission via Xbee to the network Coordinator.

B. Temperature Sensor

1. Hardware

For this project the DS18B20 temperature sensor is used. The DS18B20 measures temperatures from -55°C to +125°C (–67°F to +257°F) with ±0.5°C accuracy from –10°C to

+85°C. Input voltage ranges from 3V-5.5V which may be extracted from the data line enabling a two wire connection scheme.

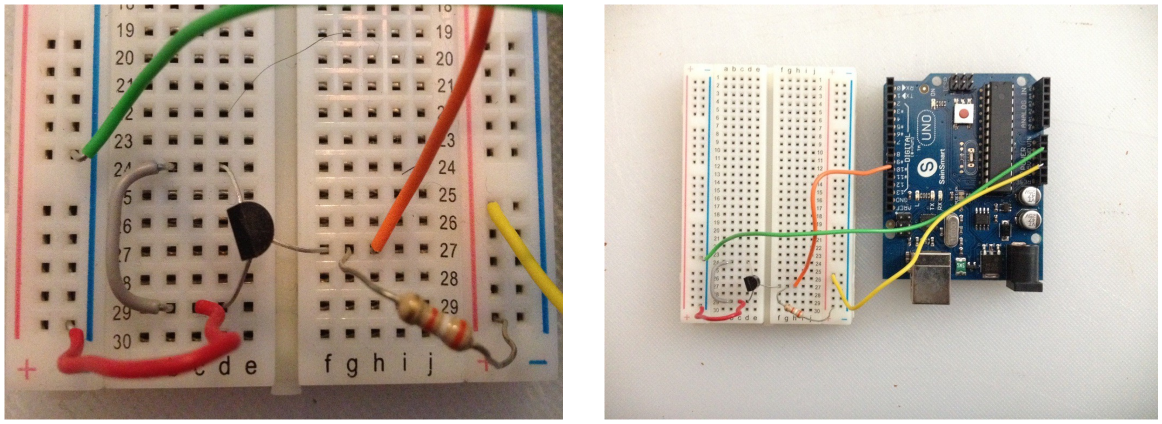


Figure 11. Temperature sensor hardware

The BS18B20 operates as a standard one wire sensor, with a signaling scheme that performs half-duplex bidirectional communications between a master and slaves sharing a common data line. Both power and data are transferred along this single line, with slaves using an internal capacitor to charge power when data signal is high for use when signal drops low. Pins GND and V++ are connected to ground and a 47K pull-up resistor is wired in series with the data pin which is then fed into an Arduino digital input port.

2. Software

the Arduino uses both the *OneWire* and *DallasTemperature* Arduino libraries to read temperature values from the DS18B20 sensor.

OneWire library incorporates the one wire sensor signaling scheme mentioned above into simple function calls while *DallasTemperature* adapts the general *OneWire* library to the specific use of the DS18B20 sensor. Basic loop runs as follows:

import the requisite libraries:

```
#include <OneWire.h>
#include <DallasTemperature.h>
```

Initialize your states by declaring a `OneWire` instance on the data input port and passing that reference to `DallasTemperature`

```
#define ONE_WIRE_BUS 10
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);
```

In the `setup()`, start the library by calling `begin()` on the sensor instance.

```
void setup(){
     sensors.begin()
     }
```

The loop functions by sending the sensor a request for a temperature measurement, then calling `getTemperatures()` on the sensor instance. Pull from the index what is needed, this project reads temperature in degrees Celsius.

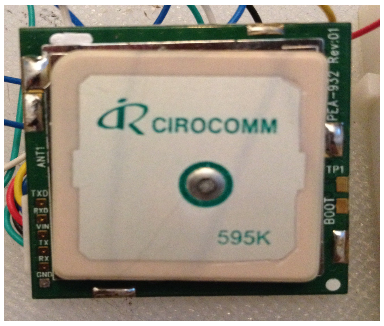
```
void loop(){
sensors.requestTemperatures();
sensorValue = sensors.getTempCByIndex(0);
}
```

Links to the *OneWire* and *DallasTemperature* Github repositories in Appendix

C. GPS Measurement

1. Hardware

This project uses the Parallax PMB-648 SiRF GPS Module. The PMB-648 module operates on NMEA0183 v2.2 protocol, contains a built in antennae, and 20 satellite communication channels. Module requires 3.3-5V input, has TTL or RS-232 asynchronous serial interfaces at 4600 baud with position accuracy +/- 5m. The PMB-648 requires no additional hardware to interface with the Arduino input ports, connect 5V and GND to corresponding Arduino ports, device transfers GPS data via TTLTx line.



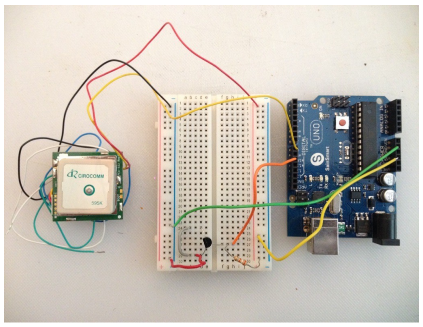


Figure 12. Parallax PMB-648 SiRF GPS Module Arduino Interface

2. Software

PMB-648 module uses *TinyGPS* and *SoftwareSerial* Arduino libraries for parsing NMEA GPS strings.

TinyGPS is designed to alert the Arduino when NMEA GPS data is received, and to extract useful parameters from the data. Since NMEA data comes from a serial GPS unit, either the built in Arduino hardware serial interface, or a software serial port must be used.

Xbee communication uses the hardware serial port, so the *SoftwareSerial* library is required.

Basic code structure is as follows:

Import the requisite libraries

```
#include <SoftwareSerial.h>
#include <TinyGPS.h>
```

Initialize the software serial communication ports and declare an instance of `TinyGPS`

```
SoftwareSerial gpsSerial(8, 9);
TinyGPS gps;
```

Tell Arduino to listen to the software serial port, and wait until data is available. TinyGPS library contains a boolean function `encode()` that parses data one character at a time and returns true when a complete statement has been received. Proper functionality states to hinge an if statement on the boolean return of `encode()`. Once inside the if statement, *TinyGPS* contains function calls for retrieving parameters of interest from the data string. This project is only interested in the latitude and longitude of the device.

```
gpsSerial.listen();
while(gpsSerial.available()){
     if(gps.encode(gpsSerial.read())){
          gps.get_position(&lat,&lon);
     }
```

links to the full *SoftSerial* and *TinyGPS* Arduino libraries Github repositories in Appendix

D. Xbee Communication

1. Xbee Setup

The Arduino serves as the networks data collection node, transmitting data packets to the Coordinator. The ZigBee equivalent firmware is End Device (ZED) or router (ZR), ZED requires less power as a non-fully functional device, so it is preferred. XCTU is used to set PAN ID arbitrarily to 3001, API mode = 2 for API with escape parameters following the procedure outlined in the Xbee section above.

2. Hardware

Xbee pins are spaced at non-standard intervals requiring additional hardware for Arduino interfacing. Proprietary Xbee shields are relatively inexpensive solutions, allowing for a top-mount platform coupling the Xbee pins to their corresponding Arduino inputs. Shields contain basic circuit protection such as voltage regulation, and jumpers to switch between programming Arduino and Xbee, both of which require the Arduinos serial communication ports. The Xbee shield receives power and GND through ICSP pins, leaving the 5V and GND headers on Arduino available for external circuits. Xbee Tx and Rx pins are connected to Arduino Digital port Rx and Tx respectively, Digital pins 8-13 are left open for additional sensors.

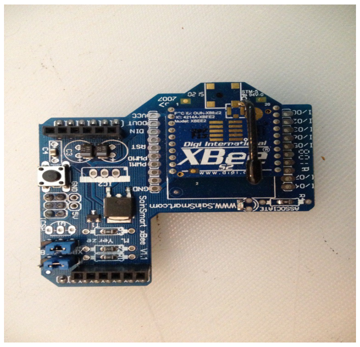
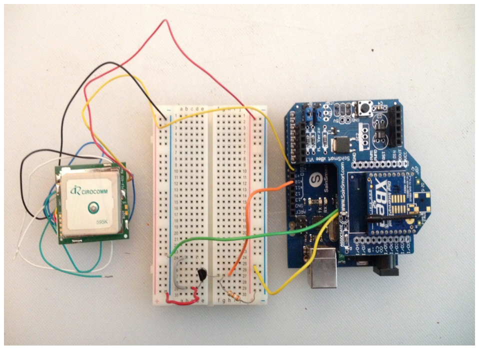



Figure 13. Xbee Series 2 with Xbee shield and hardware for End Device

3. Software

The *Arduino-Xbee* communication library by Andrew Rapport is used to send Xbee transmission requests. The *Xbee* library handles the potential complexity of API message formatting with message type declarations constructing the API headers and addressing. Import the requisite libraries

```
#include <XBee.h>
```

An `Xbee` instance is declared and the message payload is defined as an unsigned int of the necessary size. This projects transmits packets with 10 bits of data.

```
XBee xbee = Xbee();
uint8_t payload[] = { 0,0,0,0,0,0,0,0,0,0};
```

Xbee destination address must be specified. Since we are using API mode this address can change throughout runtime without additional configuration, but for the star topology implemented, only the Coordinator address must be specified. Additionally, declare an instance of the API message type that will be used, here we format an API transmission request.

```
    XBeeAddress64 addr64 = XBeeAddress64(0x0013a200,
0x40d8d723); // address of Coordinator Xbee
    ZBTxRequest zbTx = ZBTxRequest(addr64, payload,
sizeof(payload)); // ZigBee Transmission with addressing,
// payload and checksum
```

In `setup()`, initiate the hardware serial instance at the proper baud rate and pass the `Serial` instance to the `Xbee` instance

```
    void setup(){
        Serial.begin(9600); // connect serial
        xbee.setSerial(Serial);
    }
```

Data can be read in as any type, payload is formatted as hexadecimal strings with escape characters. For this project, temperature and GPS data are initially read as floats from their respective sensors. Convert floats to byte sized numerical strings and pass into the payload array for Xbee transmission. Here `pin5` is the temperature sensor float value.

```
void loop(){
        pin5 = int(100*sensorValue);
        payload[8] = pin5 >>8 ;
        payload[9] = pin5;
```

GPS data is stored similarly. Once payload is correctly formatted, call the `send()` function on the updated data packet for Xbee transmission

```
    xbee.send(zbTx);
    }
```

Link to the *Arduino-Xbee* library Github repository can be found in Appendix

III. Raspberry Pi as Data Aggregate Node/Cloud Gateway

A. Overview

Raspberry Pi is a small, inexpensive, fully functional linux computer. Founded in may 2009, the Pi was initially intended as an educational tool, a hackable bare-bones shell adaptable for many uses. It has since been embraced by the maker community and others as a competent module for small scale processing applications requiring an OS kernel for asynchronous multitasking.

For this project, a Raspberry Pi 2 model B is used. The model B comes equipped with a 900MHz quad-core ARM Cortex-A7 CPU, 1GB of onboard RAM, four USB ports for interfacing various technologies, an ethernet port, 40 GPIO pins for signal I/O, a micro SD slot, and HDMI cable for a GUI via monitor.

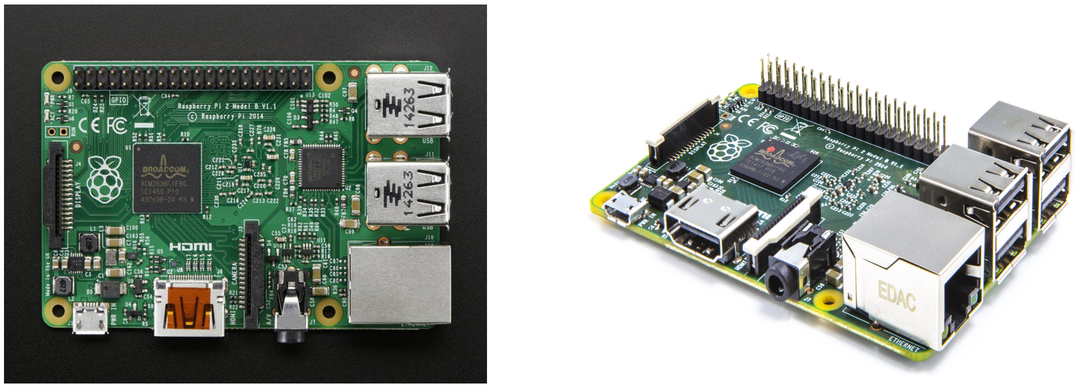


Figure 14. Raspberry Pi Model 2B. Images taken from Raspberry Pi website

The Cortex-A7 processor is capable of running the full range of GNU/Linux distributions including Windows 10. Due to its reliability and large support community, OS Raspbian Jessie was used via a 16GB flash drive.

The data aggregate node/internet gateway must receive large amounts of sensor data and transmit them via wifi, both relatively energy intensive tasks. The Pi requires a strict 5V regulated power supply with current draw up to 2A depending on application hardware. A standard wall wart meeting the above specifications is used.

B. Xbee Communication

1. Xbee Setup

The Raspberry Pi serves as the networks data aggregate node, receiving data packets from wireless sensor nodes. ZigBee equivalent firmware is network coordinator (ZC). XCTU is used to set PAN ID arbitrarily to 3001, API mode = 2 for API with escape parameters. Procedure is identical to process described in previous sections.

2. Hardware

The Coordinator Xbee draws power from the Raspberry Pi's 3.3V and GND power lines, and Xbee Tx and Rx lines are connected to serial Rx and Tx. All RPI connections are via the Pi's GPIO pins. Female to male wires are used to carry the corresponding lines to a breadboard, where an Xbee breakout board converts the Xbee pin spacing to the 2.54mm breadboard standard to facilitate connection.

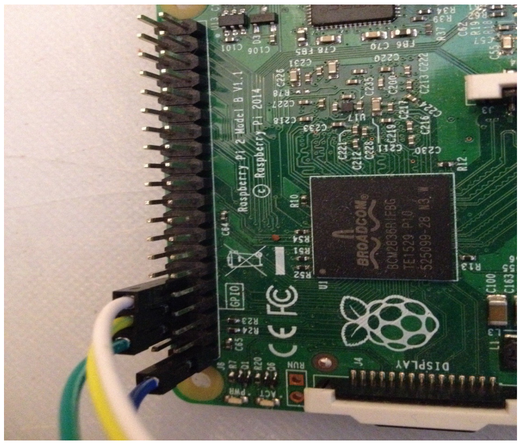
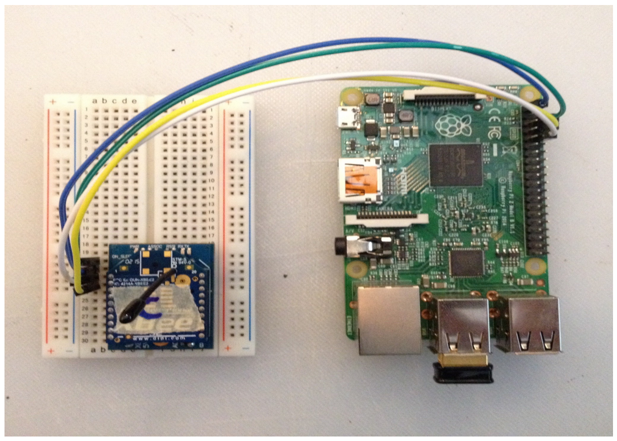



Figure 15. Raspberry Pi hardware setup

3. Os Initialization

The Raspberry Pi GPIO serial port is referenced as /dev/ttyAMA0 by Jessie OS. Default OS configuration has Jessie passing kernel messages via ttyAMA0, which must be disabled for clean communication. Jessie uses sytemd for initialization, and /boot calls cmdline.txt to initialize kernel message passing.

Edit /boot/cmdline.txt and remove any reference of ttyAMA0. Functionally, this means removing

```
console=ttyAMA0, 115200
```

from the cmdline.txt file.

Additionally, disable systemd services on ttyAMA0 to clear for incoming communications

```
$ sudo systemctl stop serial-getty@ttyAMA0.service
```

If functioning correctly, opening a terminal emulator (i.e. minicom) on ttyAMA0 will be clear while Xbee is disconnected.

4. Software

The Raspberry Pi uses pythons' Xbee, ASCII, and serial libraries to receive and parse communication on the ttyAMA0 serial GPIO. The Python Xbee library is similar to Arduino Xbee, containing instance declaration of Xbee types with function calls for API message transmission, formation, and reception. Serial Python library provides a backend for serial port communication. ASCII `hexlifier()` parses the hexadecimal API data transmission payloads into human readable format for data analysis and storage. Code is structured as follows:

Import the requisite libraries

```
import serial
import xbee
from binascii import hexlify
```

Initialize the serial port library by declaring a serial instance on ttyAMA0 with a baud rate of the Xbee at the End Devices

```
SERIAL_PORT = '/dev/ttyAMA0'
BAUD_RATE = 9600
ser_port = serial.Serial(SERIAL_PORT,BAUD_RATE)
```

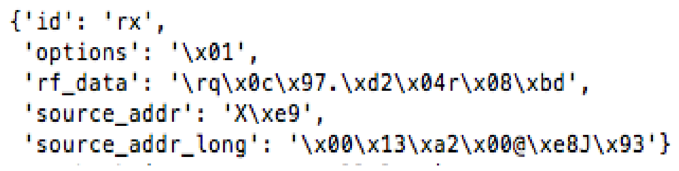
Pass the reference to a ZigBee instance in the Xbee library

```
xbee1 = xbee.zigbee.ZigBee(ser_port,escaped=True)
```

use the `wait_read_frame()` function on the Xbee instance to receive data.

```
data_samples = xbee1.wait_read_frame()
```

upon the event of signal reception, `data_samples` containes the API transmission packet in raw format. The Xbee library parses received statements into a dictionary that can be indexed by the user. Python function pprint can display the transmission in its raw form.



```
{'id': 'rx',
 'options': '\x01',
 'rf_data': '\rq\x0c\x97.\xd2\x04r\x08\xbd',
 'source_addr': 'X\xe9',
 'source_addr_long': '\x00\x13\xa2\x00@\xe8J\x93'}
```

Figure 16. Print to Console of Received Tx Packet in raw form

Here we can see a full list of received parameters and their dictionary identifiers. For this project, the important parameters are the 'source_addr_long', the MAC address of the transmitting sensor node, and 'rf_data', the data payload of the transmission in hexadecimal format. Extract the relevant parameters using standard indexing, and since Xbee API mode "2" formats transmissions in hexadecimal with escape parameters, convert data using the `hexlifier()` function.

```
address = data_samples['source_addr_long']
address=hexlify(address)

samples = data_samples['rf_data']
samples = hexlify(samples)
dataStr = str(samples)
```

variables `dataStr` and `address` are now human readable. The End Device Xbee sends GPS and temperature values in the same data string, so `dataStr` must be indexed and further assignments made. Initial Arduino code sends data as hexadecimal, which makes floats difficult to format. Therefore, float sensor variables such as temperature, latitude, and longitude are converted to unsigned integers before transmission, and must be converted back to floats in Raspberry Pi *Python* script. Knowledge of significant figures for sensor values on the Arduino side is necessary for proper indexing and conversion. *Python* script is gnostic to separation points in initial `dataStr` variable and uses the information to assign the correct values.

```
        lat = int(dataStr[1:4],16)*.01 + int(dataStr[5:8],16)*.000001
        long = -1*int(dataStr[8:12],16)*.01 -
int(dataStr[13:16],16)*.000001
        samples = int(dataStr[17:20],16)*.01
```

Latitude and longitude are transmitted with a precision of 6 decimal places and temperature a precision of 2. The first [1:8] of dataStr are the float value of latitude, [8:16] of dataStr are float values of longitude, and [17:20] of dataStr are float temperature in degrees Celsius. Since project development was in Santa Barbara, longitude is a negative value.

C. Cloud Gateway

1. Hardware

Built in Ethernet and wifi capabilities make transforming the Raspberry Pi into an internet gateway simple. Ethernet cable or external USB wifi card offer plug-and-play functionality for internet communication. Due to its spacial flexibility, an Edimax EW-7811Un external wifi card is used.

2. Software

Python *requests* library is used to send get requests to cloud server. *Requests* is an abstract HTTP API library that is simple to implement and popular. The cloud server is configured to update the database with URL parameters received on a specific URL. Implementation with requests library is straightforward:

import requisite libraries

```
    import requests
```

format URL with sensor data as parameters, as well as additional useful parameters of interest such as static Coordinator GPS and MAC address. Python string replacement is used to fill a standard URL with values on each loop. Here, the cloud server is located at `sensornodeserver.heroku.com`

```
    url = "sensornodeserver.heroku.com/reports/create?
report[temp]={0}&report[sensor_mac]={1}&report[controller_mac]={2}&
report[lat]={3}&report[long]={4}&report[controller_lat]={5}&report[
controller_long]={6}".format(samples,address,macAddr,lat,long,contr
oller_lat,controller_long)
```

send url to update database

```
    r = requests.get(url)
```

Link to Python *requests*, *ASCII*, *Xbee*, and *serial* Github repositories available in the Appendix

3. Boot at startup

The design of the Coordinator node necessitates a continuous runtime for the duration of the sensor network. Two options are available: use a Raspberry Pi `screen` program to maintain a computing environment with a break in ssh, or create a `systemd` service to be run during system boot. The `screen` solution will fail with a momentary power outage since `screen` must be manually initiated to persist, thus a `systemd` service is more robust to failure modes.

Create a new service

```
$ sudo vim /lib/systemd/system/[service name].service
```

Edit service script with following text to execute a Python script after the `user` environment is available

```
[Unit]
Description=[service description]
After=multi-user.target

[Service]
Type=idle
ExecStart=/usr/bin/python ["path to python script"]

[Install]
WantedBy=multi-user.target
```

Change the permissions of the newly created service to be accessed by `systemd`

```
$ sudo chmod 644 /lib/systemd/system/[service name].service
```

Inform *systemd* of new service

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable [service name].service
```

Pi will now run the Python script on boot as a `systemd` service.

IV. Cloud Server

A. Overview

The basic web development stack consists of an Operating System, a web server, a web application layer, and a database.

This project uses Heroku as a platform as a service, which takes care of setting up and maintaining the operating system (*Linux*), web server (*Puma*), and database structure (*SQLite*), leaving the developer to tailor the web application layer to their particular use case. Because of its reputation as one of the easiest initial learning curves for development, *Ruby on Rails* is used as a web application framework.

Ruby is a high level language that abstracts a lot of structural complexities, while *Rails* sets a powerful framework that automates standard resource generation for streamlined development. *Rails* framework splits the web application layer into three processes separated by core functionality: Model, View and Controller.

B. Rails MVC Architecture

The Model (*ActiveRecord*) is in charge of database queries and database management. *ActiveRecord* is an object relational mapping (ORM) technique that includes data access logic as part of the object structure. The Model receives requests from the Controller and performs object oriented database operations.

Browser requests are received by a router, which directs the request to the corresponding Controller. The Controller (*ActionController*) interprets the browser request and directs the Model and View to compile the correct response. Generally, the Controller sends requests to the Model for Object Relational Mapping of database parameters, and requests to the View for HTML rendering. The Controller can be thought of as a middle man for the Model and View.

The View (*ActionView*) accepts data retrieved by the Model via the Controller and compiles the response to be sent to the browser. *ActionView* templates are written using embedded *Ruby* in tags mingled with HTML, and compile the browser response to the end user.

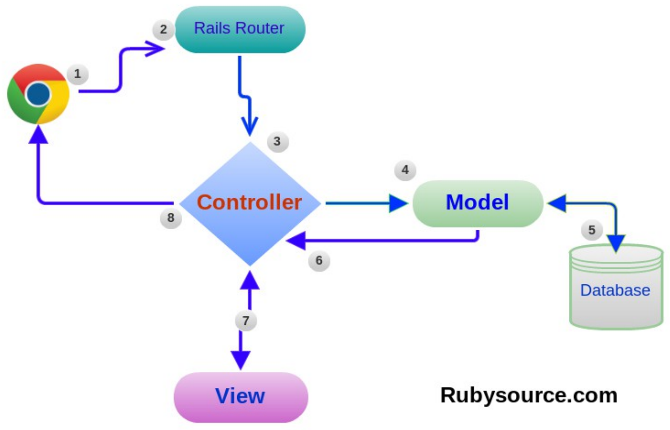


Figure 17. Block Diagram of high level overview of Rails MVC structure. Image via Rubysource.com

C. Web Application Design

The Web Application has two main functions: accept data as URL parameters to update the *SQLite* database, and compile data in a useful human readable form with a browser request. This can be accomplished with the use of a single Controller/View setup.

1. Routes and Database

Rails uses *SQLite* as a relational database management system to construct an embedded database for local storage on application hardware. Unlike *MySQL*, *SQLite* does not operate with a client-server relationship and is instead self-contained and server-less, *SQLite* reads and writes locally to ordinary disk files.

Rails creates a new database object via migrations, a *Ruby* DSL that allows the database schema and database changes to be database independent (see Appendix for example migrations).

There are eight parameters of interest for each sensor report: Temperature (C), Sensor Latitude, Sensor Longitude, Sensor MAC Address, Controller MAC Address, Controller Latitude, Controller Longitude, and Timestamp (hh/mm/dd). The database can be updated to contain all parameters with a database migration and `rake` (*Ruby* make). An outline of the database schema can be seen in the projects `schema.rb` (see Appendix).

The *Rails* router functions at an abstract level to direct incoming traffic flow to the requisite Controller. The `routes.rb` file exists in a do loop, specifying each request URL, its get or post API, and its Controller destination. For a single Controller/View web app, only a few lines need to be entered into the `routes.rb` file.

The project web app receives get requests on the /reports and reports/create URLs and directs them to the reports Controller. The following code is placed inside the `routes.rb` `do` statement

Specify a URL that is passed to the Controller named `index` as the variable `reports`

```
 get '/reports/', to: "reports#index", as: "reports"
```

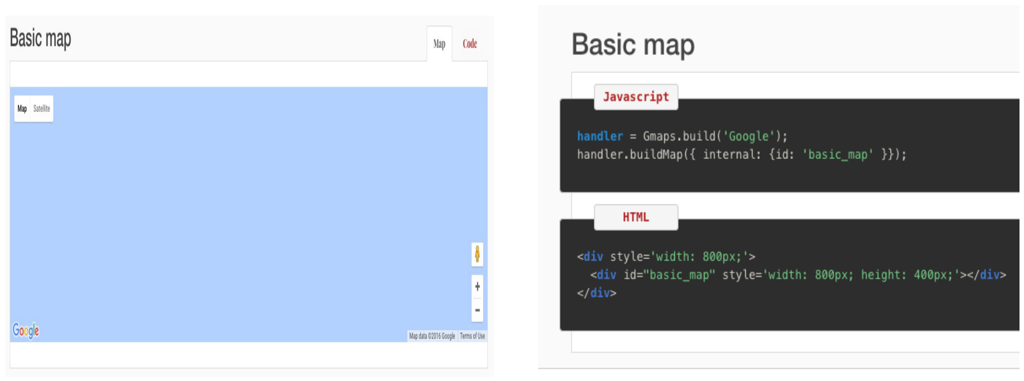
Pass the URL /reports/create to the Controller function `create`

```
get 'reports/create'
```

See Appendix for the full `routes.rb` file.

2. Gmaps4Rails

In order to present GPS and sensor data in a convenient and useful format, a map is rendered and each End Device and Coordinators location is superimposed. *Gmaps4Rails* is a *Ruby* gem (plug and play library) that incorporates the google maps API into a *Rails* project for easy implementation. *Gmaps4Rails* gem works with the Controller to collect data from the local *SQLite* database to pass to the View for rendering. The View then calls a google *JavaScript* executable to render a map, and populates the map with markers. More extensive description of the *Gmaps4Rails* code used in this project can be seen in the Controller subsection of this chapter.



```
handler = Gmaps.build('Google');
handler.buildMap({ internal: {id: 'basic_map' }});
```

```
<div style='width: 800px;'>
  <div id="basic_map" style='width: 800px; height: 400px;'></div>
</div>
```

Figure 18. Basic map Gmaps4Rails HTML rendering (left) and corresponding code (right). Image taken from Gmaps4Rails live examples page: http://apneadiving.github.io/

Before using the *Gmaps4Rails* gem, the correct dependencies must be satisfied. *Underscore* and *gmaps/google* must be added to the project pipeline and *underscore.js* installed in the *Javascript* assets folder of *vendor*. Additionally, specific *JavaScript* dependencies must be sourced from the *Gmaps4Rails* Github at the beginning of each HTML file. Full installation instructions can be found at the *Gmaps4Rails* Github cited in the references.

For example *Gmaps4Rails* Controller and View implementation see code in Appendix.

3. Controller

This project contains one core Controller named `reports_controller`. `Reports_controller` has two main functions: send incoming data to the Model for database storage, and act as a data transfer middleman between the Model and View. Therefore, two main functions are required.

The functions `create` and `report_params` handle passing incoming URL parameters from /reports/create to the Model for database storage. `Create` saves incoming data as the global variable `@report`, and calls the function `report_params` on the `@report` instance.

```
@report = Report.new report_params
```

`report_params` instructs the Model to save URL parameters fitting the description of keywords for each relevant data-type. Parameters must contain the keyword “report” and are stored under their corresponding identifier (temp, lat, long, etc.).

```
params.require(:report).permit(:temp, :lat, :long,
:sensor_mac, :controller_mac, :sleep, :voltage,
:controller_lat, :controller_long)
```

Function `index` handles retrieving data via Model and constructing a map framework to pass to View for rendering. `Index` defines a global *Ruby* variable named `@reports` as collection of latest received unique sensor MAC addresses.

```
@reports = Report.pluck(:sensor_mac).uniq.sort.map {|mac|
Report.where(sensor_mac: mac).last }
```

`@hash` uses *Gmpas4Rails* library to construct a global hash of each parameter in `@reports` to generate markers for the final map. Each marker consists of a latitude and longitude value, as well as a custom infobox displaying relevant sensor data when clicked by a user.

```
@hash = Gmaps4rails.build_markers(@reports) do |report,
          marker|
marker.lat report.lat
 marker.lng report.long
```

Sensor markers are generic blue gmaps markers pulled from the gmaps API.

```
marker.picture({
        :url => "http://chart.apis.google.com/chart?
chst=d_map_pin_letter&chld=S|FF0000|000000",
        :width  => 32,
        :height => 32
})
```

The project was developed with the idea of a cloud server acting as a shared database for multiple ZigBee star networks. Therefore, even though a Coordinator MAC address is unique for a single ZigBee network, a Coordinator MAC addresses in the *SQLite* database may not be unique. Since the user is interested in all unique controller MAC addresses as well as unique sensor MAC addresses, a second loop is run to find each unique controller MAC address for all parameters in `@reports`:

```
@controllers = Report.pluck(:controller_mac).uniq.sort.map
{|mac| Report.where(controller_mac: mac).last }
@hashed = Gmaps4rails.build_markers(@controllers) do |
controller, marker|
```

The final `@hash` variable is a combination of `@hash` and `@hashed` parameters and is passed to View for rendering.

```
@hash += @hashed
```

Full `reports_controller` code available in Appendix.

4. View

Project uses a single View. Each *Rails* View contains multiple html files with initialization, but project development focused on a single html file named `index.html` to render sensor data upon a browser request. `Index.html` begins by initializing a map with an external src script provided by google via their maps Github.

```
    <script src="//maps.google.com/maps/api/js
v=3.23&sensor=false&client=&key=&libraries=geometry&language=
&hl=&region="></script>
  <script
src="//cdn.rawgit.com/mahnunchik/markerclustererplus/master/d
ist/markerclusterer.min.js"></script>
```

The `index.html` main loop initializes the map parameters in HTML and executes a *JavaScript* script calling the *Gmaps4Rails* library to populate the map with data received by the `reports_controller`. Markers are passed as the `@hash` parameter, and *Gmaps4Rails* library handles zoom and formatting.

```
    handler = Gmaps.build('Google');
    var mapOptions = { mapTypeId:
    google.maps.MapTypeId.SATELLITE };
    handler.buildMap({ provider: mapOptions, internal: {id:
'map'}}, function(){
    markers = handler.addMarkers(<%=raw @hash.to_json %>);

    handler.bounds.extendWith(markers);
    handler.fitMapToBounds();
```

Additionally, an HTML script is executed beneath the map to form a table listing the latest data parameters for each unique sensor node contained in `@reports`.

Initialize the table with standard HTML and populate the first row with column titles

```
    <table style="width:100%">
     <tbody>
      <tr>
       <td>Sensor Mac Address</td>
       <td>Controller Mac Address </td>
       <td>Temperature (C)</td>
       <td>Latitude</td>
       <td>Longitude</td>
       <td>Time Logged</td>
     </tr>
```

View is capable of reading identifiers for nested *Ruby* code, making variable passing simple. Tell `index.html` to execute *Ruby* statement with a "`<%`" identifier, and index variables via *Ruby* to populate the HTML table. Here `report` is the global variable defined by the controller containing the latest Report object for each unique sensor MAC address.

```
</tr>
  <% @reports.each do |report| %>
  <tr>
   <td><%= report.sensor_mac %></td>
   <td><%= report.controller_mac %></td>
   <td><%= report.temp %></td>
   <td><%= report.lat %></td>
   <td><%= report.long %></td>
   <td><%= report.updated_at %></td>
  </tr>
  <% end %>
 </tbody>
</table>
```

Full `index.html` code available in the Appendix.

5. Model

Generally, object oriented frameworks require a lot of configuration code. However, Rails operates via inheritance and only requires additional configuration if an object deviates from the default. The project Model contains two parameters: `sensors` and `reports`. Each inherit from `ActiveRecord::Base` specifying the default structure of the class object. Additional configuration uses *Ruby* keywords to dictate relationship between two objects; each `report has_one :sensor` and each `sensor has_many :reports`. Model responds by updating the database object structure of `Report` parameters for easy indexing later on. For full Model initialization see Appendix.

V. Power Analysis

A. Overview

The major incentive for the 802.15.4 communication protocol is its low frequency low power transmission scheme. In this context it is relevant to examine the average power consumption of an End Device implementing the ZigBee communication protocols.

Arduino is a prototyping board due to the general nature of its components. In addition to the AtMega328P processor, the stock Arduino Uno v3 hardware includes an AtMega16U2 microcontroller for converting the USB signal to serial, an LCP1117 linear voltage regulator for Vin to 5V conversion, an LP2985-N linear voltage regulator for power supply to 3.3V conversion, and a power LED, all of which draw current while the Arduino is battery powered.

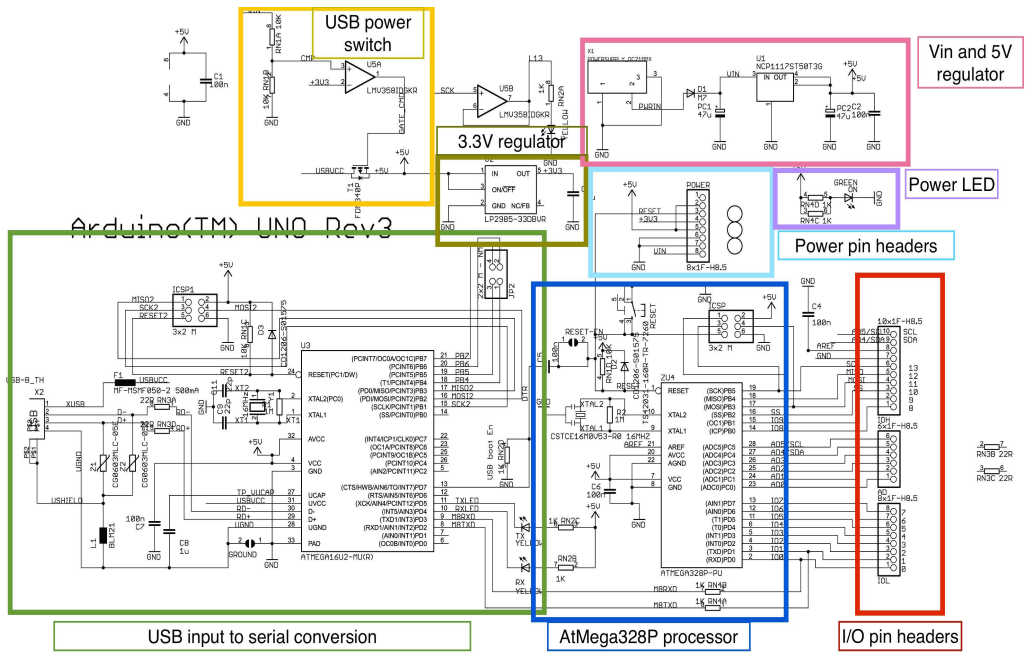


Figure 19. Arduino Uno v3 annotated schematic

However, if standard Arduino Uno hardware is desired then End Device power consumption can be minimized in three ways.

First, Xbee devices have a programmable sleep mode that can drastically decrease power consumption. Unlike Arduino, Xbee does not have a voltage regulator to dissipate energy, devices only contain a RF communicator and a small microprocessor, and require a regulated 3.3V input. In order to limit power consumption of their onboard processor, Xbee devices can implement a watchdog timer based sleep cycle to stop computation while the Xbee is idle.

Second, the Arduino is similarly capable of using a sleep mode with a watchdog timer interrupt on its AtMega328P processor. While the Arduino is idle, the AtMega328P can halt unnecessary hardware functionalities to decrease overall power consumption.

Third, the GPS sensor draws relatively large current while running (~32mA). If the End Devices are stationary, then only one valid GPS string needs to be sent at node initialization. The GPS sensor can be powered by an Arduino output pin which is capable of sourcing up to 40mA. Since only one valid reading is needed, GPS code can be run in the Arduino startup loop which is run once upon system initialization. Once a valid GPS string is received the Arduino shuts power to the GPS device by writing the output port to voltage low, reducing the nodes average power consumption.

In order to test the change in node efficiency when implementing the above methodologies, a single End Devices hardware and software was modified.

B. GPS Modification

The projects initial End Device hardware setup powers the Parallax PMB-648 SiRF GPS Module from the Arduino 5V source. In order to limit GPS power consumption, the device is instead powered via a digital I/O pin that can be specified at high (5V) or low (0V). Power to the GPS unit is enabled until an accurate latitude and longitude reading is known by the device.

In order to implement the changes, a few new variables must be added to the Arduino main loop, and code blocks shifted.

Define the Digital output pin the GPS unit draws power from, as well as an int named `GPScount`. The first few GPS readings are error prone, a more accurate value is achieved after a few strings have been received. `GPScount` specifies the number of times GPS strings are to be read before latitude and longitude vales are within acceptable error to their true values.

```
int GPSpower = 11;
int GPScount = 0;
```

in the setup loop, `GPSpower` must be specified as an output

```
pinMode(GPSpower, OUTPUT);
```

Since we only need a single GPS measurement at End Device initialization, the GPS code block can be executed entirely in Arduino `startup()`. Allow power to the GPS unit and follow a similar procedure to the one outlined in the GPS subsection above. At the end of the loop, turn power to GPS off by writing the Digital pin `GPSpower` low.

```
     while(GPScount < 10){
          digitalWrite(GPSpower,HIGH);

         gpsSerial.listen();
         while(gpsSerial.available()){ // check for gps data

            if(gps.encode(gpsSerial.read())){ // encode gps
data
                 gps.get_position(&lat,&lon); // get latitude
and longitude
```

While Inside this `if()` statement, save the GPS string to the payload variable and update the `GPScount`

```
               latInt = int(lat/10000);
               payload[0] = latInt >>8 ;
               payload[1] = latInt;

               ... //follow a similar procedure for storing
               rest of data

               GPScount = GPScount +1;
                    }
               }
          }
          digitalWrite(GPSpower,LOW);
     }
```

C. Xbee Sleep

Xbee series 2 End Devices make implementing sleep mode simple. Xbee sleep mode is configurable through XCTU, with a number of options for tailoring to a specific use. In this project, Xbee pin sleep is used.

1. XCTU Configuration

The specific Xbee sleep method must be specified and outbound flow control must be disabled. Both can be configured via XCTU, change the End Device sleep setting to pin mode (1), and disable the Xbee d7 parameter by setting the value to 0.

Additionally, the Coordinator node is gnostic to the approximate report frequency of the sEnd Devices. The XCTU parameters SP and SN are configurable as End Device sleep cycle duration (in hexadecimal) and End Device sleep cycles per transmission respectively. In order for the Coordinator to receive transmissions reliably from a sleeping Xbee, SP and SN must be set accordingly

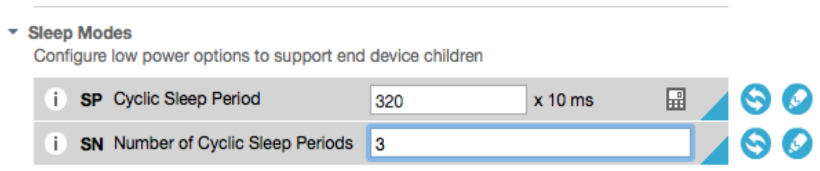

Sleep Modes
Configure low power options to support end device children
SP Cyclic Sleep Period 320 x 10 ms
SN Number of Cyclic Sleep Periods 3

Figure 20. Example SP and SN settings. The above configuration is for an End Device transmission every 24 seconds.

2. Hardware Modification

In the projects previous iteration, a proprietary Xbee shield was used to connect the Xbee to the Arduino. In order to change Xbee pin sleep status, the voltage on the Xbee DTR pin is changed. Logic high (3.3V) on the DTR pin enables Xbee sleep mode, and a logic low (0V) disables Xbee sleep. Since the Arduino Uno v3 is a stock 5V device, a secondary voltage regulator is needed to supply the 3.3V to the DTR pin. The Xbee device is placed on an external breadboard via an Xbee explorer unit. The Xbee is powered via the Arduino 3.3V source, and Tx and Rx lines are connected to the Arduino manually via wires.

In order to change the voltage level on the Xbee DTR pin, an Arduino Digital output pin is connected to the input of a Sparkfun breadboard power supply stick. The output of the power supply stick is connected to the Xbee DTR pin. When the Arduino digital output is configured high (5V), the breadboard supply stick outputs 3.3V to the DTR pin and the Xbee receives the command to sleep. When the digital output is sent low (0V), the Xbee receives the command to wake up.

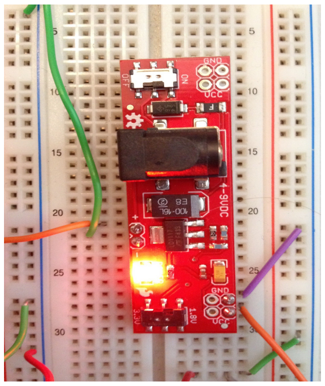


Figure 21. Sparkfun breadboard power supply stick

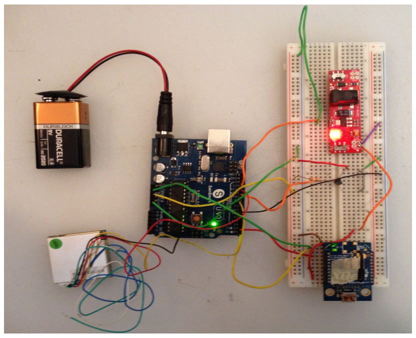


Figure 22. Modified End Device hardware

3. Software Modification

Specify the digital output pin to control the breadboard supply stick

```
int XbeeSleep = 12;
```

in the setup loop, specify the digital pin `XbeeSleep` as an output

```
pinMode(XbeeSleep, OUTPUT);
```

in the main loop, Xbee sleep state can be changed by either writing the `XbeeSleep` pin high (Xbee sleep) or low (Xbee wake)

```
digitalWrite(XbeeSleep,LOW); // wake the Xbee
digitalWrite(XbeeSleep,HIGH); //put the Xbee to sleep
```

D. AtMega328P Sleep

Putting the Arduino AtMega processor to sleep can be done via the AVR/sleep.h library. No additional hardware components are required, the requisite hardware is internal to the AtMega processor.

There are various types of sleep cycles available dependent on what the user is able to sacrifice. There is a trade off between processes left running and energy saved during a sleep session, with the deepest sleep mode (`SLEEP_MODE_PWR_DOWN`) disabling the most peripherals.

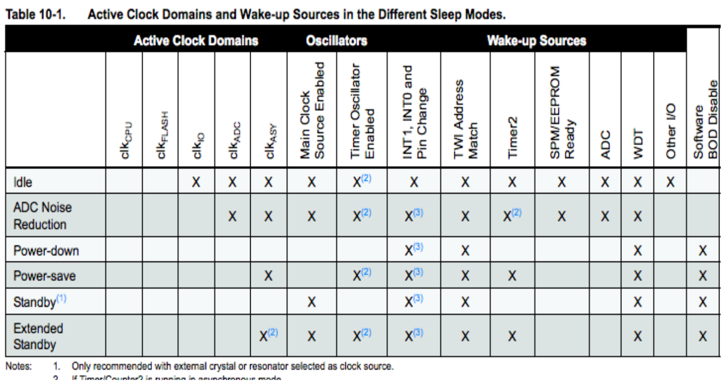


Table 10-1. Active Clock Domains and Wake-up Sources in the Different Sleep Modes.

	Active Clock Domains					Oscillators		Wake-up Sources							
	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT1, INT0 and Pin Change	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other I/O	Software BOD Disable
Idle			X	X	X	X	X[2]	X	X	X	X	X	X	X	
ADC Noise Reduction				X	X	X	X[2]	X[3]	X	X[2]	X	X	X		
Power-down								X[3]	X				X		X
Power-save					X		X[2]	X[3]	X	X			X		X
Standby[1]						X		X[3]	X				X		X
Extended Standby					X[2]	X	X[2]	X[3]	X	X			X		X

Notes: 1. Only recommended with external crystal or resonator selected as clock source.

Figure 23. Chart of available sleep modes and hardware tradeoff for the AtMega328P processor. Image taken from the AtMega328P datasheet

Since maximal energy conservation is important for this project, the `PWR_DOWN` sleep mode is used. `PWR_DOWN` sleep mode can be interrupted by the AtMega328P internal watchdog timer, one of the few peripherals left running. The watchdog timer (`WDT`) runs off a 128 kHz clock cycle, and is capable of delivering an interrupt or reset or both.

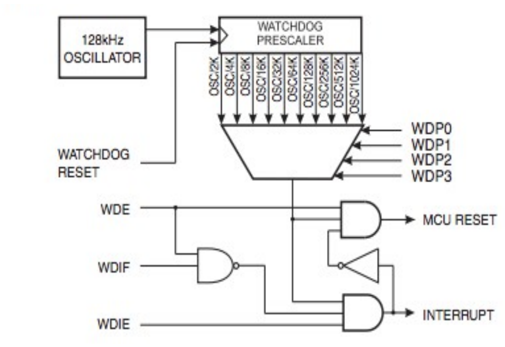


Figure 24. Watchdog timer. Image taken from AtMega328P datasheet.

Import the library

```
#include <avr/sleep.h>
```

Define a volatile int that counts the number of sleep cycles

```
volatile int sleep_count = 0;
```

in the `setup()` loop, configure the watchdog timer by setting the reset flag (bit 3) on the Micro Controller Unit Status Register (`MCUSR`). Additionally, set the watchdog configuration enable `WDCE` (bit 4) and watchdog enable `WDE` (bit 3) in the watchdog timer configuration status registry (`WDTCSR`). This allows changes to be made to the watchdog prescalers and resets the watchdog timer.

```
MCUSR = MCUSR & B11110111;
WDTCSR = WDTCSR | B00011000;
```

Set the watchdog timer prescaler value to 1024k, which will reset in about 8 seconds. This will be the time interval for one complete sleep cycle.

```
WDTCSR = B00100001;
```

lastly, enable the watchdog timer interrupt by changing the correct bits on the `MCUCSR` and the `WDTCSR`

```
WDTCSR = WDTCSR | B01000000;
MCUSR = MCUSR & B11110111;
```

With the watchdog timer configured, the Arduino may be sent to sleep by a series of function calls in the `main()` loop

```
set_sleep_mode(SLEEP_MODE_PWR_DOWN); // Specify sleep mode.
sleep_enable(); // Enable sleep mode.
sleep_mode(); // Enter sleep mode.
//Processor sleeps until WDT sends interrupt
sleep_disable(); // Disable sleep mode after waking.
if (sleep_count == sleep_total)
{ //execute periodic sampling and reset sleep_count }
```

Once all changes are implemented, a Duracell 9v 310mAh battery powers the modified End Device until exhaustion over a range of transmission frequencies to get an estimate of the average End Device current as a function of transmission frequency.

From an analysis point of view, it is interesting to know how the AtMega328P as a standalone processor would effect the average current of an Xbee End Device. This section attempts to determine the average power consumption of both the Arduino Uno/Xbee End Device hardware built in this project, and estimate the average current of an Xbee End Device with only the AtMega328P processor as additional hardware.

There are three main phases of the End Device software, each with a different average current. The node initialization includes Arduino startup, the node GPS configuration, and the Xbee End Device configuration process, all of which requires a set charge denoted as $Q_{startup}$. Since the node initialization is a standardized process, $Q_{startup}$ is assumed to be a constant parameter in the final formula. After the node initialization, the battery charge required by the End Device goes to powering two things.

First, the Arduino hardware (linear voltage regulators for 5V and 3.3V, and the AtMega16U2) requires some constant charge from the battery for the entirety of End Device runtime, denoted as Q_{idle} . Since the Arduino hardware is constantly drawing current, $Q_{idle}=(i_{USB}+i_{5\mathrm{V}\,reg}+i_{3.3\mathrm{V}\,reg})*T_{tot}=i_{idle}*T_{tot}$, where T_{tot} is the total time spent transmitting.

Second, the End Device is periodically responsible for reading and transmitting sensor data to the Coordinator via Xbee. The battery charge required by this process can be broken into two subsections: Q_{sleep} and Q_{Tx} .

Q_{sleep} is the battery charge required by the AtMega328P and Xbee processors during the End Device sleep process. $Q_{sleep}=(i_{AtMega_{sleep}}+i_{Xbee_{sleep}}+i_{Sprkfn\,reg})*t_{sleep}$, where t_{sleep} is the time the End Device spends sleeping.

Q_{Tx} Is the battery charge required during End Device data sampling and transmission. The Arduino is responsible for powering the temperature sensor and communicating with the Xbee via their serial connection while the Xbee receives from the Arduino and transmits to the Coordinator. Thus, $Q_{Tx}=(i_{AtMega_{awake}}+i_{Xbee_{awake}}+i_{Tmp\,sensor})*t_{awake}$ where t_{awake} is the time the End Device spends awake, and $t_{sleep}+t_{awake}=T_{tot}$.

The complete equation for the battery charge can be written as

$Q_{battery}=Q_{sleep}+Q_{Tx}+Q_{idle}+Q_{startup}$. Rewriting in terms of measurable quantities gives

$Q_{battery}=(i_{AtMega_{sleep}}+i_{Xbee_{sleep}}+i_{Sprkfn\,reg})*t_{sleep}+(i_{AtMega_{awake}}+i_{Xbee_{awake}})*t_{awake}+i_{idle}*T_{tot}+Q_{startup}$.

If we know the expected End Device lifetime, the number of Arduino loops can be found. A full Arduino loop is completed every $t_{loop}=t_{sleep\,cycle}+t_{Tx}$ seconds where $t_{sleep\,cycle}=1/f_{report}$ is the time spend sleeping for one sleep cycle, and t_{Tx} is the time spend on one transmission. The number of loops completed in an End Device lifetime is given by $n_{loops}=T_{tot}/t_{loop}=T_{tot}/(t_{Tx}+t_{sleep\,cycle})$. Then $t_{sleep}=t_{sleep\,cycle}*n_{loops}$ and $t_{awake}=t_{Tx}*n_{loops}$ seconds. Using this description, the charge of the battery can be rewritten in terms of the variables $t_{sleep\,cycle}$ and T_{tot} , and the number of loops n_{loops} :

$Q_{battery}=(i_{sleep})*t_{sleep\,cycle}*n_{loops}+(i_{awake})*t_{Tx}*n_{loops}+(i_{idle}*T_{tot})+Q_{startup}$
Where $i_{sleep}=i_{AtMega_{sleep}}+i_{Xbee_{sleep}}+i_{Sprkfn\,reg}$ and $i_{awake}=i_{AtMega_{awake}}+i_{Xbee_{awake}}+i_{Tmp\,sensor}$

VI. Results

A. ZigBee Network

Implementation of an Arduino-Xbee sensor node with Raspberry Pi ZigBee Coordinator/Cloud gateway to Rails database server was successful. Existing Arduino *Xbee*, *TinyGPS*, and *softserial* libraries collaborated for simple ZigBee data transmission to central Coordinator node. *Python Xbee* and *requests* library run smoothly on Raspberry Pi for receiving and parsing remote data and acting as internet gateway.

Prototype sensor and controller nodes are assembled.

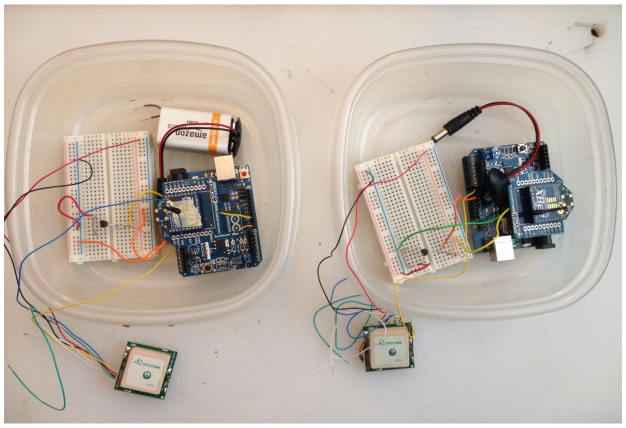


Figure 25. End Device prototype hardware

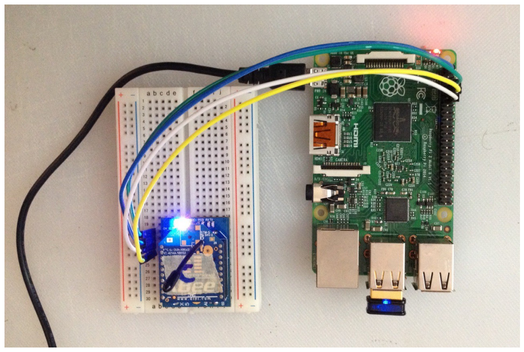


Figure 26. Controller node prototype hardware

Initial prototype was two ZigBee End Point Sensor Nodes and one ZigBee Controller star network.

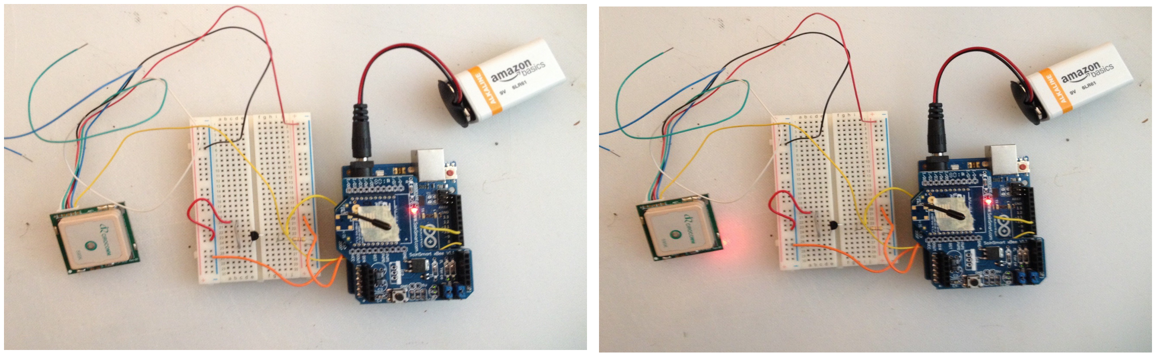
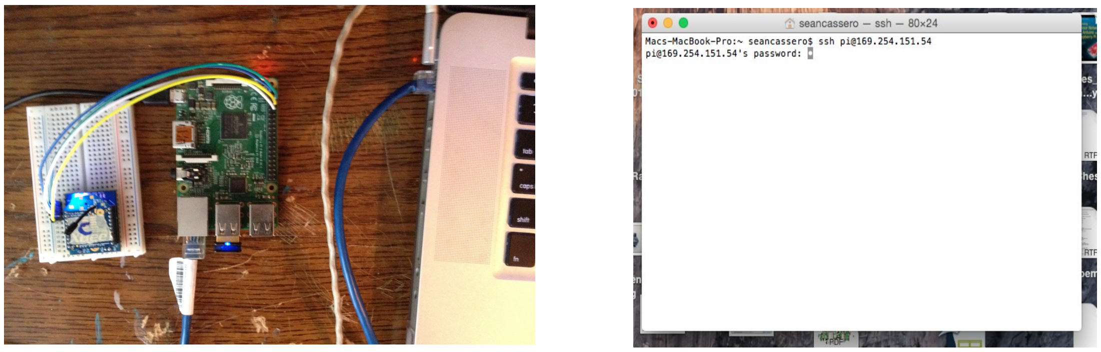


Figure 27. GPS of Sensor Node Flashing Off and On Before Lock

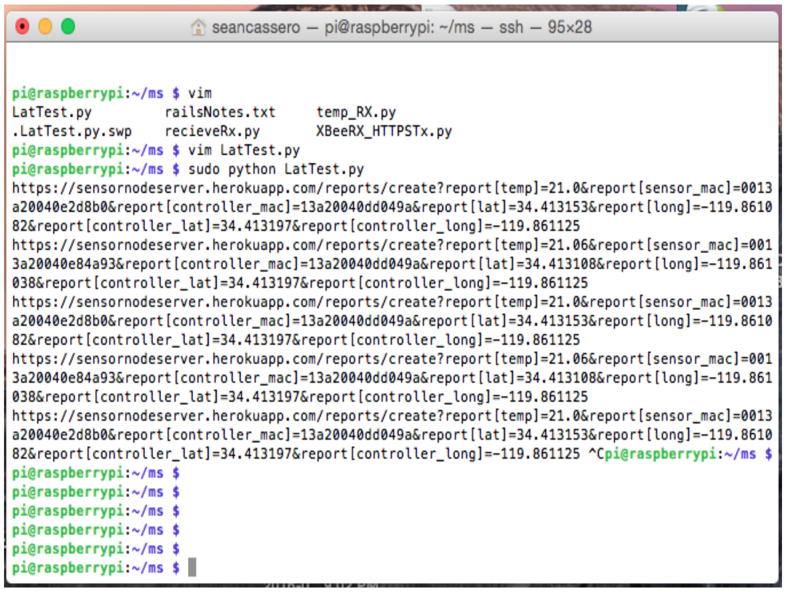
GPS sensors must lock on satellite signal before transmission is allowed to begin. Red LED on the GPS device flashes on and off while sensor is locking and switches to solid once a lock has been established. Cold start time ranges from 1-10min for indoor sensors.



```
Macs-MacBook-Pro:~ seancassero$ ssh pi@169.254.151.54
pi@169.254.151.54's password:
```

Figure 28. Headless Raspberry Pi ssh connection via Laptop (left) SSH terminal window (right)

Although previous development allow Raspberry Pi to function as Coordinator/ Cloud Gateway without user initialization, parameters may still be adjusted and scripts run via ssh over direct ethernet connection.



```
pi@raspberrypi:~/ms $ vim
LatTest.py          railsNotes.txt      temp_RX.py
.LatTest.py.swp     recieveRx.py        XBeeRX_HTTPSTx.py
pi@raspberrypi:~/ms $ vim LatTest.py
pi@raspberrypi:~/ms $ sudo python LatTest.py
https://sensornodeserver.herokuapp.com/reports/create?report[temp]=21.0&report[sensor_mac]=0013
a20040e2d8b0&report[controller_mac]=13a20040dd049a&report[lat]=34.413153&report[long]=-119.8610
82&report[controller_lat]=34.413197&report[controller_long]=-119.861125
https://sensornodeserver.herokuapp.com/reports/create?report[temp]=21.06&report[sensor_mac]=001
3a20040e84a93&report[controller_mac]=13a20040dd049a&report[lat]=34.413108&report[long]=-119.861
038&report[controller_lat]=34.413197&report[controller_long]=-119.861125
https://sensornodeserver.herokuapp.com/reports/create?report[temp]=21.0&report[sensor_mac]=0013
a20040e2d8b0&report[controller_mac]=13a20040dd049a&report[lat]=34.413153&report[long]=-119.8610
82&report[controller_lat]=34.413197&report[controller_long]=-119.861125
https://sensornodeserver.herokuapp.com/reports/create?report[temp]=21.06&report[sensor_mac]=001
3a20040e84a93&report[controller_mac]=13a20040dd049a&report[lat]=34.413108&report[long]=-119.861
038&report[controller_lat]=34.413197&report[controller_long]=-119.861125
https://sensornodeserver.herokuapp.com/reports/create?report[temp]=21.0&report[sensor_mac]=0013
a20040e2d8b0&report[controller_mac]=13a20040dd049a&report[lat]=34.413153&report[long]=-119.8610
82&report[controller_lat]=34.413197&report[controller_long]=-119.861125 ^Cpi@raspberrypi:~/ms $
pi@raspberrypi:~/ms $
pi@raspberrypi:~/ms $
pi@raspberrypi:~/ms $
pi@raspberrypi:~/ms $
pi@raspberrypi:~/ms $
pi@raspberrypi:~/ms $
```

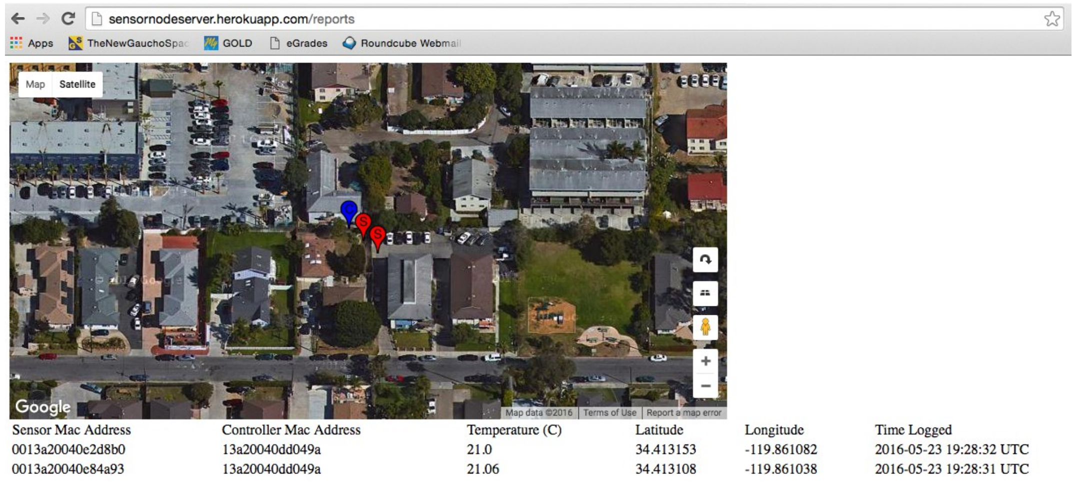
Figure 29. Terminal window showing received transmissions. *Python* script prints URLs to console before sending.

Notice Sensor MAC address parameters switch between the two transmitting sensors while Controller MAC stays the same. In this development iteration GPS strings are received with each End Device transmission, but variation in GPS signal is nonexistent for stationary sensors. Since both prototype End Devices are placed in the same room they read virtually identical temperature values.

B. Cloud Database Server

The *Rails* server successfully receives get requests for data storage and uses *Gmaps4Rails* to compile data to user friendly interface. The Raspberry Pi runs a *Python* script to send data as URL parameters to the cloud server.

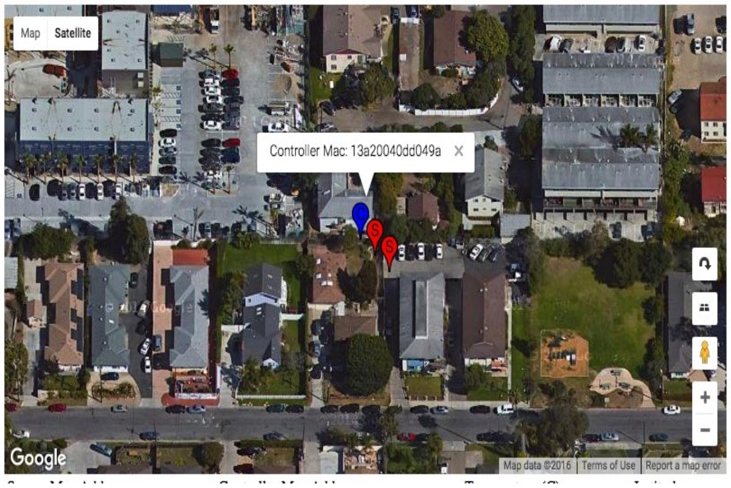
The Web Application URL was adjusted to `sensornodeserver.heroku.com`. New data is sent to [URL]/reports/create for the `reports_controller` to direct the Model to update the database.



Sensor Mac Address	Controller Mac Address	Temperature (C)	Latitude	Longitude	Time Logged
0013a20040e2d8b0	13a20040dd049a	21.0	34.413153	-119.861082	2016-05-23 19:28:32 UTC
0013a20040e84a93	13a20040dd049a	21.06	34.413108	-119.861038	2016-05-23 19:28:31 UTC

Figure 30. Heroku Server GUI interface for tEnd Device and one Coordinator star network

The initial network was tested at a local residence, all sensors transmitting inside a single room. The UI shows inconsistencies in recorded GPS data, likely due to GPS signal reflections when receiving indoors. The table beneath the map shows the corresponding data for each sensor, which is identical to the URL parameters contained in the sample URLs printed to console in figure above. The markers are clickable by the user to display relevant information of the corresponding End Device or Coordinator. The prototype infobox displays the latest temperature and MAC address for End Devices. Coordinator markers only display MAC address since they are not responsible for logging temperature values.



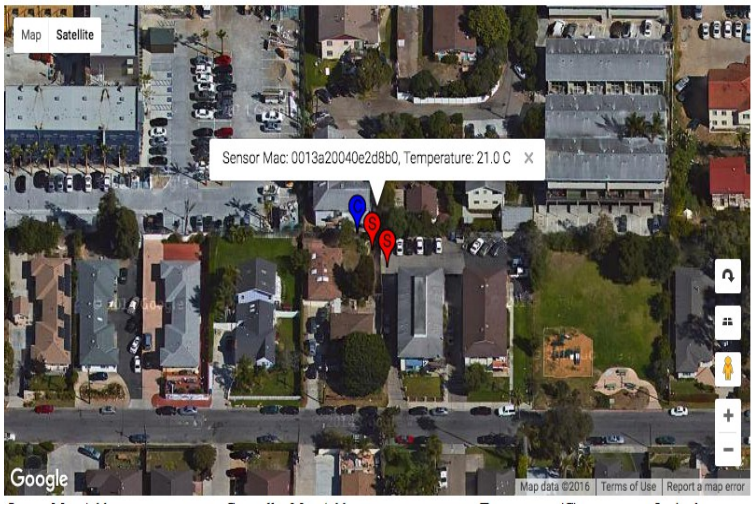
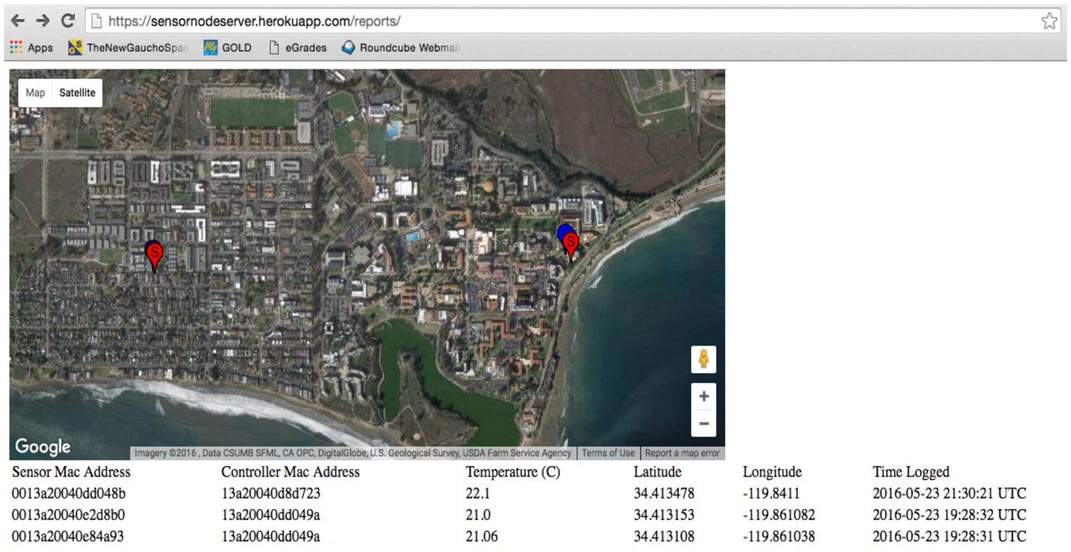


Figure 31. Marker Infobox for Coordinator (top) and End Device (bottom)

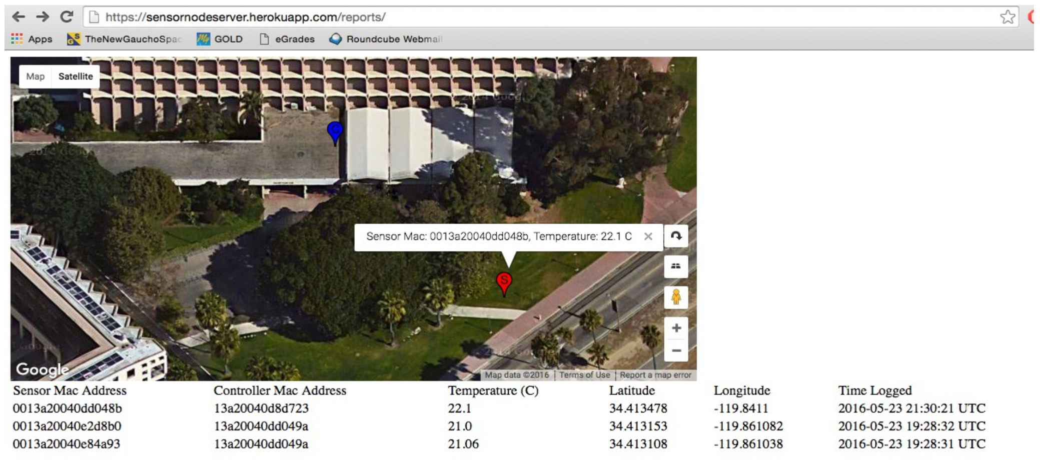
In order to show the cloud server as a shared database function, a second ZigBee network is simulated consisted of a single End Device and Coordinator. The second network is placed near UCSBs Harold Frank Hall, a few miles away from the first prototype network. The inclusion of Coordinator and Sensor MAC addresses as URL parameters will handle the uniqueness of each report. *Gmaps4Rails* handles map scaling and marker generation.


https://sensornodeserver.herokuapp.com/reports/
Apps
TheNewGauchoSpa
GOLD
eGrades
Roundcube Webmail
Map
Satellite
Google
Imagery ©2016 , Data CSUMB SFML, CA OPC, DigitalGlobe, U.S. Geological Survey, USDA Farm Service Agency
Terms of Use
Report a map error

Sensor Mac Address	Controller Mac Address	Temperature (C)	Latitude	Longitude	Time Logged
0013a20040dd048b	13a20040d8d723	22.1	34.413478	-119.8411	2016-05-23 21:30:21 UTC
0013a20040e2d8b0	13a20040dd049a	21.0	34.413153	-119.861082	2016-05-23 19:28:32 UTC
0013a20040e84a93	13a20040dd049a	21.06	34.413108	-119.861038	2016-05-23 19:28:31 UTC

Figure 32. Map rendering on `sensornodeserver.heroku.com/reports` with two ZigBee star networks reporting to a single Web Application

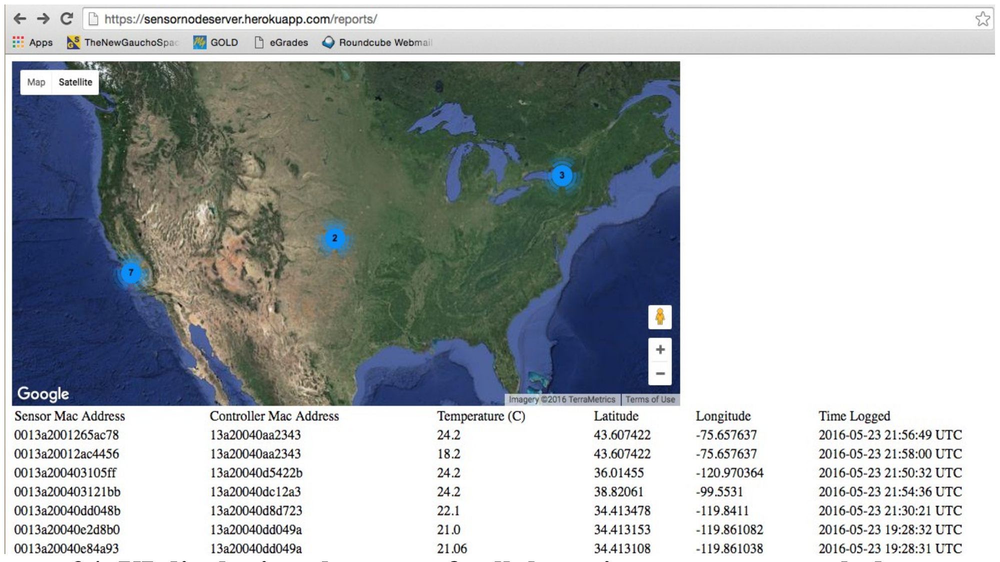
Initially, the *Gmaps4Rails* function in the cloud server handles map scaling. A user can adjust the map scaling using the clickable buttons on the bottom right to closely examine areas of interest. In the figure below, a user has zoomed in to a single ZigBee network.



Sensor Mac Address	Controller Mac Address	Temperature (C)	Latitude	Longitude	Time Logged
0013a20040dd048b	13a20040d8d723	22.1	34.413478	-119.8411	2016-05-23 21:30:21 UTC
0013a20040e2d8b0	13a20040dd049a	21.0	34.413153	-119.861082	2016-05-23 19:28:32 UTC
0013a20040e84a93	13a20040dd049a	21.06	34.413108	-119.861038	2016-05-23 19:28:31 UTC

Figure 33. A specific network can be highlighted for analysis of behaviors of interest. Zoomed view of the second ZigBee star network is rendered on the above map.

A third simulation highlights the possibility of larger numbers of complete ZigBee networks with varying topologies storing data on a shared *Rails* server. Below is an image of the *Rails* server generating the UI for a collaborative multi-network sensor scheme.



Sensor Mac Address	Controller Mac Address	Temperature (C)	Latitude	Longitude	Time Logged
0013a2001265ac78	13a20040aa2343	24.2	43.607422	-75.657637	2016-05-23 21:56:49 UTC
0013a20012ac4456	13a20040aa2343	18.2	43.607422	-75.657637	2016-05-23 21:58:00 UTC
0013a200403105ff	13a20040d5422b	24.2	36.01455	-120.970364	2016-05-23 21:50:32 UTC
0013a200403121bb	13a20040dc12a3	24.2	38.82061	-99.5531	2016-05-23 21:54:36 UTC
0013a20040dd048b	13a20040d8d723	22.1	34.413478	-119.8411	2016-05-23 21:30:21 UTC
0013a20040e2d8b0	13a20040dd049a	21.0	34.413153	-119.861082	2016-05-23 19:28:32 UTC
0013a20040e84a93	13a20040dd049a	21.06	34.413108	-119.861038	2016-05-23 19:28:31 UTC

Figure 34. UI displaying the map of collaborative sensor network data.

In the figure above, various sensor networks are sending temperature data to the *Rails* server. The blue markers indicate the number of End Device and/or Coordinator markers located within their circle. The table beneath the map displays the latest parameters for each unique End Device.

C. End Device Power Analysis

Below are the results of the power analysis trials. Trials ranged from 8-360 second End Device sleep cycle duration.

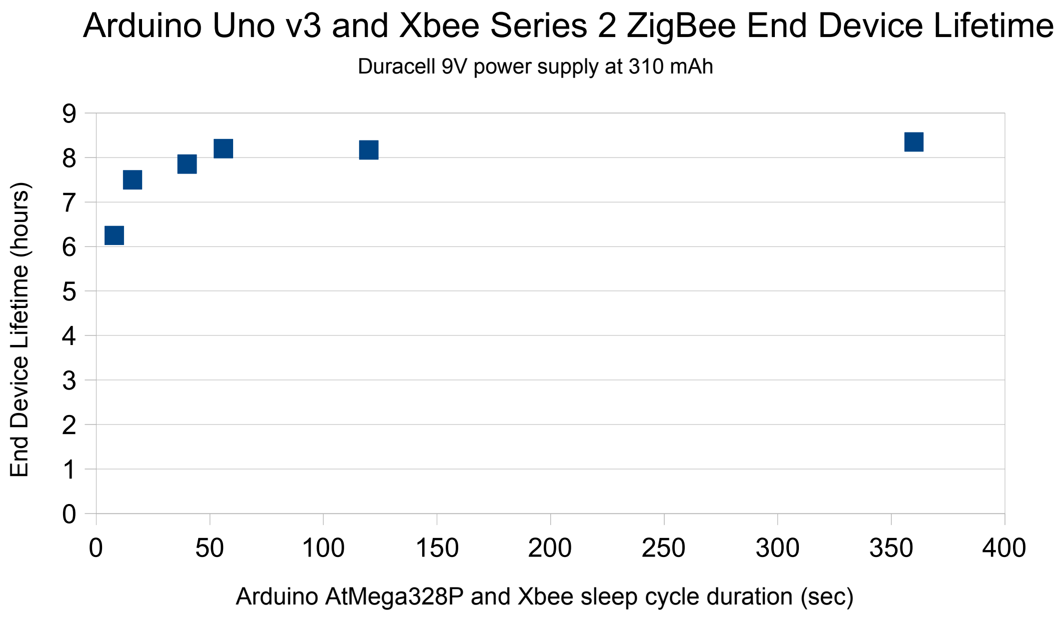


Figure 35. Graphical results of End Device power analysis

The End Device lifetime increases initially due to the reduction of current to the Atmega328P processor and Xbee device during the sleep cycle phase, but reaches a saturation point due to the constant current draw $i_{idle}=Q_{idle}/T_{tot}$ of the Arduino Uno hardware components listed in the Power Analysis section. Further increase in End Device efficiency must be through modification to the Arduino Uno hardware.

Analysis on End Device average current consumption independent of the Arduino Uno hardware is relevant to future designs. Using the formulation derived in the power analysis subsection, or

$Q_{battery}=(i_{sleep})*t_{sleep\,cycle}*n_{loops}+(i_{awake})*t_{Tx}*n_{loops}+(i_{idle}*T_{tot})+Q_{startup}$, an experimental value for the optimal End Device hardware power consumption as a function of End Device sleep cycle duration can be derived.

Begin by updating the above equation with known parameters.

From observation, the data transmission portion of the End Device cycle takes approximately 2.6s to complete, resulting in $t_{Tx}=2.6\text{s}$.

Q_{sleep} boils down to three things, the Xbee sleeping current consumption (~.01mA from the Digi datasheet), the AtMega328P `PWR_DWN` sleep current consumption (~.0039mA from the Atmega datasheet), and the 5V to 3.3V Sparkfun voltage regulator supplying the Xbee DTR pin with 3.3V (measured at ~ 2.3mA). Thus Plugging in values gives

$Q_{sleep}=(.0039\,mA+.01\,mA+2.3\text{mA})*t_{sleep}=2.3139\text{mA}*t_{sleep}$.

A single 8 pack of Duracel 9V Alkaline batteries was used to test the End Device lifetimes. Due to the relatively high discharge rate for a standard 9V battery, a 310 mAh charge is assumed, resulting in $Q_{battery}=310\,mAh=1116000\,mAs$.

The $t_{sleep\,cycle}$ and n_{loops} variables change for each trial, below is a table summarizing the values

sleep interval (s)	node lifetime (h)	node lifetime (s)	Tx time (s)	n loops	t sleep (s)	t awake (s)
8	6.25	22500	2.6	2112.6	16900.8	5492.76
16	7.5	27000	2.6	1451.6	23225.6	3774.16
40	7.85	28260	2.6	663.4	26536	1724.84
56	8.2	29520	2.6	503.8	28212.8	1309.88
120	8.17	29412	2.6	239.9	28788	623.74
360	8.35	30060	2.6	82.9	29844	215.54
360	8.35	30060	2.6	82.9	29844	215.54

$t_{startup}$ proved difficult to accurately measure due to a high variability in GPS locking (1-10 min), so $Q_{startup}$ was omitted from the analysis. But since $Q_{startup} \ll Q_{idle}+Q_{Tx}+Q_{sleep}$, $Q_{battery} \approx Q_{idle}+Q_{Tx}+Q_{sleep}$ is valid to within a small degree of error.

Using the data in the table above, a series of equations can be generated relating the variables of interest. The expression for End Device current consumption is organized as known parameters on the LHS and unknown parameters on the RHS , giving

$$Q_{battery}-(i_{sleep}*t_{sleep\,cycle}*n_{loops})=(i_{awake}*t_{Tx}*n_{loops})+(i_{idle}*T_{tot})$$

Plugging in the results from the table above gives 6 equations with 2 unknowns (i_{awake} , and i_{idle}).

1. $1076724\ mAs = (i_{awake} * 5492.76\text{s}) + (i_{idle} * 22500\text{s})$
2. $1062281\ mAs = (i_{awake} * 3774.16\text{s}) + (i_{idle} * 27000\text{s})$
3. $1046973\ mAs = (i_{awake} * 1724.84\text{s}) + (i_{idle} * 28260\text{s})$
4. $1054627\ mAs = (i_{awake} * 1309.88\text{s}) + (i_{idle} * 29520\text{s})$
5. $1050753\ mAs = (i_{awake} * 623.74\text{s}) + (i_{idle} * 29412\text{s})$
6. $1049416\ mAs = (i_{awake} * 215.54\text{s}) + (i_{idle} * 30060\text{s})$

Using the above equations, multiple regression analysis is performed to get a least-squares estimate for the unknown parameters.

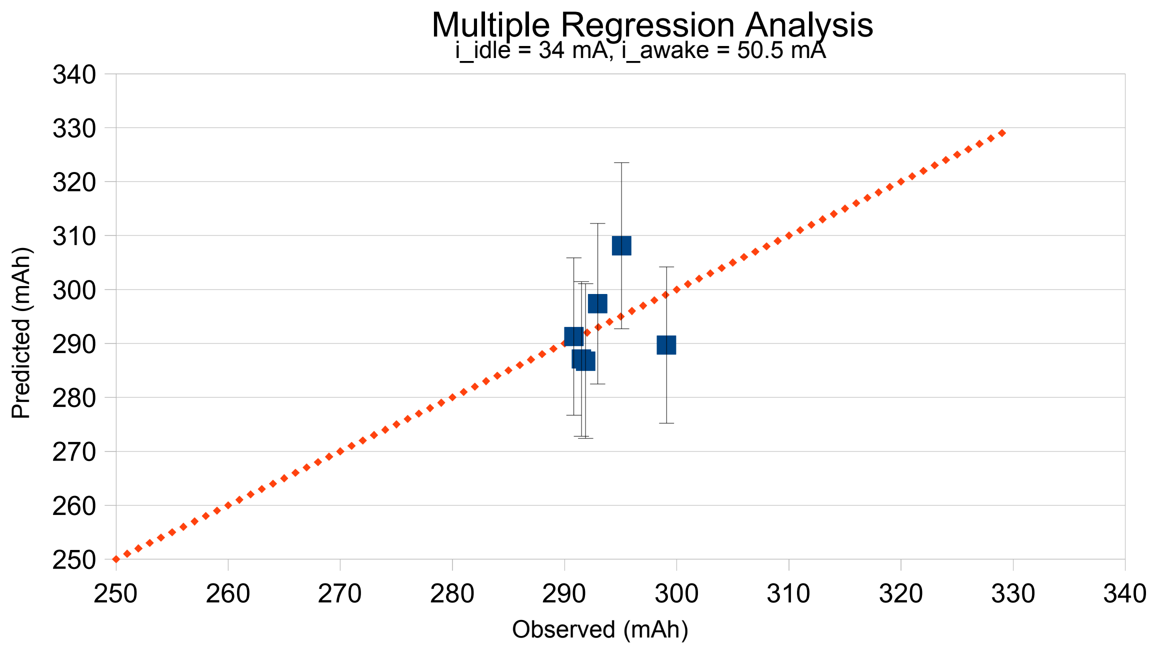


Figure 36. Predicted vs observed mAh results of multiple regression analysis

The regression analysis resulted in vales of $[i_{idle}, i_{awake}] = [34.02\, mA, 50.5\, mA]$. The above graph is the predicted vs observed values for $Q_{battery} - (i_{sleep} * t_{sleep\,cycle} * n_{loops})$ given $[i_{idle}, i_{awake}] = [34.02\, mA, 50.5\, mA]$, superimposed over the line y = x. All predictions fall within 5% error of the observed values (shown by the error margins in the plot above). Due to the potential variability in $Q_{battery}$ from trail to trial, and the absence of $Q_{startup}$ due to difficult measurement, a 5% error window is considered statistically significant.

From the above results, a formula for the End Device average current during transmission as a function of sleep cycle duration can be written as follows

$$I_{avg} = \frac{\left(t_{sleep} * i_{sleep} + 2.6\text{s} * i_{awake}\right)}{\left(t_{sleep} + 2.6\text{s}\right)} + i_{idle} = \frac{\left(2.3\,mA * t_{sleep} + 131.3\,mC\right)}{\left(t_{sleep} + 2.6\text{s}\right)} + 34.0\text{mA}$$

Without the Arduino Uno hardware drawing idle current during the End Device runtime, the average current can be reduced. In order to get an estimate of the average current for an End Device consisting of solely an AtMega328P and Xbee device, the above equation can be modified by removing i_{idle} and modifying i_{awake} , i_{sleep} , and t_{Tx} to reflect potential optimization or use cases. For this project, the One Wire temperature sensor draws 4mA during runtime, and requires a nontrivial amount of time (~1s) to complete a reading. Also, the Sparkfun breadboard power supply to provide the Xbee DTR pin with 3.3V during sleep cycle is not necessary for hardware platforms using an AtMega processor at 3.3V. Converting the AtMega to 3.3V and removing the Sparkfun supply would greatly decrease i_{sleep} .

VII. Conclusion

Xbee series 2 devices successfully implement star network with minimal initial configuration. API mode with escaping provides reliable data transmission scheme with integrated parameters of interest in packet formation. Integration of Xbee communicators with existing microprocessor and Linux computer platforms offers streamlined prototyping and smooth data transmission.

Although implementation was successful, hardware flaws on stock Arduino Uno make power consumption suboptimal. LP2985-N linear voltage regulator has maximum efficiency of .71% with Arduino recommended voltage input range. Incorporating an Arduino as End Device hardware must manually bypass the stock linear regulator to achieve desired battery longevity depending on future project requirements. Simple wireless networks requiring sensor data with minimal signal processing might consider alternate hardware platforms[1]. Arduino does serve as a sufficient prototyping module to get initial idea up and running with minimal development time due to simple hardware integration and wide user community providing sensor library support. The wide selection of Arduino plug and play sensors with existing library integration make prototyping fast and simple.

The development of an Environmental monitoring sensor networks using pressure, humidity, temperature, and GPS data for sensor node data transmission is simply a matter of editing Arduino hardware and software configuration for a particular application. This project is similarly suited for other remote sensing/controlling applications such as smart lighting, smart homes, or smart cars. However, in such cases the minimal processing

[1]Alternative hardware platforms are discussed in the “Future Work” section of this thesis

capabilities of Xbee devices are potentially sufficient for a standalone sensor node. The addition of an Arduino in simple wireless sensor network schemes is most likely unnecessary. Arduino may have a place as a data-aggregate node for large scale personal area ZigBee networks with tree or mesh topology where additional computational power is required for routers to maintain network functionality. However, considerations into the inefficient nature of the Arduino linear voltage regulator must be taken into account for the intended low-energy ZigBee network to function properly in such a design structure.

Raspberry Pi stock hardware provides an adequate platform for Coordinator node with minimal software initialization. Pi's limited onboard memory and processing capabilities are addressed with a cloud database server for remote data storage. Python as an interpreted language offers no noticeable significant latencies for small scale networks with low frequency data transmission rates.

Since the Coordinator node requires wifi connectivity for cloud communication, it is assumed it is in range of a wall power source. Therefore minimizing power consumption of Raspberry Pi Coordinator node is unimportant for analysis purposes since it is assumed the Raspberry Pi has a dedicated power source. The project uses a 2A 5V wall wart to provide sufficient power for the Coordinator node.

The cloud server is an adequate framework for a simple cloud database. The implementation of a single Controller View setup is sufficient for a basic user interface and database management. Server allows for multiple complete ZigBee sensor networks to collaborate on a shared database for geographically separate network configurations.

Gmaps4Rails handles map sizing for proper zooming and map marker implementation creating an easily readable and interactive user interface.

The complete prototype wireless sensor network is fit for scalable network sizes and a variety of potential applications. The remote sensor nodes use ZigBee protocols to automatically register unique sensor MAC address with network Coordinator upon sensor node initialization, allowing for modular sized networks.

Additionally, the cloud database structure offers scalability to various project sizes with demand-specific processing based off incoming traffic. The cloud database also addresses the minimal local storage capacity on Raspberry Pi Coordinator node by outsourcing data storage to a more capable structure.

The shared database structure is useful for large scale geographically separate sensor networks such as monitoring collective health of the California Redwood forests as, monitoring radiation hotspots on coastal region after Nuclear plant meltdown leading to ocean contamination, coastal health monitoring, nation wide temperature measurement systems for climate evolution, or other large scale networks not able to transmit data solely via network repeaters to a local database.

The implemented structure of Arduino for additional local processing capabilities in ZigBee star network topology is potentially useful in networks where ZigBee End Devices require higher processing than simple wireless sensor networks, such as a mobile robotics network. While wireless mobile robotics networks generally require high data rate transmission frequencies for variables such as PID parameters or video and speech processing, the ZigBee networks low frequency transmission scheme may be useful for sensor data transmission that is not as frequent, such as GPS data on a relatively stationary unit. The ZigBee low power transmission scheme may still be relevant in high power systems such as robotics if there is a desire for a periodic low power transmission scheme functioning alongside a high frequency parameter transmission, or as a failsafe communication system in low power settings.

End Device power consumption was tested by recording End Device lifetimes over a series of data transmission frequencies. A duracel 9V battery powered the End Device hardware until exhaustion and the total transmission time for an End device was recorded. Analysis of the End Device hardware components was used to derive a mathematical relationship between battery charge and average current draws and duration of the main loop functions of an End Device given by $Q_{battery}=(i_{sleep})*t_{sleep\,cycle}*n_{loops}+(i_{awake})*t_{Tx}*n_{loops}+(i_{idle}*T_{tot})+Q_{startup}$. The End Device lifetimes were used to generate values for n_{loops} for each of the trials run. Known current draws and direct measurements on operating End Devices were used to eliminate all variables in the above equation except i_{idle} and i_{awake} , resulting in 6 equations with two unknowns. Multiple regression analysis provided least-squares estimates for the above parameters of $[i_{idle}, i_{awake}]=[34.02\,mA, 50.5\,mA]$, coming within 5% of predicting the observed values for all trials. Using the experimental results, an equation for the average current draw of the Arduino/Xbee End Device is given by

$$I_{avg}=\frac{\left(t_{sleep}*i_{sleep}+2.6\text{s}*i_{awake}\right)}{\left(t_{sleep}+2.6\text{s}\right)}+i_{idle}=\frac{\left(2.3\,mA*t_{sleep}+131.3\,mC\right)}{\left(t_{sleep}+2.6\text{s}\right)}+34.0\text{mA}$$

where t_{sleep} is the End Device sleep cycle duration.

VIII. Future Work

End Device hardware can be improved in a number of ways. First, an efficient switching regulator in place of the Arduino stock linear regulator. The LCP1117 linear voltage regulator has an efficiency of $V_i/V_o*100=55.55$ % at 9v. This is inefficient, and since the voltage regulator runs continuously the regulator efficiency places an upper bound on the overall node efficiency. Similarly, the LP2985-N linear regulator is not optimal for this project since the Xbee draws relatively large current while operating (~27 mA) and is powered by the Arduino 3.3V source. End Device power consumption can be decreased by bypassing the integrated voltage regulators for a more efficient switching regulator. Future designs should consist of a single switching voltage regulator to convert input voltage to 3.3V.

While the AtMega16U2 makes updating the Arduino code simple, a more efficient End Device hardware design should place the USB to serial conversion off the main board to limit idle current draw. The switching regulator should power a single AtMega processor running at 3.3V to handle signal processing in tandem with the Xbee device for communication. Elimination of the 5V to 3.3V drop between the Xbee and the AtMega will eliminate the need for a voltage regulator powering the Xbee DTR pin during the sleep cycle, increasing the End Device lifetime by lowering i_{sleep}. A t_{Tx} value of 2.6s is unreasonably high for most applications. The high t_{Tx} was mostly for experimental purposes, such as using an LED to signify successful transmission and using a One Wire temperature sensor operating on a ~1s/reading signaling scheme. Reduction of the t_{Tx} parameter can be achieved with AtMega main loop code optimization and high speed sensor selection, decreasing average End Device current as well.

Renewable energy generation for outdoor sensor nodes such as solar power can be feasible to greatly extend node lifetime without human interaction. A DC power solar panel can periodically charge the End Device battery while resources are available. Automatic registration with the Coordinator node will allow an End Device to comply with the potentially sporadic power supply of direct sunlight by automatically resuming data transmission when adequate power requirements are met.

If a network size grows considerably larger, the relatively high inherent latency of *Python* implemented in the Coordinator node may become problematic for timely data transmission. Developing the Raspberry Pi Coordinator node and cloud gateway script in compiled language such as *C* will improve code efficiency of the Coordinator node.

Similarly, as network size increases, the cloud server framework may need to change to address increased traffic. *SQLite* database is sufficient for small prototype development, but heavier alternatives much as *MySQL* or *Mongo.db* may be necessary to handle database structure after a certain network node number threshold is crossed.

This project focused on information transfer from End Device to cloud database, further development could focus on reliable bidirectional communication. There is potential usefulness in modifying certain End Device parameters such as sleep rate via a browser request to the cloud server. Developers could update the *Rails* controller to send a request to the Coordinator node when the browser toggles a certain interface in the *Rail*s View. Each Coordinator has a unique MAC address and IP, the Model could update the `Reports` object in the *SQLite* database with an incoming IP address. The *Rails* request would contain the Sensor MAC address, parameter to be changed, and would be addressed to the IP of the corresponding Coordinator MAC address. Part of the *Python* loop in the Raspberry Pi would check for incoming messages from the *Rails* server and, if a new transmission is received, parse the Sensor MAC and parameter to change from that message. ZigBee API mode allows for wireless node configuration with correct the API packet transmission, Raspberry Pi would construct the necessary API message and transmit to Sensor MAC address. Modular sleep rates for End Devices could be useful in tailoring sensor network to report areas of interest more frequently. Game theoretic AI in controller could calculate optimal transmission frequency of End Devices based on variability in received parameters; more variability means more frequent transmission.

Alternative communication APIs such as *MQTT* might be more efficient for specific use cases. Potential investigation into efficiencies and benefits of other communication methodologies is potentially relevant.

Investigation into security risks of Xbee series 2 ZigBee network is relevant for networks with sensitive information. Transporting data as URL parameters also has potential security risks if the Raspberry Pi communicates via http not https. A security key might be useful for admitting new reports.

References

1. C. Y. Chong and S. P. Kumar, “Sensor networks: Evolution, opportunities, and challenges,” Proceedings of the IEEE, vol. 91, pp. 1247- 1256, August 2003.

2. T. Arampatzis, J. Lygeros, and S. Manesis,"A Survey of Applications of Wireless Sensors and Wireless Sensor Networks," IEEE International Symposium on Intelligent Control, Mediterrean Conference on Control and Automation, pp.719,724, 27-29 June 2005.

3. J. S. Lee, “Performance evaluation of IEEE 802.15.4 for low-rate wireless personal area networks,” IEEE Trans. Consum. Electron., vol. 52, no. 3, pp. 742–749, Aug. 2006.

4. N. S. A. Zulkifli, F. K. C. Harun, and N. S. Azahar, "XBee wireless sensor networks for Heart Rate Monitoring in sport training," in Biomedical Engineering (ICoBE), 2012 International Conference on, 2012, pp. 441-444.

5. X. Liu, H. Chen, M. Wang, and S. Chen, “An xbee-pro based energy monitoring system,” in Telecommunication Networks and Applications Conference (ATNAC), 2012 Australasian. IEEE, 2012, pp. 1–6.

6. Ghariani, N., Chaoui, M., Ghariani, H., Lahiani, M., “Design of a digital communication system based on a XBee module for biomedical applications”, 8th International Multi-Conference on Systems, Signals and Devices, pp. 1-5, 2011.

7. C. Evans-Pughe, “Bzzzz [ZigBee wireless standard],” IEE Review, vol. 49, issue 3, pp. 28-31, March 2003.

8. Milan Matijevic, and Vladamir Cvjetkovic, “Overview of architectures with Arduino boards as building blocks for data acquisition and control systems” in Remote Engineering and Virtual Instrumentation (REV), 2016 International Conference on, pp. 56-63, 2016.

9. Syed Zahurul, Norman Mariun, Leong Kah, and Hashim Hizam, “A novel Zigbee-based data acquisition system for distributed photovoltaic generation in smart grid” in Smart Grid Tehnologies – Asia (ISGT ASIA), 2015 IEEE Innovative, pp. 1-6, 2015.

10. Mohd Faris, and Mohd Fuzi, “HOME FADS: A dedicated fire alert detection system using ZigBee wireless network” in Control and System Graduate Research Colloquium (ICSGRC), 2014 IEEE 5th, pp. 53-58, 2014.

11. Charles Bell "Beginning Sensor Networks with Arduino and Raspberry Pi" New York: Springer Science+Business Media, 2013.
12. Nikhil Agrawal, Noida Siemens, and Smita Singhal "Smart drip irrigation system using raspberry pi and arduino" in Computing, Communication & Automation (ICCCA), 2015 International Conference on, pp. 928-932, 2015.

13. Sudhir G. Nikhade, "Wireless sensor network system using Raspberry Pi and zigbee for environmental monitoring applications" in Smart Technologies and Management for Computing, Controls, Energy, and Materials (ICSTM), 2015 International Conference on, pp 376-381, 2015.

14. G. V. Vivek, and M. P. Sunil, "Enabling IOT services using WIFI-ZigBee gateway for a home automation system" in 2015 IEEE International Conference on Research in Computational Intellegence and Communication Networks (ICRCICN), pp 77-80, 2015.

15. Ferreira, H. G. C.; Canedo, E. D.; Sousa Júnior, R. T. (2013). IoT Architecture to Enable Intercommunication Through REST API and UPnP Using IP, ZigBee and Arduino. 2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), pp.53,60. doi: 10.1109/WiMOB.2013.6673340.

Appendix A

Routes.rb file

```
Rails.application.routes.draw do

 get '/reports/', to: "reports#index", as: "reports"
 get 'reports/create'
 get 'reports/update'
 get 'reports/destroy'
end
```

Database Schema

Schema.rb

```
ActiveRecord::Schema.define(version: 20160429014601) do

 create_table "reports", force: :cascade do |t|
  t.float  "temp"
  t.float  "lat"
  t.float  "long"
  t.string  "sensor_mac"
  t.string  "controller_mac"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.float  "controller_long"
  t.float  "controller_lat"
 end

 create_table "sensors", force: :cascade do |t|
  t.string "mac"
 end
```

Model Initialization

sensor.rb

```
class Sensor < ActiveRecord::Base
 has_many :reports
end
```

report.rb

```
class Report < ActiveRecord::Base
 has_one :sensor
end
```

Database Migration

```
class CreateReports < ActiveRecord::Migration
 def change
  create_table :reports do |t|
   t.integer :sensor_id
   t.float :temp
   t.float :lat
   t.float :long
  end
 end
end
```

Rails View Source Code

index.html.erb

```
<script src="//maps.google.com/maps/api/js?
v=3.23&sensor=false&client=&key=&libraries=geometry&lang
uage=&hl=&region="></script>
<script
src="//cdn.rawgit.com/mahnunchik/markerclustererplus/mas
ter/dist/markerclusterer.min.js"></script>

<div style='width: 800px;'>
 <div id="map" style='width: 800px; height: 400px;'></div>
</div>

<script type="text/javascript">
 handler = Gmaps.build('Google');
 var mapOptions = { mapTypeId:
google.maps.MapTypeId.SATELLITE };
 handler.buildMap({ provider: mapOptions, internal: {id:
'map'}}, function(){
  markers = handler.addMarkers(<%=raw @hash.to_json %>);

  handler.bounds.extendWith(markers);
  handler.fitMapToBounds();

 });
</script>

<table style="width:100%">
 <tbody>
  <tr>
   <td>Sensor Mac Address</td>
   <td>Controller Mac Address </td>
   <td>Temperature (C)</td>
   <td>Latitude</td>
   <td>Longitude</td>
   <td>Time Logged</td>

  </tr>
  <% @reports.each do |report| %>
  <tr>
   <td><%= report.sensor_mac %></td>
   <td><%= report.controller_mac %></td>
   <td><%= report.temp %></td>
   <td><%= report.lat %></td>
   <td><%= report.long %></td>
   <td><%= report.updated_at %></td>
  </tr>
  <% end %>
 </tbody>
</table>
```

Rails Controller Code

reports_controller.rb

```
class ReportsController < ApplicationController
 protect_from_forgery :except => :create

 def index
  @reports = Report.pluck(:sensor_mac).uniq.sort.map { |mac|
Report.where(sensor_mac: mac).last }
```

```
  @hash = Gmaps4rails.build_markers(@reports) do |report,
marker|
   marker.lat report.lat
   marker.lng report.long
   desc = "Sensor Mac: #{report.sensor_mac}, Temperature:
#{report.temp} C"
   marker.infowindow desc
   marker.picture({
    :url => "http://chart.apis.google.com/chart?
chst=d_map_pin_letter&chld=S|FF0000|000000",
    :width  => 32,
    :height => 32
   })


 end




  @controllers =
Report.pluck(:controller_mac).uniq.sort.map {|mac|
Report.where(controller_mac: mac).last }


   @hashed = Gmaps4rails.build_markers(@controllers) do |
controller, marker|
    marker.lat controller.controller_lat
    marker.lng controller.controller_long
    desc = "Controller Mac: #{controller.controller_mac} "
    marker.infowindow desc
    marker.picture({
     :url => "http://chart.apis.google.com/chart?
chst=d_map_pin_letter&chld=C|0000FF|000000",
     :width => 32,
     :height  => 32
    })


  end
  @hash += @hashed

 end



 def create
  logger.debug report_params
   @report = Report.new report_params
  @report.save
  render nothing: true
 end



 private

  def report_params
   params.require(:report).permit(:temp, :lat, :long,
   :sensor_mac, :controller_mac, :sleep, :voltage,
:controller_lat, :controller_long)
  end
end
```

Appendix B

List of Github Repositories

Arduino One Wire Github. Website: https://github.com/PaulStoffregen/OneWire

Python requests Github. Website: http://docs.python-requests.org/en/master/

Python serial Github. Website: https://github.com/pyserial/pyserial

Python Xbee Github. Website: https://github.com/nioinnovation/python-xbee

Arduino-Xbee Github. Website: https://github.com/andrewrapp/xbee-arduino

Arduino TinyGPS Github. Website: http://arduiniana.org/libraries/tinygps/

Soft Serial Github. Website: https://github.com/arduino/Arduino/tree/master/hardware/arduino/avr/libraries/SoftwareSerial

Arduino Dallas Temperature Github. Website: https://github.com/milesburton/Arduino-Temperature-Control-Library

Ruby Gmaps4rails Github. Website: https://github.com/apneadiving/Google-Maps-for-Rails