Cognitive Radio using Radio Resource Management

A Major Qualifying Project

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the degree requirements for the

Degree in Bachelor of Science

in

Electrical and Computer Engineering

by

____

Michael Ghizzoni

____

Mathew Kelley

____

Conor Rochford

Project Number: AW-002

Date: 10/23/09

Sponsoring Organization:

University of Limerick:

Project Advisors:

____

Professor Alexander Wyglinski, Advisor

____

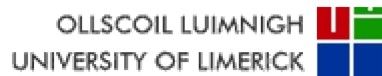
Professor Richard Vaz, Advisor

Abstract

This report presents a cognitive radio network that optimizes routing to preserve battery capacity while maintaining an acceptable signal quality. This is implemented by collecting data on the current link quality and battery charges of the nodes in the network and by performing a routing algorithm to optimize the signal quality of the links and the battery life of the nodes. The network performs the entire optimization process successfully 87.5% of 40 test trials.

Acknowledgements

Without the help of many individuals and resources available to us this project would not have been possible. There were many people who both assisted with any problems or questions that we had and there were others who made sure that we had appropriate accommodations and all of the resources that we need.



Thank you to the University of Limerick for sponsoring our project, providing us with office space, and all of the hardware that we needed. Special thanks to Sean McGrath for overseeing the project and acting as our primary faculty advisor. Thank you to Michael Barry for answering all of our questions and providing daily guidance and direction to our project.

Thank you to Digi International, specifically the live chat tech support staff for all of the hardware specific questions and concerns that we came across. This made working with the XBee chips run significantly smoother.

Thank you to Richard Hoptroff with Flexipanel ltd for all of his support and help with the UZBee dongles.

A thank you to Bob Recny, a Senior Product Engineer at FTDI Chip, for his help with the UZBee Dongles regarding the UZBee+ Virtual COM Port drivers.

Of course, thank you to Worcester Polytechnic Institute, specifically the Interdisciplinary and Global Studies Division for making all of the necessary arrangements for the project and project center. Special thanks go out to Professor Alexander Wyglinski and Professor Richard Vaz for advising the projects and lending appropriate guidance and encouragement during the weekly teleconferences and throughout the project.

Finally, thank you to Charlotte Tuohy for being our liaison for the project center. She secured living arrangements for us and made sure that we had everything we needed upon arrival and throughout the experience.

Executive Summary

Cognitive radio systems, which acquire hardware and network conditions through sensors, transmissions and user inputs to use them as feedback to control radio operation, has become increasingly important as more institutions, companies and homes adopt wireless technologies. Medical institutions in particular, which have adopted wireless networks for patient monitoring, emergency paging and records transfer without wires, require these cognitive systems. Pagers and "motes", which are medical instrumentation with embedded RF modules, need to be routed to maintain the necessary link quality while consciously preserving battery charge of these battery-powered nodes to maximize their duration of operation.

The proposed solution of this problem is to implement an ad hoc network with a conscious coordinator that collects each links bit error rate (BER) and received signal strength indication (RSSI) and each nodes battery charge and performs a routing algorithm to promote battery charge preservation without compromising signal quality. The coordinator also derives the appropriate transmission power for each node that will accommodate the quality of service (QoS), but also minimize interference and current draw. After careful analysis and experimentation of the capabilities of a variety of RF modules, this network was implemented using Xbee Series 2 ZB RF modules with application programming interface (API) firmware which comprised the ZigBee stack.

The coordinator of the network was programmed to initiate and synchronize the feedback cycle that is used to update the routes and power levels of the routers in the network with a node discovery request. When the routers receive this transmission, all the nodes sequentially perform a node discovery to find and store all the addresses of the nodes within single hop range. Each node then sends two link assessment requests to each node found during node discovery at two different power levels. At the same time, the nodes receive link assessment requests and use them to derive BER and RSSI of each single hop link at the two different transmission powers. Each router then sends its link quality data along with its battery charge to the coordinator, which performs the routing algorithm and transmit power level assignment.

Two routing algorithms were implemented that use two different methods to optimize the routing by meeting the application's QoS while considering the battery life of each node. The first algorithm is called the transmission power limiting algorithm. This algorithm calculates link costs by adding weighted terms dependent on BER, RSSI, transmit power level, and battery charge. The routing

algorithm will then select the multi-hop routes from the routers to the coordinator and back with the lowest maximum link cost.

The path limiting algorithm considers the increase in current draw for every route a node is an intermittent hop in and minimizes the burden of relaying packets for the nodes with the least charge. This is accomplished by first passing every found link through a BER screen. The links that meet this QoS are then used to find all paths that can be used to circumvent nodes with low battery charge. Dijkstra's algorithm is performed using solely RSSI to create initial paths and then the battery life is calculated for each node. The nodes with the least battery life are then circumvented by using the alternative routes around it.

The coordinator then assigns the minimum transmit power levels to each node that accommodate to the QoS requirements of all the links they are required to use for the assigned routes. The coordinator then sets its own routes and sends the assigned routes and power levels to their respective router. The routers then adjust their route to the coordinator. The entire cycle is repeated shortly after to keep up with real time environment and hardware changes.

When the XBee Series 2 ZB RF modules arrived in week five, the network protocol that initiated node discovery, performed node discovery, assessed link quality, transmitted cognitive data to the coordinator, and sent source routes was coded and debugged simultaneously. Many timing and settings adjustments were made to decrease the packet loss rate, node discovery inconsistency and unintended router resets in the next five weeks. With all these adjustments, the unintended router resets were decreased to 10%, the packet loss rate was decreased to 6.7%, and the node discovery consistency was brought up to 94%. These values are based on 40 documented tests of the entire cycle.

The routing algorithms that were programmed for the coordinator operated as expected 100% when tested using simulations and when integrated into the coordinator's code. The transmission power limiting algorithm correctly calculates the quality of routes based on the cognitive parameters received and routes accordingly. The path limiting algorithm maximizes the nodes' battery lives giving priority to the nodes with the least battery charge.

Table of Contents

Abstract II

Acknowledgements III

Executive Summary V

List of Figures XIII

List of Tables XVII

1. Introduction 1

2. Background 4

2.1. Cognitive Radio 4

2.1.1. The Original Concept of Cognitive Radio 5

2.1.2. Current Cognitive Radio Research 6

2.2. Cognitive Parameters 6

2.2.1. Received Signal Strength Indicator (RSSI) 7

2.2.2. Signal to Interference and Noise Ratio (SINR) 7

2.2.3. Error Detection 9

2.2.4. Spectrum Sensing 11

2.3. Radio Resource Management (RRM) Techniques 16

2.3.1. Transmission Power 16

2.3.2. Admission Control 17

2.3.3. Forward Error Correction (FEC) 20

2.3.4. Automatic Repeat Request (ARQ) 21

2.4. Existing Protocols 23

2.4.1. WiFi Protocol (802.11) 23

2.4.2. Bluetooth Protocol (802.15.1) 24

2.4.3. ZigBee Protocol (802.15.4) 24

2.4.4. Comparison of Wireless Technologies 26

2.5. Multi-hopping Methods for ZigBee 26

2.5.1. Ad Hoc On-Demand Distance Vector Routing 27
2.5.2. Broadcasting 27
2.5.3. Many-to-One 27
2.5.4. Source Routing 27
2.6. Routing Algorithms 28
2.6.1. Dijkstra's Algorithm 28
2.6.2. A* Routing 32
2.7. Chapter Summary 32
3. Project Goals and Design Decisions 33
3.1. Cognitive Radio for Medical Devices 33
3.2. Product Requirements 34
3.2.1. Reliability 34
3.2.2. Portability 34
3.2.3. Energy Efficiency 34
3.2.4. Data Rate 34
3.3. Project Management and Tasks 35
3.4. Optimization Process 36
3.5. Cognitive Parameters 38
3.6. Radio Resource Management 38
3.7. Hardware 38
3.7.1. UZBee Dongles 39
3.7.2. XBee Development Boards 44
3.7.3. Conclusions 49
3.8. Firmware 49
3.8.1. Transparent Operation Firmware 50
3.8.2. Data Parsing in Transparent Operation 50
3.8.3. Altering Transmission Power in Transparent Operation 54

3.8.4. Multi-Hopping with Transparent Firmware ... 55
3.8.5. API Operation ... 56
3.8.6. Conclusions ... 58
3.9. Chapter Summary ... 58
4. Prototype Implementation ... 59
4.1. Node Discovery Request ... 59
4.1.1. Node Discovery ... 60
4.2. Link Assessment ... 61
4.2.1. BER Check ... 63
4.2.2. RSSI Extraction ... 64
4.2.3. Cognitive Data Packet ... 65
4.3. Routing Algorithm ... 66
4.3.1. Transmission Power Limiting Algorithm ... 66
4.3.2. Path Limiting Algorithm ... 75
4.4. Source Routes ... 90
4.4.1. Setting the Coordinator Source Routes ... 90
4.4.2. Setting the Source Route and Power Level of the Routers ... 91
4.5. Chapter Summary ... 92
5. Testing and Verification ... 93
5.1. Node Discovery ... 93
5.1.1. Node Discovery for the Coordinator ... 93
5.1.2. Node Discovery for Routers ... 95
5.2. Cognitive Parameter Acquisition ... 97
5.2.1. Cognitive Parameter Acquisition for the Coordinator ... 97
5.2.2. Cognitive Parameter Acquisition for Routers ... 98
5.3. Routing Algorithm ... 99
5.3.1. Transmission Power Limiting Routing Algorithm ... 102

5.3.2. Path Limiting Routing Algorithm 104

5.3.3. Parsing and Setting Source Routes in the Router 107

5.4. Chapter Summary 108

6. Conclusions 109

7. Future Work 112

7.1. Battery Monitoring 112

7.2. Transmission Power in Path Limiting Algorithm 113

7.3. Cognitive Radio in other Protocols 114

8. References 115

A. Appendix A 117

Transparent Mode Methods 117

setupXbee 117

findRssi 117

powerLevel 118

sendSensorData 118

Transparent Mode Setup and Loop 119

Router 119

API Mode Router Methods 121

findNetwork 121

receiveData 121

modeOne 123

modeTwo 123

modeThree 124

nodeDiscovery 124

powerLevel 125

arrayLength 126

apiTx 126

createRoute.. 127

sendDiagnostic.. 127

rssiStart 128

findRssi... 128

berCheck 129

clearArrays 130

API Mode Router Setup and Loop... 130

API Mode Coordinator Methods.. 131

open_port 131

arrayLength 131

apiTx... 132

findRssi... 132

receiveData 133

nodeDiscovery.. 135

powerLevel... 136

sendDiagnostic.. 136

berCheck 137

storeData.. 137

clearData 137

routeMe 138

getQ.. 139

setPower 141

createRoute.. 141

sendRoutes... 142

createLinks 143

coordNeighbors.. 143

findPaths 144

convertPaths 145

initialRoutes 150

battMaximize 151

setPower 154

routingAlgoritm 155

API Mode Coordinator Main 155

Transmission Power Limiting Routing Algorithm 155

Path Limiting Routing Algorithm 157

B. Appendix B 159

List of Figures

Figure 1: Cognition Cycle of Mitola's Radio (from [1]) ... 4
Figure 2: The signal to noise and interference ratio is the quotient of the original signal divided by the effects of the channel interference plus the additive noise ... 8
Figure 3: This example is of an eight bit two redundancy scheme ... 10
Figure 4: This is an example of a parity scheme where the eighth bit is an odd parity bit ... 10
Figure 5: In (a), a signal in the frequency domain is depicted as a narrow band spike. In (b), the noise of a channel is randomly distributed power across a wide frequency range. (c) is the received signal which is the noise plus the original signal. ... 12
Figure 6: Power Spectral Density Block Diagram (from [5]) ... 13
Figure 7: Implementation of an Energy Detector (from [6]) ... 13
Figure 8: Spectral Correlation Block Diagram (from [5,5]) ... 15
Figure 9: Implementation of Cyclostationary Feature Detector (from [5]) ... 15
Figure 10: MAC Beacon Payload in ZigBee (from [9]) ... 19
Figure 11: Sample ZigBee Network before Shifting ... 19
Figure 12: Sample ZigBee Network after Shifting ... 20
Figure 13: Plot of Target Signal to Interference and Noise Ratio as a Function of ARQs ... 22
Figure 14: Wifi Network with three devices connected to a single router ... 23
Figure 15: ZigBee Mesh Network ... 25
Figure 16: Dijkstra Routing: Set Node Costs to Infinity ... 29
Figure 17: Dijkstra Routing: Calculate Nearby Node Costs ... 29
Figure 18: Dijkstra Routing: Change Node and Calculate Costs Again ... 30
Figure 19: Dijkstra Routing: Change Paths when Lower Cost is Found ... 31
Figure 20: Dijkstra Routing: Algorithm Finished when Destination is Reached ... 31
Figure 21: Initial Project Gantt Chart ... 35
Figure 22: Final Project Gantt Chart ... 36
Figure 23: This is the network optimization flow that is performed cyclically to keep up with real time changes ... 37
Figure 24: This is a 802.15.4 UZBee + dongle (from [16]). ... 39
Figure 25: This is an XBee Series 2 ZB RF Module. ... 44
Figure 26: The coordinator transmitter (a) and end device receiver (b) performing a successful selective transmission is witnessed in the virtual COM port. ... 47

Figure 27: The first hop end device (b) was able to successfully parse the address from the destination payload sent by the coordinator transmitter (a). The second hop end device (c) did not receive it. 48
Figure 28: This is a typical header to communicate identification, power transmission level, battery power and received signal strength. 51
Figure 29: This the test packet sent out by the end device in order for the router to parse, append its own data to and re-transmit. 52
Figure 30: The packet from the end device was parsed and measured for RSSI. The router then appended its own cognitive data to the back of the packet and retransmitted it. 53
Figure 31: Node 35 (a) and node 52 (b) send out a RSSI requests to each other. Both nodes reply with the measured RSSI values of the request packet. Both nodes receive this data and parse out the reading. .. 54
Figure 32: Node 35 (a) and node 52 (b) find receive the RSSI of their transmitted RSSI request packet. With this information, they both adjust their power level down one step because the RSSI received was above the upper boundary. 55
Figure 33: This is an API transmit request frame. 56
Figure 34: This API frame adjusts the power to power level 1 or -2dBm. 57
Figure 35: This is an API frame that sets a route through node EEFF to node 3344. 58
Figure 36: The coordinator broadcasts a node discovery request to initiate the cognitive parameter generation and feedback 59
Figure 37: Each node performs a node discovery to find all the feasible links the coordinator can use while performing the routing algorithm 61
Figure 38: Each node's node discovery is space out by 8 seconds to ensure that all the feasible single hops are found. 61
Figure 39: Each node sends out a link assessment request to all the nodes on its node table. 62
Figure 40: Each node sends 25 ASCII G's for a BER link test, its ASCII node ID, the transmit power level and the mode 2 request in each link assessment request payload. 62
Figure 41: Each node sends its power one link assessment requests in the first 20 seconds and its power 3 link assessment requests in the next 20 seconds. 63
Figure 42: The cognitive data is sent back with a comprehensive list of all the parsed link assessment requests it received. 65
Figure 43: The routers send the cognitive data collected on the links in the link assessment and the node's personal battery level and ID to the coordinator to perform the routing algorithm. 66
Figure 44: Powel Level Decision Based on Two Tests 69
Figure 45: Routing Algorithm Step One 73

Figure 46: Routing Algorithm Step Two 73
Figure 47: Routing Algorithm Step Three 74
Figure 48: Routing Algorithm Step Four 74
Figure 49: The path finding algorithm never circuitously routes two nodes (b) that can be linked directly (a). 76
Figure 50: R6 circumvents R2 and R3 in case their charge is too low to relay R6 packets. 77
Figure 51: The node neighbors R2, R3 and R7 of the coordinator are found. 77
Figure 52: The path extends from the coordinator to the first router ordered numerically which is R2 in this case. 78
Figure 53: The first router finds its own node neighbors R4, R5 and R8 while disqualifying R3 and R7 who are the coordinator's neighbors. 78
Figure 54: R4 cannot find any node neighbors that aren't already neighbors of R2. 79
Figure 55: The path returns to R2 which still has R5 and R8 to explore and a new path is created that extends to R5, the next lowest router on R2's neighbor list. 79
Figure 56: The paths out from the coordinator are converted to paths in from the routers. 80
Figure 57: The power level for each link is selected by comparing the link assessment packets received RSSI against the threshold power boundaries -80dBm to -60dBm. 81
Figure 58: The RSSI found from link assessment packets at power level one and three are used to derive required link power and RSSi path cost. 82
Figure 59: R2, R3 and R7, the neighbors of the coordinator, are routed directly to the coordinator. 82
Figure 60: R4 chooses the route 1 because it has the smallest total path cost. 83
Figure 61: The initial source routes are selected based on RSSi path weights. The number of routes pass through each node is recorded. 84
Figure 62: The battery capacities and the number of routes each node is a part of is used to calculated battery life and the order the nodes are checked. 85
Figure 63: The first alternative path, route 2, from R4 is compared with the original route, route 1. The battery life of R2 is increased from 9.7 hours to 10.7 hours, so the route becomes permanent. 86
Figure 64: The limiting node R2 and R6, the node with the second shortest battery life, are unaffected by this route change. The route is still made permanent because the node with the third shortest battery life, R3, increases from 17.6 hours to 19.3 hours. 87
Figure 65: When checking the alternative routes for the nodes that originally went through R3, The limiting node R2 still takes priority. Re-routing R5 through R2 caused its battery life to drop from 11.9 hours to 10.7 hours, therefore the route is discarded. 88

Figure 66: Even though the re-route boosts R3's battery life from 21.4 to 23.9 hours. The route is unchanged because R7's battery life would fall to 20.3 hours which is lower R3's original battery life, 21.4 hours. These are the final route paths that the algorithm produced. 89
Figure 67: This is an example of how the source route is stored before creating the source route. 90
Figure 68: This is an example of the payload when the coordinator sends the source route and power level back to the routers. 91
Figure 69: Node Discovery Error in Coordinator 94
Figure 70: Debugged Node Discovery in Coordinator 94
Figure 71: Node Discovery Error in Router 95
Figure 72: Program Restart Error in Router 95
Figure 73: Debugged Node Discovery in Router 96
Figure 74: RSSI Extraction Error in Coordinator 97
Figure 75: Debugged Cognitive Parameter Acquisition in Coordinator 98
Figure 76: Cognitive Read Error in Router 98
Figure 77: Debugged Cognitive Parameter Acquisition in Router 99
Figure 78: Sample Successful Node Discovery 100
Figure 79: Sample Successful Link Quality Receives 100
Figure 80: Sample Successful Cognitive Parameter Receive 101
Figure 81: Sample Source Route Setting from Router 1 101
Figure 82: BER Test Case for Power Limiting Routing Algorithm 102
Figure 83: RSSI and Battery Level Test Case for Power Limiting Algorithm 104
Figure 84: Routing configuration with all nodes directly connected to the coordinator 105
Figure 85: Routing configuration with R1 routing through R2 106
Figure 86: Routing configuration with R1 routing through R3 and R3 routing through R2 107
Figure 87: Layout of Source Route Data Packet 107
Figure 88: Source routes and power level being set for router 1 108

List of Tables

Table 1: Radio Knowledge Reference Language Frame [1] 5

Table 2: Radio Knowledge Reference Language's Spatial Inference Hierarchy [1] 6

Table 3: Wireless Transmission Power Regulations by Frequency and Country (from [7]) 17

Table 4: WiFi Standards Comparison (from [12] and [13]) 24

Table 5: Pros and Cons for Wireless Technologies 26

Table 6: Pros and Cons of Dijkstra's Algorithm 32

Table 7: Pros and Cons of A* Algorithm 32

Table 8: XBee Power Levels [18] 45

Table 9: Pros and Cons of Hardware Choice 49

Table 10: List of API Frame Types (from [15]) 57

Table 11: PWM Percentages (from [18]) 64

Table 12: Sample Routing Battery Levels Table 67

Table 13: Sample Routing RSSI Table 67

Table 14: Sample Routing BER Table 68

Table 15: Sample Routing Previous Power Levels of Nodes 68

Table 16: Sample Routing Power Level per Link Table 69

Table 17: Sample Routing RSSI Parameter for Q 70

Table 18: Sample Routing BER Parameter for Q 70

Table 19: Sample Routing Energy Cost per Link 71

Table 20: Sample Routing Q Values for all Links 72

Table 21: Sample Test of Path Limiting Routing Algorithm 99

Table 22: Test Values Set for Tx Power Limiting Algorithm 103

Table 23: Transmission Power Limiting Algorithm Tests 159

Table 24: Path Limiting Algorithm Tests 160

1. Introduction

Wireless technologies can greatly improve healthcare devices. The three major benefits that wireless technologies add are increased mobility, lower cost, and improved signal quality. The necessity for a cognitive system to be in place is due to the high demands of these devices. The goal of a cognitive system is to make the network perform flawlessly while being virtually invisible to the user.

For our purposes cognitive radio is defined as an external engine to the MAC layer with the purpose of detecting system performance and providing feedback to the RRM module. The problem in question is to create a network using the principles of cognitive radio with the intention of optimizing the network to a defined quality of service (QoS) standard. For our application this will be the high performance demand of medical instrumentation.

We propose implementing an IEEE 802.15.4 protocol system that can support both star and mesh networks which senses the environment and adapts accordingly. To do this the network must be aware of the quality of all possible links in the system. By reading these cognitive parameters the network will be able to decide how to apply the different applied RRM techniques.

A ZigBee protocol network will be put in place that can assess the signal strength and quality of every connection. The coordinator of the network can then evaluate the signal strength and signal quality of every link, while being power conscious of every device. By assessing these conditions, the coordinator can determine the best multi-hopping path for the device. This route will be determined by the cost that every connection will have on itself and on the network.

By incorporating this style of routing the network will be categorized as an energy spreading network. This theoretically brings the battery power of all nodes down at a flat rate rather than a single device ever being drained faster than the rest. By using energy spreading, the system promotes longer battery life for every device in the network without compromising the received power or signal quality of any transmission. All of these traits translate to a more reliable system.

Cognitive radio is still a fairly new concept. However, there are a few examples of its application today. The main applications of cognitive networks found were combating bit errors, spectrum sensing, energy spreading, and admission control with shifting.

The cognitive parameter of bit-error rate (BER) is commonly used in forward error correction (FEC) and automatic repeat request (ARQ). By detecting the number of bit-errors in a packet, the receiver can decide the number of repetitions to request or the style of error correction necessary to process the

data. These applications are very relevant to the goals of this project as they are radio resource management responses to a cognitive parameter.

Spectrum sensing is a classification of a few different techniques to detect the behavior of other signals within a frequency band. This can include simple techniques like energy detection that simply finds which frequencies currently have a lot of activity. This is done by finding local maxima on the Fourier transform of a received signal. There are more complicated techniques like cyclostationary feature detection which is capable of determining whether interference is simple noise or a modulated signal. These techniques are commonly applied to adaptive filtering and are usually restricted to influences in the physical layer.

Finally, admission control techniques such as energy spreading and shifting are examples of cognitive radio. Energy spreading considers the battery level as a cognitive parameter of all nodes in the system and routes multi-hops away from the nodes that can no longer tolerate the extra power drain. Shifting is a technique where a node is assigned a cognitive parameter. If a node is capable of being routed through multiple paths it is flagged as shiftable. If there is failure in the network or a new node wishes to join the network the shiftable nodes are easily moved to another route to allow for new devices to use the route that it originally used. This is a good technique for avoiding the overpopulation of a particular node.

What makes this project unique is the incorporation of multiple cognitive parameters and RRM techniques to optimize the network. What we have done is measure the RSSI and BER of every possible connection in the network and organized all of this data in a table at the coordinator node. The coordinator then evaluates this data and finds all nodes that meet the BER requirement of the network. It then routes the nodes in such a way to make the energy consumption of the entire network more efficient. The final thing that our network does that is unique is it sets the power level to ensure that the received signal strength is strong enough for a reliable connection but lower such that there will be less interference to other nodes and as a proof of concept can extend the battery life of the node.

This report will first present background information for the reader, including the various cognitive parameters, radio resource management techniques, wireless protocols, multi-hopping methods, and routing algorithms available for implementation.

The report will then describe the need for a cognitive network in the medical field. This chapter will describe the network requirements and quality of service (QoS) conditions that the product should meet.

We will then move on to the design decisions that must be considered. Here it is explained why the cognitive parameters used in the final network were chosen. Similarly, there is a rationalization for the radio resource management techniques that the system uses to improve performance.

An investigation into the hardware used for this project is discussed. The hardware models considered are the UZBee dongle, the XBee Series 1 and the XBee Series 2 ZB. This section will explain the challenges faced with the dongles and the ultimate decision to consider the XBee devices. Next the capabilities of the firmware for the XBee are explored. The selection of the firmware ultimately determines whether the XBee Series 1 or Series 2 ZB is used in the network.

The prototype implementation is then described. A step-by-step walkthrough of the network environment sensing is explained. This begins as a discovery of all neighboring nodes and assessing the link between them. Following this, the coordinator receives all link assessments in the network and determines the best way to route the multi-hops of each node. The coordinator tells each node what path to take in transmission and the power level that it should be operating at.

All testing, verification, and results are presented. In this chapter the reliability and functionality are evaluated. All cases of failure and the frequency of occurrence will be explained. Any cases of false convergence in optimization will be uncovered and an explanation of how it was combated is described.

Finally all conclusions are presented and topics for future work are recommended. This includes recommendations for an appropriate power monitoring system to be incorporated as a measurable cognitive parameter. There will also be recommendations for other routing algorithms and other radio resource management techniques. A continuation of the project from environment sensing to spectrum sensing is also recommended.

2. Background

Before making any important design decisions, it is important to understand the current state of the art. This chapter will explore what cognitive radio is and the different tools that are available for implementation. This will include cognitive parameters, radio resource management techniques, existing protocols, multi-hopping methods, and routing algorithms.

2.1. Cognitive Radio

The cognitive radio, first conceived by Joseph Mitola III and Gerald Maguire Jr., is a software defined radio that has the unique capabilities of observation, learning, orientation, planning, decision and action depicted in Figure 1. Environmental parameters such as radio frequency bands, air interfaces, protocols and spatial and temporal patterns that moderate the radio spectrum can be sensed, interpreted and used as feedback to reprogram the cognitive radio to meet the user requirements. Over time, as it learned the behavioral patterns of the owner including travel, occupational and leisure habits, it adjusts faster to changing environments and communication applications [**1**].

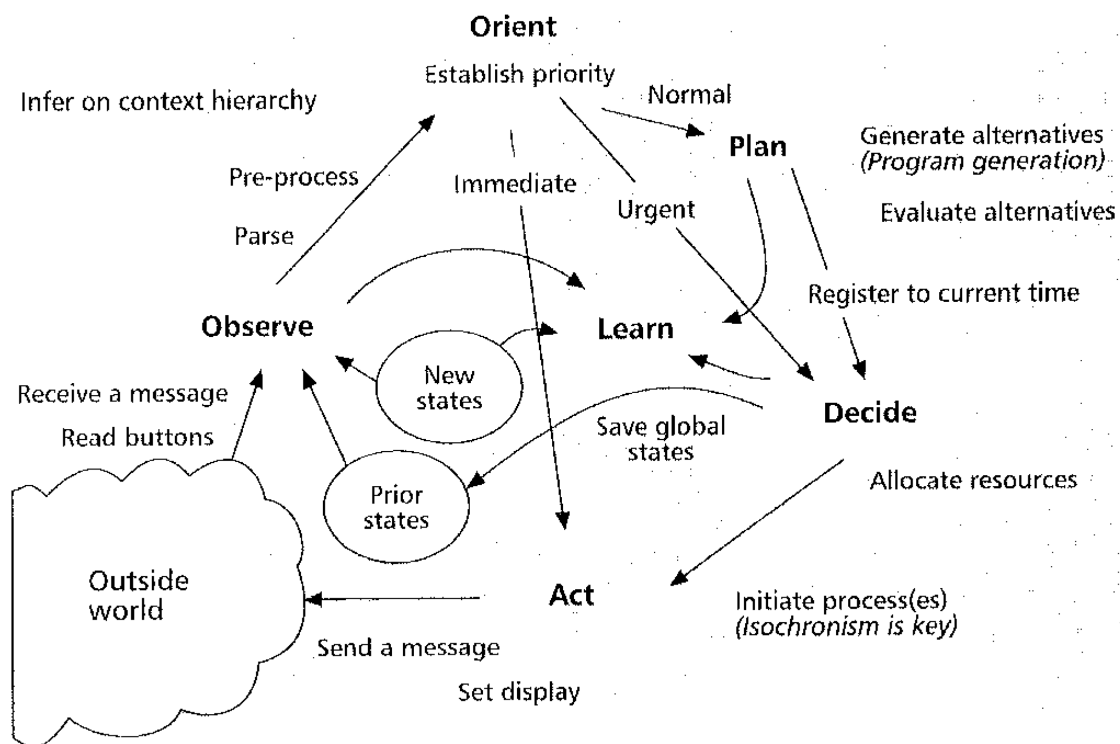


Figure 1: Cognition Cycle of Mitola's Radio (from [1])

2.1.1. The Original Concept of Cognitive Radio

The Radio Knowledge Representative Language gives radios a priori understanding of radio etiquette, internal hardware, software modules, propagation, networks, user needs and application scenarios. The Radio Knowledge Representative Language is a combination of the Specification and Description Language, Unified Modeling Language, Interface Definition Language, Hardware Description Languages, Knowledge Query and Manipulation Languages, and Knowledge Interchange Formatting. Radio Knowledge Representative Language is capable of detailed description of internal hardware and software modules, understanding interface requirements, parsing natural language based messages, expressing conditions, forming plans and performing calculations and adjustments to carry out these plans[**1**].

Consequently, a vast amount of a priori knowledge can be built into the memory of cognitive radios. It is programmed in frames that describe particular aspects of concepts one frame at a time. The handle consists of the object being described within the frame. The body is the information that describes the handle of the frame. The model of the frame represents the relationship between the handle and the body. The context acts as the directory of the frame to help keep the system efficient and organized. An example of a frame can be found in Table 1, which helps describe the world to the radio. This example uses natural language, which and describes South America as being a part of the global plane [**1**].

Table 1: Radio Knowledge Reference Language Frame [1]

Handle	Model	Body	Context
Global plane	Contains	South America	Physical world model/universe/RKRL 0.1

Using these frames, the system is able to parse messages with a level of detail which allows it to learn from and make plans around the content of these messages. These frames describe all relevant information to the radio from its own hardware and software to the world around it. They are even updated from internal and external information sources on a set basis which is defined by the spatial inference hierarchy depicted in Table 2. An example of one of these sources is the sixth plane “Fine Scale” that updates frames, which describe noise, multipath, and interference patterns every microsecond by analyzing the equalizer taps [**1**].

Table 2: Radio Knowledge Reference Language's Spatial Inference Hierarchy [1]

	Plane	Objects	Space	Time	Information Sources
1	Global	Regions	10,000 km	1 year	Travel Itinerary
2	Regional	Cities	1000 km	1 week	Weekly planner
3	Metropolitan	Districts	100 km	1 day	Commuting pattern
4	Local	Buildings	1 km	1 hour	GPS, lunch?
5	Immediate	Rooms	100 m	1s – 1 min	Dead reckoning
6	Fine scale	Body parts	1 m	1µs	Equalizer taps
7	Internal (radio)	HW, SW	.1 m	1ns	Architecture

Since Mitola and Maguire's conception of the cognitive radio is designed to interpret and react to differences in radio transmission standards throughout the world, its design is very comprehensive and it is not suitable for the scope of this project. However, the framework that Mitola and Maguire have laid down can be employed to design more efficient radio resource management techniques and can be applied to wireless medical networks in hospitals if scaled down properly to the planes 5 through 7 [**1**].

2.1.2. Current Cognitive Radio Research

There are two primary types of cognitive radio being researched recently. One type focuses on using parameters, such as received signal strength indicator (RSSI), bit error rate (BER), and signal to interference and noise ratio (SINR), in order to improve signal quality in real time. The other type of cognitive radio focuses on spectrum sensing techniques that allow a wireless device to be aware of the status of the spectrum and adapt its own spectral usage to optimize the network it is on [**2**]. This project will focus on the first type of cognitive radio using a couple cognitive parameters to ensure optimal signal quality.

2.2. Cognitive Parameters

Cognitive parameters are also known as decision variables since their purpose is to provide the information needed to a cognitive algorithm that will optimize the system. The cognitive parameters presented here are RSSI, SINR, and BER.

2.2.1. Received Signal Strength Indicator (RSSI)

The received signal strength indicator is one of the most common cognitive parameters to be measured in wireless systems. The principle concept of RSSI is that the transmitted power is proportionately related to the received power. The received power decreases quadratically with the propagation distance. This can be modeled by [**3**]:

$$P_R = P_T * G_T * G_R * \left(\frac{\lambda}{4\pi}\right) * \left(\frac{1}{d}\right)^n. \quad (1)$$

In this equation, P_R and P_T are the received and transmitted powers respectively. Likewise G_T and G_R are the gains of both the receiver and transmitter antennas. The wavelength is represented with λ. The distance between transmitter and receiver is d. As the distance from the transmitter increases there is a quadratic decrease in signal strength. The n in the system for free space is 2, but can be higher in different medium. It can be seen that the larger the wavelength of the propagating wave the less susceptible it is to path loss. Therefore at higher frequencies, radio waves cannot travel as far with the same transmission power [**3**].

Friis' free space equation is a generalization of the signal strength that will be received [**3**]. However, when interference is considered the signal strength may no longer follow the equation. This interference may be due to multi-path signals or other devices in the frequency band. This interference may be constructive, which will appear to be higher signal strength than the power received from the target. On the other hand, destructive interference will cause the device to read signal strength lower than the equation would predict.

RSSI is a widely used cognitive parameter that is readily accessible on most devices. Though the measured RSSI may not correspond exactly to the power of the desired signal it is a fairly reliable guide to the general performance of the system [**3**].

2.2.2. Signal to Interference and Noise Ratio (SINR)

The signal to interference and noise ratio (SINR) is a crucial cognitive parameter which describes the clarity of the message received by an intended receiver to a transmitter and is correlated with bit error rate (BER) and therefore the quality of service (QoS). A flow diagram which depicts how interference and noise alter the original transmitted signal can be seen in Figure 2.

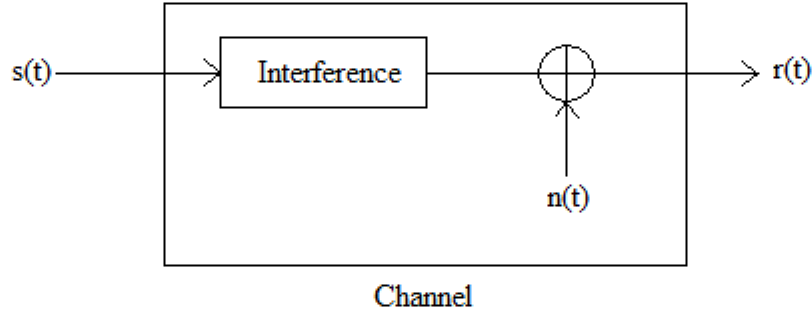


Figure 2: The signal to noise and interference ratio is the quotient of the original signal divided by the effects of the channel interference plus the additive noise

The first step for deriving the SINR is calculating the distances between the transmitting node of each link and the receiver for which the SINR is being calculated. The distance between the nodes can be derived through the relationship of the localization coordinates. From explicitly broadcasted coordinates from the receiving node and every other links' transmitter, along with its own coordinates, the transmitter can calculate the necessary distances using the Pythagorean Theorem in [**4**]:

$$d_{ij} = \sqrt{\left|x_i - x_j\right|^2 + \left|y_i - y_j\right|^2}. \tag{2}$$

The coordinates of the ith node is represented by the set (x_i, y_i). The set (x_j, y_j) describes the coordinates of the jth node. The distance between the ith node and the jth node is represented by d_{mn}.

The distances calculated between the nodes are used to derive the channel gains between each links' transmitters and the nth link's receiver for which SINR is being calculated. The typical channel gain between the mth link's transmitter and the nth link's receiver in an indoor setting can be modeled as [**4**]:

$$g_{mn} = \frac{G_0}{d_{mn}^{\rho}}. \tag{3}$$

G_0 is $2.1023E^{-6}$ and ρ is 3.5 for most non line of sight indoor scenarios. The variable d_{mn}, which is derived in (2), is the distance between the mth link's transmitter and the nth link's receiver.

The channel gain between the nth link's transmitter, which is the node whose transmission is the intended signal, and the nth link's receiver, the node for which SINR is being calculated, must be found to find the received signal strength. In order to find the received signal strength of the node in question, the channel gain of the nth link g_{nn}, the nth link's transmitter's average transmission power $P_{Tx,n}$ and the nth link's transmitter's bit rate $R_{B,n}$ must be known in order to compute [**4**]:

$$P_{Rx,n} = \frac{g_{nn}\, P_{Tx,n}}{R_{B,n}}. \tag{4}$$

Note that the average transmission power and the bit rate are already known by the transmitter.

The gains of the channels between the rest of the transmitters in the network and the nth link's receiver for which SINR is being calculated are used to estimate the multiuser interference. The multiuser interference is estimated using the pulse shape $p_0(t)$, PPM time shift ε, the channel gain g_{mn}, and the transmission power of each transmitter broadcasted explicitly with the coordinates. They are related to the total multiuser interference power by [**4**]:

$$P_{MUI} = \left(\int_{-\infty}^{+\infty}\left[\int_{-\infty}^{+\infty} p_0(t+\varphi)[p_0(t) - p_0(t-\varepsilon)]\, dt\right]^2 d\varphi\right) \sum_{m=1,m\neq n}^{N} g_{mn}\, P_{Tx,m} \; . \tag{5}$$

The noise estimation η_0 can be computed simply by taking the single-sided power spectral density of additive white Gaussian noise n(t). Another method of achieving the interference and noise power is using energy detection prior to calculation. To derive the SINR, the received signal strength $P_{Rx,n}$ has to be divided by the multiuser interference power P_{MUI} plus the noise power η_0. The final equation for SINR γ_i can be found using [**4**]:

$$\gamma_i = \frac{P_{Rx,i}}{P_{MUI} + \eta_0} \; . \tag{6}$$

2.2.3. Error Detection

Error detection is an important cognitive parameter used to decipher how much the incoming transmission is affected by the interference and noise within the environment. Error detection uses different techniques to identify corrupted data and incomplete transmission. Some of the techniques that will be discussed are repetition, parity, and checksum.

By using repetition within the transmission it can be easy to detect corruption in the data. It also provides clues as to what the correct sequence should be. By repeating bits or blocks a predetermined number of times then they can be compared with each other for continuity. If all of the blocks or bits are the same in the sequence it can be determined that an error did not occur. If there is a difference in bits then it is common to quantize the bit to the high mode of the string. Therefore if an odd number of repetitions are sent in a binary decision, whichever of the two choices was transmitted more times is more likely to be the original transmission. This technique is much more effective when the blocks of data repeated are small or are repeated many times. This gives a clearer mode of the data or can make it such that there are less ways for it to fail and therefore makes it easier to determine

which sequence is the correct one. The negative aspect to this technique is that it is very inefficient. By transmitting the same information with much redundancy the overall transmission will be longer and at a given symbol period, it will appear that there is a slower bit rate or a delay. In Figure 3, there is an example of an eight bit to redundancy scheme.

10010101 | 10010101 | 01001000 | 01001000

Figure 3: This example is of an eight bit two redundancy scheme

Parity can be used for a more efficient practice. In this technique a parity bit is assigned for a known number of bits to be determined. The parity scheme can either be even or odd as decided by the programmer. The parity algorithm counts how many ones are in the stream behind it. If the parity scheme is odd then the parity bit will assign a one if there are an odd number of ones in the stream behind it. It will assign a zero for an even number of ones. An even parity scheme would work in the opposite way. This means that there only needs to be one bit every sequence to determine error rather than redundancy of the transmission. This makes the process faster by making the total data shorter. There are several issues with the reliability of this method. For instance, there is a problem if the number of corrupted bits is a multiple of two. This will not change if there is an even or an odd number of ones in the sequence. Therefore, the parity bit will return a "no error" when one has occurred. The other problem is when the parity bit itself is corrupted while the data is correct. This will return an "error" even when one has not occurred. Calling for parity bits more often will reduce the possibility of the first error happening while increasing the possibility of the second error happening. This will be a design decision implemented into any forward error correction that may be used. An example of a parity scheme can be seen in Figure 4. In each byte in the figure, the number of bits set to '1' is made even by setting the last bit, known as the parity bit, to '1' or '0'. If the number of bits set to '1' is not even in each byte when the signal is received, then there has been a bit error in the byte checked.

3 ones +1	2 ones+0	3 ones+1	2 ones+0
10010101	01001000	10010101	01000100
Odd Parity	Even Parity	Odd Parity	Even Parity

Make each byte even by using the parity bit in red

Figure 4: This is an example of a parity scheme where the eighth bit is an odd parity bit

The technique of checksum is used for overall transmission. It assigns a sequence at the end of the transmission that can be decoded to an equivalent of the number of bits that should have been received. This is so that the system knows if it has received the entire sequence or if packets had been dropped. The system counts the received bits and compares its checksum with the equivalent sequence that is generated for the number of bits received. If the checksum and the counted number of bits correspond, the entire sequence is deemed to have been transmitted.

2.2.3.1. Bit Error Rate (BER)

Using the previously discussed techniques, the bit error rate can be determined. This is a ratio of the number of error bits to the number of received bits for a given test time. The test time will be determined by how often the cognitive receiver will want the value to be refreshed. This is an important decision as BER will be an important cognitive value in the system. It will be necessary for knowing how poorly the filtering in the system is working and how much noise is present in the system. The test time can be determined with:

$$t = -\frac{\ln(1-c)}{b*r}, \tag{7}$$

t being the test time, *c* the degree of confidence level desired, *b* the upper bound of the BER, and *r* the data rate.

2.2.4. Spectrum Sensing

Spectrum sensing is where the focus of cognitive radio research is starting to move towards. It is the capability of a device being aware of the frequency domain, or the radio frequency spectrum, of its surroundings. This can detect various forms of interference. It can be as simple as evaluating where there is high energy in the system or as complicated as detecting unknown modulated signals over random noise.

2.2.4.1. Energy Detection

Energy detection is a method of spectrum sensing that attempts to identify the desired signal and the interfering noise. Below is an illustration of distorting affects noise has on a signal. It affects the clarity of a signal received and can jeopardize the data transmission by increasing the bit error rate.

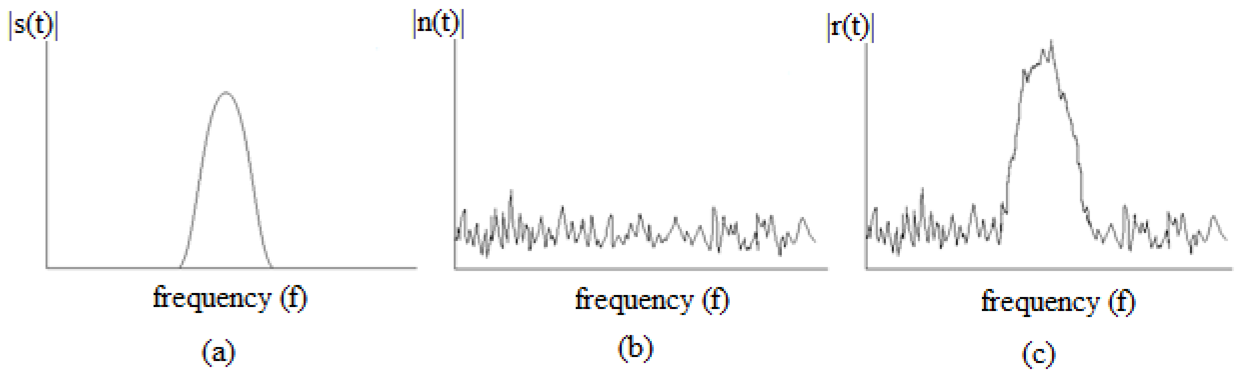


Figure 5: In (a), a signal in the frequency domain is depicted as a narrow band spike. In (b), the noise of a channel is randomly distributed power across a wide frequency range. (c) is the received signal which is the noise plus the original signal.

To detect the energy of the different frequencies of a signal, the autocorrelation of the signal must be found first. Since the signals that are being analyzed in a spectrum sensing scenario are of infinite energy, the autocorrelation function is based on the average. The autocorrelation function is expressed in [**5**]:

$$R_x(\tau) = E[x(t)x(t-\tau)^*]. \tag{8}$$

Since the autocorrelation is a time domain transform the energy of each frequency cannot be analyzed along an axis with this function alone. Through Weiner's relationship, the Fourier transform of the autocorrelation function is the power spectral density, expressed in [**5**]:

$$S_x(f) = \int_{-\infty}^{+\infty} R_x(\tau) e^{-j2\pi f\tau}\, d\tau. \tag{9}$$

The power spectral density is a means of detecting energy across a frequency range.

To measure the power spectral density, the received signal needs to first be put through a band pass filter to localize the signal at a particular frequency that needs to be analyzed. This eliminates all of the noise that is not superimposed onto the signal. Then, the band passed signal needs to be squared in order to analyze the power. The squaring demodulates the signal by multiplying the signal by its own carrier frequency, thereby zeroing the signal and putting a copy at twice the original carrier. A low pass filter will then keep the original signal and eliminate the much of the remaining noise. By demodulating the signal more we isolate it from the noise that is at varying frequencies. Finally, the signal needs to be averaged over a period of time in order to smooth it and increase the signal to noise ratio. The block diagram for measuring the power spectral density is shown in Figure 6 [**5**].

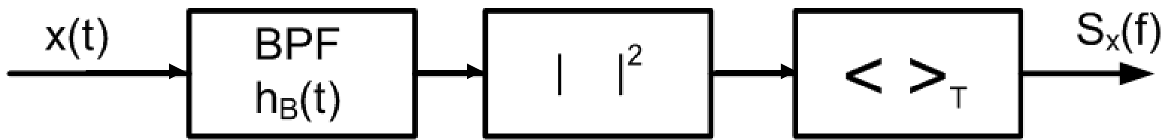


Figure 6: Power Spectral Density Block Diagram (from [5])

To implement an energy detector, the analog received signal first has to be converted to a digital signal such that squaring and averaging may be done discretely. It is then mixed with a threshold and input into a fast Fourier transform (FFT). Subsequently, the signal is averaged over a period of time, T, in order to smooth out the noise. It is then applied as an input to an energy detecting filtering system that adjusts the threshold that the input must meet. The reason the averaging is necessary is that it needs to combat the main drawback of this process: the noise is still superimposed on the desired signal, which increases the threshold that is necessary if it is constructive. By averaging the signal, the constructive and destructive interference theoretically cancel each other out and the energy of the original signal is all that remains. The implementation of the energy detector is depicted in Figure 7 [**6**].

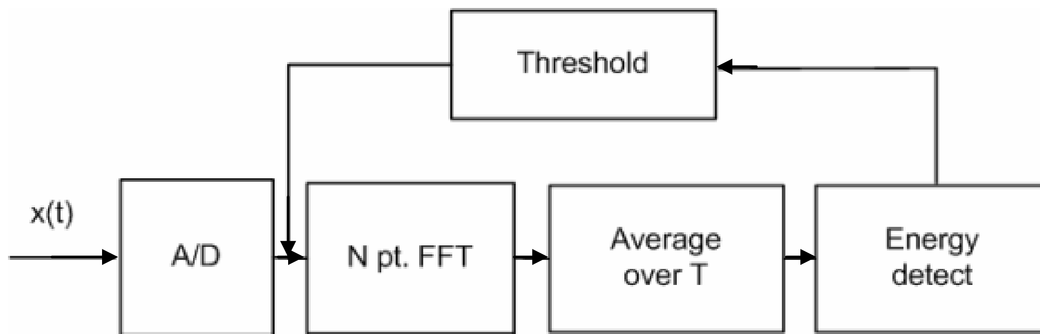


Figure 7: Implementation of an Energy Detector (from [6])

2.2.4.2. Cyclostationary Feature Detection

Cyclostationary feature detection is a method of differentiating primary user signals from noise without prior knowledge of their modulation schemes or protocols. All signals can be modeled as stochastic processes which are probability functions with random variables through time. Stochastic processes are broken up into subcategories based on how random they are. Wide sense stationary processes are stochastic processes with a constant mean such as noise which has zero mean. Cyclostationary processes are stochastic processes with statistical properties that vary cyclically. Modulated signals are cyclostationary processes because they are double sided with sine wave carriers, they have a fixed symbol period, and each modulation type has its own unique cyclostationary features [**5**].

In a more mathematic definition, cyclostationary processes have periodic autocorrelation functions where wide sense stationary signals do not. This means that the autocorrelation of cyclostationary processes can be written as a Fourier coefficient. The Fourier coefficient form of autocorrelation for cyclostationary processes is expressed in [**5**]:

$$R_x^{\alpha}(\tau) = \lim_{T\to\infty} \frac{1}{T}\int_T x\left(t+\frac{\tau}{2}\right) x\left(t-\frac{\tau}{2}\right)^* e^{-j2\pi\alpha t}\, dt. \tag{10}$$

This is called cycle autocorrelation. If the statistically correlated periodic features in a cyclostationary process repeat every T then the cycle autocorrelation has a cycle of α. Since the autocorrelation function is a quadratic transform the features of modulated signals that are functions of symbol rate and carrier can be detected.

The cycle autocorrelation is only a time domain transform. This means it falls short in measuring power spectral density of modulated signals. This is because the system is demodulating many different signals. The maximum energy of all the modulated signals is unlikely to be equivalent. Therefore a threshold would only be set by the highest energy signal. The frequency domain equivalent is called the spectral correlation function, which is expressed in [**5**]:

$$S_x^{\alpha}(f) = \lim_{\Delta t\to\infty} \lim_{T\to\infty} \frac{1}{\Delta t}\frac{1}{T}\int_{-\Delta t/2}^{\Delta t/2} X_T\left(t, f+\frac{\alpha}{2}\right) X_T^*\left(t, f-\frac{\alpha}{2}\right) dt \tag{11}$$

$$X_T(t,f) = \int_{t-T/2}^{t+T/2} x(u) e^{-j2\pi f u}\, du.$$

Through Weiner's relationship, the Fourier transform of cycle autocorrelation is the spectral correlation function. The spectral correlation function is a two dimensional complex transform with to frequency-based axis cycle α and frequency f that is used for feature detection [**5**].

To measure spectral correlation of a function, the received signal can be frequency shifted α Hz and –α Hz in two parallel mixers. This searches many different frequencies for possible carriers. After each frequency shifted signal is passed through identical band pass filters, the signal that was frequency shifted –α Hz is then complex conjugated. After the complex conjugation, the two signals match in carriers of both the real and imaginary plane for attempted demodulation. The two signals are then frequency multiplied back together and averaged over a period of time. If the final signal possess a high energy level, this means that the cycle α corresponds to a carrier frequency of a modulated signal [**5**]. The block diagram is depicted in Figure 8.

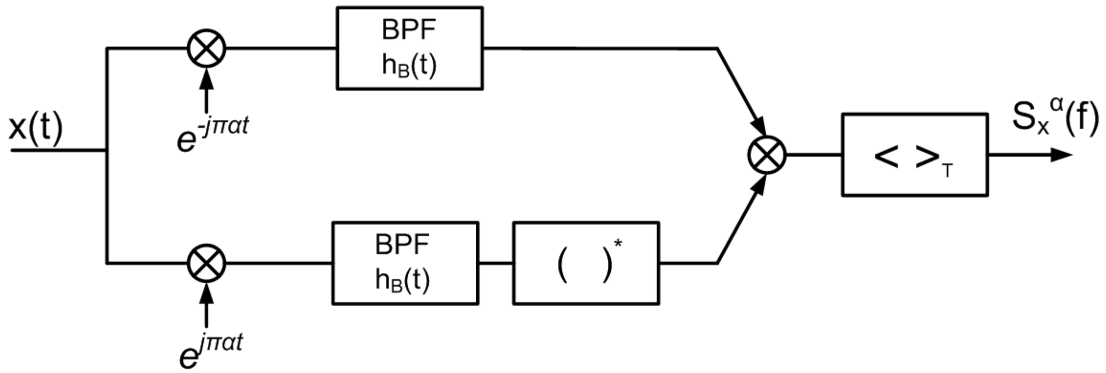


Figure 8: Spectral Correlation Block Diagram (from [5,5])

The frequency shifting, bandpass filtering and the complex conjugation can all be implemented using a fast Fourier transform for any f and α. To implement the cyclostationary feature detector the received analog signal must be converted to digital. The digital signal is then input into an N point FFT. The conjugate outputs are then correlated and averaged over a period T. The feature detection then senses the peaks for α>0 and can even distinguish between different modulation schemes and protocols with simplified matched filtering [**5**]. The implementation is depicted in Figure 9.

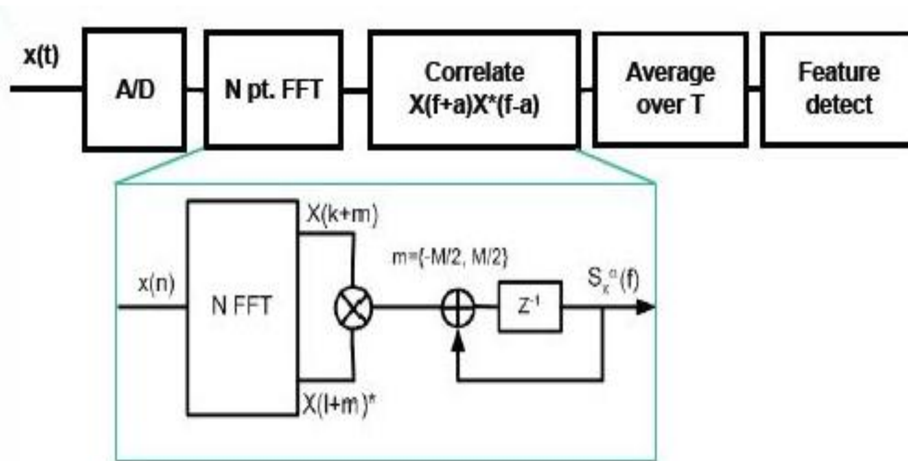


Figure 9: Implementation of Cyclostationary Feature Detector (from [5])

2.2.4.3. Matched Filter

Matched filtering is the optimal method of signal detection since it effectively maximizes received signal to noise ratio. It follows the full cognitive radio model described earlier in that a priori knowledge of the primary user signal at both the physical and medium access control layers is required. This knowledge is the pulse shape that is used for the transmission. The radio has stored memory effectively describing the modulation type, order, pulse shaping and packet format to demodulate the

signal. It is also required to synchronize with the carrier and time scheme as well as perform channel equalization [**6**].

Match filtering takes advantage of the fact that a filter that is the time reverse of the pulse shape used, when convolved with the signal, optimizes the energy of the signal. This is due to the fact that the filter is time reversed again and passed over the system when the convolution occurs. If the filters match, it will be a perfectly constructive superposition. However, a major drawback is the programming of all the different standards into the memory of the radio and dedicating enough receivers to detect all the primary users [**6**]. For the scope of this project, it is feasible to use three matched filters for one standard of Wifi, Bluetooth and ZigBee by integrating three daughter cards with the microcontroller. The medical application makes it feasible to only provide matched filtering for the modulation and protocol standards that occur within a hospital.

2.3.Radio Resource Management (RRM) Techniques

2.3.1. Transmission Power

Transmission power variability is an RRM technique that can promote better signal quality or save on power consumption given its application. According to Equation (1), the received signal strength indicator (RSSI) labeled as received power P_R can be modeled as a direct relationship to the transmitted power:

$$P_R = P_T * G_T * G_R * \left(\frac{\lambda}{4\pi}\right) * \left(\frac{1}{d}\right)^n. \quad (12)$$

Therefore, to produce a better quality signal with a higher signal to interference and noise ratio (SINR) than is currently available, one method of optimization is increasing the transmitted power.

There are two significant restrictions to this method. The first is power consumption which puts strain on the battery life. To combat this restriction the transmission power can be designated with a quality of service (QoS) range that must be met. If the signal strength is strong enough for transmission the transmission power can be lowered to save on energy. If the QoS is not being met or suddenly changes due to new interference or barriers, the power can be increased to combat the change before failure.

The second constraint to transmission power is the power bound. In unlicensed frequency bands there are maximum legal transmission power etiquettes such that multiple devices can operate

within the frequencies. This limits interference with neighboring users but likewise limits the functionality of this RRM technique.

Table 3: Wireless Transmission Power Regulations by Frequency and Country (from [7])

Frequency Band	Geographical Region	Maximum Conductive Power/ Radiated Field Limit	Regulatory Document
2400 MHz	Japan	10 mW/MHz	ARIB STD-T66 [B22]
	Europe (Except Spain and France)	100 mW EIRP or 10 mW/MHz peak power density	ETSI EN 300 328 [B26] and [B27]
	United States	1000 mW	Section 15.247 of FCC CFR47 [B29]
	Canada	1000 mW (with some limitations on installation location)	GL-36 [B32]
902-928 MHz	United States	1000 mW	Section 15.247 of FCC CFR47 [B29]
868 MHz	Europe	25 mW	ETSI EN 300 220 [B25]

2.3.2. Admission Control

Admission control is the manner in which any system permits or denies access to users or devices wishing to use the system. In wireless networks, admission control is very important since the quality of the system decreases as more devices connect to it. All wireless standards have their own version of admission control. One example is ZigBee networks, where admission control is set up primarily by three parameters. These are the number of devices connected to each node, the number of devices connected to each node that can have other devices connect to them, and the number of devices that can be in one path from the primary device to the device furthest from it. By setting these values, the ZigBee standard does not allow any devices to connect if the maximum number of devices is already using the network. The problem with this system is that it has no way of restructuring itself in such a way that the maximum number of devices is reached before admission is denied.

2.3.2.1. Energy Spreading

Energy balanced routing is a way of selecting a path based on the remaining battery life of all nodes in the path. The main idea is to perform energy balancing when a node wishes to select a path such that the decision will be made with the residual power in each path considered. Using this strategy, the total energy of the mesh network will be balanced equally between nodes rather than having some paths with very low power remaining and some with a large amount of power remaining.

ZigBee uses accumulating path loss as its main mechanism for determining a route. Several factors from this mechanism can be used to improve ZigBee routing. These factors are the energy of the adjacent nodes, the energy of the node itself, and the quality o f the link. Simulations using these factors show that this type of algorithm can drastically improve the battery life of all nodes in the system [**8**].

2.3.2.2. Admission Control through Shifting

Admission control can be based on variables such as the number of devices currently in the system, the amount of noise currently in the system, or the strength of the signal to the device that wishes to join. In ZigBee, the admission control is done mainly through a distributed address assignment mechanism with three related configuration parameters. These are the maximum number of children devices allowed to a node, the maximum number of routers allowed under a node, and the maximum depth of the system, the longest path allowed from the coordinator to an end device [**9**].

One idea that has been presented is to allow devices to shift between nodes in such a way that the best configuration can be reached regardless of the order that the nodes joined the network [**9**]. For example, if a node had already reached its maximum number of children it would no long be able to accept a join request. However, if you allow one of its children to shift to another node close by the first node would now have one open spot for a child and would be able to accept a join request.

There is a MAC beacon payload in ZigBee for a node that is already in the network. Data from this payload can be used to check if a node has a neighbor that it could possibly join rather than its current parent. The MAC payload shown below in Figure 10 has two boolean values that can be used for this admission mechanism: *Router Capacity* and *EndDevice Capacity*. When a connected device receives a beacon frame from a neighbor, it can check these values to see if it could possibly shift to that neighbor [**9**].

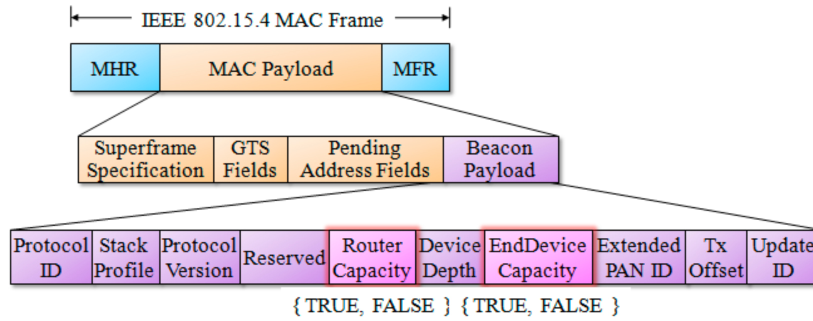


Figure 10: MAC Beacon Payload in ZigBee (from [9])

Using this data supplemented by code a shift operation can be created. One example of a situation that would benefit from shifting is discussed here. In Figure 11, there is a sample ZigBee network. In this network, parameters have been set that limit the number of nodes connected to any one node to three. The network also limits that the longest path in the network be no longer than 2 hops. If a new end device, E_5, wishes to join the network, it would normally be denied access since it can only join through R_1, which currently has the maximum number of allowed connected nodes as seen in red.

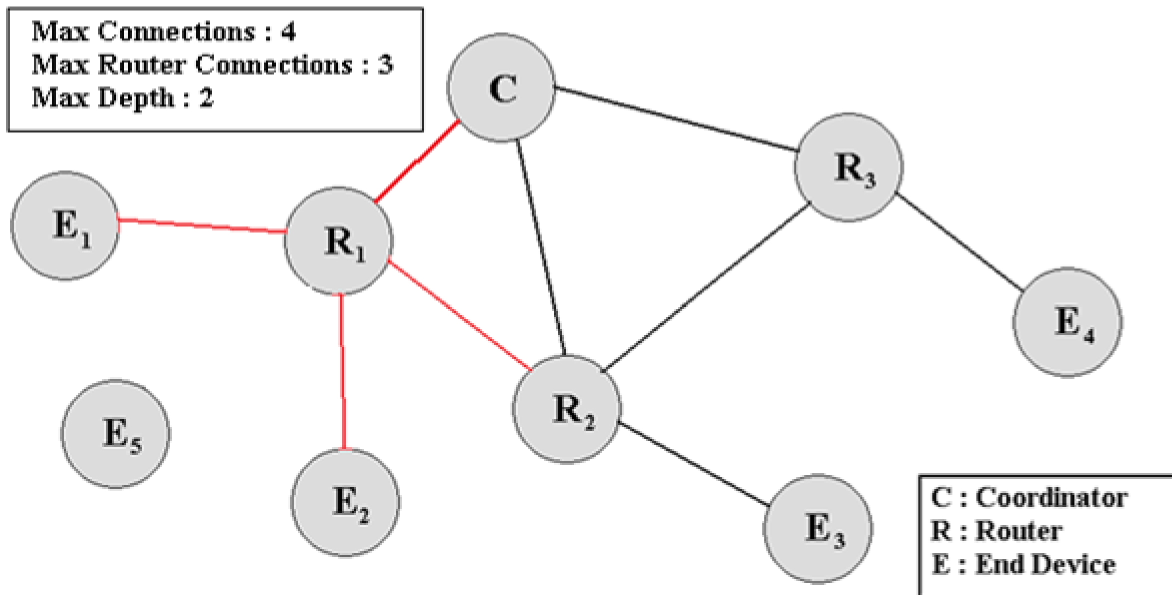


Figure 11: Sample ZigBee Network before Shifting

Introducing shifting into this system would allow E_3 to disconnect from R_2 and connect to R_3. This in turn allows E_2 to disconnect from R_1 and connect to R_2 which means E_5 can then connect to the network through R_1 since R_1 is no longer connected to E_2. The final system after shifting is seen below in Figure 12.

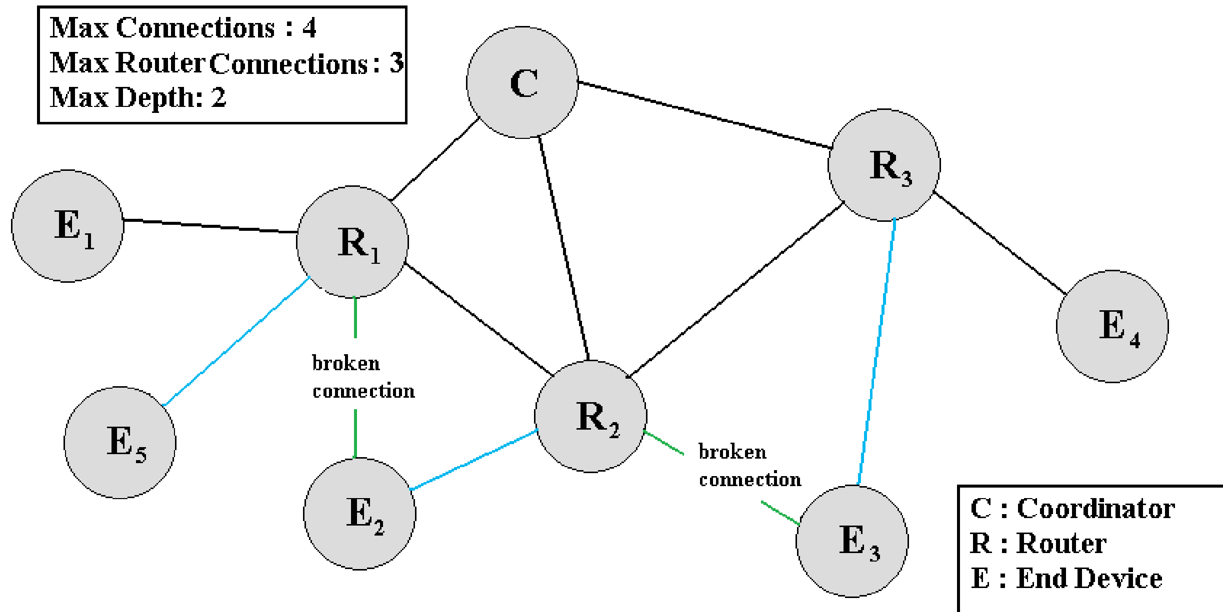


Figure 12: Sample ZigBee Network after Shifting

According to simulations this shifting mechanism can increase connectivity in ZigBee by 6% to 13% [**9**].

2.3.3. Forward Error Correction (FEC)

Forward error correction techniques are designed to allow a receiver to reconstruct data that has either been lost or corrupted in transmission. This can either be accomplished bit wise or by block. Bit wise FEC is called convolutional coding. The appeal of convolutional coding is that there are only a few bits of delay in the system, making it a very fast process. The implementation does usually require a decoder to be designed into the receiver called a Viterbi Decoder. This decoder will add complexity and processing necessity to the receiver. Block coding is used for error correction in bytes, words, or packets of any determined size. These block coding methods employ many error detection and data reconstruction methods of varying complexity and performance.

Traditionally forward error correction is not implemented for data transmission. This is due to low cost effectiveness when compared to that of automatic repeat request (ARQ) techniques. Forward error correction usually needs more processing power than that of ARQ from its complexity. Therefore,

it is easier and at times more reliable to substitute an ARQ technique for that of an FEC technique. FEC also tends to need more bandwidth than that of ARQ which leads to the latter being implemented more often. [**10**]

2.3.4. Automatic Repeat Request (ARQ)

Automatic repeat requests (ARQ) are re-transmissions that occur consecutively after the initial transmission to improve the bit error rate (BER). The number of automatic repeat requests N_R can be adjusted to maintain the quality of service (QoS) parameters of minimum correct packets F_i within a given maximum delay D_i. These QoS are correlated to the varying signal to interference and noise ratio (SINR) at the receiving node acquired during cognition [**4**].

The first step to deciding how many ARQ are required to maintain the QoS is assessing the maximum MAC protocol data unit packet loss rate PL_i as a function of ARQ N_R. This function is a probability function that depends on minimum correct packets F_i, maximum packet size M, and effective bits in the payload L_{eff}. The packet loss rate function is in [**4**]:

$$PL_i(N_R) = \left(1 - \left(\frac{F_i}{100}\right)^{\frac{1}{\lceil M/L_{eff} \rceil}}\right)^{\frac{1}{1+N_R}}. \tag{13}$$

From the packet loss rate function the bit error rate BER_i function can be derived. In order to derive the BER as a function of ARQ, the payload bits L_p, and the forward error correction bits L_{FEC} must be known. BER as a function of ARQ can be calculated using [**4**]

$$BER_i(N_R) = \left(1 - PL_i(N_R)\right)^{\frac{1}{L_p - (L_{FEC}/2)}}. \tag{14}$$

The maximum number of ARQ needs to be derived in order to find narrow down the domain of these functions and to ultimately find the necessary number of ARQ to maintain the QoS. The minimum of ARQ is always 0 and the maximum value of ARQ is [**4**]:

$$ARQ_{MAX} = \lfloor (D_i - D_{sys})/RTT \rfloor, \tag{15}$$

It is dependent on maximum delay D_i, system delay D_{sys}, and return trip time RTT.

From the BER function the target SINR γ^T_i function can be derived. The target signal to interference and noise ratio function expressed in [**4**]:

$$\gamma_i^T(N_R) = 2\left(erfc^{-1}\left(2BER_i(N_R)\right)\right)^2 \qquad (16)$$

The target SINR can be plotted again the domain of ARQs from 0 to the maximum feasible. From this generated range of target SINRs, the number ARQ closest to the corresponding acquired SINR is the minimum number of ARQ to meet the QoS. The system then adjusts its ARQ accordingly.

A test was run using this algorithm to make sure it worked properly. The maximum packet size M was set to 512 bits, the payload L_p was set to 400 bits, the minimum correct packet percentage F_i was set to 99.999%, the maximum delay D_i was set to 150ms, the system delay D_{sys} was set to 1ms and the return trip time RTT was set to 5ms. The forward error correction bits L_{FEC} and by definition the effective bits L_{eff} was varied to see the effect if variable forward error correction was implemented. The forward error correction bits range was set from 0 to 100. It can be seen in Figure 13 that the lower the SINR the more ARQs necessary to maintain the QoS.

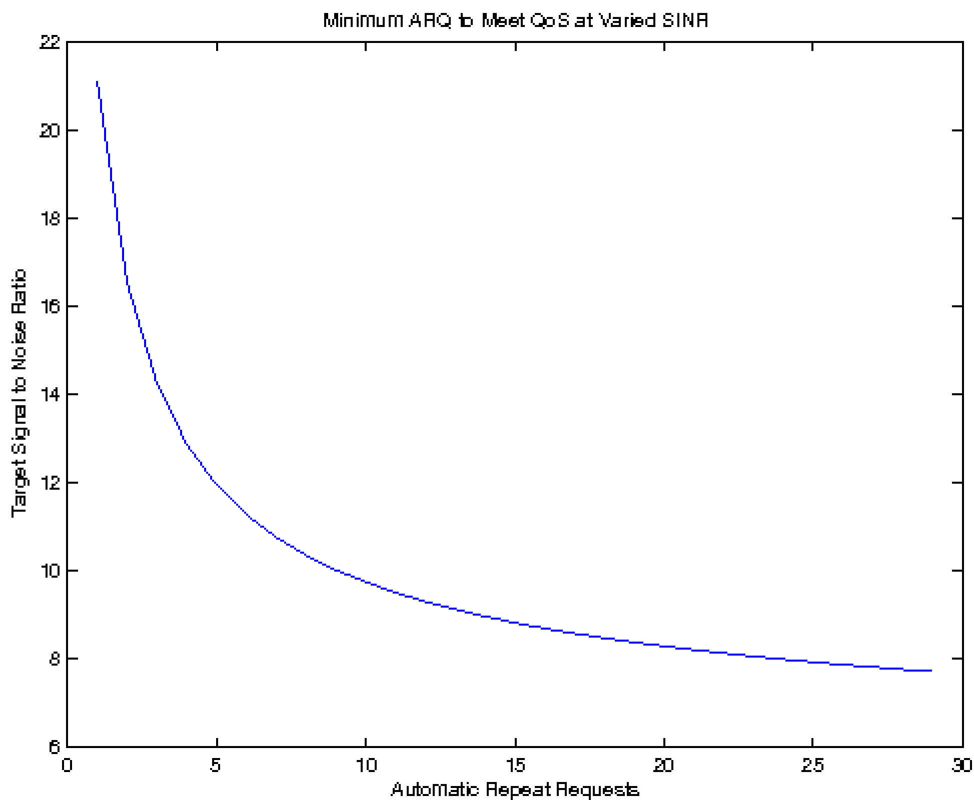


Figure 13: Plot of Target Signal to Interference and Noise Ratio as a Function of ARQs

2.4. Existing Protocols

Many different protocols have been defined under the IEEE standards. In this report, three specific standards are analyzed and one is implemented further on. The three standards discussed are WiFi or IEEE 802.11, Bluetooth or IEEE 802.15.1, and ZigBee or IEEE 802.15.4.

2.4.1. WiFi Protocol (802.11)

WiFi is the primary wireless standard in use today. Common applications of WiFi are personal computer operating systems, video game consoles, laptops, smartphones, and printers. Compared to Bluetooth and ZigBee, WiFi has the highest data rates anywhere from 1Mbps up to 144Mbps. It has an indoor range of up to 300ft and the highest fiscal cost with the shortest battery life [**11**]. As a result, WiFi is best suited for applications where large amounts of data need to be transferred quickly where a steady power supply is readily available.

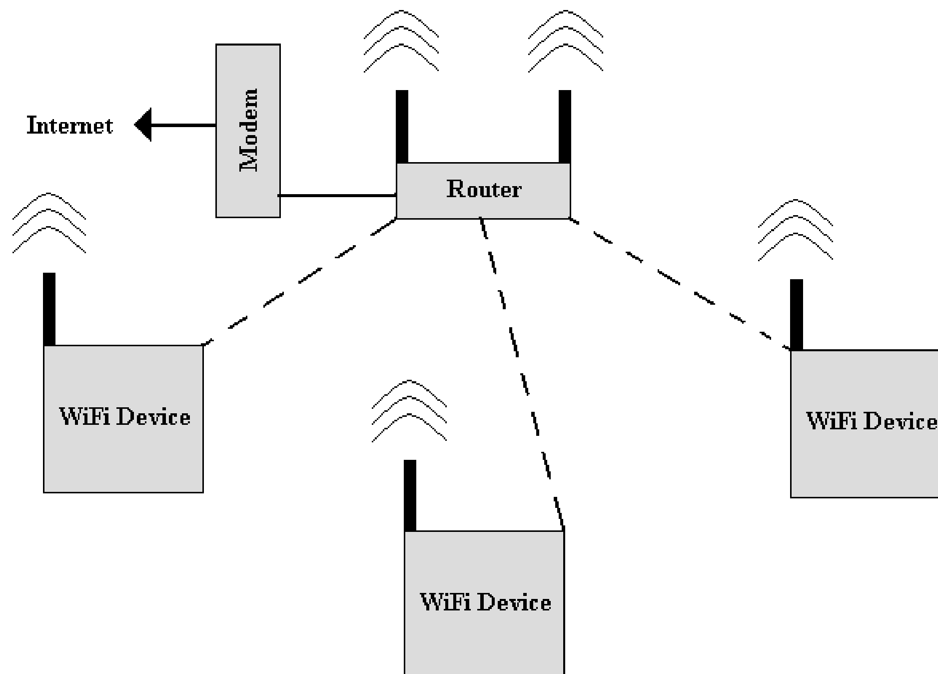


Figure 14: Wifi Network with three devices connected to a single router

As seen in Figure 14, WiFi uses a single router that connects all devices to the modem over either the 2.4 GHz or the 5 GHz spectrum. For modulation, WiFi uses various formats depending on the data rate and the 802.11 standard being used. Each of these standards is described in Table 4.

Table 4: WiFi Standards Comparison (from [12] and [13])

802.11 Type	Data Rate	Indoor Range	Modulation	Coding
A	27 Mbps	50 feet	OFDM	BPSK, QPSK, 16-QAM, 64-QAM
B	5 Mbps	150 feet	DSSS	CCK (5.5Mbps)
G	22 Mbps	150 feet	OFDM, DSSS	DBPSK (1Mbps), DQPSK (2Mbps), CCK (5.5Mbps and up)
N	144 Mbps	300 feet	OFDM, DSSS	BPSK, QPSK, 16-QAM, 64-QAM

2.4.2. Bluetooth Protocol (802.15.1)

Bluetooth is a wireless technology that allows single-hop connections between devices such as laptops, personal computers, cell phones, and headsets. It is based on the IEEE 802.15.1 specification and has three different versions, namely class I to class III. Bluetooth class I devices have an indoor range of around 300ft and last up to a day on two AA batteries, which is much longer than WiFi devices. However, Bluetooth's data rate is much less than WiFi at 0.8Mbps throughout. This makes it best suited for applications where two devices need to communicate low amounts of information [**11**]. For a modulation scheme Bluetooth uses Gaussian frequency shift keying (GFSK) [**7**].

2.4.3. ZigBee Protocol (802.15.4)

ZigBee is a standard based protocol for wireless sensor network applications that is used on top of the IEEE 802.15.4 specification. In terms of the OSI model, ZigBee adds the five top layers to the MAC and PHY specified in IEEE's 802.15.4. It is mainly used in situations where energy needs to be conserved and data rates do not need to be high. Several typical applications include home, building, and industrial automation. In other words, ZigBee is mainly used to control devices such as lights, appliances, and locks using a mesh network controlled by a remote location. In comparison to other wireless protocols ZigBee offers very long battery life, secure and reliable mesh networking, and low overall cost [**14**].

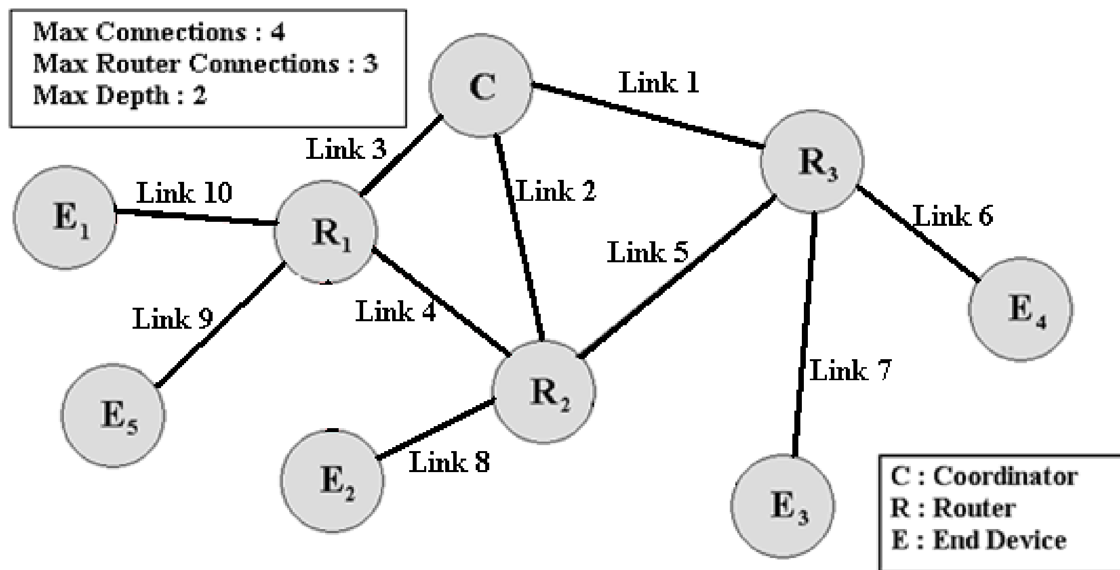


Figure 15: ZigBee Mesh Network

A ZigBee mesh network is made up of three different devices as seen in Figure 15. These are the coordinator, the routers, and the end devices. Coordinators are the devices that start the network. The network is started when a coordinator selects a panID and a channel. Coordinators also usually contain the trust center and the repository for all the network's security keys. The Trust Center is the mechanism which does authentication on devices wishing to join the network. It also maintains and distributes network keys and enables end to end security between devices.

ZigBee routers extend the networks coverage. Any device that wishes to connect to the network does not need to connect directly to the coordinator, instead if can just find and connect to the nearest router. A router or end device is added to the network by sending out a join request signal and receiving a join response signal from a nearby node. When a ZigBee mesh network expands using more and more routers, they can also provide multiple paths to transmit data giving options to find the best signal quality.

End devices are just how they sound; they are devices which cannot route data to other nodes in the network. They can only talk to one other device, their parent device, much like a Bluetooth connection. Data in the network is usually sent first to the coordinator and then on to its destination. However, it is possible to route a message from one device to another without going through the coordinator. This can be done using a route that is decided by the coordinator and stored in the routers [**14**].

ZigBee uses the same 2.4 GHz spectrum as WiFi and Bluetooth. However, ZigBee also has channels at 868MHz for Europe and 916MHz for the US. As it is made for long battery life and low data rates ZigBee has a throughput of only 0.25 Mbps but can last on two AA batteries for two to four days [**11**]. For a modulation scheme, ZigBee uses QPSK in the 2.4 GHz range and BPSK in the lower frequency ranges. More complex QPSK modulations are also available in the lower frequencies for better performance [**7**].

2.4.4. Comparison of Wireless Technologies

Since each wireless technology has its own set of pros and cons, it is important to any project that the appropriate type be used. Table 5 details the pros and cons of the three technologies the project has examined.

Table 5: Pros and Cons for Wireless Technologies

WiFi		Bluetooth		ZigBee	
Pros	**Cons**	**Pros**	**Cons**	**Pros**	**Cons**
Data Rate	Cost	Battery Life	Single Hop Only	Battery Life	Data Rate
	Battery Life	Low Cost	Data Rate	Mesh Network	

ZigBee was chosen for this project because of its unique mesh network capabilities and because it offers the most adaptive routing. Using ZigBee, a cognitive system could route around obstacles and have access to many more paths than it would with Bluetooth or Wifi. The main challenge with ZigBee is its low data rates.

2.5. Multi-hopping Methods for ZigBee

Multi-hopping is a form of ad hoc network routing, which allows a node to transmit packets to destination nodes outside its standard transmit radius by transmitting to nearby nodes and requesting they relay the packet on to the next node until it reaches its destination. The main issue with multi-hopping is how the nodes know which intermittent nodes to send their packet through for it to reach the destination. A few methods of deriving these routes are built into the ZigBee stack and will be described in this section. The routing algorithms designed in this project are able to decide these routes and implement them using source routing, which allows the specification of desired routes and intermittent hops of a node through the application programming interface [**15**].

2.5.1. Ad Hoc On-Demand Distance Vector Routing

Ad hoc on-demand distance vector routing is a link quality assessment and routing algorithm designed by the ZigBee alliance. This algorithm broadcasts a route discovery request to all the other nodes in the network. Each hop along the path calculates a path cost field that adds on more cost per hop until it reaches the destination. The destination compares all the path cost fields from each route until it finds the least expensive route and will send a MAC and application layer acknowledgement back to the source node with the chosen route. This method is a suitable means for providing a route that considers BER and RSSI of each connection, but fails to consider battery capacity preservation. It also requires each transmitting node to broadcast for route discovery. In networks with many nodes, this quickly leads to flooding. Since there is no means of adjusting this automated form of multi-hopping, other methods of multi-hopping were researched [**15**].

2.5.2. Broadcasting

Broadcasting is transmitting a message that is indiscriminately sent to the rest of the network. It is a common functionality in most wireless protocols, but in ZigBee, it is performed uniquely. The ZigBee broadcast function sends a message to the node it associated with to join the network, or its parent node, and all the nodes that associated with it to join the network, or its children nodes. These nodes are then responsible for transmitting to their parent node and children nodes. This process is repeated for the amount of hops set by the broadcast. The default and maximum value is set to 32 hops and can be set to the minimum of one hop [**15**].

2.5.3. Many-to-One

The many-to-one multi-hopping method is essentially a reverse broadcast. An aggregator, which is the coordinator in most cases, sends out a broadcast that requests each node to compile a reverse route as it travels through the nodes of the network. Once it has reached all the end devices, the nodes use their reverse routes to send transmissions back to the aggregator. These packets contain the routes that were traversed from the aggregator to the end devices. After the broadcast is made the end devices know the reverse route to use to reach the aggregator for transmissions in the future. These reverse broadcast requests are intended to be repeated periodically in order update the routes [**15**].

2.5.4. Source Routing

The fourth method of multi-hopping is called source routing. It is intended to be used in conjunction with many-to-one routing as a means of creating individual routes from the aggregator to the end devices without having to broadcast route request packet for each transmission. However, this method can also be used to create customized routes that can be dictated by a unique routing

algorithm. This method was ideal for implementing the system defined in this project's objective. Although the system would require utilization of all three methods of multi-hopping, the source routing would be used to implement optimized routes that the coordinator calculates. The only issue was that this method requires API firmware, which uses a different means of universal asynchronous receiver transmitter (UART) serial communication with the microcontrollers and PC. Updating the firmware was simple enough, but adjusting the code written for the AT command firmware had to be translated [**15**].

2.6.Routing Algorithms

There are a variety of different routing methods for discovering which path is ideal for either the network or the user. In this section, Dijkstra's algorithm and A* (a modification of the former) are evaluated. These algorithms were chosen based on their ability to be used in a mesh network as well as their ease of implementation.

2.6.1. Dijkstra's Algorithm

Dijkstra's algorithm is designed to find all paths in a network and trace the route with the lowest cost associated with it. It does this by assigning a cost to every node and updating it if it finds a route with a lower cost. These costs are determined by edge weights associated with every possible connection between two nodes.

The first thing that the algorithm does is it sets the cost of every node except the source as infinity. This ensures that the first time it is evaluated it will definitely be a larger cost than what it can be updated to. Having a larger cost than the value being updated facilitates the next step in the algorithm which compares previous costs to newly calculated costs. A node is chosen to evaluate its links if it is unvisited and has the lowest cost. When the program begins, all nodes are unvisited and the source is the only node that doesn't have a cost of infinity, as seen in Figure 16.

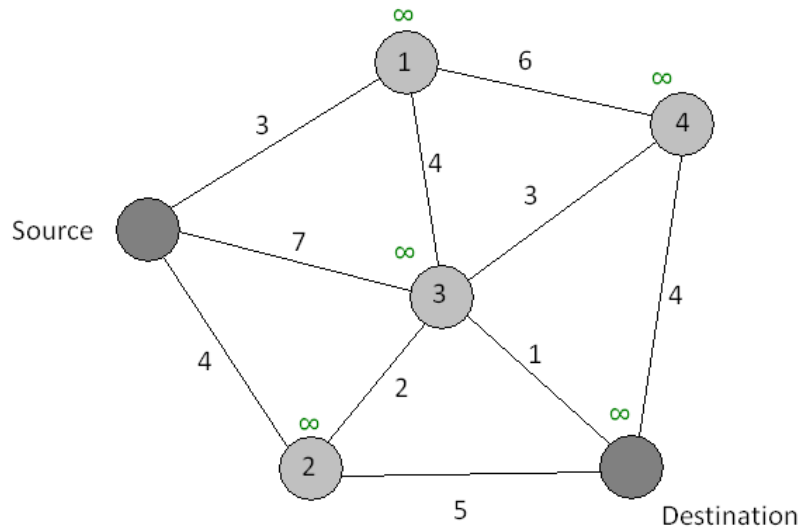


Figure 16: Dijkstra Routing: Set Node Costs to Infinity

The selected node then attempts to set the cost of all unvisited connected nodes. If the current cost at the node is greater than the sum of the cost of the sender and the edge weight of the connection, the node's cost is reset to that sum. Then, the updated node sets the sender as the ideal connection for routing. Finally, the sender node is set to the visited state. The first time this process is run, all nodes within a single hop of the source will be set since their previous costs were set to infinity. The resulting network with node costs is shown in Figure 17. Nodes 1, 2, and 3 are neighbors to the node that is currently being visited. They are updated to the cost of the edge weight between them because the edge weight is less than the current value of infinity.

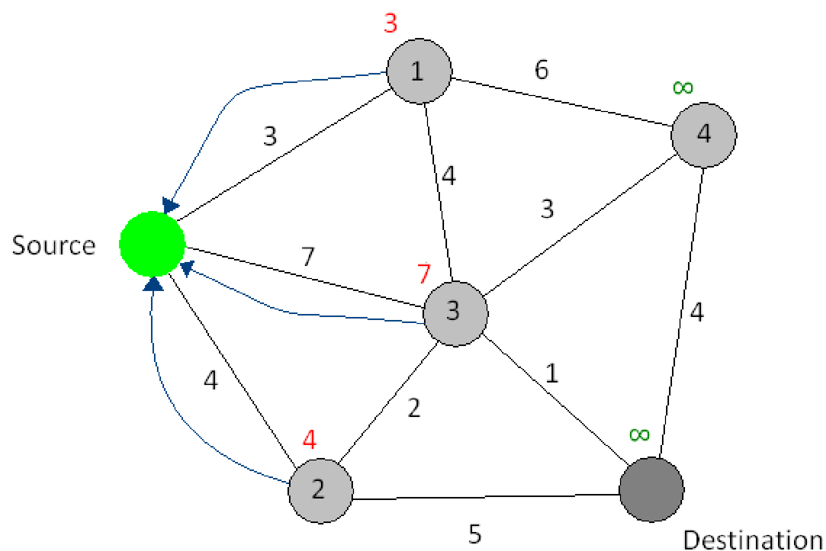


Figure 17: Dijkstra Routing: Calculate Nearby Node Costs

The algorithm will repeat the process. It will select the lowest cost unvisited node to update the cost of its neighbors. The cost of nodes and ideal path back will only be updated if the new cost would be less than the original. In the example in Figure 18, the top node has the lowest cost, 3, and is therefore the first selected in the routing algorithm. The algorithm updates node 4 since its current cost is infinity and does not update node 3 since its current cost is equivalent to the new value. The algorithm then determines there is no improvement by going through the current node being visited.

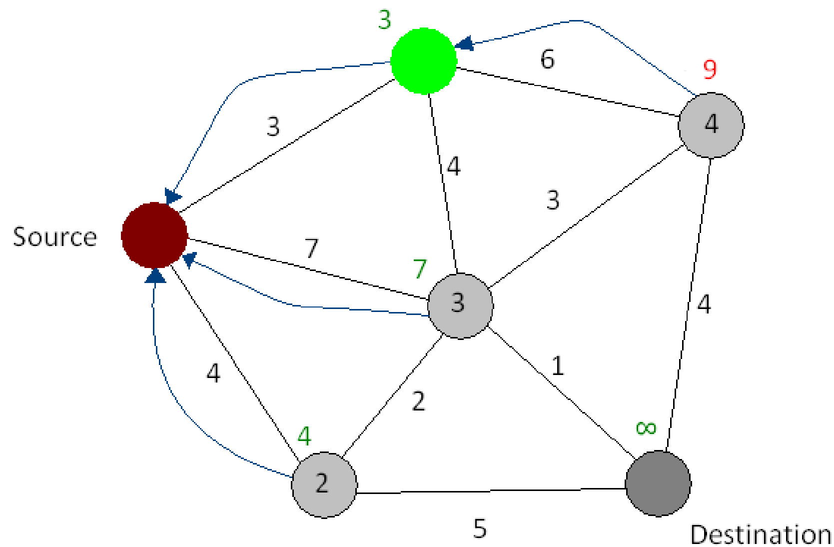


Figure 18: Dijkstra Routing: Change Node and Calculate Costs Again

If a path is found that is a lower cost than the original, then the old cost of the node is changed along with the path back. In the example in Figure 19, node 3 had a cost of seven by going directly to the source. However, the path through the current node in green is 6, 2 plus 4, and therefore the path from node three and the cost of node three are changed.

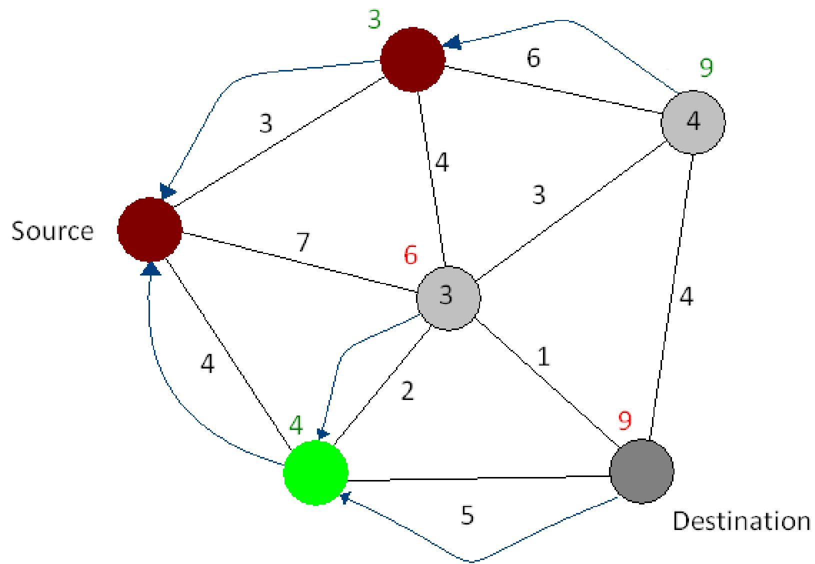


Figure 19: Dijkstra Routing: Change Paths when Lower Cost is Found

When the destination is the lowest cost unvisited node the process is over since the other routes are already a higher cost than the path that has already been chosen. This is seen in Figure 20 where the destination has a cost of 7 and the only other unvisited node has a cost of 9. The final path is made using three hops to get back to the source.

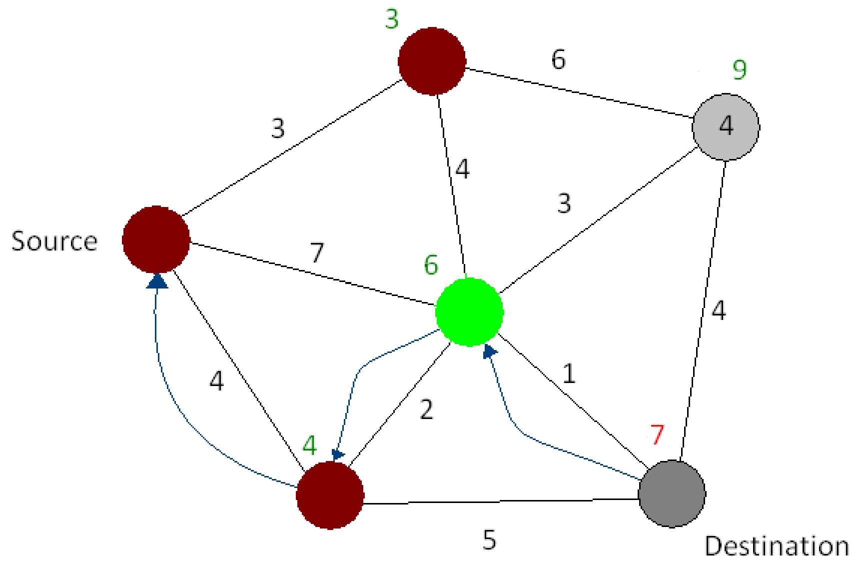


Figure 20: Dijkstra Routing: Algorithm Finished when Destination is Reached

This algorithm is very time efficient and can handle many nodes in a network. The drawback of the algorithm is that it cannot process an edge weight of negative value. A modification of Dijkstra's algorithm, called the Bellman-Ford algorithm, could be implemented if the edge weights used are

negative. If all weights can be adjusted to be positive Dijkstra's Algorithm is a very good method of routing.

Table 6: Pros and Cons of Dijkstra's Algorithm

Dijkstra's Algorithm	
Pros	*Cons*
Always finds best route	Cannot Use Negative Edge Weights

2.6.2. A* Routing

A* routing is a heuristic model that is based on Dijkstra's algorithm. It calculates the cost of each node by the cost of the path while also incorporating the actual cost between the node and the start point. The cost in A* routing is usually applied using distances. In that case g(x) is the straight distance between the current node and the start point. The value h(x) is the distance traveled through possible links. The cost that is applied to the node is f(x), the sum of g(x) and h(x).

This algorithm is not very applicable to a wireless network. This is due to the fact that g(x) isn't a readily available value. With distances, the Pythagorean Theorem can be applied to the coordinates of the node to find the direct distance. If a connection to the node is not possible there is no accepted value for the direct cost between the node and the source.

Table 7: Pros and Cons of A* Algorithm

A* Algorithm	
Pros	*Cons*
Fast Run Time	Doesn't Always Find Best Path
Memory Efficient	Cannot Use Negative Edge Weights Direct Link not Readily Available

2.7. Chapter Summary

Cognitive radio, as defined by this application, is an external engine to the MAC and physical layers with the purpose of detecting system performance and providing feedback to RRM modules. The cognitive parameters that will be explored are RSSI, and BER. The RRM techniques that will be implemented are transmission power and admission control with energy spreading. This admission control will establish source routes for a ZigBee network.

3. Project Goals and Design Decisions

In order to determine the type of network that we wish to design we must first find an application. This application will set the product requirements such that we can choose the desired protocol and radio resource management (RRM) techniques to incorporate into our system. The application that we chose was hospital instrumentation and medical devices. We identified a need for a cognitive network within this field and used the requirements found to design our network.

When designing the network the following there are many decisions that need to be made. The first decision is which cognitive parameters are most relevant and readily attainable to determine the performance of the system according to the network requirements. The second decision is which RRM techniques to implement in order to use these parameters effectively to optimize the system. The third and forth decisions are which hardware to use and the firmware that should be associated with it. Finally the routing algorithm which most effectively optimizes the network according to the requirements must be implemented. These choices and selections must be evaluated and decided upon.

3.1. Cognitive Radio for Medical Devices

Wireless technologies can greatly improve healthcare devices. The three major benefits that wireless technologies add are increased mobility, lower cost, and improved signal quality. The necessity for a cognitive system to be in place is due to the high demands of these devices. Most hospitals still use wired connections, despite their lack of mobility, because wireless communication is considered to be too unreliable. Instrumentation in hospitals needs to be both portable and reliable, such that a doctor can continuously monitor a patient as they are moved from one room to another. By adding a cognitive system to wireless medical devices, loose cords and constant rewiring will no longer be required. It allows for a constant connection between hospitals records while in transit. Additionally, the wireless devices may be used for location of patients in hospitals, psychiatric wards, or nursing homes.

To create a cognitive network, certain design decisions need to be made. The technology used should be both inexpensive and reliable. Embedding all processing necessary for all instrumentation is costly. Rather than programming the algorithms directly into the devices, it is much more cost effective to have a central computer to do all the calculations given the raw data. This data can be transmitted wirelessly to be processed and have a result sent back. For this purpose, the ZigBee hardware is less expensive than the WiFi and Bluetooth protocols. In respect to reliability, wireless devices have a better signal quality due to the shorter signal path. Rather than sending the signal through a series of wires to

attenuate and potentially be distorted, it is immediately sampled and transmitted digitally to be evaluated. Assuming a high enough sampling rate, this makes the data more accurate and reliable.

3.2. Product Requirements

The requirements for this project can be divided into four categories. These categories are: reliability, portability, energy efficiency, and data rate. For the instrumentation that will incorporate the cognitive network each category must be addressed.

3.2.1. Reliability

As with all medical devices, reliability is of the highest importance. Since every product is designed to potentially save somebody's life there is no room for error. This means that the RSSI must remain in an acceptable range such that the signal will not be lost. The BER of the connection must be very low as incorrect data may lead to a misdiagnosis. The device must have a high reliability above all else.

3.2.2. Portability

Doctors and patients are constantly moving in a hospital. This means that the device needs to be able to update its connections to the coordinator often. The routes may constantly be changing due to the new location of every node in the network. The portability means that the device also must be able to run on battery power with an acceptable battery life.

3.2.3. Energy Efficiency

If a node of the network needs to be carried by the doctor it needs to be able to operate for an entire shift or longer. Doctors should not have to worry about the battery of their devices in the middle of a work day. This requires long battery life of every node in the system as there is no time for the devices to constantly be recharged.

3.2.4. Data Rate

Different medical devices will require different data rates. For example, image processing would require a much higher data rate than a heart monitor. Higher data rate devices would be restricted to using the 802.11 protocol. The devices that this paper will target are the low data rate instruments for the purpose of using ZigBee as a feasible option.

3.3. Project Management and Tasks

In order to be successful in a time restricted project, a schedule must be made and maintained that sets deadlines and milestones that must be met. The main medium used for the time management in this project was the Gantt chart shown in Figure 21.

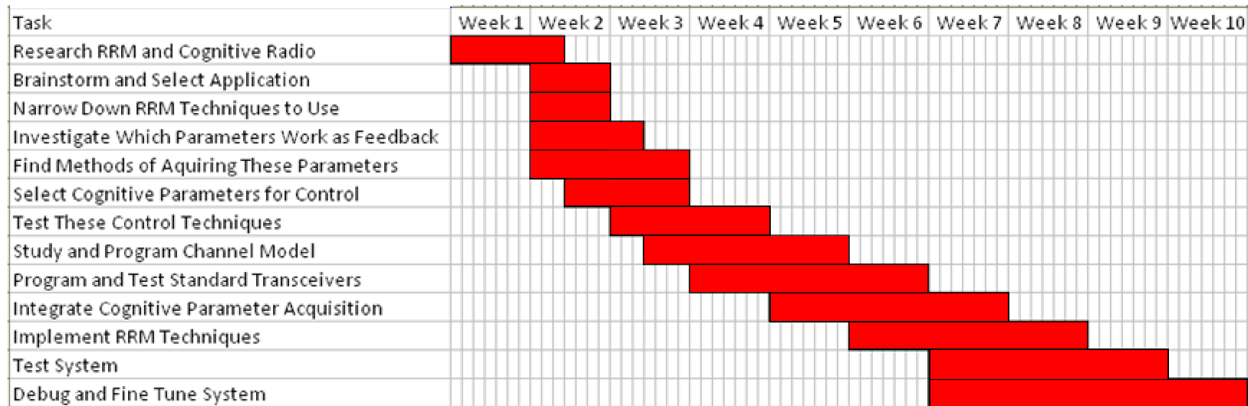


Figure 21: Initial Project Gantt Chart

The project was scheduled using three major stages. The first stage was to research and understand the problem. We had to gather as many sources as possible to quickly understand why cognitive radio is needed in wireless hospital networks. Second, we needed to examine all major choices that needed to be made such as hardware and software selections. Third, the system needed to be built and thoroughly tested. These three goals were split into tasks and given deadlines over the ten weeks available.

In the beginning of the project we decided that the final project should include a simulation for testing the accuracy of the network optimization. This simulation would use a channel model to simulate different environments where interference would be known and a result from the system could be predicted. In week 4, this plan was removed from the project due to concerns with time restrictions. The final project Gantt chart, shown in Figure 22, does not include the previous task of studying and programming a channel model. There is also an extension, shown in red, in one of the project tasks for testing control techniques. This delay was caused by a change in the model of the hardware we were using as discussed in Section 3.7.2.3.

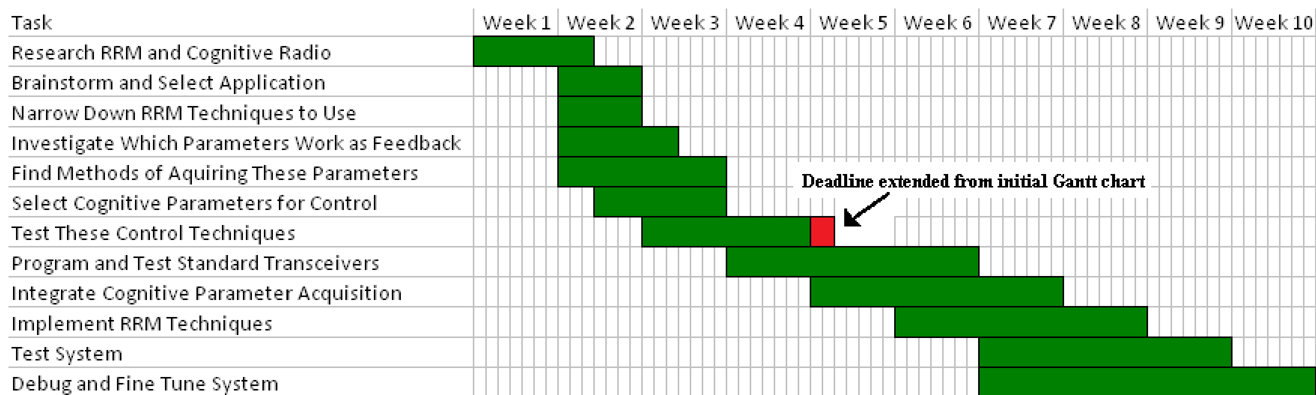


Figure 22: Final Project Gantt Chart

In the beginning of the project, we created the initial Gantt chart in Figure 21 by assuming a large amount of delay in all tasks. The final Gantt chart demonstrates how well this delay was approximated since all of the tasks, except testing control techniques, were completed either on schedule or early.

3.4. Optimization Process

The optimization process will have three main steps. These steps are environment sensing, processing, and parameter setting. In this report, environment sensing will include finding RSSI, BER, and energy capacity for all possible connections.

The environment sensing step starts by sending out a request to begin the optimization process. This synchronizes all routers and tells them to carry out the next step. Every router that receives this request, along with the coordinator, then performs a node discovery. During the node discovery, the routers sense all other nodes that are within a single hop. All of the addresses found are contacted such that a BER and RSSI reading can be evaluated from the transmission. Finally, all of the cognitive parameters found are sent back to the coordinator for processing.

In the processing step, the coordinator receives all of the cognitive parameters from all of the links and organizes it into readily accessible data. The coordinator then performs a routing algorithm to determine the ideal routes for multi-hopping paths in the system while taking power consumption into consideration. It is also responsible for choosing what transmission power level each node should be operating at.

The final step is parameter setting where the paths of transmission and power levels are sent out to the routers such that they may set the given parameters. All of the nodes are then using the ideal

settings for the network and may continue with operation. Figure 23 shows the step by step process of optimizing the network. This process is broken down and fully explained in chapter 5.

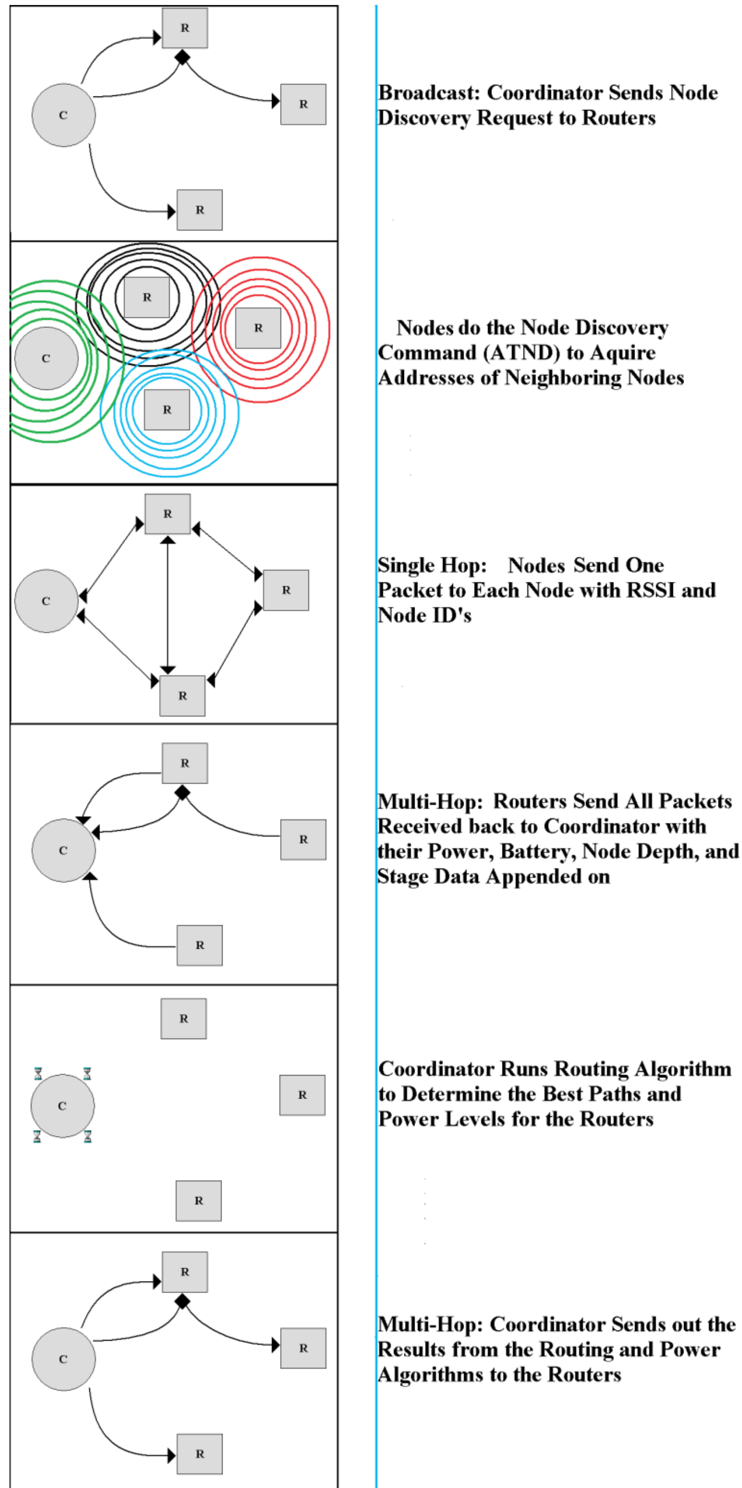


Figure 23: This is the network optimization flow that is performed cyclically to keep up with real time changes

3.5. Cognitive Parameters

The cognitive parameters that we chose for the network were RSSI and BER of each link, and the battery level of every node. These were chosen due to the fact that these parameters were readily available in the hardware chosen, the XBee module from Digi [**15**]. More about this hardware can be found in Section 3.3.2. They were also determined to be highly relevant to the network requirements that we set up. The RSSI can be accessed from pin 6 on the XBee chips in the form of a pulse width modulated (PWM) signal. The duty cycle of this PWM corresponds to the received signal strength of the most recently received packet. The higher the duty cycle of the PWM is, the stronger the signal coming in was. The BER of a link can be found by sending a known sequence and comparing the received sequence to the expected one. By using a bitwise XOR operation on the received sequence compared to the expected sequence, all the bit errors will be logic high. By counting the number of bit errors and dividing by the number of bits in the known sequence the percentage BER is found. The battery level of the node is set manually for our purposes. However, with a proper power monitoring an accurate battery level parameter can be set. This can either be expressed in charge, Coulombs, or in remaining battery life as a unit of time.

3.6. Radio Resource Management

The first radio resource management technique selected by the group was transmission power. It was determined that this was a relevant RRM technique for our network requirements. A higher transmit power promotes a healthier RSSI and BER. This will raise the link qualities in the system to meet the QoS requirements. If these requirements are satisfied, the device can begin to decrease transmit power. This is beneficial to the network for two reasons. The first reason is that this theoretically has less of a toll on the battery. The second reason is that it is less likely to interfere with other nodes in the network.

The other RRM technique implemented is admission control. Specifically it is a routing system that spreads the energy of the system. By doing this we can prevent a node with a low battery from being overworked and thereby extend its battery life. This allows us to find the routes for transmission of high quality while having a low strain on the network.

3.7. Hardware

Selecting a hardware device for the project defines much of the successive steps in the design process. In other words, a device with more memory will allow for a more memory intensive code. A device with a restriction on transmit power or transmit data rate will also drastically change the project.

The firmware used is completely dependent on what firmware the hardware supports. Consequently, two distinct hardware options are studied in this report. These are the UZBee dongles from Flexipanel [**16**] [**17**] and the XBee development board from Maxstream [**18**]. These two options are distinct in the way the entire system will be implemented. Specifically, dongles are connected to computers permanently and therefore would have access to the vast amount of memory the computer has. However the dongles would also be limited by the slow USB connection. In contrast, the XBee boards are limited by their on-board memory. Their benefit is that they are standalone systems that only use the computer for uploading code and for power.

3.7.1. UZBee Dongles

Initially, a UZBee 802.15.4 dongle [**17**] and a UZBee+ 802.15.4 dongle [**16**] were provided for the project. The two dongles, manufactured by Flexipanel are capable of acting as fully functioning devices in the 802.15.4 protocol, meaning they could act as network coordinators, routers or end devices. The ZigBee protocol could also be implemented on both dongles. The UZBee+ could act as a full function device in the ZigBee standard, whereas the UZBee could only act as an end device. An image of the UZBee+ can be seen in Figure 24.

Figure 24: This is a 802.15.4 UZBee + dongle (from [16]).

In order to communicate commands to the two dongles and parse communications, the right computers had to be selected. Two Eee PC Linux systems are used in this project. The two Eee PC systems were initially running a primitive kernel and shell, until they were installed with Ubuntu 9.04. Also used were three personal laptops, two running MS Windows 64-bit Vista and one running MS Windows XP. The first step taken was to install the two dongles to these computers.

According to the Flexipanel website, the UZBee dongle is compatible with MS Windows XP and MS Windows Vista. The UZBee dongle uses the `USBoot.inf` driver information file in conjunction with the `usbser.sys` file. This driver creates a virtual COM port or VCP from the USB ports. A COM port is an old standard serial port that can allow communication between modems and the computer.

The usbser.sys file is built into the Windows/System32/drivers folder for MS Windows XP. The USBoot.inf file can also be installed to MS Windows Vista, as long as the usbser.sys file is placed in the Windows/System32/drivers folder.

Attempts were made to install the UZBee on MS Windows Vista and MS Windows XP. Since two MS Windows Vista machines were available for the project, the first attempt to install the UZBee was on MS Windows Vista. The usbser.sys file was added to the driver folder, but MS Windows Vista did not recognize the USBoot.inf file. After some more research into the problem was done, it was found that the usbser.sys is only compatible with MS Windows 32-bit Vista. Since the Vista systems run MS Windows 64-bit Vista, the UZBee will not work with them. However, running the USBoot.inf driver on MS Windows XP was successful. The 802.15.4 protocol firmware for the UZBee called MACdongle was then installed in the end device configuration and a MAC address was allocated to the dongle. With one of the dongles working, the UZBee+ had to be set up in order to create a two node network.

The Flexipanel website states that the UZBee+ requires a new driver for the FTDI chip implemented in the UZBee+. FTDI provided several VCP drivers for Macintosh, Linux, and MS Windows systems up to and including MS Windows XP. For MS Windows Vista, a combined driver model or CDM was provided that integrated VCP drivers and D2XX drivers, which allows application software to directly access the UZBee+ through a series of DLL function calls [**19**].

Attempts were made to install the UZBee+ on MS Windows Vista and Ubuntu. The first attempts of installation were made on Vista. The CDM driver files from FTDI were not being recognized by MS Windows Vista. After this, an install of the Linux FTDI VCP driver on Ubuntu was attempted. It did not seem to be working, so another look was taken at the FTDI website. It stated on the "3rd Party Driver" page that the latest version of Ubuntu runs Linux kernel 2.6.28 and the USB to serial converter driver was already installed on the system. The serial port is labeled in the dev folder as ttyACM0 or abstract control model zero. With this serial port, an attempt to install the 802.15.4 firmware for the UZBee +, PixieMAC was made and resulted in failure [**19**].

Another attempt to install the CDM files on MS Windows Vista was made, with closer attention to the troubleshooting guide in the FTDI installation instructions. Flexipanel said to check the vendor identification and product identification of both the .inf files in the CDM folder and the pre-programmed EEPROM values of the UZBee+. In order for the system to work these numbers needed to

match and they did not. It was discovered that the same circuit board with the same model number was printed on both dongles: `UZBr8`. The UZBee was mislabeled by the casing.

With two UZBees and only one MS Windows XP Machine, two decisions were made. Another MS Windows XP machine needed to be found, and in the mean time the UZBees should be tested on the Ubuntu Eee PCs. Using the MS Windows XP system, The MACdongle firmware was installed to the second UZBee in the coordinator configuration and then a second MAC address was assigned to it. Then, the UZBees with the loaded firmware and defined MAC address were plugged into the two Ubuntu Eee PCs. Since MACdongle is a command based application interface, `gtkterm`, a clone of hyperterminal was used to communicate through the serial port `ttyACM0` to the UZBees. Using the MACdongle datasheet [**20**], several commands were tested to try to establish single hop communication.

First, the `gtkterm` was tested for its ability to communicate with the dongles. A presence detection command was used in order to test whether or not the dongle recognized the commands. It confirmed its presence, vendor identification, product identification, the MACdongle version number, firmware release date, and device configuration. The corresponding commands appeared as follows in the terminal communicating with the coordinator UZBee:

```
+DVRR  (Requests version)
+DVRC=0B4001111035220512 0601  (Confirmation with coordinator version number)
```

The last byte of the DVRC confirmation differs for the two UZBees because of the two different configurations. The corresponding commands appeared as follows in the terminal communicating with the end device UZBee:

```
+DVRR  (Requests version)
+DVRC=0B40011110035220512 0602  (Confirmation with end device version number)
```

Timeouts were disabled for diagnostic purposes. The suppress timeout request extends the timeout period for certain commands that require a response from the host. If diagnostic is done typing

line by line, the timeout suppression is essential in keeping up with the UZBees. The UZBees both confirmed the request. These commands are written below:

+DHTR (Request to disable timeouts)
+DHTC (Confirmation that it was disabled)

A long address mode packet transmission was then attempted to perform a single hop between the two dongles. The data packet transmission request command specifies the packet length, frame type, transmit options, source PAN ID, source address mode, transmission address mode, source address, transmission address, destination PAN ID, data handle and payload. Two attempts were made. One was from the coordinator UZBee to the end device UZBee and one was from the end device UZBee to the coordinator UZBee. Although packet transmission confirmations were sent back from the dongle, the data handles did not match in both attempts and therefore were unsuccessful. No packet indication appeared of the receiving terminal in both attempts. The commands that were intended to send "90ABCDEF" from the coordinator to the end device are written below:

+MDAR=04000101FFFF03031405000FBFC815001505000FBFC81500FFFF8890ABCDEF
(Request to send data 90ABCDEF of length 04 from 0015C8BF0F000514 to 0015C8BF0F000515 in long address mode)
+MDAC=E9000101FFFF03031405000FBFC815001505000FBFC81500FFFF10
(Confirmation stating that transmission was not successful)

When trying to send the packet "90ABCDEF" from the end device to the coordinator, the data handle did not match again and therefore was unsuccessful, as confirmed by there being no packet data indication on the coordinator terminal. The commands intended to send "90ABCDEF" from the end device to the coordinator are written below:

```
+MDAR=04000101FFFF03031505000FBFC815001405000FBFC81500FFFF8890ABCDEF
```

(Request to send data 90ABCDEF of length 04 from 0015C8BF0F000515 to 0015C8BF0F000514 in long address mode)

+MDAC=E9000101FFFF03031405000FBFC815001505000FBFC81500FFFF00

(Confirmation stating that transmission was not successful)

A network was set up on the coordinator UZBee to see if the end device could scan it and then join it. First, a short address has to be configured by the coordinator. When this command was given to the coordinator UZBee, it confirmed the short address. Second, the coordinator must use the start operation command to start a network. The start operation command identifies which operation, in this case PAN coordinating, the PAN ID, and the frequency channel. When this command was communicated to the coordinator UZBee, the network was confirmed. The network was then set to permit association. These three commands and their confirmations are shown below:

+MSTR=530000 (Request to set short address to 0000)

+MSTC=0053 (Confirmation that short address was sent)
+MSRR=0100000034121100000000000F0F (Request to start PAN ID 1234)

+MSTC=00 (Confirm that PAN ID was started)
+MSTR=4101 (Request to permit joining)

+MSTC=0041 (Confirmation that joining is permitted)

The channels were then scanned by the end device UZBee to sense the coordinator UZBee's network. The scan operation request specifies the type of scan, in this case active, the scan duration and the channels scanned, in this case all 22 channels. The confirmation was sent back from the end device dongle communicating no returned results. The scan command and confirmation are shown below:

+MSCR=00000000000010800F8FF07 (Request active scan on all channels)

+MSCC=EA00000000000108FF0700F8 (Confirmation nothing was found)

After securing another MS Windows XP machine the same commands were performed again to see if these issues would be resolved from using the operating system for which it was designed. The problem was not resolved even using two MS Windows XP machines. At this point in the project, the XBee development boards had already been successfully installed and the idea of using UZBee dongles was abandoned.

3.7.2. XBee Development Boards

Investigated in this section is the XBee OEM RF Module, as seen in Figure 25, using an Arduino Duemilanove development board to transmit and receive over IEEE 802.15.4 wireless. The XBee module comes in two distinct versions, series 1 and series 2. Series 1 XBee does not contain the ZigBee stack whereas the series 2 XBee does contain the ZigBee stack. Examined in this section is the choice made between the two and the general transmit and receive capabilities of the XBee development boards.

Figure 25: This is an XBee Series 2 ZB RF Module.

3.7.2.1. Transmitting on the XBee

To start this section a preexisting program [**21**] was used that acquired and transmitted readings of temperature, light, humidity, and dew points. This program gave the project a point to work off of in which the XBee boards were already receiving and transmitting packets with one another. From this point research was done looking at the manual [**18**] supplied by the XBee board's manufacturer, MaxStream.

Transmitting on the XBees is done by writing the data to the serial port with a `Serial.print()` command. The default setup of the XBee chips sent all data printed to the serial port directly to all nodes in the system. While this was good for using the chips right out of the box, the project requires that the transmissions can be directed to certain nodes in such a way that routing can be done separate from the original routing software.

All settings on the XBee chips can be altered using the built in AT commands as described in the product manual [**15**]. These commands allow the user to change the channel and panID of the network being used and even let the user change the power level at which the chip was transmitting. The AT command ATPL sets the XBee at one of its five power levels for transmissions. These power levels can be seen below in Table 8.

Table 8: XBee Power Levels [18]

Power Level	XBee	XBeeZB
0	-10 dBm	-4 dBm
1	-6 dBm	-2 dBm
2	-4 dBm	0 dBm
3	-2 dBm	2 dBm
4	0 dBm	4 dBm

Using AT commands the transmissions of the XBee chips can be sufficiently controlled to meet the needs of this project.

3.7.2.2. Receiving on the XBee

The original receiver code [**18**] was set to receive from the end device, parse the data, and write it both to serial and to Ethernet as a server update. The Ethernet aspect of the receiver has been taken out and additional parsing schemes have been added to the receiver to read and print out all data received. Using this information, all transmissions can be checked for errors.

Receiving on the XBee chips is done by filling a buffer whenever data is available. When a `Serial.read()` command is issued, one byte of data is sent as the output of the command to be saved to a variable and is subsequently deleted from the buffer. One note about the system is that the buffer is about 200 bytes, meaning if data fills the buffer up to 200 bytes, data will begin to be lost. This means that any code planning on reading in data should constantly be checking if there is data to be read such that no data will be lost.

3.7.2.3. XBee Series 1 and Series 2

The functionality of the two different XBee modules varies between the 802.15.4 Series 1 modules and the ZigBee Series 2 modules. There are many similarities and some of the implementation was performed on the Series 1 boards. However, there are some key differences that caused the

decision to switch from 802.15.4 to the ZigBee XBee modules. This section describes the experimentation and implementation on the 802.15.4 boards that led to the decision to switch from 802.15.4 to ZigBee XBee Boards.

The XBee 802.15.4 Series 1 modules are configured to transmit messages indiscriminately to all the other XBees. However, selective transmission can be performed using addressing. This protocol can support two modes of addressing. The first mode of addressing is long address mode where each device has a predefined 64-bit address that can be used to direct transmissions. The ZigBee stack also has this 64-bit address as it is defined in the physical layer in 802.15.4. The second mode of addressing is the short address mode which uses 16-bit addresses.

Long address mode reduces complexity of selective transmission at a cost of longer addresses. The long addresses of the Xbees were predefined from manufacturing and cannot be changed through commands. Attempts were made to use the long address mode for selective transmission. Using the ATSH and ATSL commands, the Series 1 Xbee was asked for its long address and returned its address. This address was then used to set the destination of a transmitting XBee. The message successfully transmitted, but long addresses are too long to transmit as a header for multi-hopping purposes. Since the ultimate goal was to define multi-hopping paths based on cognitive feedback this method was abandoned.

These conclusions lead to experimentation with the short address mode. The short address mode added a little complexity to the programming of the RF module because it required that a personal area network, or PAN ID, be set up. This PAN ID is necessary because it creates an organized network defined by a coordinator that associates with other XBee devices. These associations require all the nodes in the network to confirm the connection of the other nodes which allows the addressing to be less formal.

One of the Xbee devices was configured as a coordinator. Initially, a command was used that allowed the coordinator XBee node to scan all the channels for predefined PAN networks to make sure it was not using a PAN ID or channel already in use. This feature was considered useful in case the scope of the project required more than one coordinator. However, there were many complications involved with this approach. First, whenever the coordinator scanned the channels to pick a PAN ID and channel, it always seemed to pick the default PAN ID and default channel. Second, when an end device was configured to automatically associate with the coordinator, it did not seem to work.

It was decided that the PAN ID, the channel and the address could be manually entered in order to perform selective transmission. Using the ATID and ATCH command the channel and PAN ID were entered in the coordinator and the coordinator was told to start a PAN using these parameters. Another end device was configured to join the network with the same PAN ID and channel as the coordinator. A short address was entered into the other XBee device as the source address and the same short address was entered into the coordinator as the destination address. A test packet was successfully sent from the coordinator to the other XBee. Yet another XBee was set up to receive packets and that did not receive the packet that was sent selectively to the XBee node that was defined as the destination. The successful transmission from the coordinator transmitter in Figure 26 (a) and the end device receiver from Figure 26 (c) can be viewed through the virtual com ports in Figure 26.

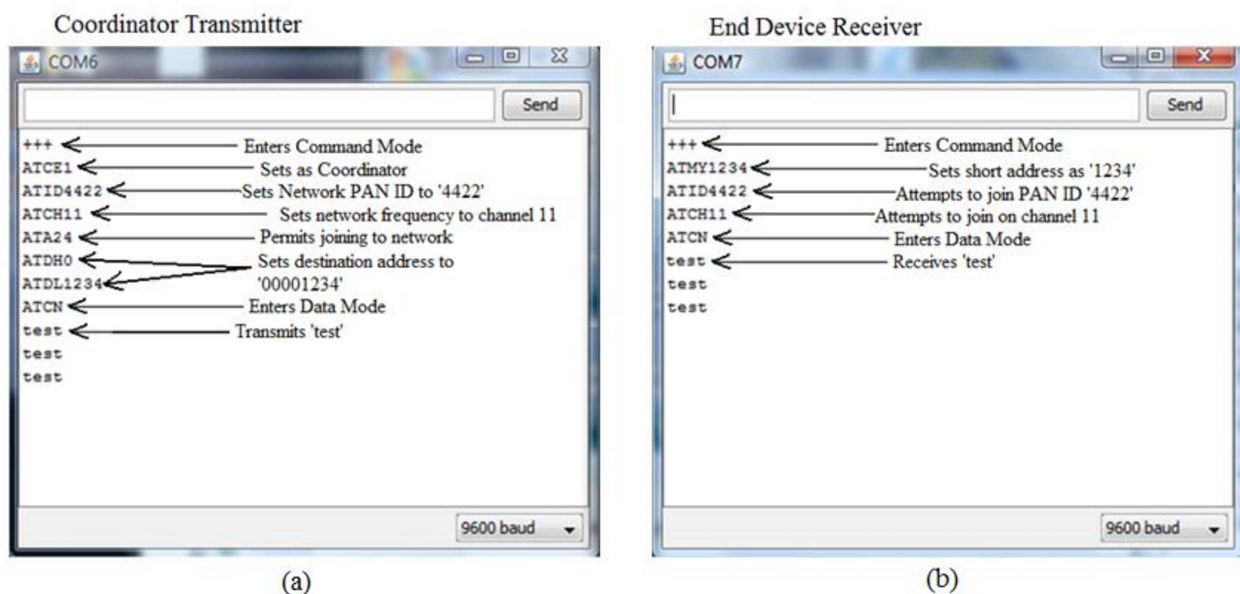


Figure 26: The coordinator transmitter (a) and end device receiver (b) performing a successful selective transmission is witnessed in the virtual COM port.

It can be seen that the channels and PAN IDs of each XBee were programmed to match. The source address in the receiver code matches the destination address of the transmitter code. From this success, it seemed logical that multi-hopping could be performed by sending a packet with an address in the header, parsing out the address, adapting the destination address with the header address, and re-transmitting the payload.

Since the single hop was performed successfully, modifications to the code were then made to try and perform a multi-hop. The coordinator transmitter, shown in Figure 27 (a), was configured to send a packet with a destination address embedded in it. The receiver code was modified as an intermittent node that could parse the destination address from the payload that was supposed to be

sent to the destination. The destination address was then used to adjust the destination address of the intermittent node, shown in Figure 27 (b). The payload was then transmitted with the intention to reach a third XBee device, shown in Figure 27 (c), configured to be a participant in the same network and to have the same address that was parsed out of the initial transmission. Although, the address parse was successful, the third node did not receive the packet addressed to it as seen in Figure 27 (c). The unsuccessful multi-hop using the XBee Series 1 modules can be seen in Figure 27.

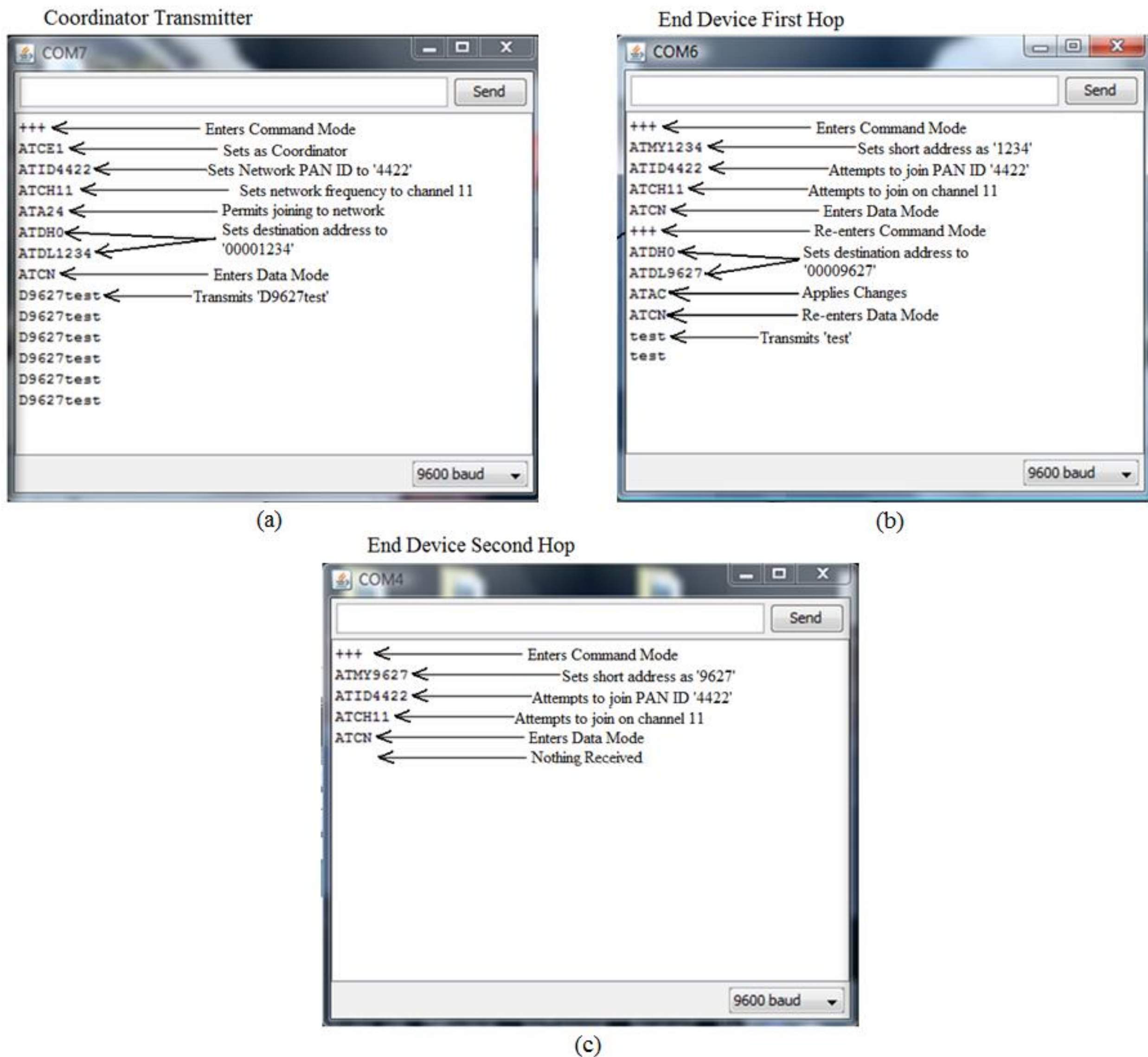


Figure 27: The first hop end device (b) was able to successfully parse the address from the destination payload sent by the coordinator transmitter (a). The second hop end device (c) did not receive it.

From this trial, the decision was made to switch to the XBee ZB Series 2 modules. This was due to the fact that the ZigBee stack already includes multi-hopping. It was evident from this simple test that the time spent working on achieving a multi-hop and implementing it into the system with 802.15.4 could be better spent. There were more pertinent problems that had to be pursued. The built-in multi-

hopping that the ZigBee stack provided freed up time to work on the radio resource management and cognitive elements that the objective of this project dictates.

3.7.3. Conclusions

The choice between the UZBee dongles and the XBee development boards is a major turning point in this project. Below in Table 9, a comparison can be seen between the two devices.

Table 9: Pros and Cons of Hardware Choice

UZBee		XBee	
Pros	Cons	Pros	Cons
Memory	Ease of Installation	Ease of Installation	Memory
	Real-World Use	Real-World Use	
	Complex Coding	Friendly Code Interface	
	Data Rate	Data Rate	

In this project time is short so the ability to install and learn the interface quickly is essential. The main problem with using the XBee would be the memory available in the system. This problem can be overcome through careful coding. The most important characteristics are the ease of implementation on the XBee, the portability, and the independence from a computer.

3.8. Firmware

Firmware is base level code to internally operate a hardware device. Hardware companies that manufacture electronics, which are used for development or original equipment manufacturing, sometimes offer multiple versions of firmware for a specific electronic device. This provides developers flexibility when interfacing electronics. Different firmware version can provide different levels of functionality and varied compatibility. Expanded capability is typically at the cost of complexity. There is really no clear distinction between firmware and software except for the distinction in that firmware generally covers the lower end functionality, while software builds upon it.

Since the projects goal was to implement a software-defined cognitive engine for a radio with radio resource management, the selection of firmware was crucial. This is because the firmware, being what the cognitive software was to interact with, defined the limits in the control the software had in the system. Selecting the correct firmware for the Xbee Series 2 ZB hardware to achieve the objectives of

this project was a process of elimination based on which functionality was necessary and which was of no use to the project.

3.8.1. Transparent Operation Firmware

Transparent operation refers to the ability of the firmware to allow a user to use a device for communication as though it were not present. This definition seemed contradictory to the objective of the project from the beginning, because it inhibited the ability of the software to learn about and adapt the operation parameters of the radios. However, the implementations of the transparent firmware for the UZBee dongles, and the XBee modules were not as restrictive as the definition might make them seem. The XBee Series 1 802.15.4 modules, which were used initially before the XBee Series 2 modules, were configured to transparent operation by default.

The transparent firmware designed by Digi for the XBee Series 1 modules provide some capabilities to adapt certain hardware operation. The transparent firmware offered by Digi was called the AT firmware because the commands follow the Hayes command set. This firmware offers two distinct modes called data mode and command mode. When in data mode, every packet sent to the buffer of the radio was interpreted as transmission data and was transmitted when the radio had the opportunity. The second mode is called the command mode. This mode is initialized by sending a '+++', a carriage return and linefeed to the radio. In the command mode a number of functions could be performed. Some of these functions were useful in implementing a cognitive radio, such as the ability to read the RSSI of the last packet sent, the ability to change the transmission power level of the radio and the ability to perform node discovery. Some of the implementation and testing even began on the XBee Series 1 radios using this firmware before the XBee Series 2 radios arrived.

3.8.2. Data Parsing in Transparent Operation

Data parsing is essential for designing a system which uses transmissions from other nodes in the network as feedback for optimization. Transparent operation made this very simple as all the data received from transmissions are stripped of all headers and only the payload is sent the microcontrollers buffer. To parse data, the router first reads in the serial information and stores it in an array. The received packet has four cognitive parameters and a node address. A typical packet received in transparent operation can be seen in Figure 28.

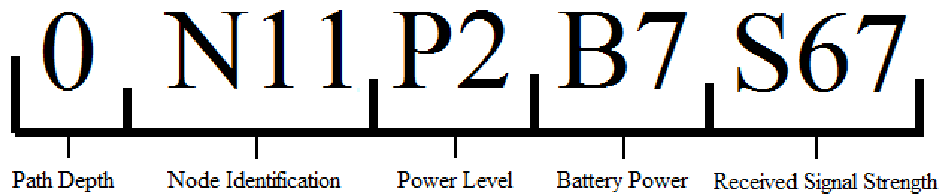


Figure 28: This is a typical header to communicate identification, power transmission level, battery power and received signal strength

Without any header or frames to identify, parsing the packet was very simple. The example above is in the same format as the packets that were used during the data parsing tests in transparent mode. The first character in the header is the path depth. There is only one instance of the path length in the transmitted data. While transparent operation was used, the router read this first character as an integer and then incremented the path depth. It then updated this header to indicate that it was from the next router in the path for when it was re-transmitted. The coordinator uses the path depth to indicate how many sets of cognitive data it will need to parse. This was originally placed in the first byte because the code was originally going to be controlling the multi-hopping. Since the ZigBee built-in multi-hopping was adopted later, path depth was adapted to links assessed and was appended to the back of the header as it was easier to locate this way.

Originally, the next three characters were the letter 'N' and the two digit node address. This is the unique address assigned manually to each node. The coordinator uses these node addresses to identify the different nodes in the system when selecting the multi-hopping path. The node address was then changed to a byte long node identifier to save space in the payload which is limited by 84 bytes.

The next two characters are the letter 'P' and a one digit transmission power level. There are five different transmission power levels at which the both XBee modules can transmit. While the transparent firmware was being used to implement the system, this power level was set at each node by parsing the RSSI and deciding if the power level needed to be increased to promote a better signal or if it could be reduced to save power. Originally, the path length was important for this operation as it was used to find the address of the node closest to the receiving router in the path using this equation

$$Closest\ Node\ Address = 10 * Path\ Depth + 8. \quad (17)$$

Since the new firmware has been adopted, the coordinator decides the minimum required transmit power level to accommodate the QoS of every link used for each router. This is because the transparent firmware or API firmware is not capable of changing the power level fast enough to do so for each node a router communicates with.

The next two characters are the letter ‘B’ and a one digit battery level. The battery level is measured at each node and logged as its own cognitive parameter to be interpreted later as a cognitive parameter for the routing algorithms. Since the update in firmware, the battery life is sent only to the coordinator as the other routers have no use for this parameter and the cognitive data no longer needs to traverse along an unknown path. The ZigBee built-in multi-hopping and permanent coordinator address allowed for a simpler approach which will be described in implementation.

While using transparent firmware, the final three characters were the letter ‘S’ and a two digit RSSI reading. This RSSI is assumed to be a negative value with the unit dBm. The RSSI is read directly from pin 6 on both versions of XBee modules. The pin outputs a pulse width modulated signal (PWM) that is interpreted at the node and logged as its own RSSI. When ASCII characters were used to denote RSSI, all packets received below sensitivity, or below -92dBm, printed “No” rather than a two digit number to indicate that there is no connection present. Since the decision was made to use one byte for RSSI, the byte 0xFF is used instead. If the PWM duty cycle was near 100% it was printed as ‘Mx’ to indicate maximum signal strength. The byte 0x00 is now used to indicated a packet received with power greater than -51dBm.

Each router appended its own header of cognitive parameters onto the end of the received packet. This along with the updated path length was sent to the XBee buffer as the new packet for transmission to be interpreted at the next node. The test using the transparent firmware proved the capabilities of the transparent firmware in data parsing. An end device without adaptive transmission power was configured to output a constant packet with the parameters seen in Figure 29.

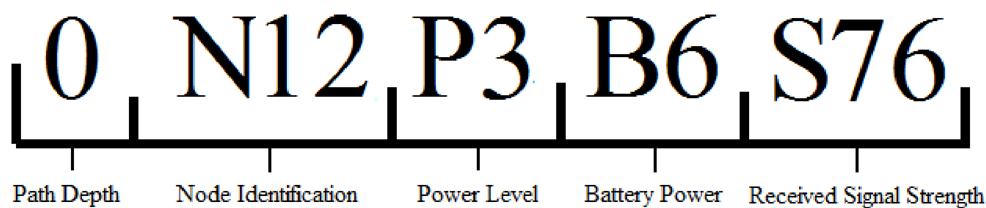


Figure 29: This the test packet sent out by the end device in order for the router to parse, append its own data to and re-transmit.

In transparent operation, the XBee needs to be configured in command mode. Command mode is enable by sending the radio ‘+++’, a carriage return and a linefeed. The XBee was set to enable sleep mode with the ATSM1 command, write low to pin seven using the ‘ATD70’ command, enable RSSI readings on PWM0 using ‘ATRPFF’, reduce the power level to -10dBm using ‘ATPL0’ and re-enter data mode using ‘ATCN’. When the XBee re-entered data mode, it received the test packet. The router

incremented the path length of the packet by one and updates the array. This is to indicate that it is the first re-transmission of the data. The cognitive parameters of the previous node are preserved and the cognitive parameters of the router are appended to the end of the packet. The packet is then retransmitted. The RSSI of the device is reported "Mx" when both devices were in close proximity. As the router was slowly moved further and further from the end device and barriers were introduced "No" was eventually reported. The capabilities of the transparent firmware to parse the packet and retransmit it with its own data are displayed partially in Figure 30.

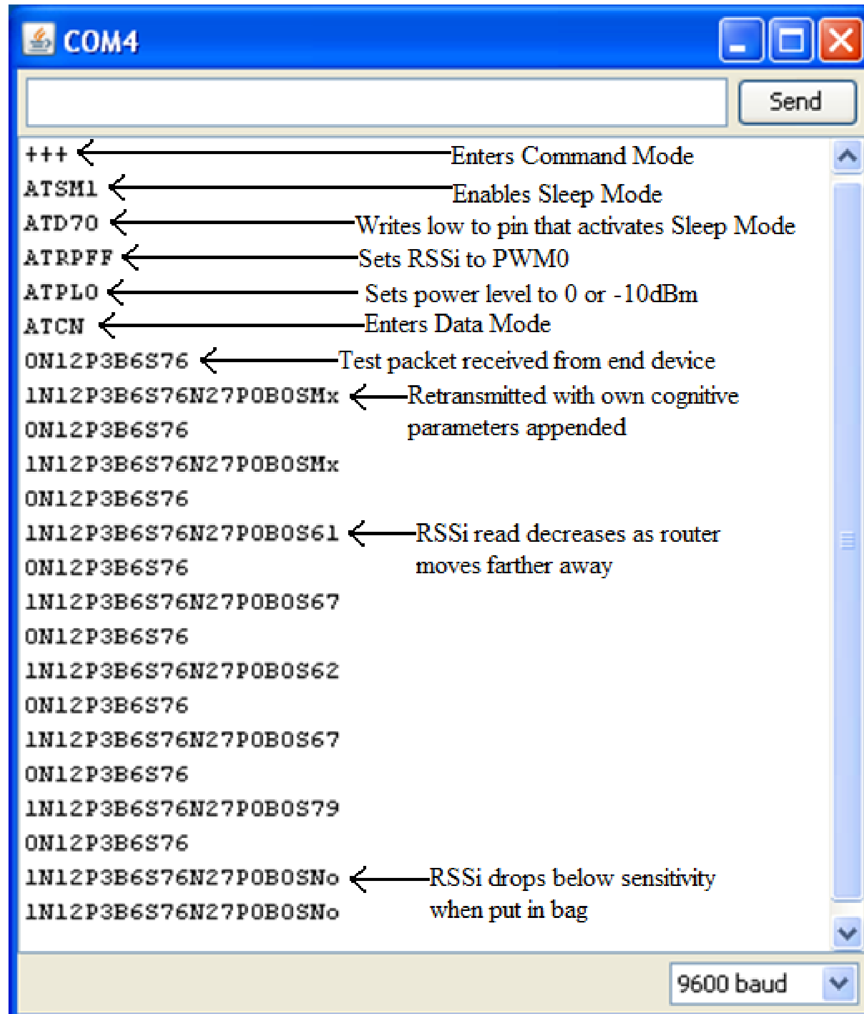


Figure 30: The packet from the end device was parsed and measured for RSSI. The router then appended its own cognitive data to the back of the packet and retransmitted it.

The router using transparent firmware was proven capable in reporting its own cognitive parameters such as RSSI, updated path length, and retransmitting packets. The simplicity of implementing this data parsing made the decision to switch firmware difficult. However, as the limitations of the transparent firmware such as the slow transmission power adjustment, the necessity

to harness the expanded capability of the API firmware began to take precedent over the convenience of the transparent firmware.

3.8.3. Altering Transmission Power in Transparent Operation

Altering transmission power based on RSSI, one of the goals of this project, is feasible in the transparent operation firmware. The received signal strength measurement was used to cognitively alter the transmission power of the device sending the signal. To find the RSSI of the last packet received, the AT command 'ATRP' is sent to the XBee buffer in command mode. It is followed by a byte that defines the length of time the RSSI could be read after the last packet was received. This command allows for the RSSI to be read on pin 6 or PWM0. The software used to adapt the PWM signal into dBm is in the implementation chapter. The RSSI measurement read by one node is sent back to the transmitting node via a header. Since in transparent operation the received data is sent to the serial buffer of the microcontroller alone, parsing the data was simple. This data was read in and parsed using if loops looking for each of the pieces of data by their given letter markers. For example, the RSSI was positioned in the packet after an 'S' and contained two numerical ASCII characters representing the positive equivalent to the negative RSSI value. The parsed RSSI values transmitted back and forth can be seen in Figure 31.

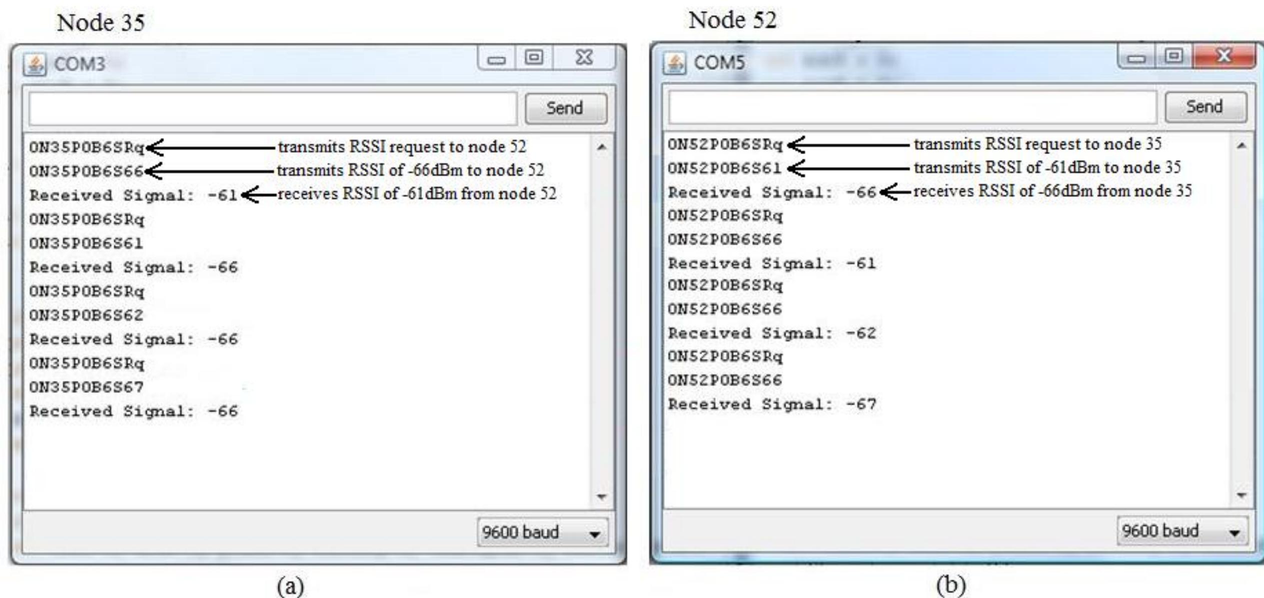


Figure 31: Node 35 (a) and node 52 (b) send out a RSSI requests to each other. Both nodes reply with the measured RSSI values of the request packet. Both nodes receive this data and parse out the reading.

To use this parsed RSSI measurement, boundaries were set in the code as constants so they could be easily changed later. Based on whether the RSSI was higher, lower, or within the boundaries the transmit power level could then be changed accordingly. In order to change the power the AT command

'ATPL' is used in the command mode followed by a byte from zero to four representing the power level to set. The AT commands used to change the power level can be seen in Figure 32. The '+++' to begin command mode required a second of setup time and a second of hold time to stabilize the system. The 'ATCN' command exited the command mode and sent it back into data mode. The two seconds required to enter command mode was a cost on the speed of the system, which was a consideration in the final decision to switch to API firmware.

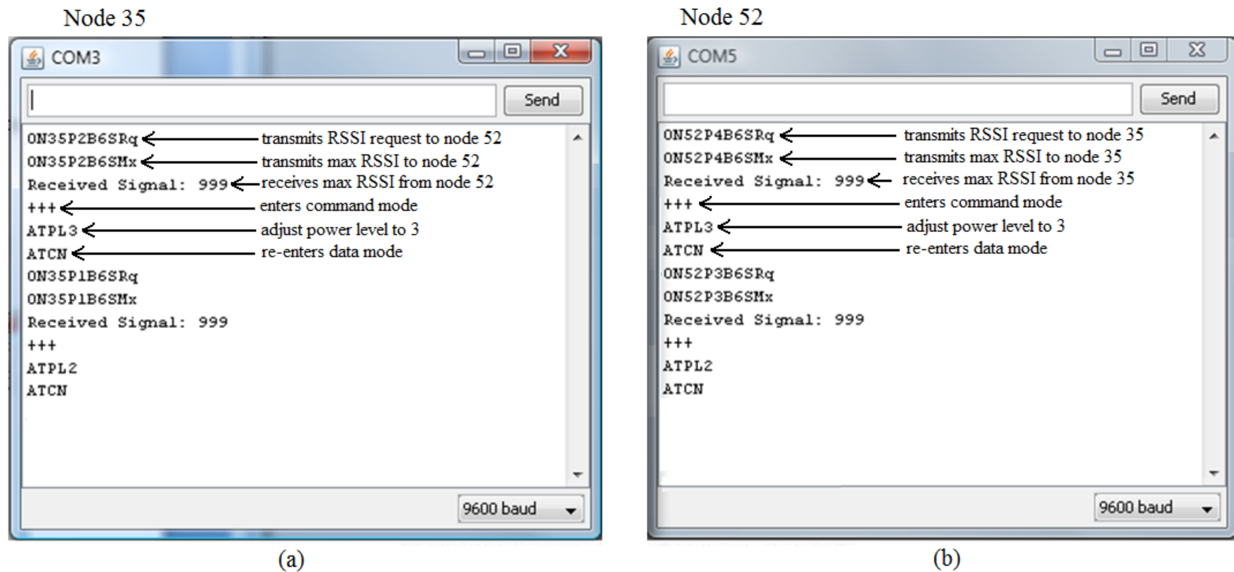


Figure 32: Node 35 (a) and node 52 (b) find receive the RSSI of their transmitted RSSI request packet. With this information, they both adjust their power level down one step because the RSSI received was above the upper boundary.

The limiting minimum 2 second delay before any physical layer adjustments, including the power level, was perceived as a nuisance and could have been worked around. However, the decision to switch to API firmware was still made. This was due to the limitations in multi-hopping that specified by the Digi design team. Fortunately, this limitation was identified early and avoided.

3.8.4. Multi-Hopping with Transparent Firmware

As explained in the hardware section, multi-hopping, a necessary component to an adapted ad hoc network was attempted with the transparent firmware on the XBee Series 1 802.15.4 radios. This attempt used a short header with a destination address that was to be parsed out in order to adjust the destination address of the intermediate hop. The intermediate hop parsed out the address and changed its destination but the destination node never received the payload intended for it. The combination of the burden in implementing and integrating multi-hopping on the 802.15.4 platform and the built-in multi-hopping of the ZigBee stack made switch to the XBee Series 2 ZB modules simple.

This attempt was indicative of the complications that can arise with supposedly simple implementations. Because of this, the routing capabilities of the XBee Series 2 ZB modules were read thoroughly. Before writing any code for the newly purchased XBee Series 2 ZB modules, it was discovered that multi-hopping could be performed with these radios. However, the transparent firmware only allowed for the automated ad hoc on-demand distance vector, broadcast, and many-to-one routing which restricted the capability to create customized source routes. Since the routes had to be defined by the cognitive system, these three routing methods were of limited use. It was found that if the API firmware was installed, then a third multi-hopping method called source routing could be used.

3.8.5. API Operation

API stands for application programming interface. A system designed to interface with an application is better suited to meet the needs of the objective of this project. The XBee Series 2 API firmware had many distinct advantages over the transparent firmware. The API firmware is a frame based system. A typical transmit request frame is depicted in Figure 33.

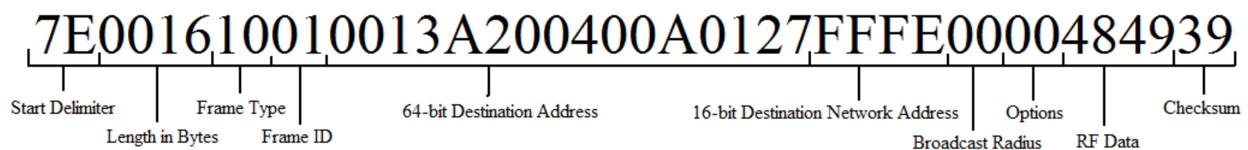


Figure 33: This is an API transmit request frame.

There are seventeen frame types that are interpreted by the firmware as different classes of operations. The frame type of a frame is identified by the frame type byte that is always the byte offset by three. In Figure 33, the frame type for transmit request is 0x10. All the frame types are listed below in Table 10.

Table 10: List of API Frame Types (from [15])

API Frame Names	API ID
AT Command	0x08
AT Command – Queue Parameter Value	0x09
ZigBee Transmit Request	0x10
Explicit Addressing ZigBee Command Frame	0x11
Remote Command Request	0x17
Create Source Route	0x21
AT Command Response	0x88
Modem Status	0x8A
ZigBee Transmit Status	0x88
ZigBee Receive Packet	0x90
ZigBee Explicit Rx Indicator	0x92
ZigBee IO Data Sample Rx Indicator	0x92
Xbee Sensor Read Indicator	0x94
Node Identification Indicator	0x95
Remote Command Response	0x97
Over-the-Air Firmware Update Status	0xA0
Route Record Indicator	0xA1

The firmware is able to interpret these frame types quickly and without the need for the different modes as in transparent operation. There is no need to switch from data mode to command mode, so the program could potentially send the transmit request frame in Figure 33 to send data then immediately switch power using the following frame from in Figure 34.

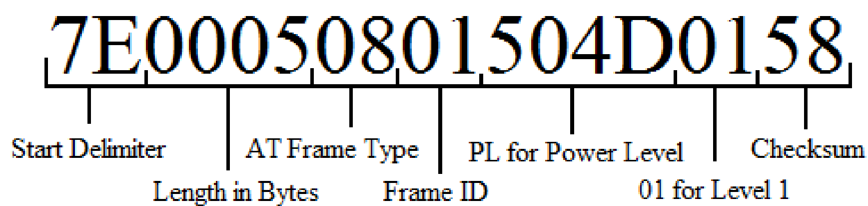


Figure 34: This API frame adjusts the power to power level 1 or -2dBm.

Most importantly, the source route of a node could be adjusted simply and quickly using the create source route API frame. This frame uses short 2 byte addresses for the intermediate hops. This

makes the transfer of source routes from the coordinator to the routers extremely efficient as the payload only requires 2 bytes per node in the route. This functionality is truly ideal for generating an independent routing technique from those built-in the ZigBee stack. An example of the create source route frame that would be used to define the path of hopping in an ad hoc network is in Figure 35.

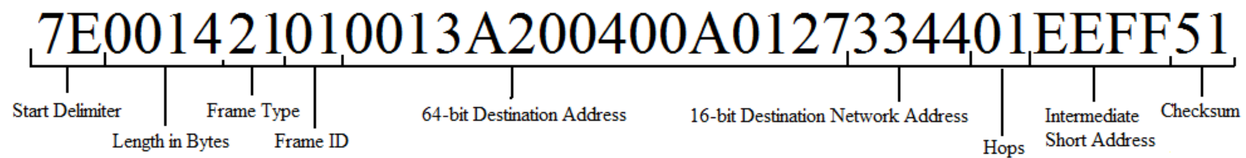


Figure 35: This is an API frame that sets a route through node EEFF to node 3344.

3.8.6. Conclusions

The API firmware was selected to implement the cognitive system because it allowed source routing. Aside from that the command mode and data mode are integrated so that system adjustment, which is important for cognitive radio, is smoother, more reliable and faster. The API frames also provided more detail about received packets. These features are at the expense of increased complexity and the necessity to adapt the parts of the cognitive system already implemented using transparent firmware. These setbacks and inconveniences were insignificant compared to the flexibility of the API platform.

3.9. Chapter Summary

The network that we want to design has a high signal quality, updates its routing paths frequently and quickly, and promotes a long battery life for every node in the system. For this reason an energy spreading network with a strict QoS limit on RSSI and BER will be required.

After investigating a variety of cognitive parameters and radio resource management techniques, it was decided that the cognitive system would use RSSI, BER and battery capacity as cognitive parameters. These parameters were to perform transmission power optimization and routing that would preserve battery life without sacrificing link QoS like BER and RSSI. Narrowing down the hardware and firmware platforms to implement this system was another challenge.

The investigation of the potential hardware and firmware combinations for feasibility in creating the power conscious cognitive engine arrived at the Xbee Series 2 ZB modules with API firmware. The Xbee Series 2 ZB modules were simple to use and program, had the capability to adjust power and read RSSI, and had built-in multi-hopping. The API firmware for these boards provided the ability to create custom multi-hop routes and allow for smooth transitions between system adjustment and data transmissions.

4. Prototype Implementation

It is necessary to deliver the cognitive parameters of each router to the coordinator for evaluation, power setting, and routing. There are several challenges in implementing this method. Each node has to effectively communicate the link quality by means of RSSI and BER of each feasible single hop link it could use in a route. Each node also needs to include its own battery capacity and node ID. Second, the process needs to be relatively fast since the routers are also required to send packets of sensor data to the coordinator during normal operation. It is crucial to finish optimization quickly so that the primary functionality of the RF modules may be used. Finally, the process must be power efficient. It has to complete the diagnostic period with as few transmissions as possible to save power.

4.1. Node Discovery Request

The first step to optimization is a transmission to all of the nodes in the network to synchronize the node discovery process. To initiate the cognitive parameter generation and feedback, the coordinator must broadcast a node discovery request. This is important because all the routers must know when it is time to begin the process. The coordinator request is simply a packet with two bytes "3131" or two ASCII ones to signify mode 1 or node discovery. This is done using an API frame similar to the one in Figure 33 but with the broadcast address "000000000000FFFFFFFE" to reach all the routers in the network and the payload "3131". The routers are programmed to interpret this transmission as the beginning of the node discovery process. The coordinator sends out this node discovery request periodically to initiate the cognitive feedback, so it can update the source routes frequently. The coordinator is seen broadcasting the node discovery request to all the routers in the network in Figure 36.

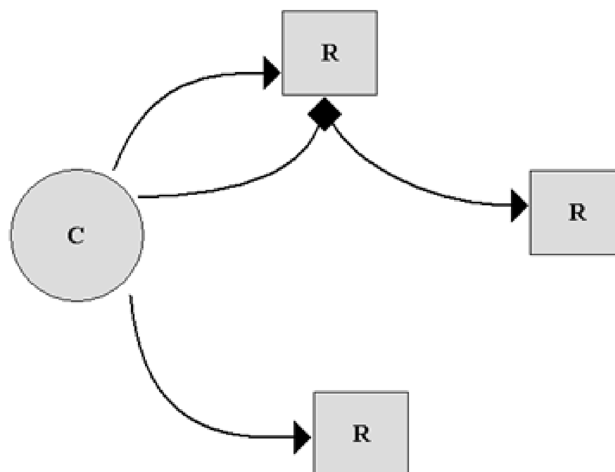


Figure 36: The coordinator broadcasts a node discovery request to initiate the cognitive parameter generation and feedback

4.1.1. Node Discovery

Once the routers know it is time to start the cognitive feedback process, the routers and the coordinator perform a node discovery sequentially in order to assess all the feasible single hop links in the network. Before each node performs their node discovery, the maximum broadcast hops are adjusted using the broadcast hop adjustment AT command. The command is similar to the API frame in Figure 34 except the ASCII “PL” is changed to “BH”. This adjusts the node discovery to only find the nodes that are a single hop away from the aggregator. The node discovery timeout command was also modified to adjust the time a node took to perform node discovery. Originally, it was reduced to the minimum value 3.2 seconds from default timeout period of 6 seconds since the node discovery can be used to find nodes that are more than a single hop away. However, the value was then adapted to 8 seconds to make it easier to monitor the node discovery and to make the discovery more consistent in finding the surrounding nodes. This also uses a similar API frame as Figure 34, except with “NT” instead of “PL” and “08” instead of “01”.

After these adjustments are made, node discovery is performed using the node discovery AT command built into the API firmware. The node discovery is initiated by using a packet similar to Figure 34 except the “PL” is “ND” and there is no byte “01”. The node discovery works by sending out a broadcast that is received by ZigBee nodes associated with any network. Each node that receives the node discovery waits a random period of time within the timeout period set by the node discovery timeout AT command described before. It then sends back its node discovery acknowledgement that contains its short address, its serial number, its node identifier, its parent network address, its device type, its status, its profile ID and its manufacturer ID. In the meantime, the node that performed the node discovery polls for address then stores all short address and serial numbers of all the neighboring nodes in a node table for 8.25. Each node performing a node discovery to find all of its single hop neighbors can be seen in Figure 37.

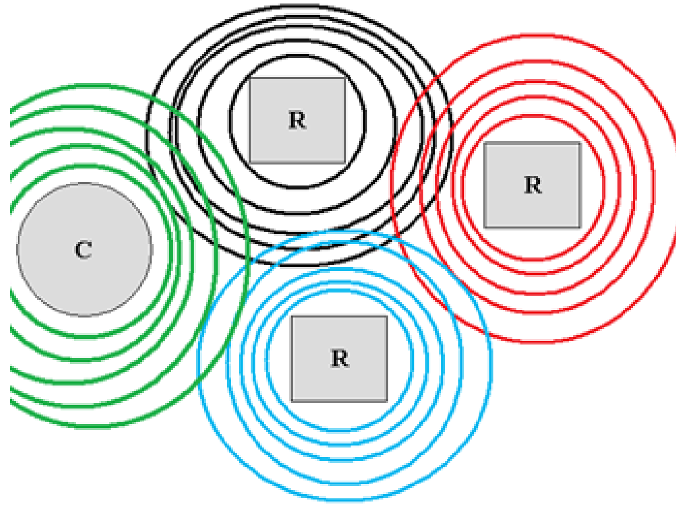


Figure 37: Each node performs a node discovery to find all the feasible links the coordinator can use while performing the routing algorithm

Originally, every node performed its node discovery at the same time. This was later adapted because the simultaneous node discovery broadcasts interfere with one another and end up jeopardizing the reliability in terms of finding nearby nodes. The system now requires each node to wait 8 times its ASCII node ID as seen in Figure 38. Once a node completes its node discovery it waits until 34 seconds after the node discovery request was sent out as depicted in Figure 38.

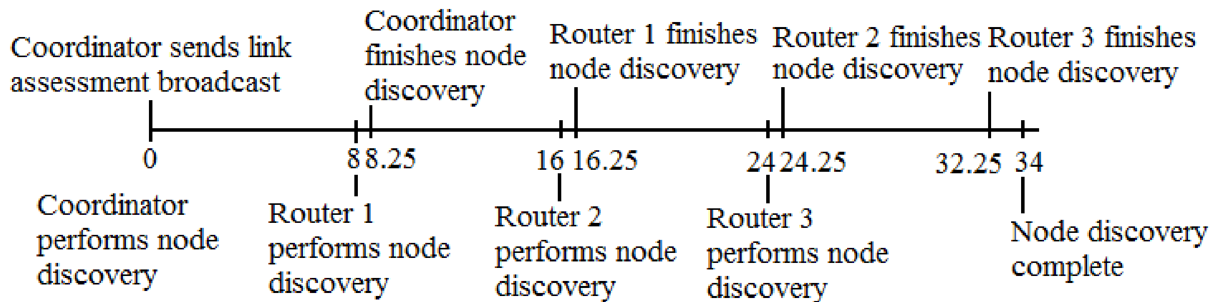


Figure 38: Each node's node discovery is space out by 8 seconds to ensure that all the feasible single hops are found.

4.2. Link Assessment

The next step in the process is to evaluate the single hop links between all the neighbors in the network. To do this, link assessment requests are transmitted to all of the addresses that were recorded during the node discovery. This step is intended to isolate the RSSI and the BER in every connection of the system. This data will be used in by the coordinator for power setting and routing purposes. The nodes transmitting link assessment requests to all their neighbors found in node discovery can be seen in Figure 39.

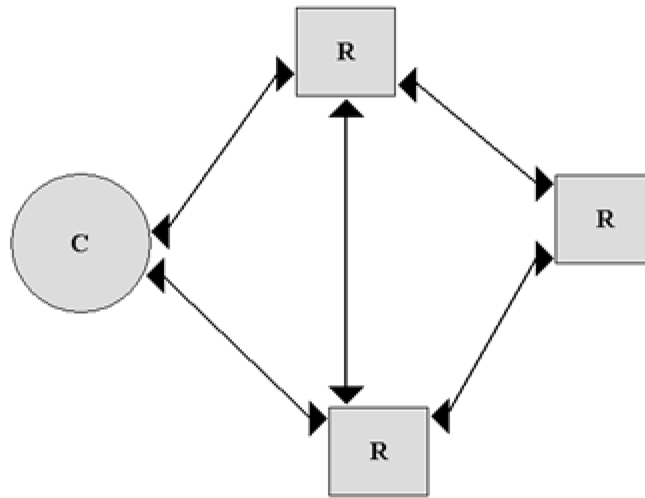


Figure 39: Each node sends out a link assessment request to all the nodes on its node table.

For each single hop neighbor discovered by a node in the node discovery. Two link assessments are sent using a similar transmit request API frame as Figure 33. The address corresponds with those in the node table. The maximum hops byte is set to “01” so that the RSSI and BER represent the link between the source and destination nodes. If the maximum hops byte is set higher, then the RSSI and BER could be generated by the intermittent hops retransmitting the link assessment and not by the original transmission from the source. The payload of a link assessment request can be seen in Figure 40.

Figure 40: Each node sends 25 ASCII G's for a BER link test, its ASCII node ID, the transmit power level and the mode 2 request in each link assessment request payload.

Two link assessment requests are sent. They are identical except the first is sent out at power level one or -2dBm, and the second are sent at power level three or 2dBm. This is done so that the coordinator can decide the correct power level to meet the QoS requirement. Each node takes 20 seconds to send out all its power level one link assessment requests as seen in the first 20 seconds of Figure 41. The node will split up the 20 seconds by the number of nodes it found and send out link assessment requests at random time within these divided time slots to each node on its node list. It then repeats this process in the next 20 seconds for the power level 3 link assessment requests as seen in the last 20 seconds of Figure 41. Meanwhile each node is also receiving and parsing link assessment requests from other nodes nearby.

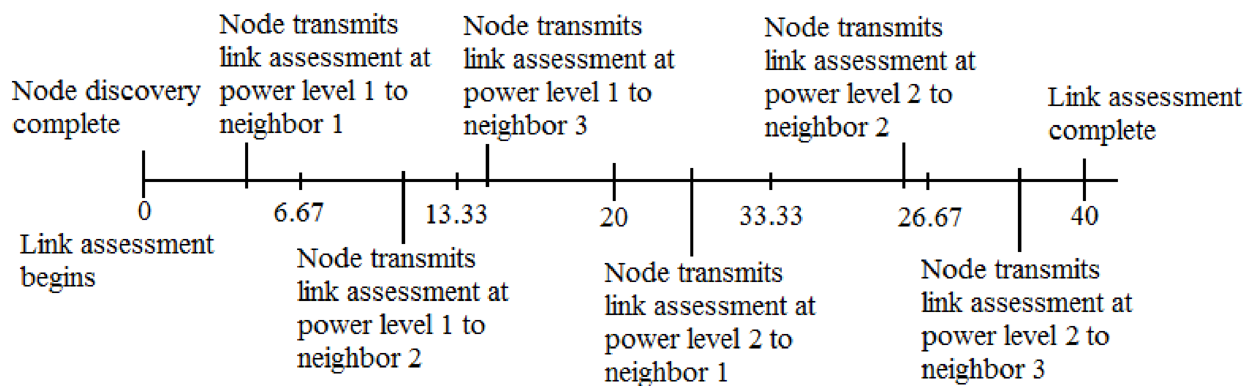


Figure 41: Each node sends its power one link assessment requests in the first 20 seconds and its power 3 link assessment requests in the next 20 seconds.

When the node receives a link assessment request and parses out the mode, it derives BER and RSSI of the link. It then stores it into a table where it is associated with the node ID of the link assessment request. This is so the information can be passed on to the coordinator to perform the routing algorithm and power setting.

4.2.1. BER Check

To find the BER of a link 25 G's are sent to all of the addresses found in the node discovery in each of the two link assessment requests. All messages are evaluated when received to check for any bits that may have been corrupted in transmit.

4.2.1.1. Transmitted Packet

The easiest way to calculate BER without adding parity or redundancy is to check a received message against a known, expected message. Therefore a payload including twenty-five repetitions of the ASCII character 'G' was elected to be transmitted in the link assessment requests. The number twenty-five was chosen to meet the acceptable dynamic range, while allowing for reasonable processing time.

Twenty-five bytes in repetition translates to two hundred bits to check. This means two things. It first means that the step size of the BER check is half a percent. Second, checking two hundred bits means is there will be approximately four hundred computations made per packet. This is due to the fact that there are two computations made per bit.

4.2.1.2. Received Packet

When the node receives a link assessment request it calls the method `berCheck(i)`. The variable '`i`' corresponds to the origin node of the link assessment. The method then begins two nested

for loops. The first loop cycles twenty five times takes one character from the link assessment at a time and performs a bitwise exclusive or (XOR) operation against the hexadecimal equivalent of 'G'. This XOR returns a logic one at every bit where there was a mismatch. The second loop cycles eight times. Its operation is to perform a modulo-two operation on the byte. This will return a one if the least significant bit is a one. The loop then shifts the byte to the right by one bit. This deletes the least significant bit and replaces it with a new bit to be evaluated.

Every time that a one is found from a modulo-two command, a variable called "errors" is incremented. When the two nested loops have finished running the total errors are then returned. The BER is now any byte between zero and two hundred.

4.2.2. RSSI Extraction

In the XBee manual [**18**], it was discovered that pin six on the board was used to send the received signal strength indicator (RSSI) as a pulse width modulated (PWM) signal. Thus the first goal of to extract RSSI is to acquire this signal and convert it back to a digital RSSI reading.

When a link assessment packet is received by a router, it counts the total number of clock cycles and the number of high logic level counts over three pulses or six logic level changes of the RSSI PWM. Using these two numbers a duty cycle is approximated which would correspond to the RSSI. In the manual, it states that the duty cycle of the PWM is correlated to a number of decibels above the sensitivity point of the XBee module. Knowing that the sensitivity of the module used is -92dB, Table 11 is utilized to generate a formula that directly relates RSSI to duty cycle of PWM0.

Table 11: PWM Percentages (from [18])

dB Above Sensitivity	RSSI (dBm)	PWM Percentage (High/Total)
10	-82	41%
20	-72	58%
30	-62	75%

As confirmed by Table 11, the equation for finding RSSI from a PWM0 duty cycle is [**22**]:

$$RSSI = \frac{(10.24 * Dc_{PWM}) - 295}{17.5}. \quad (18)$$

The only problem with this approach is that it makes the program get stuck in an infinite loop if no packet is received or if the received packets RSSI is above -51dBm. To fix this bug, an 'if' loop is

inserted that checked if the total number of cycles within six logic level changes exceeded the predicted value of [**22**]:

$$3072\ counts = \left(\frac{3*64\mu s\ pulse}{62.5ns\ clock\ \ cycl\,e}\right). \tag{19}$$

In order to allow space for slight error, 3072 is rounded up to 3100 counts. After 3100 counts in the loop, the variable `maxR` is set to one if the PWM is at a high logic level or `minR` is set to one if the PWM is at a low logic level. If `maxR` is set high than the RSSI is returned as 0x00 and if `minR` is set than the RSSI is returned as 0xFF. Later in the program when RSSI is printed, these variables are used to print `MxS` (Max Signal), if the received packet RSSI is greater than -51dBm, or `NoS` (No Signal), if the received packet is below -92dBm.

4.2.3. Cognitive Data Packet

All of the cognitive data that the coordinator needs must be sent in order to perform power level selection and route optimization. The routers send the cognitive parameters from each of their received link assessment packets to the coordinator. These link quality parameters include node ID, derived RSSI, calculated BER, and power level. The routers also send their own battery life and node ID as well as two ASCII threes that act as the mode 3 request. Every router compiles and sends a packet to the coordinator similar to Figure 33 except the destination address is set to the coordinator address "0000000000000000000". The payload of the transmission can be seen in Figure 42.

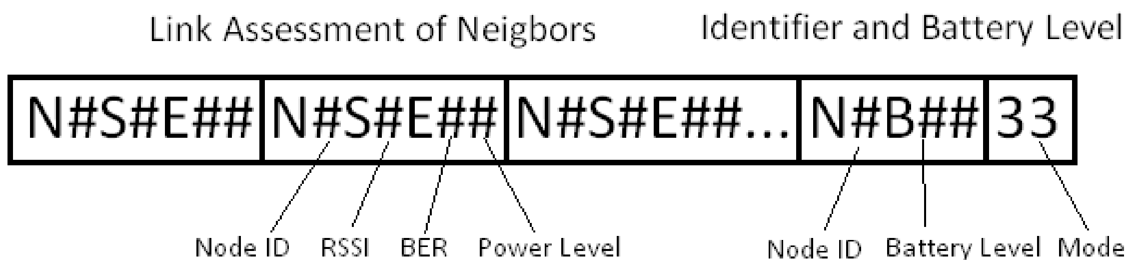


Figure 42: The cognitive data is sent back with a comprehensive list of all the parsed link assessment requests it received.

After the link assessment has been performed, the coordinator waits twelve seconds to receive the cognitive data packets from all the routers in the network. It will send a message to the routers confirming a successful transmission upon reception. If the router has not been alerted of a successful transmission, it will automatically send a redundant packet after five seconds and waits for the acknowledgement again. It sends a final packet if no acknowledgement is received again. The routers sending their cognitive data back to the coordinator can be seen in Figure 43.

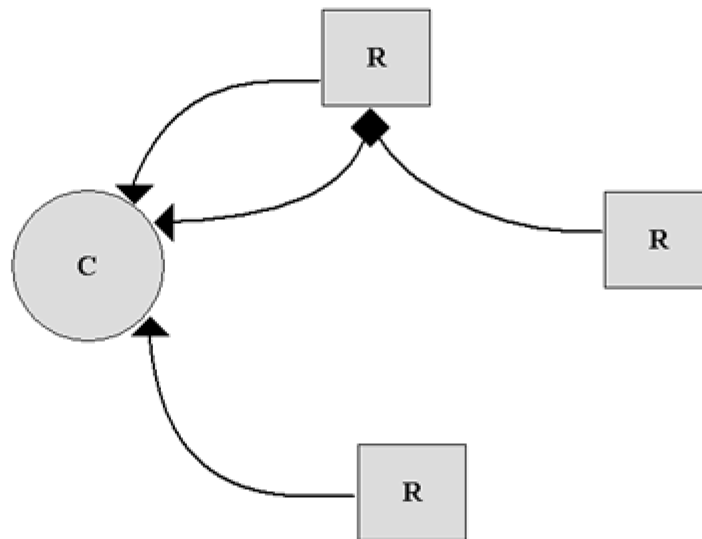


Figure 43: The routers send the cognitive data collected on the links in the link assessment and the node's personal battery level and ID to the coordinator to perform the routing algorithm.

4.3. Routing Algorithm

Two different routing algorithms are available for the cognitive network. Each has a different goal by using a different assumption. The first algorithm assumes that transmission power has an effect on the current drain of the transmitter. This algorithm attempts to optimize the network by limiting the transmission power of nodes and identifying a feasible path that can be used under these power conditions. The second algorithm works under the assumption that the transmission power has little to do with the current draw. Its goal is to restrict the number of paths a node can be included in and thereby lowering the amount of time that the device is in operation.

4.3.1. Transmission Power Limiting Algorithm

This routing algorithm works by assigning a cost for every connection in the network; the higher the cost, the less lucrative the connection. The cost for the connection is a combination of the BER, RSSI, battery level, and change in transmission power. Then, as a path is evaluated, the highest cost connection in any given route becomes the cost for the route. This means that any path is only as strong as its weakest link.

4.3.1.1. Cognitive Parameter Storage

When the cognitive data packet is received by the coordinator it needs to separate the data into useful tables. The way that this is accomplished is by creating three raw tables of data. These tables are `battArray[i]`, `rssiTable[m][i][j]`, and `berTable[m][i][j]`.

The first is table is a one dimensional array of the current battery levels of every node in the system. The address in the array corresponds to the node id which makes it `battArray[node id]`. An example of this using test conditions is shown below. The coordinator is set to a battery level of 100 because ideally this is a device that can work on AC power.

Table 12: Sample Routing Battery Levels Table

Battery Level Received of Nodes			
Coord	R_1	*R2*	*R3*
100	43	89	77

The second table needed for organizing the raw data is a three dimensional array of the RSSI of every link in the network. A sample table can be seen in Table 13. The coordinator needs to know which power level the test packet was sent at and the two nodes that are linked together. If a packet is not received it automatically sets the RSSI to 999. This is because the routing optimization algorithm selects the lowest cost to the system. Such a high value will never be selected. An example of this is depicted below.

Table 13: Sample Routing RSSI Table

All Received RSSI Values Table				
	Coord	R_1	R_2	R_3
Coord_PL1	X	70	88	999
Coord_PL3	X	53	63	83
R_1_*PL1*	70	X	81	57
R_1_*PL3*	53	X	59	53
R_2_*PL1*	88	81	X	67
R_2_*PL3*	63	59	X	54
R_3_*PL1*	999	57	67	X
R_3_*PL3*	83	53	54	X

The third table of cognitive parameters is a three dimensional array of the BER of each link. This is organized as `berTable[power level of test packet][sender][receiver]`. This will allow the coordinator to choose which of the values is more relevant to the conditions the node will be facing next. Similar to the table of RSSI values, if a BER value is not available from the parsed packet it will be considered 999. The BER values for this example can be seen in Table 14.

Table 14: Sample Routing BER Table

All Received BER Values Table				
	Coord	R_1	R_2	R_3
Coord_PL1	X	0	0	999
Coord_PL3	X	0	0	0
R_1_*PL1*	0	X	1	0
R_1_*PL3*	0	X	0	0
R_2_*PL1*	0	4	X	1
R_2_*PL3*	0	0	X	0
R_3_*PL1*	999	0	0	X
R_3_*PL3*	3	0	0	X

There is one more table of accessible information for the coordinator that is not parsed from the received packet. It contains the power levels set by the routing algorithm the last time it was used. This table will be used to calculate the energy cost to the system. The addresses in the array correspond to the node id. The values used in this example are seen in Table 15.

Table 15: Sample Routing Previous Power Levels of Nodes

Previous Power Levels of Nodes			
Coord	R_1	*R2*	*R3*
4	2	2	1

4.3.1.2. Finding a Link's Power Level

The first parameter that needs processing is the power level necessary for a given link. The way that this is calculated is by evaluating the RSSI readings from the two test messages. If the RSSI is less than the defined threshold the power level will increment. Likewise, if the RSSI is above the defined threshold the power level will decrement. To utilize five different possible power levels efficiently, this is evaluated at two power levels.

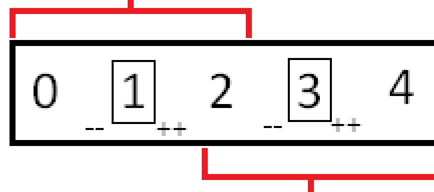


Figure 44: Powel Level Decision Based on Two Tests

This is accomplished by looking at the three-dimensional array `rssiTable`. The RSSI is first evaluated at the lower power level. This would be found in `rssiTable[0][sender][receiver]`. If the value satisfies the condition to increase the power, then `pL[sender][receiver]` is set to 4. If that is not the case and the condition to decrease the power is not met, `pL[sender][receiver]` is set to 3. Otherwise, `rssiTable[1][sender][receiver]` is evaluated. In this case if the device should neither increase, nor decrease its power `pL[sender][receiver]` will be 1. If the power needs to increase or decrease the `pL[sender][receiver]` is set to 2 and 0 respectively.

For the given table the RSSI levels per link, the 2-dimensional array `pL[sender][receiver]` would be the given values shown in Table 16.

Table 16: Sample Routing Power Level per Link Table

Power Level Needed per Link				
	Coord	*R1*	*R2*	*R3*
Coord	X	1	3	4
R_1	1	X	2	0
R2	3	2	X	1
R3	4	0	1	X

This array will be used for selecting which BER and RSSI measurements to use, to calculate the energy cost to the network, and will be evaluated when setting the final power level after routing.

4.3.1.3. Setting the BER and RSSI of a Link

There is a measurement of the BER and RSSI at each of the two test packets. Choosing which to use will be determined by the minimum power level that the node is allowed to use. If the minimum

power is zero, one or a two, the BER and RSSI found in the low power test packet will be applied to the link quality. Similarly, if the minimum power is either three or four, the BER and RSSI of the high power test packet will be applied to the link quality. For the given power level table, the following RSSI and BER tables, Table 17 and 17 respectively, will be used toward the link quality of the system.

Table 17: Sample Routing RSSI Parameter for Q

RSSI Parameter Applied to Q				
	Coord	R_1	*R2*	*R3*
Coord	X	70	63	83
R_1	70	X	81	57
R2	63	81	X	67
R3	83	57	67	X

Table 18: Sample Routing BER Parameter for Q

BER Parameter Applied to Q				
	Coord	R_1	*R2*	*R3*
Coord	X	0	0	0
R_1	0	X	1	0
R2	0	9999	X	1
R3	9999	0	0	X

As demonstrated above, any BER over the QoS threshold (2 in this case) is immediately quantized to 9999. This is to add an extremely high cost to the link such that it will never be picked as a desirable path. This was the case with two of the links above. However, if the BER did not meet the necessary QoS but the BER at the higher power level can (assuming the failed link is of the lower power level), the tables for applied BER and power Level are updated to the values of the higher power level.

4.3.1.4. Energy Cost

When the power level is changed there is a cost to the battery life of the system. This cost should be greater for a low battery. This is due to the fact that it is more beneficial, for the network, to lower the power level of a node with low battery than to lower the power level of a node with a full battery. The formula for the energy cost of the system is:

$$\text{Energy Cost} = (B_{MAX} - B + 1) * (P_M - P_0 + 4) \tag{20}$$

In this equation the variable "B" is the battery value received. It is subtracted from the maximum value that could be received and one is added such that it cannot go to zero. This is multiplied by the change in power levels. "P_0" is the power level assigned the previous diagnostic. The four is added to keep the energy cost as a positive value. The previous power level will be assumed to be four as an initial condition because that is the default configuration of the XBee chip. "P_M" is the newly calculated minimum power level that the node can operate at. An increase of power to the system will yield a high cost to the system. If there is no change to the power level the cost will be four multiplied by the battery drain. This allows the system to still assign a high cost to a low battery. Table 19 contains the energy cost of every connection in the network.

Table 19: Sample Routing Energy Cost per Link

Energy Cost Parameter Applied to Q				
	Coord	R_1	*R2*	*R3*
Coord	X	4	4	4
R_1	174	X	232	116
R2	60	445	X	36
R3	168	72	96	X

These energy cost values will be used to calculate the overall link quality of any desired connection in the network.

4.3.1.5. Link Quality

For the optimization routing algorithm, every link in the system will be assigned a numeric indicator of the performance of the link. The remaining cognitive parameters of each link will be used to calculate the link quality, of the desired connection.

$$Q = (\%R * RSSI) + (\%B * BER) + (\%C * EC) \quad (21)$$

This is a weighted combination of the three remaining parameters: RSSI, BER, and error cost. A high value of Q indicates a poor link quality. Q values for the given tables are depicted below in Table 20. It is arranged as a two dimensional array with every row representing a link. The address `links[i][0]` is the sender, `links[i][1]` is the receiver, and `links[i][2]` is the edge weight that will be applied to the routing algorithm.

Table 20: Sample Routing Q Values for all Links

Q Values for All Links in Network		
Sender	*Receiver*	Q
Coord	R_1	70.5
R_1	Coord	91.75
Coord	R2	63.5
R2	Coord	70.5
Coord	R3	83.5
R3	Coord	9999
R_1	R2	210
R2	R_1	136.625
R_1	R3	71.5
R3	R_1	66
R2	R3	171.5
R3	R2	79

The weight for BER in this demonstration is 100, the weight for RSSI is 1, and the weight for energy cost is 0.125. This sets the cost for each parameter to be equal at a range of approximately 0 to 100. These weights can be adjusted to change the importance of each parameter specific to any application.

4.3.1.6. Modification of Dijkstra's Algorithm

The routing used for the given link table is a modification of Dijkstra's algorithm. Rather than a summation of the costs in a path it sets the cost of a route to the highest edge weight in that path. This is important because rather than updating the node with `cost[sender]+Q` the algorithm needs to isolate the worst links in the path. To do this when a node attempts to update a neighbor, it first checks if the new link or its current cost are higher. This higher value is compared to the neighbor's cost. If this value is lower, the cost of the neighbor is then updated to the weakest link in the new ideal path.

A walkthrough of the modification on Dijkstra's algorithm will be shown below. All costs are set very high such that the source will be evaluated first.

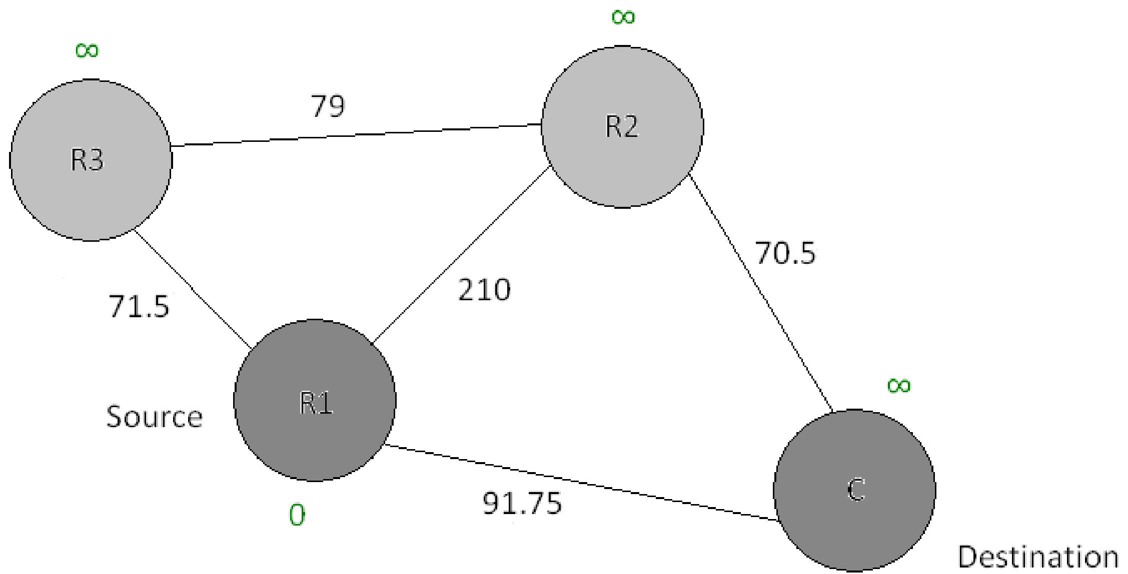


Figure 45: Routing Algorithm Step One

Next the source will update all of its neighbors if the worst connection in the path is less than its current value.

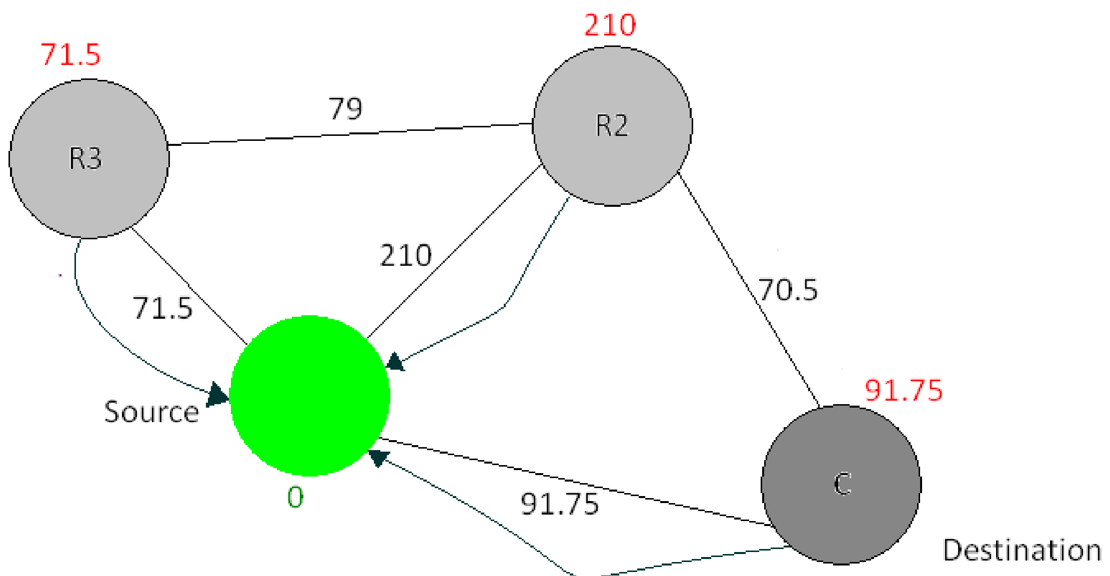


Figure 46: Routing Algorithm Step Two

The next node to visit whichever node is unvisited and has the lowest average cost. This continues until the destination is the node selected as the next to update its neighbors. At this point the other paths cannot provide a lower cost so the routing has been optimized.

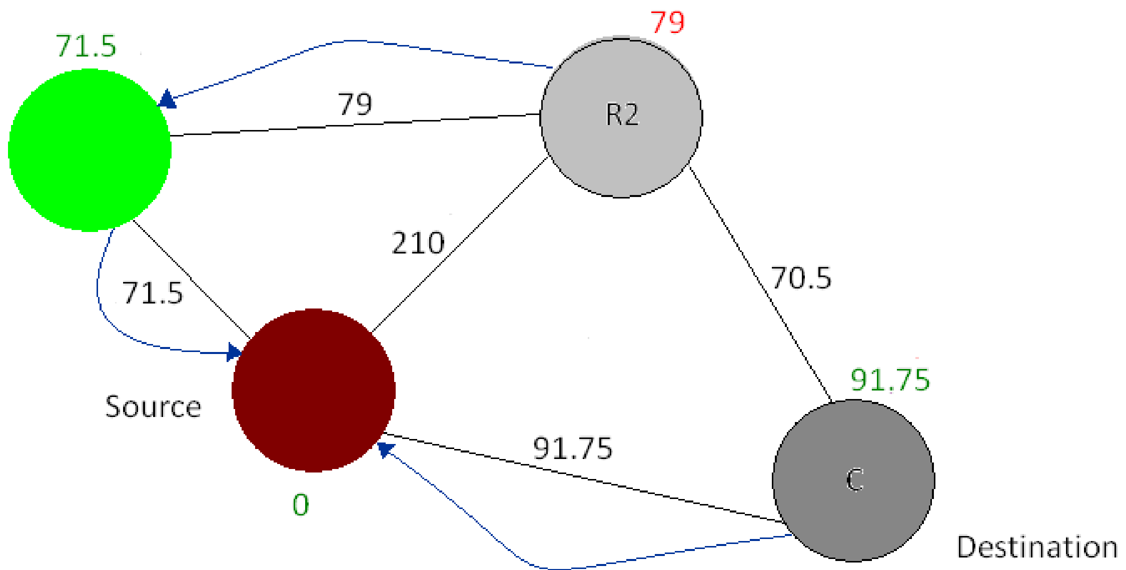


Figure 47: Routing Algorithm Step Three

Just like in Dijkstra's algorithm, every node is associated with the best node to come from. This is stored in an array called `track[receiver]`. The lines going from the destination to the source indicate the node that the `track` array. They are used to identify the path as it is being written.

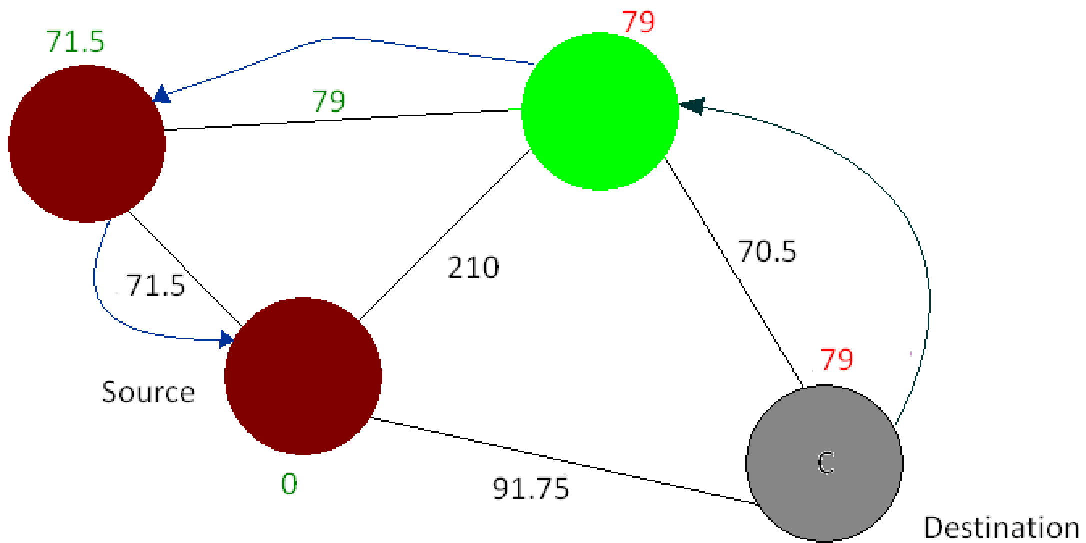


Figure 48: Routing Algorithm Step Four

When the best path is identified the algorithm then traces the path back to the source. First it finds `track[destination]`. The value inside this array is the node before it. It stores this in a two dimensional array called `pathTable[source][i]`. This will start with the coordinator and work itself backwards until `track[i]` equals `source`.

4.3.1.7. Setting Power Levels

To determine what each node's power level should be the source routes that are determined must be examined to find all present links. To do this the newly written `pathTable` must be examined. The program iterates through every node to view its route. The existing array after the `nodeID` is input is a backward listing of all the links in the network.

The program evaluates all links in the network and finds the necessary power level in the table `pL[sender][receiver]`. The final entries in `pathTable` of `R1` are listed as:

$$pathTable[1][j] = \{0,2,3,1\}\,. \tag{22}$$

The first link to be seen is between node 0 and node 2. Since this array is backwards, node 2 (`j+1`) is the sender and node 0 (`j`) is the receiver. Therefore the power level of node 2 is set to the value of `pL[2][0]`. In this example the value is 3. This process loops until `j+1` is the last entry or `j+1 = 0`, meaning that less than the maximum hops are used in the route and a default value has been found. The power levels are written into the same array that will be evaluated the next time that the diagnostic is run to determine the current power level when calculating the energy cost.

4.3.2. Path Limiting Algorithm

This algorithm discovers all possible paths that meet BER and RSSI QoS and can circumnavigate problem nodes. Next, it chooses the paths that preserve the battery lives of the nodes with the least charge and greatest re-transmission burden. The goal is to extend the operational duration of the entire network by means of putting the least burden on the nodes with the least battery charge. Rather than using a weighing system for the single hop connections, it ranks the QoS parameters instead. The primary QoS parameter is the BER of the links, the secondary QoS is node battery charge, and the tertiary QoS is the RSSI of the links. This ranking system is accomplished by performing the algorithm in a specific order. The bit error rate defines the links that can be used, the RSSI, then, defines the initial temporary paths, and the battery charge of the nodes updates the initial routes to optimize the networks duration of operation.

4.3.2.1. Path Finding

Each individual link found is first evaluated against the BER requirement to make sure that the weak links are not used. The bit error rate QoS parameter is set to 0.5%. If a link is found to be anywhere above this mark, it will not be used for routing. With these verified links, a path finding algorithm is performed. These paths include all feasible routes that hop directly to the coordinator as well as alternative routes that circumvent specific nodes to potentially preserve their charge if needed. The

battery life of an XBee node is based on its battery charge, which is fixed, and the number of packets it relays for other routers. Due to this, the path finder has to be optimized to ensure that the routes do not needlessly pass through nodes. For example, if a node, such as R7 in Figure 49, could connect directly to the coordinator, the direct route is the only path stored. This reason this is done is if R7 routed to a closer node, such as R2 in Figure 49 (b), that could pass its message along to the coordinator, it is still consuming the same current while forcing the closer node to consume current by having them re-transmit all its packets.

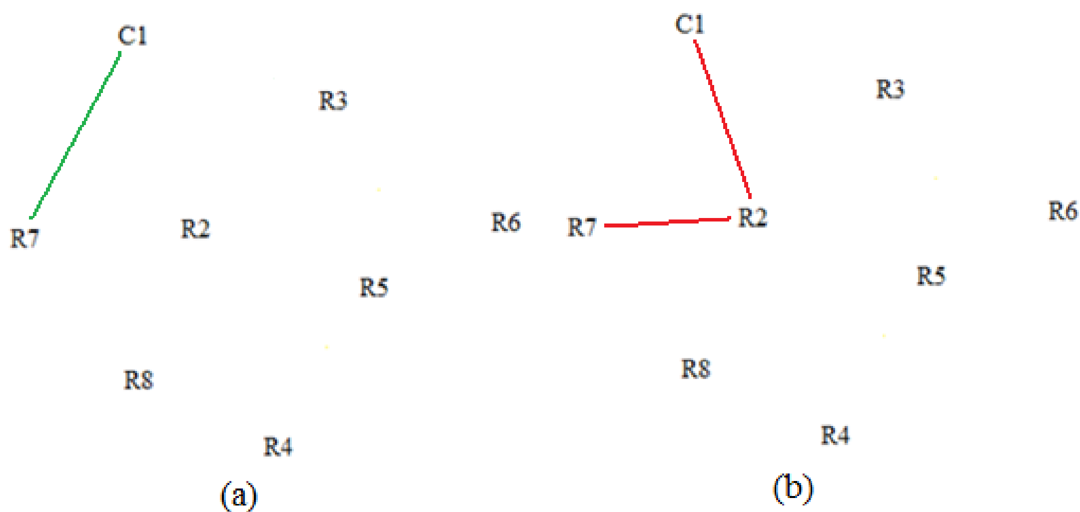


Figure 49: The path finding algorithm never circuitously routes two nodes (b) that can be linked directly (a).

The same principle holds true for the rest of the nodes in the system. A node will not connect to another node in a circuitous fashion if the two nodes have a link that meets the BER QoS parameters. Due to the fact that it needlessly increases the number of re-transmissions the nodes in the circuitous path are supporting. If there are nodes, such as R3 and R2 in Figure 50, between another node, such as R6 in Figure 50, and the coordinator, the routing algorithm tries to find a route to avoid middle nodes. Even if the route, such as the route in Figure 50, is much longer than the direct route, the path is still stored. This process provides the energy spreading algorithm feasible paths that can be used to place less burden on routers between the coordinator and other routers.

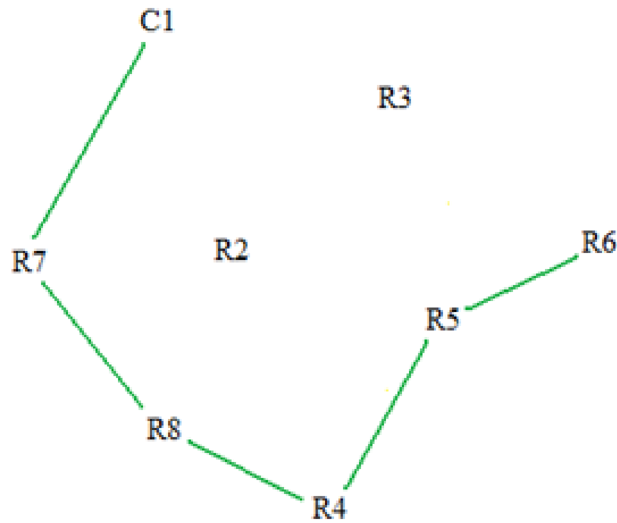


Figure 50: R6 circumvents R2 and R3 in case their charge is too low to relay R6 packets.

The path finding algorithm begins by finding all the neighbor nodes of the coordinator, such as R2, R3 and R7 in Figure 51. The neighbor nodes are the routers whose links with the coordinator meet the BER requirement. These nodes are automatically configured to transmit directly to the coordinator.

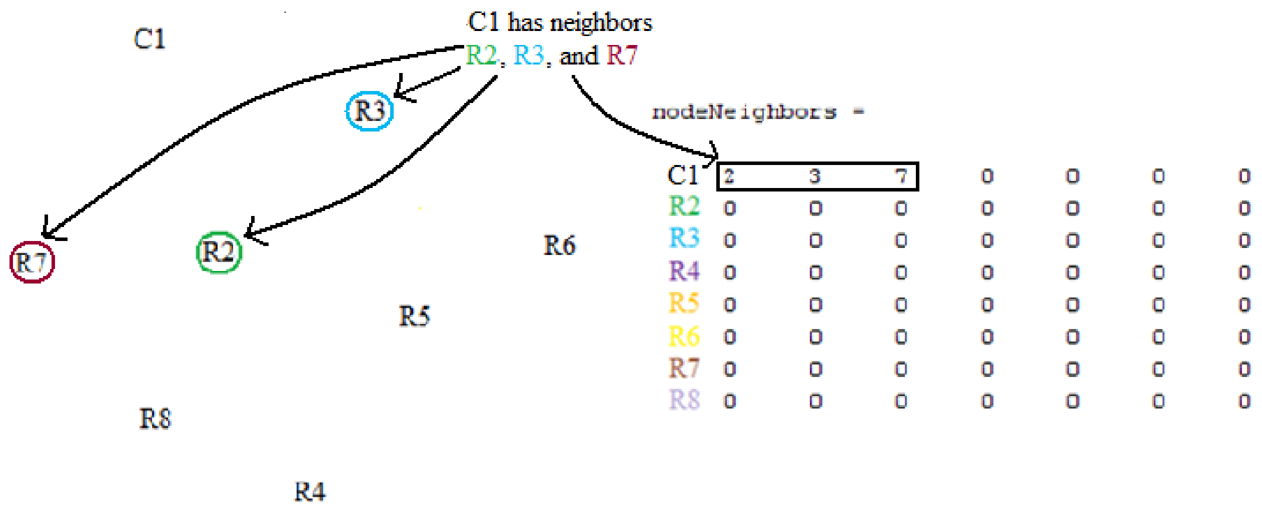


Figure 51: The node neighbors R2, R3 and R7 of the coordinator are found.

The router with the lowest assigned numeric number that neighbors the coordinator, R2 in Figure 52, is selected first. The first path is then extended to that router, R2 in Figure 52, where the neighbors of this router will be assessed. Meanwhile, the neighbors of the coordinator are stored for when the algorithm has found all of the paths that pass through this first router.

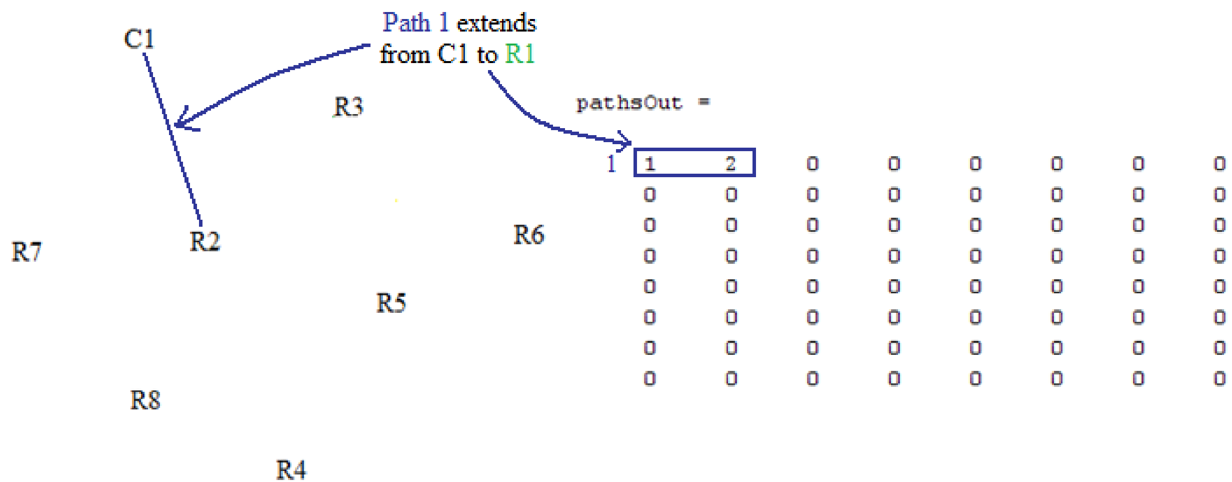


Figure 52: The path extends from the coordinator to the first router ordered numerically which is R2 in this case.

The router is then assessed for its neighbors. However, the links not only have to meet the BER QoS but a second condition as well. If the considered nodes, such as R3, R4, R5, R7, and R8 in Figure 53, are neighbors of a previous node in the same path, such as C1 in Figure 53, they is discarded. In Figure 53, R3 and R7 are discarded because they are neighbors of C1. This ensures that no node sends a message through another node, when it can do it itself.

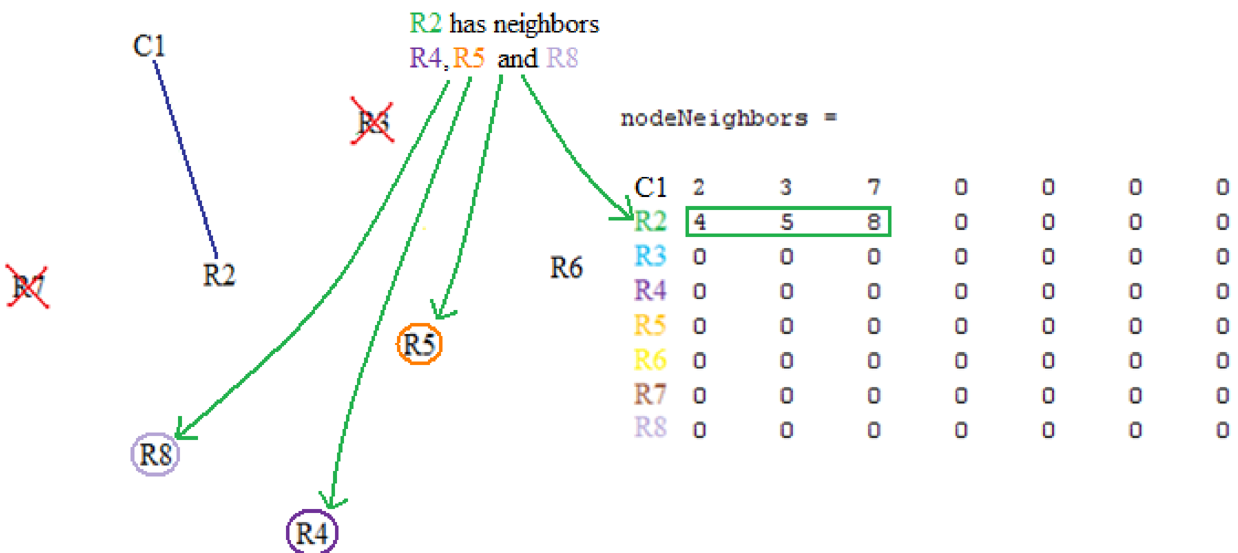


Figure 53: The first router finds its own node neighbors R4, R5 and R8 while disqualifying R3 and R7 who are the coordinator's neighbors.

The path extends again to the next node of the lowest numerical value, such as R4 in Figure 54. It stores all the neighbors of the nodes in the path up to this point, such as C1 and R2 in Figure 54, so it can explore them after the current path is set. This process is repeated until the path reaches a node that only has neighbors common to nodes that came before it, such as R4 in Figure 54.

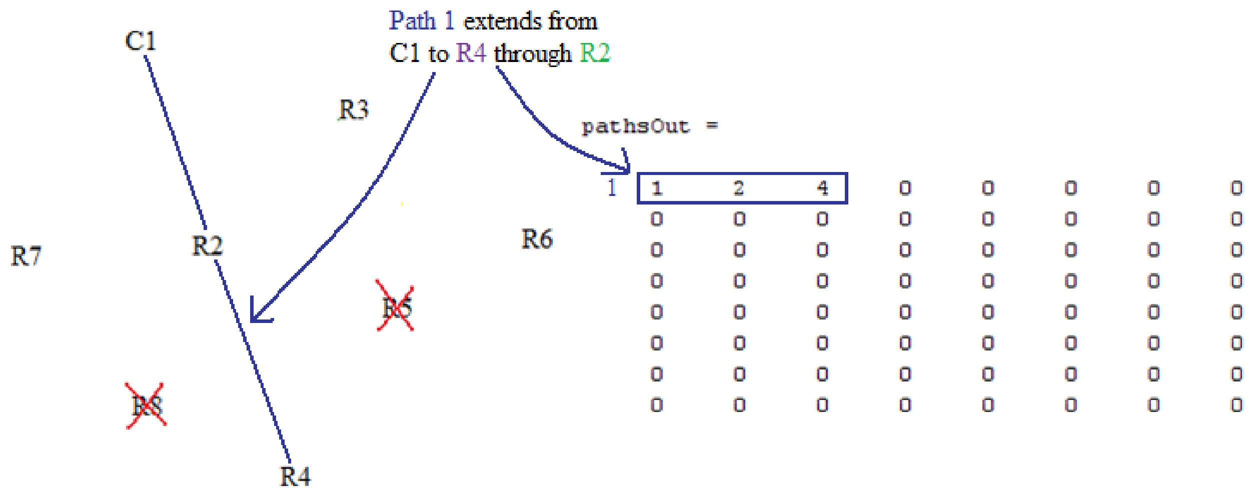


Figure 54: R4 cannot find any node neighbors that aren't already neighbors of R2.

Once a path has reached an end point, such as R4 in Figure 55, the algorithm will step back to the node before it, R2 in Figure 55, and check to see if it has any additional neighbors to explore, R5 and R8 in Figure 55. If it does, it will create a new path, such as path 2 in Figure 55, and repeat the path finding process in that direction. If it does not, it will step back to the previous node in the path to evaluate any neighbors of the previous node. This process is repeated until a node in the path with additional neighbors to explore is found.

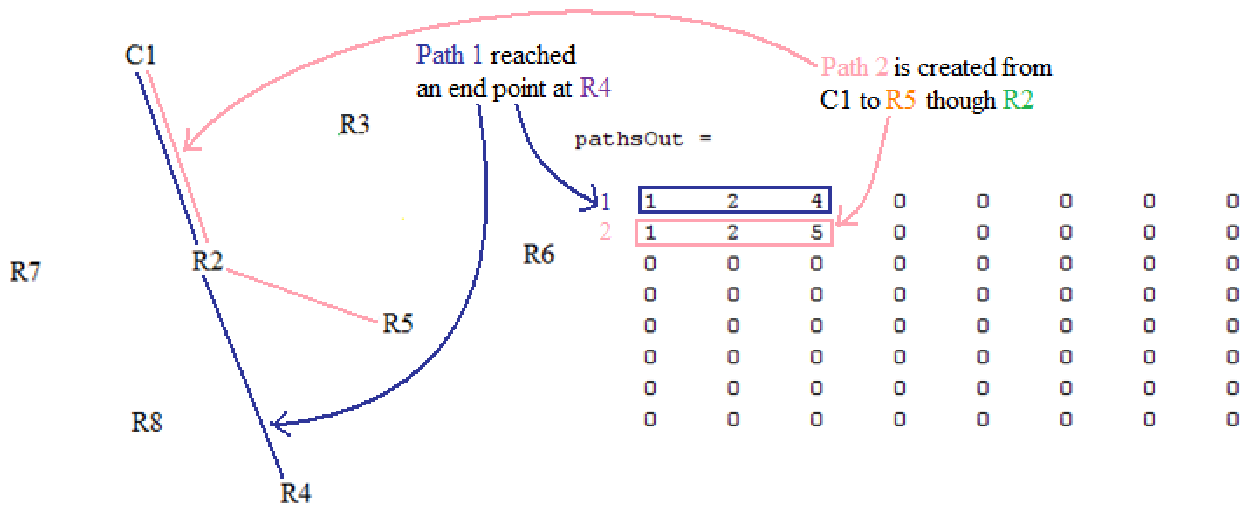


Figure 55: The path returns to R2 which still has R5 and R8 to explore and a new path is created that extends to R5, the next lowest router on R2's neighbor list.

When the path finding returns to the coordinator and there are no additional neighbors to explore, the path finding algorithm is complete. When the process is completed the generated paths out from the coordinator are converted into paths in from each router as seen in Figure 56. This makes it

simpler for the routing algorithm to find the impact of each alternative route on every nodes battery life in the network. All the redundant paths are deleted to increase the efficiency of the routing.

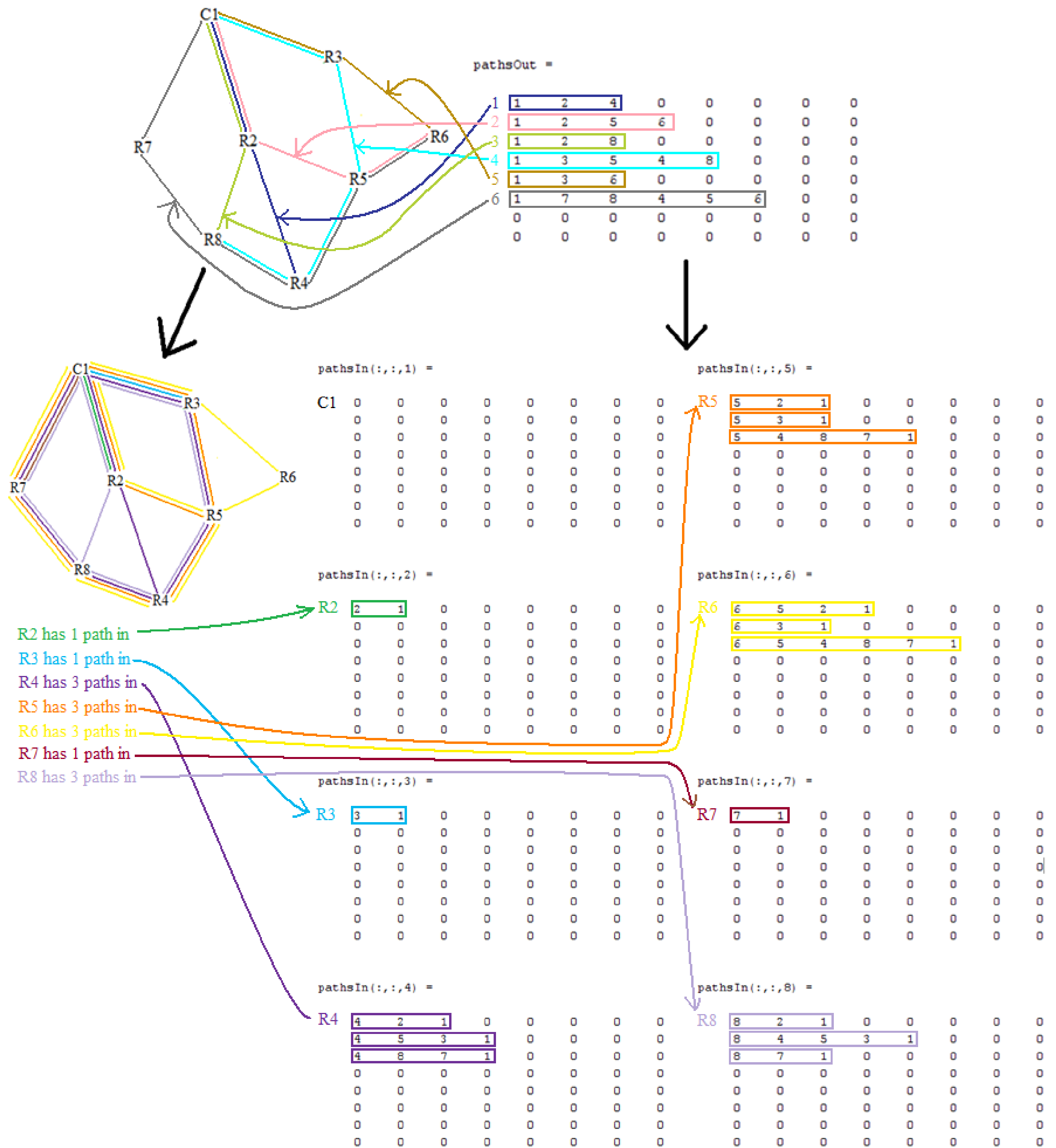


Figure 56: The paths out from the coordinator are converted to paths in from the routers.

4.3.2.2. Power Setting and RSSI Evaluation

When the paths in have been compiled, Dijkstra's algorithm is performed using RSSI as the sole cost of each link. The RSSI cost of each link is derived from four RSSI readings. The routers on both ends of one link each send out two link assessment packets on power levels one and three. If the coordinator receives an RSSI reading from both ends of a particular link, it averages the power level one RSSI readings of each end node together and the power level three reading of each end node together. If the coordinator only receives the cognitive data for a specific link from one node, it will only use the two RSSI values from that node to calculate the final RSSI reading. The averaged or independent RSSI readings from power level one and power level three are run through a series of conditionals, seen in Figure 57, to find which power level should be assigned to ensure that the link stays between -80dBm and -60dBm. These conditions set the required power level.

$$-60dBm < RSSi_{PL1} \quad \rightarrow PL0$$

$$-80dBm < RSSi_{PL1} < -60dBm \quad \rightarrow PL1$$

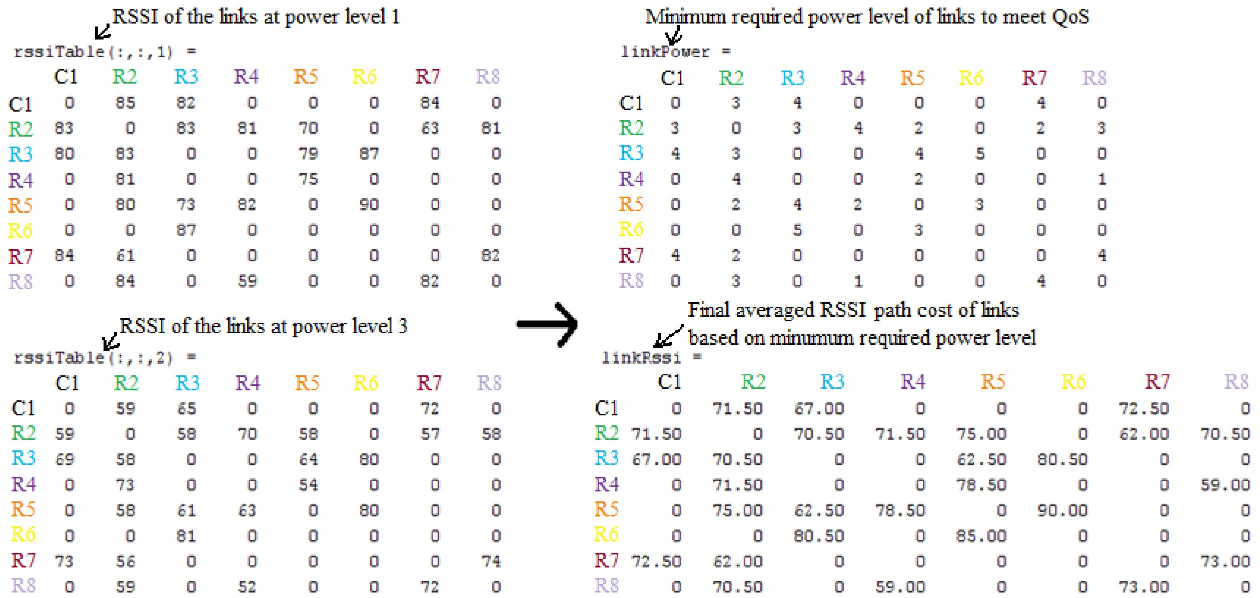
$$RSSi_{Pl1} < -80dBm \qquad -60dBm < RSSi_{Pl3} \quad \rightarrow PL2$$

$$-80dBm < RSSi_{PL3} < -60dBm \quad \rightarrow PL4$$

$$RSSi_{Pl3} < -80dBm \quad \rightarrow PL5$$

Figure 57: The power level for each link is selected by comparing the link assessment packets received RSSI against the threshold power boundaries -80dBm to -60dBm.

Based on the required power level of a particular link, the RSSI cost of the link is derived. If the power level has been set to zero or one then the RSSI cost is equivalent to the averaged or independent RSSI reading from the link assessment packet sent at power level one, as seen in the some of the links in Figure 58. If the power level has been set to three or four, the RSSI cost is set to the averaged or independent RSSI value of the power level three link test packet, as seen in the some of the links in Figure 58. If the power level has been set to 2, the average of the two varied power level RSSI readings is used as the path cost, as seen in the some of the links in Figure 58. This matching is done so that the initial links created from the RSSI path cost will perform the best at the power levels set for each node.



RSSI of the links at power level 1

rssiTable(:,:,1) =

	C1	R2	R3	R4	R5	R6	R7	R8
C1	0	85	82	0	0	0	84	0
R2	83	0	83	81	70	0	63	81
R3	80	83	0	0	79	87	0	0
R4	0	81	0	0	75	0	0	0
R5	0	80	73	82	0	90	0	0
R6	0	0	87	0	0	0	0	0
R7	84	61	0	0	0	0	0	82
R8	0	84	0	59	0	0	82	0

Minimum required power level of links to meet QoS

linkPower =

	C1	R2	R3	R4	R5	R6	R7	R8
C1	0	3	4	0	0	0	4	0
R2	3	0	3	4	2	0	2	3
R3	4	3	0	0	4	5	0	0
R4	0	4	0	0	2	0	0	1
R5	0	2	4	2	0	3	0	0
R6	0	0	5	0	3	0	0	0
R7	4	2	0	0	0	0	0	4
R8	0	3	0	1	0	0	4	0

RSSI of the links at power level 3

rssiTable(:,:,2) =

	C1	R2	R3	R4	R5	R6	R7	R8
C1	0	59	65	0	0	0	72	0
R2	59	0	58	70	58	0	57	58
R3	69	58	0	0	64	80	0	0
R4	0	73	0	0	54	0	0	0
R5	0	58	61	63	0	80	0	0
R6	0	0	81	0	0	0	0	0
R7	73	56	0	0	0	0	0	74
R8	0	59	0	52	0	0	72	0

Final averaged RSSI path cost of links based on minumum required power level

linkRssi =

	C1	R2	R3	R4	R5	R6	R7	R8
C1	0	71.50	67.00	0	0	0	72.50	0
R2	71.50	0	70.50	71.50	75.00	0	62.00	70.50
R3	67.00	70.50	0	0	62.50	80.50	0	0
R4	0	71.50	0	0	78.50	0	0	59.00
R5	0	75.00	62.50	78.50	0	90.00	0	0
R6	0	0	80.50	0	85.00	0	0	0
R7	72.50	62.00	0	0	0	0	0	73.00
R8	0	70.50	0	59.00	0	0	73.00	0

Figure 58: The RSSI found from link assessment packets at power level one and three are used to derive required link power and RSSi path cost.

With the RSSI path costs of each link found, Dijkstra's algorithm can be performed. All of the routers that only have one path because they are neighbors of the coordinator, such as R2, R3 and R7 in Figure 59, are assigned there only route regardless of path cost. This is due to the fact that the number of routes that pass through a node takes precedent over the RSSI reading and if a node can link directly to the coordinator, it should not force other nodes to relay its packets for it.

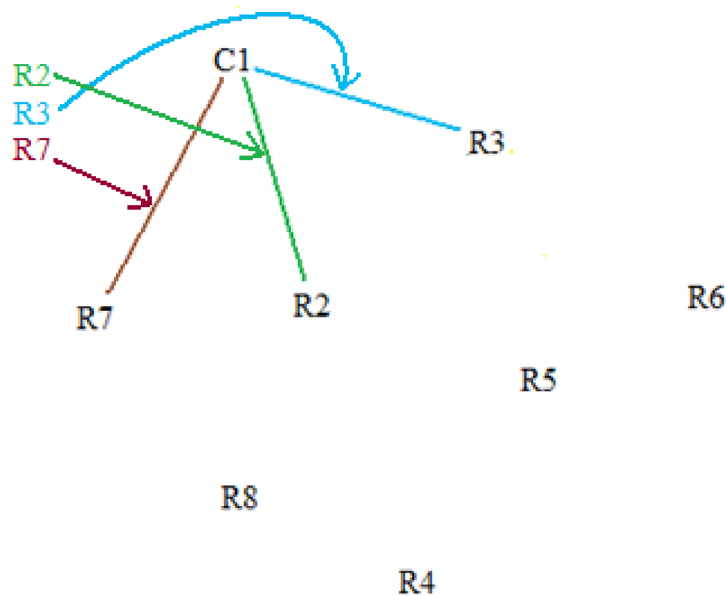


Figure 59: R2, R3 and R7, the neighbors of the coordinator, are routed directly to the coordinator.

For each router that has more than one path, such as R4 in Figure 60, the algorithm simply adds the RSSI path cost of each link of its first path together to find the total path cost, which is 143 in Figure 60. The total path cost is found in the same manner for second path of each router, which is 208 is Figure 60. The path with the least total cost, route 1 in Figure 60, is selected as the new route. The total cost of the third path, 204.5 in Figure 60, is then compare to the total cost of the declared route, and the lesser costing path is assigned as the new route, which is route 1 in Figure 60.

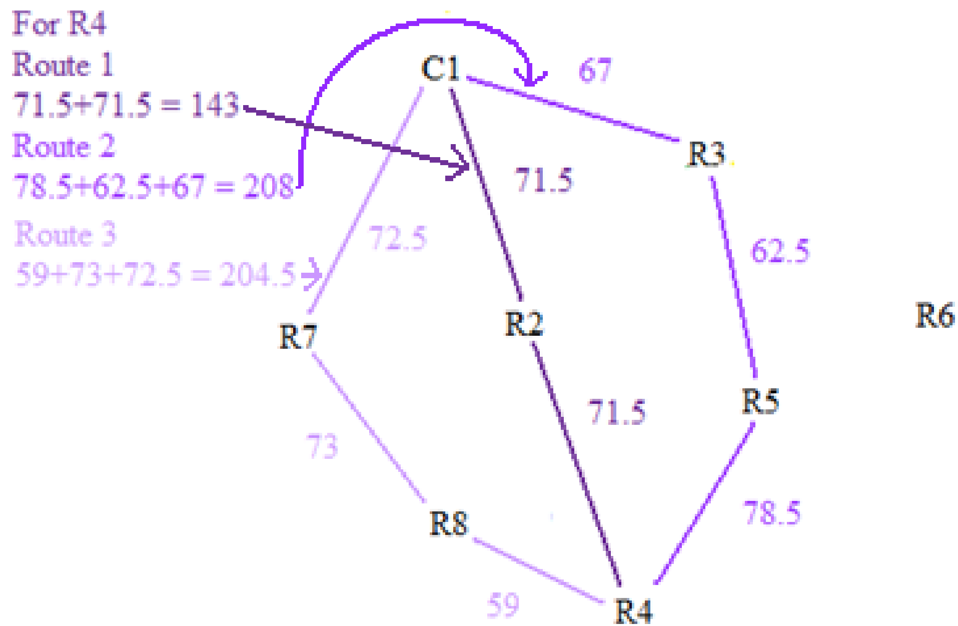


Figure 60: R4 chooses the route 1 because it has the smallest total path cost.

This process is repeated until the path with the least total RSSI cost has been selected for all routers in the network, as seen in Figure 61. While it records these paths it also records the routes passing through each node, and the source nodes of these paths. From this information it is simple to derive the number of routes passing through a particular node in order to calculate its battery life. If the source nodes of the routes that pass through a particular node are stored then it is simple to find alternative routes to avoid the node if it has low charge.

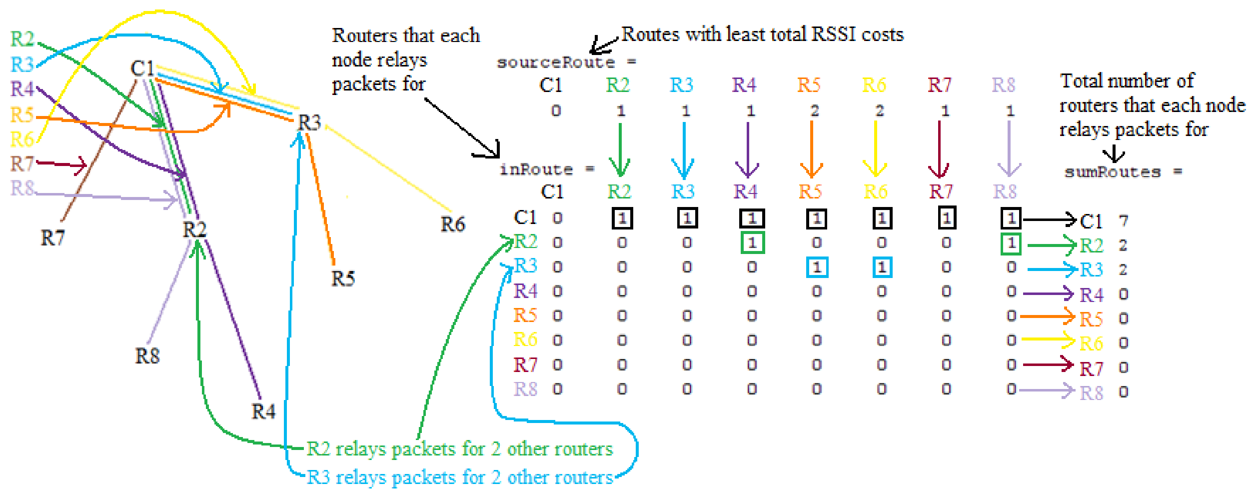


Figure 61: The initial source routes are selected based on RSSi path weights. The number of routes pass through each node is recorded.

4.3.2.3. Battery Life Evaluation

After the initial routers are set, the battery life for each node is calculated. The knowledge of the number of routes each node is a part of is found in the initial routing. The battery charge is acquired when each node sends its cognitive parameters to the coordinator. Therefore the battery life of each node can be calculated with the expression:

$$t_{battery} = \frac{Q}{N_{routes}\ D\left(\frac{I_{Rx}}{2}+\frac{I_{Tx}}{2}\right)+(1-N_{routes}\ D)I_{idle}+I_{microcontroller}}. \quad (23)$$

Q is the battery charge, N_{routes} is the number of routes the node is a part of, and D is the duty cycle of a node, which was set to 0.1. The current draw when the XBee is receiving is 45mA. When it is transmitting, it draws 40mA and when it is idle, it draws 15mA. The ATmega168 microcontroller draws 5.5mA constantly. All the battery lives of the nodes from the initial routing are stored and a permutation of the node list is stored ordering the nodes from least battery life to most battery life, as seen in Figure 62.

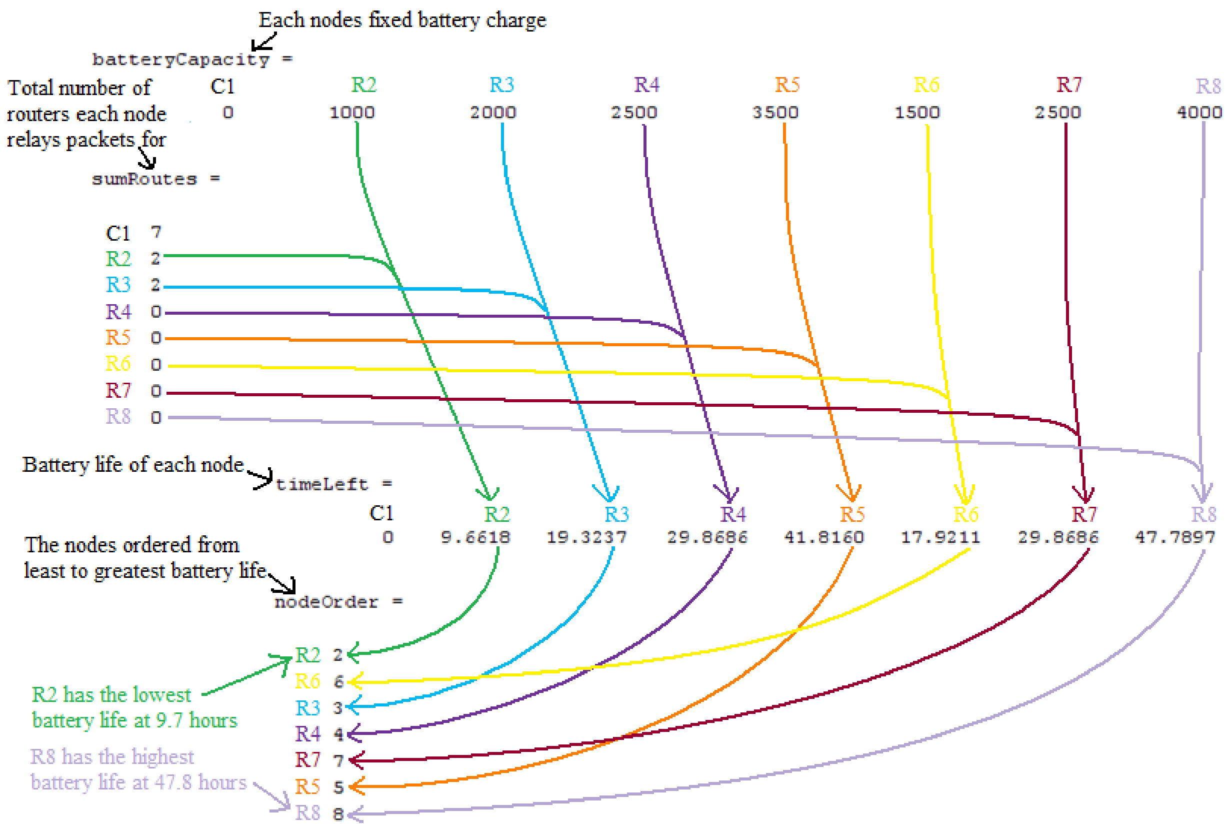


Figure 62: The battery capacities and the number of routes each node is a part of is used to calculated battery life and the order the nodes are checked.

The first node in the node order list, R2 in Figure 62 and Figure 63, is the node with the shortest battery life. A search for the nodes that have initial routes through the node with the shortest battery life is performed. The first node with one of these routes, such as R4 in Figure 63, is then re-routed temporarily using the first path stored that is not the path currently selected, such as route 2 of R4 in Figure 63. Temporary values of the number of routers each node has to relay packet for based on the temporary route change is used to calculate temporary battery lives of each node. A comparison is made between the original battery life and the new temporary battery life for each node in the order from the limiting node to the node with the greatest battery life. If the battery life of the limiting node improved from the route change, it will store the temporary routes as the permanent routes. In Figure 63, the battery life of R2 increased from 9.7 hours to 10.7 hours from R4 switching from route 1 to route 2. If the battery life of the limiting node decreased from the route change, the route will be disqualified and the next route will be checked, which did not occur in Figure 63.

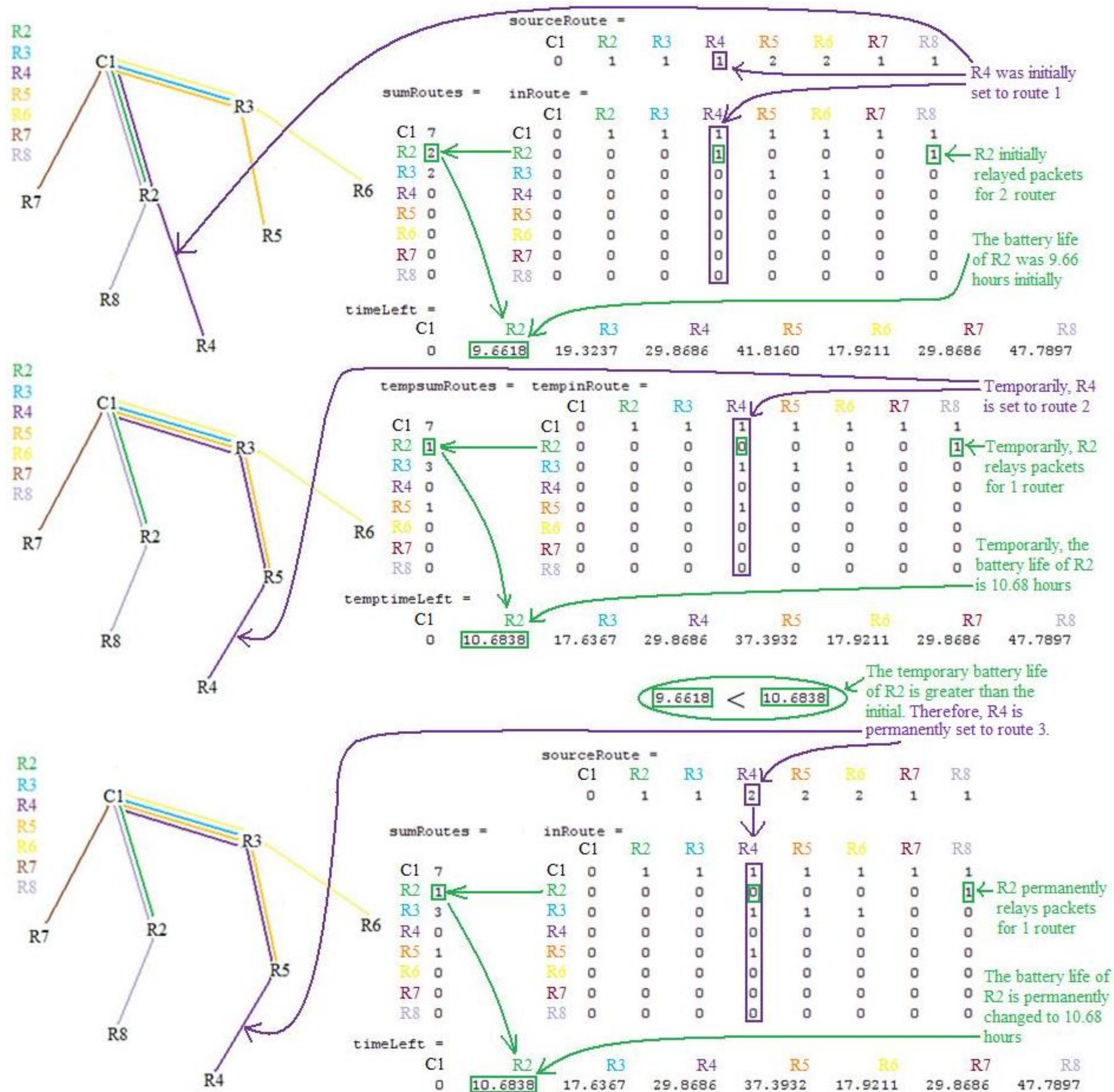


Figure 63: The first alternative path, route 2, from R4 is compared with the original route, route 1. The battery life of R2 is increased from 9.7 hours to 10.7 hours, so the route becomes permanent.

Then, the next path, route 3 of R4 in Figure 64, is then checked for improvements. If the battery life was lengthened from the previous route adaptation, the limiting node, R2 in Figure 64, cannot be improved from this step. But, it is possible the nodes with greater battery life, such as R3 in Figure 64 can still be improved. If the next path checked, route 3 of R4 in Figure 64, doesn't improve the battery life of the limiting node, the node with the second least short battery life is checked for fluctuations in the same fashion. If the node with the second least battery life is unchanged, this process is repeated for each node in order of battery life until a change is found. If the change increases a nodes battery life, then the route is made permanent. In Figure 64, the battery life of R3 is increased from 17.6 hours to

19.3 hours. If not it is disqualified. This ensures that the nodes in the network are prioritized by their battery life during re-routing.

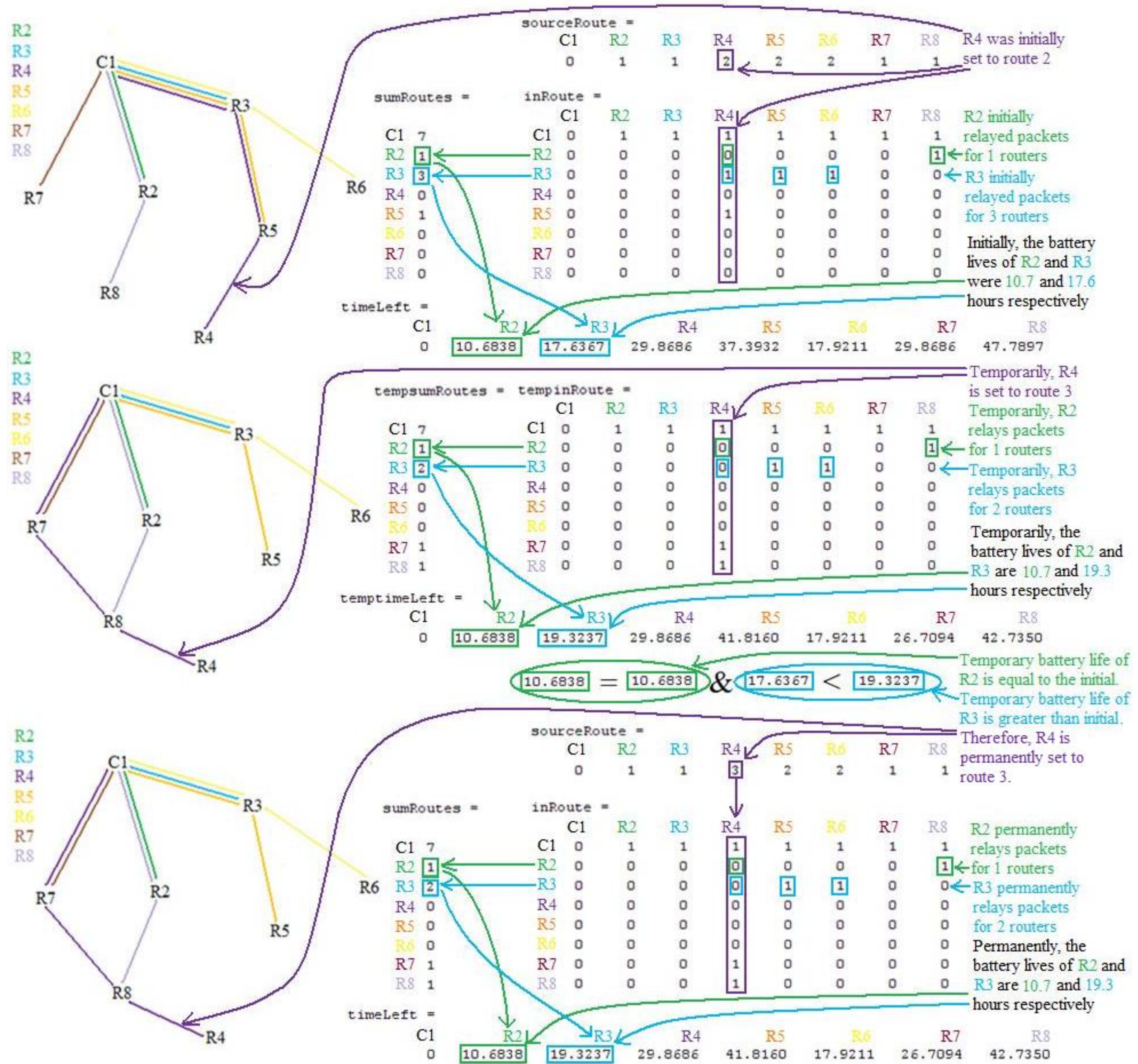


Figure 64: The limiting node R2 and R6, the node with the second shortest battery life, are unaffected by this route change. The route is still made permanent because the node with the third shortest battery life, R3, increases from 17.6 hours to 19.3 hours.

After all the paths of the first node, whose initial route hopped through the limiting node, are checked, the next node, whose initial path hops through the limiting node, is checked for alternative routes. These alternative routes are selected in the same fashion as before. Once all the nodes, whose initial routes hopped through the limiting node, are checked for alternative routes, the order of the nodes from shortest to longest battery life is updated based on the new route selections. If the order changes, alternative routes are then checked in order to circumvent the new limiting node.

If the order remains unchanged, then the nodes, whose routes run through the node with second shortest battery life, are checked for alternative routes. If there are no nodes, whose routes hop through the node with the second shortest battery life, such as the case for R6 in Figure 65, then alternative routes are checked for the node with the third least battery life, R3 in Figure 65. Alternative routes are checked for all the nodes one at a time from the limiting node to the node with the greatest battery life. When storing these alternative routes as permanent routes the same conditions are used as before. The priority always goes to the nodes with the shortest battery lives.

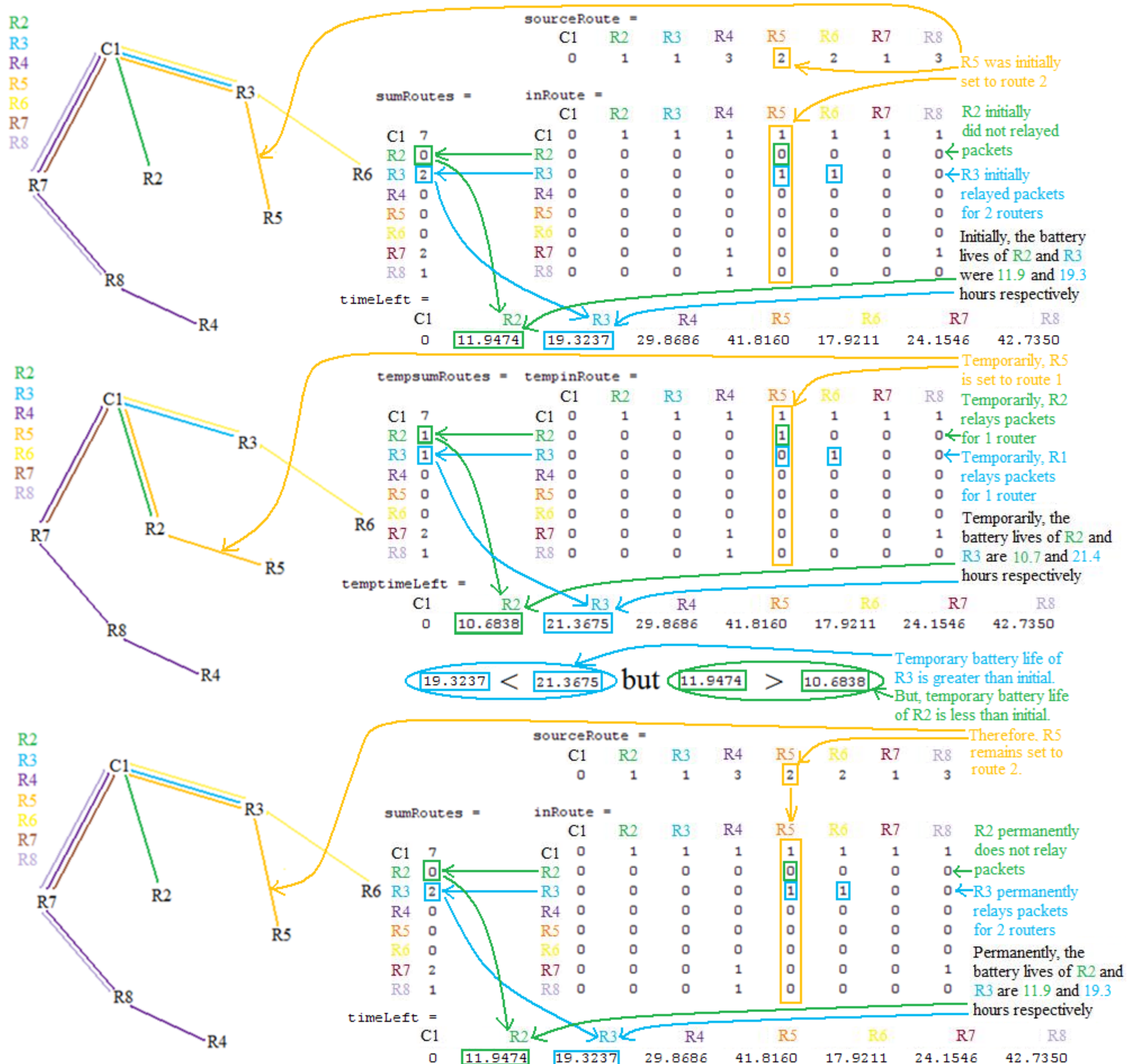


Figure 65: When checking the alternative routes for the nodes that originally went through R3, The limiting node R2 still takes priority. Re-routing R5 through R2 caused its battery life to drop from 11.9 hours to 10.7 hours, therefore the route is discarded.

When checking route alterations, the affect on the list of nodes in the order of battery lives is taken into account. If a route alteration were to cause the battery life of a node, such as 20.3 hours from node R7 in Figure 66, to fall below the original battery life of the node for which the re-route is being performed, such as 21.4 from R3 in Figure 66, then the re-route will not become permanent. This is an adaptive method to make sure that the permanent re-routes do not compromise the intention of the algorithm; to extend the battery lives of the nodes with the shortest battery lives as much as possible.

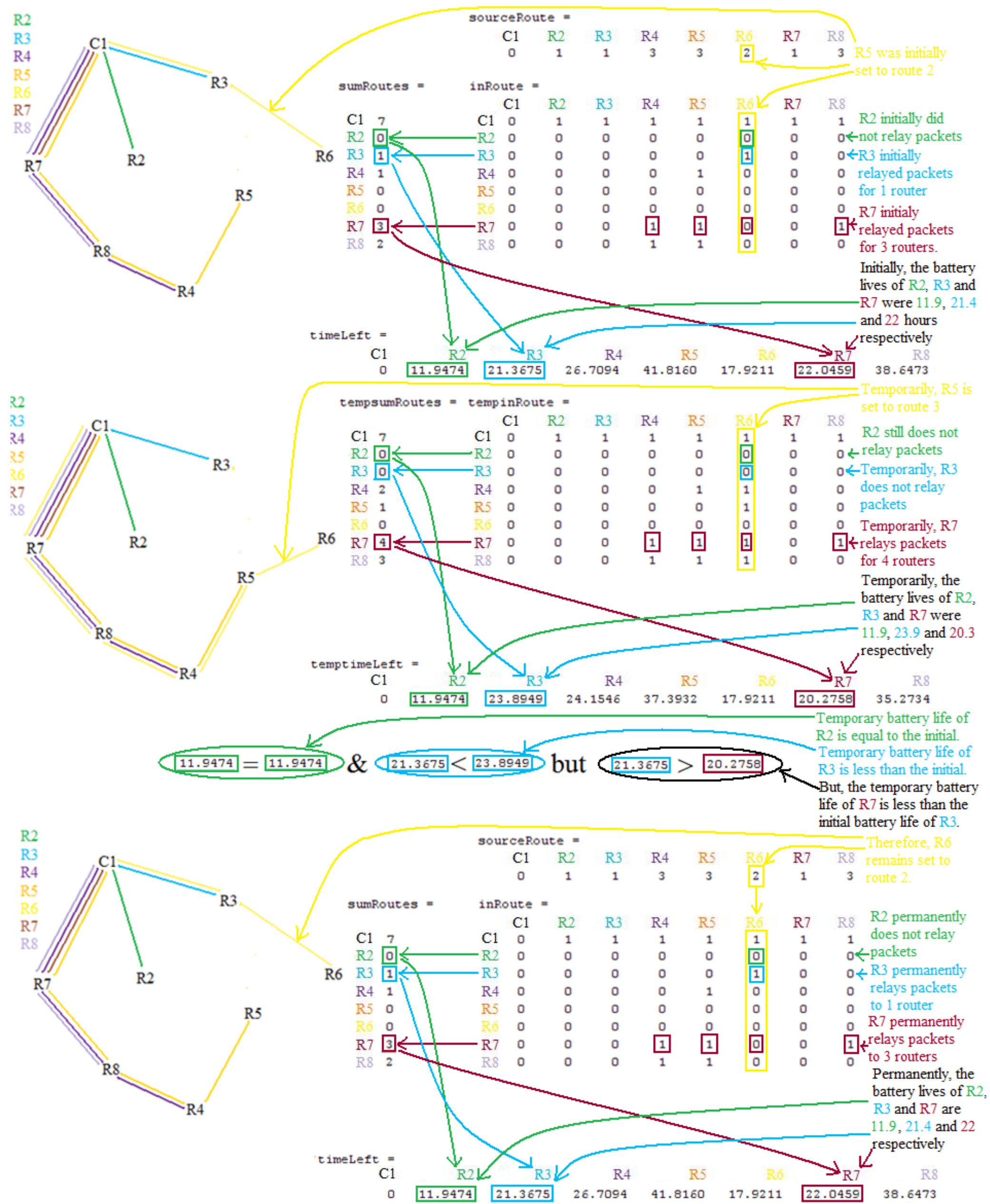


Figure 66: Even though the re-route boosts R3's battery life from 21.4 to 23.9 hours. The route is unchanged because R7's battery life would fall to 20.3 hours which is lower R3's original battery life, 21.4 hours. These are the final route paths that the algorithm produced.

4.4. Source Routes

After the coordinator has performed the routing algorithm, it must set its source routes, and then send the source routes and power levels back to the routers. This is necessary for the optimized routes to be used and for the changes made through the cognitive feedback to take effect. Without this step the routers will continue to route using the automated built ad hoc on demand distance vector routing, which causes request flooding and does not factor in battery charge.

4.4.1. Setting the Coordinator Source Routes

The coordinator must set its source routes before it sends the source routes and power levels back to the routers, so that the impact of the routing algorithm takes effect immediately after it is performed. The transmission limiting algorithm has defined routes back to the routers, where as the path limiting algorithm simply uses the source routes from the routers to the coordinator in reverse.

To create the source route, the coordinator routes are stored in a two dimension array called `setRoute`. One dimension has the node ID of the destination router. The other dimension is structured to fit into to the create source route frame similar to that of Figure 35, and is equal to:

$$N_{hops} = (2 * (N_{nodes} - 2)) + 3. \quad (24)$$

This is to ensure that all the necessary information can be stored in the worst case scenario, which is when all the nodes in the network are used in one path. The first byte of this dimension for all routes is the character 'S', as seen in Figure 67, to indicated to the `arrayLength` method that it is a `setRoute` array. This condition adapts the method `arrayLength` to count the number of bytes in the array iteratively. It increases the length count iteratively and checks each iteration if the entry at the length count of the `setRoute` array is the end delimiter: another 'S', as seen in Figure 67. The second byte of the `setRoute` array is the number of hops in between the source and destination of the nodes, shown in Figure 67. The two byte intermittent short node addresses are then stored in the order of the path from the node closest to the destination to the node closest to the source, as depicted in Figure 67, by matching their node ID to their address in `addressTable`. An example of this `setRoute` array dimension for one node is displayed in Figure 67.

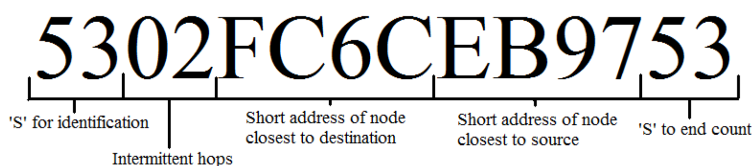


Figure 67: This is an example of how the source route is stored before creating the source route.

Once the route to each node is stored in this format it is input into a method called `createRoute` along with the long and short destination addresses of the destination router. These two arrays are then transformed into the create source route API frame that resembles Figure 35. Once this create source route frame is sent to the UART of the XBee, then the coordinator will use those routes when sending the routes back to the routers. The coordinator then compiles the source routes and power levels for each node and sends them to the routers using a similar transmit request frame as Figure 33. The payload of this message is stored with 'R' surrounding them so that the `arrayLength` method can count its byte length in the same way as the `setRoute` array. The final payload of this message is sent without the 'R's in the same format as Figure 68.

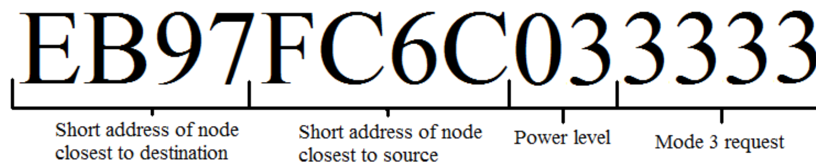


Figure 68: This is an example of the payload when the coordinator sends the source route and power level back to the routers.

4.4.2. Setting the Source Route and Power Level of the Routers

Once the routers have received the source route and power level for optimized routing, they change their own source routes in the same fashion the coordinator did. The routers parse out the source route from the transmission and store it in an array in the same order that the coordinator did when it set its source route. The number of intermittent hops is found through the packet length of the received packet API frame using the equation:

$$N_{hops} = (N_{payload\ \ bytes} - 3)/2. \tag{25}$$

Three is subtracted from the payload bytes because the transmission power and two mode 3 bytes are appended to the end. Then it is divided by two because each short address is two bytes long. Once the source route is in the same form as the coordinator had it in Figure 67, then it sends the create source route API frame, similar to Figure 35, to the UART of the XBee ZB radio. Once the source route has been set, the power level of the node needs to be set. The power level is parsed out as the third to last byte of the payload. It then writes the power level adjustment AT command API frame with the optimized power level in the same format as Figure 34. Once this is complete, the entire cycle repeats and the routes are optimized again in order to adjust for real time changes.

4.5. Chapter Summary

The system works by first performing environment sensing techniques, evaluating them, and setting operation parameters. These environment sensing techniques include node discovery, and a link assessment at two different power levels to measure BER and RSSI. These parameters are used to set the most ideal source route per node for the network as well as an appropriate power level of operation.

5. Testing and Verification

After writing the program for performing optimization of a network using cognitive parameters, the code needs to be debugged. Since the code is very complex, this section is broken down into subsections which debug each section separately and then debug the system step by step as features are added on. This process is broken down further into two sets of code, one for all the routers and one for the coordinator of the network. The first component in both the router code and the coordinator code that is emphasized is the node discovery, where one node identifies all nodes in the surrounding area by their long and short address. Then the code for acquiring the cognitive parameters, BER and RSSI, is analyzed in both types of nodes. In the final step of the coordinator, the code for routing is examined, and last for the routers code, the receiving and setting of the source routes sent by the coordinator is debugged. All figures in this section are of the output from the system. All code used in the final project can be found on page 117 in the appendix.

5.1. Node Discovery

The node discovery needs to be able to run smoothly. Obtaining addresses of all nodes near to the device that is running node discovery and allowing information to be read in while the process is being completed is necessary.

5.1.1. Node Discovery for the Coordinator

The main problem with the coordinators original node discovery code is that it often fails to find all nodes in the neighboring area. Below in Figure 69, the output of the code is displayed when three nodes were in range of the coordinator. Outlined in red are the two addresses found both in the received messages and in the node table at the bottom. The two at the end of the display illustrates that two nodes were found during the node discovery. This print out demonstrates that although the node discovery is working, the algorithm used has to be made more reliable such that all neighboring nodes are identified.

```
SD: 7e
length: 26
7e001988014e4400c66e0013a200403c2c412000fffe0100c105101e00
Node found
Iteration because c6 != 00 and 6e != 00, i = 0
Node recorded
SD: 7e
length: 26
7e001988014e4400b3a70013a200403c2c3f2000fffe0100c105101edc
Node found
Iteration because b3 != c6 and a7 != 6e, i = 1
Node recorded
0013a200403c2c41c66e
0013a200403c2c3fb3a7
2
```

Received Packet Containing One Address found in Node Discovery

short address

long address

Here 2 nodes were found out of the 3 the existed in the network, the short and long addresses are reversed to make sending easier.

Figure 69: Node Discovery Error in Coordinator

To make the node discovery more reliable, the time given to find all nodes in the system is increased from the minimum of 3.2 second to 5 seconds and then increased again to 8 seconds. The output of the finished code is shown in Figure 70.

```
SD: 7e
length: 26
7e001988014e4400b3a70013a200403c2c3f2000fffe0100c105101edc
Node found
Iteration because b3 != 00 and a7 != 00, i = 0
Node recorded
SD: 7e
length: 26
7e001988014e4400af930013a200403c2c3d2000fffe0100c105101ef6
Node found
Iteration because af != b3 and 93 != a7, i = 1
Node recorded
SD: 7e
length: 26
7e001988014e4400c66e0013a200403c2c412000fffe0100c105101e00
Node found
Iteration because c6 != af and 6e != 93, i = 2
Node recorded
0013a200403c2c3fb3a7
0013a200403c2c3daf93
0013a200403c2c41c66e
```

Received Packet Containing One Address found in Node Discovery

Here all 3 nodes were found after increasing the time given to perform node discovery to 8 seconds instead of 3.2 seconds

Figure 70: Debugged Node Discovery in Coordinator

With the node discovery completed in the coordinator, the acquired data is then used to analyze and debug the performance of the node discovery code for the routers. Specifically, the coordinator can

see all addresses of the nodes that are within range. Using this information, it can be seen which addresses are not being found during the routers node discovery.

5.1.2. Node Discovery for Routers

The node discovery code on the routers is almost identical to that of the coordinators. The main differences are that the routers code is written in the Arduino environment using communication functions with the virtual COM port whereas the coordinators code is written using C in Emacs. Communication between the computer and the coordinator is done using a file descriptor. This means that the output of the routers can be made clearer since the code can be simpler. Below in Figure 71, the original node discovery code from the router is only able to find one node, the coordinator.

```
~    NDd
Input Data: ~  ˆ ND    ¢ @<,!  ÿþ  Á   U
Add Node
Node Table:
  ¢ @<,!
Nodes: 1
~    PL Z
Test == 200
~ -     ¢ @<,!    GGGGGGGGGGGGGGGGGGGGGGGGGN3P 22J
Input Data: ~  <      s
Input Data: ~ +   ¢ @<,!   GGGGGGGGGGGGGGGGGGGGGGGGGNOP 22Ï
Input Data: ~ +   ¢ @<,ÄÆn GGGGGGGGGGGGGGGGGGGGGGGGGN3P 22w
```

Here the device identifies 1 node, the coordinator.

Here the device gets signals from the coordinator as well as 1 other node that it did not find in node discovery

Figure 71: Node Discovery Error in Router

After applying the same fix as the coordinator, increasing the time for node discovery to 8 seconds from 3.2 seconds, the router can find all nodes in the network. Another problem is discovered in Figure 72 below.

```
Input Data: ~    ¢ @<,!   11
Mode 1
~    PL X
~    BH j
~    NT2"
~    NDd
Input Data: ~  ˆ ND    ¢ @<,!  ÿþ  Á   U
Add Node
  Pan ID setup: ~    ID q
Scan Channels: ~    SCÿÿb
Apply any changes: ~    ACr
Start RSSI: ~    RPÿ
```

Visible Characters of the coordinators address

Here the program restarts during node discovery, the pan ID setup command is the first command in the program.

Figure 72: Program Restart Error in Router

After the address of the coordinator is found using node discovery, the program is restarting. A couple of solutions were applied, the first was to reduce the amount of memory used in the code and the second was to give more time for node discovery to run. However, the problem still exists. The final technique used was to spread out the node discoveries of all nodes in the system into separate time slots. The node discovery command itself is a broadcast that causes interference in the network when it is used. When the code performs multiple node discoveries at once, one for each router in the system, the network gets overloaded and causes the code to fail and restart. To decrease the strain on the system each node waits a period of time equal to eight seconds multiplied by the ID of that node. For example, node three in the system must wait 24 seconds before starting node discovery. Eight seconds is used since that is the amount of time chosen for how long a node discovery has to find all neighboring nodes in the network. The finished output of node discovery can be seen below in Figure 73.

```
Mode 1
~    PL X
Delay To Node Discovery: 24
~    BH j
~    NTP
~    NDd
Input Data: ~   ˆ ND Æn  ¢ @<,A  ÿþ  Á
Add Node
Input Data: ~   ˆ ND ¯«  ¢ @<,=  ÿþ  Á   ö
Add Node
Input Data: ~   ˆ ND     ¢ @<,!  ÿþ  Á   U
Add Node
Node Table:
  ¢ @<,AÆn
  ¢ @<,=¯«
  ¢ @<,!
Nodes: 3
```

Node 03 gets
8 seconds * 03 = 24 seconds

All 3 other nodes found, short and long addresses above, and order reversed in node table to the left for easier processing

Figure 73: Debugged Node Discovery in Router

This output is from node 03 in the system as shown by the delay to node discovery that it gets. Although this process greatly decreases the number of restarts, there are still restarts occurring in the system. After testing the code 20 times, which amounts to 60 node discoveries performed on the Arduino code, the routers will reset approximately 12% of the time. With node discovery working most of the time in both sets of code, coordinator and router, the next stage is to debug the code that will read in test packets sent out to all nodes that are found in the node discovery. This send is done using the `apiTx` method developed in both codes. A simple message is sent composed of 25 identical characters, in this case the character 'G', with the node ID, power level of the send, and two additional bytes to identify the packet as containing cognitive information. The sends are done at random intervals

to each node found in order to avoid collisions at the receiver. If multiple packets arrive at the receiver simultaneously, the RSSI read from the packet will be less accurate since there is no way to tell from which packet the RSSI is coming.

5.2. Cognitive Parameter Acquisition

The acquisition of cognitive parameters in both the coordinator and the routers is done at each packet receive. The information will be parsed, organized, and saved to a multidimensional array quickly such that another incoming packet does not get delayed. This goal is accomplished partly by the random sends as discussed earlier and partly by the way the test packets are parsed and organized by the code.

5.2.1. Cognitive Parameter Acquisition for the Coordinator

The original cognitive parameter acquisition in the coordinator seems to run well, but the RSSI values stored into the table were constantly 0xb7, -183 dBm, which is an impossible RSSI value for this system since the sensitivity of the XBee chip is only -92dBm. To discover the problem in the code, print messages are added that show what data is being read and what data is being stored. The resulting output of the coordinator is shown below in Figure 74.

```
SD: 7e
length: 44
7e002b900013a200403c2c41c66e01474747474747474747474747474747474747474747474747474e325001323278
Mode[0]32
Mode[1]32
Find RSSI
SD: 7e
length: 7
7e06881444202648
RSSI: b7
Get RSSI: 53b700000000000000000000
Store to Table: 474747474747474747474747474747474747474747474747474e325001
```

Here the RSSI received in the signal is 0x26, but the RSSI recorded is 0xB7. This shows the error existes somewhere in the parse and not in the findRssi function.

Figure 74: RSSI Extraction Error in Coordinator

As seen above, the RSSI read in is 0x26. The error then occurs when it is stored to the array containing all the other RSSI values. The problem with the code was a math error that calculates the number of bytes read in. The code only stores the RSSI if the data read in is long enough. The value of 0xb7 that was being stored was just data that was stored in that memory location at some other point in the code. After adjusting the calculation of bytes read in and clearing out the variable used beforehand, the cognitive parameter acquisition in the coordinator is completely debugged. The debugged output is shown below in Figure 75.

```
SD: 7e
length: 44
7e002b900013a200403c2c41c66e014747474747474747474747474747474747474747474747474747474e325001323278
Mode[0]32
Mode[1]32
Find RSSI
SD: 7e
length: 7
7e06881444202 6ca
RSSI: 26
Get RSSI: 532600000000000000000000
Store to Table: 4747474747474747474747474747474747474747474747474747474e325001
```

Figure 75: Debugged Cognitive Parameter Acquisition in Coordinator

From Figure 75 it can be observed that the RSSI acquisition is functioning well and that the rest of the data is accurately stored to a multidimensional array. This is done such that the code for acquiring cognitive parameters runs quickly. Later in the code, the 25 ‘G’ characters or 0x47 as seen above will be checked for bit errors to calculate BER. The last eight bytes of information is the node ID, 4e32 translates to N02, and the power level used in the transmission, 5001 translates to P01. The position of the cognitive data in the array also indicates which RSSI corresponds to it. For example, since this data is stored in the first row of the multidimensional array, the RSSI that corresponds to that transmission is stored in the first two bytes of the RSSI array. The cognitive parameter acquisition in the coordinator is now operational and verified.

5.2.2. Cognitive Parameter Acquisition for Routers

Using similar code to the coordinator, the routers cognitive parameter acquisition code takes in the data, parses it and saves the BER, node ID, and power level of the transmission in a multidimensional array and stores the RSSI in a separate one dimensional array. The original code used in the router has an output as shown below in Figure 76.

```
Input Data: ~ +   ¢ @<,AÆn GGGGGGGGGGGGGGGGGGGGGGGGGN2P 22x
Read Mode: 22
Input Data: ~ +   ¢ @<,!   GGGGGGGGGGGGGGGGGGGGGGGGGNOP 22Î
Read Mode: 22
```

Here the code should read that the mode is mode 2 and it should parse the data, but it fails to parse.

Figure 76: Cognitive Read Error in Router

The output above shows that the parse of the data failed to occur. To debug this error a print was added into the code that displayed the variables that are used to decide whether or not to enter the parsing code. It was found that one such variable, `sysStarted`, was not being set properly. This part of the code is working after the fix and the debugged output is shown below in Figure 77.

```
Input Data: ~ +   ¢ @<,AÆn GGGGGGGGGGGGGGGGGGGGGGGGGN2P 22x
Read Mode: 22
System Started?: Yes
Get RSSI: Sÿ
Store to Table: N2P E
Input Data: ~ +   ¢ @<,!   GGGGGGGGGGGGGGGGGGGGGGGGGNOP 22Î
Read Mode: 22
System Started?: Yes
Get RSSI: SÿSÿ
Store to Table: NOP E
```

BER is calculated using the 'G' characters during the data acquisition to save memory. This is done since the routers have very little memory to work with.

Figure 77: Debugged Cognitive Parameter Acquisition in Router

The output of the routers cognitive parameter acquisition shows that the data is successfully parsed and saved to both the RSSI and input data arrays. Instead of storing all the 'G' characters like the coordinator does, the router code calculates the BER during acquisition in order to save memory space in the input array. The RSSI found in the output above is displayed as 'ÿ' which translated to 0xFF. This value for RSSI simply means that the maximum RSSI that the XBee chip can read was exceeded. Because the actual value of RSSI cannot be determined, the value 0xFF is stored to depict the maximum signal strength. With all cognitive parameters acquired, all the data is sent to the coordinator using the function `apiTx` and the routing algorithm is ready to be performed to determine the optimal routes for the system.

5.3.Routing Algorithm

The testing of the routing algorithm is the most important to this project since the goal is to route all nodes in real time using cognitive parameters. Two routing algorithms are presented, the first selects paths based on limiting the transmission power and the second selects paths based on limiting the number of nodes in any one path. There are only three possible configurations for testing discussed since only three routers and on coordinator were available to us.

Table 21: Sample Test of Path Limiting Routing Algorithm

	C	R1	R2	R3
ND	3	2	3	3
LT	6	4	6	6
CP	3	1	1	1
SR	3	1	1	1

In the sample test above the top row is organized by node, C being the coordinator and each R being a router. The first column is organized into four values recorded in each test for each node. The first value is the number of nodes found in node discovery, ND. This should be three in our system as seen in the serial port screen capture below.

```
Delay To Node Discovery: 8
~    BH j
~    NTP
~    NDd
Input Data: ~  ˆ ND     ¢ @<,!  ÿþ  Á   U
Add Node
Input Data: ~  ˆ ND ë-  ¢ @<,?  ÿþ  Á   ´
Add Node
Input Data: ~  ˆ ND XÅ  ¢ @<,=  ÿþ  Á
Add Node
Node Table:
  ¢ @<,!
  ¢ @<,?ë-
  ¢ @<,=XÅ
Nodes: 3 ←
```

Figure 78: Sample Successful Node Discovery

The second value is the number of link quality test packets successfully sent, LT, which should be six, two sent to each other node. This value was found by checking the input data messages on the serial printout as seen below in Figure 79.

```
Input Data: ~ +   ¢ @<,AØ¡ GGGGGGGGGGGGGGGGGGGGGGGGGN1P 224
Read Mode: 22
System Started?: Yes
Get RSSI: S
Store to Table: N1P E        Successful Receive from N1
                             (Router 1)
Test == 198
~ -   ¢ @<,AØ¡  GGGGGGGGGGGGGGGGGGGGGGGGGN2P 22ª
Input Data: ~  ‹ Ø¡   ú
Input Data: ~ +   ¢ @<,!   GGGGGGGGGGGGGGGGGGGGGGGGGN0P 22Î
Read Mode: 22
System Started?: Yes
Get RSSI: S S6
Store to Table: N0P E        Successful Receive from N0
                             (Coordinator)
```

Figure 79: Sample Successful Link Quality Receives

The third value, CP, is the number of successful cognitive packets sent by the routers or received by the coordinator. These are the packets that contain all link quality test packets received by the routers that need to be sent to the coordinator. These values should be one for all the routers and three

for the coordinator. The values are recorded from the data the coordinator receives as seen below in Figure 80.

```
Mode[0]33          Mode 3 (Hex 33) receives and
Mode[1]33          stores cognitive data
Data Printed:
7e001010010013a200403c2c3dfc6c0300393977
Store Cognitive Data: Node 0 to 3
PL: 1; RSSI: 1, BER: 0
Store Cognitive Data: Node 1 to 3
PL: 1; RSSI: 1, BER: 0
Store Cognitive Data: Node 2 to 3     Cognitive data received from Router 3
PL: 1; RSSI: 1, BER: 0                contains data on link qualities from all
Store Cognitive Data: Node 0 to 3     other nodes to Router 3
PL: 3; RSSI: 1, BER: 0
Store Cognitive Data: Node 1 to 3
PL: 3; RSSI: 1, BER: 0
Store Cognitive Data: Node 2 to 3
PL: 3; RSSI: 1, BER: 0
Store Node 3 Battery: 35      It also contains the current battery life
                              of the node that is sending the data.
```

Figure 80: Sample Successful Cognitive Parameter Receive

The final row, SR, is the number of source routes set in the routers and sent by the coordinator. One source route being set also includes the fact that the power level for the route was set as well. These should be one source route set for each router and three source routes successfully sent out by the coordinator. These values are found by looking at each router and seeing if the source routes and power levels were received and set. A sample of this from router one can be seen in Figure 81 below.

```
                    This print out is from Router 1

Input Data: ~     ¢ @<,!   ë-F33Â  "ë-" is the short address of Router 2
Read Mode: 33
System Started?: Yes
Set power level to 0
~   PLF
Set source route             The Source Route is set here to
~  !            ë-[ <------  multihop to the coordinator
                             through Router 2
```

Figure 81: Sample Source Route Setting from Router 1

The sample test in Table 21 on page 99 shows that the coordinator and routers two and three all executed successfully. Router one only found two nodes in its node discovery and because of this only could only successfully send four link quality packets. Although it only found two nodes, paths in this

routing algorithm have the same ranking in both directions. Due to this trait, one node not being discovered by another can be covered by the reverse path.

5.3.1. Transmission Power Limiting Routing Algorithm

Twenty tests were performed on the transmission power limiting routing algorithm. A sample test can be seen in Table 21 at the beginning of section 6.3.

This routing algorithm gives a weight value to three different parameters in order to calculate the quality of a link between two nodes. These parameters are the RSSI, the BER, and the energy cost of the system as defined by equation 20 on page 70 . To test this system, two cases were defined. The first case is when the BER of a link is 1% or greater, the link should not be used. The value of the BER for the following links was set to 5%.

- Coordinator to Router 3
- Coordinator to Router 2
- Router 1 to Coordinator
- Router 3 to Coordinator
- Router 1 to Router 2

With these links' qualities set very low for the routing algorithm, Figure 82 is the solution that should be chosen.

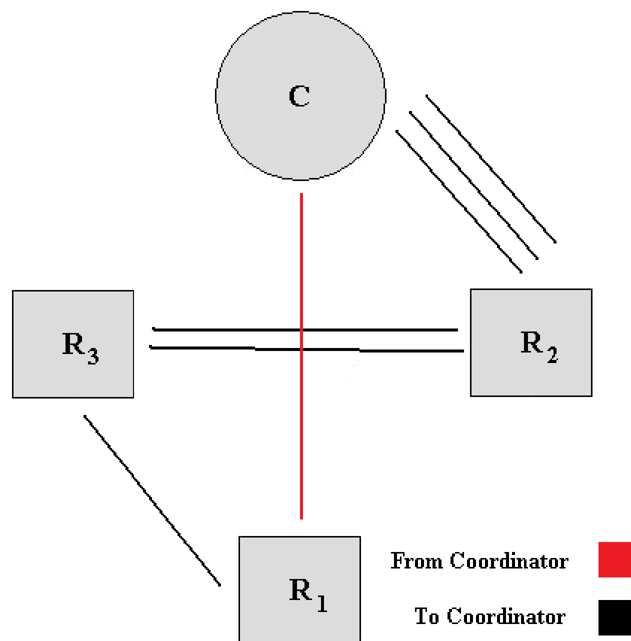


Figure 82: BER Test Case for Power Limiting Routing Algorithm

Five tests were performed using these settings in which the code was allowed to run through network optimization twice. From this, 10 sets of data were recorded in the same format as the sample test mentioned before. In every one of the 10 network optimizations, the routing algorithm chose the path shown in Figure 82, proving that the BER weighting was working correctly. This also proves that this routing algorithm is capable of routing different paths in both directions. The only problem with bi-directional routing occurs in small networks like the one we were testing the algorithm on. When data from a path is part of a failed send, the path automatically is ignored since it is impossible to know the quality of that path. In larger networks this can be ignored since many paths are available, however in this test network there are very few paths and therefore undesirable paths can sometimes be chosen.

The last 10 tests taken for this algorithm were again done using five tests that ran through the optimization process twice. These tests were performed this way due to one of the properties of the routing algorithm. The energy cost is calculated using the previous power level of a node. Since all nodes start at power level four by default, the energy cost parameter is skewed in the first run through of the code and works better in the second run through.

The second set of tests was used to check if the RSSI and battery parameters could equally affect the path chosen by the routing algorithm. The values set to test the system are shown in Table 23.

Table 22: Test Values Set for Tx Power Limiting Algorithm

Data Being Set	Parameter Being Set	Value Being Set
Coordinator to Router 1	BER	5%
Router 1 to Coordinator	BER	5%
Router 1 to Router 2	RSSI	-90 dBm
Router 1 to Router 3	RSSI	-90 dBm
Router 1	Battery Remaining	90/100
Router 2	Battery Remaining	90/100
Router 3	Battery Remaining	20/100

These values gave the system a scenario where router one should route through router 3 to get to the coordinator and the coordinator should also route through router 3 to get back to router 1. This configuration can be seen below in Figure 83.

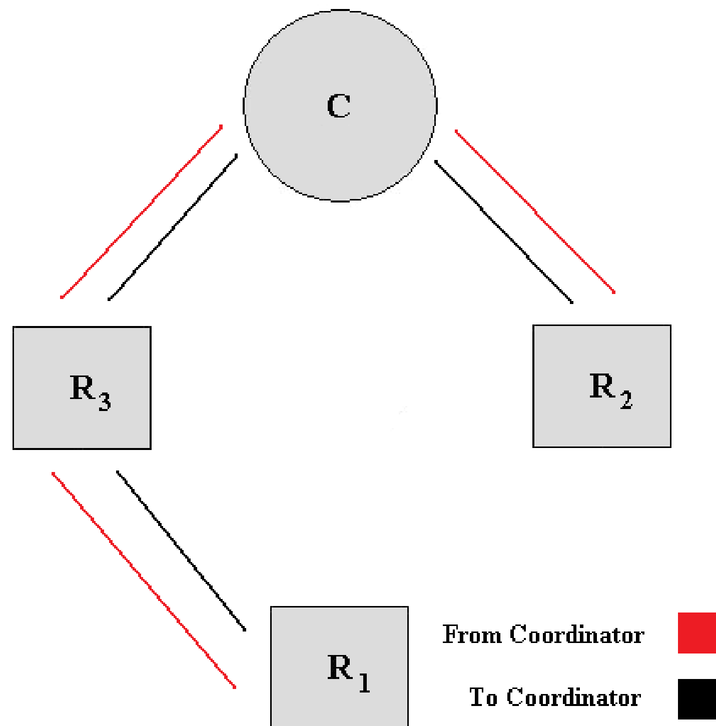


Figure 83: RSSI and Battery Level Test Case for Power Limiting Algorithm

Over ten tests the routing algorithm successfully routed through router 3 six times. Upon closer investigation, the algorithm did not route through router 3 every time due to the loss of packets during the link quality test packet sending. Since the routing is greatly affected by any one of the links having bad or no BER data, the routes in a small system such as this one cannot be easily controlled. However the routes in the system are always optimized best based on the collected data.

5.3.2. Path Limiting Routing Algorithm

Twenty tests were also performed on the path limiting routing algorithm. A sample test can be seen in Table 21 at the beginning of Section 6.3.

The first case that needs to be tested is the unaltered system. Since all the nodes in the system are currently attached to one laptop, they can each communicate with the coordinator without needing to go through another node. Since this is the case, the routing of the system should look like Figure 84 below.

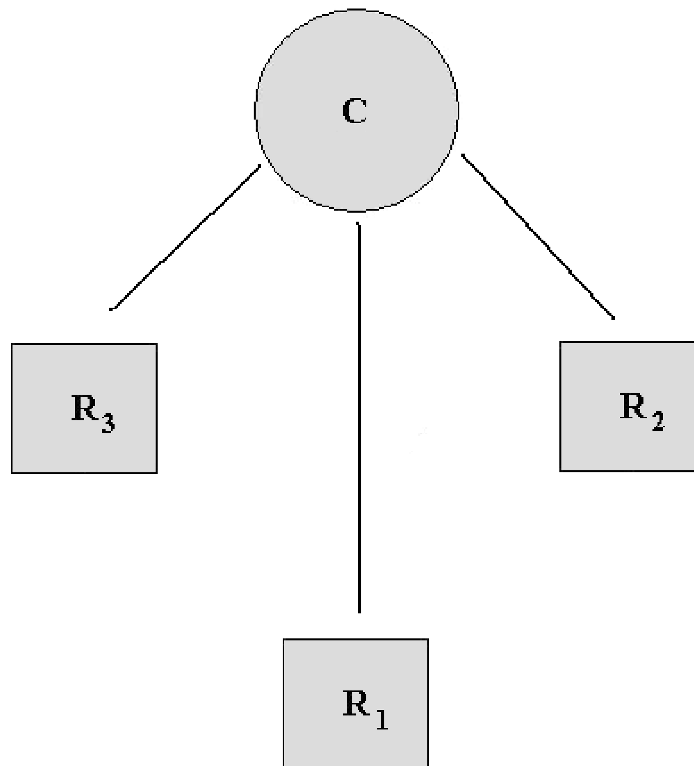


Figure 84: Routing configuration with all nodes directly connected to the coordinator

Five successive tests were performed with the values taken from the devices unaltered. In all five tests the routing algorithm set the source routes to connect each router directly to the coordinator. The main issues found in testing were the reliability of messages sent from one device to reach another. This reliability for all twenty tests done for this routing algorithm is approximately 80.6%. Considering this system does not use automatic repeat requests, this reliability is acceptable.

The second set of tests done with this algorithm set the BER of the path between the coordinator and router one to 5%. Since the threshold of the system is 1%, any links that have a BER over 1% will not be used. This test will confirm that the routing algorithm will set a route to router one by multihopping through either router three or router two. This configuration can be seen below in Figure 85.

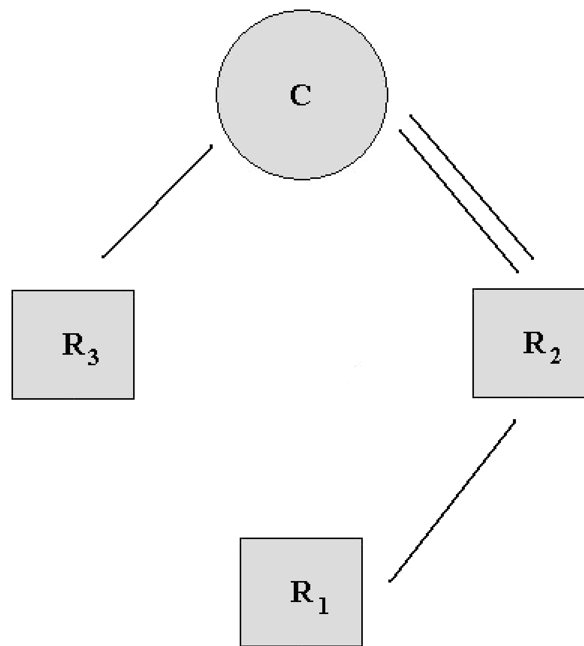


Figure 85: Routing configuration with R1 routing through R2

Five successive tests were taken again using the preset BER value for the link between the coordinator and router one. Out of the five tests, router one was set to multihop through router two twice and through router three, three times. The decision between multihopping through router two or router three was made based on which link had better RSSI. As a supplement to this test, the battery life of router three was then set to 20 and routers one and two were set to 90. Originally all nodes had their battery lives set to 53. With this set, router one should always go through router two such that it does not burden the battery life of node three by multihopping through it. Five more tests were taken to see if the routing algorithm did this correctly and all five times the route set by router one was through router two as it should have been.

The final five tests were done with the BER of three paths being corrupted. The paths between router 1 and the coordinator, router one and router two, and router three and the coordinator were all set to have a BER of 5%. The routing algorithm should not use these paths in this case since their BER is beyond the acceptable range. Not using these paths can only result in one configuration which is shown below in Figure 86.

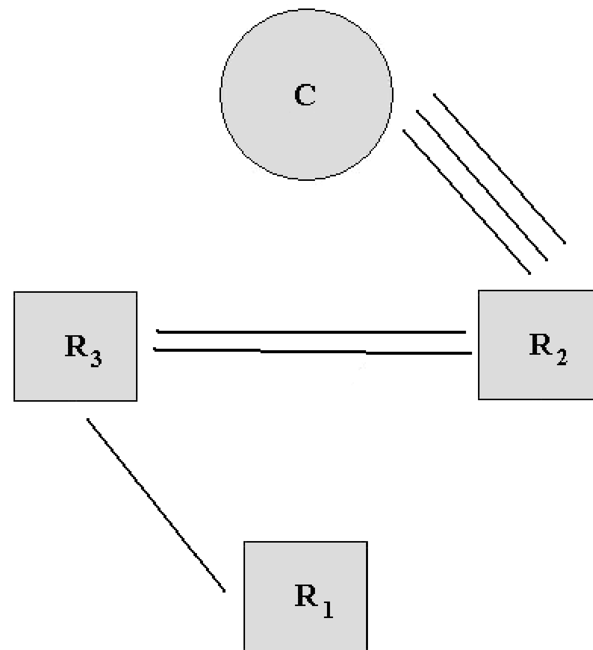


Figure 86: Routing configuration with R1 routing through R3 and R3 routing through R2

As seen above the only path available in this scenario is that router three multihop through router two to get to the coordinator and that router one multihop twice through both router three and router two to get to the coordinator. All five tests showed that the routing algorithm selected this path and that the source routes of all routers were set successfully.

5.3.3. Parsing and Setting Source Routes in the Router

When the router receives the source routes from the coordinator it must then parse them and set them using an API frame. The format of the received data containing the source route can be seen below in Figure 87.

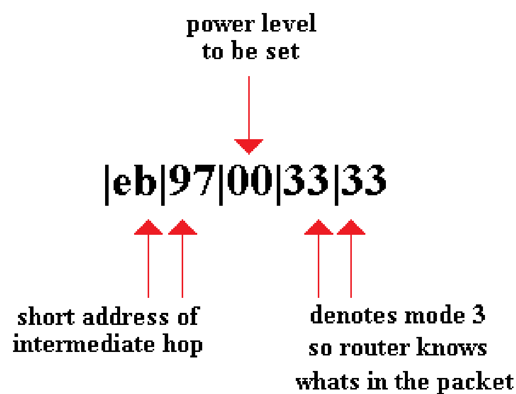


Figure 87: Layout of Source Route Data Packet

The layout above contains one short address for an intermediate hop. However, the message can include zero, one or two intermediate hop addresses depending on how many intermediate hops are in the source route that needs to be set. The main issues with coding the parse come in when attempting to index the array to exactly where each short address is. After a couple test and fixes, the addresses were all taken from the data packet and placed in an API frame that set the source route. Figure 88 below shows the source route and power level parsing and setting working in one of the 20 tests done while testing the path limiting routing algorithm.

This print out is from Router 1

```
Input Data: ~     ¢ @<,!   ë-F33Å
Read Mode: 33
System Started?: Yes
Set power level to 0
~    PLF
Set source route
~  !              ë-[
```

"ë-" is the short address of Router 2

The Source Route is set here to multihop to the coordinator through Router 2

Figure 88: Source routes and power level being set for router 1

5.4. Chapter Summary

Testing is very important for any system. In our system we documented 20 tests of each routing algorithm to verify that the routes each created were set properly. These tests were also used to verify that the rest of the system: node discovery, link quality packet sending, and cognitive parameter storage, were also working. The complete tests can be seen in the appendix B. The final results show that the node discovery is 94% successful in discovering all nodes. They also show a packet loss rate of 6.7%. On top of these tests, many undocumented tests led to the working versions that were tested. All this testing shows that the system is completely debugged and operational with the exception to the cause of the random resets that occur 12% of the time based on 120 documented trials.

6. Conclusions

The final product of the cognitive network has two fully functional coordinators. Either of these coordinators is capable of receiving the cognitive data of all nodes and the connections between them. Once the data is received the multi-hopping paths are routed and the transmission power is set in a way that optimizes both the signal quality and battery lives of every device in the system.

The objectives of this project were to create a cognitive network that is capable of handling the high demands in QoS that medical instrumentation has. Specifically it has to be reliable with respect to signal integrity, highly mobile, and energy efficient. The data being transmitted cannot be lost or corrupted in transit. It must optimize its performance quickly and frequently while nodes move around. Finally, it must preserve the battery levels of all devices in the system such that frequent charging is not required.

The cognitive parameters RSSI, BER, and battery level are found by sending a message to all of the nodes in the network, telling them to initiate the environment sensing process. This process has two phases. These phases are the node discovery, and the link assessment.

The node discovery is used to find all of the nodes in the network that are within one hop of communication. This is the first step in environment sensing and our first success at a cognitive system. The nodes become aware of all of its neighbors that it can connect to. The device now has a list of all the addresses of its neighbors.

The second phase of environment sensing is link assessment. In this step we implemented an efficient way to calculate the BER and RSSI of a test message. This is the second success of implementing a cognitive network. The node is now aware of both its neighbors and the quality of the link between them. It can now send this data to the coordinator such that the coordinator can be aware of the possibilities of connection for the entire network.

The environment sensing was extensively tested and two main problems can happen during the process. For still unknown reasons, the routers randomly reset. This always happens during the node discovery and takes the node out of consideration for the optimization. It will not send link assessment packets and cannot communicate to the coordinator, its cognitive assessment of its environment. The other problem is occasionally the packet containing the cognitive data is dropped when being transmitted to the coordinator. Redundancy was added to this message because of its high importance but there are still cases where the data is not received. After 40 trials it was calculated that the environment sensing successful 87.5% of the time.

This is where the two different coordinators can optimize the system using radio resource management techniques. There are two methods of optimization. Both of them set the ideal power level of the node and establish source routes for the multi-hopping path. One of the routing algorithms is a demonstration of an optimization using a hybrid of two RRM techniques. The other is an algorithm geared toward optimizing the network specific to the hardware in it.

The first algorithm explored was the transmission power limiting algorithm. This bi-directional algorithm assigns link costs for all connections in the network. It is a weighted value that is highly adjustable for the needs of the user. The link cost is a combination of the RSSI, BER, battery level and change in transmission power of the node with respect to the previous optimization. It then traces all possible paths and assigns a path cost equivalent to the highest link cost in the route. The path of lowest cost is selected and the minimum power levels of all nodes in the path are set.

The second algorithm optimizes the battery life of the XBee hardware that was used for the network by means of restricting the number of paths that a low battery node may be in. For this reason it has been named the path limiting algorithm. This uses a ranking of cognitive parameters system. It first eliminates all redundant and unreliable, with respect to BER, links such that the final options of paths circumvent specific nodes. A preliminary path is then set using RSSI values and the remaining battery life is calculated in hours. This path is adjusted and battery life is compared to the original value until the low battery nodes have maximized their remaining time of operation.

The two routing algorithms are the most important successes in creating a cognitive network. They embody the spirit of cognitive radio by taking the cognitive information of the surroundings and using the radio resource management techniques available to optimize the network.

Many trials of testing were performed on both algorithms to confirm that nodes were routed correctly given certain defined environments. It was found that both algorithms route the nodes correctly and set the correct power levels every time that they are called. The only cases when routing is calculated in a different way than is expected were the trials where nodes were reset or cognitive packets were dropped. In these cases, the routes are based on the fragmented data received. The routes and power levels are still calculated correctly for the data available, but may be different than expected.

The cognitive network designed was an embodiment of cognitive radio and is capable of handling moderate QoS requirements for medical instrumentation. This cognitive network may, in the future, be

used to make wireless medical instrumentation as reliable as it possible when used to potentially save lives.

7. Future Work

This section discusses various considerations and ideas that were outside the scope of this ten week project but could be added to the cognitive system to improve upon and expand its functionality. The full Mitola cognitive radio has never been fully implemented and the concept of cognitive radio is still expanding a decade after its original conception, so there are many opportunities and new directions to build upon this project. The implementation of radio resource management and intelligent routing through cognitive parameter feedback in this project can be improved upon in numerous scopes.

The overall reliability of the system of the system can be improved to ensure that the system is effective and capable of real world application such as medical facilities. The speed of the system can be increased so that the updates can be more impactful and reflective of real-time channel and situational changes. The fine tuning of the radio resource management through more analytical consideration of the cognitive feedback could be improved to increase responsiveness and benefit of the system. Improvements of the precision and accuracy of the cognitive parameter acquisition could make the radio resource management more effective in adapting to subtler channel and network changes. The addition of new cognitive parameters and radio resource techniques could be implemented to make the system more comprehensive or application oriented. The porting of the cognitive system to new radio protocols could open up new possibilities for its capabilities.

7.1. Battery Monitoring

The system designed in this project is geared towards incorporating battery capacity of each remote node into routing algorithms, which normally only consider RSSI and BER. This idea is novel and has applications in medical, emergency and sensor networks. A major issue, which was not addressed in the scope of this project, was the issue of battery charge acquisition. Without a sub circuit that lets the routers know how much charge they have left, the system remains incomplete and cannot be applied to actual networks. Battery monitoring exists in many handheld devices and radios that communicate to the user to change the battery. For the purposes of the cognitive network designed in this project, the accuracy of the battery life monitoring should be high enough so that the battery life calculations are as close to reality as possible and the battery charge weights the most effective in decision making.

There are many techniques to measure battery charge such as high impedance battery output voltage measurements that can compared to a standard discharge curve of the battery used. This method can be implemented with a voltage divider that has output impedance lower than 20kΩ, as

specified by the AT168Mega microcontroller datasheet, and a switching MOSFET to cut off the voltage divider current when the microcontroller is not taking a measurement.

It is important to compensate this reading by factoring in internal resistance of the battery and the known affects of the temperature on the battery. The microcontroller can then convert the voltage to charge by comparing it to a programmed discharge curve. However, the actual discharge curve can degrade over time as the state of health of the battery declines.

A recommended method is combining the voltage measurement method with “coulomb counting” or current integration. An amp meter can be logged into the microprocessor which integrates the current through time to find the charge consumed. The main issue with this method is that it suffers from drift which will make the derived measurement more inaccurate over time. This is why the method should occasionally be synchronized or calibrated using the voltage reading. The coulomb counting can also be used to measure the state of health of the battery over time to update the programmed discharge curve. If the battery monitoring were to be implemented, its integration would also require the adjustment of the battery charge weight to adapt the impact of battery life on the transmission power limiting routing algorithm to fit the application.

7.2. Transmission Power in Path Limiting Algorithm

The variable transmission power was not factored into the battery life calculation of the path limiting algorithm because the XBee Series 2 ZB modules have a constant current draw of 40mA at variable transmission power. The transmission power limiting algorithm factors in transmission power in its energy cost because it simulates modules with variable current draw. The battery life calculation of the path limiting algorithm can be adjusted to factor in variable current draw from different transmission power levels and should be if the system were to be ported to a radio which has a variable transmitting current draw.

To incorporate transmission power, adjustments must be made to the battery life in equation 23 that is used to make decisions between routes. Before each battery life calculation is made the minimum transmission power level required to accommodate all the links QoS must be found for each node. This is done in the same manner that the final transmission power is derived for each node, with the equation:

$$P_{Tx} = \max(P_{TxLink})_{hops}. \tag{26}$$

The transmission power must then be mapped to a variable current draw using the radio manufacturer specifications. The relationship could be linear, exponential or non-linear depending on the radio. It is important to decipher this expression and program it into the microprocessor so that the microprocessor can associate the known required transmission power of each node to the variable current draw that is used in the updated version of equation 23, which is:

$$t_{battery} = \frac{Q}{N_{routes}\ D\left(\frac{I_{Rx}}{2}+\frac{I_{TxVar}}{2}\right)+(1-N_{routes}\ D)I_{idle}+I_{microcontroller}}. \quad (27)$$

The algorithm would then use the battery life of each node in the same manner it did before, prioritizing the nodes with the least battery life. The only adaptation is to make sure that power level is incorporated into the battery life expression before each time it is calculated. With these simple additions, the system could be suitable for any battery powered radio with variable current draw based on the transmission power.

7.3. Cognitive Radio in other Protocols

Two other possible applications of cognitive radio in medical instrumentation are protocol selection, and ultra wide band (UWB) implementation. Both of these alternate applications were considered as directions for this project. Either would be a good extension of the results achieved in this report.

Protocol selection would require a device to have multiple protocols available for transmission and reception. As each protocol has its own strengths and weaknesses, the selection of which to use will be specific to the type of data that needs to be transmitted. As an example, high data-rate usage like image processing would require WiFi to be used. However, if WiFi is unavailable due to heavy traffic, or interference, the data may attempt to be transmitted using Bluetooth. If a signal needs high security and/or low data rate with high energy efficiency, it may be more lucrative to use the Bluetooth or ZigBee transmitter on the device. This project would require detection of the type of media that needs to be broadcast and protocol synchronization with the receiving device before a message can be sent.

Ultra-wide band (UWB) is a fairly new protocol with a lot of potential. A project more geared toward the Physical Layer of a UWB device would be a good extension of this paper. Multiple access methods must be explored and timing synchronization needs to be written before pursuing a cognitive network. It is recommended to incorporate spectrum-sensing techniques to improve the adaptive filters and equalizer of the PHY layer and to select the bands to use for multiple access.

8. References

[1] J. I. Mitola and G. Q. J. Maguire, "Cognitive Radio: Making Software Radios More Personal," *IEEE Personal Communications Magazine*, pp. 13-18, Aug. 1999.

[2] T. R. Newman, et al., "Cognitive Engine Implementation for Wireless Multicarrier Transceivers," The University of Kansas, 2006.

[3] M. Srbinovska, C. Gavrovski, and V. Dimcev. (2008, Sep.) Localization Estimation System Using Measurment of RSSI Based on ZigBee Standard. [Online]. http://fett.tu-sofia.bg

[4] G. Giancola, C. Martello, F. Cuomo, and M.-G. Di Benedetto, "Radio Resource Management in Infrastructure-Based and Ad Hoc UWB Networks," *Wireless Communications and Mobile Computing*, pp. 5:581-597, 2005.

[5] A. Sahai and D. Cabric. (2005, Nov.) Electrical Engineering and Computer Science UC Berkeley. [Online]. http://www.eecs.berkeley.edu/~sahai/Presentations/DySPAN05_part2.ppt

[6] D. Cabric, S. M. Mishra, and R. W. Brodersen, "Implementation Issues in Spectrum Sensing for Cognitive Radios," Berkeley Wireless Research Center, Berkeley, Lab Research Report, 2005.

[7] IEEE Computer Society. (2006, Sep.) IEEE Standards Association. [Online]. http://standards.ieee.org/

[8] R. Peng, S. Mao-heng, and Z. You-min, "ZigBee Routing Selection Strategy Based on Data Services and Energy-balanced ZigBee Routing," *IEEE Xplore*, Aug. 2009.

[9] T.-W. Song and C.-S. Yang, "A Connectivity Improving Mechanism for ZigBee Wireless Sensor Networks," *IEEE Xplore*, Aug. 2009.

[10] E. Ayanoglu, R. D. Gitlin, T. F. La Porta, S. Paul, and K. K. Sabnani, "Adaptive Forward Error Correction System," U.S. System Design 5,600,663, Feb. 4, 1997.

[11] Duke-River Engineering. (2006) Medical Wireless Technologies. Power Point.

[12] R. Flickenger. (2007, Dec.) Wireless Networking in the Developing World. [Online]. http://wndw.net/

[13] IEEE. (2007, Sep.) IEEE 802.11g-2003: Further Higher Data Rate Extension in the 2.4 GHz Band. [Online]. http://standards.ieee.org/

[14] D. Networks. (2009, Aug.) ZigBee Primer. [Online]. http://www.daintree.net/

[15] Digi International Inc. (2009, Aug.) XBee®/XBee-PRO® ZB RF Modules. [Online]. www.digi.com

[16] FlexiPanel. (2008, Oct.) UZBee Plus: ZigBee / 2.4GHz IEEE 802.15.4 RF transceiver with USB interface. [Online]. http://www.flexipanel.com

[17] FlexiPanel. (2007, Feb.) UZBee: 2.4GHz IEEE 802.15.4 RF transceiver with PIC microcontroller USB interface. [Online]. http://www.flexipanel.com

[18] MaxStream. (2006, Oct.) XBee™/XBee-PRO™ OEM RF Modules. [Online]. ssdl.stanford.edu/

[19] FTDI Chip. (2009) FTDI Chip Web site. [Online]. www.ftdichip.com

[20] Flexipanel. (2006, Dec.) MACdongle™ IEEE 802.15.4 MAC layer firmware for UZBee USB adapter. www.flexipanel.com.

[21] M. Barry. (2009, Aug.) XBee Web Update Programs. Arduino Code.

[22] G. J. Loubser, "Demodulation and Display of th RSSI Signal of the MaxStream™ XBee," A South African Sensor Network Initiative Final Year Undergraduate Project, Jun. 2006. [Online]. http://asasni.cs.up.ac.za/

A. Appendix A

Transparent Mode Methods

setupXbee

```
void setupXbee(void) {              //***
  // open the Xbee interface
  Serial.begin(9600);
  delay(1000);
  Serial.println("+++");
  delay(1000);
  Serial.println("ATSM1");
  delay(1000);
  Serial.println("ATD70");
  delay(1000);
  Serial.println("ATRPFF");
  delay(1000);
  Serial.println("ATPL0");
  delay(1000);
  Serial.println("ATCN");
  delay(1000);
  digitalWrite(XBee_pin, LOW);
}
```

findRssi

```
int findRssi() {
  int myRssi = 0, rssiCounter = 0, totalCount = 0, errCount = 0, changes = 0, state;
  if(digitalRead(XBee_pin2) == HIGH)
    state = 1;
  else
    state = 0;
  while(changes<7){
    minR = 0;
    maxR = 0;
    int temp;
    if(digitalRead(XBee_pin2) == HIGH)
      temp = 1;
    else
      temp = 0;
    if(state != temp){
      changes++;
      state = (state + 1) % 2;
    }
    if(changes > 0){
      if(temp)
        rssiCounter++;
      totalCount++;
    }
    errCount++;
    if(errCount > 4000){
      if(state == 1){
        maxR = 1;
        minR = 0;
      }
      if(state == 0){
        maxR = 0;
        minR = 1;
      }
      changes = 8;
    }
  }
```

```
  int rssiPercent = ((double)rssiCounter/totalCount) * 100;


  myRssi = ((10.24 * rssiPercent) - 295) / 17.5 - 92;

  if(maxR == 1)
    myRssi = 999;
  if(minR == 1)
    myRssi = -999;

  rssiCounter = 0;
  totalCount = 0;
  errCount = 0;
  //myRssi = 73;
  return myRssi;
}
```

powerLevel

```
void powerLevel(){

  if(rssi < POOR_BOUND && myPower<4){
    myPower++;
    delay(1000);
    Serial.println("+++");
    delay(1000);
    Serial.print("ATPL");
    Serial.println(myPower);
    delay(1000);
    Serial.println("ATCN");
    delay(1000);
  }
  else if(rssi > GOOD_BOUND && myPower>0){
    myPower--;
    delay(1000);
    Serial.println("+++");
    delay(1000);
    Serial.print("ATPL");
    Serial.println(myPower);
    delay(1000);
    Serial.println("ATCN");
    delay(1000);
  }
}
```

sendSensorData

```
void sendSensorData(int nodeID, int powerLevel, int battery, int rssi)
{
  Serial.print('N');
  if (nodeID < 10) {
    Serial.print('0');
    Serial.print(nodeID);
  }
  else if ((nodeID < 100) && (nodeID >= 10)) {
    Serial.print(nodeID);
  }
  else
    Serial.print("XX");

  // make sure we always get a 3 digit number sent

    Serial.print('P');
  if (powerLevel < 10) {
    Serial.print(powerLevel);
```

```
  }
  else
    Serial.print('X');


  Serial.print('B');
  if (battery < 10) {
    Serial.print(battery);
  }
  else
    Serial.print('X');

  Serial.print('S');
  if(rssi == -999)
    Serial.println("No");
  else if(rssi == 999)
    Serial.println("Mx");
  else if(rssi>-99){
    rssi*=(-1);
    Serial.println(rssi);
  }
  else
    Serial.println("XX");
}
```

Transparent Mode Setup and Loop

Router

```
void setup()
{
  Serial.begin(9600);
  pinMode(XBee_pin2, INPUT);
  pinMode(XBee_pin, OUTPUT);
  digitalWrite(XBee_pin, LOW);
  setupXbee();
  delay(1000);
}

void loop() {

  int newData;
  int validData;
  int count = 0, dataReady=0;
  int i;
  int moreData;

  node = 0;
  rssi = 0;
  power = 0;
  battery = 0;

  // Read data from the serial input
  //  we only have a 52 byte buffer when the buffer is full or
  // after every linefeed (ASCII 10) process the data in the buffer
  moreData = 1;
  while (moreData == 1) {


    // if there is no data in the buffer then exit the loop
    if(!Serial.available())
      moreData = 0;
    // Find own Cognitive Parameters
```

```
//insert Rssi and Power Level methods

//Rssi
myRssi = findRssi();

//Battery Level
//

// if there is data in the buffer procees it
newData = 0;
count = 0;
while(Serial.available()){
  newData = 1;
  char temp = Serial.read();
  if(((temp-48)>=0 && (temp-48)<=9) && dataReady==0){
    dataReady = 1;
  }
  if(dataReady==1){
    if ((count  >= 51) || (temp == 10)) {
      break;
    }
    else{
      inputData[count] = temp;
      count++;
    }
  }
}
// If data is received, parse the rssi of previous node
if(newData == 1) {

  firstRead = inputData[0]-48;
  i = (firstRead * 10) + 8;
  firstRead++;

  Serial.println(inputData);
  Serial.flush();

  inputData[0] = firstRead+48;
  if (inputData[i] == 'S') {
    rssi = 0;
    rssi += ((inputData[i + 1] - 48 ) * 10);
    rssi += (inputData[i + 2] - 48);
    rssi *= -1;
    //Serial.print("Rssi: ");
    if(rssi == -367){
      //Serial.println("No");
      rssi = -999;
    }
    else if(rssi == -362){
      //Serial.println("Mx");
      rssi = 999;
    }
    else{
      //Serial.println(rssi);
    }

    //Power Level
    powerLevel(rssi);

    int validData = 0;

  }
  else validData = -1;
}
```

```
  }

  // Final Serial Print of new Header
  delay (10000);
  Serial.print(inputData);
  sendSensorData(NODE_ID, myPower, myBattery, myRssi);


  for (i = 0; i++; i < count) {
    inputData[count] = 0;
  }
}
```

API Mode Router Methods

findNetwork

```
void findNetwork(){
  byte panID[]= {
    0x7E, 0x00, 0x05, 0x08, 0x01, 0x49, 0x44, 0x00,
    0x71                                       };
  byte scanChan[]= {
    0x7E, 0x00, 0x06, 0x08, 0x01, 0x53, 0x43, 0xFF, 0xFF,
    0x62                                       };
  byte applyChanges[]= {
    0x7E, 0x00, 0x04, 0x08, 0x01, 0x41, 0x43, 0x72
};
  int count = 0;
  Serial.begin(9600);
  delay(100);
  Serial.print("Pan ID setup: ");
  Serial.write(panID, 9);
  Serial.println();
  delay(100);
  Serial.print("Scan Channels: ");
  Serial.write(scanChan, 10);
  Serial.println();
  delay(100);
  Serial.print("Apply any changes: ");
  Serial.write(applyChanges, 8);
  Serial.println();
  delay(100);
  Serial.flush();
}
```

receiveData

```
void receiveData(){
  int count = 0;
  int length = 0;
  int newData;
  int j;
  int currPlace = 0;
  if(Serial.available()){
    delay(50);
    count = 0;
    inputData[count] = Serial.read();
    count++;
    if(inputData[count-1] == 0x7E){
      delay(2);
      while(count<3){
        inputData[count] = Serial.read();
        count++;
```

```
      }
      length = inputData[2] + 4;
      //delayMicroseconds(834*length);        // 5/6 milliseconds per byte
      if(length > 50){
        Serial.println("Read Error: Invalid Length");
        return;
      }
      delay(length);
      while(length > count){
        newData = 1;
        inputData[count] = Serial.read();
        count++;
      }
      if(newData == 1){
        Serial.print("Input Data: ");
        Serial.write(inputData, count);
        Serial.println();
        if(inputData[3] == 0x90 && inputData[count - 3] > 0x30 && inputData[count - 3]
< 0x40){
          Serial.print("Read Mode: ");
          mode[0] = inputData[count - 3];
          mode[1] = inputData[count - 2];
          Serial.print(mode[0]);
          Serial.println(mode[1]);
          Serial.print("System Started?: ");
          if(sysStarted>0)
            Serial.print("Yes");
          else
            Serial.print("No");
          Serial.println();
          if(mode[0] == mode[1])
            validMode = 1;
          else
            validMode = 0;
          if(validMode==1 && mode[0]==0x32 && sysStarted>0){
            int currPlace = currLink*2;
            inputRssi[currPlace] = 'S';
            inputRssi[currPlace+1] = findRssi();
            Serial.print("Get RSSI: ");
            Serial.write(inputRssi, 12);
            Serial.println();
            byte signalBer = berCheck();
            for(j=0; j<4; j++)
              linkTable[currLink][j] = inputData[j+40];
            linkTable[currLink][4] = 'E';
            linkTable[currLink][5] = signalBer;
            linkTable[currLink][6] = inputData[count-4];
            Serial.print("Store to Table: ");
            Serial.write(linkTable[currLink], 7);
            Serial.println();
            currLink++;
          }
          else if(validMode==1 && mode[0]==0x39 && sysStarted>0){
            Serial.println("Send Successful");
            coordReceive = 2;
          }
        }
      }
    }
  }
}
```

modeOne

```
void modeOne(){
  int addrNum;
  int i;
  int j = 0;
  Serial.println("Mode 1");
  powerLevel(3);
  delay(100);
  addrNum = nodeDiscovery(); //Make Node Discovery Request
  if(addrNum > 0)
    sendDiagnostic(addrNum);
  while(j < 10){
    receiveData();
    delay(100);
    j++;
  }
  Serial.println("Link Table:");
  for(i=0; i<currLink; i++){
    for(j=0; j<7; j++)
      Serial.write(linkTable[i][j]);
    Serial.println();
  }
  Serial.print("Links: ");
  Serial.write((byte)(currLink + 0x30));
  Serial.println();
  if(mode[0] == 0x31)
    mode[0] = 0x00;
  return;
}
```

modeTwo

```
void modeTwo(){
  Serial.println("Mode 2");
  byte coordAddr[] = {
    0,0,0,0,0,0,0,0,0,0                        };
  byte txArray[(currLink*7+7)];
  int i = 0;
  int j = 0;
  //find own cognitive parameters
  for(i=0; i<currLink; i++){
    txArray[j] = 'N';
    txArray[j+1] = linkTable[i][1];
    txArray[j+2] = 'S';
    txArray[j+3] = inputRssi[((i*2)+1)];
    txArray[j+4] = 'E';
    txArray[j+5] = linkTable[i][5];
    txArray[j+6] = linkTable[i][6];
    j += 7;
  }
  byte myBattery = 0x5A;
  txArray[j] = 'N';
  txArray[j+1] = (byte)NODE_ID;
  txArray[j+2] = 'B';
  txArray[j+3] = myBattery;
  txArray[j+4] = (byte)currLink;
  txArray[j+5] = 0x33;
  txArray[j+6] = 0x33;
  apiTx(txArray, coordAddr, 0);
  int k = 0;
  while(coordReceive < 2){
    receiveData();
    delay(100);
    k++;
```

```
    if(k>50){
      k=0;
      coordReceive++;
      apiTx(txArray, coordAddr, 0);
    }
  }
  coordReceive = 0;
  for(i=0; i<currLink*6+7; i++)
    txArray[i] = (byte)0;
  for(i=0; i<currLink*2; i++)
    inputRssi[i] = (byte)0;
  mode[0] = 0x00;
}
```

modeThree

```
void modeThree(){
  byte sourceRoute[7];
  byte hops;
  int power;
  int i;
  Serial.println("Awesome");
  hops=(inputData[2]-14)/2;
  power=inputData[15+2*hops];
  Serial.print("Set power level to ");
  Serial.println(power);
  powerLevel(power);
  sourceRoute[0]='R';
  sourceRoute[1]=hops;
  for(i=0; i<2*hops; i++)
    sourceRoute[i+2]=inputData[i+15];
  sourceRoute[2*hops+2]='R';
  Serial.println("Set source route");
  createRoute(sourceRoute);
  clearArrays();
  sysStarted = 0;
  currLink = 0;
}
```

nodeDiscovery

```
int nodeDiscovery(){
  byte broadHops[] = {
    0x7E, 0x00, 0x05, 0x08, 0x01, 0x42, 0x48, 0x02, 0x6A
}; //sets broadcast hops to 1
  byte discTimeout[] = {
    0x7E, 0x00, 0x05, 0x08, 0x01, 0x4E, 0x54, 0x50, 0x04
}; // sets node discovery timeout to 8s
  byte nodeDisc[] = {
    0x7E, 0x00, 0x04, 0x08, 0x01, 0x4E, 0x44, 0x64
}; // requests node discovery
  int i = 0;
  int j = 0;
  int count = 0;
  int addrNum;
  for(i=0; i<3; i++){
    for(j=0; j<10; j++)
      nodeTable[i][j] = 0;
  }
  i = -1;
  j = 0;
  Serial.print("Delay To Node Discovery: ");
  Serial.println((NODE_ID-0x30)*8);
  delay(8000*(NODE_ID-0x30));
  delay(200);
  Serial.write(broadHops, 9);
```

```
  Serial.println();
  delay(200);
  Serial.write(discTimeout, 9);
  Serial.println();
  delay(200);
  Serial.write(nodeDisc, 8);
  Serial.println();
  delay(200);
  Serial.flush();
  while(1){
    if(count > 150)
      break;
    for(int i=0; i<50; i++){
      inputData[i] = (byte)0;
    }
    receiveData();
    if((inputData[5]) == 'N' && (inputData[6]== 'D')){
      i++;
      Serial.println("Add Node");
      for(j=0; j<8; j++){
        nodeTable[i][j] = inputData[j+10];
      }
      for(j=0; j<2; j++){
        nodeTable[i][j+8] = inputData[j+8];
      }

      if((nodeTable[i][4] == 0x00) || (nodeTable[i][4] == 0xFF)){
        i--;
        Serial.println("Bad Node");
      }
    }
    addrNum = i;
    count++;
    delay(55);
  }
  Serial.println("Node Table: ");
  for(i=0; i<addrNum+1; i++){
    for(j=0; j<10; j++)
      Serial.write(nodeTable[i][j]);
    Serial.println();
  }
  Serial.print("Nodes: ");
  Serial.write((byte)(addrNum+0x31));
  Serial.println();
  delay(8000*(4-(NODE_ID-0x30)));
  rssiStart();
  delay(1000);
  return(addrNum+1);
}
```

powerLevel

```
void powerLevel(int power){
  delay(100);
  byte powerAdjust[] = {
    0x7E, 0x00, 0x05, 0x08, 0x00, 0x50, 0x4C                          };
  Serial.write(powerAdjust, 7);
  Serial.write((byte)power);
  byte checkS = 0xFF - ((0xA4 + (byte)power) % 0x100);
  Serial.write((byte)checkS);
  Serial.println();
  delay(100);
}
```

arrayLength

```
// Calculates Array Length as a byte using a the pattern of the Array
// Length works for Cognitive Sends and Address Sends
byte arrayLength(byte inArray[]){
  byte length = 0;
  int type = 0;          // Address Send (Type 0) or Cognitive Data Send (Type 1)
  if(inArray[length]=='R'){
    length++;
    while(inArray[length]!='R'){
      length++;
    }
    return (length-1);
  }
  while(type < 2){
    while(inArray[length] == 'N'){
      length += 7;
      type = 1;
    }
    if(type == 0){
      length = 31;
      return length;
    }
    else if(type == 1){
      return length;
    }
  }
}
```

apiTx

```
void apiTx(byte txData[], byte destAddress[], byte hops){
  byte arrayLen = arrayLength(txData);          // Determine the length of the array
to be sent
  arrayLen += 0x0E;                             // Increase by the Length of Data
before Payload in the API Frame
  byte checksum = 0;                            // Increase Checksum as we go and calc
at bottom

  // Start Delimiter
  Serial.write(0x7E);
  // Length
  Serial.write((byte)0);
  Serial.write(arrayLen);
  // Frame Type
  Serial.write(0x10);
  checksum += 0x10;
  // Frame ID
  Serial.write(0x01);
  checksum += 0x01;
  // Long and Short Destination Addresses
  for(int i=0;i<10; i++){
    Serial.write(destAddress[i]);
    checksum += destAddress[i];
  }
  // Broadcast Radius
  Serial.write(hops);
  checksum += hops;
  // Options
  Serial.write((byte)0);
  // Packet Data
  int i=0;
  for(i=0;i<arrayLen-0x0E;i++){
    Serial.write(txData[i]);
    checksum += txData[i];
```

```
  }
  // Checksum
  checksum = checksum % 0x100;
  checksum = 0xFF - checksum;
  Serial.write(checksum);
  Serial.write('\n');
  Serial.write('\r');
}
```

createRoute

```
void createRoute(byte sourceRoute[]){
  byte arrayLen = arrayLength(sourceRoute);          // Determine the length of the
array to be sent
  arrayLen += 0x0D;                                // Increase by the Length of Data
before Payload in the API Frame
  byte checksum = 0;                               // Increase Checksum as we go and calc
at bottom
  byte coordAddress[]={
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00    };

  // Start Delimiter
  Serial.write(0x7E);
  // Length
  Serial.write((byte)0);
  Serial.write(arrayLen);
  // Frame Type
  Serial.write(0x21);
  checksum += 0x21;
  // Frame ID
  Serial.write((byte)0);
  // Long and Short Destination Addresses
  for(int i=0;i<10; i++)
    Serial.write(coordAddress[i]);
  Serial.write((byte)0);
  // Packet Data
  int i=0;
  for(i=1;i<arrayLen-0x0C;i++){
    Serial.write(sourceRoute[i]);
    checksum += sourceRoute[i];
  }
  // Checksum
  checksum = checksum % 0x100;
  checksum = 0xFF - checksum;
  Serial.write(checksum);
  Serial.write('\n');
  Serial.write('\r');
}
```

sendDiagnostic

```
//Send Diagnostic Packets
void sendDiagnostic(int addrNum){
  int testPower = 1;
  byte berData[31];
  int i, j;
  currLink = 0;
  for(i=0;i<25;i++)
    berData[i] = 'G';
  berData[25] = 'N';
  berData[26] = (byte)NODE_ID;
  berData[27] = 'P';
  berData[29] = 0x32;
  berData[30] = 0x32;
  //needs actual number of addresses set as loop scope
  while(testPower<4){
```

```
    berData[28] = testPower;
    powerLevel(testPower);
    for(i=0; i<addrNum; i++){
      int randDelay = random(0,(200/addrNum));
      int exDelay = 200/addrNum - randDelay;
      Serial.print("Test == ");
      Serial.println(addrNum*(randDelay+exDelay));
      j = 0;
      while(randDelay > j){
        receiveData();
        delay(100);
        j++;
      }
      apiTx(berData, nodeTable[i], 1);
      j = 0;
      while(exDelay > j){
        receiveData();
        delay(100);
        j++;
      }
    }
    testPower += 2;
  }
  j = 0;
  while(100 > j){
    receiveData();
    delay(100);
    j++;
  }
  powerLevel(4);
  delay(100);
}
```

rssiStart

```
void rssiStart(){
  Serial.print("Start RSSI: ");
  byte rssiStart[] = {
    0x7E, 0x00, 0x05, 0x08, 0x01, 0x52, 0x50, 0xFF, 0x55                         };
  Serial.write(rssiStart, 8);
  Serial.println();
  Serial.flush();
}
```

findRssi

```
//Rssi Check
byte findRssi() {
  int maxR = 0;
  int minR = 0;
  byte myRssi = 0;
  int rssiCounter = 0, totalCount = 0, errCount = 0, changes = 0, state;
  if(digitalRead(XBee_pin2) == HIGH)            // Capture Starting State of the PWM
Signal
    state = 1;
  else
    state = 0;
  while(changes<7){
    minR = 0;
    maxR = 0;
    int temp;
    if(digitalRead(XBee_pin2) == HIGH)          // Temp changes as the state changes
      temp = 1;
    else
      temp = 0;
```

```
    if(state != temp){                              // Increment Changes whenever State
and Temp Disagree
      changes++;
      state = (state + 1) % 2;                      // Change State to make it equal to
Temp again
    }
    if(changes > 0){                                // Ignore the First Change due to
random starting place
      if(temp)
        rssiCounter++;
      totalCount++;
    }
    errCount++;
    if(errCount > 4000){                            // If PWM is 100% or 0%, jump out and
set Max or Min Signal Flags
      if(state == 1){
        maxR = 1;
        minR = 0;
      }
      if(state == 0){
        maxR = 0;
        minR = 1;
      }
      changes = 8;                                  // Set Changes to Jump out of while
loop
    }
  }
  int rssiPercent = ((double)rssiCounter/totalCount) * 100;    // Calc Duty Cycle of
PWM
  myRssi = (((10.24 * rssiPercent) - 295) / 17.5 - 92) * -1;   // Convert Duty Cycle
to RSSI in dBm
  if(maxR == 1)
    myRssi = 0x01;                                            // 0xFF = Maximum Signal
Strength
  if(minR == 1)
    myRssi = 0xEE;                                            // 0xEE = No Signal
Detected
  return myRssi;
}
```

berCheck

```
//BER Check
byte berCheck()
{
  byte temp=0x00;
  int errors=0;
  for(int j=0; j<25; j++)
  {
    temp = inputData[j+15] ^ 0x47;
    for(int i=0; i<8; i++)
    {
      if(temp % 0x02 == 0x01)
        errors++;
      temp>>1;
    }
  }
  byte errorRate = errors;
  return errorRate;
}
```

clearArrays

```
void clearArrays(){
  int i,j;
  for(i=0; i<12; i++){
    inputRssi[i] = (byte)0;
  }
  for(i=0; i<50; i++){
    inputData[i] = (byte)0;
  }
  for(i=0; i<6; i++){
    for(j=0; j<7; j++){
      linkTable[i][j] = (byte)0;
    }
  }
  for(i=0; i<3; i++){
    for(j=0; j<10; j++){
      nodeTable[i][j] = (byte)0;
    }
  }
}
```

API Mode Router Setup and Loop

```
//Node ID is set by hand
#define NODE_ID      0x31

int XBee_pin2 = 11;
int coordReceive = 0;
byte inputData[] = {

0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0};
byte mode[2];
int validMode = 0;
byte nodeTable[3][10];
byte inputRssi[12];
byte linkTable[6][7];
int currLink=0;
int sysStarted;

void setup(){
  delay(100);
  clearArrays();
  findNetwork();
  delay(100);
  pinMode(XBee_pin2, INPUT);
  delay(100);
  sysStarted = 0;
  powerLevel(4);
  delay(100);
}

void loop() {
  int addrNum = 0;

  receiveData();

  if(validMode == 1 && mode[0] == 0x31){
    sysStarted = 1;
    modeOne();
  }
  if(validMode==1 && mode[0]==0x32 && currLink>0 && sysStarted>0){
    sysStarted = 2;
    modeTwo();
```

```
  }
  if(validMode==1 && mode[0]==0x33 && sysStarted>1){
    modeThree();
  }
}
```

API Mode Coordinator Methods

open_port

```
int open_port()
{
  int fd; // file descriptor
  // open for read/write, maintains control over terminal,
  fd = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY | O_NDELAY);
  if (fd == -1) // if port doesn't open
    {
      perror("open_port: Unable to open /dev/ttyUSB0 - ");
    }
  else{
    printf("successful port opening\n\r");
    fcntl(fd, F_SETFL, FNDELAY); // set file status flag
  }
  srand(201);
  return(fd);
}
```

arrayLength

```
unsigned char arrayLength(unsigned char inArray[]){
  unsigned char length = 0x00;
  if(inArray[0] == 'R'){
    length++;
    while(inArray[length]!='R'){
      length++;
    }
    return (length-1);
  }
  if(inArray[0] == 'S'){
    length++;
    while(inArray[length]!='S'){
      length++;
    }
    return (length-1);
  }
  if(inArray[0] == 0x31 || inArray[0] == 0x39){
    length = 2;
    return length;
  }
  if(inArray[0] == 'G'){
    length = 31;
    return length;
  }
  while(inArray[length] == 'A')
    length += 3;
  if(inArray[0] == 'A'){
    length += 4;
    return length;
  }
```

```
}
```

apiTx

```
int apiTx(int fd, unsigned char txData[], unsigned char destAddress[],
unsigned char hops){
  unsigned char transData[50];
  transData[0] = 0x7E;
  transData[1] = 0x00;
  unsigned char arrayLen = 0;
  arrayLen = arrayLength(txData) + 0x0E;
  transData[2] = arrayLen;
  transData[3] = 0x10;
  transData[4] = 0x01;
  unsigned char checksum = 0;
  int i = 0;
  unsigned char precs = 0x11;
  unsigned char *transMem;
  transMem = &transData[0];
  for(i=0; i<10; i++){
    transData[i+5] = destAddress[i];
    precs += destAddress[i];
  }
  transData[15] = hops;
  precs += hops;
  transData[16] = 0x00;
  if(txData[0]=='R'){
    for(i=0; i<(arrayLen-14); i++){
      transData[i+17] = txData[i+1];
      precs += txData[i+1];
    }
  }
  else{
    for(i=0; i<(arrayLen-14); i++){
      transData[i+17] = txData[i];
      precs += txData[i];
    }
  }
  precs = precs % 0x100;
  checksum = 0xFF - precs;
  transData[arrayLen + 3] = checksum;
  write(fd, transMem, arrayLen+4);
  printf("Data Printed: \n\r");
  for(i=0;i<arrayLen+4;i++){
    printf("%02x", transData[i]);
  }
  printf("\n\r");
}
```

findRssi

```
//Rssi Check
unsigned char findRssi(fd) {
  printf("Find RSSI\n\r");
  unsigned char rssiRequest[] = {0x7E, 0x00, 0x04, 0x08, 0x01, 0x44, 0x42,
0x70};
  unsigned char *dbmem;
  unsigned char myRssi = 0;
  int length;
  int i;
```

```
  while(1){
    dbmem = &rssiRequest[0];
    write(fd, dbmem, 8);
    usleep(100000);
    int g = read(fd, packetmem, 1);
    if(g == 1 && packet[0] == 0x7E){
      printf("SD: %02x\n\r", packet[0]);
      usleep(2000);
      g += read(fd, packetmem+1, 2);
      if(g == 3){
      length = packet[2]+1;
      printf("length: %d\n\r", length);
      usleep(900*length);
      g += read(fd, packetmem+3, length);
      for(i=0; i<(length+3); i++)
        printf("%x", packet[i]);
      printf("\n\r");
      if(packet[3]==0x88 && (packet[5] == 'D') && (packet[6]== 'B') && (g ==
10))
        myRssi = packet[8];
      if(myRssi>0x19 && myRssi<0x5D)
        printf("RSSI: %x\n\r", myRssi);
      if(myRssi == 0)
        return 0xFF;
      return myRssi;
      }
    }
  }
}
```

receiveData

```
int receiveData(int fd){
  int g = read(fd, packetmem, 1);
  int length;
  int i;

  if(g == 1 && packet[0] == 0x7E){
    printf("SD: %02x\n\r", packet[0]);
    usleep(1700);
    g += read(fd, packetmem+1, 2);
    if(g == 3){
      length = packet[2]+1;
      printf("length: %d\n\r", length);
      usleep(834*length);
      g += read(fd, packetmem+3, length);
      if(g > 0){
      for(i=0; i<(length+3); i++)
        printf("%02x", packet[i]);
      printf("\n\r");
      if(packet[3]==0x90 && packet[length]>0x31 && packet[length]<0x34){
        mode[0] = packet[length];
        mode[1] = packet[length+1];
        printf("Mode[0]%x\n\r", mode[0]);
        printf("Mode[1]%x\n\r", mode[1]);
        if(mode[0] == mode[1])
          validMode = 1;
        else{
```

```
          validMode = 0;
          return -1;
        }
        if(validMode == 1 && mode[0] == 0x32){
            int currPlace = currLink*2;
            inputRssi[currPlace] = 'S';
            inputRssi[currPlace+1] = findRssi(fd);
            printf("Get RSSI: ");
            for(i=0; i<12; i++)
            printf("%02x", inputRssi[i]);
            printf("\n\r");
            for(i=0; i<29; i++)
              linkTable[currLink][i] = packet[i+15];
            printf("Store to Table: ");
          for(i=0; i<29; i++)
            printf("%02x", linkTable[currLink][i]);
            printf("\n\r");
            currLink++;
        }
        else if(validMode == 1 && mode[0] == 0x33){
          unsigned char receivedA[10];
          unsigned char receivedM[2] = {0x39, 0x39};
          for(i=0;i<10;i++){
            receivedA[i] = packet[i+4];
          }
          apiTx(fd, receivedM, receivedA, 3);
          usleep(1000000);
          for(i=0; i<10; i++)
            addressTable[packet[length-4]-0x30][i]=receivedA[i];
          for(i=15;i<(length+2);i+=7){
            if(packet[i] == 'N' && packet[i+2] == 'S'){
            int powerL = 0;
            if(packet[i+6]==0x01)
              powerL = 0;
            else
              powerL = 1;
            printf("Store Cognitive Data: Node %x to %x\n\r", (packet[i+1]-
0x30), (packet[length-4]-0x30));
            printf("PL: %x; RSSI: %x, BER: %x\n\r", packet[i+6], packet[i+3],
packet[i+5]);
            rssiTable[powerL][packet[i+1]-0x30][packet[length-4]-0x30] =
packet[i+3];
            berTable[powerL][packet[i+1]-0x30][packet[length-4]-0x30] =
packet[i+5];
            }
            else if(packet[i] == 'N' && packet[i+2] == 'B'){
            printf("Store Node %x Battery: %x\n\r", packet[i+1],
packet[i+3]);
            battTable[packet[i+1]-0x30] = packet[i+3];
            }
          }       // End of Store Data Loop
        }         // End of Mode 3 Loop
        for(i=0;i<70;i++)
          packet[i] = 0x00;
        mode[0] = 0;
        mode[1] = 0;
      }           // End of if Receive Type Loop
      }             // End of if Data Read
    }               // End of if Length Read
```

```
  }                 // End of found Start Delimiter
  return g;
}
```

nodeDiscovery

```
int nodeDiscovery(int fd){
  unsigned char broadHops[] = {0x7E, 0x00, 0x05, 0x08, 0x00, 0x42, 0x48, 
0x01, 0x6D}; //sets broadcast hops to 1
  unsigned char discTimeout[] = {0x7E, 0x00, 0x05, 0x08, 0x00, 0x4E, 0x54, 
0x32, 0x22}; //sets timeout to 3.2s
  unsigned char nodeDisc[] = {0x7E, 0x00, 0x04, 0x08, 0x01, 0x4E, 0x44, 
0x64}; // requests node discovery
  unsigned char *bhmem;
  unsigned char *dtmem;
  unsigned char *ndmem;
  int i = 0;
  int j = 0;
  int count = 0;
  int addrNum;
  int length;
  bhmem = &broadHops[0];
  dtmem = &discTimeout[0];
  ndmem = &nodeDisc[0];
  packetmem = &packet[0];
  for(i=0; i<9; i++){
    write(fd, bhmem+i, 1);
  }
  usleep(200000);
  for(i=0; i<9; i++){
    write(fd, dtmem+i, 1);
  }
  usleep(200000);
  for(i=0; i<8; i++){
    write(fd, ndmem+i, 1);
  }
  i=-1;
  usleep(200000);
  //tcflush(fd, TCIFLUSH);
  while(count<150){
    int g = receiveData(fd);
    //printf("Read Status: %d, %c, %c\n\r", g, packet[5], packet[6]);
    if((packet[5] == 'N') && (packet[6]== 'D') && (g > 17)){
      printf("Node found\n\r");
      if((packet[8] != nodeTable[i][8]) && (packet[9] != nodeTable[i][9])){
      printf("Iteration because %02x != %02x and %02x != %02x, i = %d\n\r", 
packet[8], nodeTable[i][8], packet[9], nodeTable[i][9], (i+1));
      i++;
      }
      for(j=0; j<8; j++)
        nodeTable[i][j] = packet[j+10];
      for(j=0; j<2; j++)
      nodeTable[i][j+8] = packet[j+8];
      printf("Node recorded\n\r");
    }
    count++;
    usleep(55000);
  }
  addrNum = i;
```

```
  for(i=0; i<addrNum+1; i++){
    for(j=0; j<10; j++)
      printf("%02x", nodeTable[i][j]);
    printf("\n\r");
  }
  printf("Wait for Node Disovery: 34 seconds\n\r");
  usleep(34000000);
  return(addrNum+1);
}
```

powerLevel

```
void powerLevel(int fd, unsigned char power){
  usleep(1000000);
  unsigned char powerAdjust[] = {0x7E, 0x00, 0x05, 0x08, 0x00, 0x50, 0x4C};
  unsigned char checkSum = 0xFF - ((0xA4 + power) % 0x100);
  unsigned char *pamem;
  unsigned char *powermem;
  unsigned char *csmem;
  int i;

  pamem = &powerAdjust[0];
  powermem = &power;
  csmem = &checkSum;
  for(i=0; i<7; i++)
    write(fd, pamem+i, 1);
  write(fd, powermem, 1);
  write(fd, csmem, 1);
  usleep(1000000);
}
```

sendDiagnostic

```
sendDiagnostic(int fd, int addrNum){
  unsigned char testPower = 0x01;
  unsigned char berData[31];
  int i;
  int randCount;
  int exCount;
  int j;

  for(i=0;i<25;i++)
    berData[i] = 'G';
  berData[25] = 'N';
  berData[26] = NODE_ID;
  berData[27] = 'P';
  berData[29] = 0x32;
  berData[30] = 0x32;
  while(testPower < 4){
    powerLevel(fd, testPower);
    berData[28] = testPower;
    for(i=0; i<addrNum; i++){
      randCount = rand()%(200/addrNum);
      exCount = 200/addrNum - randCount;
      printf("Random count: %d and %d\n\r", randCount, exCount);
      j = 0;
      while(randCount > j){
        receiveData(fd);
        usleep(100000);
        j++;
```

```
      }
      apiTx(fd, berData, nodeTable[i], 1);
      j = 0;
      while(randCount > j){
        receiveData(fd);
      usleep(100000);
        j++;
      }
    }
    testPower += 2;
  }
}
```

berCheck

```
unsigned char berCheck(int link){
  unsigned char temp = 0x00;
  int errors = 0;
  int j;
  int i;
  int errorRate;
  for(j=0; j<25; j++){
    temp = linkTable[link][j] ^ 0x47;
    for(i=0; i<8; i++){
      if(temp%0x02 == 0x01)
      errors++;
      temp>>1;
    }
  }
  errorRate = errors;
  return errorRate;
}
```

storeData

```
void storeData(){
  int i;
  unsigned char tempBer = 0x00;
  int powerL = 0;
  for(i=0;i<6;i++){
    if(linkTable[i][0] == 'G'){
      tempBer = berCheck(i);
      if(linkTable[i][28] == 0x01)
      powerL = 0;
      else
      powerL = 1;
      printf("Store Cognitive from Link Table\n\r");
      printf("From node %x: PL=%x, RSSI=%x, BER=%x\n\r", (linkTable[i][26]-
0x30), linkTable[i][28], inputRssi[(2*i)+1], tempBer);
      rssiTable[powerL][linkTable[i][26]-0x30][0] = inputRssi[(2*i)+1];
      berTable[powerL][linkTable[i][26]-0x30][0] = tempBer;
    }
  }
}
```

clearData

```
void clearData(){
  int i, j, k;
  for(i=0;i<12;i++)
```

```
    inputRssi[i] = 0x00;
  for(i=0;i<4;i++)
    battTable[i] = 0x00;
  for(i=0;i<70;i++)
    packet[i] = 0x00;
  for(i=0;i<6;i++){
    for(j=0;j<7;j++){
      linkTable[i][j] = 0x00;
    }
  }
  for(i=0;i<3;i++){
    for(j=0;j<10;j++){
      nodeTable[i][j] = 0x00;
    }
  }
  for(i=0;i<2;i++){
    for(j=0;j<4;j++){
      for(k=0;k<4;k++){
      rssiTable[i][j][k] = 0xFF;
      }
    }
  }
  for(i=0;i<2;i++){
    for(j=0;j<4;j++){
      for(k=0;k<4;k++){
      berTable[i][j][k] = 0xFF;
      }
    }
  }
  currLink = 0;
  validMode = 0;
}
```

routeMe

```
void routeMe(int source, int dest, int numNodes){
  int visited[numNodes];
  int track[numNodes];
  int cost[numNodes];
  int i, s, r, q;
  int numLinks = numNodes*numNodes;

  int whosNext(){
    int nextNode;
    int min=999999;
    int k;
    for(k=0;k<numNodes;k++){
      if(visited[k]==0){
      if(cost[k]<min){
        min=cost[k];
        nextNode=k;
      }
      }
    }
    return nextNode;
  }
  int j;
  for(i=0;i<numNodes;i++){
    cost[i] = 9999;
```

```
    track[i] = 0;
    visited[i] = 0;
  }
  cost[source]=0;

  //All node costs set very high
  //Source cost set to 0 ensuring it goes first
  while(visited[dest]==0){
    s = whosNext();
    visited[s]=1;//when dest is selected all paths were evaluated
    for(i=0;i<numLinks;i++){
      if(links[i][0] == s && visited[(int)(links[i][1])]==0){
        int r=links[i][1];
        int q=links[i][2];
      int maxCost=q;

        if(cost[s]>q)
        maxCost = cost[s];
      if(cost[r]>maxCost){
        cost[r]=maxCost;
        track[r]=s;
      }
      }
      int pathNode=dest;
      int j;
      for(j=1;j<numNodes;j++){
      pathTable[source][j]=track[pathNode];
      pathNode=track[pathNode];
      if(pathNode==source)
        break;
      }
    }
  }
}
```

getQ

```
void getQ(){
  battTable[2]=20;
  battTable[3]=90;
  int i, j, k=0;
  int rssiQ[4][4];
  int berQ[4][4];
  float ecQ[4][4];
  for(i=0;i<4;i++){
    for(j=0;j<4;j++){
      pL[i][j]=0;
    }
  }
  for(i=0;i<16;i++){
    for(j=0;j<3;j++)
      links[i][j] = 9999;
  }

  for(i=0;i<4;i++){
    for(j=0;j<4;j++){
      if(rssiTable[1][i][j]>decBound){
        rssiQ[i][j]=rssiTable[1][i][j];
        berQ[i][j]=berTable[1][i][j];
```

```
  if(rssiTable[1][i][j]<incBound){
    if(pL[i][j]<3){
    pL[i][j]=3;
    if(i==0)
      pL[i][j]=4;
    ecQ[i][j]= (101-battTable[i])*(pL[i][j]-powerB4[i]+4);
  }
  if(i==0)
    pL[i][j]=4;
  if(i==j)
    ecQ[i][j]=9999;
  }
  else{
    pL[i][j]=4;
    ecQ[i][j]= (101-battTable[i])*(pL[i][j]-powerB4[i]+4);
  if(i==j)
    ecQ[i][j]=9999;
  }
}
else{
  rssiQ[i][j]=rssiTable[0][i][j];
  berQ[i][j]=berTable[0][i][j];
if(berQ[i][j]>1){
  if(berTable[1][i][j]<2){
    berQ[i][j] = berTable[1][i][j];
    pL[i][j]=3;
    ecQ[i][j]= (101-battTable[i])*(pL[i][j]-powerB4[i]+4);
  }
  else{
    berQ[i][j] = 9999;
  }
}
}
if(rssiTable[0][i][j]<decBound){
//pL[i][j]=0;
if(i==0)
  pL[i][j]=4;
  ecQ[i][j]= (101-battTable[i])*(pL[i][j]-powerB4[i]+4);
if(i==j)
  ecQ[i][j]=9999;
}
else if(rssiTable[0][i][j]<incBound){
if(pL[i][j]<1){
  pL[i][j]=1;
  if(i==0)
    pL[i][j]=4;
  ecQ[i][j]= (101-battTable[i])*(pL[i][j]-powerB4[i]+4);
}
if(i==0)
  pL[i][j]=4;
if(i==j)
  ecQ[i][j]=9999;
}
else{
if(pL[i][j]<2){
  pL[i][j]=2;
  if(i==0)
    pL[i][j]=4;
  ecQ[i][j]= (101-battTable[i])*(pL[i][j]-powerB4[i]+4);
```

```
      }
      if(i==0)
        pL[i][j]=4;
      if(i==j)
        ecQ[i][j]=9999;
      }
      // Test Values Set
      berQ[0][1] = 9999;
      berQ[1][0] = 9999;
      rssiQ[1][3] = 90;
      rssiQ[1][2] = 90;
      // Store Links to Table
      links[k][0] = i;
      links[k][1] = j;
      links[k][2] = wtR*rssiQ[i][j]+wtB*berQ[i][j]+wtP*ecQ[i][j];
      k++;
    }
  }
}
```

setPower

```
void setPower(){
  int i, j = 0;
  for(i=0;i<4;i++)
    powerB4[i]=0;
  for(i=0;i<4;i++){
    while(pathTable[i][j+1]!=0 && j<3){
      int r=pathTable[i][j];
      int s=pathTable[i][j+1];
      if(powerB4[s]<pL[s][r]){
      powerB4[s]=pL[s][r];
      }
      j++;
    }
  }
  powerB4[0]=4;
}
```

createRoute

```
int createRoute(int fd, unsigned char sourceRoute[], unsigned char
destAddress[]){
  printf("Set Coordinator Source Route\n\r");
  unsigned char transData[50];
  transData[0] = 0x7E;
  transData[1] = 0x00;
  unsigned char arrayLen = 0;
  arrayLen = arrayLength(sourceRoute) + 0x0D;
  transData[2] = arrayLen;
  transData[3] = 0x21;
  transData[4] = 0x00;
  unsigned char checksum = 0;
  int i = 0;
  unsigned char precs = 0x21;
  unsigned char *transMem;
  transMem = &transData[0];
  for(i=0; i<10; i++){
    transData[i+5] = destAddress[i];
    precs += destAddress[i];
```

```
  }
  transData[15] = 0x00;
  for(i=0; i<(arrayLen-13); i++){
    transData[i+16] = sourceRoute[i+1];
    precs += sourceRoute[i+1];
  }
  precs = precs % 0x100;
  checksum = 0xFF - precs;
  transData[arrayLen + 3] = checksum;
  write(fd, transMem, arrayLen+4);
  printf("Data Printed: \n\r");
  for(i=0;i<arrayLen+4;i++){
    printf("%02x", transData[i]);
  }
  printf("\n\r");
}
```

sendRoutes

```
int sendRoutes(int fd){
  int n;
  int i;
  int d;
  int j = 0;
  unsigned char sendRoute[numNodes][2*numNodes+1];

  for(n=1; n<numNodes; n++){
    for(i=0; i<(2*numNodes+1); i++)
      sendRoute[n][i]='R';
  }

  for(n=1; n<numNodes; n++){
    printf("Node %d address:\n\r", n);
    for(i=0; i<10; i++)
      printf("%c ", addressTable[n][i]);
    printf("\n\r");
  }
  int pathDone=0;
  for(n=1;n<numNodes;n++){
    printf("Intermediate hops of %d are:\n\r", n);
    for(d=1; d<numNodes; d++){
      if(d == numNodes-1){
      pathDone=1;
      }
      else if(pathTable[n][d+1] == 0){
      pathDone=1;
      }
      else if(pathTable[n][d]!=0 && pathDone==0){
      printf("%d<-", pathTable[n][d]);
      for(i=1; i<3; i++){
        sendRoute[n][i+j]=addressTable[pathTable[n][d]][i+7];
      }
      j += 2;
      }
    }
    printf("\n\r");
    sendRoute[n][j+1]=powerB4[n];
    sendRoute[n][j+2]=0x33;
    sendRoute[n][j+3]=0x33;
```

```
    j=0;
    pathDone=0;
  }
  for(n=1;n<numNodes;n++){
    printf("Send route for Node %d\n\r", n);
    int m;
    for(m=0;m<9;m++)
      printf("%d<-", sendRoute[n][m]);
    printf("\n\r");
    apiTx(fd, sendRoute[n], addressTable[n], 0);
  }
  return 0;
}
```

createLinks

```
int createLinks(){
  int p;
  int c;
  int r;
  int i=0;
  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++){
      routingTable[r][c]=0;
    }
  }
  for(p=0; p<2; p++){
    for(r=0; r<nodesIn; r++){
      for(c=0; c<nodesIn; c++){
      if(berTable[p][r][c]<2 || berTable[p][c][r]<2){
        routingTable[r][c]=1;
        routingTable[c][r]=1;
      }
      }
    }
  }
  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++){
      //printf("%d ", routingTable[r][c]);
    }
    //printf("\n\r");
  }
  return 0;
}
```

coordNeighbors

```
int coordNeighbors(){
  int c;
  int i;
  int r;

  for(c=0; c<nodesIn; c++){
    for(i=0; i<nodesIn-1; i++){
      nodeNeighbors[c][i]=0;
    }
  }

  i=0;
```

```
  for(r=1; r<nodesIn; r++){
    if(routingTable[r][0]==1){
      nodeNeighbors[0][i]=r;
      i++;
    }
  }

  i=0;

  for(i=0; i<nodesIn-1; i++){
    //printf("%d ", nodeNeighbors[0][i]);
  }
  //printf("\n\r");

  return 0;
}
```

findPaths

```
int findPaths(){
  int depth = 0;
  int path = 0;
  unsigned char checkingNeighbor[] = {0,0,0,0,0,0,0,0};
  int c = 0;
  int i;
  int n;
  int r;
  unsigned char redundancy;
  int d;
  int neighbor;

  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++){
      pathsOut[r][c]=0;
    }
  }
  c=0;

  while(1){
    //printf("next neighbor %d\n\r", nodeNeighbors[c][checkingNeighbor[c]]);
    if(nodeNeighbors[c][checkingNeighbor[c]]>0){
      depth++;
      pathsOut[path][depth]=nodeNeighbors[c][checkingNeighbor[c]];
      checkingNeighbor[c]++;
      c = nodeNeighbors[c][checkingNeighbor[c]-1];
      //printf("path %d depth %d node %d\n\r", path, depth, c);
    }
    else{
      while(depth>-1){
      depth--;
      if(depth<0)
        break;
      //printf("depth %d\n\r", depth);
      c=pathsOut[path][depth];
      //printf("next neighbor of previous node %d is %d\n\r",c,
nodeNeighbors[c][checkingNeighbor[c]]);
      if(nodeNeighbors[c][checkingNeighbor[c]]>0){
        path++;
        //printf("path %d:\n\r", path);
```

```
        for(d=0; d<depth+1; d++){
          pathsOut[path][d]=pathsOut[path-1][d];
          //printf("%d", pathsOut[path][d]);
        }
        depth++;
        pathsOut[path][depth]=nodeNeighbors[c][checkingNeighbor[c]];
        //printf("%d", pathsOut[path][depth]);
          //printf("\n\r");
        checkingNeighbor[c]++;
        c=nodeNeighbors[c][checkingNeighbor[c]-1];
        //printf("path %d depth %d node %d\n\r", path, depth, c);
        break;
      }
      else
        checkingNeighbor[c]=0;
      }
    }
    if(depth<0)
      break;
    i=0;
    //printf("depth %d\n\r", depth);
    for(n=0; n<nodesIn-1; n++)
      nodeNeighbors[c][n] = 0;
    for(r=1; r<nodesIn; r++){
      redundancy = 0;
      for(d=0; d<depth; d++){
      for(neighbor=0; neighbor<nodesIn-1; neighbor++){
        if(nodeNeighbors[pathsOut[path][d]][neighbor]==r){
          redundancy = 1;
          //printf("router %d redundancy %d\n\r", r, redundancy);
        }
      }
      }
      if(redundancy==0)
      //printf("%d and %d linked? %d\n\r", r, c, routingTable[r][c]);
      if(routingTable[r][c]>0 && redundancy==0){
      nodeNeighbors[c][i]=r;
      //printf("neighbor found %d=%d\n\r", r, nodeNeighbors[c][i]);
      i++;
      }
    }
  }
  return path;
}
```

convertPaths

```
int convertPaths(int paths){
  int p;
  int pp;
  int d;
  int source;
  int dd;
  int in;
  int s;
  int r;
  int c;
  int deletes;
  int pathErased;
```

```
  int pathBreak;
  int sourceUpdate;
  int ddd;
  int ppp;

  for(s=0; s<nodesIn; s++){
    for(r=0; r<nodesIn; r++){
      for(c=0; c<nodesIn; c++)
        pathsIn[s][r][c]='n';
      //printf("%d", pathsOut[s][r]);
    }
    //printf("\n\r");
    inPath[s]=0;
  }


  for(p=0; p<paths+1; p++){
    for(d=1; d<nodesIn; d++){
      if(pathsOut[p][d]==0)
      break;
      source=pathsOut[p][d];
      if(source!=0){
      //printf("%d", source);
      in=inPath[source];
        for(dd=0; dd<d+1; dd++){
        pathsIn[source][in][dd]=pathsOut[p][d-dd];
        //printf("%d", pathsIn[source][in][dd]);
        }
        //printf("\n\r");
        inPath[source]++;
      //printf("%d has %d paths\n\r", source, inPath[source]);
      }
    }
  }
  for(s=1; s<nodesIn; s++){
    //printf("%d has %d paths\n\r", s, inPath[s]);
    in=inPath[s];
    deletes=0;
    for(p=0; p<in-1; p++){
      //printf("check path %d\n\r",p);
      for(pp=p+1; pp<in; pp++){
        //printf("against path %d\n\r", pp);
        pathErased=0;
        for(d=0; d<nodesIn; d++){
        //printf("at depth %d until %d\n\r", d, nodesIn-1);
        if(pathsIn[s][p][d]!=pathsIn[s][pp-deletes][d]){
          //printf("difference break\n\r");
          break;
        }
        if(pathErased>0){
            //printf("pathErased break?\n\r");
          break;
        }
          //printf("is this n? %d\n\r", pathsIn[s][p][d]);
        if(pathsIn[s][p][d]=='n'){
            //printf("found repeat in %d of path %d at path %d\n\r", s, p,
(pp-deletes));
          for(ppp=pp+1; ppp<in; ppp++){
            //printf("shift %d back to %d\n\r", ppp, (ppp-1));
```

```
            for(dd=0; dd<nodesIn; dd++){
              if(pathsIn[s][ppp][dd]!='n'){
              pathsIn[s][ppp-1][dd]=pathsIn[s][ppp][dd];
              //printf("%d",pathsIn[s][ppp-1][dd]);
            }
              else{
              //printf("\n\rshifted %d back to %d\n\r", ppp, (ppp-1));
              break;
            }
            }
          }
          pathErased = pp;
          deletes++;
          inPath[s]--;
          for(ddd=0; ddd<nodesIn; ddd++)
            pathsIn[s][in-deletes][ddd]='n';
          //printf("erased %d and filled %d with n's. %d deletes\n\r", pp,
(in-deletes), deletes);
        }
        }
        //printf("yes\n\r");
      }
    }
  }
  //for(s=0; s<nodesIn; s++){
  //for(r=0; r<nodesIn; r++){
  //  for(c=0; c<nodesIn; c++)
        //printf("%02x", pathsIn[s][r][c]);
      //printf("\n\r");
  //}
    //printf("\n\r");
  //}
return 0;
}

int findlinkRssi(){
  int r;
  int c;

  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++){
      linkRssi[r][c]=0;
      linkPower[r][c]=0;
    }
  }

  for(c=0; c<nodesIn; c++){
    for(r=0; r<nodesIn; r++){
      if(routingTable[r][c]>0 && ((rssiTable[0][r][c]>0 &&
rssiTable[0][c][r]>0) || (rssiTable[1][r][c]>0 && rssiTable[1][c][r]>0))){
      if((rssiTable[0][r][c]+rssiTable[0][r][c])/2 < 60){
        linkPower[r][c]=1;
        linkPower[c][r]=1;
        linkRssi[r][c]=(rssiTable[0][r][c]+rssiTable[0][c][r])/2;
        linkRssi[c][r]=(rssiTable[0][r][c]+rssiTable[0][c][r])/2;
      }
      if((rssiTable[0][r][c]+rssiTable[0][r][c])/2 > 60 &&
(rssiTable[0][r][c]+rssiTable[0][r][c])/2 < 80){
        linkPower[r][c]=2;
```

```
        linkPower[c][r]=2;
        linkRssi[r][c]=(rssiTable[0][r][c]+rssiTable[0][c][r])/2;
        linkRssi[c][r]=(rssiTable[0][r][c]+rssiTable[0][c][r])/2;
      }
      if((rssiTable[0][r][c]+rssiTable[0][r][c])/2 > 80 &&
(rssiTable[1][r][c]+rssiTable[1][r][c])/2 < 60){
        linkPower[r][c]=3;
        linkPower[c][r]=3;

linkRssi[r][c]=(rssiTable[0][r][c]+rssiTable[0][c][r]+rssiTable[1][r][c]+rssi
Table[1][c][r])/4;

linkRssi[c][r]=(rssiTable[0][r][c]+rssiTable[0][c][r]+rssiTable[1][r][c]+rssi
Table[1][c][r])/4;
      }
      else{
        if((rssiTable[1][r][c]+rssiTable[1][r][c])/2 > 60 &&
(rssiTable[1][r][c]+rssiTable[1][r][c])/2 < 80){
          linkPower[r][c]=4;
          linkPower[c][r]=4;
          linkRssi[r][c]=(rssiTable[1][r][c]+rssiTable[1][c][r])/2;
          linkRssi[c][r]=(rssiTable[1][r][c]+rssiTable[1][c][r])/2;
        }
        if((rssiTable[1][r][c]+rssiTable[1][r][c])/2 > 80){
          linkPower[r][c]=5;
          linkPower[c][r]=5;
          linkRssi[r][c]=(rssiTable[1][r][c]+rssiTable[1][c][r])/2;
          linkRssi[c][r]=(rssiTable[1][r][c]+rssiTable[1][c][r])/2;
        }
      }
      }
      if((rssiTable[0][r][c]==0 || rssiTable[0][c][r]==0 ||
rssiTable[1][r][c]==0 || rssiTable[1][c][r]==0) && routingTable[r][c]>0){
      if(rssiTable[0][r][c] < 60 || rssiTable[0][c][r] < 60){
        linkPower[r][c]=1;
        linkPower[c][r]=1;
        if(rssiTable[0][r][c]>0){
          linkRssi[r][c]=rssiTable[0][r][c];
          linkRssi[c][r]=rssiTable[0][r][c];
        }
        if(rssiTable[0][c][r]>0){
          linkRssi[r][c]=rssiTable[0][c][r];
          linkRssi[c][r]=rssiTable[0][c][r];
        }
      }
      if((rssiTable[0][r][c] > 60 || rssiTable[0][c][r] > 60) &&
(rssiTable[0][r][c] < 80 || rssiTable[0][c][r] < 80)){
        linkPower[r][c]=2;
        linkPower[c][r]=2;
        if(rssiTable[0][r][c]>0){
          linkRssi[r][c]=rssiTable[0][r][c];
          linkRssi[c][r]=rssiTable[0][r][c];
        }
        if(rssiTable[0][c][r]>0){
          linkRssi[r][c]=rssiTable[0][c][r];
          linkRssi[c][r]=rssiTable[0][c][r];
        }
      }
```

```
      if((rssiTable[0][r][c] > 80 && rssiTable[1][r][c] < 60) ||
(rssiTable[0][c][r] > 80 && rssiTable[1][c][r] < 60)){
        linkPower[r][c]=3;
        linkPower[c][r]=3;
        if(rssiTable[0][c][r]>0 && rssiTable[1][c][r]>0){
          linkRssi[r][c]=(rssiTable[0][c][r]+rssiTable[1][c][r])/2;
          linkRssi[c][r]=(rssiTable[0][c][r]+rssiTable[1][c][r])/2;
        }
          if(rssiTable[0][r][c]>0 && rssiTable[1][r][c]>0){
          linkRssi[c][r]=(rssiTable[0][r][c]+rssiTable[1][r][c])/2;
          linkRssi[r][c]=(rssiTable[0][r][c]+rssiTable[1][r][c])/2;
        }
      }
      else{
        if((rssiTable[0][r][c] > 60 || rssiTable[0][c][r] > 60) &&
(rssiTable[0][r][c] < 80 || rssiTable[0][c][r] < 80)){
          linkPower[r][c]=4;
          linkPower[c][r]=4;
          if(rssiTable[1][r][c]>0){
            linkRssi[r][c]=rssiTable[1][r][c];
            linkRssi[c][r]=rssiTable[1][r][c];
          }
          if(rssiTable[1][c][r]>0){
            linkRssi[r][c]=rssiTable[1][c][r];
            linkRssi[c][r]=rssiTable[1][c][r];
          }
        }
        if(rssiTable[1][r][c] > 80 || rssiTable[1][c][r] > 80){
          linkPower[r][c]=5;
          linkPower[c][r]=5;
          if(rssiTable[1][r][c]>0){
            linkRssi[r][c]=rssiTable[1][r][c];
            linkRssi[c][r]=rssiTable[1][r][c];
          }
          if(rssiTable[1][c][r]>0){
            linkRssi[r][c]=rssiTable[1][c][r];
            linkRssi[c][r]=rssiTable[1][c][r];
          }
        }
      }
      }
    }
  }
  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++){
      //printf("%d ", linkRssi[r][c]);
    }
    //printf("\n\r");
  }
  //printf("\n\r");
  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++){
      // printf("%d ", linkPower[r][c]);
    }
    //printf("\n\r");
  }
  return 0;
}
```

initialRoutes

```
int initialRoutes(){
  int s;
  int in;
  int tempRssi;
  int totalRssi;
  int r;
  int c;
  int p;
  int d;

  for(r=0; r<nodesIn; r++){
    sourceRoute[r]=0;
    for(c=0; c<nodesIn; c++){
      inRoute[r][c]=0;
    }
  }

  for(s=1; s<nodesIn; s++){
    in=inPath[s];
    //printf("%d has %d paths\n\r", s, in);
    if(in==1){
      sourceRoute[s]=0;
      inRoute[0][s]=1;
    }
    else{
      tempRssi=0;
      for(p=0; p<in; p++){
      totalRssi=0;
      for(d=0; d<nodesIn-1; d++){
        if(pathsIn[s][p][d+1]!='n'){
          totalRssi += linkRssi[pathsIn[s][p][d]][pathsIn[s][p][d+1]];
        }
        else
          break;
      }
      if((totalRssi< tempRssi) || tempRssi==0){
        tempRssi=totalRssi;
        sourceRoute[s] = p;
        for(r=0; r<nodesIn; r++)
          inRoute[r][s]=0;
        for(d=1; d<nodesIn; d++){
          if(pathsIn[s][p][d]!='n')
            inRoute[pathsIn[s][p][d]][s]=1;
          else
            break;
        }
      }
      }
    }
  }
  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++){
      //printf("%d", inRoute[r][c]);
    }
    //printf("\n\r");
  }
  //printf("\n\r");
```

```
  printf("Initial routes based on RSSI:\n\r");
  for(r=1; r<nodesIn; r++){
    printf("for router %d: ", r);
    for(d=0; d<nodesIn; d++){
      if(pathsIn[r][sourceRoute[r]][d]!='n')
      printf("%d ", pathsIn[r][sourceRoute[r]][d]);
      else
      break;
    }
    printf("\n\r");
  }
  printf("\n\r");
  return 0;
}
```

battMaximize

```
int battMaximize(){
  unsigned char sumRoutes[nodesIn];
  unsigned char tempsumRoutes[nodesIn];
  int r;
  int c;
  int cc;
  int n;
  int p;
  int d;
  int i;
  int nn;
  unsigned char nodeOrder[nodesIn-1];
  float minTime;
  int minNode;
  float timeLeft[nodesIn];
  float temptimeLeft[nodesIn];
  unsigned char tempnodeOrder[nodesIn];
  unsigned char tempinRoute[nodesIn][nodesIn];
  int changeMade=1;
  int orderNum=0;
  float battLife[nodesIn];
  int newminLess;
  int in;
  int orderChange;

  for(r=0; r<nodesIn; r++){
    tempsumRoutes[r]=0;
    sumRoutes[r]=0;
    timeLeft[r]=0;
    temptimeLeft[r]=0;
    for(c=0; c<nodesIn; c++)
      tempinRoute[r][c]=inRoute[r][c];
  }

  for(r=0; r<nodesIn-1; r++){
    nodeOrder[r]=0;
    tempnodeOrder[r]=0;
  }

  for(r=0; r<nodesIn; r++){
    for(c=0; c<nodesIn; c++)
      sumRoutes[r] += inRoute[r][c];
```

```
  }
  printf("Initial battery life of each node:\n\r");
  for(n=1; n<nodesIn; n++){
    timeLeft[n]=(20*((unsigned int)battTable[n]))/(9.9*(((unsigned
int)sumRoutes[n])+1)+73.8);
    printf("%f ", timeLeft[n]);
  }
  printf("\n\r");
  for(n=0; n<nodesIn; n++){
    temptimeLeft[n]=timeLeft[n];
  }
  for(i=0; i<nodesIn-1; i++){
    minTime=0;
    minNode=0;
    for(n=1; n<nodesIn; n++){
      if((minTime > temptimeLeft[n] && temptimeLeft[n]>0) || (minTime==0 &&
temptimeLeft[n]>0)){
      minTime=temptimeLeft[n];
      minNode=n;
      }
    }
    nodeOrder[i]=minNode;
    temptimeLeft[minNode]=0;
  }

  while(1){
    while(1){
      if(changeMade==0)
        break;
      changeMade=0;
      for(c=1; c<nodesIn; c++){
      if(inRoute[nodeOrder[orderNum]][c]==1){
        in=inPath[c];
        for(p=0; p<in; p++){
          if(p!=sourceRoute[c]){
            for(r=0; r<nodesIn; r++){
            tempinRoute[r][c]=0;
            }
            for(d=1; d<nodesIn; d++){
            if(pathsIn[c][p][d]!='n'){
              tempinRoute[pathsIn[c][p][d]][c]=1;
            }
            else
              break;
            }
            for(r=0; r<nodesIn; r++){
                tempsumRoutes[r] = 0;
            for(cc=0; cc<nodesIn; cc++){
              tempsumRoutes[r] += tempinRoute[r][cc];
            }
            }
              for(n=1; n<nodesIn; n++){

      temptimeLeft[n]=(20*((int)battTable[n]))/(9.9*(((int)tempsumRoutes[n])+
1)+73.8);
            }
              for(n=0; n<nodesIn; n++)
            battLife[n]=temptimeLeft[n];
              for(i=0; i<nodesIn-1; i++){
```

```
            minTime=0;
            minNode=0;
            for(n=1; n<nodesIn; n++){
              if((minTime > battLife[n] && battLife[n]>0) || (minTime==0 &&
battLife[n]>0)){
                minTime=battLife[n];
                minNode=n;
              }
            }
            tempnodeOrder[i]=minNode;
            battLife[minNode]=0;
            }
            newminLess=0;
            for(n=0; n<nodesIn-1; n++){
            if((temptimeLeft[nodeOrder[n]] > timeLeft[nodeOrder[n]]) &&
(nodeOrder[n] != tempnodeOrder[n])){
              for(nn=n; nn<nodesIn-1; n++){
                if(nodeOrder[n]==tempnodeOrder[nn])
                  break;
                //printf("Test12 %d\n\r", tempnodeOrder[nn]);
                if(temptimeLeft[tempnodeOrder[nn]]<timeLeft[nodeOrder[n]])
                  newminLess=1;
              }
            }
                if((temptimeLeft[nodeOrder[n]] > timeLeft[nodeOrder[n]]) &&
newminLess!=1){
                  for(nn=0; nn<nodesIn; nn++)
                timeLeft[nn]=temptimeLeft[nn];
              for(r=0; r<nodesIn; r++){
                for(cc=0; cc<nodesIn; cc++)
                  inRoute[r][cc]=tempinRoute[r][cc];
              }
              sourceRoute[c]=p;
              changeMade=1;
              break;
            }
            if(temptimeLeft[nodeOrder[n]]<timeLeft[nodeOrder[n]] ||
newminLess==1)
              break;
            }
          }
        }
      }
      }
    }
    for(n=0; n<nodesIn; n++)
      temptimeLeft[n]=timeLeft[n];
    for(i=0; i<nodesIn-1; i++){
      minTime=0;
      minNode=0;
      for(n=1; n<nodesIn; n++){
      if((minTime > temptimeLeft[n] && temptimeLeft[n]>0) || (minTime==0 &&
temptimeLeft[n]>0)){
        minTime=temptimeLeft[n];
        minNode=n;
      }
      }
      tempnodeOrder[i]=minNode;
      temptimeLeft[minNode]=0;
```

```
    }
    orderChange=0;
    for(n=0; n<nodesIn-1; n++){
      if(nodeOrder[n]!=tempnodeOrder[n]){
        for(nn=0; nn<nodesIn-1; nn++){
        nodeOrder[nn]=tempnodeOrder[nn];
      }
      orderChange=1;
      }
    }
    if(orderChange==0)
      orderNum++;
    changeMade=1;
    if(orderNum==7)
      break;
  }
  printf("Final adapted routes based on RSSI:\n\r");
  for(r=1; r<nodesIn; r++){
    printf("for router %d: ", r);
    for(d=0; d<nodesIn; d++){
      if(pathsIn[r][sourceRoute[r]][d]!='n')
      printf("%d ", pathsIn[r][sourceRoute[r]][d]);
      else
      break;
    }
    printf("\n\r");
  }
  printf("\n\r");
  printf("Final battery life with adapted routes:\n\r");
  for(r=1; r<nodesIn; r++)
    printf("%f ", timeLeft[r]);
  printf("\n\r");

  for(r=0; r<nodesIn-1; r++)
    return 0;
}
```

setPower

```
int setPower(){
  int n;
  int d;

  for(n=0; n<nodesIn; n++)
     minPower[n]=0;

  for(n=0; n<nodesIn; n++){
    for(d=0; d<nodesIn; d++){
      if(pathsIn[n][sourceRoute[n]][d+1]!='n'){
      if(minPower[n]<linkPower[pathsIn[n][sourceRoute[n]][d]][pathsIn[n][sour
ceRoute[n]][d+1]])

minPower[n]=linkPower[pathsIn[n][sourceRoute[n]][d]][pathsIn[n][sourceRoute[n
]][d+1]];
       else
         break;
      }
    }
  }
```

```
  printf("Final power level:\n\r");
  for(n=0; n<nodesIn; n++)
    printf("for %d: %d", n, minPower[n]);
  printf("\n\r");
  return 0;
}
```

routingAlgoritm

```
int routingAlgorithm(){
  int outPaths;
  createLinks();
  coordNeighbors();
  pathsOut[0][0]=0;
  outPaths = findPaths();
  convertPaths(outPaths);
  findlinkRssi();
  initialRoutes();
  battMaximize();
  setPower();
  return 0;
}
```

API Mode Coordinator Main

Transmission Power Limiting Routing Algorithm

```
int main(){
  clearData();
  battTable[0] = 100;
  int i, j;
  for(i=0;i<4;i++){
    for(j=0;j<4;j++)
      pathTable[i][j] = 0;
  }
  packetmem = &packet[0];
  int fd = open_port();
  tcflush(fd, TCIFLUSH);
  struct termios options; // sets up options structure

  tcgetattr(fd, &options); // stores current configuration in structure
  cfsetispeed(&options, B9600); // sets baud in
  cfsetospeed(&options, B9600); // sets baud out
  // enables the receiver and set local mode
  options.c_cflag |= (CLOCAL | CREAD);
  options.c_cflag &= ~CSIZE; // mask the character size bits
  options.c_cflag &= ~PARENB;
  options.c_cflag &= ~CSTOPB;
  options.c_cflag |= CS8; //set no parity 8N1
  options.c_cflag &= ~CRTSCTS; // disable HW flow control
  options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); // raw input enable
  // options.c_lflag != ECHO;
  options.c_iflag &= ~(IXON | IXOFF | IXANY); // disables SW flow control
  options.c_oflag &= ~OPOST; // enables raw output
  tcsetattr(fd, TCSANOW, &options); // sets the new options

  unsigned char reqNodedisc[] = {0x31,0x31};
  unsigned char broadCast[] =
{0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0xFF,0xFE};
```

```
  int addrNum;
  unsigned char rssiSetup[] = {0x7E, 0x00, 0x05, 0x08, 0x01, 0x52, 0x50,
0xFF, 0x55};
  unsigned char *rssiMem;
  rssiMem = &rssiSetup[0];
  //for(i=0;i<9;i++)
  //write(fd, rssiMem,9);
  for(i=0;i<4;i++)
    powerB4[i] = 4;
  while(1){
    write(fd, rssiMem,9);
    printf("\n\rStart Diagnostic\n\r");
    tcflush(fd, TCIOFLUSH);
    apiTx(fd, reqNodedisc, broadCast, 3);
    printf("Mode 1\n\r");
    addrNum = nodeDiscovery(fd); // Make node discovery request
    printf("%d\n\r", addrNum);
    if(addrNum > 0)
      sendDiagnostic(fd, addrNum); //Send Diagnostic Packets
    i = 0;
    while(100 > i){
      receiveData(fd);
      usleep(100000);
      i++;
    }
    storeData();
    printf("Do Routing Algorithm\n\r");
    getQ();
    printf("Q Values: \n\r");
    int j;
    for(i=0;i<16;i++){
      printf("Link %d is %f to %f: %f\n\r",i,links[i][0],links[i][1],
links[i][2]);
    }
    for(i=1;i<4;i++){
      routeMe(i,0,numNodes);
      routeMe(0,i,numNodes);
      unsigned char setRoute[7];
      setRoute[0] = 'S';
      j=0;
      int k=0;
      while(k<2 && pathTable[0][k+1]!=0){
      setRoute[j+2]=addressTable[pathTable[0][k+1]][8];
      setRoute[j+3]=addressTable[pathTable[0][k+1]][9];
            j+=2;
      k++;
      }
      setRoute[1]=k;
      setRoute[j+2] = 'S';
      unsigned char destinationA[10];
      for(j=0;j<10;j++){
      destinationA[j] = addressTable[i][j];
      }
      createRoute(fd, setRoute, destinationA);
    }
    for(i=0;i<4;i++){
      printf("Path table for %d: ", i);
      for(j=0;j<4;j++){
      if(pathTable[i][j]!=0 || j==0)
```

```
        printf("%d<-", pathTable[i][j]);
      }
      printf("\n\r");
    }
    setPower();
    for(i=0;i<4;i++)
      printf("Node %d Power Level: %d\n\r", i, powerB4[i]);
    sendRoutes(fd);
    usleep(6000000);
    clearData();
  }
  return 0;
}
```

Path Limiting Routing Algorithm

```
int main(){
  battTable[0] = 100;
  packetmem = &packet[0];
  int fd = open_port();
  tcflush(fd, TCIFLUSH);
  struct termios options; // sets up options structure

  tcgetattr(fd, &options); // stores current configuration in structure
  cfsetispeed(&options, B9600); // sets baud in
  cfsetospeed(&options, B9600); // sets baud out
  // enables the receiver and set local mode
  options.c_cflag |= (CLOCAL | CREAD);
  options.c_cflag &= ~CSIZE; // mask the character size bits
  options.c_cflag &= ~PARENB;
  options.c_cflag &= ~CSTOPB;
  options.c_cflag |= CS8; //set no parity 8N1
  options.c_cflag &= ~CRTSCTS; // disable HW flow control
  options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); // raw input enable
  // options.c_lflag != ECHO;
  options.c_iflag &= ~(IXON | IXOFF | IXANY); // disables SW flow control
  options.c_oflag &= ~OPOST; // enables raw output
  tcsetattr(fd, TCSANOW, &options); // sets the new options

  unsigned char reqNodedisc[] = {0x31,0x31};
  unsigned char broadCast[] =
{0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0xFF,0xFE};
  int addrNum;
  unsigned char rssiSetup[] = {0x7E, 0x00, 0x05, 0x08, 0x01, 0x52, 0x50,
0xFF, 0x55};
  unsigned char *rssiMem;
  rssiMem = &rssiSetup[0];
  int i = 0;
  //for(i=0;i<9;i++)
  //write(fd, rssiMem,9);
  for(i=0;i<4;i++)
    powerB4[i] = 4;
  while(1){
    write(fd, rssiMem,9);
    printf("\n\rStart Optimization\n\r");
    tcflush(fd, TCIOFLUSH);
    apiTx(fd, reqNodedisc, broadCast, 3);
```

```
    printf("Mode 1\n\r");
    addrNum = nodeDiscovery(fd); // Make node discovery request
    printf("%d nodes found\n\r", addrNum);
    if(addrNum > 0)
      sendDiagnostic(fd, addrNum); //Send Diagnostic Packets
    i = 0;
    while(1200 > i){
      receiveData(fd);
      usleep(10000);
      i++;
    }
    storeData();
    printf("Do Routing Algorithm\n\r");
    routingAlgorithm();
    sendRoutes(fd);
    usleep(10000000);
    clearData();
  }
  return 0;
}
```

B. Appendix B

Table 23: Transmission Power Limiting Algorithm Tests

BER Test 1st time					BER Test 2nd time					Batt Test 1st time					Batt Test 2nd time				
1	C	R1	R2	R3	1	C	R1	R2	R3	1	C	R1	R2	R3	1	C	R1	R2	R3
ND	3	3	3	3	ND	2	3	3	3	ND	3	3	3	3	ND	3	3	1	2
LT	6	5	6	6	LT	3	5	6	4	LT	6	6	4	6	LT	6	6	0	6
CP	3	1	1	1	CP	3	1	1	1	CP	3	1	1	1	CP	2	1	0	1
SR	3	1	1	1	SR	3	1	1	1	SR	3	1	1	1	SR	2	1	0	1
2					2					2					2				
ND	3	3	3	1	ND	3	2	3	3	ND	3	3	3	3	ND	3	3	3	3
LT	6	6	6	0	LT	6	3	6	5	LT	6	6	4	6	LT	6	6	6	6
CP	2	1	1	0	CP	3	1	1	1	CP	3	1	1	1	CP	3	1	1	1
SR	2	1	1	0	SR	3	1	1	1	SR	3	1	1	1	SR	3	1	1	1
3					3					3					3				
ND	3	3	3	3	ND	3	3	3	2	ND	3	2	3	3	ND	2	1	3	3
LT	6	5	6	6	LT	6	6	0	4	LT	6	4	6	6	LT	3	0	6	6
CP	3	1	1	1	CP	2	1	0	1	CP	3	1	1	1	CP	2	0	1	1
SR	3	1	1	1	SR	2	1	0	1	SR	3	1	1	1	SR	2	0	1	1
4					4					4					4				
ND	3	3	1	3	ND	3	3	3	3	ND	3	3	3	2	ND	3	3	3	3
LT	6	6	2	6	LT	6	6	6	6	LT	6	6	5	4	LT	6	6	6	6
CP	3	1	1	1	CP	3	1	1	1	CP	3	1	1	1	CP	3	1	1	1
SR	3	1	1	1	SR	3	1	1	1	SR	3	1	1	1	SR	3	1	1	1
5					5					5					5				
ND	3	3	3	3	ND	3	3	3	3	ND	3	2	3	1	ND	3	3	3	3
LT	6	6	6	6	LT	6	6	6	6	LT	6	4	6	0	LT	6	6	6	6
CP	3	1	1	1	CP	3	1	1	1	CP	2	1	1	0	CP	3	1	1	1
SR	3	1	1	1	SR	3	1	1	1	SR	2	1	1	0	SR	3	1	1	1

Table 24: Path Limiting Algorithm Tests

Unaltered					Path (1,0) Bad BER					Paths (1,0) (3,0) (1,2) Bad BER					Path (1,0) Bad BER; Node R3 Bad Batt				
1	C	R1	R2	R3	1	C	R1	R2	R3	1	C	R1	R2	R3	1	C	R1	R2	R3
ND	3	3	3	1	ND	3	3	3	3	ND	3	3	3	3	ND	3	3	3	3
LT	6	6	5	2	LT	5	4	0	6	LT	6	5	6	6	LT	6	6	6	6
CP	3	1	1	1	CP	2	1	0	1	CP	2	1	1	0	CP	3	1	1	1
SR	3	1	1	1	SR	2	1	0	1	SR	2	1	1	0	SR	3	1	1	1
2					2					2					2				
ND	3	3	3	3	ND	3	3	3	3	ND	3	3	3	3	ND	3	3	3	1
LT	6	6	6	6	LT	6	5	4	6	LT	6	6	5	5	LT	6	6	5	0
CP	3	1	1	1	CP	3	1	1	1	CP	3	1	1	1	CP	2	1	1	0
SR	3	1	1	1	SR	3	1	1	1	SR	3	1	1	1	SR	2	1	1	0
3					3					3					3				
ND	3	2	1	2	ND	3	3	3	3	ND	3	3	2	2	ND	3	2	1	2
LT	6	3	0	4	LT	6	6	5	4	LT	6	6	4	4	LT	6	4	0	4
CP	2	1	0	1	CP	3	1	1	1	CP	1	1	0	0	CP	1	1	0	0
SR	2	1	0	1	SR	3	1	1	1	SR	0	0	0	0	SR	0	0	0	0
4					4					4					4				
ND	3	3	3	3	ND	3	2	3	3	ND	3	1	2	3	ND	3	2	3	3
LT	6	6	6	6	LT	6	4	6	6	LT	6	0	4	5	LT	1	4	4	6
CP	3	1	1	1	CP	3	1	1	1	CP	2	0	1	1	CP	0	0	0	0
SR	3	1	1	1	SR	3	1	1	1	SR	2	0	1	1	SR	0	0	0	0
5					5					5					5				
ND	3	3	3	3	ND	3	3	3	2	ND	3	3	3	1	ND	3	3	2	3
LT	6	5	6	6	LT	6	6	6	0	LT	6	5	6	0	LT	6	6	2	6
CP	2	1	1	0	CP	2	1	1	0	CP	2	1	1	0	CP	3	1	1	1
SR	2	1	1	0	SR	1	0	1	0	SR	2	1	1	0	SR	3	1	1	1