An Adaptive Single-Hop Medium Access Control Layer For Noisy Channels

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

Derek T. Sanders

Certificate of Approval:

Richard O. Chapman
Associate Professor
Computer Science and Software Engineering

John A. Hamilton, Jr., Chair
Associate Professor
Computer Science and Software Engineering

David A. Umphress
Associate Professor
Computer Science and Software Engineering

Martin C. Carlisle
Professor
Department of Computer Science
United States Air Force Academy

George T. Flowers
Dean
Graduate School

An Adaptive Single-Hop Medium Access Control Layer For Noisy Channels

Derek T. Sanders

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama
August 10, 2009

An Adaptive Single-Hop Medium Access Control Layer For Noisy Channels

Derek T. Sanders

Signature of Author

Date of Graduation

Dissertation Abstract

An Adaptive Single-Hop Medium Access Control Layer For Noisy Channels

Derek T. Sanders

Doctor of Philosophy, August 10, 2009
(Master of Software Engineering, Auburn University, AL, 2008)
(Bachelor of Wireless Engineering, Auburn University, AL, 2006)

351 Typed Pages

Directed by John A. Hamilton, Jr.

The work presented in this dissertation is for a contribution to the data link layer and its responsibility of managing the physical channel in a mobile ad-hoc network (MANET). Wireless networks in general are susceptible to noise in the spectrum, which can result in low throughput, loss of critical resources, and high re-transmission rates at the physical and link layers. As a result, there is a growing need for wireless technology that can continue to operate in the presence of noise. A mathematical algorithm has recently been developed which uses concurrent and super-imposed codes, which when applied to wireless communications allows for jam-resistant communications without a pre-shared secret. By leveraging this algorithm, the research for this dissertation will create a jam-resistant single-hop medium access control (MAC) protocol that adapts to the level of noise in the channel. The protocol will dynamically adjust the parameters for encoding to overcome the varying levels of interference.

The new protocol will allow for communications to continue in the presence of noise or jamming attacks.

Acknowledgments

- A special thank you to my parents and my six siblings who have supported me during this milestone in my life.

- Thank you to my graduate advisor and committee members for their support, advice, and encouragement.

- Thank you to the Information Assurance Center graduate students at Auburn University for your support.

- Thanks to the US Air Force Academy Professors Bill Bahn, Leemon Baird, and Martin Carlisle for their support, advice, and expertise.

- Thank you to my friend, Mark Kuhr, for keeping me motivated.

- Thank you to all my other friends who have been there for support and knowledge.

- Thank you to RAM Laboratories, Inc. who have funded me through this research.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliography follows van Leunen's *A Handbook for Scholars*.

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The earliest wireless communications and transmission control project known as ALOHA [Abramson 1970] evolved into the wireless technology seen today. The Packet Radio Network (PRNET) [Jubin and Tornow 1987] grew out of the development of ALOHA and became one of the earliest multi-hop multiple access packet networks. To distinguish multi-hop from single-hop, multi-hop systems or layers are concerned with the movement of packets across multiple nodes or hops. Single-hop, on the other hand, is only concerned with the link between two nodes or hops. As technology progressed, the hardware needed to create diverse networks shrank into a more manageable form and has subsequently allowed for the evolution of small mobile devices in which each device can act as a repeater. This sort of network is commonly referred to as a Mobile Ad-Hoc Network (MANET). These networks can be viewed as simple peer-to-peer networks in which each node will receive packets and either keep it for itself or forward it onto the next destination. Each of the nodes in this network communicates via the wireless medium with other nodes in range without relying on internal infrastructure. The MANET is said to be self-organizing since it should automatically detect any new nodes and infuse them with the rest of the network effortlessly. Due to the high mobility of these networks it is often difficult to effectively coordinate access to the medium. This is due to the fact that incoming nodes and moving nodes transmissions can easily inject noise into a previously reserved channel.

Some of the characteristics that can be generalized for MANETs are as follows [Agrawal and Zeng 2006]:

- **Dynamic Topologies:** Mobility in the network causes the network topology to change at random, and it is often hard to predict where a node may be for the next transmission. The high degree of mobility also has an impact on the power constraints for the nodes. As nodes move, a previously visible node may not be reachable due to one node being limited by antenna power.

- **Bandwidth Constraints:** Bandwidth is a two-fold constraint when referring to wireless communications. There is the physical bandwidth of the channel and there is also the overall throughput of the link's bandwidth. Most ad-hoc networks are constrained to the Industrial, Scientific, and Medical (ISM) band, and are in turn required by FCC mandates that any radio in the ISM use either Frequency Hopping Spread Spectrum (FHSS) or Direct Sequence Spread Spectrum (DSSS). When considering the bandwidth of a MANET it is important to note that the already limited bandwidth is further reduced after the effects of multiple access, fading, and interference have been considered. A problem of particular concern is channel saturation, which leads to congestion. This problem is compounded by the addition of noise and jamming attacks. This is due to the fact that the wireless spectrum is inherently error prone which further reduces the effective throughput of the channel.

- **Limited Energy Potential:** Due to the nature of mobile networks it is likely that the nodes will be running on a battery store. This impacts the nodes transmitting power. As previously mentioned, the different transmission powers coupled with mobility can sometimes create a unidirectional link. That is, one node is able to transmit to another but not vice versa.

- **Limited Physical Security:** Due to the fact that MANETs have no infrastructure, security is a particular problem. Negotiating security trust levels and key exchanges is a hard problem with no central authority to handle these credentials. Further problems can be found in the areas of Denial of Service (DoS) attacks, man-in-the-middle attacks, and eavesdropping.

Noise and jamming, whether caused by other nodes in the network or intentionally injected into the channel by an adversary, can significantly affect the ability of network communications to carry on. With the increase in noise there is also an increase in channel saturation, leading to higher re-transmissions at the data link layer which can pose further threats to the sustainability of the node in terms of its power source. Current mechanisms for handling noise (or avoiding collisions) are handled by the Medium Access Control (MAC) layer protocol. The current protocols do not necessarily handle the noise, rather, they try to avoid collisions in the channel by using various techniques including sensing it before transmission or by splitting the channel into smaller channels. It is inefficient to divide the channel due to the limited bandwidth already asserted in the ISM, and current channel-sensing protocols can starve their node in the presence of noise. These issues can have a significant impact on the ability on the network to sustain data flow and for other layers in the stack to properly carry out tasks.

To address interference, a new MAC layer is needed that can continue to operate in the presence of noise. A new error-correcting code based upon concurrent codes forms the backbone of this new protocol. The BBC (named after the creators Baird, Bahn, and Collins) algorithm [Baird, Bahn, Collins, Carlisle and Butler 2007], a subset of concurrent codes, is the specific coding scheme that will be applied to this research. Early research with this algorithm has shown that it can aid in the recovery of a message that has been affected by noise or collisions. Noise can affect transmissions by flipping bits, either 0's to 1's or vice versa. However, by leveraging this new algorithm the messages contained in the transmission can still be recovered up to a certain bit-error rate (BER). This research will create a new MAC layer, and its supporting facilities, which take advantage of the error correcting abilities of the BBC algorithm.

The data link layer transforms the raw transmission ability of the physical layer into a reliable link, which is responsible for the hop-to-hop communications. This layer also transforms the data from the network layer into manageable chunks of data called frames. It is the responsibility of the MAC sub-layer to handle error correction either through correcting code or by retransmitting corrupted frames. Furthermore, the MAC layer is responsible for solving channel access conflicts and coordinating the transmission from one node to the next. The data link layer is traditionally thought to handle these four main tasks:

- **Framing:** The data link layer separates messages either from the network layer into smaller transmissions frames, or combines frames from the physical layer into their original message for delivery to the network layer. The degree of framing (variable-size) has a direct impact on the error control facility.

- **Flow Control:** This facility coordinates the data that can be outstanding before an acknowledgement is received for proper transmission. Flow control effectively determines the channel saturation from the viewpoint of a single node.

- **Error Control:** During transmission at the physical layer the data can become corrupted. It is the responsibility of this facility to either detect and request retransmissions, or attempt to correct the bit errors. Types of correcting codes are block codes, linear block codes, and cyclic codes. One of the simplest forms of error detection falls into checksums. Another aspect of error control is to determine which frames have been lost and need to be retransmitted.

- **Medium Access Control:** In literature the MAC sub-layer is traditionally in control of the previous mentioned facilities in the data link layer. The main focus however is coordinating the access to a shared medium. It is responsible for resolving the problems that can arise when multiple nodes wish to access the channel.

The MAC layer's ability to manage the physical medium has a direct effect on how reliable a link is. It also has an impact on how efficient the link is in terms of overall data throughput. In other words, the MAC layer is the ultimate decider in the level of Quality of Service (QoS) a network can maintain. For this reason the design of a MAC protocol which handles the varying reliability of channel is of considerable importance.

1.1 Goals

At the conclusion of this research effort, this dissertation should demonstrate:

- A contribution to the area of MAC protocols for MANETs. This research will incorporate the BBC algorithm into a new MAC layer, called BBC-MAC. BBC-MAC will be a new approach to providing adaptive jam-resistant communications without a pre-shared secret. This will be validated using software-defined radios.
- A contribution to the current BBC algorithm by providing methods to vary the coding parameters to allow for various levels of jam-resistance to be used. This will allow BBC-MAC to adjust to varying levels of interference. This will be validated using software-defined radios.
- The main research contribution contained in this proposal is a MAC layer solution to providing jam-resistant communications that can continue during a

jamming attack. The layer will achieve this by dynamically adjusting the current level of jam resistance with respect to the level of interference.

1.2 Challenges

Creating a new MAC layer that is jam-resistant and that can be proven on a real-world test bed presents several challenges. As previously mentioned, the MAC layer has a direct impact on the QoS for a link, and creating one that in the end improves upon the current state of MAC protocols is the main challenge. However, by incorporating the BBC algorithm into this new layer, the current state of MANET communications can be advanced to provide greater data transfer reliability. However, the following challenges will need to be addressed.

- The new layer must be able to effectively incorporate the BBC algorithm such that the coding parameters can be altered on a link-state basis. If the link has a low degree of noise, the level of encoding can be changed such that throughput goes up. Conversely, if the link has a high degree of noise, the layer should adjust the algorithm such that greater jam-resistance is achieved.

- The new layer must effectively handle the congestion and saturation of the channel by incorporating the proper flow and error control facilities. These facilities must be adopted to take advantage of the important jam-resistant nature of the BBC algorithm.

- The MAC protocol must effectively coordinate access to the transmission channel amongst multiple nodes while maintaining the proper level QoS. It is unclear how to properly configure the MAC protocol to control all the other facilities of the data link layer, and is the main focus of this research.

1.3 Outline

The remainder of this document is organized as follows:

- Chapter 2 gives an overview of the lower functions of wireless communications including radio propagation and the physical multiplexing that occurs at the physical level. The chapter concludes with an overview of the BBC algorithm and its operations.

- Chapter 3 introduces the specific duties of the data link layer with a main focus on the discussion of current and past medium access control (MAC) protocols for consideration.

- Chapter 4 discusses the initial design of the protocol.

- Chapter 5 covers the protocol design and implementation.

- Chapter 6 covers the initial experiements for determining the proper configurations needed to create the adaptive protocol.

- Chapter 7 covers the experiments and validation for the adaptive protocol.

- Chapter 8 discusses the contribution to the research field.

- Chapter 9 concludes with a discussion of this dissertation and future work.

Chapter 2

Wireless Technology Overview

2.1 Chapter Introduction

Understanding the important functions of the lower layers of the wireless protocol stack is crucial for gaining insight into the problems that mobile wireless networks are faced with. The lowest level of interaction is at the physical layer and it is at this layer where noise makes its impact. A layer up sits the data link layer that is tasked with transforming the raw data transmissions provided by the physical layer into a reliable data link usable by the upper layers. This chapter is focused upon giving the reader an overview of the important components in wireless communications. The important concepts within radio propagation will be covered. Additional topics include an explanation of the physical multiplexing techniques, an overview of signal jamming, and an in depth overview of the BBC algorithm.

Before continuing into the details of wireless communications a lexicon of terms is provided for the reader as a friendly reminder of the definitions [Forouzan 2007].

Terminology:

- **Bandwidth:** The difference between the highest and the lowest frequencies of a composite signal.

- **Channel:** A communications pathway.

- **Guard Band:** The bandwidth separating two signals in a composite signal.

- **Link:** The physical communications pathway that transfers data from one device to another.

- **Multiplexing:** The process of combining signals from multiple sources for transmission across a single data link.

- **Spectrum:** The range of frequencies of a signal.

- **Spread Spectrum:** A wireless transmission technique that requires a bandwidth several times the original bandwidth.

2.2 Mobile Radio Propagation

Communications in a MANET use a wireless transmission medium in order to exchange data. For this reason it is important to understand the distinguishing characteristics for radio propagation. Ideally, radio waves would move freely in space without any obstacles and free from interference. However, this is not possible in the real world except in a lab environment where the waves are propagating through a vacuum. When a radio wave does encounter an obstacle it can affect the wave through reflection, diffraction, or scattering. [Agrawal and Zeng 2006]

1. **Reflection:** Reflection occurs when the radio wave encounters an object that is larger compared to the size of its wavelength. This can be seen when the radio wave hits the side of a building, where it will be reflected off the building.

This scenario can be viewed as a positive event since it allows more waves to reach the receiver than would normally, but it also presents a problem since the receiver will have multiple copies of the same wave.

2. **Diffraction:** Diffraction occurs when radio waves are blocked by an object with sharp irregular edges. The radio waves will bend around the corner to reach the receiver. Like reflection this allows waves to reach the receiver even in situations where line of sight does not exist.

3. **Scattering:** Scattering occurs when the radio wave encounters an object that is smaller compared to the size of its wavelength and the incoming wave is scattered into several weaker outgoing signals. An example would be when a radio wave hits street signs or lampposts.

2.3 Physical Multiplexing and Spreading Techniques

The physical layer of the network stack is charged with the physical movement of bits from one node to the next. It is the interface that connects the rest of the protocol stack to the physical medium for transport. This physical layer operates on a stream of bits that are encoded or modulated into an electrical or optical signal for transport. The layer is also concerned with the data rate over the medium. The upper bound on the communications network is always going to be the number of sustainable bits sent each second over the physical medium. The physical layer also handles the synchronization that is required at the bit level for communications to take place. The final important aspect to the physical layer is how it multiplexes the digital stream of bits into a transport form over the wireless medium. Applying specific multiplexing and spreading techniques can efficiently use the bandwidth of the channel. When using multiplexing the goal is to create an efficient use of the channel by combining multiple signals into a single signal. Spreading the signal allows for privacy and the resistance to signal jamming. These techniques generally fall into the domains of time, frequency, and spreading [Forouzan 2007, Agrawal and Zeng 2006].

- Frequency-Division Multiplexing (FDM): Frequency-division multiplexing is an analog multiplexing technique that combines multiple signals. This is used when the bandwidth (hertz) of a link is greater than that of the combined signals being transmitted. The individual signals generated by the devices are modulated on different carrier frequencies. These are then combined into a single composite signal to be sent out over the medium. The carrier frequencies are sufficiently separated by guard bands to prevent overlap between the individual signals.

- Time-Division Multiplexing (TDM): Time-division multiplexing is a digital multiplexing technique that combines multiple low-rate channels into a single high-rate channel. In contrast to FDM, TDM shares time on the medium versus frequency as in FDM. Each node that is connected to the medium is given a certain portion of time on the link in which it can occupy. There are two prevailing methods of doing TDM: synchronous and statistical. The main difference between the two is that in synchronous mode, a node is allocated a time slot even if the node does not have any data to send. Statistical TDM dynamically allocates time units as needed which improves the bandwidth efficiency.

- Orthogonal Frequency-Division Multiplexing (OFDM): Orthogonal frequency-division multiplexing is a technique to split high-rate radio signals into several low-rate signals that are then transmitted over several orthogonal carrier frequencies. The sending node breaks down the high-rate streams into n parallel low-speed streams that are then modulated. The key difference between OFDM and FDM is that in OFDM all the sub-bands are used by a single source at one time, instead of in FDM where the sub-bands are taken up by separate sources. OFDM is used as the multiplexing technique in 802.11a/g/n.

- Spread Spectrum (SS): Spread spectrum is a technique like multiplexing that brings together multiple signals for transmission. SS was originally designed for military use to avoid jamming in the wireless spectrum. In wireless communications, nodes must be able to share the medium in a manner that allows for privacy from eavesdropping and without being susceptible to jamming. SS takes the original signal's required bandwidth and expands it such that the spreaded bandwidth is much larger (usually twice) that of the original bandwidth. After the signal has been created, the spreading process uses a spreading code or chip-sequence, which determines how the original signal is spread in the new bandwidth. Currently there are two main techniques for spreading the bandwidth: Direct Sequence Spread Spectrum (DSS) and Frequency Hopping Spread Spectrum (FHSS).

1. Direct Sequence Spread Spectrum (DSSS): Direct sequence spread spectrum multiplies the original signal by a pseudorandom sequence of bits that is much larger than the original signal, effectively spreading the original signals bandwidth. In other words, each data bit in the original signal is multiplied by the chip sequence using polar non-return to zero (NRZ) encoding. DSSS provides privacy from eavesdropping as long as no other nodes have access to the code. DSSS is resistant to interference in the spectrum if each node uses a different spreading sequence.
2. Frequency Hopping Spread Spectrum (FHSS): Frequency hopping spread spectrum uses a pseudorandom sequence to spread the original signal across a larger bandwidth. The sequence determines how the radio signal hops between the multiple carrier frequencies.

2.4 Signal Jamming

Wireless communications are prone to errors during transmission. Signal jamming disrupts the transmission and can occur through un-intentional means such as interference, collisions, or noise. This type of jamming can occur in situations of high network saturation where competing nodes are causing collisions in the spectrum. A more significant threat are jamming attacks from adversaries attempting to disrupt or bring down the network.

- **Unintentional Jamming:**

  Friendly jamming is a common occurrence in current wireless communication systems such as 802.11. The collisions that occur at the physical layer are resolved by the data link layer, and generally go unnoticed by the user operating at the application layer. It is only in situations of high network congestion and noise where the problem can be seen in terms of lost packets and high latency. Collisions occur when multiple stations transmit at the same time onto a channel that was designed to only support one transmission. When this happens the signals are combined, which effectively destroys or corrupts the data from the individual transmissions. The two most familiar situations that can cause unintentional jamming are the exposed and hidden terminal problems [Forouzan 2007].

  1. *Hidden Terminal Problem:* The hidden terminal problem is depicted in Figure 2.1. In this situation terminal A is able to see the signals broadcasted from both B and C, but B and C are hidden from each other with respect to A. Consider the scenario when terminal B is sending data to terminal A. While this transmission is occurring terminal C also wishes to send data to terminal A. The problem is terminal C can't sense the channel to see that terminal B is transmitting since C is out of range of

B's transmission radius. When the two begin to transmit it will cause a collision corrupting the data A is receiving.

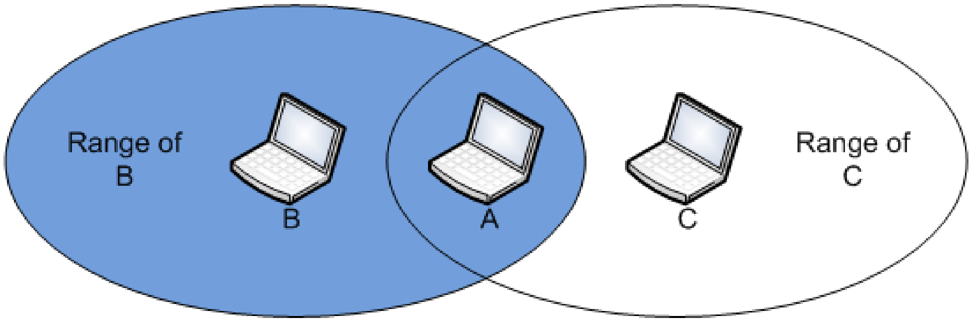

Figure 2.1: Hidden Terminal Problem

2. *Exposed Terminal Problem:* The exposed terminal problem is depicted in Figure 2.2. In this situation the problem is that terminal C is exposed to the transmissions from terminal A to B. Consider the scenario where terminal C wishes to send data to terminal D, and at the same time terminal A is transmitting data to terminal B. Terminal C could send data to D without interfering with the data from A to B, however, since it is being exposed to the transmissions from A to B, it will not begin transmitting to D. In other words, terminal C is wasting time and the actual channel availability by waiting for terminal A to complete its transmission to B.

- **Intentional Jamming:**

  As mentioned before the second type of jamming occurs when an adversary wishes to attack a network. Jamming is a relatively easy task since in the general case no special hardware is needed to carry out the attack, and it can be

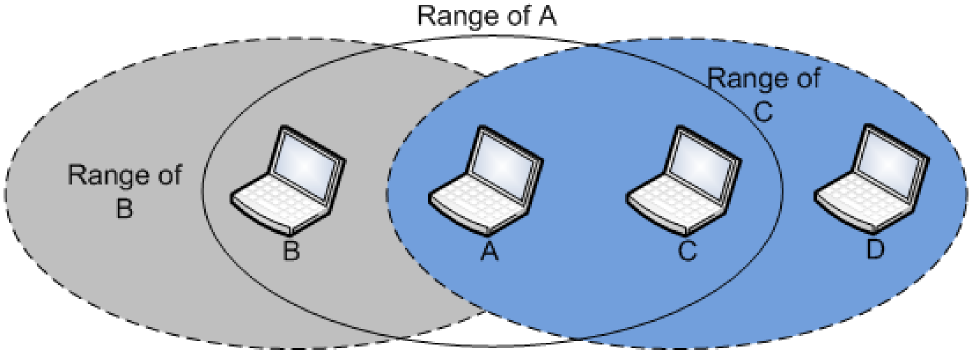

Figure 2.2: Exposed Terminal Problem

implemented by merely listening to the medium and broadcasting at the same frequency, and when carried out correctly it can lead to significant network and communications disruptions [Awerbuch, Richa and Scheideler 2008]. The method of attack is usually targeted at the physical medium for the network, but more sophisticated attacks can target the specific way the MAC protocol operates in the data link layer. Methods for carrying out jamming attacks have been studied and validated through simulation [Chiang and Hu 2007, Law, van Hoesel, Doumen, Hartel and Havinga 2005, Li, Koutsopoulos and Poovendran 2007, Xu, Trappe, Zhang and Wood 2005]. Current defenses against jamming focus on special techniques at the physical layer, such as spreading techniques [Forouzan 2007, Liu, Noubir, Sundaram and Tan 2007, Navda, Bohra, Ganguly and Rubenstein 2007]. Current wireless technologies like 802.11b use a form of spread spectrum. However, 802.11b uses narrow spreading which allows an attacker to jam only a small set of frequencies rendering spread spectrum useless. Furthermore, the MAC protocol in 802.11 does not offer any protection to even the simplest jamming techniques [Forouzan 2007, Bayraktaroglu, King, Liu, Noubir, Rajaraman and Thapa 2008].

2.5 BBC Algorithm Overview

The goal of this research is to create a new MAC layer that provides jam-resistance without a pre-shared secret by taking advantage of the BBC algorithm [Baird, Bahn, Collins, Carlisle and Butler 2007]. Given its pivotal role in this research effort it is important to gain a clear understanding of how it will allow the new layer to accomplish the task of maintaining a reliable link even in the presence of noise. This section will cover the terminology used when referring to BBC operations, explain by example how the algorithm conducts its encode and decode steps, and finally noise will be added to the example to illustrate how it overcomes that obstacle.

2.5.1 Introduction

Current technologies such as spread spectrum provide jam-resistance, however, the two communicating parties must possess the same chip sequence in order to communicate in a private and jam-resistant manner [Forouzan 2007]. Managing the chip sequences for every node is similar to the problem that was faced by the cryptographic community prior to the movement to a public key infrastructure. Prior to public key cryptography both parties had to know the symmetric key in order to cipher messages between the each other. In order to overcome this problem, new wireless technologies are needed that allow for communications to occur which provide jam-resistance and privacy, but also eliminate the need for secret knowledge (chip sequence). The creators of the BBC algorithm had this problem in mind when creating the algorithm. The algorithm allows the communicating parties to talk without a pre-shared key while affording jam-resistance.

Terminology

The following glossary of terms is presented for the reader. Many of these terms will be used when referencing the BBC algorithm in this Section and the remainder of the dissertation.

Indelible Mark The location of a 1 bit, or a high pulse in a transmission. It is assumed that the mark can never be transformed from a 1 to a 0.

Data The payload that is encapsulated in a message.

Message The fully constructed message including the necessary checksum bits and header information.

Packet This is the combination of multiple messages that are combined with a bitwise OR. The packet is the final data which the BBC algorithms are enacted upon.

The BBC algorithm operates in two modes: encoding and decoding. The encoding stage transforms binary data into a form, which determines how it is to be physically transmitted. The parameters given to the encoder determine the level of jam-resistance it affords. The following sections show by example how the BBC algorithm operates.

2.5.2 BBC Encoding

Algorithm 1 BBCEncode(M)

This function encodes an m-bit message $M[1...m]$ adding k checksum bits to the end of the message. H is a hash function. The definition of H and the value of m and k are public (not secret). The definition of "indelible mark" and "location" are specific to the physical instantiation of BBC used.

```
Append k zero bits to the end of $M$
for $i = 1 ... m + k$ do
    Make an indelible mark at the location given by H($M[1...i]$)
end for
```

The BBC Encoding algorithm is shown by Algorithm 1 [Baird, Bahn, Collins, Carlisle and Butler 2007]. It is a fairly straightforward process, compared to the steps taken during decoding. The first thing that is done is the original message is appended with k checksum (zero) bits. The number of bits is determined in advance based upon the coding parameters and the expected number of errors that is determined by the current interference detected. Next, each prefix of the bit string is sent through a hash function that maps a variable length bit string to some desired mapping where the indelible mark will be. In this example pulse broadcast is used and so it is conceptually mapped to a bucket number representing a period of time where a pulse would be. Using the algorithm, and the example in [Baird, Bahn, Collins, Carlisle and Butler 2007] which uses 25 buckets and 2 checksum bits, the encoding of the message $M = 1000$ proceeds as follows:

1. Append two checksum zeros to M: 1000**00**.

2. Encode each prefix string, *s*, using the hash function, *H* as shown in Table 2.1.

s	**H(s)**
1	21
10	9
100	20
1000	14
10000	6
100000	10

Table 2.1: Prefix Hash Table

3. Broadcast this message by transmitting a pulse where there is a corresponding 1 in the buckets from Table 2.1. The result of this broadcast in the time period conceptualized by the buckets [0,25] is seen in Table 2.2.

Bucket	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1000	0	0	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0

Table 2.2: Transmission Buckets

2.5.3 BBC Decoding

Decoding on the receiver's end is considerably more complex than the steps taken to encode the message. This is because the receiver is unaware of what is sent and must begin at the very beginning of any root message and systematically reduce the set of possible messages. The algorithm for decoding the received message is given by Algorithm 2 [Baird, Bahn, Collins, Carlisle and Butler 2007]. It is assumed that the receiver knows the hash function, length of messages, and the length of k. From the encoding example, assuming there was no noise induced in the message, and

Algorithm 2 BBCDecode(n)

This recursive function can be used to decode all the messages found in a given packet by calling BBCDecode(1). There must be a global $M[1...m+k]$ which is a string of $m+k$ bits. The number of bits in a message is m, and the number of checksum zeros appended to the message is k. The definition of H and the value of m and k are public (not secret). The definition of ”indelible mark” and ”location” are specific to the physical instantiation of BBC used.

```
if $n = m + k + 1$ then
  print "One of the messages is:" $M[1...m]$
else
  if $n > m$ then
    $limit \Leftarrow 0$
  else
    $limit \Leftarrow 1$
  end if
  for $i = 0$ ... $limit$ do
    $M[n] \Leftarrow i$
    if there is an indelible mark at location H($M[1...n]$) then
      BBCDecode($M$,$n + 1$)
    end if
  end for
end if
```

the receiver began listening at the proper time the queue would look like Table 2.2. Following the decoding algorithm the steps to decode this message would proceed as follows:

1. Determine whether a 0 or 1 was transmitted. H(0) = 4 and H(1) = 21. The receiver will listen for pulses at time slots 4 and 21. From Table 2.2 it is seen that bucket 21 has a pulse, and thus the receiver knows that the message begins with a 1. M′ = 1.

2. Next, the current set of prefixes are appended with 0 and 1 to account for all prefixes. M' = 10, 11. H(1**0**) = 9 and H(1**1**) = 21. Both of these locations have pulses and will survive onto the next decoding iteration. $M' = 10, 11$.

3. Again, the current set of prefixes are appended with a 0 and 1. $M' = 100, 101, 110, 111$. H(10**0**) = 20, H(10**1**) = 24, H(11**0**) = 16, and H(11**1**) = 2. Cross-referencing with Table 2.2, it can be seen that only bucket 20 has a pulse and thus 100 is the only survivor. $M' = 100$.

4. Appending the current set of prefixes gives $M' = 1000, 1001$. H(100**0**) = 14, H(100**0**) = 1. Only bucket 14 has a pulse and reduces the set to $M' = 1000$.

5. At this stage the length of the original message has been reached. Thus, from this point on the surviving prefixes will be appending with the checksum bits (0-bits) for at most k times. H(1000**0**) = 6. This does have a pulse and continues onto the final decoding stage. $M' = 10000$.

6. This is the last decoding step since this is the last checksum bit to be appended. H(10000**0**) = 10, and bucket 10 does indeed have a pulse. Removing the k checksum bits from the surviving set of M' reveals that the only message was sent = 1000, and this matches up with what was encoded in Section 2.5.2.

2.5.4 BBC Decoding With Noise

The decoding example in section 2.5.3 illustrated the basic concept of how to use the BBC algorithm for physical encoding and decoding of the messages. However, it lacked the illustration of how the algorithm will decode when a few of the bits are flipped during transmission. It is assumed that the induction of power into the spectrum can only flip the bits from 0 to 1 and not vice versa. For the sake of completeness it will be shown how the algorithm decodes the message when just two bits are flipped, or in this case, there are two buckets that get pulses. Using Table 2.2 as the basis, the following buckets are given pulses: 2 and 24. These buckets are marked with an X in Table 2.3 to differentiate them from the true pulses sent out by the sender.

Bucket	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1000	0	X	0	0	0	1	0	0	1	1	0	0	0	1	0	0	0	0	0	1	1	0	0	X	0

Table 2.3: Received Buckets With Noise

The decoding proceeds as follows:

1. Determine whether a 0 or 1 was transmitted. H(0) = 4 and H(1) = 21. The receiver will listen for pulses at time slots 4 and 21. From Table 2.2 it is seen that bucket 21 has a pulse, and thus the receiver knows that the message begins with a 1. M′ = 1

2. Next, the current set of prefixes are appended with 0 and 1 to account for all prefixes. M' = 10, 11. H(1**0**) = 9 and H(1**1**) = 21. Both of these locations have pulses and will survive onto the next decoding iteration. M′ = 10, 11.

3. Again, the current set of prefixes are appended with a 0 and 1. M′ = 100, 101, 110, 111. H(10**0**) = 20, H(10**1**) = 24, H(11**0**) = 16, and H(11**1**) = 2. Cross-referencing with Table 2.3, it can be seen that buckets 20, 24, and 2 have pulses. M′ = 100, 101, 111.

4. Appending the current set of prefixes gives M′ = 1000,1001,1010,1011,1110,1111. H(100**0**) = 14, H(100**0**) = 1, H(101**0**) = 15, H(101**0**) = 2, H(111**0**) = 14, H(111**0**) = 23. Buckets 14 and 23 have pulses, and two of the prefixes mapped to 14 which gives the set M′ = 1000, 1011, 1110.

5. At this stage the length of the original message has been reached. Thus, from this point on the surviving prefixes will be appending with the checksum bits (0-bits) for at most k times. H(1000**0**) = 6, H(1011**0**) = 14, H(1110**0**) = 13. Bucket 6 and 14 have transmissions and the follow prefix set survives: M′ = 10000, 10110.

6. This is the last decoding step since this is the last checksum bit to be appended. H(10000**0**) = 10, H(10110**0**) = 12. Only bucket 10 has a pulse leaving M′ = 100000. Removing the k checksum bits from the surviving set of M′ reveals that only message was sent = 1000, and this matches up with what was encoded in Section 2.5.2.

This example illustrated clearly how the algorithm reduces the set of possible prefixes to get the actual message even in the presence of corrupted transmissions. The interesting occurrences are the two surviving messages past the original message's length. These two messages that got eliminated are called hallucinations as termed by the authors of [Baird, Bahn, Collins, Carlisle and Butler 2007]. This example illustrates the importance of the k checksum bits. In the previous decoding example it might have been thought to just stop decoding since the original message was recovered. However, in this example if that were to have occurred three messages would have thought to been received, and even onto the second to last stage two messages would have been received. It is only after the final decoding step where it is determined how many true messages were sent.

2.6 Chapter Conclusion

This chapter began with an introduction to wireless communications and the operations that occur at the physical layer for transport. An introduction to the various signal jamming scenarios was presented, and the chapter concluded with an introduction to the BBC algorithm. A toy example was worked through that demonstrated how the BBC algorithm encodes and decodes data, and finally, a decoding example where noise was artificially induced was given.

Chapter 3

Medium Access Control Layer

3.1 Chapter Introduction

In relationship to the OSI model, the IEEE 802 standard specifies that the data link layer be divided into two sub-layers: the logical link control (LLC) layer and the medium access control (MAC) layer. The separation is made to allow the MAC protocol layer to be specific to the type of physical medium used. For example, the LLC in the IEEE 802 standard is the same across all the different local area network (LAN) protocols, and it is only the MAC that is modified as necessary [Forouzan 2007]. For instance, IEEE 802.3 Wired Ethernet LAN uses a Carrier Sense Multiple Access (CSMA) with Collision Detection, and IEEE 802.11 Wireless LAN uses a specialized version of CSMA with Collision Avoidance, but both use the same IEEE 802.2 LLC.

3.2 Flow and Error Control Protocols

The specific duties assigned to the LLC according the IEEE 802 standard are to provide flow and error control. Flow control refers to the set of mechanisms that dictate the number of outstanding frames that the sender can transmit without receiving an acknowledgement (ACK) frame. Error control is the correction of the problem when an acknowledgement is never received, the receiver never gets an unexpected frame, or the receiver gets a corrupted frame. It is based on the concept of automatic repeat request, or the retransmission of data. Flow and error control are accomplished through the use of a single protocol, with the exception of the error detection mechanisms. The following lists the most familiar protocols to accomplish error and flow control [Forouzan 2007].

- **Stop-and-Wait Automatic Repeat Request:**

  This is the simplest of the flow and error control protocols. This protocol will have one outstanding frame at any point in time. It will send one frame, and then wait for an ACK frame to be returned from the receiver, or for a timer to expire, in which it will automatically re-transmit the unacknowledged frame. The error control is achieved through the use of the timer. It is assumed that if there isn't an ACK received after a specific time that an error has occurred during transmission and the frame never arrived or the receiver received a corrupted frame. Sequence numbers are used for identifying frames, based on modulo-2 arithmetic. The sequence number inside the returned ACK frame is the number for the frame that the receiver is expecting next.

- **Go-Back-N Automatic Repeat Request:**

  This protocol expands upon the previous one by allowing multiple frames to be sent at once. An abstraction known as the sliding window is used, where within the sliding window resides the frames with the sequence numbers that have not been acknowledged. The window can only slide when a valid acknowledgement is received. Generally, the window can only slide one slot at a time. However, since it is assumed that the receiver only sends back the ACK for the next frame expected, the window can slide directly to that frame number, and send out any frames in the window. This is because the receiver may not send back an ACK for every frame it receives, just the most recent in order frame. Like the Stop-and-Wait protocol, this also uses timers for error control. However, there is only one timer that is kept track of, versus one for each frame. The timer is only maintained for the oldest outstanding frame, and if that timer expires all of the frames within the window are retransmitted.

- **Selective Repeat Automatic Repeat Request:**

  The previous two protocols simplified the process carried out at the receiver's end. The receiver only had to maintain one buffer, the space for the next frame expected. Any out of order frames that were received were simply discarded, and this is a very inefficient use of the link. This problem can be further compounded in noisy channels, like that in wireless communications, where a frame has a high probability of being corrupted. These retransmissions use valuable link bandwidth and further add to the probability of a corrupted frame. Selective Repeat is a protocol meant for noisy channels. In this protocol instead of the sender sending back all the frames in the window, it only retransmits those that have not been acknowledged or have been corrupted. This allows for efficient use of the link, but makes the processing on the receiver's end much more complex. The receiver can receive as many out of order frames as the window size and will store them until enough in order frames arrive to deliver to the upper layer. The receiver makes use of non-acknowledgment (NAK) frames, which are sent to the sender to remind them to retransmit a specific frame. A final level of complexity in this protocol is that every frame is given a timer, since it is only sending back the corrupted frames, and not all the frames in the window.

These protocols provide the necessary functionality for error and flow control at the data link layer. This research will however not distinguish them from the MAC layer, and will instead consider them to be under the control of the protocol guiding the access to the medium.

3.3 Wireless Medium Access Control Protocols

The MAC layer is responsible for solving the errors and anomalies that can occur at the physical layer. It is the responsibility of this layer to resolve the conflicts that arise when multiple nodes wish to use a single channel. The specific protocol used can have a direct impact on the efficiency of the link and for this reason it is important to consider the quality of service (QoS) constraints when designing a new MAC layer. The protocols can be divided into two main categories: contention free and contention based schemes, and then there are those that combine the two to form hybrid protocols. Contention based schemes can be further divided into those which operate on random access versus those that attempt to reserve the channel and resolve collision. These protocols can be further divided into single channel, multi channel, power aware, and quality of service (QoS) based protocols [Kumar, Raghavan and Deng 2006].

3.3.1 Contention Free Schemes

Contention free schemes are those that divide the channel in such a way that no two nodes should ever be competing for access to the channel at any point in time. These are sometimes called channelization access schemes. These types of schemes divide the available bandwidth of the link into multiple channels through time, frequency, or through codes, and others use polling or are token-based systems. The most familiar of these protocols are

- **Frequency Division Multiple Access (FDMA):**

  Frequency division multiple access (FDMA) divides the available bandwidth into multiple frequency bands. Each node is then allocated a specific band on which it can transmit data. Like in FDM, guard bands separate the individual bands. However, while FDM and FDMA conceptually operate the same there is a key difference. As mentioned in Section 2.3, FDM is a physical layer multiplexing technique that combines the data from multiple low-bandwidth channels and transmits them over a single high-bandwidth channel. The difference is that FDMA tells the physical layer to make a band pass signal from the data that is given to it limiting the frequency that the node is transmitting on. The signals from each station are then transmitting at different frequencies and are combined when they put on the single channel [Forouzan 2007]. FDMA has been applied to various multiplexing schemes in literature including the OFDM-FDMA and OFDM-interleaved-FDMA [Wong, Cheng, Lataief and Murch 1999] schemes.

- **Time Division Multiple Access (TDMA):**

  Time division multiple access (TDMA) divides the channel in time. Each node is given a specific time slot in which data can be transmitted on its behalf. TDMA suffers from a synchronization issue since each station needs to know exactly when a new time slot is beginning in order to effectively transmit at the correct time. Again, it needs to be clear that TDMA and TDM as mentioned in Section 2.3 are conceptually the same, but achieve different goals. TDM combines the data of slower channels into a single faster channel using a multiplexer that interleaves the data. TDMA however tells the physical layer to use a specific time slot [Forouzan 2007]. TDMA has been supplemented as an access scheme in literature [Wang and Xiang 2006, van Hoesel, Nieberg, Kip and Havinga 2004, Kanzaki, Hara and Nishio 2007, Gerla and Tzu-Chieh Tsai 1995].

- **Code Division Multiple Access (CDMA):**

  Code division multiple access (CDMA) is a scheme in which a single channel carries all the data from multiple nodes simultaneously. It is based on coding theory, much like the spreading techniques described in Section 2.3. While

CDMA and DSSS might seem similar there is a clear distinction. CDMA uses multiple orthogonal spreading sequences to allow for the multiple node access on the same frequency. However, in the implementation of 802.11b DSSS, every node uses the same spreading sequence, but allows the nodes to choose from multiple frequencies for simultaneous operation. In CDMA each station is given a specific code called a chip sequence. By assigning each station their own code multiple stations can communicate on a single channel without interfering with other communicating nodes, assuming they know each other's chip sequence [Forouzan 2007]. CDMA has been proposed [Muqattash and Krunz 2003, Garcia-Luna-Aceves and Raju 1997, Joa-Ng and Lu 1999, Lee and Cho 1995, Sousa and Silvester 1988] and tested [Hui 1984] as a protocol for MANETs in literature.

3.3.2 Contention Based Schemes

Protocols that operate on the foundation that nodes must compete for access to the channel are considered contention-based schemes. These are generally called random access protocols where no station is considered to be superior to another. For this reason the MAC layer for contention based schemes can be considerably more complicated than those for controlled access or channelized layers.

- **ALOHA:**

  ALOHA is the earliest random access protocol developed by the University of Hawaii in the early 1970's. The original protocol is sometimes referred to as

pure ALOHA. The protocol is simple in that whenever a station wishes to send a frame it does so. To recover from errors the protocol uses acknowledgments from the receiver. If the sender doesn't receive an acknowledgment after a time-out period it assumes that frame has been lost. This is similar to the error control protocols discussed in Section 3.2. When the timeout does occur pure ALOHA requires that the sending node wait a random amount of time before retransmitting. By waiting a random period time, the idea is to avoid more collisions. Additionally, in order to avoid congestion from retransmits the protocol further dictates that after a specified number of retransmits the station must give up on that frame. A later modification to ALOHA is with **slotted ALOHA**. Much like TDMA, slotted ALOHA divides channel access into periods of time called slots, where a node is only allowed to transmit during their specified slot. However, collision can still occur if two stations try to send at the same time slot [Forouzan 2007, Abramson 1970].

- **Carrier Sense Multiple Access (CSMA):**

  Carrier Sense Multiple Access (CSMA) is a protocol where a station is required to sense the medium prior to transmitting. CSMA is an evolution of ALOHA in the sense that it is reducing the chance of a collision because it senses the channel, but it cannot eliminate collisions. When the station senses that the channel is idle or busy there are three methods that can be used to determine how to precede [Agrawal and Zeng 2006].

- **1-Persistent Method:** If the channel is idle, send the frame immediately. If it is busy, keep listening until it is idle and then transmit.
- **Non-persistent Method:** If the channel is idle, send the frame immediately. If it is busy, wait a random amount of time and then sense the channel again.
- ***p*-Persistent Method:** In this method time is considered to be slotted. Each time slot is considered to be the contention period, usually equal to the round trip propagation time. When there is a frame to send the station first senses the channel. If it finds the channel to be idle it follows these steps:

    1. With probability p, the station sends its frame.
    2. With probability q = 1 - p, the station waits for the beginning of the next time slot and senses the channel again.

        (a) If the line is idle, proceed to step 1.

        (b) If the line is busy, acts the same way as in a collision. Waits a random amount of time and starts all over.

- **Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA):** The basic CSMA with collision avoidance (CSMA/CA) protocol relies upon three important strategies in order to provide collision avoidance: inter-frame space, contention window, and acknowledgements. When a station has a frame to send it senses the channel. If it is idle the station will not send immediately, but rather defer transmitting for a specified amount of time called the inter-frame space (IFS). After this time it senses the channel again, if it is idle the station can transmit after waiting for a period of time to pass called the contention time. The contention window is a time period divided into slots. When the station is ready to transmit it must chose a random number of slots to wait before the transmission can occur. At each time slot the channel is sensed, and if it is busy it must stop its timer, and can only start the timer when the channel is sensed idle again. Once the timer goes to zero, the transmission can occur. Even with the other precautions a collision can occur. The protocol uses timers and acknowledgments to recover from corrupt or lost frames.

The basic CSMA/CA protocol still suffers from the hidden and exposed terminal problems discussed in Section 2.4. In order to overcome these problems channel reservation mechanisms have been amended to CSMA/CA. The most basic solution is through the use of request-to-send (RTS) and clear-to-send (CTS) control frames. When a station wishes to send a frame it sends a RTS frame to the receiver. Tobagi and Kleinrock first proposed the exchange of RTS/CTS in the split-channel reservation multiple access (SRMA) scheme [Tobagi and Kleinrock 1976]. The receiver will reply with a CTS frame. The exchange of these frames still follows the protocol explained previously for sending a data frame. The idea with this exchange is that the channel has been reserved for communication between these two stations. The RTS also lets the receiver know that data is coming and gives it the ability to allocate buffer space for the transmission. Any other station that received these RTS or CTS frames will defer its own transmissions to further reduce the chance of collisions [Forouzan 2007, Agrawal and Zeng 2006].

- **Multiple Access Collision Avoidance (MACA):**

  The Multiple Access Collision Avoidance (MACA) protocol was proposed by Karn to overcome the problems faced by the basic CSMA/CA protocol discussed above, namely the hidden and exposed terminal problems [Karn 1990]. MACA uses small signaling packets like the RTS and CTS control packets used in CSMA/CA. However, the author drops the carrier sense aspect of CSMA, and instead focuses on extending the CA aspect. The dropping of the CS amounts to the ALOHA protocol with RTS and CTS control frames. As mentioned previously, when other stations overhear a RTS or CTS control frame they are required to defer their transmission for some period of time. It is unclear however how long these stations should defer using the channel, and this is where Karn's work is important. In MACA the sender includes the size of the data being transferred in the RTS frame, and the receiver will return that same information in the CTS frame. Any node that receives these frames will know approximately how long they should defer transmitting based on the size of the data. A further benefit of this protocol is that the size of the RTS/CTS frames is quite small, compared to the data packets, reducing the probability and risk that collisions between them present. However, MACA does not use acknowledgement frames for the data packets at the MAC layer, and instead leaves it up to the error control facility in the transport layer.

An extension to the basic MACA scheme was proposed by Bharghavan et al. [Bharghavan, Demers, Shenker and Zhang 1994] called MACA for wireless (MACAW). MACAW adds acknowledgement frames to the protocol giving it the ability to recover from errors faster than MACA. Other variations of MACA include MACA for underwater acoustic networks with packet train for multiple neighbors (MACA-MN) [Chirdchoo, Soh and Chua 2008], MACA by invitation (MACA-BI) [Talucci and Gerla 1997], and the Floor Acquisition Multiple Access protocol [Garcia-Luna-Aceves and Fullmer 1999].

- **Floor Acquisition Multiple Access (FAMA):**

  The Floor Acquisition Multiple Access (FAMA) protocol proposed by Garcia-Luna-Aceves et al. is a MACA based scheme that dictates that every transmitting node must acquire explicit control of the channel prior to transmitting [Garcia-Luna-Aceves and Fullmer 1999]. This protocol differs from the MACA and MACAW schemes since it requires that both the sender and the receiver take an active role in the collision avoidance process. To "acquire the floor", as the authors put it, the sender sends out a RTS frame either by the FAMA non-persistent packet sensing (FAMA-NPS) or the FAMA non-persistent carrier sensing (FAMA-NCS) scheme. The receiver will reply with a CTS containing the address of the initiating station. Any other station that receives an error free CTS frame will know that the terminal addressed in the CTS frame has reserved the channel. This is the floor acquisition aspect of FAMA. To further ensure that the channel has been reserved, the CTS are repeated enough times in order to jam any hidden station who did not hear the original RTS acknowledgment.

- **Multiple Access Collision Avoidance by invitation (MACA-BI):**

  The MACA by invitation (MACA-BI) proposed by Talucci and Gerla is a receiver initiated based protocol [Talucci and Gerla 1997]. In sender-initiated protocols the sender will attempt to gain access to the channel by initiating the RTS-CTS handshake. However, MACA-BI requires that the receiver request the data from the sender by using a ready-to-receive (RTR) frame. This reduces the overhead in the exchange by making it an RTR-DATA versus an RTS-CTS-DATA process. This protocol appears to be meant for service networks where the communications are one way, that is, where the receiver has information it knows the other party has, it will send out a RTR that will be followed by the data.

- **Collision-free Receiver Oriented MAC (CROMA):**

  Collision-free Receiver Oriented MAC (CROMA) [Coupechoux, Baynat, Bonnet and Kumar 2005] is a receiver initiated MAC protocol similar to MACA-BI. CROMA divides time into frames, where each frame is further divided into a fixed number N time-slots. Each slot is broken into three sub-slots: request

(REQ), ready-to-receiver (RTR), and a data (DATA) slot. The REQ slot is used by nodes to send a REQ frame to a receiving node. The RTR slot is used to acknowledge the REQ frames sent and to poll the nodes that previously sent a successful request and reservation. A DATA frame is then sent once the sender in the RTR slot has been successfully polled. The reservation of the channel is achieved through the polling frames sent in the RTR slot, and this is what makes CROMA a receiver oriented protocol versus a sending oriented one. CROMA differs from MACA-BI since it does not need to use a traffic prediction algorithm. The division of these slots makes CROMA a collision-free contention-based protocol.

- **IEEE 802.11 MAC:**

  The IEEE 802.11 standard defines two MAC sub-layers: the point coordination function (PCF) and the distributed coordination function (DCF). The DCF mode of operations was meant for ad hoc networks whereas the PCF mode was meant for infrastructure-supported networks. The DCF function uses CSMA/CA with acknowledgment and RTS/CTS frames (RTS-CTS-DATA-ACK). The protocol operates much in the same way as described in the CSMA and CSMA/CA description with several differences [Forouzan 2007, Agrawal and Zeng 2006].

  1. Sending node first senses the channel by monitoring the energy level on the carrier frequency.

    (a) If found to be busy a persistent strategy with a back-off timer is used until the channel is idle.

    (b) Once the channel is found to be idle, the station must wait for a period time called the distributed interframe space (DIFS). After this duration the sender transmits a RTS frame.

2. Upon receiving the RTS frame, the receiver must wait for a period of time called the short interframe space (SIFS) to pass prior to replying with a CTS frame.

3. When the sender successfully receives the CTS frame it must wait for the SIFS to pass prior to sending a data frame.

4. Upon successful reception of a data frame, the receiver must wait for the SIFS to pass prior to returning an ACK frame.

This protocol makes use of the SIFS to further reduce the chance of collisions. An additional important addition of the DCF is the network allocation vector (NAV). When a sending station transmits a RTS it includes the duration of time it expect to occupy the channel. Any stations that receive the RTS or replying CTS use this time to create a timer that determines how much time must pass before the station can sense the channel for idleness. Anytime a node wishes to check the channel for idleness it must first check to see whether the NAV timer has expired. This is similar to the protocol used by MACA where the size of the data is sent along with the RTS/CTS frames. The DCF is the most widely used protocol for wireless local area networks (LANs). It has its roots in the previously explained protocols, and many of the protocols [Fang, Bensaou and Yuan 2004, You, Yeh and Hassanein 2003, Wang and Zhuang 2008, Lau and Chan 2006] found in literature use it as a base of reference.

- **Dual-Channel MAC (DUCHA):**

  Zhai et al. propose the Dual-Channel MAC (DUCHA) scheme that uses two distinct channels to overcome the receiver blocking problem and the hidden and exposed terminal problems [Zhai, Wang and Fang 2006, Zhai, Wang, Fang and Wu 2004]. The receiver blocking problem is a specialized case of the exposed terminal problem where a receiver cannot respond to incoming RTS intended for itself due to the transmissions occurring in its sensing range. DUCHA separates the channels into one for control and the other for data. It also uses a busy tone, much like busy tone multiple access (BTMA) [Wu and Li 1988] and dual busy tone multiple access (DBMTA) [Haas and Deng 2002], to establish channel control to overcome the hidden terminal problem. The blocking receiver problem is solved through negative CTS (NCTS) frames. The authors don't use ACK frames since they claim collisions in the data channel are guaranteed to not occur. However, they do use a NACK busy tone from the receiver that will be used if the receiver thinks it has received corrupted data. The message exchange proceeds as follows:

1. *RTS:* The sender follows the rules employed by 802.11 with regard to the use of the SIFS and DIFS wait times prior to sending a message. Any node must sense the control channel to be free from a signal or busy tone for a period equal to DIFS prior to sending. If the channel is found to be busy it waits for a period of time to pass prior to sending its frame.

2. *CTS/NCTS:* In DUCHA any node that overhears a RTS responds with a CTS frame after waiting a period equal to the SIFS regardless if the control channel is busy if the data channel is idle. If both are busy, it will ignore the RTS to avoid interfering with the reception of CTS frames at the sender. NCTS frames are returned when the control channel has been found to be idle for at least one CTS frame length long, and the data channel is busy. The NCTS also provides the sender an estimation for how much longer the data channel will be busy .

3. *DATA:* Once the sender receives a CTS it should begin to send the data if no busy tone signal is present. If it receives a NCTS, it will defer transmitting for the estimated time included in the NCTS frame. If neither is received, it assumes a collision has occurred on the control channel and uses a back off strategy accordingly.

4. *Busy Tone:* The receiver will begin to sense the channel data channel prior to sending the CTA frame to listen for the data from the sender. If it doesn't begin to receive the first bits of the frame in due time (determined

by the information in the RTS) it will assume the sender couldn't transmit. Otherwise, once the receiver begins to receive data it will transmit a busy tone signal on the control channel to prevent hidden terminals from transmitting.

5. *NACK:* The NACK is used by the receiver to notify the sender of a problem receiving the data. The receiver uses a timer to determine how long it should take for the data frames to finish sending. If the timer expires and the receiver hasn't collected the correct data packet, it assumes a problem has occurred and will extend the busy tone signal for a period past the timer expiration. If it successfully received the packet it discontinues the busy tone signal. The sender assumes that if it doesn't hear the NACK busy tone during the NACK period that the transmission succeeded, otherwise if it sees the signal it will begin its retransmission procedure.

- **Multi Channel CSMA MAC:**

  A multi-channel CSMA protocol was proposed by Nasipuri et al. [Nasipuri, Zhuang and Das 1999] where the total bandwidth of the channel was divided into N distinct channels. The channels can be divided either through CDMA or FDMA. The protocol follows the basic principles described in the section of CSMA. When a station wishes to send, it first senses the last channel it used to determine whether it is available. If the channel is not free a new one is chosen at random, and if no free channel is located it uses a back off protocol to retry later. The author's later extended the protocol in [Nasipuri and Das 2000] where the optimal channel is chosen based on the power of the signal observed at the sender side. It was further supplemented in [Jain, Das and Nasipuri 2001] to add an additional control channel to the N divided data channels. This channel is used to exchange control frames that allow the sender to determine the best channel to send the data on. The optimal channel is chosen based on the signal-to-noise ratio (SNR) observed at the receiver.

- **Hop-Reservation Multiple Access (HRMA):**

  Hop-reservation multiple access (HRMA) [Yang and Garcia-Luna-Aceves 1999, Tang and Garcia-Luna-Aceves 1998] is a multi-channel protocol for radios using the FHSS spreading technique described in Section 2.3. Previous work has been done with frequency hopping radios [Pursley 1987, Ephremides, Wieselthier and Baker 1987] to use CDMA in an effective way that required the radios to switch frequencies part way through data packets. HRMA uses very-slow FHSS in order to take advantage of the time-slotting properties that allow an entire frame to be sent in the same hop. HRMA does not do any carrier sensing prior to transmission, and employs the use of control frames in order for a pair of communicating nodes to reserve a hopping sequence (channel). HRMA requires synchronization where one the N available frequencies is dedicated to synchronization. The remaining frequencies are further divided into $\lfloor \frac{N-1}{2} \rfloor$ pairs, where the first frequency is used for the hop reservation (HR), CTS, RTS, and data frames, and the second frequency is reserved exclusively for ACK frames. This protocol allows for collision free communications even in the presence of hidden terminals [Yang and Garcia-Luna-Aceves 1999].

- **Multi-Channel Medium Access Control (MMAC):**

  Multi-channel MAC (MMAC) is a protocol meant to extend the functionality of the DCF in IEEE 802.11 by allowing it to dynamically switch between the 11 available channels [So and Vaidya 2003]. Although 802.11 has the support for these multiple channels it can only utilize one channel at a time. This is for backwards compatibility since hosts with a half-duplex radio can either be in receiving or transmit modes. The protocol divides time into multiple fixed-time beacon intervals. At the beginning of each of the intervals is an ad-hoc traffic indication message (ATIM) window in which ATIM frames are exchanged between communicating nodes so as to coordinate channel assignments. This protocol is efficient since it doesn't require that the nodes have multiple radio transceivers as is the case for other multi-channel protocols [Jain, Das and Nasipuri 2001, Wu, Lin, Tseng and Sheu 2000, Tseng, Wu, Lin and Sheu 2001]. The protocol does however require that at the beginning of these ATIM windows, or beacon intervals, every node must synchronize itself with all other nodes on a synchronization channel in which these ATIM frames are exchanged. Additionally, each node maintains a preferred channel list (PCL) that keeps track of the channels for prioritization. The authors validated the protocol through simulations, and their results demonstrated that MMAC outperformed IEEE 802.11 with regards to throughput.

- **A Jamming-Resistant MAC Protocol for Single-Hop Wireless Networks:**

  Awerbuch et al. propose a MAC protocol for maintaining link capacity in the presence of adaptive adversarial jamming attacks [Awerbuch, Richa and Scheideler 2008]. The authors assume that all nodes are synchronized in time steps, and that an adversary can only jam a $(1-\epsilon)$-fraction of the time steps for some constant $\epsilon > 0$, and that it must make a decision to jam that time step prior to knowing the actions of other nodes at the current time step. As is expected, the nodes on the network are unable to distinguish between adversarial jamming and whether other nodes on the network are simply using the channel. The nodes then use mathematical probabilities in order to determine when they are able to transmit. The nodes keep track of the overall time in which the channel is idle and when exactly one successful transmission occurs. It then uses this information to adjust the probability of a time step in which the transmission can occur. The nodes however, do not consider the time steps in which their transmissions have been blocked making the decision algorithm robust to jamming attacks. The algorithm attempts to adjust the probabilities such that the number of time steps that the channel is found idle is equal to the number of time steps in which exactly one message transmitted. If this is not the case, than the probabilities are adapted to make this true. The authors claim that this protocol is robust to adaptive jamming attacks and is energy efficient. However, the paper does not include any simulation results or data from a physical implementation. Furthermore, the authors assume that the adversary is limited by the number of time steps that they can jam, and is limited to "bursty jamming". Another interesting problem is the protocol relies upon the knowledge of when a successful transmission occurred, and it assumes it will know when this is true.

- **Advanced MAC (aMAC):**

  Lau and Chan propose a new protocol call advanced MAC (aMAC) [Lau and Chan 2006] that is based off a previous protocol called the Fair MAC with Cooperation between Sender and Receiver (FMAC/CSR) [Li, Gupta and Nandi n.d.]. The goal of FMAC/CSR is to maintain fairness between contending flow for single-hop flows. However, Lau and Chan show that when it is extended to multi-hop flows the fairness breaks down. This is attributed to the use of the 802.11 binary exponential back off (BEB) algorithm that is used for contention resolution which has been shown to be unfair [Li, Nandi and Gupta 2006, Kloul and Valois 2005, Razafindralambo and Valois 2006]. aMAC aims to resolve the unfairness issues in FMAC/CSR by replacing BEB with the exponential increase exponential decrease (EIED) back off algorithm proposed by Song et al. [Song, Kwak, Song and Miller 2003]. The protocol follows four steps: channel estimation, unfairness detection, sender contention, and the EIED algorithm. Channel estimation monitors the channel to estimate flow's fair share and actual share. Unfairness detection compares the actual share to the fair share to determine how much the actual shared has deviated from the fare share. The sender contention determines the state of a MAC flow (aggressive, normal, or restrictive). Finally, the EIED algorithm is used to govern the contention window. By integrating EIED the authors state that preliminary results show that aMAC maintains superior medium fairness when compared to similar fairness oriented schemes.

- **Real-time MAC (RT-MAC):**

  Real-time MAC (RT-MAC) is a quality of service (QoS) oriented scheme proposed by Baldwin et al. that is a variation of the IEEE 802.11 protocol [Baldwin, Nathaniel J. Davis and Midkiff 1999]. When IEEE 802.11 is used with real-time traffic constraints two issues impact the efficiency of the network: expired deadlines and collisions. Since IEEE 802.11 has no method of determining whether a frame has exceeded its deadline it will continue to re-transmit these frames, even though they are no longer useful to the receiver. These collisions and re-transmits waste resources needed by other frames to meet their deadlines. RT-MAC remedies this by avoiding the transmission of expired frames. This is achieved by adding transmission deadlines to the packets received from the network layer and by using an enhanced collision avoidance mechanism. Whenever a packet is marked as real-time it is marked with a time stamp from the originating station indicating when the packet should be transmitted. The check for an expired packet occurs at several points: when the back off timer expires, prior to sending, and upon the expiration of the timer for the acknowledgment frame. If at any of these points the frame missed the deadline, it is dropped from the transmission queue. RT-MAC has a unique method for improving the collision avoidance mechanism of 802.11. Prior to sending the frame the sending nodes chooses the next back off counter value and records it in this frames header. Any station that overhears this frame being sent out will see this back off value and chose one such that it is different. This further eliminates the possibility of collisions.

- **Controlled Access CDMA (CA-CDMA):**

  The authors [Muqattash and Krunz 2003] present the Controlled Access CDMA (CA-CDMA) multi-channel protocol which is based on CDMA. It was mentioned in the prior section that CDMA is a contention free algorithm, however, the authors of CA-CDMA make the statement their modification is actually a contention-based scheme. Much like CSMA/CA, CA-CDMA makes use of the control RTS and CTS packets as a channel reservation mechanism. These control packets are transmitted on a control channel separate from the data channel, at fixed power. Just like IEEE 802.11 every node receives these packets, however, nodes may continue to transmit if they meet certain criteria determined by the interference margin algorithm presented in [Muqattash and Krunz 2003]. The nodes use the power levels of the received CTS and RTS packets to determine the power that the node can transmit at without interfering with other transmissions.

3.4 Chapter Conclusion

This chapter reviewed the protocols necessary to understanding the current state of MAC layers for ad-hoc wireless networks. The protocols vary from those which rely on contention free schemes, to those concerned about quality of service (QoS), and to the protocols that use multiple channels for either doing control or for multiple data paths. In relation to the jam-resistant goals of this research only one such protocol was found which directly concerned itself with adversarial jamming of the wireless channel. Awerbuch et al. [Awerbuch, Richa and Scheideler 2008] proposed the Jamming-Resistant MAC Protocol for this purpose, but only handled jamming by attempting to send when it predicted no adversarial jamming was occuring. Many of the other protocols address the hidden and exposed terminal problems through the use of control frames, and multiple channels. Protocols such as CSMA/CA, MACAW, DUCHA, and FAMA provide many different approaches to solving the hidden and exposed terminal problems, and will be considered for the design of BBC-MAC.

Chapter 4

BBC-MAC Initial Protocol Design

4.1 Chapter Introduction

The medium access control layer for noisy channels will build upon the current state of ad hoc wireless communications by creating a reliable data link through the use of the BBC algorithm and its error-correcting properties. Combining BBC with the traditional facilities of the data link layer will transform the raw data transmission facility provided by the software-defined radios into a jam-resistant communications link for mobile ad-hoc networks. This layer will work as a single-hop protocol that will only be concerned with delivery of data to the next terminal, and will not consider multi-path routes for node-to-node delivery. It will be left to the network layer to determine how to properly route the data.

4.2 Protocol Requirements

To achieve a reliable data link the new layer must address the issues of framing, addressing, flow and error control, and a primary focus on controlling the channel. A final requirement for the algorithm to operate properly is the ability to dynamically adjust the coding properties of the BBC algorithm to adjust to the level of noise. By adjusting the coding properties that determine the level of jam-resistance, the layer can sustain link communications up to a certain level of noise.

- **Framing:** Physical limitations of the software defined radios and coding parameters of the BBC algorithm will require that larger packets received from the network layer be broken into smaller frames for encoding and transmission. On the receiving end these will have to be re-combined for proper delivery to the upper layers. The size of the frames can be a fixed or dynamic size, but since the size of the frames could be in direct relation to the message length that is encoded by the BBC algorithm, it will have to be a dynamic size. While this was initially determined a requirement, I realized that the BBC algorithm already does framing of data, and for the prototype implementation I considered any data the upper layer passed to be a single data frame.

- **Addressing:** Every node in the transmission range must have a unique identifier so that a node receiving a message knows whether they are to be discarded or when the message was meant for it . As mentioned previously, the routing will be considered a job of the network layer, and it assumed the network layer will have a unique identifier for this purpose. This same identifier will be used at the data link layer for addressing. Considering the requirement of the routing, it is anticipated at this time that the data link layer will not keep state information pertaining to nodes in its transmission range. However, this might prove to be useful during research and will not be taken away from consideration. The final protocol prototype assumes that it will be given the address of the node upon initialization.

- **Flow and Error Control:** An important property for creating a reliable link is in the flow and error control algorithms. Flow control must alleviate congestion of the link by limiting the amount of data it sends, and error control must be able to recover from frames that the BBC algorithm's error correcting facility could not handle. It is anticipated a modified version of the selective repeat automatic repeat request algorithm described in Section 3.2 will be used for this purpose. The final implementation does error control on the single data frame it sends by using an acknowledgment frame. Future work would be directed at doing error control on the individual BBC codec frames as well.

- **Access Control:** Controlling access to the channel is the most important aspect of this protocol. While the BBC algorithm allows for communications to continue even in the presence of interference, avoiding channel saturation is important. If nodes were left to freely transmit whenever they chose, the level of noise (jamming/collisions) on the channel would continue to grow to the point where the error correcting aspect of BBC could not overcome the problem. To overcome the hidden and exposed terminal problems described in Section 2.4, techniques inspired by the protocols discussed in Section 3.3 will be used. It is anticipated that the protocol will not rely on carrier sensing much like MACA and MACAW, but will incorporate control frames to reserve channel access. The control frames will carry several important pieces of information. The first is the the size of the data that is going to be transferred in the DATA frame. This will allow any node that overhears this transmission to know how long it should defer its transmissions. The second parameter included in both frames will be the received signal strength indicator (RSSI) value. Upon sending a RTS frame, the sender will include its most recent RSSI, and the receiver will similarly reply with its most recent RSSI value. This is used to prepare the nodes for the proper level of jam-resistance. The final protocol implementation does use the control frames to reserve the channel for the two communicating nodes for a limited amount of time.

- **Dynamic BBC:** Maintaining the link will rely upon the BBC algorithm to overcome noise in the channel. However, the level of noise is likely to be dynamic in relation to the number of nodes active in the network, and the level of determination by an adversarial jammer. For this reason the layer must be able to adjust the coding parameters of BBC to allow for dynamic jam-resistance. The specific properties at this time that changes the level of jam-resistance are the hash function and the expansion size of the original data. These two properties determine how large the BBC packet is and where the indelible marks can be placed. The RSSI value included RTS/CTS frames will be used to determine at which level of jam-resistance the two nodes will communicate. The node with the highest RSSI value will be the determining level that the remaining DATA-ACK communications occur at. This service will also be available for upper layers to adjust. This is a requirement for allowing varying levels of priority from upper layer packets. The final prototype achieves dynamic BBC by altering the packet expansion in the BBC codec, and adjusting this value based upon the Received Signal Strength Indicator (RSSI) value contained in the control frames.

4.3 Chapter Conclusion

The Single-Hop Medium Access Control Layer for Noisy Channels protocol has been conceived to address the many problems that currently affect the current state of medium access control for MANETs. The protocol will be designed to maintain a reliable communications link in the presence of noise, via either intentional or unintentional jamming, and will dynamically adjust either by the layers own mechanism or as dictated by upper layers. The protocol aims to provide data transfer reliability by developing a new layer built upon the foundation of the BBC algorithm. By focusing on maintaining a communications link in the presence of jamming, it is expected that the protocol will be able to overcome obstacles such as adversarial attacks, pre-shared secrets, and the hidden and exposed terminal problems that other protocols fail to.

Chapter 5

Protocol Design and Implementation Phase

5.1 Chapter Introduction

The previous chapter discussed the initial protocol design and reviewed some of the basic requirements and rudimentary methods for achieving the goals of the protocol. This chapter covers the end design for the protocol, and how it is implemented. The protocol requires the use of many different software and hardware components. Implementation and testing for so many pieces becomes more difficult as the complexity of the layer increases. Initially, many of the pieces were built simultaneously and then an attempt at testing was made. However, later development required that new pieces be tested individually in order to reduce the new number of variables which needed to be accounted for when the component failed.

As previously noted, one of the goals of this research is to implement and validate the protocol on physical hardware. However, the MAC layer requires that a physical layer exist prior to any implementation on it occuring. The creators of the BBC [Baird, Bahn, Collins, Carlisle and Butler 2007] algorithm had created a basic physical layer implementation for the purposes of research. Their implementation takes a file as an input, encodes it using the BBC algorithm, and then modulates it for proper transmission with the Software Defined Radios (SDRs). The modulated data is then transmitted with a python script that repeats until user-terminated. A similar series of steps occurs on the receiver's end. A python script receives data from the USRP until user terminated. Then the demodulator is run on the received data and the decoder. If a successful transmission occurred, the same file sent should be in the receivers folder. The code base from this prototype was used as the starting point for the creation of the new upper-layer MAC protocol.

The remainder of this chapter begins with a breakdown of the system components used for the creation of this new layer, and then follows with a detailed look at the various operations at the physical layer and those which occur in the BBC-MAC layer.

5.2 System Components

The final implementation presented here relies upon many software and hardware components to create the end prototype. This section covers the different components in order to familiarize the reader with the equipment used. Certain components have a direct relation to the way the protocol was designed, specifically, components like the Universal Software Radio Peripheral (USRP) and the type of daughterboard used have an impact on the hardware abilities of the layer.

5.2.1 Hardware Components

- **Universal Software Radio Peripheral (USRP):**

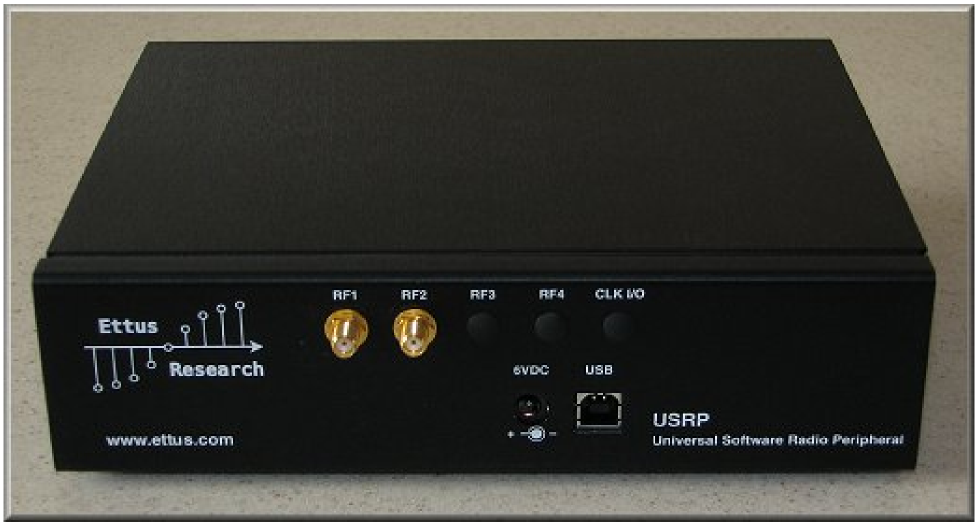

Figure 5.1: Universal Software Radio Peripheral External View

The Universal Software Radio Peripheral (USRP) is the main hardware component used for developing Software Defined Radios (SDRs). Figure 5.1 shows the external casing of this component. The USRP1, developed by Ettus Research, LLC, pictured in Figure 5.2, was used for the development and testing during this research. The USRP1 contains an Altera Cyclone Field Programmable Gate Array (FPGA), and has four extension sockets that support up to four daughterboards. The FPGA drives four high-speed 12-bit analog-to-digital converters (ADC) capable of 64 Mega-Samples/second and four high-speed 14-bit digital-to-analog (DAC) converters capable of 128 Mega-Samples/second. The

ADCs are used during the receive chain, and the effective sampling rate is determined by the decimation rate. Likewise, the DACs are used during the transmit chain and the effective sampling rate is determined by the interpolation rate. The USRP connects to the external computer through a Cypress EZ-USB FX2 High-speed USB 2.0 controller that allows for speeds approaching 32 Mbytes/s. USB 2.0 specification allows for up to 480 Mbit/s or 60 Mbytes/sec, but the current FPGA used doesn't support the full bandwidth of USB 2.0. This is because the Cypress USB controller uses the bulk transfer mode of USB 2.0, which is limited to roughly 32 Mbytes/s.

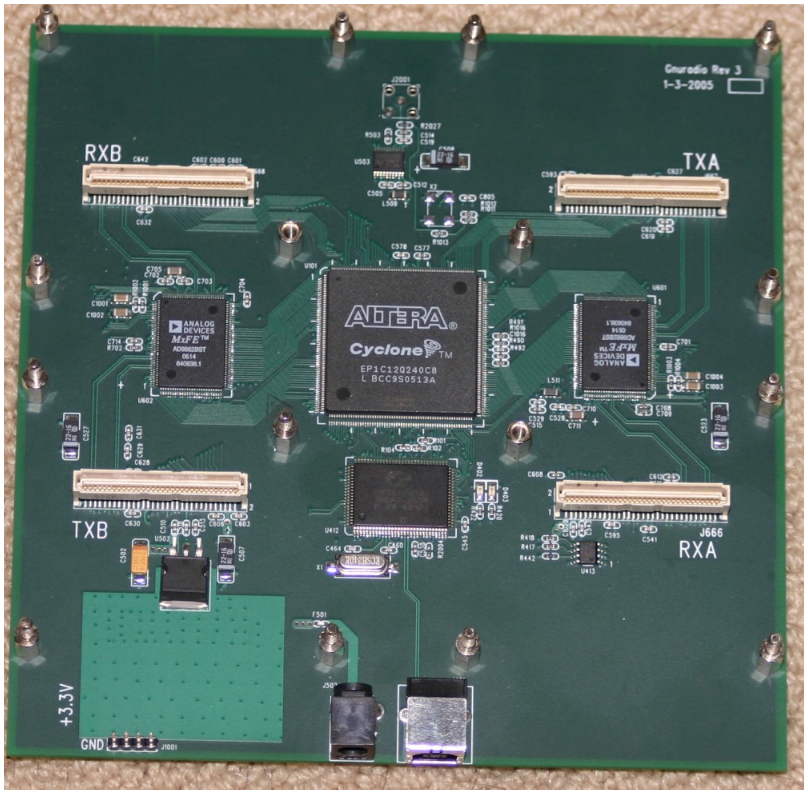

Figure 5.2: Universal Software Radio Peripheral Internal Hardware

- **RFX-1200 Daughterboard:**

  The USRP has the support for two transmit sockets and two receive sockets, allowing for up to two receive daughterboards and two transmit daughterboards, or two transceiver daughterboards. The daughterboard used during this research is the RFX-1200 transceiver, pictured in Figure 5.3, that operates in the 1150-1450MHz frequency range with a transmit power of 200+mW (23dBm). The board supports both transmitting and receiving on the same connector, but also supports an auxiliary receive port which allows transmit and receive to occur on separate frequencies. The board has a 30 MHz bandwidth and 70 dB Automatic Gain Control (AGC) range with adjustable transmit power. The final useful feature which is crucial to this research is the built-in analog Received Signal Strength Indicator (RSSI) measurement from an auxiliary ADC. This research uses two RFX-1200s in each USRP.

- **VERT400 Vertical Omnidirectional Antenna:**

  The final component for the radios is the VERT400 omnidirectional antenna pictured in Figure 5.4. This is a seven inch tri-band antenna operating at the 144Mhz, 400Mhz, and 1200Mhz frequencies. Each USRP has two daughterboards and so there are two of these antennas on each USRP.

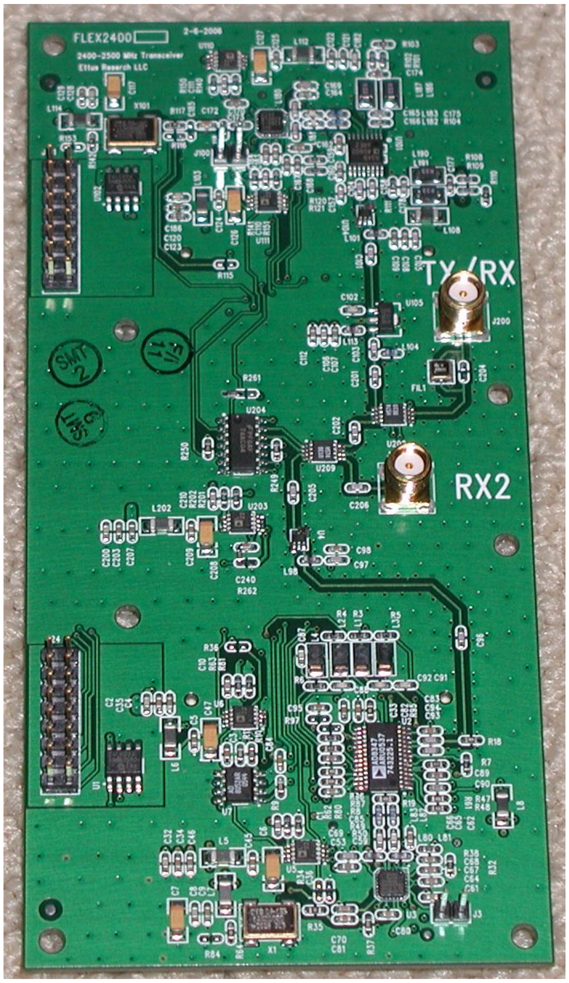

Figure 5.3: RFX-1200 Transceiver Daughterboard

- **Laptop Computer:**

  The final hardware component is the computer used to drive the USRPs. In this research, an Apple MacBook Pro running an Intel Core 2 Duo operating at 2.53 GHz with 4GB of DDR3 system memory was used. The computer has an impact in several stages of this research. The first is at the USB interface where this laptop has two USB 2.0 ports. The second is during the decoding stage of

Figure 5.4: VERT400 Antenna

received data. The speed of the processor can have a significant impact on how fast this stage occurs. Furthermore, the amount of system memory available impacts how many samples from the received sink file the computer can load into memory at a single time for decoding.

5.2.2 Software Components

The software components involved in this system are limited to C and Python. The BBC program and the jammer for this research were both written in C. The remainder of the software components were all written in Python due to its ease of development and because the GNU Radio API for USRP interaction is written in Python.

- **GNU Radio Software Library:**

  GNU Radio is a free software toolkit created to give users the ability to learn and create wireless protocols. The GNU Radio Library contains all the necessary runtime and processing blocks to interact with the USRP. The client library is largely written in Python with the signal processing blocks developed in C++.

- **BBC Encoder:**

  The BBC software is written in C and can be found at the site maintained by William Bahn [Bahn March 2009]. This software performs the BBC encoding and also takes on the physical layer task of modulating data into the proper format for the radio transmit script for transmission. The algorithmic details of this software were discussed in Section 2.5.

  When data is being encoded the following steps occur:

  1. The data to be encoded is dumped to a file specified by the configuration file, and loaded into memory.
  2. The data is then split up into BBC Frames with the following format:
     - **StreamID:** 16-bit integer that identifies the data stream.
     - **Checksum:** 32-bit checksum value for the payload.
     - **Sequence Number:** 16-bit integer indicating which sequence number in the stream this frame is.
     - **Data Bits:** 16-bit integer indicating that actual number of bits contained in the payload
     - **Data:** The payload for this frame. Size is determined by the configuration file.

3. Each frame is then sent to the BBC Encoder where the frame is encoded and placed into the packet buffer.
4. The contents of the buffer are then modulated into the proper format and placed in a sink file for transmission.

The software was largely left untouched with the exception of a modification for the configuration file to accept absolute paths to the source and sink files. The source is compiled into a binary executable which is called upon in the interface.

- **BBC Decoder:**

  The BBC decoder is part of the same piece of software as the encoder and was written by William Bahn [Bahn March 2009]. The BBC decoder performs similarly to the encoder, but in reverse. When there is data available for processing the following steps occur:

  1. The data that has been received from the radios is loaded into memory and sent to the demodulator, where the received data is transformed into bytes and placed in a buffer for processing.
  2. Each buffer read location is then sent to the decoder where it attempts to decode valid messages. Those that it does find are sent to the sink module.
  3. The sink module collects the messages belonging to the same streamid and places them in order for output to the sink file.

4. At the end of the execution the sink module purges its contents to the sink file.

This part of the code has also been left mostly unaltered, with one exception. If, during the purge of the sink, it is discovered that there are missing sequence numbers, the sink will not dump the contents to the file. BBC-MAC is not doing frame control on the BBC frames at this time, it is unnecessary to output to the file if parts of the transmission are missing. From the point of view of the layer it is just considered a failed transmit or receive. The final source code for the BBC software used during this research is listed in Section A.3.

- **Python USRP Receiver Script:**

  In order for the layer to interact with the radios, a receiver script is necessary. This is a modified version of the example usrp_rx_cfile.py script that comes with the GNU Radio library. The source code can be found in Section A.2.1. The script accepts several important parameters:

  - **freq:** This is the frequency that the radio should be tuned to.
  - **nsamples:** This is used to tell the receiver it should collect *nsamples* samples and then exit. This parameter is used to limit the size of the file the BBC decoder can load into memory.
  - **decim:** This parameter is the decimation rate of the FPGA. The FPGA can receive at 64 Mega-Samples/second. If the decimation rate is set to 128 then the effective sampling rate is $\frac{64e6}{128} = 500000$ Samples/sec.

Beyond initiating the communications with the radios and collecting the samples, this script also performs the important task of collecting the Received Signal Strength Indicator (RSSI) value. Upon starting the receiver script a separate thread of execution is initiated that continuously calls the auxiliary ADC and asks for the RSSI measurement. The thread averages the last 1141 calls and outputs the highest average from the last twenty averages to a file located in the folder for the specified radio. These operations can be found in Listing 5.1. The number of reads to the ADC in a single second is roughly 1141, and by only returning the highest average in the last twenty seconds we are able to give the layer a better idea of what level of jamming has recently occured. This is a better safe than sorry approach to the configuration of the jam-resistance. It allows the communicating nodes to configure their jam-resitance levels to an appropriate level of jamming which has recently been measured, and could possibly occur again in the middle of the transmission.

```
def GetRSSI(self, d, t):
    reads = []
    avgs = []
    while self.rssi_run:
        tmp = self.u.read_aux_adc(self.rx_subdev[0],0)
        reads.append(tmp)
        self.rssi = sum(reads[-1140:])/1140
        avgs.append(self.rssi)
        file = open(receive.fn+”ssi”, ”wt”)
        file.write(str(max(avgs[-20:])))
        file.close()
```

Listing 5.1: usrp_rx_cfile.py lines 190-200

- **Python USRP Transmitter Script:**

  The transmitter script handles the duties of transmitting the encoded data with the USRPs. The source for this file can be seen in Section A.2.2. It is a modified version of the bbc_tx.py script included on the BBC Real-time Research Engine website. The important parameters for this script are:

  - **rf_freq:** This is the frequency that the radio should be tuned to.
  - **interp:** This parameter is the interpolation rate of the FPGA. The FPGA can transmit at 128 Mega-Samples/second. If the interpolation rate is set to 256 then the effective sampling rate is $\frac{128e6}{256} = 500000$ Samples/sec.
  - **jammer:** This parameter tells the script how to create the transmission flow graph. 0 = no jammer, 1 = pulse jammer, 2 = Gaussian jammer.
  - **jammer_level:** This parameter indicates the jamming level to be used if a jammer type is specified.
  - **tx_time:** This parameter tells the flow graph how long it should transmit the data. This time is determined by the BBC-MAC layer based upon the size of data and the configuration options for the encoder.

  The modifications were made so that it was possible to transmit on both daughterboards simultaneously. This is required since whenever a valid transmission is occurring there are three possible configurations:

  1. The encoded data is being transmitted on one of the daughterboards only.

2. The encoded data is being transmitted on one daughterboard and simultaneously the pulse jammer is being transmitted on the other.
3. The encoded data is being transmitted on one daughterboard and the Gaussian jammer is being transmitted on the other.

- **Pulse Jammer:**

  In order to test the error-correcting ability of the BBC algorithm I created a novel jammer that would send out data at the same symbol rate and modulation scheme as is used by the BBC executable. The main program is a modified version of the BBC source code with only the necessary components remaining. The source code for this jammer can be found in Section A.4. It accepts as parameters the jammer level to be used and the number of samples to create. The jammer level is a value in the range of [0,64]. The level indicates that for every 64 time steps, that *level* time steps should contain a high pulse. The relevant code for this is seen in Listing 5.2. For example, if the jamming level was 13, the program would randomly select 13 locations to set the bit to high in an 64-bit variable. The program also guarantees that there will be 13 locations by ensuring that each new value was not already chosen. The program then pushes the four bytes of that variable onto the buffer and send it to the modulator. This is repeated for as many samples as are needed.

```
for(i = 0;i<(samples/(32*sizeof(unsigned long long)));i++){
  *buf_number = 0;
  *ran_number = 0;
  for(j=0;j < jammer_level;j++){
    *ran_number = rand()%(8*sizeof(unsigned long long));
    while (marked[*ran_number]==1){
      *ran_number =  rand()%(8*sizeof(unsigned long long));
    }
    marked[*ran_number] = 1;
    //set the bit at ran_number to 1
    *buf_number |= (1 << *ran_number);
  }
  memcpy(buffer->buffer+buffer->write, buf_number, sizeof(unsigned long long));
  buffer->write+=sizeof(unsigned long long);

  for(j=0;j<sizeof(unsigned long long);j++){
    buffer->ready = 1;
    Modulate(config, buffer, modem, sink);
  }
  memset(marked,0x00, sizeof(unsigned long long)*8);
}
```

Listing 5.2: jammer.c lines 71-91

The goal of the jammer is to create an attack on the protocol by transmitting random data in the same fashion as the valid encoded data. It aims to test the limits of the error correction of the BBC algorithm and demonstrate an actual interference in a similar fashion as the toy example with noise did in Section 2.5.4. To illustrate how this will affect the data being transmitted, Figure 5.5 shows what BBC encoded transmission looks like on a software oscilloscope without interference, and Figure 5.6 shows how it looks once we run this jammer at level 20 jamming. While these images are not taken at the same time in the transmission, it is clear that in Figure 5.5 the pulses are fairly distinct with proper spacing, but in Figure 5.6 we see some interference and a significantly higher density.

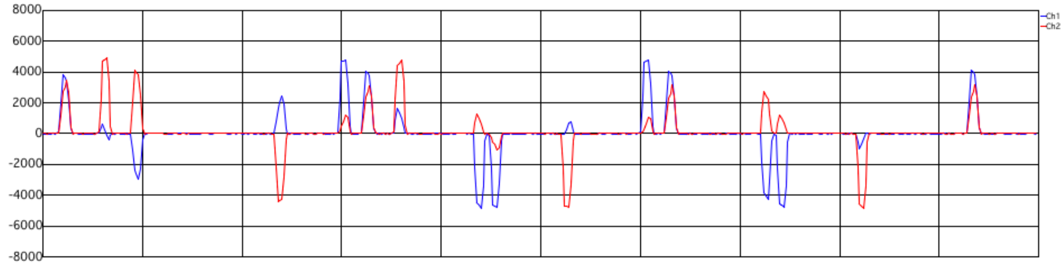

Figure 5.5: BBC Encoded Transmission without Noise

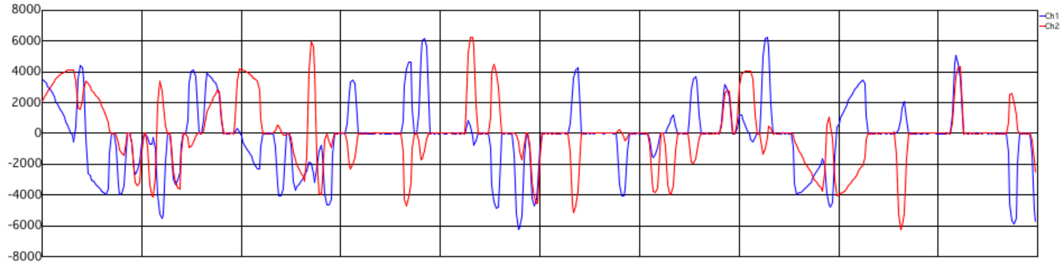

Figure 5.6: BBC Encoded Transmission with Pulse Jammer Noise

- **Gaussian Jammer:**

The other jammer used during testing is a Gaussian noise source generator that is part of the GNU Radio library. The generator asks for an amplitude as a parameter which determines the max amplitude of the signal containing the noise. Again, in the script the jammer level is in the range of [0,64] where the level is multiplied by 500 to determine the max amplitude to pass to the Gaussian noise generator.

5.3 Physical Layer Implementation

This section will cover in greater capacity the operations that are carried out at the physical layer to allow for the BBC-MAC layer to function properly. The physical layer is responsible for the physical transmission of the data that the upper layer passes to it. These responsibilities include the communication with the USRPs, BBC encoding and decoding, and modulation of data for proper transmission. This protocol assumes that all operations with the BBC algorithm are considered part of the physical layer, and the upper layers simply control the operations through configuration adjustments as necessary.

The task of communicating with the USRPs is handled by the radio scripts discussed in Section 5.2.2. The scripts are then controlled by the interface class of the BBC-MAC software which handles the control of the physical interface. This class can be found in Section A.1.1. It also handles the execution of the BBC executable for the encoding and decoding of data. The BBC executable handles the complex task of encoding and modulation and the reverse task of demodulation and decoding as discussed in Section 5.2.2. Figure 5.7 shows a high-level depiction of the operations at the physical layer.

Whenever the radio is in receiver mode, the decoder is being continually ran on the data that has been received. The transition from receiver to idle can occur as the result of the following situations:

1. When the interface has been told to transmit data by the BBC-MAC layer.

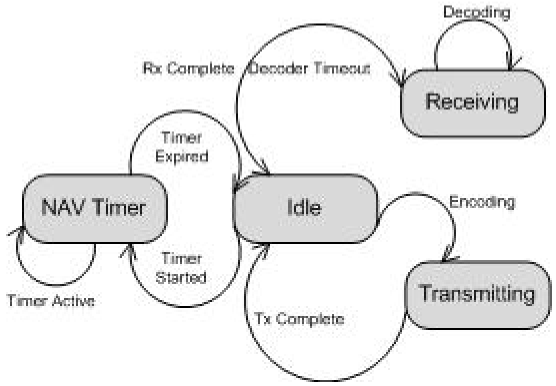

Figure 5.7: Physical Layer State Diagram

2. If the decoder signals that that it has successfully decoded data, the receiver script is exited and the received data purged, and then the receiver is started again. The data that has been decoded is passed onto the BBC-MAC Receiver class for processing.

3. When the receiver script has collected the number of samples specified by the *nsamples* parameter. This implementation will exit after collecting 16 million samples. The decoder is allowed to finish a final attempt to decode the data before the sink is purged and the receiver restarted.

4. If the decoder has been running longer than the time allotted for it to run. In this situation the data in the sink file is considered unusable, the receiver is stopped, and the sink data is purged before beginning the receiver again.

The node is never in an intentional continuous state of idleness. Any received data is handed to the upper layer for processing and the node returns to receiving as quickly as possible. In this sense idle is a transitional state.

A final component of the interface is the Network Allocation Vector (NAV) timer. Recall from the overview of the 802.11 protocol in Section 3.3.2 that it uses this timer to determine how long it should defer transmitting for. Likewise in this protocol, if this value has been set by the BBC-MAC layer, then the interface will not transmit any frames until it has expired. The majority of the states that the physical layer can be in are controlled by the BBC-MAC layer. Design characteristics of that layer determine whether a node can transmit data or if it should be in receive mode.

5.4 BBC-MAC Implementation

BBC-MAC is where the majority of all work has been directed. It controls and directs the physical layer and transforms it into a reliable medium for communications. The implementation of the protocol is a non-trivial approach to creating a jam-resistant Medium Access Control (MAC) layer that can adapt to the level of noise in the channel. The protocol adapts by measuring the interference in the channel and changing the coding configuration used at the physical layer. The layer is made up of many different software components that can be seen in Section A.1. This section will cover the main components of the protocol in such a way as to give complete coverage of operations at the layer. Figure 5.8 shows the high level operations in the BBC-MAC implementation.

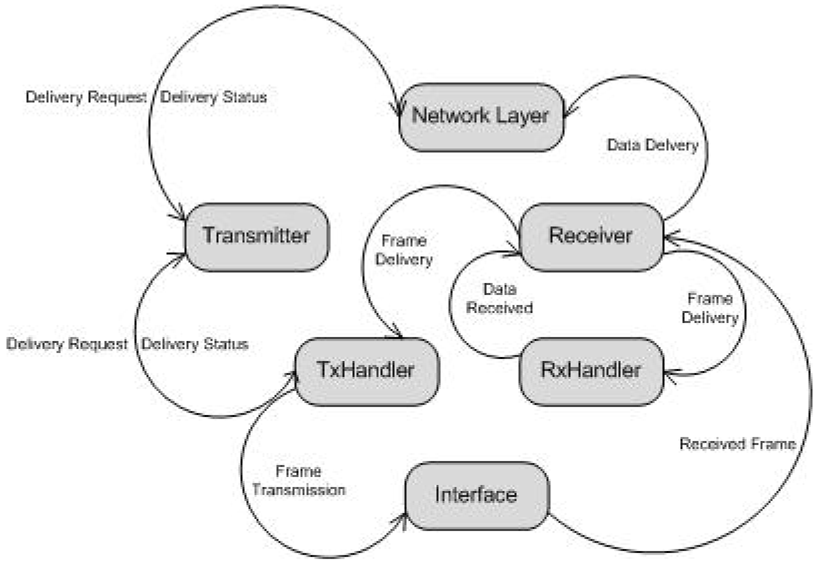

Figure 5.8: BBC-MAC State Diagram

- **BBC-MAC Frame:**

  The layer uses a common header format for all frame types. The header is relatively small, only 21 bytes. The format of the header is as follows:

  – **Destination Address:** This is a 16-bit integer that indicates the address of the node where this data is being delivered to.

  – **Source Address:** This is a 16-bit integer indicating the address of originating node for this message.

  – **Type:** This is an 8-bit integer indicating the type of the frame. 1 is a Request-to-Send (RTS) frame. 2 is a Clear-to-Send (CTS) frame. 3 is a Data frame and 4 is an Acknowledgement (ACK) frame.

  - **Source Stream ID:** This is a 16-bit integer indicating the stream ID that this data belongs on the transmitter. It is used for identifying the handler the data is destined to on the transmitters end.
  - **Destination Stream ID:** This is a 16-bit integer indicating the stream ID that this data belongs to on the recipients end. It is used for proper delivery of data to the handler for the stream on the receivers end.
  - **RSSI:** This is a 16-bit integer for the Received Signal Strength Indicator (RSSI) level at the time of transmission. It is used during the RTS/CTS handshake for configuration of the encoding for subsequent data frames.
  - **CRC:** This is a 16-bit integer containing the Cyclic Redundancy Check (CRC) of the payload.
  - **Timestamp:** This is a 64-bit integer containing the time at which this frame was transmitted.
  - **Payload:** Field containing the payload of the frame.

- **Receiver:**

  The Receiver is the module that interacts with the interface class for all inbound data. The layer views all communications in the forms of streams, where every stream should have an ID associated with it. The Receiver maintains a list of all handlers managing existing streams for both inbound and outbound streams. Whenever data has been received at the interface it is enqueued in the Receiver's inbound queue. Once the receiver is ready to process the data it will examine the Destination Stream ID field in the header and the following steps take place:

- **ID is zero:**
  1. First attempt to locate a handler which has marked its destination stream ID as the one listed as the Source Stream ID in the current frame.
  2. If no such handler exists then this is a new stream. Create a new RxHandler with a stream ID unique to this node.
- **ID is non-zero:**
  1. First attempt to locate a handler with the ID matching this ID, and pass the data to it.
  2. If no such handler exists, create a new RxHandler with a stream ID matching the Destination ID from the header.

The final component of the Receiver is the callback routine. This is used when an RxHandler has ended itself. If the handler was controlling the interface it will relinquish control of the interface. This will remove the handler from the list of handlers, clean up the memory it was occupying, and then if the statistics module has been turned on it will print out the statistics that it maintained for that stream. The routine also accepts the data that the RxHandler received, if any, and passes it to the upper layer.

- **Transmitter:**

  The Transmitter module handles all interactions with the upper layer for data that is outbound. Whenever the upper layer has data to transmit it enqueues it in this module's queue. The data is immediately popped from the queue and a TxHandler is created with a unique Stream ID to handle the remainder of the operations associated with the outbound data. Like the Receiver, the Transmitter has a callback routine that is used when the handler has terminated itself. The handler will pass the routine a message to deliver to the upper layer indicating a failure or a success. It will then remove the handler from the list of handlers and clean up the memory being occupied by the handler. If this handler was controlling the interface it relinquishes control of the interface. Finally, if the statistics module is on it will print out the statistics that the handler maintained for the stream.

- **Handlers:**

  The handlers are where most of the decisions in the protocol are made. They maintain the data and operations associated with a stream. There is a handler for inbound streams called an RxHandler and likewise for outbound streams there is a TxHandler. The interface maintains a queue of handlers that are waiting for control of the interface. The handlers also are what make the adaptive protocol possible. After the RTS/CTS handshake is made the handlers will adjust the configuration to the appropriate jam-resistance level based on the RSSI value. There are five configurations where each weigh the benefits of throughput versus jam-resistance. The following outlines the detailed operations of the handlers:

- **TxHandler:**

  As soon as the Transmitter has data in its queue it creates a TxHandler with a new stream ID. This ID becomes the source stream ID in any outbound BBC-MAC frames from this handler. The handler maintains two queues. The first is a send queue that is used by the interface to get the next frame that it should transmit from this handler. The second is the receive queue. Recall that when the interface receives data it passes it onto the Receiver module, which then locates the proper handler for the data. This queue is where the Receiver will push the data. Finally, the handler maintains a BBC Configuration object that holds the configuration options that should be passed to the BBC encoder/decoder executable whenever this handler has control of the interface. The following steps outline what happens after the TxHandler has been created:

  1. Set the configuration's source ID to the current stream ID. Recall that at the BBC encoding stage each time data is sent to the encoder it creates individual BBC frames with sequence numbers belonging to a stream ID. This is that value.

2. If the interface is operating in dynamic mode the configuration should be set for the highest jam-resistance level, otherwise leave the configuration as is.
3. Create a Request-to-Send frame with the node's current RSSI value. RTS frames contain the size of the data to be transmitted as the payload.
4. Enqueue the RTS frame in the queue and then place this handler in the interface's handler queue.

Once this initial phase has occurred, the handler must now wait for the interface to signal it via a callback routine that it has transmitted the frame. When the callback is signaled, the interface passes it a copy of the frame it just transmitted and the length of time the interface passed to the bbc_tx.py script. The following outlines what occurs when the callback is called with the two frame types that the transmitter sends:

∗ **Request-to-Send Frame:** The handler makes a blocking call to the receive queue that times out after a length of time equal to the transmit time plus buffer time.

  · **Blocking Call Returns:**

    1. The received frame is checked to make sure it is a RTS frame. If it is not, a node is responding with an incorrect frame and it is ignored.

2. The RSSI value in the frame is compared to the RSSI value which was sent in the RTS. If it is larger than our RSSI value, then it is used as the determining value for the jam-resistance level.

3. If the interface is in dynamic mode, adjust the configuration to the appropriate jam-resistance level based on the RSSI value. Once the RSSI value is obtained, the handler will do a secondary check on the current RSSI value at the node. If the current value would cause the configuration to jump to the next level of jam-resistance, a new RTS frame is created and sent out to re-adjust the receiver. This step is ignored if we are at the limit of RTS re-transmit attempts, and continue on to transmitting the data frame.

4. Create the data frame and place it in our outbound queue.

- **Blocking Call Times Out:**

  1. Check to make sure we have not exceeded our retransmit attempt value. If we have, then the handler will terminate itself and pass a failure message to the Transmitter module.

  2. Otherwise, the frame is updated with the current RSSI value and placed in our outbound queue.

∗ **Data Frame:**

The handler will make a blocking call to the receive queue to get data that will timeout after thirty seconds.

· **Blocking Call Returns:**

1. The received frame is checked to make sure it is an ACK frame. If it is not, a node is responding with an incorrect frame and it is ignored.
2. Otherwise, the handler has received an ACK frame and this stream has been completed. The handler will terminate itself and signal a successful transmission to the Transmitter module.

· **Blocking Call Times Out:**

1. Check to make sure we have not exceeded our retransmit attempt value. If we have, then the handler will terminate itself and pass a failure message to the Transmitter module.
2. Otherwise, the frame is re-enqueued in our outbound queue.

If the handler is capable of receiving the CTS from the other communicating node, it will claim ownership of the interface and start a timer equal to the length of time in the CTS frame that was received. The handler will continually check to see if the timer has expired, and if it has then the handler must give up control of the interface and signal a failure to the upper layer. This is to ensure fairness on this node for other handlers to gain control of the interface, as well as maintain the mutual agreement of how long any two communicating nodes can claim ownership of the channel.

- **RxHandler:**

  The RxHandler is created by the Receiver class, and has a significantly less complex role than the TxHandler. Upon creation, it is given a stream ID to use as a self identifier. Any frames that it transmits will use this as the Source Stream ID in the BBC-MAC frame header. Like the TxHandler, the RxHandler maintains two queues. The first is the receive queue where any date from the Receiver module is placed, and the second is the send queue used by the interface to get the data it is to send from this handler. When data is placed in the handler's receive queue, the following steps occur for each of the following frames:

  * **RTS Frame:**

    1. The frame is first checked to make sure the destination address is equal to the address specified in the interface. If it is not, then the frame is discarded.
    2. Compare the RSSI value in the frame to the current RSSI at this node. Select the higher value as the RSSI value for the purposes of configuration adjustments.
    3. Create a CTS frame with the selected RSSI value and enqueue it in our send queue.

4. At this point this node should have the higher of the two RSSI values. Using the size of the data in the RTS payload this node will estimate the time needed on the channel for the codec configuration based on the RSSI value and the size of the data and place this in the CTS frames payload.
5. The handler will now wait for the interface to signal that it has transmitted this frame and begin a timer equal to the length of time that was previously estimated that the channel would be needed for.
6. Adjust the jam-resistance level based on the RSSI value and claim ownership of the interface.

∗ **CTS Frame:** If an RxHandler gets a CTS frame it indicates that this CTS was not destined for this node and two other nodes are trying to claim the channel. The value in the payload is extracted and the NAV timer on the interface is updated with that value.

∗ **Data Frame:**

1. The frame is checked to make sure it was destined for this node. If not, it is discarded.
2. An ACK frame is created and placed in the send queue.
3. The handler signals the Receiver class of a successful data reception via the Callback and the handler is terminated.

- **ACK Frame:** This frame should have been delivered to a TxHandler if it was meant for this node. The frame is discarded and the handler terminated.

5.5 Chapter Conclusion

The purpose of this chapter was to introduce the reader to the main components that make up the implementation of the BBC-MAC protocol. The chapter began with an introduction to the hardware and software components that make it possible to achieve the goal of testing the protocol on physical hardware. We then discussed the new jammer that was created to be used during the subsequent experimental phases. The implementation details of the physical layer and the BBC-MAC protocol were covered in appropriate detail in order familiarize the reader with how the layers interact in order to create the reliable data link layer. The implementation presented in this chapter was meant to create a working prototype in order to demonstrate the feasibility in creating a MAC layer that can adapt to the level of noise detected on the channel. Some of the design decisions in the protocol could have been taken in other directions, but the prototype that has been developed here suits the purpose of demonstrating technical capability of incorporating the BBC algorithm into a protocol stack in order to provide adaptive jam-resistant communications.

Chapter 6

Phase I Experiments: Adaptive Coding Investigation

6.1 Chapter Introduction

As noted in the previous chapter, the protocol must adjust the coding parameters used on the BBC algorithm to meet the current needs of the channel. If the channel has a low degree of noise then minimal jam-resistance is needed to increase the throughput. The reverse situation must also be addressed. If there is a high level of interference then greater jam-resistance should be used in order to overcome the noise. In the explanation of the protocol, I settled on using five levels of jam-resistance, plus an additional level for the smaller RTS/CTS frames. By using five levels of jam-resistance, the protocol will be able to demonstrate its capacity to adapt to the level of noise in the channel. The goal of this experiment is to test each configuration against different levels of jamming, and against several jammers. The experiment clearly showed the tradeoff between jam-resistance and throughput, and illustrated that with just one parameter adjustment of the BBC algorithm different levels of resistance can be achieved. However, in both jammer types an upper bound was reached where the adjustment made no statistical difference in the reliability of the configuration.

6.2 Experiment Setup

The hardware configuration for this experiment required two USRPs, and at least one must have two daughterboards. The USRPs were fairly close to each other; only four feet separated them. Inside each USRP are two daughterboards, where the antennas in each are separated by 3.5 inches.

The experiment calls for a jammer to be running while a transmission is occurring. I made the design decision to always have the jammer running for at least as long as the transmission was occurring. This allows me to guarantee that the originating signal is always being jammed from the source of the transmission. For this reason, the jammer code was incorporated in the transmitter script so this could be achieved. Two types of jammers were used during this phase of experiments. The first is a pulse jammer that I created to be a targeted attack on the decoder and the modulation scheme currently used. The second is a Gaussian noise source generator that is part of the GNU Radio API. Each jammer has a range of [0,64] to select for the jamming level. On the pulse jammer the level indicates how many time steps should contain a high pulse, and the Gaussian jammer uses the level to determine the max amplitude. For this experiment the data used for encoding and transmission was a single 802.3 ethernet frame that is 1514 bytes long. The hexadecimal string for this data is listed in Section B.1.

After examining notes about the BBC algorithm and reviewing the source code, it is fairly obvious how greater jam-resistance can be achieved. An examination of

Algorithm 2 in Section 2.5.3 shows that the upper bound on the decoder is $O(2^n)$. The decoder time grows exponentially with respect to the number of 1 bits in the "buckets" for the current message being decoded, or essentially the mark density. On the BBC Real-time Engine website [Bahn March 2009], it is stated that in general, in order to achieve greater jam-resistance the configurations need to lower the mark density during the encoding. There are two parameters which determine the length of a packet and therein the mark density. The first is the *codec_message_bits* paremeter, which says how many bits long should a message be. The second is the *codec_expansion* parameter, which determines the length in which a BBC packet will be. Multiplying the *codec_message_bits* × *codec_expansion* gives the length of a BBC packet or the packet_bits parameter. The value also impacts the spreadability of the prefix hashing done when determining where a pulse should be located at, and in some cases how many marks there will be.

```
// Generate mark location for present prefix
location = 0;
for (i = 0; i < SHA1_HASH_DWORDS; i++)
  location += ((codec->digest)->Message_Digest[i])<<i;
location %= c->packet_bits;
```

Listing 6.1: codec.c lines 343-347

The code snippet above shows where in the code this is of importance. The upper bound of where a location can be is in the unsigned integer variable location. However, it is further impacted by the *packet_bits* discussed before. By raising or lowering the *codec_expansion* different mark densities can be achieved. A rough estimation for the stream density can be done by dividing the number of marks the encoder created, *N*, by the number of samples that were created, *S*. The increase in expansion also increases the number of marks produced. This is because the modular operation on the location is on a larger number, and thus fewer prefix hashes will map to the same location increasing the spreading of the marks.

Expansion	**N**	**S**	**Density**	**Transmit Time**	**Throughput**
50	31768	716832	4.43%	6s	2019bps
75	32865	1075232	3.06%	9s	1346bps
100	33451	1433632	2.33%	12s	1009bps
125	33873	1792032	1.89%	15s	807bps
150	34054	2150432	1.58%	18s	673bps
175	34257	2508832	1.37%	21s	578bps
200	34404	2867232	1.19%	23s	527bps
RTS @ 500	3406	1433632	0.24%	12s	19bps

Table 6.1: Expansion Factor Impact

I decided to leave the *codec_message_bits* parameter at the default of 512 bits or 64 bytes, and instead vary the *codec_expansion*. This decision was made because I wanted to reduce the number variables changed during testing and implementation, and instead focus on adjusting the parameter which varies the mark density. Table 6.1 shows the result of running the encoder for the chosen levels of expansions. It also lists the transmit time required to transmit each configuration. Recall that for these tests a 1514 byte ethernet frame was used. The transmission time is a simple calculation based around the number of samples that were created. The sampling rate of the USRP's is 500,000 samples a second, and for each bit modulated there are 4 samples created, and so the time required to transmit is $\lceil 4 * \frac{S}{500000} \rceil$. The ceiling of that value is used in order to obtain an integer result. This information also lets us display the nominal throughput as a function of time. The table clearly illustrates that as we increase the expansion factor we decrease the stream mark density, but also decrease the throughput.

For each expansion factor, thirty messages were sent at each jamming level until it reached two levels in which no messages made it through. It was decided to stop the tests at that point since even if one or two messages got through in subsequent levels it would be statistically irrelevant. A script was created which would begin the receiver, then begin the transmitter, and once the transmitter was finished it would end the receiver script and begin the decoder. The decoder was given thirty seconds to decode the data received. Once that time was expired it was considered a failed transmission. Successful decodes where then checked for proper CRC16 values, and only those that had matching CRC16 values were considered a success. A final test was carried out on the highest expansion factor that resulted in a reasonable amount of transmit time for a RTS frame in order to determine its resilience to jamming. This is important since these frames are significantly smaller than the ethernet frame used during the rest of the tests, and are required in the protocol for establishing channel control and coding configurations for the communicating parties. The RTS frame is just a BBC-MAC header plus a payload containing the time the initiating node estimates it will need the channel for. For these tests it was only 28 bytes, which, conveniently, an expansion of 500 resulted a transmit time of 12 seconds with an extremely low mark density of just 0.24%.

6.3 Experiments

6.3.1 Jammer RSSI Experiment

To gain an initial insight into how the two jammers would affect the channel, the first experiment was to run both jammers at each level of jamming and collect the RSSI value. As noted in the previous chapter, the RSSI value is continuously collected during the receiver's script. The experiments were executed such that only the noise data generated by the respective jammers were transmitted.

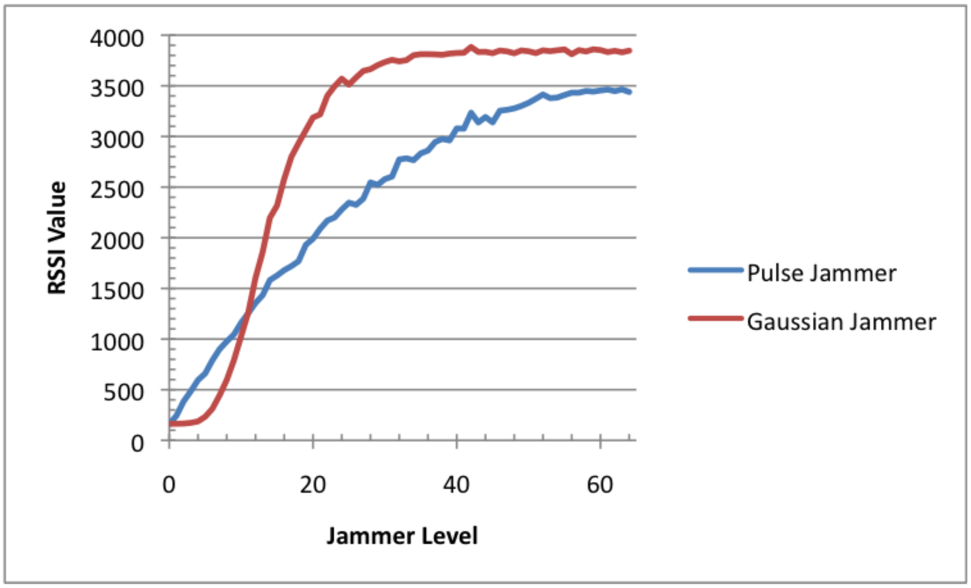

Figure 6.1: RSSI Value vs Jamming Level

Figure 6.1 shows the results of this experiment. The pulse jammer presents a uniform increase in RSSI value, while the Gaussian jammer displays a half bell curve increase. This is not surprising since the random generator used in the pulse jammer is of uniform distribution, while the Gaussian uses a Raleigh distribution. This information is necessary in order to correlate the results of the next series of tests with the configuration needed for a specific range of RSSI values.

6.3.2 Pulse Jammer Experiment

The pulse jammer that I created is meant to attack the decoder in a similar way as my demonstration of the BBC decoder in Section 2.5.4. The jammer places a high pulse where it otherwise would not be. As explained in the previous chapter, the jammer accepts an input level in the range [0,64], where the level given determines how many time steps will contain a pulse. For example, if the level is 13, the jammer will output a sink file where every 64 time steps is guaranteed to contain 13 pulses. This can create a significant amount of error and noise for the decoder, but it can not guarantee that it will induce a $\frac{13}{64} \simeq 20.31\%$ error rate. This is because if the sender was already sending a 1 where the jammer outputted a 1, it would not affect the signal, and thus it is a maximum of 20.31% bit error rate and not a guaranteed error rate.

Figure 6.2 and Table 6.2 show the results of all the tests on the ethernet frame. What is clear from the graph is that each expansion factor gives an advantage over the prior one, but as we increase the expansion we notice less improvement over the

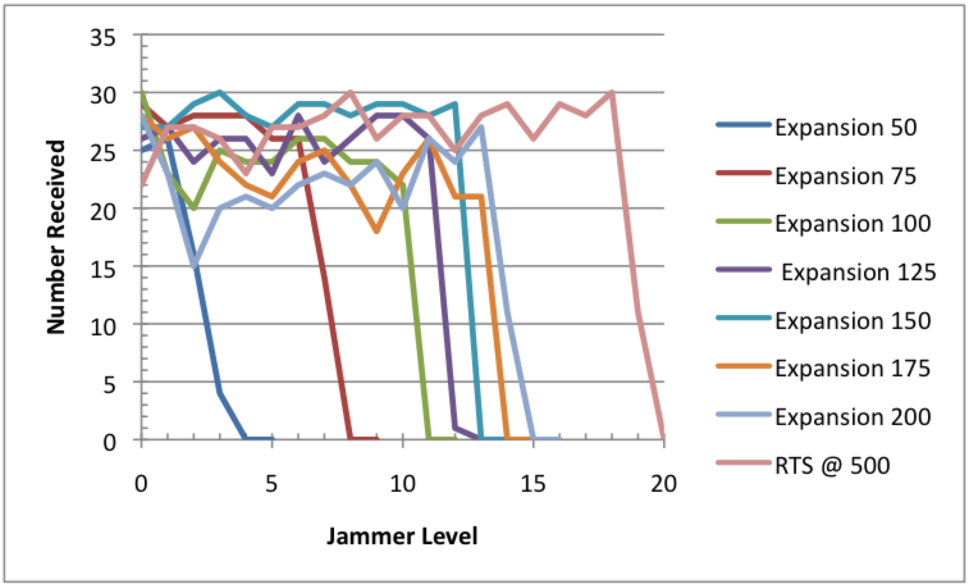

Figure 6.2: Collective Pulse Jammer Results

previous expansion. Starting at expansion 100 we begin to see the steady decrease in advantage over the previous level. This makes sense since looking back at Table 6.1, there is a very minimal decrease in mark density as we increase the expansion from 100.

Jammer Level	**50**	**75**	**100**	**125**	**150**	**175**	**200**	**RTS @ 500**
0	25	29	30	26	27	28	28	22
1	26	27	23	27	27	26	23	27
2	16	28	30	24	29	27	15	27
3	4	28	25	26	30	24	20	26
4	0	28	24	26	28	22	21	23
5	0	26	24	23	27	21	20	27
6	0	26	26	28	29	24	22	27
7	0	14	26	24	29	25	23	28
8	0	0	24	26	28	22	22	30
9	0	0	24	28	29	18	24	26
10	0	0	22	28	29	23	20	28
11	0	0	0	26	28	26	26	28
12	0	0	0	1	29	21	24	25
13	0	0	0	0	0	21	27	28
14	0	0	0	0	0	0	11	29
15	0	0	0	0	0	0	0	26
16	0	0	0	0	0	0	0	29
17	0	0	0	0	0	0	0	28
18	0	0	0	0	0	0	0	30
19	0	0	0	0	0	0	0	11
20	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0

Table 6.2: Pulse Jammer Results

Figure 6.3 shows the results of tests with an expansion of 50. This configuration does well through level one jamming where it gets 26 messages through, but then it begins to lose ground with only 16 in level two jamming, and statistically becomes unstable at level three jamming. This configuration was meant to operate in areas of little-to-no noise in order to give high throughput. It can clearly tolerate this requirement.

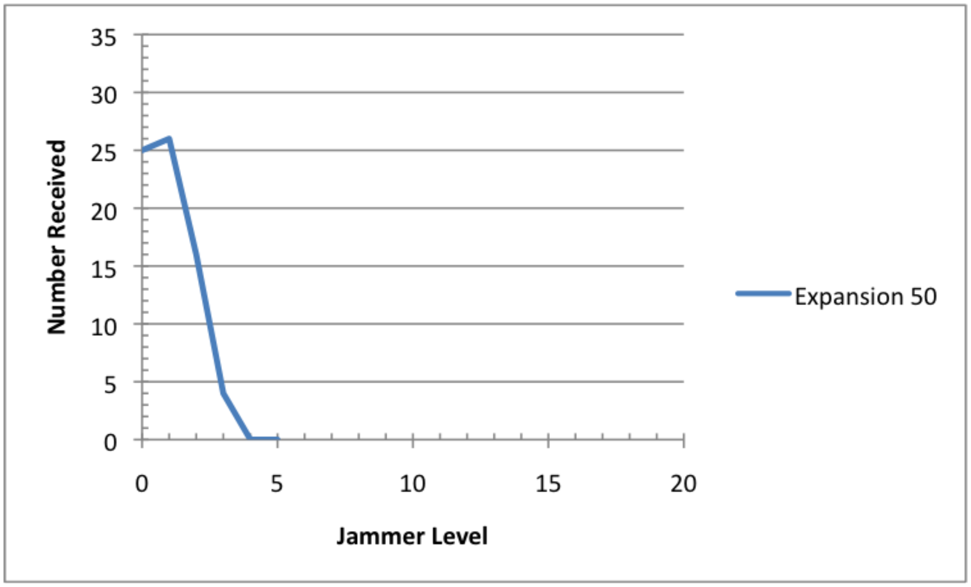

Figure 6.3: Pulse Jammer with Expansion 50

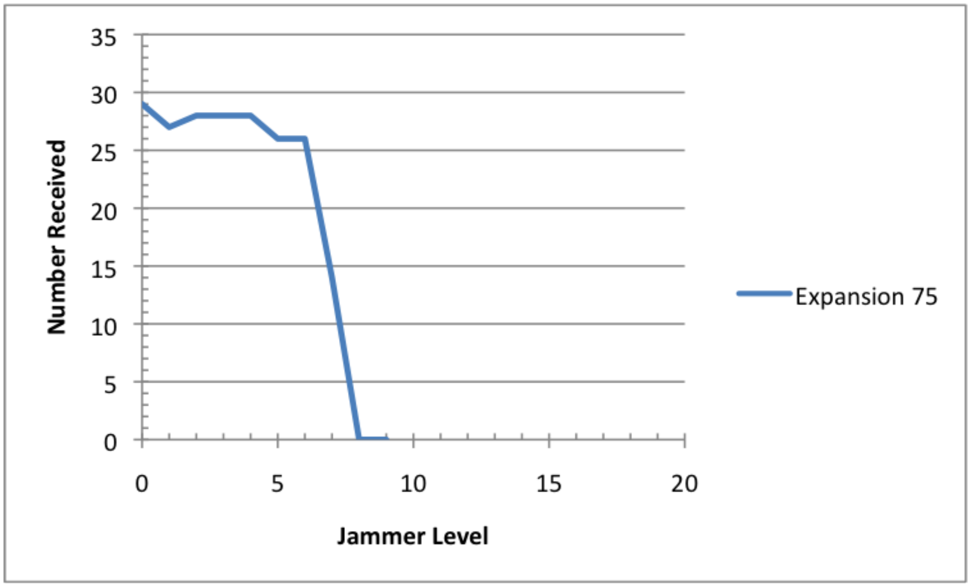

Figure 6.4: Pulse Jammer with Expansion 75

Figure 6.4 displays the results of the test on expansion 75. Expansion 75 does significantly better than 50. It manages to get 26 messages through at level six jamming, and then 14 at level seven before it goes to zero.

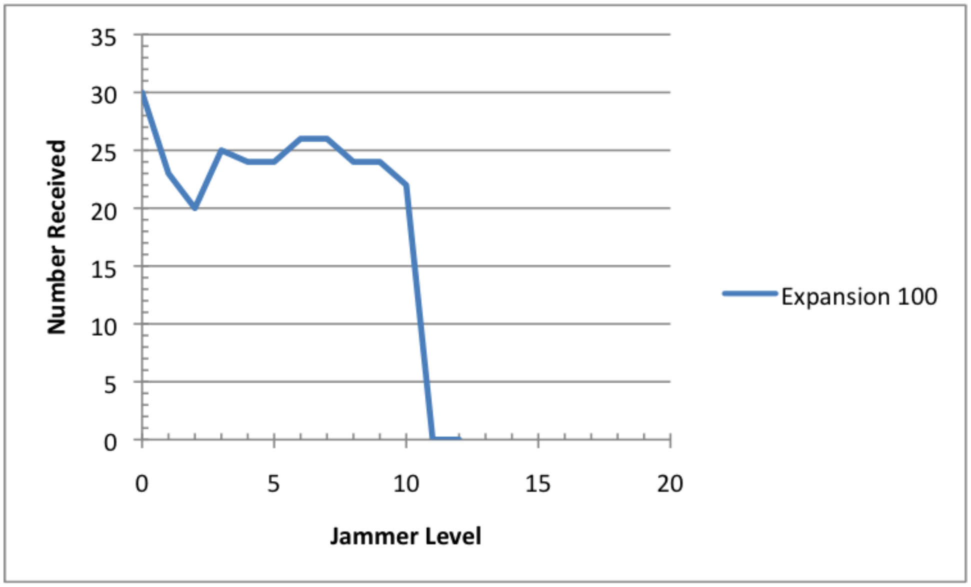

Figure 6.5: Pulse Jammer with Expansion 100

Figure 6.5 shows the results of tests on expansion 100. This expansion had several jamming levels where it had just barely over 20 messages through. However, while monitoring a lot of these tests, sometimes the radios would cause buffer under runs where the received data was lost. Combined with the randomness of the data, this can add to the low numbers on many of these early jamming levels where we should be seeing high success rates. This configuration did significantly better than the previous one. It got 22 messages through on level ten jamming, but then zero on the subsequent levels.

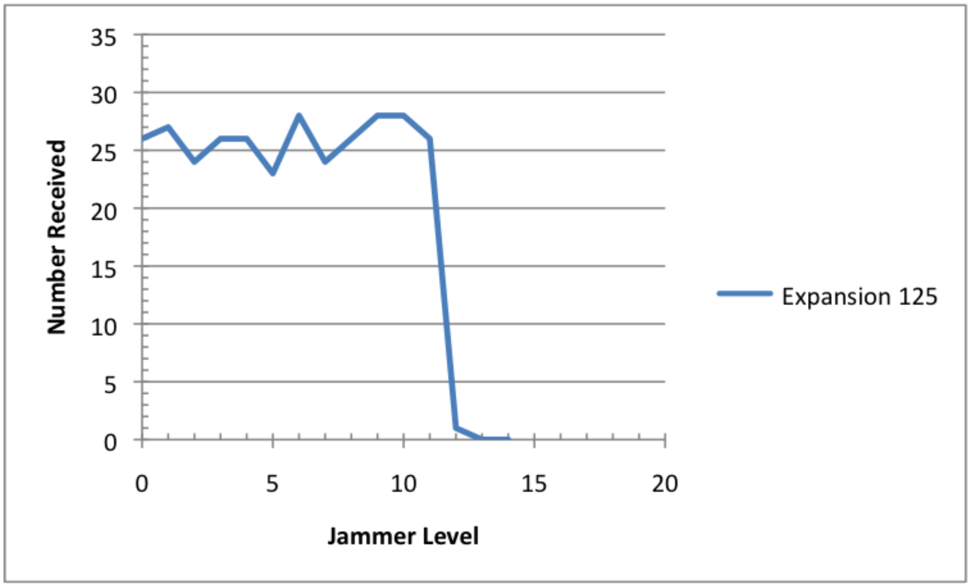

Figure 6.6: Pulse Jammer with Expansion 125

Expansion 125 shown in Figure 6.6 did only one level better than the previous one. After level 11 jamming it manages just one successful transmission before going to zero. As mentioned earlier, this is where we begin to see only minor improvements in jam-resistance.

Again, the next level of resistance at expansion 150 in Figure 6.7 shows that we are able to compensate for just one more level of jamming. This configuration got 29 messages through on level 12 jamming, but then got zero through on the subsequent tests.

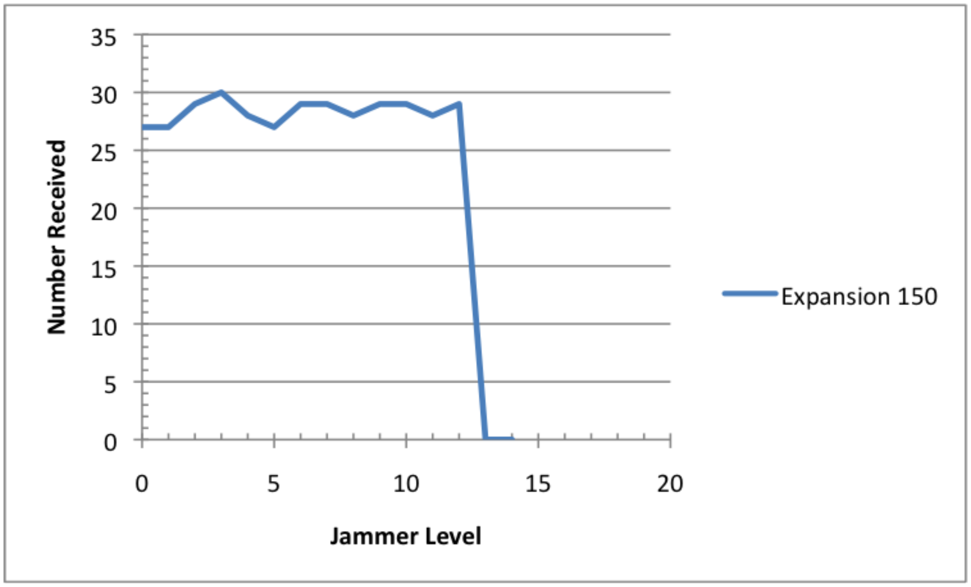

Figure 6.7: Pulse Jammer with Expansion 150

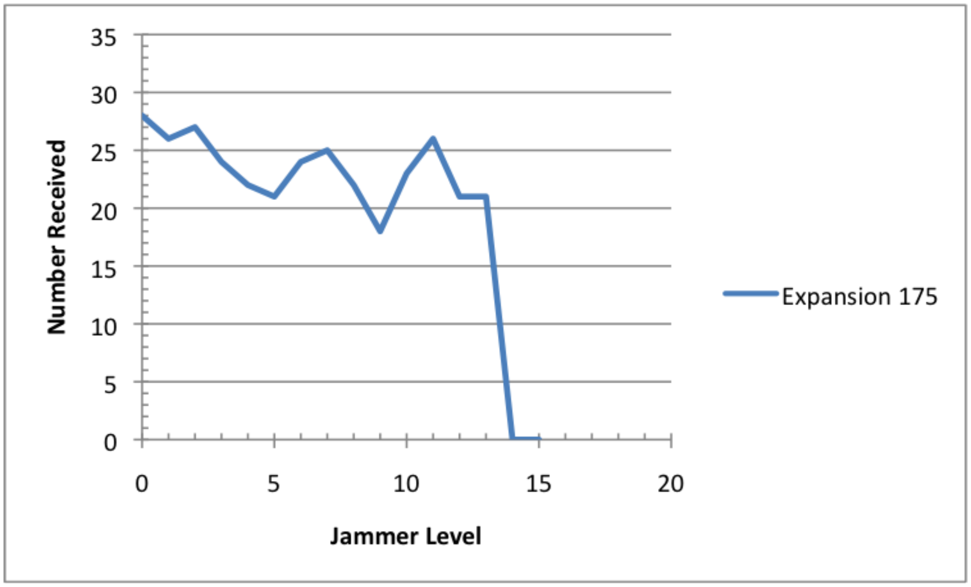

Figure 6.8: Pulse Jammer with Expansion 175

Expansion 175 shown in Figure 6.8 displays problems early during the jamming tests of just getting data decoded. This configuration manages to get to level 13 jamming with 21 of the messages getting through, and then decays to zero.

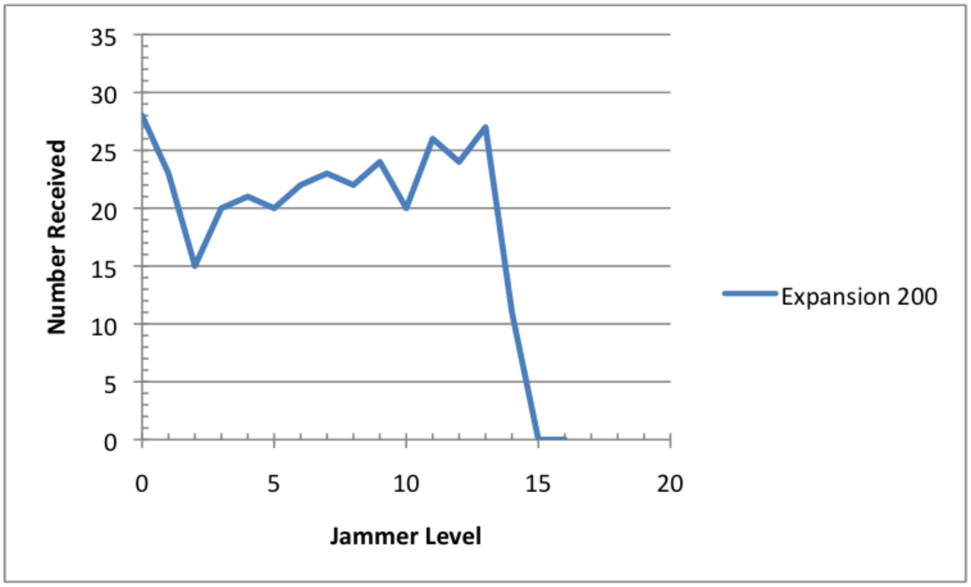

Figure 6.9: Pulse Jammer with Expansion 200

Expansion 200 shown in Figure 6.9 displays the same problems as 175 did early on in the jamming experiment. However, it did just better than 175 at level 13 with 27 messages being received, and at level 14 it got just 11 messages through. Statistically, 200 does only marginally better than 175. After reviewing the logs on this test, a lot of the problems seen in the early jamming levels were not due to the decoder timing out, but rather that the decoder was indicating a significant amount of the sequence numbers were missing and so it dropped the data.

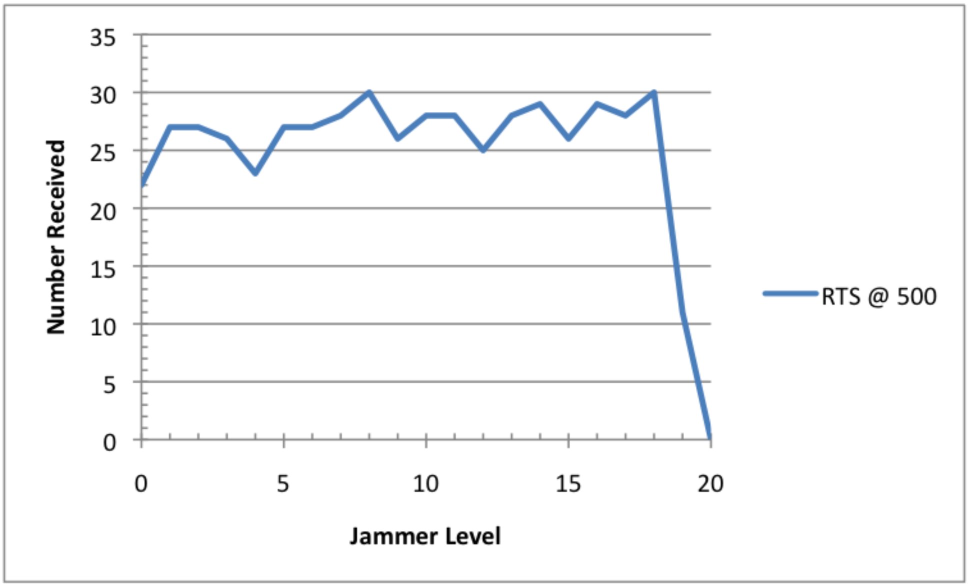

Figure 6.10: Pulse Jammer with RTS Frame at Expansion 500

The final test ran was on the RTS frame with an expansion of 500. Recalling the protocol design, the RTS and CTS frames are used for determining the proper level of jam-resistance needed between the two communicating parties. This requires that we have a high degree of certainty that the these frames are successfully transmitted. Figure 6.10 shows that this configuration allows the frame to make it through level 18 jamming with 30 messages being received, and 11 received at jamming level 19 before decaying to zero.

6.3.2.1 Pulse Jammer Experiment Conclusion

This experiment showed how the various configurations can resist the pulse jammer up to a certain level. Each of the configurations offers a benefit over the other in the form of throughput or jam-resistance. The highest expansion factor manages to resist a jamming level of 13 which equates to roughly a 20.31% bit error rate at the maximum. Finally, the tests on the RTS frame demonstrate several important factors. The first is that we can successfully transmit these frames at very high levels of jamming. The expansion of 500 resisted a jamming level of 18 which is roughly a 28.13% bit error rate at the maximum. The second important piece of information is that the smaller frame size was able to do better than the larger at getting successful receptions, but at also a significantly lower throughput.

6.3.3 Gaussian Jammer Experiment

The Gaussian jammer is a noise source generator part the GNU Radio API library. It accepts as a parameter the maximum amplitude and then uses a Gaussian distribution random generator to determine what amplitude the output signal should be. The generator accepts a jammer level in the range [0,64], where each step constitutes and increase in amplitude of 500. The maximum value for the amplitude is 32000, which conveniently works out to 64 levels.

Figure 6.11 and Table 6.3 show the results of all the tests on the ethernet frame and the last one on the RTS frame. Again we see the pattern of gradual decrease in resistance as we increase the expansion, but there also appears to be area of concentration where the limits of the expansions are met. Recalling Figure 6.2 from the pulse test, we notice that there was quite a bit more distinction between the configurations. However, looking at how the RSSI values increase from Figure 6.1 for the Gaussian jammer versus the pulse jammer, this rapid decline seems expected.

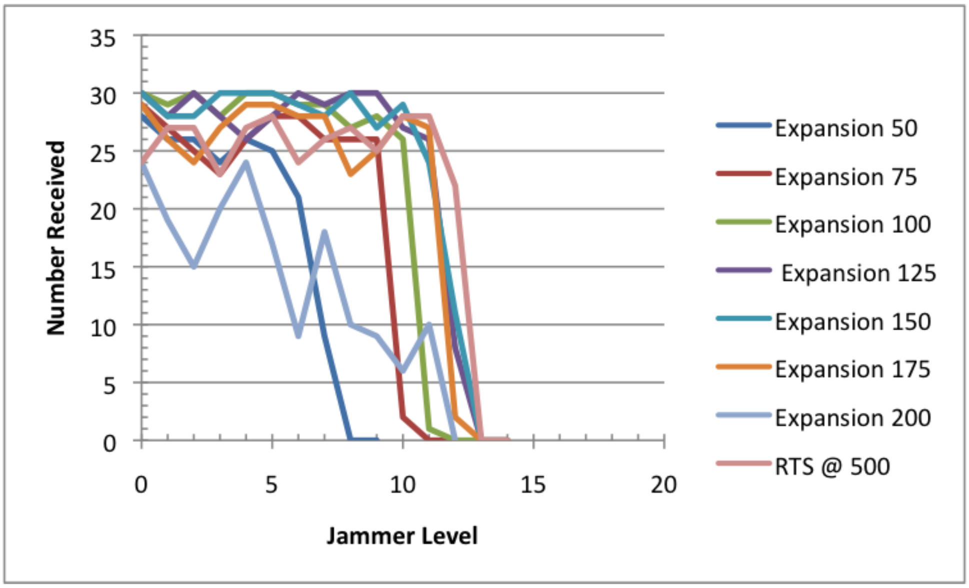

Figure 6.11: Collective Gaussian Jammer Results

Jammer Level	50	75	100	125	150	175	200	RTS @ 500
0	28	29	30	30	30	29	24	24
1	26	27	29	28	28	26	19	27
2	26	25	30	30	28	24	15	27
3	24	23	28	28	30	27	20	23
4	26	26	30	26	30	29	24	27
5	25	28	30	28	30	29	17	28
6	21	28	29	30	29	28	9	24
7	9	26	29	29	28	28	18	26
8	0	26	27	30	30	23	10	27
9	0	26	28	30	27	25	9	25
10	0	2	26	27	29	28	6	28
11	0	0	1	26	24	27	10	28
12	0	0	0	8	11	2	0	22
13	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0

Table 6.3: Gaussian Jammer Results

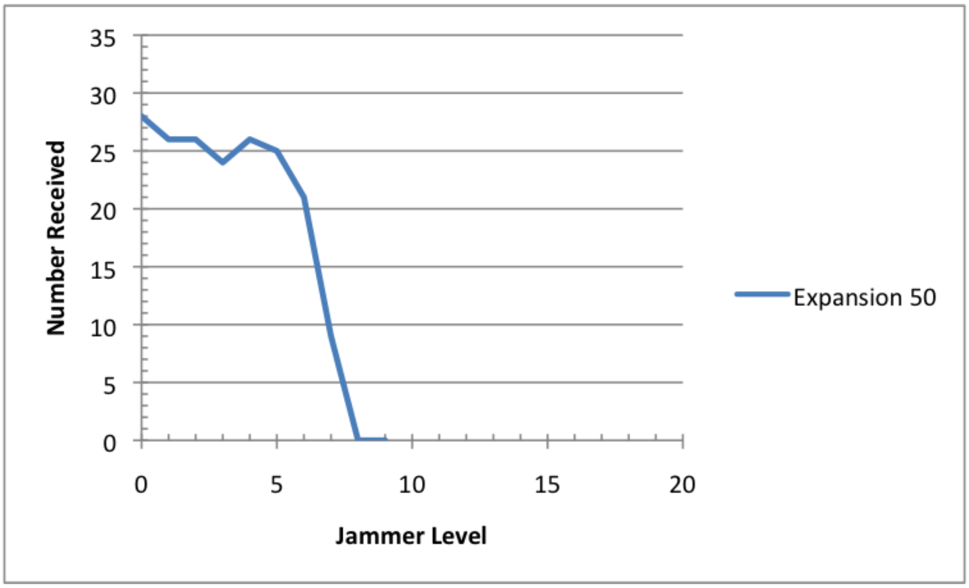

Figure 6.12: Gaussian Jammer with Expansion 50

Figure 6.12 shows the results of tests with an expansion of 50. The configuration is able to cope with jamming level six, but then declines at seven with just nine successful transmissions, and finally going to zero. Again, this configuration is meant to be used in areas of low noise so as to increase the throughput.

Figure 6.13 displays the results of the test on expansion 75. This configurations is able to edge out the previous by several levels, allowing 26 messages through on level nine jamming, and then just two on level 10 before declining to zero.

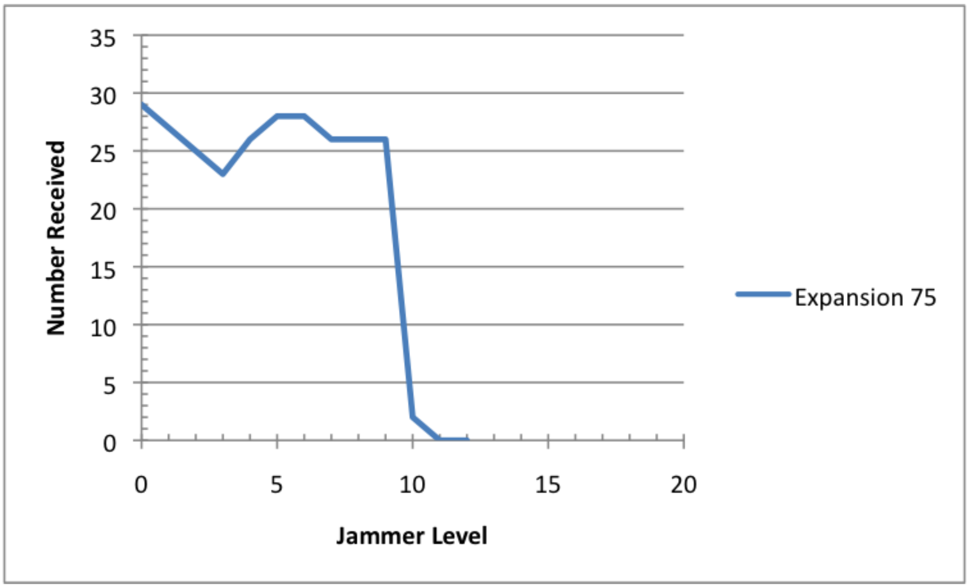

Figure 6.13: Gaussian Jammer with Expansion 75

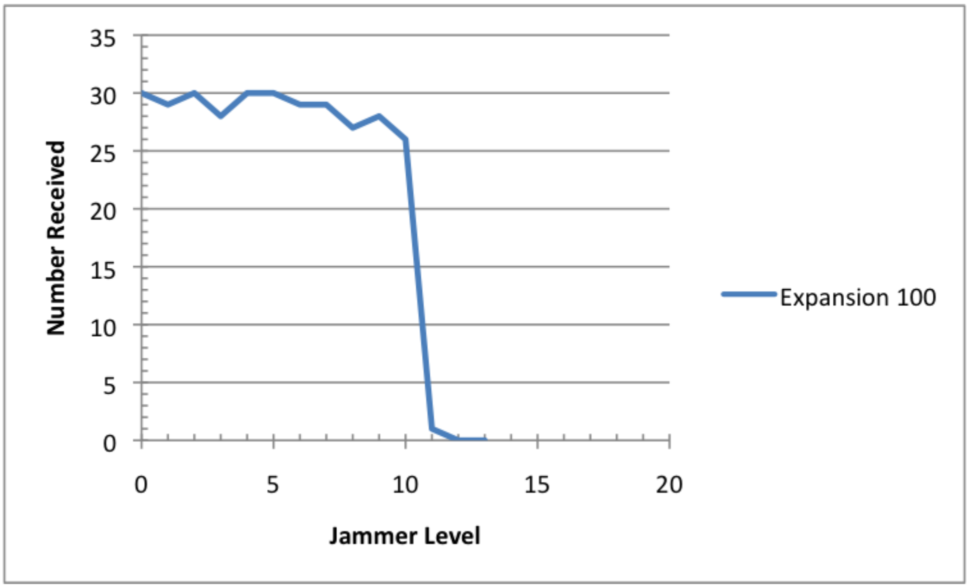

Figure 6.14: Gaussian Jammer with Expansion 100

Figure 6.14 shows the results of tests on expansion 100. This expansion was only able to do just one level better than expansion 75. There were 26 successful receives on level ten jamming, and just one on 11. At this configuration we begin to see the decline in jam-resistance advantage over the previous configuration. However, comparing the results of this test to the pulse jammer, this expansion was more stable in the Guassian jammer than in the pulse.

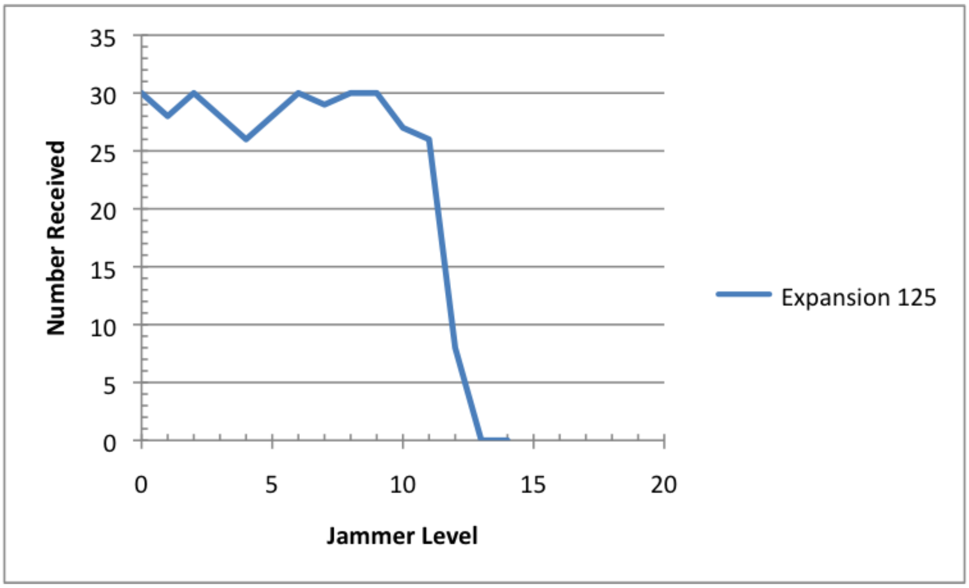

Figure 6.15: Gaussian Jammer with Expansion 125

Expansion 125 shown in Figure 6.15 did only one level better than 100. The configuration allowed 26 messages through on level 11 and eight on level 12 before the decoder began to timeout on the subsequent levels.

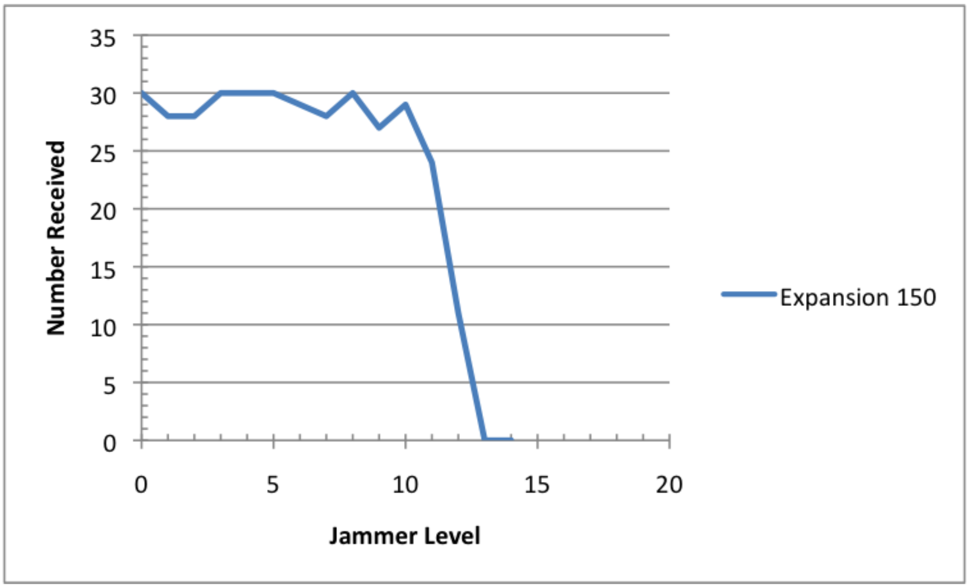

Figure 6.16: Gaussian Jammer with Expansion 150

Figure 6.16 shows the results of expansion 150. Beginning at this configuration we stop seeing a statistical advantage in raising the expansion. At level 11, 24 messages were received, and at level 12 jamming just 11 messages were received.

Expansion 175 shown in Figure 6.17 doesn't show the same problems as this respective test did on the pulse jammer. However, again this expansion was not able to do better than the previous two. It was able to get twenty-seven messages through on jamming level eleven, and just two on jamming level twelve.

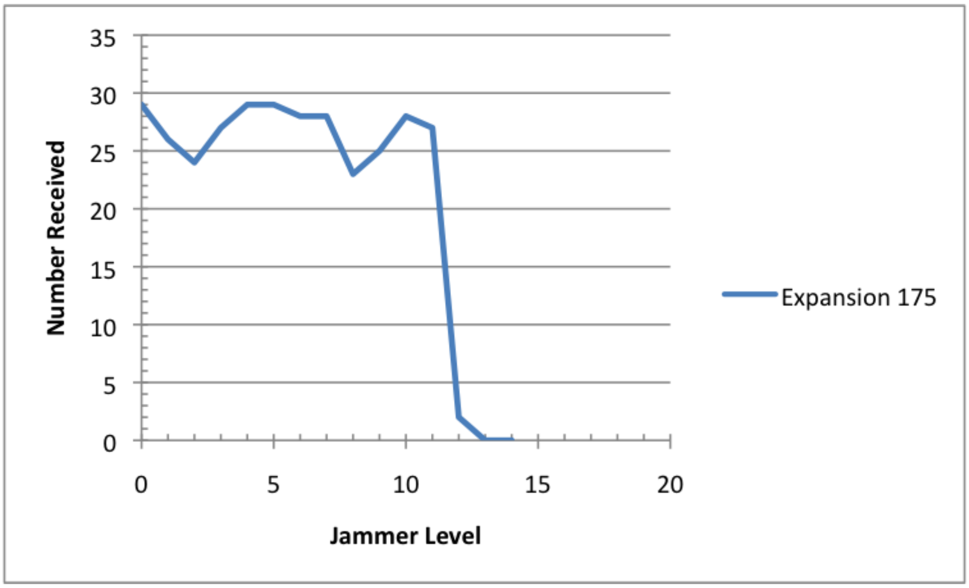

Figure 6.17: Gaussian Jammer with Expansion 175

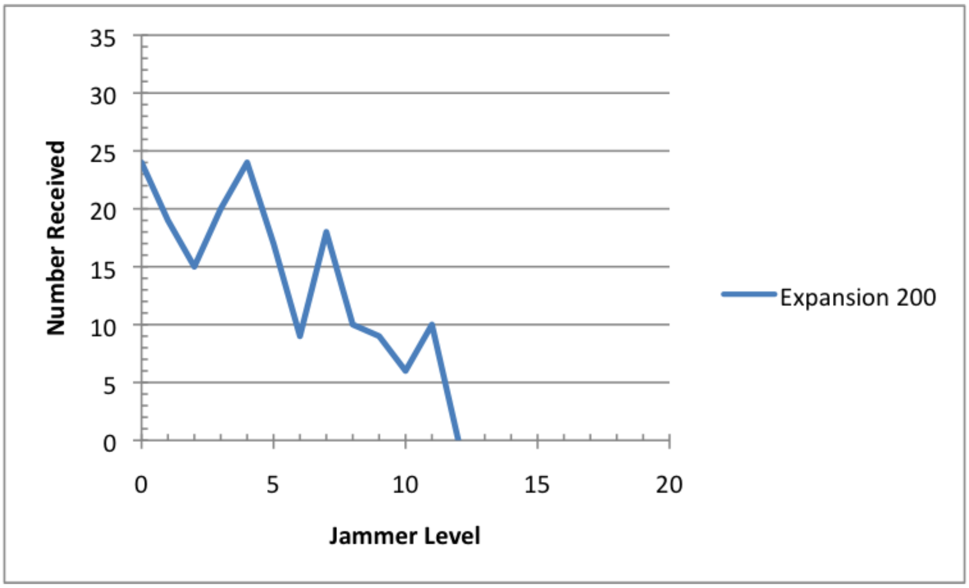

Figure 6.18: Gaussian Jammer with Expansion 200

Expansion 200 shown in Figure 6.18 displays similar issues as the respective test did in the pulse jammer, but did significantly worse. Again, looking at the logs for this test it was observed that the decoder wasn't timing out, but rather that it was missing many sequence numbers.

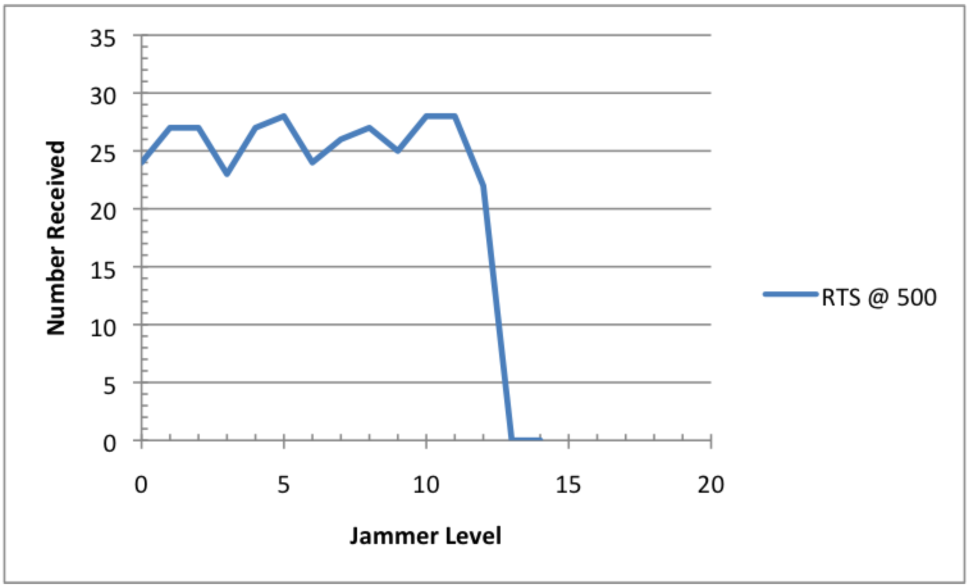

Figure 6.19: Gaussian Jammer with RTS Frame At Expansion 500

Finally, the test on the RTS frame with an expansion of 500 was ran. Figure 6.19 shows the results of this test. Again, using this configuration the RTS frame could be successfully received at a jamming level beyond what any of the data frames made it to. The configuration allowed the frame to be received 22 times on level 12 jamming before declining to zero on the remaining levels.

6.3.3.1 Gaussian Jammer Experiment Conclusion

This second experiment on the Gaussian jammer demonstrated that even against a different type of jammer the configurations can successfully use different configurations for the proper level of interference in the channel. The configurations were able to resist the jammer up to a certain point just as in the pulse test, but this time it was seen that the limit was approached far quicker than in the pulse experiment. This can be explained by the fact that the increase in jamming levels for the Gaussian noise source beyond level ten begin to increase the RSSI value significantly more than with the pulse jammer.

6.4 Chapter Conclusion

When comparing the results of the configurations from the two jammers, on the surface it seems that the lower expansions were able to resist more of the jamming levels on the Gaussian test than they did on the pulse. However, to effectively see how the configurations fared against the two jammers one must look at what RSSI value the particular jamming level creates for the respective jammer. If we consider anything less than a 50 percent success rate a failure and map the last jamming level that particular configuration was successful on, to the RSSI value from Figure 6.1, we get the results found in Table 6.4. The Gaussian jammer trials show that the upper bound for the configuration to correct the errors was reached at level ten jamming. However, what is really happening is they were not able to overcome the next level of jamming which produces an RSSI value of 1603. When the results are compared this way it illustrates that for both jammers, the RSSI value will give us the proper estimation for which configuration to use. Then, with this information in mind, we can ignore the type of jammer used and base the decision on the RSSI. This is important since the protocol will not have forehand knowledge of the jammer type and instead will only be using the RSSI as a determination factor.

Jammer	**50**	**75**	**100**	**125**	**150**	**175**	**200**	**RTS @ 500**
Pulse	385	790	1161	1256	1356	1433	1433	1769
Gaussian	312	792	1028	1263	1263	1263	596	1603

Table 6.4: RSSI Failure Levels

This phase of experiments provided the necessary information to complete the adaptive BBC-MAC protocol. The results showed that by adjusting the expansion factor, we can adapt the encoding to better suit the needs of the channel. Increasing the expansion factor reduces our throughput, but gives us greater jam-resistance. While lowering it increases our throughput but leaves us susceptible to weaker jamming attacks. The goal of this experiment was to arrive at five configurations for the data frames and an additional one for the RTS and CTS frames. I tested seven configurations on an 1514 byte ethernet frame and demonstrated each of their abilities to resist different levels of jamming on several jammers. After analyzing the results, the configurations chosen for the protocol are expansions 50, 75, 100, 150, and 175. Expansion 200 was dropped for several reasons. The most obvious is that it is entirely too unstable to be incorporated in the protocol. The second is that in both jammer tests, it gave no statistical advantage over expansion 175. Expansion 125 was not chosen since in terms of RSSI value it only did one level better than expansion 100 and I wanted a more significant buffer between the configurations. The expansion of 500 for the RTS and CTS proved to be more than adequate as evidenced by the fact that it was successfully received at RSSI levels well beyond what the configurations for the data frame achieved.

Chapter 7

Phase II Experiments: Protocol Validation

7.1 Chapter Introduction

The final phase of experiments presented in this chapter focuses on using the data from the first phase of experiments conducted in Chapter 6. In that phase, an investigation into the BBC algorithm was conducted to determine what must be done to allow for varying levels of jam-resistance. The experiments showed that in order to produce greater jam-resistance the mark density of a message must be reduced. After an examination of the algorithm, the analysis shows that in order to achieve varying levels of mark density, and thus jam-resistance, the expansion factor used in the codec must be altered. Seven variations of that value were tested against two jammers and the results showed that each variation provided a benefit in either jam-resistance or throughput. The values chosen for protocol implementation are 50, 75, 100, 150 and 175. This chapter will include those in the BBC-MAC protocol and conduct the experiments necessary to illustrate how the protocol can use these to adapt to the level of noise. The first experiment presented in this chapter shows the results of the original protocol with those values. The information from that experiment revealed a problem in the implementation and an important modification was made to improve the overall performance of the protocol.

7.2 Experiment Setup

The physical layout of these experiments remains the same as the experiments conducted in Chapter 6. The USRPs are roughly four feet apart with two transceiver daughterboards in each, where their respective antennas are 3.5 inches apart.

The information from the previous set of experiments has been used and applied to the BBC-MAC protocol implementation. Table 7.1 shows the range of RSSI values that each expansion is applied to. These ranges are based on the jamming level that the expansion was able to resist and then correlated to the average RSSI value produced at that level.

Expansion	**Min RSSI**	**Max RSSI**
50	0	300
75	301	700
100	701	1050
150	1051	1350
175	1351	4092

Table 7.1: Expansion RSSI Range

Prior to conducting the experiments a statistics module was created to allow both a RxHandler and a TxHandler to keep track of relevant information for the stream they are currently managing. The information kept by the TxHandler is:

- **RTS Count:** The number of RTS frames that were sent.

- **Data Count:** The number of data frames that were sent.

- **Send Time:** The Network Time Protocol (NTP) time that the first RTS frame was sent at.
- **ACK Time:** The NTP time that the ACK was received.
- **RTS Fail:** Flag set if this stream failed at the RTS/CTS exchange stage.
- **Data Fail:** Flag set if this stream failed at the DATA/ACK exchange stage.
- **RSSI:** Final RSSI value used for determining the expansion to use.
- **Expansion:** The expansion used for the Data transmission.
- **Channel Latency:** The difference in time between the Send and ACK time.

On the recipients end the RxHandler also maintains some information:

- **RTS Count:** The number of RTS frames that were received.
- **Data Count:** The number of Data frames that were received.
- **Data Time:** The NTP time that the data frame was received and delivered to the upper layer.
- **ACK Count:** The number of ACK frames transmitted. This should be equal to the number of data frames received.
- **CTS Count:** The number of CTS frames transmitted. This should be equal to the number of RTS frames received.

In the experiments. the following terms will be used to discuss the analysis of the data collected by the statistics on the transmit and receive end for each message that was sent:

- **False Negative:** This is where the transmitter signaled its upper layer of a failure to deliver the data, but the receiver had actually received the data and delivered it to its upper layer.

- **False Positive:** This is where the transmitter signaled its upper layer of a failure to deliver the data and the other node did not receive it.

- **Nominal Latency:** This is the difference between the Send Time in the TxHandler statistics and the Data Time in the RxHandler statistics. It is the time between when the sender first initiated communications by sending an RTS and when the recipient delivered the data to the upper layer. Under optimal conditions this is just the Channel Latency minus the time to deliver the ACK. However, under conditions where the data frame is unnecessarily re-transmited due to a missed ACK, this can be significantly lower than the Channel Latency.

- **Optimal Nominal Latency:** This is the Nominal Latency when everything worked perfectly. That is, only one RTS frame had to be sent and only one Data frame had to be sent.

The experiments were tested with the pulser jammer only. This choice was made since it gave a uniform increase in RSSI value allowing for more granularity of the tests. For each experiment 30 messages were sent at each jammer level in the range of [0,10]. The goal of the experiments are to show that the protocol would adapt to different levels of noise and not the demonstration of absolute failure. The expansions respective failure limits were demonstrated in Chapter 6. As with the experiments conducted before, whenever a data transmission was occurring on the radio, a jammer was running on the author daughterboard.

7.3 Experiments

7.3.1 Initial Protocol Implementation Experiment

The initial protocol implementation operates in the same fashion as explained in Section 5.4. The flow of a message has the following sequence:

1. The sender encodes a RTS frame with an expansion factor of 500 and transmits.
2. The recipient will respond with a CTS frame encoded with an expansion of 500.
3. The two nodes should now be in agreement on which configuration to use for subsequent frames.
4. The sender now encodes the data frame using the agreed expansion and transmits.
5. The recipient will reply with an ACK frame encoded at the agreed expansion.

This RTS-CTS-DATA-ACK flow is the complete stream assuming everything is received without error. Recall from Section 5.4 that if upon receiving the CTS the sender requests further expansion refinement, it will re-transmit a RTS to inform the recipient of a new configuration to which they should agree upon. To overcome the possible corruption of RTS and CTS frames, the sender can re-transmit an RTS frame up to two more times past the initial request. The time that the channel is allocated for the two nodes is the time needed for three transmissions of the data frame plus the time needed for a timeout. The decoder is given 30 seconds to decode the data received by the receiver before the data is considered too corrupt for processing and the data purged.

The experiment was conducted by sending 30 messages at each jammer level and recording the results from the statistics for analysis. The data used as payload is the same 1514 byte ethernet frame that was used in the experiments from Chapter 6, and the hexadecimal string of the frame can be found in Section B.1.

Beginning first with an analysis of the latency, we can see in Figures 7.1 and 7.2 that on all three measurements there is a gradual increase in latency as we increase the jammer level and thus the expansion factor needed. Tables 7.2 and 7.3 show the numbers that each graph displays, respectively. We can clearly see that there is a benefit in using the lowest jam-resistance level. Compared to the highest jam-resistance level, we are able to transmit almost 16 seconds faster in optimal conditions. Furthermore, as we go from one expansion to the next, there is an increase in latency. These two graphs demonstrate the latency tradeoff by adjusting the expansion factor.

Jammer Level	Channel Latency (s)	Nominal Latency (s)	Optimal Latency (s)
0	52.5	47.3	43.9
1	55.8	46.5	44.7
2	64.5	49.1	45.7
3	62.4	47.2	46.7
4	53.3	47.2	47.3
5	62.1	50.9	48.4
6	56.8	49.1	49.2
7	71.1	62.3	56.2
8	70.6	64.1	57.3
9	78.3	66.1	60.6
10	79.6	66.0	57.7

Table 7.2: Experiment I Latency By Jammer Level

Expansion	Channel Latency (s)	Nominal Latency (s)	Optimal Latency (s)
50	53.0	47.1	42.6
75	60.5	47.6	45.8
100	57.6	49.6	48.3
150	72.1	63.6	57.3
175	78.4	65.3	58.9

Table 7.3: Experiment I Latency By Expansion Level

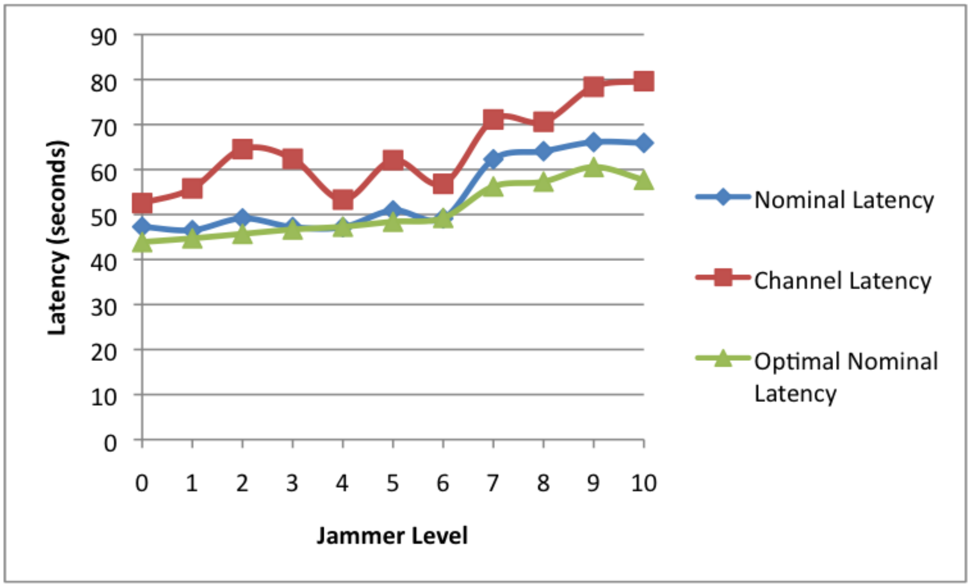

Figure 7.1: Experiment I Latency By Jammer Level

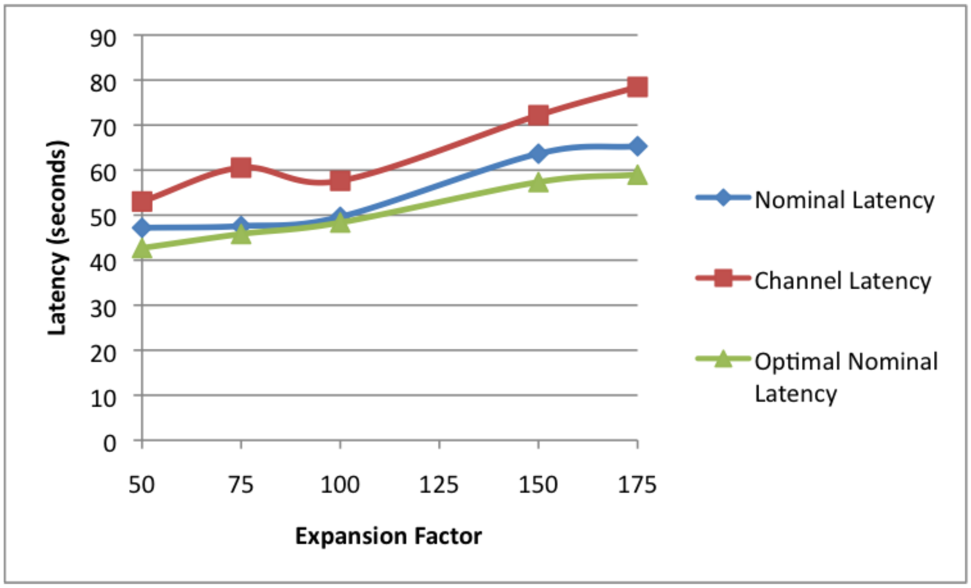

Figure 7.2: Experiment I Latency By Expansion Level

Continuing on to an analysis of the RTS frames, in Figures 7.3 and 7.4 we see how many RTS frames were sent, how many of those were received, and how many messages were sent. Under optimal conditions there should be a 1:1 ratio between the number sent and received, and not necessarily a 1:1 ratio between the number of RTS frames sent and the number of messages sent. This is because more RTS frames might be sent as needed by the protocol to adjust the expansion factor used in subsequent data and ACK frames. However, the figures clearly indicate that there was a minimal error in the RTS frames and only at level eight jamming was there increase in the adjustments made. The raw data from where this data was collected indicates several times in which expansion 175 had to be used in level eight jamming, accounting for the small separation between the number of RTS received and the number of messages sent at that level. Tables 7.4 and 7.5 numerically display the same data.

Jammer Level	**Messages Sent**	**RTS Sent**	**RTS Received**
0	30	33	30
1	30	30	30
2	30	32	30
3	30	30	30
4	30	30	30
5	30	30	30
6	30	30	30
7	30	31	30
8	30	34	33
9	30	33	30
10	30	36	32
All	330	349	335

Table 7.4: Experiment I RTS Transmits By Jammer Level

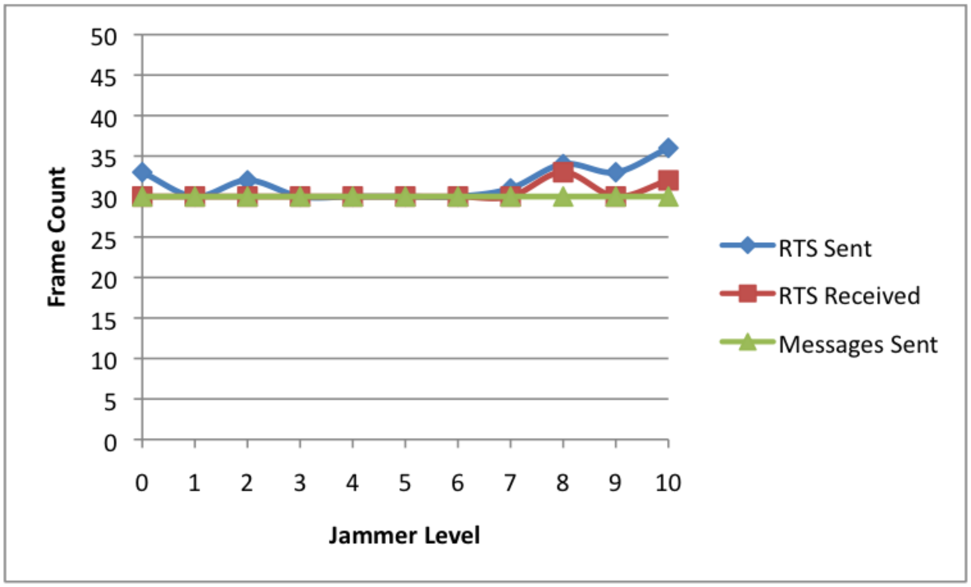

Figure 7.3: Experiment I RTS Transmits By Jammer Level

Expansion	Messages Sent	RTS Sent	RTS Received
50	24	27	24
75	96	98	96
100	91	91	91
150	71	78	74
175	48	55	50
All	330	349	334

Table 7.5: Experiment I RTS Transmits By Expansion Level

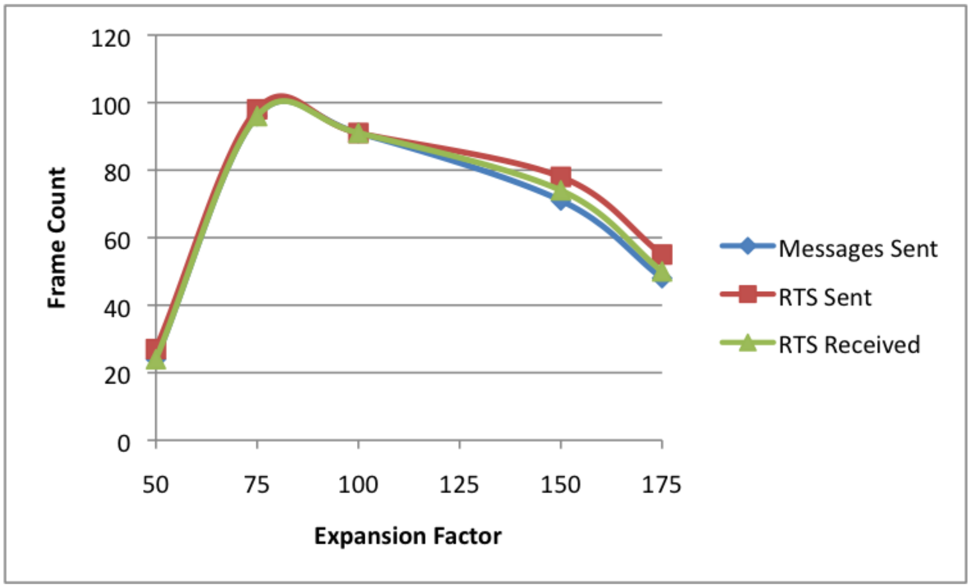

Figure 7.4: Experiment I RTS Transmits By Expansion Level

Moving onto an analysis of the data frames, Figures 7.5 and 7.6 show the results of all the data transmissions at each jammer level and expansion factor, respectively. The difference listed in Tables 7.6 and 7.7 and the figures, represents the difference between the number of data frames received and the number of messages sent. Beginning with the breakdown by jammer level, we can see that from levels one through three there is an extremely high difference between the number of messages sent and the number of data frames received, and only a minor difference between the number of data frames sent and received. This indicates a problem in data frames being acknowledged at those levels. If we move to Figure 7.6 with the breakdown by expansion, we can see that expansion 75 is the main expansion used at those levels and presents a problem getting ACK frames through. However, as we move up in expansions this problem seems to to disappear, indicating that as we increase the expansion the ACK frames have a smaller error rate.

Jammer Level	**Messages Sent**	**Data Sent**	**Data Received**	**Difference**
0	30	32	32	2
1	30	43	42	12
2	30	43	42	12
3	30	43	43	13
4	30	32	32	2
5	30	38	35	5
6	30	32	32	2
7	30	34	31	1
8	30	31	30	0
9	30	33	33	3
10	30	38	33	3
All	330	399	385	55

Table 7.6: Experiment I Data Transmits By Jammer Level

Expansion	**Messages Sent**	**Data Sent**	**Data Received**	**Difference**
50	24	26	26	2
75	96	136	134	38
100	91	103	99	8
150	71	76	73	2
175	48	58	53	5
All	330	399	385	55

Table 7.7: Experiment I Data Transmits By Expansion Level

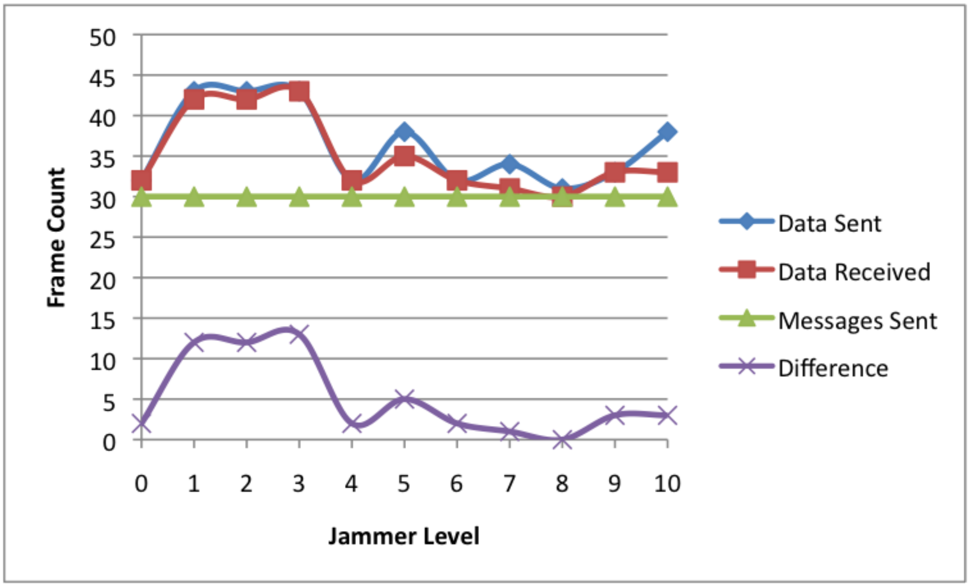

Figure 7.5: Experiment I Data Transmits By Jammer Level

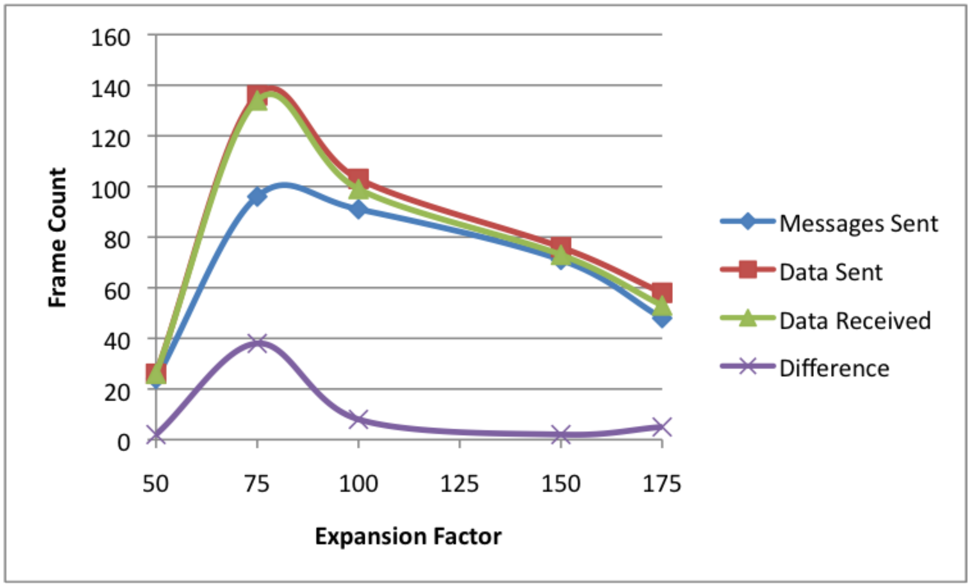

Figure 7.6: Experiment I Data Transmits By Expansion Level

Finally, we conclude by looking at the overall message failure rate broke down by jammer level and expansion factor in Tables 7.8 and 7.9. The tables display the number of false negatives (FN) and false positives (FP) at each jammer level and expansion. Recall the problems with ACK frames from the analysis of the data frames indicates how these FNs can come about. The FN for expansion 75 is at three with only one other FN occurring at the highest jammer level. The FN at the highest jammer level is not surprising since it is approaching the limit of what expansion 175 is able to tolerate. However, the most important statistic is that on no level was there a FP or a failure to actually deliver the data message.

Jammer Level	**Messages Sent**	**False Negatives**	**False Positives**
0	30	0	0
1	30	2	0
2	30	1	0
3	30	0	0
4	30	0	0
5	30	0	0
6	30	0	0
7	30	0	0
8	30	0	0
9	30	0	0
10	30	1	0
All	330	4	0

Table 7.8: Experiment I Message Errors By Jammer Level

Expansion	Messages Sent	False Negatives	False Positives
50	24	0	0
75	96	3	0
100	91	0	0
150	71	0	0
175	48	1	0
All	330	4	0

Table 7.9: Experiment I Message Errors By Expansion Level

This experiment demonstrated the initial capability of the BBC-MAC protocol and its capacity to adjust to the level of noise in the channel. The analysis of the data collected clearly shows that by adjusting the jam-resistance level we can either gain a benefit in throughput or a benefit in the ability to cope with greater jamming levels at the cost of throughput. However, the analysis of the data frames indicated that there was a serious problem at the lower jam-resistance levels in being able to return ACK frames successfully. The next section aims to address this problem by modifying the way the protocol implements the RTS-CTS-DATA-ACK frame exchange sequence.

7.3.2 Refined Protocol Implementation Experiment

After the initial protocol implementation experiment was conducted, the results show that there was a problem on the lower jam-resistance configurations in getting the ACK frames back to the sender. The analysis showed that there was a relatively small separation between the number of data frames sent and the number received, indicating the weakness in the protocols implementation of the RTS-CTS-DATA-ACK frame exchange. Upon further analysis, this was result of the ACK frames being so small in size, that on the lower resistance levels the smallest amount of noise would corrupt the frame easily. This can then result in much higher data frame re-transmission rates, and as seen, higher levels of false negatives. The simple solution then becomes to always encode the ACK frames at a high jam-resistance level. This final experiment makes only this modification to the protocol and will encode the ACK frames at the same expansion as the RTS and CTS frames. The exchange of frames now follows this series:

1. The sender encodes a RTS frame with an expansion of 500 and transmits.
2. The recipient will respond with a CTS frame encoded with an expansion of 500.
3. The two nodes should now be in agreement about which configuration to use for subsequent messages.
4. The sender now encodes the data frame using the agreed expansion and transmits. Once the transmission is complete the transmitter adjusts its codec expansion to be listening on expansion 500. If a timeout occurs, it will adjust its configuration back to the agreed upon level and re-transmit the data frame.
5. The recipient will reply with an ACK frame encoded at expansion 500. Once the transmission is complete, it will adjust its codec expansion back to the previous expansion used for the data frames. This is in case the sender still didn't receive the ACK and re-transmits the data frame.

The goal of this modification is to reduce the number of unnecessary re-transmits of the data frames, and increase the channel efficiency by eliminating the need to own it for so long. This should also further reduce the number of false negatives seen at the lower expansions. However, this will also increase the channel latency since it will take longer to transmit the ACK frame encoded at the higher expansion. The experiment is then conducted in the same manner as the one in Section 7.3.1. Thirty messages sent at each jammer level, using the pulse jammer that is always running whenever a transmission is occurring.

Beginning again by looking at the latencies in Figures 7.7 and 7.8, we see the same increase in latency from the lowest expansion up to the highest. This is not a siginifcant change in the optimal latency, as expected, since the modification doesn't affect the time it takes to get a RTS-CTS-DATA through in optimal conditions. However, we are also seeing an increase in the channel latency. It was expected to increase, but it increased significantly more than what we see in Figures 7.1 and 7.1. This larger-than-expected jump is due to a higher rate of RTS re-transmissions than what we saw in the prior experiment. However, we again see that each expansion gives us the benefit of either increased throughput or increased jam-resitance.

Jammer Level	Channel Latency (s)	Nominal Latency (s)	Optimal Latency (s)
0	58.8	48.1	43.4
1	63.2	49.2	46.3
2	69.0	56.7	47.9
3	69.7	57.5	47.8
4	71.6	57.7	51.3
5	77.3	66.4	50.7
6	78.2	68.1	54.5
7	79.8	64.6	56.4
8	84.5	71.2	57.5
9	78.5	68.0	59.4
10	86.6	74.0	60.9

Table 7.10: Experiment II Latency By Jammer Level

Expansion	Channel Latency (s)	Nominal Latency (s)	Optimal Latency (s)
50	58.8	48.1	43,4
75	67.2	54.3	47.2
100	72.8	60.9	51.2
150	79.3	66.3	56.4
175	86.1	74.0	60.1

Table 7.11: Experiment II Latency By Expansion Level

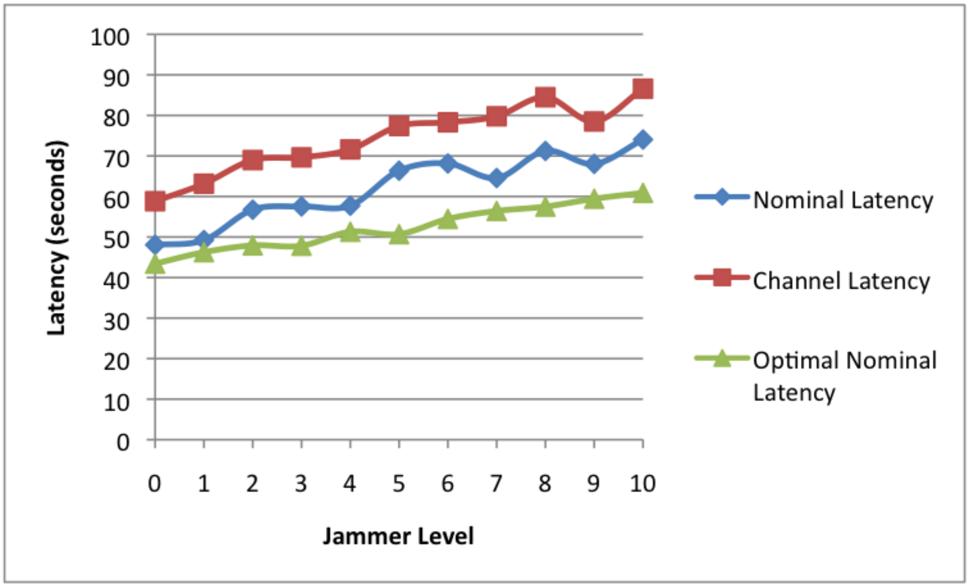

Figure 7.7: Experiment II Latency By Jammer Level

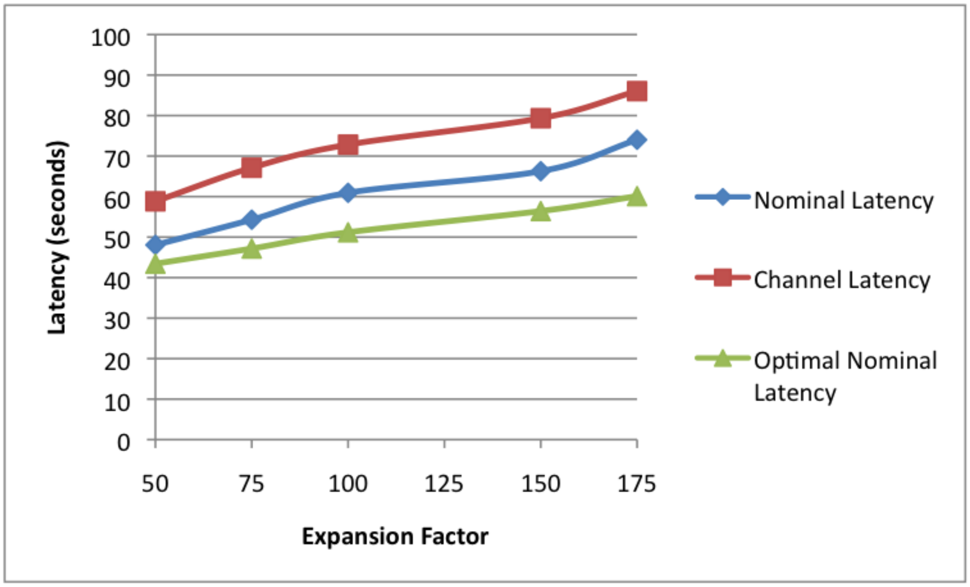

Figure 7.8: Experiment II Latency By Expansion Level

Figures 7.9 and 7.10 show the RTS transmit rates for this experiment. As noted, we see a slightly higher transmission rate of RTS frames. Some of this can be attributed to the larger number of expansion adjustments that took place. This is evidenced by the fact that the number of frames received is steadily increasing over the number of frames sent, but there are several levels that display higher RTS error rates. Reviewing the logs of the trials indicated that an unusually large number of RTS transmissions began while the receiver was in an unprepared state, or the receiver had to stop receiving due to sample limitations in the middle of an RTS transmission. However, the protocol never failed at the RTS-CTS handshake, and resolved the problems with re-transmissions.

Jammer Level	**Messages Sent**	**RTS Sent**	**RTS Received**
0	30	30	30
1	30	31	31
2	30	34	31
3	30	37	33
4	30	34	33
5	30	40	35
6	30	40	36
7	30	36	35
8	30	37	36
9	30	34	34
10	30	32	32
All	330	385	366

Table 7.12: Experiment II RTS Transmits By Jammer Level

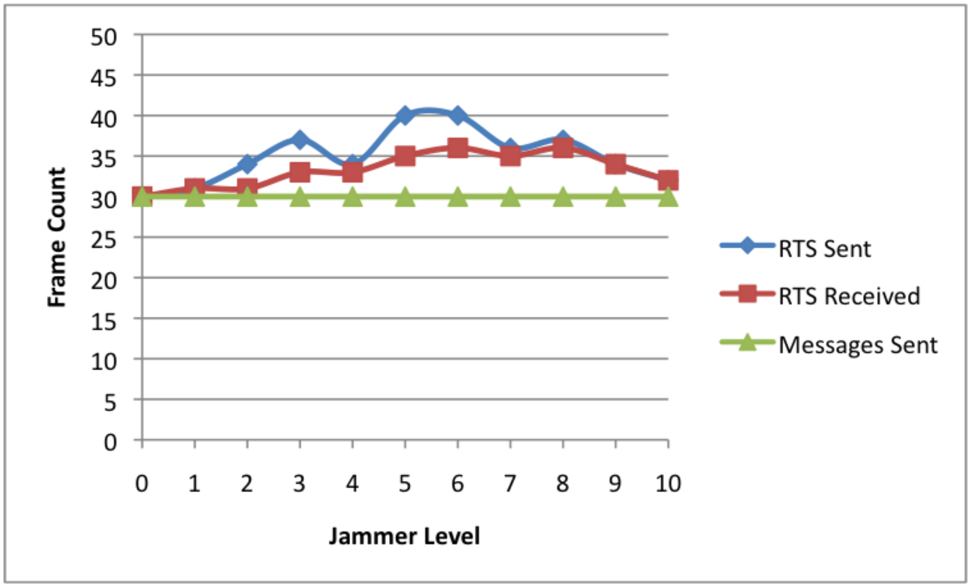

Figure 7.9: Experiment II RTS Transmits By Jammer Level

Expansion	Messages Sent	RTS Sent	RTS Received
50	30	30	30
75	86	97	90
100	71	86	80
150	74	92	87
175	69	80	79
All	330	385	366

Table 7.13: Experiment II RTS Transmits By Expansion Level

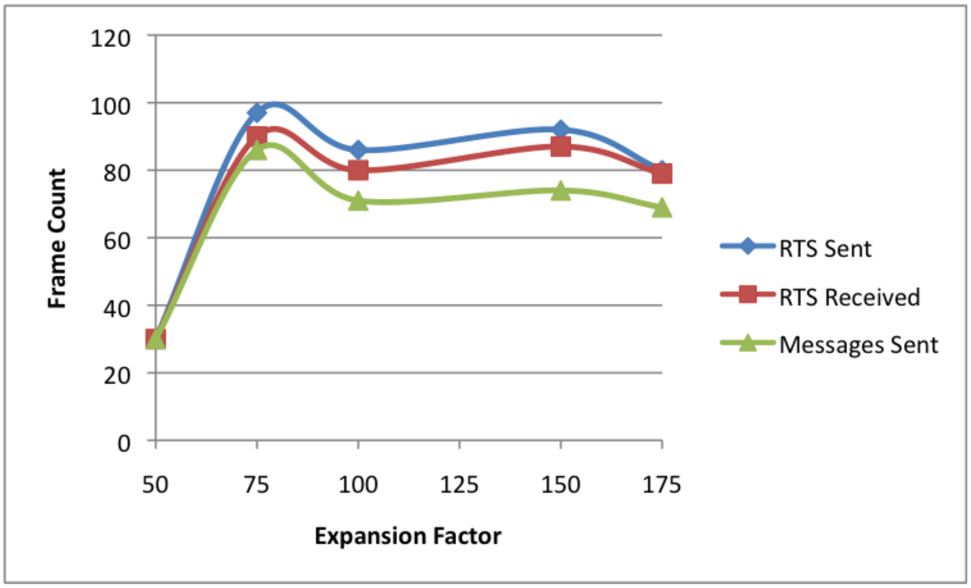

Figure 7.10: Experiment II RTS Transmits By Expansion Level

We now move onto the analysis of the data frame re-transmit rates. Figures 7.11 7.12 show the results of this experiment. The graphs clearly show that the modification resolved the problem of unnecessary re-transmissions of data frames. The difference seen from the previous experiment is included on the graphs, and there is a large margin between the two differences. This indicates that we have successfully reduced the number of data transmissions that went unacknowledged and the numbers can be seen in Tables 7.14 and 7.15.

Jammer Level	**Messages Sent**	**Data Sent**	**Data Received**	**Difference**
0	30	34	31	1
1	30	34	33	3
2	30	33	31	1
3	30	31	31	1
4	30	32	31	1
5	30	31	30	0
6	30	30	30	0
7	30	32	32	2
8	30	34	31	1
9	30	32	30	0
10	30	41	31	1
All	330	364	341	11

Table 7.14: Experiment II Data Transmits By Jammer Level

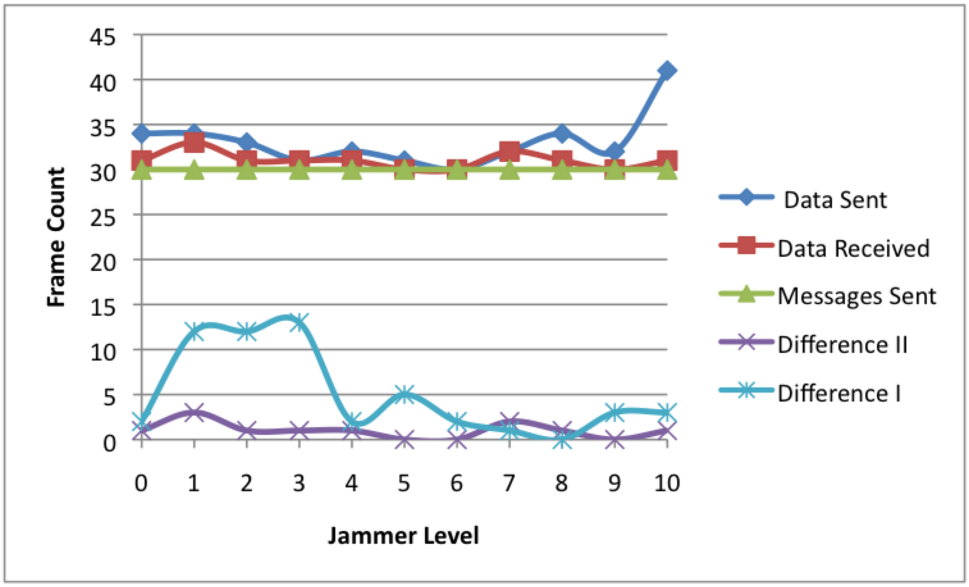

Figure 7.11: Experiment II Data Transmits By Jammer Level

Expansion	Messages Sent	Data Sent	Data Received	Difference
50	30	34	31	1
75	86	94	91	5
100	71	74	72	1
150	74	77	76	2
175	69	85	71	2
All	330	364	341	11

Table 7.15: Experiment II Data Transmits By Expansion Level

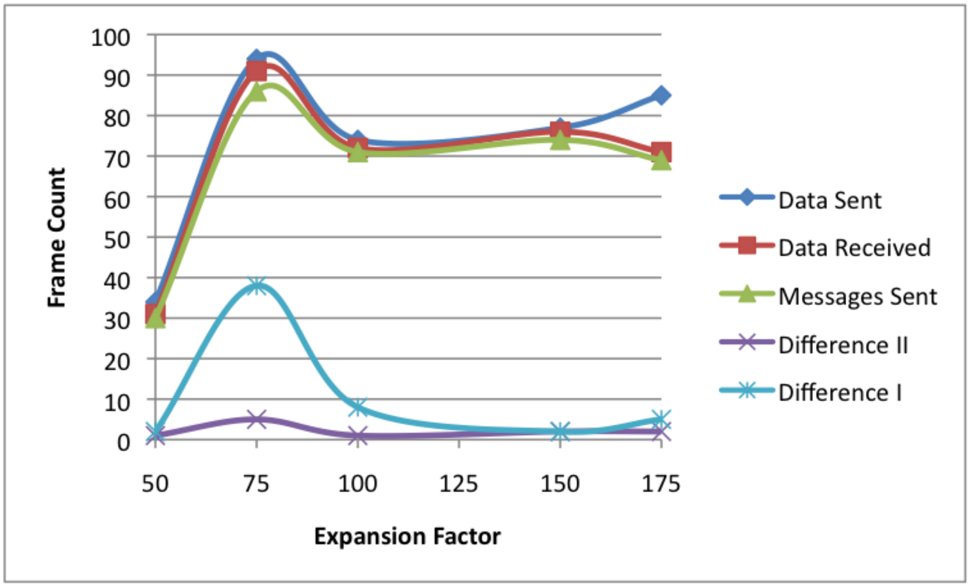

Figure 7.12: Experiment II Data Transmits By Expansion Level

Finally, we conclude by analyzing the message failure rates of this experiment. Tables 7.16 and 7.17 show that on only level ten jamming did we see false negatives, and at no point was there a false positive, or a failure to deliver the data. The seemingly static number of false negatives at jammer level ten appears to be an indication that expansion 175 is just barely capable of handling the amount of interference induced by that jammer level. However, the number of false positives at the lower jamming levels have been successfully eliminated by the modification made for this experiment.

Jammer Level	**Messages Sent**	**False Negatives**	**False Positives**
0	30	0	0
1	30	0	0
2	30	0	0
3	30	0	0
4	30	0	0
5	30	0	0
6	30	0	0
7	30	0	0
8	30	0	0
9	30	0	0
10	30	2	0
All	330	2	0

Table 7.16: Experiment II Message Errors By Jammer Level

Expansion	**Messages Sent**	**False Negatives**	**False Positives**
50	30	0	0
75	86	0	0
100	71	0	0
150	74	0	0
175	69	2	0
All	330	2	0

Table 7.17: Experiment II Message Errors By Expansion Level

This experiment focused on addressing the issue in data frame acknowledgments that we exposed in the first experiment in Section 7.3.1. The analysis of this experiment shows that with the modification of encoding the ACK frames at the same expansion level as the RTS and CTS frames, we are able to reduce the number of unnecessary re-transmissions of data frames at the lower expansion levels, and further reduce the number of false negatives.

7.4 Adaptive vs Non-Adaptive

To wrap up the discussion on the BBC-MAC protocol I would like to show how the protocol performs when we remove the adaptive portion. Removing the adaptive portion would then eliminate the RTS-CTS exchange, and instead no matter the level of noise the highest expansion of 175 would be used for the DATA frame, and then the expansion 500 on the ACK. Under perfect situations, the RTS/CTS/ACK frames encoded at 500 take 12 seconds to transmit. Table 7.18 then shows the Round Trip Time (RTT) for a single DATA-ACK exchange for the non-adaptive, and then the adaptive protocol at their respective expansion levels.

Configuration	50	75	100	150	175	Non-Adaptive
RTT (s)	42	45	48	54	57	32

Table 7.18: Adaptive vs Non-Adaptive

The table highlights an issue where even at our lowest jam-resistance level, we are not transmitting faster than the non-adaptive protocol. The problem is in the encoding of the RTS/CTS/ACK frames. Since these are encoded at such a high expansion, they require a lot of time to transmit, and thus there is a significant penalty in simply transmitting only one DATA-ACK exchange post the RTS-CTS exchange. In order to resolve this problem, we need to lower the penalty incurred by using the RTS-CTS exchange to setup both nodes. I propose that instead of encoding the RTS/CTS/ACK frames at expansion 500, we lower this to only be as high as the highest expansion used of 175. The frames will now only require five seconds to transmit, and again assuming perfect conditions, Table 7.19 shows how this modification affects the time needed to complete a transmission with a single DATA-ACK exchange. The modification will also reduce the time needed for the non-adaptive protocol since the ACK frame is no longer encoded at 500.

Configuration	50	75	100	150	175	Non-Adaptive
RTT (s)	21	24	27	33	36	26

Table 7.19: Adaptive vs Non-Adaptive with Modification

With this modification we now have significantly reduced overhead in a single DATA-ACK exchange. However, several of the adaptive configurations still take longer than the non-adaptive protocol for a single DATA-ACK exchange. Most MAC protocols support fragmentation, and thus support multiple DATA-ACK exchanges. 802.11 supports multiple of these exchanges after a single RTS-CTS handshake, as long as the individual fragments do not exceed the length specified by the station, and can support up to 16 of the DATA-ACK exchanges for a single RTS-CTS handshake [IEEE 2007]. If we now look at how the adaptive protocol performs against the non-adptive protocol when we allow up to 16 exchanges to occur we arrive at Figure 7.13. Referring to Figure 7.13, the adaptive protocol shows faster round trip transmission times than the non-adaptive protocol in all but the highest expansion level case. As expected, the adaptive 175 never splits because it always has the overhead of the RTS/CTS handshake. However, this handshake is what allows the lower expansions to be used when necessary.

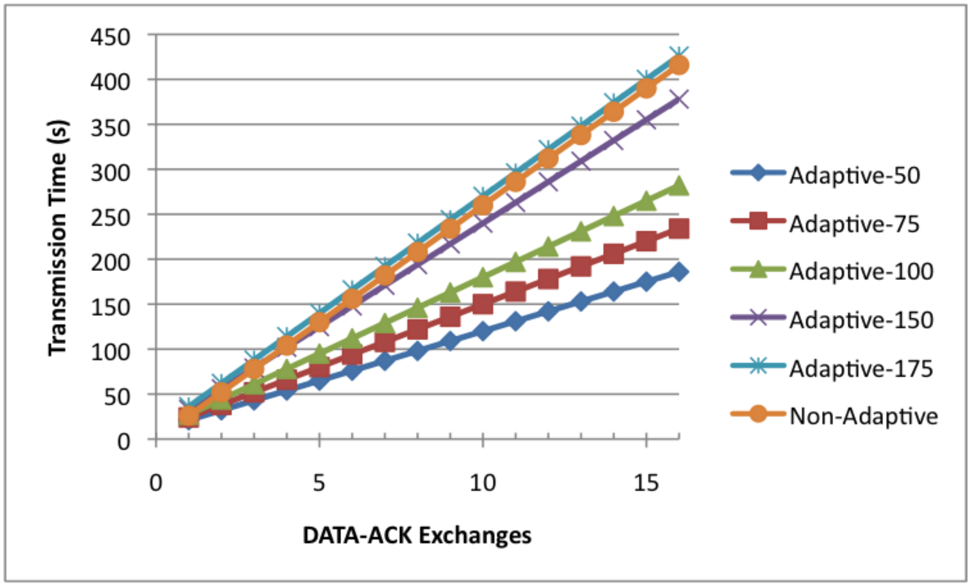

Figure 7.13: Adaptive vs Non-Adaptive by DATA-ACK Exchanges

A final thought on the non-adaptive versus adaptive is how well would it perform when an adversary is jamming for some period of time, and then stops jamming. For example, if we were to transmit 100 messages, and the adversary would jam a certain percentage of those, at what point does the non-adaptive protocol start to outperform the adaptive protocol. Assuming that the jammer is either running at full capacity for that percentage or not at all, the adaptive protocol is either using expansion 175 or expansion 50 for those respective scenarios. Figure 7.14 shows at what percentage the adaptive protocol converges with the non-adaptive protocol with respect to the number of DATA-ACK exchanges that occur after the initial RTS-CTS handshake.

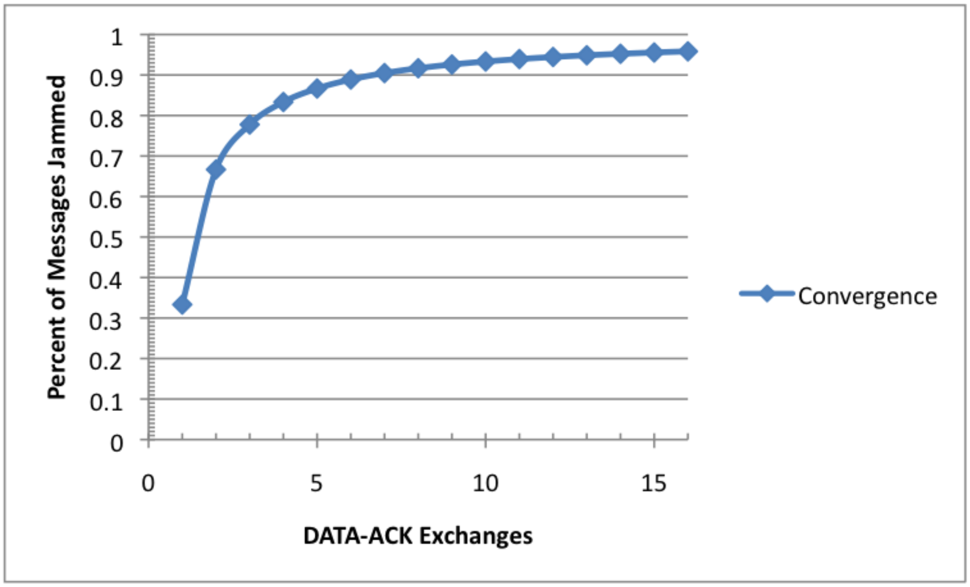

Figure 7.14: Adaptive vs Non-Adaptive Convergence

As expected, the percentage of jamming that the adaptive protocol tolerates over the non-adaptive increases with the respect to the number of DATA-ACK exchanges.

The adaptive protocol demonstrates its superiority over the non-adaptive protocol as there are more of these exchanges, and this because the penalty incurred by the RTS-CTS handshake is minimal compared to the number of DATA-ACK exchanges. However, even if there was a single DATA-ACK exchange, the adaptive protocol will still outperform the non-adaptive protocol 33% of the time. Given that we can never gauge how much of the time an adversary will jam, or how much data will need to be framed from the upper layer, we can see that the adaptive protocol will provide a benefit in either situation.

The final aspect of this modification that needs to be validated is the RTS frame encoded at expansion 175's ability to resist at least the same amount of jamming levels that that the highest expansion was able to with the data frames. To test this I ran the same resilience test that was run in Section 6.3.2. The RTS frame was encoded at expansion 175 and I transmitted the frame 30 times at each jammer level until there were two levels where zero frames were successfully received. Figure 7.15 shows the results of this trial. The graph shows that the modification allows the RTS frame to be successfully received through level 13 jamming. Recall from Section 6.3.2, that the data frame encoded at expansion 175 was also successful at completing transmissions through level 13 jamming.

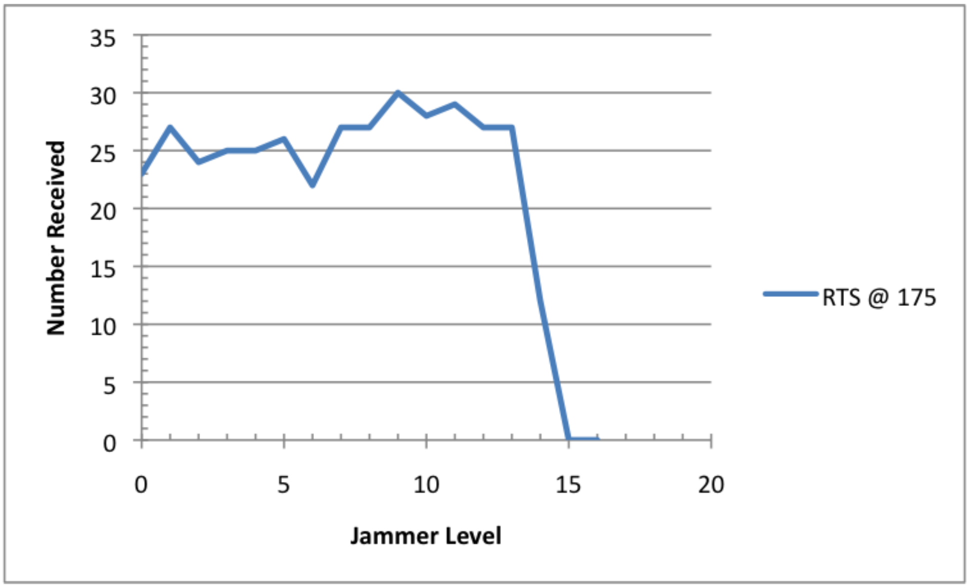

Figure 7.15: Pulse Jammer with RTS Frame at Expansion 175

The modification made to the protocol improves the performance by reducing the time needed to complete the RTS-CTS-DATA-ACK exchange of frames between the sender and the receiver. The modification does not alter the effectiveness of the protocol in adapting to the level noise, and only improves the time required to complete a transmission at all the the resistance levels. The results of the modification demonstrates the adaptive BBC-MAC protocols superiority to the non-adaptive protocol that would be using the highest expansion level at all times.

7.5 Chapter Conclusion

This chapter presented the final phase of experiments for the BBC-MAC protocol. The experiments in this chapter verified the capability of the protocol to adapt to the level of noise by controlling the configuration of the BBC encoder and decoder at the physical layer. The first experiment implemented the information obtained from the first phase in Chapter 6. After analyzing the results, it was shown that there was an issue in the frame exchange where at lower jam-resistance levels the ACK frame was easily being corrupted. In the second experiment, I proposed a solution where the ACK frames are always encoded at the same jam-resistance level as the RTS and CTS frames. With this one modification I was able to improve the efficiency of the protocol by significantly reducing the number of unnecessary data frame transmits due to a missed acknowledgment. The second experiment further solidified the protocol's ability to cope with a sundry of jamming levels by only using the necessary jam-resistance as indicated by the RSSI value. A final discussion of how the protocol performs against a non-adaptive version was given. The analysis showed that the implementation after the second set of experiments was not able to outperform a non-adaptive protocol. A modification to the protocol was presented where the RTS/CTS/ACK frames would be encoded at the highest expansion available for the DATA frames. The analysis of this modification demonstrated that it was now able to outperform the non-adaptive protocol, and the performance gap increased with respect to the number of DATA-ACK exchanges that occur for each RTS-CTS handshake. By controlling the codec configuration used at the physical layer, BBC-MAC is able to provide higher throughput in exchange for lower jam-resistance and vice versa, effectively adapting to channel needs.

Chapter 8

Key Contributions

- Designed a Medium Access Control (MAC) layer for wireless nodes that can adapt to the level of noise detected in the channel.

- Implemented a working prototype of a protocol stack for adaptive jam-resistance communications on software defined radios (SDRs) including the physical layer and a data link layer based on the design presented in the dissertation.

- Demonstrated how the BBC algorithm could be modified and controlled to provide different jam-resistance levels.

- Demonstrated that by altering the configuration options on the BBC algorithm, specific levels of jam-resistance can be achieved that provide greater throughput or greater jam-resistance.

- Proved that by combining the BBC algorithm with the BBC-MAC implementation, an adaptive protocol can be created that proactively determines the proper configurations to use based on channel needs.

- Improved upon the initial design of the BBC-MAC implementation by altering the frame exchange sequence.

- Performed a literature review on wireless communications and technologies, the BBC algorithm, and the MAC layer and its supporting facilities.

Chapter 9

Conclusion

This dissertation presents the relevant technologies and literature for wireless communications, and presents a novel approach to providing jam-resistance at the MAC layer. The current state of MAC layer research has not used the approach to solving noise in the channel that has been presented in this dissertation. Noise on the channel can be induced by many factors including environmental interference, unintentional jamming from other nodes, or intentional jamming. Current protocols attempt to solve the problems induced by unintentional jamming by relying on control frames, multiple channels, or mathematical probabilities. Only one MAC protocol has been presented that is directly concerned with adversarial wireless jamming [Awerbuch, Richa and Scheideler 2008]. However, the protocol attempts predict the time steps that jamming is not occurring, and has no mechanism to allow communications to occur while jamming is taking place. The protocol presented in this dissertation allows communications to continue in the presence of a jamming attack. Corruption of transmissions due to jamming is overcome by leveraging a recent coding algorithm for error-correction. Furthermore, the protocol dynamically adjusts the coding properties of the algorithm to change the level of jam-resistance with respect to the level of noise detected in the channel. By leveraging the BBC message encoding, this research provides a MAC layer which is resistant to jamming unlike any other MAC layer protocol currently in existence.

Future work with this protocol stack should be directed at using a new modulation scheme for the physical transmission of the encoded data. While the modulation scheme used in the prototype served the purposes of demonstrating the technical feasibility of the protocol, it also takes a significant amount of time to transmit. By combining this research with a mature modulation scheme, the latency of the protocol would be significantly improved. The ultimate test of latency is how well does the protocol support voice communications. Future work should be directed at improving the latency not only through testing different modulation schemes, but also by optimizing the BBC decoder. If the decoder could be optimized to not only have a tighter upper bound, but also to spend less time looking at invalid messages, the latency of the communications could be significantly reduced.

This research effort created a bi-layer protocol stack for wireless mobile nodes. A physical layer was implemented based on previous work that handles the coding and modulation of data for transmission, and the necessary components to interact with physical hardware. A MAC layer was then created that would control all activities at the physical layer. The layer proactively adjusts the coding configuration used at the physical layer to provide an adaptive jam-resistant protocol stack. By adapting to channel needs, BBC-MAC is able to provide only the necessary amount of jam-resistance in order to provide better throughput when possible, and greater jam-resistance when necessary. Uncommon to MAC layers in literature, this dissertation presents a prototype that has been implemented and validated on physical hardware instead of through a computer simulation. The results of the various phases of experimentation demonstrate the ability of the layer to react to channel conditions. Based on the experiments in this dissertation it was shown that the protocol is capable of adapting to the level of jamming. The dissertation contributed to the field of wireless communications by creating an adaptive single-hop MAC layer for noisy channels.

Bibliography

Abramson, N. [1970], THE ALOHA SYSTEM–Another alternative for computer communications, *in* 'Proceeding of the Fall Joint Computer Conference', Vol. 37, pp. 281–285.

Agrawal, D. and Zeng, Q. [2006], *Introduction to Wireless and Mobile Systems*, Nelson, chapter 3,6,7, pp. 57–78, 125–168.

Awerbuch, B., Richa, A. and Scheideler, C. [2008], A Jamming-Resistant MAC Protocol for Single-Hop Wireless Networks, *in* 'PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing', ACM, New York, NY, USA, pp. 45–54.

Bahn, W. [March 2009], 'BBC Real-time Engine'.
URL: *http://www.williambahn.com/bbc/software/real_time_engine/index.htm*

Baird, L. C., Bahn, W. L., Collins, M. D., Carlisle, M. C. and Butler, S. C. [2007], Keyless Jam Resistance, *in* 'Proc. IEEE SMC Information Assurance and Security Workshop IAW '07', pp. 143–150.

Baldwin, R. O., Nathaniel J. Davis, I. and Midkiff, S. F. [1999], 'A Real-time Medium Access Control Protocol for Ad Hoc Wireless Local Area Networks', *SIGMOBILE Mob. Comput. Commun. Rev.* **3**(2), 20–27.

Bayraktaroglu, E., King, C., Liu, X., Noubir, G., Rajaraman, R. and Thapa, B. [2008], On the Performance of IEEE 802.11 under Jamming, *in* 'Proc. INFOCOM 2008. The 27th Conference on Computer Communications. IEEE', pp. 1265–1273.

Bharghavan, V., Demers, A., Shenker, S. and Zhang, L. [1994], MACAW: A Media Access Protocol for Wireless LAN's, *in* 'SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications', ACM, New York, NY, USA, pp. 212–225.

Chiang, J. T. and Hu, Y.-C. [2007], Cross-Layer Jamming Detection and Mitigation in Wireless Broadcast Networks, *in* 'MobiCom '07: Proceedings of the 13th annual ACM international conference on Mobile computing and networking', ACM, New York, NY, USA, pp. 346–349.

Chirdchoo, N., Soh, W.-S. and Chua, K. C. [2008], MACA-MN: A MACA-Based MAC Protocol for Underwater Acoustic Networks with Packet Train for Multiple Neighbors, *in* 'Proc. IEEE Vehicular Technology Conference VTC Spring 2008', pp. 46–50.

Coupechoux, M., Baynat, B., Bonnet, C. and Kumar, V. [2005], 'CROMA – An Enhanced Slotted MAC Protocol for MANETs', *Mob. Netw. Appl.* **10**(1-2), 183–197.

Ephremides, A., Wieselthier, J. and Baker, D. [1987], 'A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling', *Proceedings of the*

IEEE **75**(1), 56–73.

Fang, Z., Bensaou, B. and Yuan, J. [2004], Collision-Free MAC Scheduling Algorithms For Wireless Ad Hoc Networks, *in* 'Proc. IEEE Global Telecommunications Conference GLOBECOM '04', Vol. 5, pp. 2770–2774.

Forouzan, B. [2007], *Data Communications and Networking*, fourth, international edition edn, McGraw-Hill, 1221 Avenue, New York, NY, 10020, chapter 6, 11-14, pp. 161–190, 307–444.

Garcia-Luna-Aceves, J. J. and Fullmer, C. L. [1999], 'Floor acquisition multiple access (FAMA) in single-channel wireless networks', *Mob. Netw. Appl.* **4**(3), 157–174.

Garcia-Luna-Aceves, J. J. and Raju, J. [1997], Distributed Assignment of Codes for Multihop Packet-Radio Networks, *in* 'Proc. MILCOM 97', Vol. 1, pp. 450–454.

Gerla, M. and Tzu-Chieh Tsai, J. [1995], 'Multicluster, Mobile, Multimedia Radio Network', *Wireless Networks* **1**(3), 255–265.
URL: *http://dx.doi.org/10.1007/BF01200845*

Haas, Z. and Deng, J. [2002], 'Dual Busy Tone Multiple Access (DBTMA)-A Multiple Access Control Scheme for Ad Hoc Networks', *Communications, IEEE Transactions on* **50**(6), 975–985.

Hui, J. [1984], 'Throughput Analysis for Code Division Multiple Accessing of the Spread Spectrum Channel', *Vehicular Technology, IEEE Transactions on* **33**(3), 98–102.

IEEE [2007], 'IEEE Standard For Information Technology-Telecommunications And Information Exchange Between Systems-Local And Metropolitan Area Networks-Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) And Physical Layer (PHY) Specifications', *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)* pp. C1–1184.

Jain, N., Das, S. R. and Nasipuri, A. [2001], A Multichannel CSMA MAC Protocol with Receiver-Based Channel Selection for Multihop Wireless Networks, *in* 'Proc. Tenth International Conference on Computer Communications and Networks', pp. 432–439.

Joa-Ng, M. and Lu, I.-T. [1999], Spread Spectrum Medium Access Protocol with Collision Avoidance in Mobile Ad-hoc Wireless Network, *in* 'Proc. IEEE Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM '99', Vol. 2, pp. 776–783.

Jubin, J. and Tornow, J. [1987], 'The DARPA Packet Radio Network Protocols', *Proceedings of the IEEE* **75**(1), 21–32.

Kanzaki, A., Hara, T. and Nishio, S. [2007], An Efficient TDMA Slot Assignment Protocol in Mobile Ad Hoc Networks, *in* 'SAC '07: Proceedings of the 2007 ACM symposium on Applied computing', ACM, New York, NY, USA, pp. 891–895.

Karn, P. [1990], MACA - A New Channel Access Method for Packet Radio, *in* 'Computer Networking Conference', Vol. 9, pp. 134–140.

Kloul, L. and Valois, F. [2005], Investigating Unfairness Scenarios in MANET using 802.11b, *in* 'PE-WASUN '05: Proceedings of the 2nd ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks', ACM, New York, NY, USA, pp. 1–8.

Kumar, S., Raghavan, V. S. and Deng, J. [2006], 'Medium Access Control protocols for ad hoc wireless networks: A survey', *Ad Hoc Networks* **4**(3), 326 – 358.
URL: *http://www.sciencedirect.com/science/article/B7576-4DPGSVH-1/2/58c0f2f528f8d27ecd834b2e92c21515*

Lau, T. H. and Chan, K. S. [2006], aMAC: Advanced MAC Scheme for Mobile Adhoc Networks, *in* 'Proc. Asia-Pacific Conference on Communications APCC '06', pp. 1–5.

Law, Y. W., van Hoesel, L., Doumen, J., Hartel, P. and Havinga, P. [2005], Energy-Efficient Link-Layer Jamming Attacks against Wireless Sensor Network MAC Protocols, *in* 'SASN '05: Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks', ACM, New York, NY, USA, pp. 76–88.

Lee, S. W. and Cho, D. H. [1995], Distributed Reservation CDMA for Wireless LAN, *in* 'Proc. IEEE Global Telecommunications Conference GLOBECOM '95', Vol. 1, pp. 360–364.

Li, M., Koutsopoulos, I. and Poovendran, R. [2007], Optimal Jamming Attacks and Network Defense Policies in Wireless Sensor Networks, *in* 'Proc. INFOCOM

2007. 26th IEEE International Conference on Computer Communications. IEEE', pp. 1307–1315.

Li, Z., Gupta, A. K. and Nandi, S. [n.d.], 'FMAC/CSR: A Fair MAC Protocol for Wireless Ad-hoc Networks'.
URL: *http://www.cs.jhu.edu/ zfli/fmac-csr.pdf*

Li, Z., Nandi, S. and Gupta, A. K. [2006], 'Modeling the Short-term Unfairness of IEEE 802.11 in Presence of Hidden Terminals', *Perform. Eval.* **63**(4), 441–462.

Liu, X., Noubir, G., Sundaram, R. and Tan, S. [2007], SPREAD: Foiling Smart Jammers Using Multi-Layer Agility, *in* 'Proc. INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE', pp. 2536–2540.

Muqattash, A. and Krunz, M. [2003], CDMA-Based MAC Protocol for Wireless Ad Hoc Networks, *in* 'MobiHoc '03: Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing', ACM, New York, NY, USA, pp. 153–164.

Nasipuri, A. and Das, S. R. [2000], Multichannel CSMA with Signal Power-Based Channel Selection for Multihop Wireless Networks, *in* 'Proc. 52nd Vehicular Technology Conference IEEE VTS-Fall VTC 2000', Vol. 1, pp. 211–218.

Nasipuri, A., Zhuang, J. and Das, S. R. [1999], A Multichannel CSMA MAC Protocol for Multihop Wireless Networks, *in* 'Proc. WCNC Wireless Communications and Networking Conference 1999 IEEE', pp. 1402–1406.

Navda, V., Bohra, A., Ganguly, S. and Rubenstein, D. [2007], Using Channel Hopping to Increase 802.11 Resilience to Jamming Attacks, *in* 'Proc. INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE', pp. 2526–2530.

Pursley, M. [1987], 'The Role of Spread Spectrum in Packet Radio Networks', *Proceedings of the IEEE* **75**(1), 116–134.

Razafindralambo, T. and Valois, F. [2006], Performance Evaluation of Backoff Algorithms in 802.11 Ad-Hoc Networks, *in* 'PE-WASUN '06: Proceedings of the 3rd ACM international workshop on Performance evaluation of wireless ad hoc, sensor and ubiquitous networks', ACM, New York, NY, USA, pp. 82–89.

So, J. and Vaidya, N. [2003], 'A Multi-Channel MAC Protocol for Ad Hoc Wireless Networks'.

URL: *citeseer.ist.psu.edu/so03multichannel.html*

Song, N.-O., Kwak, B.-J., Song, J. and Miller, M. [2003], 'Enhancement of IEEE 802.11 Distributed Coordination Function with Exponential Increase Exponential Decrease Backoff Algorithm', *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual* **4**, 2775–2778 vol.4.

Sousa, E. and Silvester, J. [1988], 'Spreading Code Protocols for Distributed Spread-Spectrum Packet Radio Networks', *Communications, IEEE Transactions on* **36**(3), 272–281.

Talucci, F. and Gerla, M. [1997], MACA-BI (MACA by invitation): A Wireless MAC Protocol for High Speed Ad Hoc Networking, *in* 'IEEE 6th International Conference on Universal Personal Communications Record Conference Record', Vol. 2, pp. 913–917.

Tang, Z. and Garcia-Luna-Aceves, J. J. [1998], Hop Reservation Multiple Access (HRMA) for Multichannel Packet Radio Networks, *in* 'Proc. 7th International Conference on Computer Communications and Networks', pp. 388–395.

Tobagi, F. and Kleinrock, L. [1976], 'Packet Switching in Radio Channels: Part III–Polling and (Dynamic) Split-Channel Reservation Multiple Access', *Communications, IEEE Transactions on* **24**(8), 832–845.

Tseng, Y.-C., Wu, S.-L., Lin, C.-Y. and Sheu, J.-P. [2001], A Multi-Channel MAC Protocol with Power Control for Multi-Hop Mobile Ad Hoc Networks, *in* 'Proc. International Conference on Distributed Computing Systems Workshop', pp. 419–424.

van Hoesel, L. F. W., Nieberg, T., Kip, H. J. and Havinga, P. J. M. [2004], Advantages of a TDMA based, energy-efficient, self-organizing MAC protocol for WSNs, *in* 'Proc. VTC 2004-Spring Vehicular Technology Conference 2004 IEEE 59th', Vol. 3, pp. 1598–1602.

Wang, P. and Zhuang, W. [2008], A Collision-Free MAC Scheme for Multimedia Wireless Mesh Backbone, *in* 'Proc. IEEE International Conference on Communications

ICC '08', pp. 4708–4712.

Wang, X. and Xiang, W. [2006], 'An OFDM-TDMA/SA MAC Protocol with QoS Constraints for Broadband Wireless LANs', *Wireless Networks* **12**(2), 159–170.
URL: *http://dx.doi.org/10.1007/s11276-005-5263-1*

Wong, C. Y., Cheng, R., Lataief, K. and Murch, R. [1999], 'Multiuser OFDM with adaptive subcarrier, bit, and power allocation', *Selected Areas in Communications, IEEE Journal on* **17**(10), 1747–1758.

Wu, C. and Li, V. [1988], Receiver-Initiated Busy-Tone Multiple Access in Packet Radio Networks, *in* 'SIGCOMM '87: Proceedings of the ACM workshop on Frontiers in computer communications technology', ACM, New York, NY, USA, pp. 336–342.

Wu, S.-L., Lin, C.-Y., Tseng, Y.-C. and Sheu, J.-L. [2000], A New Multi-Channel MAC Protocol with On-Demand Channel Assignment for Multi-Hop Mobile Ad Hoc Networks, *in* 'Proc. International Symposium on Parallel Architectures, Algorithms and Networks I-SPAN 2000', pp. 232–237.

Xu, W., Trappe, W., Zhang, Y. and Wood, T. [2005], The Feasibility of Launching and Detecting Jamming Attacks in Wireless Networks, *in* 'MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing', ACM, New York, NY, USA, pp. 46–57.

Yang, Z. and Garcia-Luna-Aceves, J. J. [1999], Hop-Reservation Multiple Access (HRMA) for Ad-Hoc Networks, *in* 'Proc. IEEE Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM '99', Vol. 1, pp. 194–201.

You, T., Yeh, C.-H. and Hassanein, H. [2003], CSMA/IC: A New Class of Collision-Free MAC Protocols for Ad Hoc Wireless Networks, *in* 'Proc. Eighth IEEE International Symposium on Computers and Communication (ISCC 2003)', pp. 843–848.

Zhai, H., Wang, J. and Fang, Y. [2006], 'DUCHA: A New Dual-Channel MAC Protocol for Multihop Ad Hoc Networks', *Wireless Communications, IEEE Transactions on* **5**(11), 3224–3233.

Zhai, H., Wang, J., Fang, Y. and Wu, D. [2004], A Dual-Channel MAC Protocol for Mobile Ad Hoc Networks, *in* 'Proc. IEEE Global Telecommunications Conference Workshops GlobeCom Workshops 2004', pp. 27–32.

Appendices

Appendix A

Source Code Listing

A.1 BBC-MAC Data Link Layer Code

A.1.1 Interface Class (interface.py)

```
'''
If you can find someone who can debug two million lines of code and interface
eight connection machines for what I bid for this job, I'd love to see him try
'''
import sys
import os
from stat import *
import threading
from subprocess import *
import Queue
import Receiver, Transmitter
import time, random
import ethernet_frame
import bbc_frame
from optparse import OptionParser
from gnuradio import gr, gru
from gnuradio import usrp
from gnuradio.eng_option import eng_option
from gnuradio import eng_notation
from gnuradio.eng_notation import num_to_str, str_to_num
from utilities import *
import bbc_config

class interface(threading.Thread):
    def __init__(self, usb, usrp_side, path, address, verbose, chat, dynamic, experiment_mode):
        threading.Thread.__init__(self)
        random.seed()
        self.name = "BBC-MAC Interface"
        self.mode = 1
        self.dynamic = dynamic
        self.experiment_mode = experiment_mode
        self.running = True
        self.config_change = False
        self.handlerQueue = Queue.Queue()
        self.tx_rx_pid = -1
        self.decoder_pid = -1
        self.rssi = 0
        self.address = address
        self.usb = usb
        self.usrp_side = usrp_side
        self.usrp_path = path
        self.verbose = verbose
        self.chat_mode = chat
        self.handlers = []
        self.interface_handler = None
        self.block_transmit = time.time()
        self.jammer_type = 0
        self.jammer_level = 0
        self.rcv_start = 0
        self.default_config = bbc_config.bbc_config(path)
        if self.dynamic:
            self.default_config.SetResistance(4092, 200)
        self.nav = 0.0
        self.transmitter = Transmitter.Transmitter(self.address, self)
        self.receiver = Receiver.Receiver(self.address, self)

    def ShutDown(self):
        self.mode = -1
        self.running = False
        try:
              os.kill(self.tx_rx_pid, 9)
        except:
              pass

        try:
            os.kill(self.decoder_pid, 9)
        except:
            pass

        self.receiver.ShutDown()
        for i in range(len(self.handlers)):
            try:
                self.handlers[i].Shutdown()
            except:
                pass

    def run(self):
        self.receiver.start()
        self.transmitter.start()

        while self.running:
            #This is our simple way of not doing anything until the nav expires
            if self.nav > 0:
                if self.verbose:
                    print "Deferring for",self.nav,"seconds."
                time.sleep(self.nav)
                self.nav = 0.0

            self.receive()

            if self.interface_handler == None:
                try:
                    self.interface_handler = self.handlerQueue.get(True, 1)
                    if self.verbose:
                        print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name,
                              self.interface_handler.name+" now owns the interface")
                    try:
                        if time.time() > self.block_transmit or self.experiment_mode == False:
                            frame = self.interface_handler.send_queue.get(True, 1)
                            self.transmit(frame, self.interface_handler)
                    except Queue.Empty:
                        pass
                except Queue.Empty:
                    pass
            else:
                try:
                    if time.time() > self.block_transmit or self.experiment_mode == False:
                        frame = self.interface_handler.send_queue.get(True, 1)
                        self.transmit(frame, self.interface_handler)
                except Queue.Empty:
                    pass

            self.mode = 1

    def transmit(self, frame, handler, tx_time=0):
        #dump payload to file
        frame.timestamp = time.time()
        f = open(self.usrp_path + "/t", "w")
        f.write(frame.serialize())
        f.close()

        try:
            os.kill(self.decoder_pid, 9)
```

```python
        except:
            pass

        try:
            os.kill(self.tx_rx_pid, 9)
        except:
            pass

        ret_code = call([self.usrp_path + "/usrp", handler.config.tx()], stdout=PIPE, stderr=PIPE)
        if tx_time == 0:
            tx_time = EstimateTransmitTime(len(frame.serialize()), handler.config)

        try:
            os.kill(ret_code.pid, 9)
        except:
            pass

        if frame.type == 4:
            tx_time*=1.1
        elif frame.type == 2: # or frame.type == 4:
            tx_time*=1.5

        #tx_time = 12.0
        #transmit
        if self.verbose:
            print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Radio transmitter started")

        ret_code = call([self.usrp_path + "/bbc_tx.py", "-U", self.usb, "-T", self.usrp_side, "-f", "1250M", "-i", "256", "-S", self.usrp_path + "/r", "-L", str(tx_time), "-J", str(self.jammer_type), "--jammer_level", str(self.jammer_level)], stdout=PIPE, stderr=PIPE)

        try:
            os.kill(ret_code.pid, 9)
        except:
            pass
        if self.verbose:
            print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Transmitted a frame")
            print frame

        #inform the handler that we sent the frame
        handler.Callback(frame, tx_time)
        return
    def SetJammerType(self, type):
        self.jammer_type = type

    def SetJammerLevel(self, level):
        self.jammer_level = level

    def receive(self):
        try:
            os.remove(self.usrp_path + "/t")
        except OSError:
            pass
        while self.running and self.mode == 1 and self.CheckInterfaceQueue() == False:
            try:
                os.remove(self.usrp_path + "/r")
            except OSError:
                pass

            self.tx_rx_pid = Popen([self.usrp_path + "/usrp_rx_cfile.py", "−U", self.usb, "−R",
                 self.usrp_side, "−f", "1250M", "−d", "128", "−N", "16000000", self.usrp_path+"/r"
                 ], stdout=PIPE,  stderr=PIPE).pid
            self.rcv_start = time.time()
            if self.tx_rx_pid != 0 and self.tx_rx_pid != None:
                if self.verbose:
                    print "%s %s: %s pid=%i" % (time.strftime("%H:%M:%S", time.gmtime()), self.
                         name, "Radio receiver started", self.tx_rx_pid)
            else:
                if self.verbose:
                    print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "
                         Unable to start radio receiver")


            while self.CheckReceiveExit():

                data = None
                if self.interface_handler != None:
                    data = self.Decode(self.interface_handler.config)
                    #'''
                    if self.interface_handler != None and data == None and self.interface_handler.
                         stage==1 and self.CheckReceiveExit():
                        #print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.
                             name, "Decoder timeout on handler's expansion, testing default")
                        data = self.Decode(self.default_config, 5.5)
                        if data == −1: #Don't necessarily want to trash the data because this
                             timed out
```

```
                                data = None
                    #'''
                else:
                    data = self.Decode(self.default_config)

                if data == -1:
                    print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Decoder preempted")
                    break

                elif data != None:
                    try:
                        os.remove(self.usrp_path + "/t")
                    except:
                        pass
                    self.rssi = GetRSSI(self.usrp_path)
                    #pass the data off to a receive handler
                    frame = bbc_frame.bbc_frame(data)
                    if self.verbose:
                        print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Received a frame")
                        print frame
                    self.receiver.Enqueue(frame)
                    break

            # Kill the Radio Receive
            try:
                os.kill(self.tx_rx_pid, 2)
                if self.verbose:
                    print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Radio receiver stopped")
            except:
                if self.verbose:
                    print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Radio receiver stopped")
                pass

    def Decode(self, config, timeout=30.0):
        if self.config_change:
            self.config_change = False
        #print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Decoder Start")
        self.decoder_pid = Popen([self.usrp_path + "/usrp", config.rx()], stdout=PIPE, stderr=PIPE).pid
        time_now = time.time()
        success = False
        while time.time() − time_now < timeout and self.CheckDecodeExit():

            try:
                pid, x = os.waitpid(self.decoder_pid,os.WNOHANG)
                if pid!=0:
                    success = True
                    break
            except:
                success = True
                break
            time.sleep(0.1)
        #print ”%s %s: %s” % (time.strftime(”%H:%M:%S”, time.gmtime()), self.name, ”Decoder Exit %
             is” % (time.time()−time_now))
        if success == False:
            try:
                os.kill(self.decoder_pid, 9)
            except:
                pass
            #return −1

        try:
            # Check for file existence, open stats file anyways, no need for two steps
            f = open(self.usrp_path + ”/t”, ”r”)
            data = f.read()
            f.close()
            return data
        except IOError:
            if success == False:
                return −1
            else:
                return None

    def CheckReceiveExit(self):
        return self.CheckInterfaceQueue() == False and self.mode == 1 and CheckPID(self.tx_rx_pid)
             and self.running and time.time() − self.rcv_start < 33

    def InformConfigChange(self):
        self.config_change = True

    def CheckDecodeExit(self):
        return self.CheckInterfaceQueue() == False and self.mode == 1 and self.running and self.
             config_change == False

    def CheckInterfaceQueue(self):
        if self.interface_handler == None:
            return self.handlerQueue.empty() == False
        else:
            return self.interface_handler.send_queue.empty() == False

    def RelinquishInterfaceControl(self, handler, time_left):
        self.block_transmit = time_left
        self.interface_handler = None

    def UpdateNAV(self, timer):
        if self.interface_handler == None: #Make sure someone doesn't already own the interface before setting the NAV
            self.nav = timer
            self.mode = 0

    def Enqueue(self, handler):
        if self.verbose:
            print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, handler.name +" added to handler queue")
        self.handlerQueue.put(handler)
        #self.mode = 0

    def LocateHandler(self,streamid):
        for i in range(len(self.handlers)):
            if self.handlers[i].streamid == streamid:
                return self.handlers[i]
        return None

    def LocateHandlerBySource(self,sstreamid):
        for i in range(len(self.handlers)):
            if self.handlers[i].destination_streamid == sstreamid:
                return self.handlers[i]
        return None

    def GetNewStreamID(self):
        return random.randint(1,65535)


def main():

    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-X", "--txrx_subdev_spec", type="string", default="A", dest="side",
                          help="select USRP TxRx side A or B")
    parser.add_option("-U", "--usb_num", type="string", default=0,
                          help="select USRP USB location 0 or 1 (default=0)")
    parser.add_option("-A", "--node_address", type="int", default=None, help="Address for this node")
    parser.add_option("-P", "--usrp_path", type="string", default=None, help="path to usrp folder with scripts")
    parser.add_option("-C", "--chat_mode", action="store_true")
    parser.add_option("-J", "--jammer_type", type="int", default=0)
    parser.add_option ("--jammer_level", type="eng_float", default=16e3,
                        help="set waveform amplitude to AMPLITUDE [default=%default]", metavar="AMPL")
    parser.add_option("-v", "--verbose", action="store_true", dest="verbose",
                  help="print everything to stdout")
    parser.add_option("--dynamic", action="store_true", default=False, help="Enable dynamic jam-resistance")
    parser.add_option("--experiment", action="store_true", default=False, help="Used to leave handlers running for as long as the channel was allocated")

    (options, args) = parser.parse_args ()
    f = open(options.usrp_path+"/ftp_frame_1514.fr")
    data = f.read()
    f.close()

    i = interface(options.usb_num, options.side, options.usrp_path, options.node_address, options.verbose, options.chat_mode, options.dynamic, options.experiment)
    i.start()

    while True:
        cmd = raw_input()
        cmds = cmd.split(" ")
        if cmd == "exit":
            i.ShutDown()
            raise SystemExit
        elif cmds[0] == "send":
            i.transmitter.Enqueue(data, 2222)
        elif cmds[0] == "kick":
            i.interface_handler.Shutdown()
        elif cmds[0] == "jam":
            i.SetJammerType(int(cmds[1]))
            i.SetJammerLevel(int(cmds[2]))

if __name__ == "__main__":
    main()
```

A.1.2 Receiver Class (Receiver.py)

```
import threading
import Queue
import bbc_frame
import RxHandler
import sys
import time

class Receiver(threading.Thread):
    def __init__(self, address, interface):
        threading.Thread.__init__(self)
        self.handlers = [] # This should also include TxHandlers in order to properly give them their CTS/ACK Frames
        self.queue = Queue.Queue()
        self.running = 1
        self.address = address
        self.interface = interface
        self.name = ”Receiver”

    def run(self):
        while self.running:
            try:
                frame = self.queue.get(True,1)
                if frame.dstream_id == 0:
                    #Check to see if this a duplicate and if the source streamid matches the destination of a handler
                    handler = self.interface.LocateHandlerBySource(frame.sstream_id)
                    if handler != None:
                        handler.Enqueue(frame)
                    else:
                        temp = RxHandler.RxHandler(frame, self.interface.GetNewStreamID(), self.Callback, self.address, self.interface)
                        self.interface.handlers.append(temp)
                        temp.start()
                else:
                    handler = self.interface.LocateHandler(frame.dstream_id)
                    if handler != None:
                        handler.Enqueue(frame)
                    else:
                        temp = RxHandler.RxHandler(frame, frame.dstream_id, self.Callback, self.address, self.interface)
                        self.interface.handlers.append(temp)
                        temp.start()
            except Queue.Empty:
                pass

    def Enqueue(self, frame):
        self.queue.put_nowait(frame)

    def ShutDown(self):
        self.running = 0


    def Callback(self, obj, data=None):
        print obj.stats
        if data!=None:
            print ”%s %s: %s” % (time.strftime(”%H:%M:%S”, time.gmtime()), self.name, obj.name+”
                 delivered data from stream ”+str(obj.streamid))
        try:
            if self.interface.interface_handler == obj:
                print ”%s %s: %s” % (time.strftime(”%H:%M:%S”, time.gmtime()), self.name, ”removed
                     ”+obj.name+” as interface handler”)
                self.interface.interface_handler = None
                self.interface.InformConfigChange()
            self.interface.handlers.remove(obj)
            del obj
        except:
            print ”Unexpected error:”, sys.exc_info()[0]
            pass
```

A.1.3 Receiver Handler Class (RxHandler.py)

```
import Queue
import bbc_frame
import ethernet_frame
import threading
import bbc_config
from utilities import *
from stats import *
import time

class RxHandler(threading.Thread):
    def __init__(self, frame, streamid, callback, address, interface):
        threading.Thread.__init__(self)
        self.streamid = streamid
        self.callback = callback
        self.recv_queue = Queue.Queue()
        self.send_queue = Queue.Queue()
        self.running = True
        self.address = address
        self.interface = interface
        self.stats = RxStats()
        self.stage = 0
        self.name = "RxHandler "+str(self.streamid)
        self.rssi = None
        self.data = None
        self.config = bbc_config.bbc_config(self.interface.usrp_path)
        self.config.SOURCE_ID = self.streamid
        if self.interface.dynamic:
            self.config.SetResistance(4092, 175)
        self.destination_address = frame.src_addr
        self.destination_streamid = frame.sstream_id
        self.flag = 0
        self.last_frame = None
        self.t1 = 0
        self.timeout = 0
        self.Enqueue(frame)

    def Enqueue(self, frame):
        if self.last_frame == None:
            self.last_frame = frame
        elif frame.timestamp <= self.last_frame.timestamp:
            if self.interface.verbose:
                print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Frame discarded (old or duplicate)")
            return
        elif frame.corrupt:
            if self.interface.verbose:
                print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Frame discarded (corrupt)")
            return
        self.last_frame = frame
        self.recv_queue.put_nowait(frame)

    def Shutdown(self):
        self.running = False
        #self.callback(self, self.data)

    def run(self):
        while self.running:
            try:
                frame = self.recv_queue.get(True,1)
                if frame.type == 1:
                    #Received a RTS, Check to see if this was for us.
                    if frame.dest_addr == self.address:
                        self.destination_address = frame.src_addr
                        self.destination_streamid = frame.sstream_id
                        data_len = int(frame.payload)

                        self.rssi = self.interface.rssi
                        if frame.rssi > self.rssi:
                            self.rssi = frame.rssi

                        self.timeout = EstimateChannelTime(data_len,self.config,self.config.GetExpansionByRSSI(self.rssi))

                        if self.interface.dynamic:
                            self.config.SetResistance(4092, 175)
                        new_frame = bbc_frame.bbc_frame((self.destination_address, self.address, 2, self.streamid, self.destination_streamid, self.rssi, self.timeout))
                        #return a CTS with our current RSSI
                        diff = frame.timestamp + EstimateTransmitTime(len(frame.serialize()), self.config) + 3 - time.time()
                        #while time.time() < frame.timestamp+EstimateTransmitTime(len(frame.serialize()), self.config)+2:#14.0: #Hack so I'm not transmitting while they're still transmitting this frame
                        #     continue
                        if diff > 0.0:
                            time.sleep(diff)
                        self.config.SOURCE_ID = self.streamid
                        self.send_queue.put_nowait(new_frame)
                        if self.flag == 0:
                            self.interface.Enqueue(self)
                            self.flag = 1
                        self.stats.rts_count+=1
                elif frame.type == 2:
                    #Received a CTS, Extract the time value and update the NAV timer
                    t = float(frame.payload)
                    self.interface.UpdateNAV(t)
                    self.running = False
                elif frame.type == 3:
                    #Received Data
                    if frame.dest_addr == self.address:
                        self.stage = 0
                        self.destination_address = frame.src_addr
                        self.destination_streamid = frame.sstream_id
                        new_frame = bbc_frame.bbc_frame((self.destination_address, self.address, 4, self.streamid, self.destination_streamid,self.rssi, ”ACK”))
                        #return a ACK
                        diff = frame.timestamp + EstimateTransmitTime(len(frame.serialize()), self.config) + 3 - time.time()
```

```
                        #while time.time() < frame.timestamp+EstimateTransmitTime(len(frame.serialize()), self.config)+3:#14.0: #Hack so I'm not transmitting while they're still transmitting this frame
                        #    continue
                        if diff > 0.0:
                            time.sleep(diff)
                        self.config.SOURCE_ID = self.streamid + 1
                        if self.interface.dynamic:
                            self.config.SetResistance(4092, 175)
                        self.send_queue.put_nowait(new_frame)
                        if self.flag == 0:
                            self.interface.Enqueue(self)
                            self.flag = 1
                        self.data = frame.payload
                        if self.stats.data_count==0:
                            self.stats.data_time = time.time()
                        self.stats.data_count+=1
                    elif frame.type == 4:
                        #Received an ACK
                        #this should have been given to a TxHandler, but one doesn't exist, so ignore
                        self.running = False
                except Queue.Empty:
                    if self.t1 !=0:
                        if time.time() - self.t1 > self.timeout:
                            self.running = False
                    pass
            self.callback(self, self.data)

    def Callback(self, frame, tx_time):

        if frame.type == 2: #Sent out the CTS, now adjust our config and start the timer
            if self.interface.dynamic:
                self.config.SetResistance(self.rssi)
                self.interface.InformConfigChange()
            #if self.t1 == 0:
            if self.interface.verbose:
                print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Reserved channel for "+str(self.timeout)+" seconds.")
            self.t1 = time.time()
            self.stats.cts_count+=1
            self.stage = 1
        elif frame.type == 4:
            self.stats.ack_count+=1
            if self.interface.dynamic:
                self.config.SetResistance(self.rssi)
                self.interface.InformConfigChange()

        #if frame.type == 4:
        #     self.running = False
```

A.1.4 Transmitter Class (Transmitter.py)

```
import threading
import Queue
import bbc_frame
import TxHandler
import time
import sys

class Transmitter(threading.Thread):
    def __init__(self, address, interface):
        threading.Thread.__init__(self)
        self.address = address
        self.interface = interface
        self.queue = Queue.Queue()
        self.running = True
        self.name = "Transmitter"

    def run(self):
        while self.running:
            try:
                payload, destination = self.queue.get(True,1)
                temp = TxHandler.TxHandler(self.interface.GetNewStreamID(), self.address, self.Callback, payload, destination, self.interface)
                self.interface.handlers.append(temp)
            except Queue.Empty:
                pass

    def Enqueue(self, payload, destination):
        self.queue.put_nowait((payload, destination))

    def Callback(self, obj, message=None, time_left=0.0):
        print obj.stats
        if message!=None:
            print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, obj.name+" reports that stream "+str(obj.streamid)+" was a "+message)
        try:
            if self.interface.interface_handler == obj:
                print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "removed "+obj.name+" as interface handler")
                self.interface.RelinquishInterfaceControl(obj, time_left)
                #self.interface.interface_handler = None
                self.interface.InformConfigChange()

            self.interface.handlers.remove(obj)
            del obj
        except:
            print "Unexpected error:", sys.exc_info()[0]
            pass
```

A.1.5 Transmitter Handler Class (TxHandler.py)

```
import Queue
import bbc_frame
import bbc_config
import ethernet_frame
import thread
import time
from utilities import *
from stats import *

class TxHandler:
    def __init__(self, streamid, address, callback, data, destination, interface):
        self.address = address
        self.destination_address = destination
        self.destination_streamid = 0
        self.streamid = streamid
        self.callback = callback
        self.recv_queue = Queue.Queue()
        self.send_queue = Queue.Queue()
        self.interface = interface
        self.stats = TxStats(data)
        self.name = "TxHandler "+str(self.streamid)
        self.rssi = None
        self.data = data
        self.config = bbc_config.bbc_config(self.interface.usrp_path)
        self.config.SOURCE_ID = self.streamid
        if self.interface.dynamic:
            self.config.SetResistance(4092, 175)
        self.running = True
        rts_frame = self.CreateRTS()
        self.send_queue.put_nowait(rts_frame) #Enqueue the initial frame out outbound queue
        self.interface.Enqueue(self) #Enqueue our handle in the interface
        self.stats.rts_count+=1
        self.stage = 0
        self.rtx_count = 0
        self.thread_id = None
        self.last_frame = None
        self.timeout = 0
        self.t1 = 0
        #print "Created",self.name

    def Enqueue(self, frame):
        if self.last_frame == None:
            self.last_frame = frame
        elif frame.timestamp <= self.last_frame.timestamp:
            if self.interface.verbose:
                print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Frame discarded (old or duplicate)")
            return
        elif frame.corrupt:
            if self.interface.verbose:
                print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name, "Frame discarded (corrupt)")
            return
        self.last_frame = frame
        self.recv_queue.put_nowait(frame)

    def Shutdown(self, message=None):
        try:
            self.thread_id.exit()
        except:
            pass
        self.callback(self, message, self.t1 + self.timeout)

    def Callback(self, frame, tx_time):
        if frame.type==1 and self.stats.rts_count==1:
            self.stats.send_time = time.time()-tx_time #This is the very first callback for the actual transmit
        if frame.type == 3:
            if self.interface.dynamic:
                self.config.SetResistance(4092, 175)
                self.interface.InformConfigChange()

        self.thread_id = thread.start_new_thread(self.thread,(frame,tx_time))

    def thread(self, frame, tx_time):
        if frame.type == 1: #We are waiting for a CTS
            try:
```

```
rcv_frame = self.recv_queue.get(True, 30) #Easy way to do a timer, use the queue timeout
if rcv_frame.type == 2: #We got the CTS, send out Data
    self.destination_streamid = rcv_frame.sstream_id
    if rcv_frame.rssi > self.rssi:
        self.rssi = rcv_frame.rssi

    tmp_rssi = self.interface.rssi

    self.stats.rssi = self.rssi
    diff = rcv_frame.timestamp + 1.5*EstimateTransmitTime(len(rcv_frame.serialize()), self.config) + 3 - time.time()
    #while time.time() < rcv_frame.timestamp+EstimateTransmitTime(len(rcv_frame.serialize()), self.config)+3:#14.0: #Hack so I'm not transmitting while they're still transmitting this frame
    #     continue
    if diff > 0.0:
        time.sleep(diff)

    if self.interface.dynamic:
        self.config.SetResistance(self.rssi)

    if self.interface.dynamic and self.config.GetExpansionByRSSI(tmp_rssi) > self.config.CODEC_EXPANSION and self.rtx_count < 2: #catch it early and re-transmit the RTS
        self.config.SetResistance(4092, 175)
        rts_frame = self.CreateRTS(tmp_rssi)
        if self.interface.verbose:
            print ”%s %s: %s” % (time.strftime(”%H:%M:%S”, time.gmtime()), self.name, ”Expansion adjustment needed, re-sending RTS”)
        self.send_queue.put_nowait(rts_frame)
        self.stats.rts_count+=1
        self.rtx_count+=1
    else:
        self.timeout = float(rcv_frame.payload)
        if self.interface.verbose:
            print ”%s %s: %s” % (time.strftime(”%H:%M:%S”, time.gmtime()), self.name, ”Reserved channel for ”+str(self.timeout)+” seconds.”)
        if self.t1 == 0:
            self.t1 = time.time()
        self.stats.expansion = self.config.CODEC_EXPANSION
        self.config.SOURCE_ID = self.streamid + 1
        data_frame = self.CreateDataFrame()
        self.send_queue.put_nowait(data_frame)
        self.stats.data_count+=1
```

```
                        self.rtx_count = 0
            except Queue.Empty:
                if self.rtx_count < 2:
                    self.config.SOURCE_ID = self.streamid
                    rts_frame = self.CreateRTS()
                    self.send_queue.put_nowait(rts_frame)
                    if self.interface.verbose:
                        print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name,
                            "CTS timeout, re-sending RTS")
                    self.stats.rts_count+=1
                    self.rtx_count+=1
                else: #3x is max retransmit time to die
                    if self.interface.verbose:
                        print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name,
                            "RTS retransmission maxed, giving up")
                    #signal upper layer
                    self.stats.rts_fail = True
                    self.Shutdown("failure")
        elif frame.type == 3: #We are waiting for an ACK
            try:
                rcv_frame = self.recv_queue.get(True, 30 ) #Easy way to do a timer, use the queue
                    timeout
                if rcv_frame.type == 4:
                    if self.interface.verbose:
                        print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name,
                            "received ack")
                    self.stats.ack_time = time.time()
                    self.stats.latency = self.stats.ack_time - self.stats.send_time
                    self.Shutdown("success")
            except Queue.Empty:
                if self.rtx_count < 2 and time.time() - self.t1 < self.timeout:
                    if self.interface.verbose:
                        print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name,
                            "ACK time out, re-sending data frame")
                    self.config.SOURCE_ID = self.streamid + 1
                    data_frame = self.CreateDataFrame()
                    if self.interface.dynamic:
                        self.config.SetResistance(self.rssi)
                    self.send_queue.put_nowait(data_frame)
                    #self.interface.Enqueue(self.CreateDataFrame(), self)
                    self.rtx_count+= 1
                    self.stats.data_count+=1
                else: #3x is max retransmit time to die
                    if self.interface.verbose:
                        print "%s %s: %s" % (time.strftime("%H:%M:%S", time.gmtime()), self.name,
                            "Exceeded channel allocation, giving up")
                    #Signal upper layer
                    self.stats.data_fail = True
                    self.Shutdown("failure")



    def CreateRTS(self, rssi=None):
        if rssi==None:
            self.rssi = GetRSSI(self.interface.usrp_path)
        else:
            self.rssi = rssi
        # Create a config based on the RSSI we have and use it for the estimation
        #self.timeout = EstimateChannelTime(self.data, self.config)
        return bbc_frame.bbc_frame((self.destination_address, self.address, 1, self.streamid, self
            .destination_streamid, self.rssi, str(len(self.data))))

    def CreateDataFrame(self):
        return bbc_frame.bbc_frame((self.destination_address, self.address, 3, self.streamid, self
            .destination_streamid, self.rssi, self.data))

    def Encode(self, frame):
        return
```

A.1.6 BBC Config Class (bbc_config.py)

```
from cStringIO import StringIO

class bbc_config:
    def __init__(self, path):
        self.DIAGNOSTICS = False
        self.PATH = path

        # SCHEDULER Configuration
        self.SCHEDULER_TX_notRX = 0
        self.SCHEDULER_REALTIME = 0

        # SOURCE Configuration
        self.SOURCE_NAME = "r"
        self.SOURCE_ID = 1

        # CODEC Configuration
        self.CODEC_MESSAGE_BITS = 512
        self.CODEC_RANDOM_BITS = 8
        self.CODEC_CLAMP_BITS = 1
        self.CODEC_FRAGMENT_BITS = 1
        self.CODEC_STOP_BITS = 100
        self.CODEC_EXPANSION = 500
        self.CODEC_PACKET_LOAD = 2
        self.CODEC_DECODE_LIMIT = 2

        # BUFFER Configuration
        self.BUFFER_PACKETS = 4.0
        self.BUFFER_LAMBDA = 0.4

        # MODEM Configuration
        self.MODEM_PACKET_RATE_BPS = 500000
        self.MODEM_SAMPLES_PER_BIT = 4
        self.MODEM_GAIN_DB = 80.0
        self.MODEM_CHANNEL_LOSS_DB = 8.0
        self.MODEM_THRESHOLD_PCT = 46.3744
        self.MODEM_HYSTERESIS_PCT = 5.0
        self.MODEM_JITTER_BITS = 2.0
        self.MODEM_CUSHION_PCT = 10.0

        # SINK Configuration
        self.SINK_NAME = "t"
        self.SINK_SAMPLE_LIMIT = 16000000

    def tx(self):
        self.SOURCE_NAME = "t"
        self.SINK_NAME = "r"
        self.MODEM_CHANNEL_LOSS_DB = 3.0
        self.SCHEDULER_TX_notRX = 1
        f = open(self.PATH+"/tx.ini", "wt")
        f.write(self.format())
        f.close()
        return self.PATH+"/tx.ini"

    def rx(self):
        self.SOURCE_NAME = "r"
        self.SINK_NAME = "t"
        self.MODEM_CHANNEL_LOSS_DB = 16.0
        self.SCHEDULER_TX_notRX = 0
        f = open(self.PATH+"/rx.ini", "wt")
        f.write(self.format())
        f.close()
        return self.PATH+"/rx.ini"

    def format(self):
        s = StringIO()
        for k,v in self.__dict__.items():
            if str(k) == "DIAGNOSTICS":
                if v:
                    s.write("DIAGNOSTICS\n")
            elif str(k) == "PATH" or str(k) == "SINK_NAME" or str(k) == "SOURCE_NAME":
                s.write("%s=\"%s\"\n" % (k,v))
            else:
                s.write('%s=%s\n' % (k,v))
        return s.getvalue()

    def SetResistance(self, rssi, value=None):
        if value!=None:
            self.CODEC_EXPANSION = value
            return

        self.CODEC_EXPANSION = self.GetExpansionByRSSI(rssi)

    def GetExpansionByRSSI(self, rssi):
        if rssi <= 350:
            return 50
        elif rssi <= 700:
            return 75
        elif rssi <= 1050:
            return 100
        elif rssi <= 1350:
            return 150
        else:
            return 175
```

A.1.7 BBC-MAC Frame Class (bbc_frame.py)

```
import struct
import time
from crc16 import *

class bbc_frame:
  def __init__(self, raw_frame):
    self.types = (1,2,3,4) # RTS CTS DATA ACK
    self.timestamp = time.time()
    if isinstance(raw_frame,str):
        self.raw_frame = raw_frame
        try:
            self.dest_addr, self.src_addr, self.type, self.sstream_id, self.dstream_id, self.rssi, self.crc, self.timestamp = struct.unpack("!HHBHHHHd", raw_frame[:21])
            self.payload = raw_frame[21:]
            if self.crc != crc16(str(self.payload)):
                self.corrupt = True
            else:
                self.corrupt = False
        except:
            self.corrupt = True
    else:
        self.dest_addr, self.src_addr, self.type, self.sstream_id, self.dstream_id, self.rssi, self.payload = raw_frame
        self.crc = crc16(str(self.payload))
        #self.raw_frame = self.serialize()

  def toString(self):
      s = "Destination: "+str(self.dest_addr)+"\nSource: "+str(self.src_addr)+"\nType: "+str(self.type)+"\nSource StreamID: "+str(self.sstream_id)+"\nDestination StreamID: "+str(self.dstream_id)+"\nRSSI: "+str(self.rssi)+"\nCRC: "+str(self.crc)+"\nTimestamp: "+str(self.timestamp)#+"\nPayload:\n"+str(self.payload)
      return s

  def __repr__(self):
      return self.toString()

  def serialize(self):
      return struct.pack("!HHBHHHHd", self.dest_addr, self.src_addr, self.type, self.sstream_id, self.dstream_id, self.rssi, self.crc, self.timestamp) + str(self.payload)

  def size(self):
      return len(self.serialize())
```

A.1.8 Utilities Class (utilities.py)

```
import os
import bbc_config
from crc16 import *
from math import *
crc = CRC16()

def GetRSSI(path):
    while True:
        try:
            f = open(path+"/rssi", "rt")
            rssi = int(f.read())
```

```
                f.close()
                break
            except:
                continue
    return rssi

def EstimateChannelTime(data_len, config, expansion=None):
    t = 3*(EstimateTransmitTime(data_len, config, expansion)+30) #+ 12 + 18 + 12#Worst cast estimation, ACK and RTS frames are same size
    return t

def EstimateTransmitTime(data_len, config, expansion=None):
    if expansion == None:
        expansion = config.CODEC_EXPANSION
    one = ceil( (data_len/((config.CODEC_MESSAGE_BITS/8.0) - 10.0))/config.CODEC_PACKET_LOAD  )
    two = ((config.CODEC_MESSAGE_BITS*expansion)/8)*config.BUFFER_LAMBDA
    three = ((config.CODEC_MESSAGE_BITS*expansion)/8) + 1
    res = ceil(4*(((one*two + three)*8*4)/config.MODEM_PACKET_RATE_BPS))
    return res

def CheckPID(pid):
    try:
        os.kill(pid, 0)
        return True
    except:
        return False

def StatFileSize(path):
    try:
        return os.stat(path).st_size
    except:
        return 0

def GetCRC(data):
    crc.update(str(data))
    return crc.checksum()
```

A.1.9 CRC16 Class (crc16.py)

```
# crc16.py by Bryan G. Olson, 2005
# This module is free software and may be used and
# distributed under the same terms as Python itself.

"""
     CRC-16 in Python, as standard as possible. This is
```

```
    the 'reflected' version, which is usually what people
    want. See Ross N. Williams' /A Painless Guide to
    CRC error detection algorithms/.
"""

from array import array


def crc16(string, value=0):
    """ Single-function interface, like gzip module's crc32
    """
    for ch in string:
        value = table[ord(ch) ^ (value & 0xff)] ^ (value >> 8)
    return value


class CRC16(object):
    """ Class interface, like the Python library's cryptographic
        hash functions (which CRC's are definitely not.)
    """

    def __init__(self, string=''):
        self.val = 0
        if string:
            self.update(string)

    def update(self, string):
        self.val = crc16(string, self.val)

    def checksum(self):
        return chr(self.val >> 8) + chr(self.val & 0xff)

    def hexchecksum(self):
        return '%04x' % self.val

    def copy(self):
        clone = CRC16()
        clone.val = self.val
        return clone


# CRC-16 poly: p(x) = x**16 + x**15 + x**2 + 1
# top bit implicit, reflected
poly = 0xa001
table = array('H')
```

```
for byte in range(256):
    crc = 0
    for bit in range(8):
        if (byte ^ crc) & 1:
            crc = (crc >> 1) ^ poly
        else:
            crc >>= 1
        byte >>= 1
    table.append(crc)
```

A.1.10 Stats Module (stats.py)

```
from cStringIO import StringIO


class TxStats:
    def __init__(self, data):
        self.raw_data = data
        self.rts_count = 0
        self.data_count = 0
        self.rts_fail = False
        self.data_fail = False
        self.send_time = 0
        self.ack_time = 0
        self.rssi = 0
        self.expansion = 0
        self.latency = 0

    def __repr__(self):
        return self.toString()

    def toString(self):
        s = StringIO()
        for k,v in self.__dict__.items():
            if str(k) == "raw_data":
                continue
            s.write('%s\t' % k)
        s.write('\n')

        for k,v in self.__dict__.items():
            if str(k) == "raw_data":
                continue
            s.write('%s\t' % v)
        s.write('\n')
        return s.getvalue()

class RxStats:
    def __init__(self):
        self.rts_count = 0
        self.data_count = 0
        self.ack_count = 0
        self.cts_count = 0
        self.data_time = 0

    def __repr__(self):
        return self.toString()

    def toString(self):
        s = StringIO()
        for k,v in self.__dict__.items():
            if str(k) == "raw_data":
                continue
            s.write('%s\t' % k)
        s.write('\n')

        for k,v in self.__dict__.items():
            if str(k) == "raw_data":
                continue
            s.write('%s\t' % v)
        s.write('\n')

        return s.getvalue()
```

A.2 Radio Scripts Code

A.2.1 USRP Receiver Script (usrp_rx_cfile.py)

```
#!/usr/bin/env python

"""
Read samples from the USRP and write to file formatted as binary
outputs single precision complex float values or complex short values (interleaved 16 bit signed short integers).

"""

from gnuradio import gr, gru, eng_notation
#from gnuradio import audio
```

```
from gnuradio import usrp
from gnuradio.eng_option import eng_option
from optparse import OptionParser
from usrpm import usrp_dbid
import time
import sys
import thread

class my_graph(gr.flow_graph):
#class my_graph(gr.top_block):

    def __init__(self):
        gr.flow_graph.__init__(self)
        #gr.top_block.__init__(self)
        self.rssi = 0

        usage=”%prog: [options] output_filename output_filename2”
        parser = OptionParser(option_class=eng_option, usage=usage)
        parser.add_option(”-R”, ”--rx-subdev-spec”, type=”subdev”, default=(0, 0),
                          help=”select USRP Rx side A or B (default=A)”)
        parser.add_option(”-U”, ”--usb_num”, type=”int”, default=0,
                          help=”select USRP USB location 0 or 1 (default=0)”)
        parser.add_option(”-d”, ”--decim”, type=”int”, default=16,
                          help=”set fgpa decimation rate to DECIM [default=%default]”)
        parser.add_option(”-f”, ”--freq”, type=”eng_float”, default=None,
                          help=”set frequency to FREQ”, metavar=”FREQ”)
        parser.add_option(”-g”, ”--gain”, type=”eng_float”, default=None,
                          help=”set gain in dB (default is midpoint)”)
        parser.add_option(”-8”, ”--width-8”, action=”store_true”, default=False,
                          help=”Enable 8-bit samples across USB”)
        parser.add_option( ”--no-hb”, action=”store_true”, default=False,
                          help=”don't use halfband filter in usrp”)
        parser.add_option( ”-s”,”--output-shorts”, action=”store_true”, default=False,
                          help=”output interleaved shorts in stead of complex floats”)
        parser.add_option(”-N”, ”--nsamples”, type=”eng_float”, default=None,
                          help=”number of samples to collect [default=+inf]”)
        parser.add_option(”-C”, ”--nchan”, type=”int”, default=1,
                          help=”set number of channels to use (RX on both daughterboards)”)
        (options, args) = parser.parse_args ()

        if len(args) < 1:
            parser.print_help()
            raise SystemExit, 1

        #with multiple channels, need multiple files for receiver sinks so both receivers are not
        #      writing to the same file on the driver computer
        if options.nchan > 1:
            filename_A = args[0]
            filename_B = args[1]
        else:
            filename = args[0]

        self.fn = filename
        if options.freq is None:
            parser.print_help()
            sys.stderr.write('You must specify the frequency with -f FREQ\n');
            raise SystemExit, 1

        if options.no_hb or (options.decim<8):
            self.fpga_filename="std_4rx_0tx.rbf" #Min decimation of this firmware is 4. contains 4 Rx paths without halfbands and 0 tx paths.
            if options.output_shorts:
                self.u = usrp.source_s(which=options.usb_num,decim_rate=options.decim, fpga_filename=self.fpga_filename)
            else:
                self.u = usrp.source_c(which=options.usb_num,decim_rate=options.decim, fpga_filename=self.fpga_filename)
        else:
            #standard fpga firmware "std_2rxhb_2tx.rbf" contains 2 Rx paths with halfband filters and 2 tx paths (the default) min decimation 8
            if options.output_shorts:
                self.u = usrp.source_s(which=options.usb_num,decim_rate=options.decim)
            else:
                self.u = usrp.source_c(which=options.usb_num,decim_rate=options.decim)

        #use more than 1 channel if specified
        #this will allow a USRP to TX or RX on both daughterboards simultaneously
        if options.nchan > 1:
            nchan = options.nchan
            if self.u.nddc() < nchan:
                sys.stderr.write('This code requires an FPGA build with %d DDCs.  This FPGA has only %d.\n' % (nchan, self.u.nddc()))
                raise SystemExit

            if not self.u.set_nchannels(nchan):
                sys.stderr.write('set_nchannels(%d) failed\n' % (nchan,))
                raise SystemExit

            #self.subdev = self.u.db[0] + self.u.db[1]

            self.subdev = (self.u.db[0][0], self.u.db[1][0])

            print "Using RX daughterboard %s" % (self.subdev[0].side_and_name(),)
            print "Using RX daughterboard %s" % (self.subdev[1].side_and_name(),)

            if options.gain is None:
                g_A = self.subdev[0].gain_range()
                options.gain = float(g_A[0]+g_A[1])/2

            #use the same gain for both sides
            self.subdev[0].set_gain(options.gain)
            self.subdev[1].set_gain(options.gain)
            # r = usrp.tune(self.u, i, self.subdev[i], target_freq)
            r_A = self.u.tune(0,self.subdev[0],options.freq)
            if not r_A:
                sys.stderr.write('Failed to set frequency for RX daughterboard %s\n' % (self.
                    subdev[0].side_and_name()))
                raise SystemExit, 1

            r_B = self.u.tune(1,self.subdev[1],options.freq)
            if not r_B:
                sys.stderr.write('Failed to set frequency for RX daughterboard %s\n' % (self.
                    subdev[1].side_and_name()))
                raise SystemExit, 1

        else:
            # using only 1 channel in this case
            # determine the daughterboard subdevice we're using per argument list
            self.subdev = usrp.selected_subdev(self.u, options.rx_subdev_spec)
            print "Using RX daughterboard %s" % (self.subdev.side_and_name(),)

            #set the gain
            if options.gain is None:
                # if no gain was specified, use the mid-point in dB
                g = self.subdev.gain_range()
                options.gain = float(g[0]+g[1])/2

            self.subdev.set_gain(options.gain)

            r = self.u.tune(0, self.subdev, options.freq)
            if not r:
                sys.stderr.write('Failed to set frequency\n')
                raise SystemExit, 1

        if options.width_8:
            sample_width = 8
            sample_shift = 8
            format = self.u.make_format(sample_width, sample_shift)
            r = self.u.set_format(format)
        if options.output_shorts:
            #default value is fine here for multiple channels since
            #we will be using complex floats
            self.dst = gr.file_sink(gr.sizeof_short, filename)
        else:
            if options.nchan == 1:
                self.dst = gr.file_sink(gr.sizeof_gr_complex, filename)
            else:
                #establish separate file sinks for the two channels
                self.dst_A = gr.file_sink(gr.sizeof_gr_complex, filename_A)
                self.dst_B = gr.file_sink(gr.sizeof_gr_complex, filename_B)

        if options.nsamples is None:#this is the default
            if options.nchan == 1:
                self.connect(self.u, self.dst)
            else:  #multiple channels
                di = gr.deinterleave(gr.sizeof_gr_complex)
                self.connect(self.u, di)
                self.connect((di,0),self.dst_A)
                self.connect((di,1),self.dst_B)

        else:
            if options.output_shorts:
              self.head = gr.head(gr.sizeof_short, int(options.nsamples)*2)
            else:
              self.head = gr.head(gr.sizeof_gr_complex, int(options.nsamples))
            self.connect(self.u, self.head, self.dst)

        if options.rx_subdev_spec is None:
            options.rx_subdev_spec = usrp.pick_rx_subdevice(self.u)
        self.rx_subdev = options.rx_subdev_spec

        self.u.set_mux(usrp.determine_rx_mux_value(self.u, options.rx_subdev_spec))
        #self.u.set_mux(gru.hexint(0xf3f2f1f0))

        #PRINT STATEMENTS
        print "Using USB Port %d" % (options.usb_num)
        if(options.nchan > 1):
            print "Using %d Channels" % (options.nchan)
        else:
            print "Using %d Channel" % (options.nchan)
        #display USB sample rate
        input_rate = self.u.adc_freq() / self.u.decim_rate()
        print "USB sample rate %s" % (eng_notation.num_to_str(input_rate))
        self.rssi_run = True

    def GetRSSI(self, d, t):
        reads = []
        avgs = []
        while self.rssi_run:
            tmp = self.u.read_aux_adc(self.rx_subdev[0],0)
            reads.append(tmp)
            self.rssi = sum(reads[-1140:])/1140
            avgs.append(self.rssi)
            file = open(receive.fn+"ssi", "wt")
            file.write(str(max(avgs[-20:])))
            file.close()

if __name__ == '__main__':
    try:
        receive = my_graph()
        thread.start_new_thread(receive.GetRSSI,(0,0))
        receive.run()
        print "Receiving Complete."
        receive.rssi_run = False

    except KeyboardInterrupt:
       # pass
        receive.rssi_run = False
        receive.stop()
```

A.2.2 USRP Transmitter Script (bbc_tx.py)

```
#!/usr/bin/env python

# This program reads waveform data from the file "bbc_tx.dat" and sends
# it to the USRP for broadcast.

# This file was derived from the usrp_siggen.py file that came with
# GNU Radio. It was stripped to just the essentials needed to transmit
# a baseband signal from a complex file source.

# The file format is complex IQ data pairs where both values are IEEE
# single-precision floating point numbers in little endian format.
# The first value is I and the second value is Q. The data is present
# only on the I data. The Q data is all zeros.

from gnuradio import gr, gru
from gnuradio import usrp
from gnuradio.eng_option import eng_option
from gnuradio import eng_notation
from gnuradio.eng_notation import num_to_str, str_to_num
from optparse import OptionParser
import sys
import time
import os
from subprocess import *


class bbc_tx_graph(gr.top_block):
    def __init__ (self, usb_num, sink_path, jammer, jammer_level=0, sink_path_B=None): #included
         usb_num from parameter list to define which usb

        gr.top_block.__init__(self)
        #default interpolator rate
        self.interp = 64

        if jammer==0:
            self.txfile = gr.file_source (gr.sizeof_gr_complex, sink_path,1)
            self.usrp = usrp.sink_c (usb_num, self.interp) # change from 0 to 1 if necessary
            self.connect (self.txfile, self.usrp)
        elif jammer==1:
            call([”/Users/Derek/Desktop/jammer/jammer”, ”-C” ,”/Users/Derek/Desktop/jammer/tx.ini”
                 , ”-N”, ”1000000” ,”-J”, str(jammer_level)])
            self.txfileA = gr.file_source (gr.sizeof_gr_complex, sink_path, 1)
            self.txfileB = gr.file_source (gr.sizeof_gr_complex, ”/Users/Derek/Desktop/jammer/r”,
                 1)
            self.usrp = usrp.sink_c(which=usb_num, interp_rate=self.interp,nchan=2)
            #do connect
            intl = gr.interleave(gr.sizeof_gr_complex)
            self.connect(self.txfileA, (intl, 0))
            self.connect(self.txfileB, (intl, 1))
            self.connect(intl,self.usrp)
        elif jammer==2:
            self.txfileA = gr.file_source (gr.sizeof_gr_complex, sink_path, 1)
            self.noisegen = gr.noise_source_c (gr.GR_GAUSSIAN, 500*jammer_level)
            self.usrp = usrp.sink_c(which=usb_num, interp_rate=self.interp,nchan=2)
            #do connect
            intl = gr.interleave(gr.sizeof_gr_complex)
            self.connect(self.txfileA, (intl, 0))
            self.connect(self.noisegen, (intl, 1))
            self.connect(intl,self.usrp)
        elif jammer==3:
            self.noisegen = gr.noise_source_c (gr.GR_GAUSSIAN, 500*jammer_level)
            self.usrp = usrp.sink_c (usb_num, self.interp) # change from 0 to 1 if necessary
            self.connect (self.noisegen, self.usrp)

    def usb_freq (self):
        return self.usrp.dac_freq() / self.interp

    def usb_throughput (self):
        return self.usb_freq () * 4

    def set_interpolator (self, interp):
        self.interp = interp
        self.usrp.set_interp_rate (interp)

    def set_freq_single(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """
        r = self.usrp.tune(self.subdev._which, self.subdev, target_freq)
        if r:
            print "r.baseband_freq =", eng_notation.num_to_str(r.baseband_freq)
            print "r.dxc_freq      =", eng_notation.num_to_str(r.dxc_freq)
            print "r.residual_freq =", eng_notation.num_to_str(r.residual_freq)
            print "r.inverted      =", r.inverted
            print "  OK"
            return True

        return False

    def set_freq_multi(self, side, target_freq):
        """
        Set the center frequency we're interested in.

        @param side: 0 = side A, 1 = side B
        @param target_freq: frequency in Hz
        @rtype: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """

        print "Tuning side %s to %sHz" % (("A", "B")[side], num_to_str(target_freq))
        r = self.usrp.tune(self.subdev[side]._which, self.subdev[side], target_freq)
        if r:
            print "  r.baseband_freq =", num_to_str(r.baseband_freq)
            print "  r.dxc_freq      =", num_to_str(r.dxc_freq)
            print "  r.residual_freq =", num_to_str(r.residual_freq)
            print "  r.inverted      =", r.inverted
            print "  OK"
            return True

        else:
            print "  Failed!"

        return False

def main ():
    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-T", "--tx_subdev_spec", type="subdev", default=(0, 0),
                       help="select USRP Tx side A or B (may also use A:0 or A:1 format)")
    parser.add_option ("-f", "--rf_freq", type="eng_float", default=None,
                       help="set RF center frequency to FREQ")
    parser.add_option ("-i", "--interp", type="int", default=64,
                       help="set fgpa interpolation rate to INTERP")
    parser.add_option("-U", "--usb_num", type="int", default=0,
                       help="select USRP USB location 0 or 1 (default=0)")
    parser.add_option("-J", "--jammer", type="int", default=0, help="0 = None, 1 = Pulse Jammer, 2 = Gaussian Jammer")
    parser.add_option ("--jammer_level", type="int", default=32,
                       help="set the jammer level [0,64]")
    parser.add_option("-S", "--sink_path",type="string",default=None, help="set sink file path for transmission 1")
    parser.add_option("-Q", "--sink_path_B",type="string",default=None, help="set sink file path for transmission 2")
    parser.add_option("-P", "--tx_subdev_spec_B", type="subdev", default=(0, 0),
                       help="select USRP Tx side A or B (may also use A:0 or A:1 format)")
    parser.add_option("-L", "--tx_time",type="float", default=8.0, help ="set the length to
         transmit for")
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        raise SystemExit

    if options.rf_freq is None:
        sys.stderr.write("usrp_siggen: must specify RF center frequency with -f RF_FREQ\n")
        parser.print_help()
        raise SystemExit

    fg = bbc_tx_graph(options.usb_num,options.sink_path, options.jammer, options.jammer_level,
         options.sink_path_B)

    fg.set_interpolator (options.interp)

    print "Using USB Port %d" % (options.usb_num)
    print "Sink path: %s" % (options.sink_path)
    if(options.jammer):
        print "Jammer running at level: %i" % (options.jammer_level)

    if options.jammer==0:
        print "Using 1 Channel"
        # determine the daughterboard subdevice we're using
        if options.tx_subdev_spec is None:
            options.tx_subdev_spec = usrp.pick_tx_subdevice(fg.u)

        m = usrp.determine_tx_mux_value(fg.usrp, options.tx_subdev_spec)
        #print "mux = %#04x" % (m,)
        fg.usrp.set_mux(m)
        #fg.usrp.set_mux(0xba98)

        fg.subdev = usrp.selected_subdev(fg.usrp, options.tx_subdev_spec)
        print "Using TX daughterboard %s" % (fg.subdev.side_and_name(),)

        fg.subdev.set_gain(fg.subdev.gain_range()[1])    # set max Tx gain

        if not fg.set_freq_single(options.rf_freq):
            sys.stderr.write('Failed to set RF frequency\n')
            raise SystemExit

        fg.subdev.set_enable(True)# enable transmitter
    else:  # we're using both daughterboard slots, thus subdev is a 2-tuple
        print ”Using 2 Channels”
        fg.subdev = (fg.usrp.db[0][0], fg.usrp.db[1][0])
        print ”Using TX daughterboard %s” % (fg.subdev[0].side_and_name(),)
        print ”Using TX daughterboard %s” % (fg.subdev[1].side_and_name(),)


        #m_A = usrp.determine_tx_mux_value(fg.usrp, options.tx_subdev_spec)
        #m_B = usrp.determine_tx_mux_value(fg.usrp, options.tx_subdev_spec_B)
        #print ”mux = %#04x” % (m,)
        #fg.subdev[0].set_mux(m_A)
        #fg.subdev[1].set_mux(m_B)
        #fg.usrp.set_mux(m_A)
        fg.usrp.set_mux(gru.hexint(0xBA98))

        fg.subdev[0].set_gain(fg.subdev[0].gain_range()[1])    # set max Tx gain
        fg.subdev[1].set_gain(fg.subdev[1].gain_range()[1])    # set max Tx gain

        #use same frequency for both transmitters
        fg.set_freq_multi(0, options.rf_freq)
        fg.set_freq_multi(1, options.rf_freq)
        #fg.subdev[0].set_freq(options.rf_freq)
        #fg.subdev[1].set_freq(options.rf_freq)

        fg.subdev[0].set_enable(True) # enable transmitter
        fg.subdev[1].set_enable(True) # enable transmitter

    try:
        #size = os.stat(options.sink_path).st_size
        #tx_sec = (size-163840)/(fg.usb_freq()) # num_samples/tx_samples_sec
        #if tx_sec > 8:
        #    tx_sec = 8

        print ”Transmitting for”, str(options.tx_time)
        #t1 = time.time()
        #fg.run()
        #t2 = time.time()
        #print ”%i” % (t2-t1)
        fg.start()
        time.sleep(options.tx_time)
        fg.stop()
        print ”Transmission Completed.\n”
    except KeyboardInterrupt:
        #pass
        fg.stop()
```

```
if __name__ == '__main__':
    main()
```

A.3 BBC Source Code

A.3.1 bbcftp.h

```
/****************************************************************************
 * Application Layer for the Real-time BBC Codec/Modem                      *
 ****************************************************************************
 * William L. Bahn                                                          *
 * Academy Center for Information Security                                  *
 * Department of Computer Science                                           *
 * United States Air Force Academy                                          *
 * USAFA, CO 80840                                                          *
 ****************************************************************************
 * FILE:............ bbcftp.h                                              *
 * DATE CREATED:.... 13 SEP 07                                              *
 * DATE MODIFIED:... 13 SEP 07                                              *
 ****************************************************************************
 *
 * REVISION HISTORY
 *
 ****************************************************************************
 *
 * DESCRIPTION
 *
 */

#ifndef BBCFTPdotH
#define BBCFTPdotH

//--------------------------------------------------------------------------
// REQUIRED INCLUDES
//--------------------------------------------------------------------------

#include "config.h"
#include "source.h"
#include "codec.h"
#include "buffer.h"
#include "modem.h"
#include "sink.h"
```

```
#include "dirtyd.h"

//----------------------------------------------------------------------------
// PARAMETER DEFINITIONS
//----------------------------------------------------------------------------

#define BBC_FTP_BYTES_CHECKSUM   (4)
#define BBC_FTP_BYTES_SEQNUM     (2)
#define BBC_FTP_BYTES_LOADBITS   (2)
#define BBC_FTP_BYTES_ID         (2)

#define BBC_FTP_OFFSET_CHECKSUM  (0)
#define BBC_FTP_OFFSET_ID        (BBC_FTP_OFFSET_CHECKSUM + BBC_FTP_BYTES_CHECKSUM)
#define BBC_FTP_OFFSET_SEQNUM    (BBC_FTP_OFFSET_ID       + BBC_FTP_BYTES_ID      )
#define BBC_FTP_OFFSET_LOADBITS  (BBC_FTP_OFFSET_SEQNUM   + BBC_FTP_BYTES_SEQNUM  )
#define BBC_FTP_OFFSET_PAYLOAD   (BBC_FTP_OFFSET_LOADBITS + BBC_FTP_BYTES_LOADBITS)
#define BBC_FTP_HEADER_BYTES     (BBC_FTP_OFFSET_PAYLOAD)

//----------------------------------------------------------------------------
// STRUCTURE TYPE DEFINITIONS
//----------------------------------------------------------------------------

typedef struct BBCFTP BBCFTP;

//----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct BBCFTP
{
  CONFIG *config;
  SOURCE *source;
  CODEC  *codec;
  BUFFER *buffer;
  MODEM  *modem;
  SINK   *sink;
};

//----------------------------------------------------------------------------
```

```
// PUBLIC FUNCTION PROTOTYPES
//------------------------------------------------------------------------------

BBCFTP *BBCFTP_Del(BBCFTP *p);
BBCFTP *BBCFTP_New(char *filename, DWORD *errcode);

void PrintMessage(BYTE *base);

void  SetMessageChecksum(BYTE *base, DWORD v);
void  SetMessageSeq(BYTE *base, WORD v);
void  SetMessageLoadBits(BYTE *base, WORD v);
void  SetMessageID(BYTE *base, WORD v);
void  SetMessagePayload(BYTE *base, BYTE *source, DWORD bytes, int offset);

DWORD GetMessageChecksum(BYTE *base);
WORD  GetMessageSeq(BYTE *base);
WORD  GetMessageLoadBits(BYTE *base);
WORD  GetMessageID(BYTE *base);
BYTE *GetMessagePayload(BYTE *base);


//------------------------------------------------------------------------------
#endif
```

A.3.2 bbcftp.c

```
/******************************************************************************
 * Application Layer Module for the Real-time BBC Codec/Modem FTP program    *
 ******************************************************************************
 * William L. Bahn                                                            *
 * Academy Center for Information Security                                    *
 * Department of Computer Science                                             *
 * United States Air Force Academy                                            *
 * USAFA, CO 80840                                                            *
 ******************************************************************************
 * FILE:............ bbcftp.c                                                *
 * DATE CREATED:.... 18 SEP 07                                                *
 * DATE MODIFIED:... 18 SEP 07                                                *
 ******************************************************************************
 *
 * REVISION HISTORY
 *
 ******************************************************************************
 *
 * DESCRIPTION
 *
 * This module provides the crude application layer functions for the ftp
 * demo.
 */

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------

#include <stdlib.h> // malloc(), free()
#include <stdio.h>  // printf()
#include <string.h> // memmove()

#include "bbcftp.h"

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//-----------------------------------------------------------------------------
// PRIVATE FUNCTION DEFINITIONS
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//-----------------------------------------------------------------------------

BBCFTP *BBCFTP_Del(BBCFTP *p)
{
  if (p)
  {
    p->config = CONFIG_Del(p->config);
    p->source = SOURCE_Del(p->source);
    p->codec  = CODEC_Del(p->codec);
    p->buffer = BUFFER_Del(p->buffer);
    p->modem  = MODEM_Del(p->modem);
    p->sink   = SINK_Del(p->sink);
  }
  return NULL;
}

BBCFTP *BBCFTP_New(char *filename, DWORD *errcode)
{
  BBCFTP *p;
  DWORD err;

  p = NULL;
  err = 0;
  *errcode = 0;

  p = (BBCFTP *) malloc(sizeof(BBCFTP));
  if (!p) *errcode |= 1 << 0;

  if (!*errcode)
  {
    p->config = CONFIG_New(filename, &err);
    if (err) *errcode |= 1 << 1;
  }

  if (!*errcode)
  {
    p->source = SOURCE_New(p->config, &err);
    if (err) *errcode |= 1 << 2;
    p->codec  = CODEC_New(p->config, &err);
    if (err) *errcode |= 1 << 3;
    p->buffer = BUFFER_New(p->config, &err);
    if (err) *errcode |= 1 << 4;
    p->modem  = MODEM_New(p->config, &err);
    if (err) *errcode |= 1 << 5;
    p->sink   = SINK_New(p->config, &err);
    if (err) *errcode |= 1 << 6;
  }

  return p;
}

void BBCFTP_ErrorCodes(DWORD err)
{
  if ( err & ((DWORD) 1 << 0) )
    printf("BBC-FTP System Constructor failed to allocate\n");
  if ( err & ((DWORD) 1 << 1) )
    printf("CONFIG Constructor exited with errors\n");
  if ( err & ((DWORD) 1 << 2) )
```

```
    printf("SOURCE Constructor exited with errors\n");
  if ( err & ((DWORD) 1 << 3) )
    printf("CODEC  Constructor exited with errors\n");
  if ( err & ((DWORD) 1 << 4) )
    printf("BUFFER Constructor exited with errors\n");
  if ( err & ((DWORD) 1 << 5) )
    printf("MODEM  Constructor exited with errors\n");
  if ( err & ((DWORD) 1 << 6) )
    printf("SINK   Constructor exited with errors\n");
}

void PrintMessage(BYTE *base)
{
  int i;
  int chunk_size_bytes;

  DWORD checksum;
  WORD  seqnum, loadbits, id;

  checksum = GetMessageChecksum(base);
  seqnum = GetMessageSeq(base);
  loadbits = GetMessageLoadBits(base);
  id = GetMessageID(base);

  printf("[%04lu] ", (unsigned long) checksum);
  printf("[%04lu] ", (unsigned long) seqnum);
  printf("[%04lu] ", (unsigned long) loadbits);
  printf("[%04lu] ", (unsigned long) id);

  chunk_size_bytes = loadbits/8;

  printf("[");
  for (i = 0; i < chunk_size_bytes; i++)
  {
    putc(*(base + BBC_FTP_OFFSET_PAYLOAD + i), stdout);
  }
  printf("]\n");

}

void SetMessageChecksum(BYTE *base, DWORD v)
{
  memmove(base+BBC_FTP_OFFSET_CHECKSUM, &v, BBC_FTP_BYTES_CHECKSUM);
}
```

```
void SetMessageSeq(BYTE *base, WORD v)
{
  memmove(base+BBC_FTP_OFFSET_SEQNUM, &v, BBC_FTP_BYTES_SEQNUM);
}

void SetMessageLoadBits(BYTE *base, WORD v)
{
  memmove(base+BBC_FTP_OFFSET_LOADBITS, &v, BBC_FTP_BYTES_LOADBITS);
}

void SetMessageID(BYTE *base, WORD v)
{
  memmove(base+BBC_FTP_OFFSET_ID, &v, BBC_FTP_BYTES_ID);
}

void SetMessagePayload(BYTE *base, BYTE *source, DWORD bytes, int offset)
{
  memmove(base+BBC_FTP_OFFSET_PAYLOAD+offset, source, bytes);
}

DWORD GetMessageChecksum(BYTE *base)
{
  return *((DWORD *)(base + BBC_FTP_OFFSET_CHECKSUM));
}

WORD GetMessageSeq(BYTE *base)
{
  return *((WORD *)(base + BBC_FTP_OFFSET_SEQNUM));
}

WORD GetMessageLoadBits(BYTE *base)
{
  return *((WORD *)(base + BBC_FTP_OFFSET_LOADBITS));
}

WORD GetMessageID(BYTE *base)
{
  return *((WORD *)(base + BBC_FTP_OFFSET_ID));
}

BYTE *GetMessagePayload(BYTE *base)
{
  return (BYTE *)(base + BBC_FTP_OFFSET_PAYLOAD);
}

```

```
//-----------------------------------------------------------------------
```

A.3.3 buffer.h

```
/*************************************************************************
 * Data Buffer for the Real-time BBC Codec/Modem                         *
 *************************************************************************
 * William L. Bahn                                                       *
 * Academy Center for Information Security                               *
 * Department of Computer Science                                        *
 * United States Air Force Academy                                       *
 * USAFA, CO 80840                                                       *
 *************************************************************************
 * FILE:............ buffer.h                                            *
 * DATE CREATED:.... 01 SEP 07                                           *
 * DATE MODIFIED:... 01 SEP 07                                           *
 *************************************************************************
 *
 * REVISION HISTORY
 *
 *************************************************************************
 *
 * DESCRIPTION
 *
 * The data buffer stores packet data between the codec and the modem.
 *
 * In the receiver, the buffer accepts packet data from the codec and
 * feeds that data to the modem. In the transmitter, it accepts data from
 * the modem and feeds it to the codec. While the modem, by its nature,
 * generally produces and consumes data at a uniform rate, the codec
 * can be quite erratic in its data rate. Therefore the buffer must be
 * sized sufficiently large to allow for the resulting ebb and flow.
 * This is particularly important in the case of the receiver since, if
 * the buffer can't accommodate the data as the modem delivers it, data
 * will be lost. This is not as critical with the transmitter, depending
 * on the nature of the data source and its buffering strategy, since it
 * will normally only reduce the effective data rate as opposed to causing
 * dropped packets.
 *
 * The data is stored in a circular buffer with the following variables:
 *
 *     buffer: Pointer to the block of memory where the buffer starts.
 *     read:   Index of the first byte of the present packet.
 *     write:  Index of the next unused buffer location.
 *     margin: How many bytes are in buffer beyond the scope of the decoder.
 *      unused: How many unused bytes are available in the buffer.
 *
 * The buffer is seen by two functions, the one that is demodulating the
 * data packet and the one that is decoding the resulting data. The
 * demodulating function writes to the buffer at a nominally constant
 * rate dictated by the communications link. In this application, this is
 * simulated by reading the stored waveform data from a file and querying
 * the clock to determine how many bytes to add to the buffer each time
 * the function is called. The decoding function, on the other hand, always
 * to decodes eight packets each time it is called provided sufficient data
 * is available. Specifically, it decodes the eight packets that start with
 * the bits in the byte stored at the ”read” pointer. Since it can't decode
 * packets that are not completely contained in the buffer, the decoding
 * function first checks to see if ”fill” is non-negative. If it isn't, then
 * it returns immediately. At the other end of the spectrum, the demodulator
 * may run out of unused memory to write to. If this happens, data is going
 * to be lost. It is cleaner to throw away old data instead of introducing
 * a gap in present data, therefore the demodulator will push the ”read”
 * pointer forward as it overwrites the beginning of the existing packet
 * data.
 *
 */

#ifndef BUFFERdotH
#define BUFFERdotH

//----------------------------------------------------------------------------
// REQUIRED INCLUDES
//----------------------------------------------------------------------------

#include ”config.h”
#include ”dirtyd.h”

//----------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
//----------------------------------------------------------------------------

typedef struct BUFFER BUFFER;

//----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct BUFFER
{
   size_t   size;          // Allocated size of buffer (in bytes)
   size_t   minsize;       // Minimum acceptable buffer size (in bytes)
   BYTE    *buffer;        // Pointer to the actual buffer
   DWORD    read;          // Index of next position to be read.
   DWORD    write;         // Index of next position to be written.
   DWORD    scope;         // The number of bytes recipient must.
   SDWORD   margin;        // Number of bytes beyond scope of recipient
   DWORD    empty;         // Number of bytes available for new data.
   DWORD    ready;         // Number of bytes ready for modulation.
   DWORD    buffermask;    // The used bits in the buffer size
   DWORD    overflows;     // Number of data pushes into read pointer.
};

//----------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//----------------------------------------------------------------------------

BUFFER *BUFFER_Del(BUFFER *p);
BUFFER *BUFFER_New(CONFIG *c, DWORD *errcode);

//----------------------------------------------------------------------------
#endif
```

A.3.4 buffer.c

```
/***************************************************************************
 * Data Buffer for the Real-time BBC Codec/Modem                           *
 ***************************************************************************
 * William L. Bahn                                                         *
 * Academy Center for Information Security                                 *
 * Department of Computer Science                                          *
 * United States Air Force Academy                                         *
 * USAFA, CO 80840                                                         *
 ***************************************************************************
 * FILE:............ buffer.c                                              *
 * DATE CREATED:.... 01 SEP 07                                             *
 * DATE MODIFIED:... 01 SEP 07                                             *
 ***************************************************************************
 *
 * REVISION HISTORY
 *
 ****************************************************************************
 *
 * DESCRIPTION
 *
 * The data buffer and its programmer interface is described in buffer.h.
 *
 ****************************************************************************
 */

//---------------------------------------------------------------------------
// REQUIRED INCLUDES
//---------------------------------------------------------------------------

#include <stdlib.h> // malloc(), free()
#include <string.h> // memset()

#include "buffer.h"

//---------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//---------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//---------------------------------------------------------------------------
// PRIMITIVE FUNCTION DEFINITIONS
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// PRIVATE FUNCTION DEFINITIONS
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//---------------------------------------------------------------------------

BUFFER *BUFFER_Del(BUFFER *p)
{
  if (p)
  {
    if (p->buffer) { free (p->buffer); p->buffer = NULL;  }
  }
  return NULL;
}

BUFFER *BUFFER_New(CONFIG *c, DWORD *errcode)
{
  BUFFER *p;
  DWORD err;

  p = NULL;
  err = 0;

  if (!err)
  {
    p = (BUFFER *) malloc(sizeof(BUFFER));
    if (!p)
      err |= 1 << 1;
  }

  if (!err)
  {
    p->minsize = (size_t) (c->bufferbytes_per_packet * c->buffer_packets);
    p->size = 1;
    while ((0 != p->size)&&(p->size < p->minsize))
      p->size <<= 1;
    if (0 == p->size)
      err |= 1 << 2;
  }

  if (!err)
  {
    // Allocate buffer memory
    p->buffer = (BYTE *) malloc(p->size*sizeof(BYTE));
    if (!p->buffer)
      err |= 1 << 3;

    // Initialize buffer state

    // Common to TX and RX
    p->buffermask = p->size - 1;
    p->scope = c->bufferbytes_per_packet;
    p->read = 0;
    p->write = 0;
    p->overflows = 0;

    // TX and RX specific
    if (c->scheduler_TX_notRX)
    {
      p->margin = p->size - p->scope;
      p->ready = 0;
      p->empty = 0; // Not used
    }
    else
    {
      p->margin = -((SDWORD)p->scope);
      p->ready = 0; // Not used
      p->empty = p->size;
    }
  }

  // Clear entire buffer
  if (!err)
    memset(p->buffer, 0, p->size);

  if (err)
    p = BUFFER_Del(p);

  if (c->diagnostics)
  {
    // Diagnostic Report
    printf("----------------------------------------\n");
    printf("PACKET BUFFER\n");
    printf("  Creation:............... %s\n", ((err)? "FAILED":"SUCCEEDED"));
    printf("  Location:............... %p\n", (void *) p);
    printf("  Minimum buffer size:.... %lu bytes\n", (unsigned long) p->minsize);
    printf("  Buffer size:............ %lu bytes\n", (unsigned long) p->size);
    printf("  Buffer location:........ %p\n", (void *) p->buffer);
    printf("  Packet size in buffer:.. %lu bytes\n", (unsigned long) p->scope);
    printf("  read:................... %lu bytes\n", (unsigned long) p->read);
    printf("  write:.................. %lu bytes\n", (unsigned long) p->write);
    printf("  empty:.................. %lu bytes\n", (unsigned long) p->empty);
    printf("  ready:.................. %lu bytes\n", (unsigned long) p->ready);
    printf("  margin:................. %li bytes\n", (long) p->margin);
    printf("  overflows:.............. %lu bytes\n", (unsigned long) p->overflows);
    printf("----------------------------------------\n");
  }

  *errcode = err;
```

```
    return p;
}

//-----------------------------------------------------------------------
```

A.3.5 bytes.h

```
/* ======================================================================
 * PROGRAMMER "BAHN, William"
 * TITLE      "Integer Storage Size Type Definitions"
 * CREATED    06 FEB 07
 * MODIFIED   06 FEB 07
 * FILENAME   "bytes.h"
 * ======================================================================
 * GENERAL DESCRIPTION
 *
 * This file contains type definitions so that porting from one processor
 * to another is simpler.
 * ======================================================================
 * SIZE DEFINITIONS
 *
 * The following definitions are used:
 *
 *    SIZE    UNSIGNED   SIGNED
 *    8-bit     BYTE     SBYTE
 *   16-bit     WORD     SWORD
 *   32-bit     DWORD    SDWORD
 *   64-bit     QWORD    SQWORD (not available on most systems)
 *
 * ======================================================================
 * To Verify Sizes
 *
 * Use the VerifySIZES() function passing the largest integer size, in
 * bits, that is of interest.
 *
 * The function returns TRUE if conflicts are found.
 *
 * If an argument of 0 is used, then the return value has a bit set for
 * each type definition that didn't verify, starting with the shortest
 * length in the LSB.
 *
 * Example - you are interested only in integer sizes up to 32-bits.
 *
 * VerifySIZES(32) or VerifySIZES(BITSinDWORD);
 *
 * =====================================================================
 */

#ifndef BYTESdotH
#define BYTESdotH

#define BITSinBYTE  ( 8)
#define BITSinWORD  (16)
#define BITSinDWORD (32)
#define BITSinQWORD (64)

/* =====================================================================
 * Normal definitions
 *
 * This the only section that should need to be changed.
 *
 * Determine which integer type is the correct number of bits and update
 * the following list. Do not worry about signed/unsigned.
 *
 * It is not recommended that you actually use these definitions in your
 * code - they are simply used in the following type definitions.
 * =====================================================================
 */

#define NBYTE  char
#define NWORD  short
#define NDWORD int
#define NQWORD long

/* =====================================================================
 * UNSIGNED TYPE DEFINITIONS
 * =====================================================================
 */

typedef unsigned NBYTE   BYTE;
typedef unsigned NWORD   WORD;
typedef unsigned NDWORD  DWORD;
typedef unsigned NQWORD  QWORD;

/* =====================================================================
 * SIGNED TYPE DEFINITIONS
 * =====================================================================
 */

typedef signed   NBYTE   SBYTE;
typedef signed   NWORD   SWORD;
typedef signed   NDWORD  SDWORD;
typedef signed   NQWORD  SQWORD;

/* ====================================================================
 * UTILITY FUNCTIONS
 * ====================================================================
 */

unsigned int VerifySIZES(unsigned int maxlength);

#endif
```

A.3.6 bytes.c

```
/* ====================================================================
 * PROGRAMMER ”BAHN, William”
 * TITLE      ”Integer Storage Size Type Definitions”
 * CREATED    06 FEB 07
 * MODIFIED   06 FEB 07
 * FILENAME   ”bytes.c”
 * ====================================================================
 * GENERAL DESCRIPTION
 *
 * NOTE: ANY AVAILABLE ”USER GUIDE” IS IN THE ASSOCIATED HEADER FILE.
 *
 * This file contains type definitions so that porting from one processor
 * to another is simpler.
 *
 * ====================================================================
 */

#include ”bytes.h”

int VerifyBYTE(void)
{
  return (8*sizeof(BYTE) != BITSinBYTE);
}

int VerifyWORD(void)
{
  return (8*sizeof(WORD) != BITSinWORD);
}

int VerifyDWORD(void)
```

```
{
  return (8*sizeof(DWORD) != BITSinDWORD);
}

int VerifyQWORD(void)
{
  return (8*sizeof(QWORD) != BITSinQWORD);
}

unsigned int VerifySIZES(unsigned int maxlength)
{
  unsigned int flags;
  unsigned int mask;

  // Generate a flag vector with a 1 set anyplace that does not
  // verify properly. Note that the bit position is equal to base-2
  // log of the number of bytes in the integer type.

  flags = 0;
  flags = (flags << 1) + VerifyQWORD();
  flags = (flags << 1) + VerifyDWORD();
  flags = (flags << 1) + VerifyWORD();
  flags = (flags << 1) + VerifyBYTE();

  // Convert length from bits to smallest compatible number of bytes.

  maxlength = (maxlength/8) + ((maxlength%8)?1:0);

  // Generate a mask that is set only in those flag positions of interest.

  if (maxlength) // report on sizes up to and including maxlength.
    for (mask = 0; maxlength > 0; maxlength /= 2)
    {
      mask = (mask << 1) + 1;
      if ((maxlength > 1)&&(maxlength%2))
        mask = (mask << 1) + 1;
    }
  else // report on all defined sizes
    mask = ~0;

  return (flags & mask);
}
```

A.3.7 codec.h

```
/***************************************************************************
 * CODEC for the Real-time BBC Codec/Modem                                 *
 ***************************************************************************
 * William L. Bahn                                                         *
 * Academy Center for Information Security                                 *
 * Department of Computer Science                                          *
 * United States Air Force Academy                                         *
 * USAFA, CO 80840                                                         *
 ***************************************************************************
 * FILE:............ codec.h                                               *
 * DATE CREATED:.... 06 SEP 07                                             *
 * DATE MODIFIED:... 06 SEP 07                                             *
 ***************************************************************************
 *
 * REVISION HISTORY
 *
 ***************************************************************************
 *
 * DESCRIPTION
 *
 * The codec encodes and decodes messages to/from BBC-encoded packets.
 *
 ***************************************************************************
 */

#ifndef CODECdotH
#define CODECdotH

//--------------------------------------------------------------------------
// REQUIRED INCLUDES
//--------------------------------------------------------------------------

#include ”config.h”
#include ”source.h”
#include ”buffer.h”
#include ”sink.h”
#include ”dirtyd.h”
#include ”sha1.h”

typedef struct thread_data thread_data;


//--------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
```

```
//-----------------------------------------------------------------------------

typedef struct CODEC CODEC;

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct CODEC
{
  // State information
  SHA1Context *state;       // Pointer to single SHA1 structure
  SHA1Context *digest;      // Pointer to single SHA1 structure

  // Decode buffer
  BYTE   *msg;              // Array containing the message bit contents (1 bit per byte)
  BYTE   *checkbit;         // Array indicating whether each bit is a message or check bit
  SHA1Context *hashstate;   // Array of SHA1 structures
};

struct thread_data
{
    CONFIG *config;
    BUFFER *buffer;
    CODEC *codec;
    SINK *sink;
    int *running;
    int number;
};

//-----------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//-----------------------------------------------------------------------------

CODEC *CODEC_Del(CODEC *p);
CODEC *CODEC_New(CONFIG *c, DWORD *errcode);
void Encode(CONFIG *c, SOURCE *source, CODEC *codec, BUFFER *buffer);
void Decode(CONFIG *c, BUFFER *buf, CODEC *codec, SINK *sink);

```

```
/* DECODER
 *
 * The decoder decodes all eight of the packets that start with each of the
 * eight bits in the byte located at the present "read" location of the buffer.
 *
 * The value of the variable "originbit" determines which of the eight offsets
 * from the beginning of the byte the present packet starts at. The variable
 * "location" refers to the location of the bit in question relative to the
 * beginning of the packet. Therefore, relative to the beginning of the byte
 * where the packet starts, the location is simply "origin + location". This
 * combined location must then be turned into an index and and offset. The
 * "index" refers to which byte within the buffer contains the bit of interest
 * while the "offset" identifies the bit within that byte. The "index" value
 * must further account for the fact that the first byte in the packet is
 * located at the "read" point within the index and that the buffer is circular.
 * The "offset" value must be used to mask the byte being examined so that only
 * the bit of interest is considered. For speed purposes, this mask is provided
 * by a lookup table "bitmask".
 *
 * Taking all of this into account, the following steps will check if a
 * particular packet bit is set:
 *
 * index  = {read + floor[(location + originbit)/8]} mod bufferlength
 * offset = (location + originbit) mod 8
 * status = buffer[index] & bitmask[offset]
 *
 * Since the buffer length is exactly 2^n long, the residue of the index can
 * be taken by simply retaining only the lower n bits. Similarly, the residue
 * of the offset modulo-8 can be taken by only retaining the lower 3 bits. Both
 * of these can be done by performing a bitwise-AND with an appropriate mask.
 * Finally, the division of the effective location within the packet can be
 * performed by right-shifting the sum by 3 bits. Hence we have the following
 * equations:
 *
 * index = (read + ((location + originbit) >> 3)) & buffermask;
 * offset = (location + originbit) & 0x00000007;
 * status = buffer[index] & bitmask[offset]
 *
 * The most challenging part of the decoding algorithm is the backtracking that
 * must take place when the present partial message is finished, either because
 * it was found to be a dead end or because it resulted in an actual message.
 * The basic task is to traverse the decoding tree backwards until the last
 * partial message bit that was a zero is found. Then that bit is changed to a one
 * and decoding moves forward again. Two special cases have to be taken into
 * account. First, if there are no message bits that are zero, then the decoding
 * of that packet is finished. Second, checksum bits are always zero and the
 * decoder must skip over them without turning them to ones.
 * index    0123456789....
 * check    1001001001....
 * msg      0010110110....
 *
 */

//-----------------------------------------------------------------------------
#endif
```

A.3.8 codec.c

```
/**************************************************************************
 * CODEC for the Real-time BBC Codec/Modem                                *
 **************************************************************************
 * William L. Bahn                                                        *
 * Academy Center for Information Security                                *
 * Department of Computer Science                                         *
 * United States Air Force Academy                                        *
 * USAFA, CO 80840                                                        *
 **************************************************************************
 * FILE:............ codec.c                                              *
 * DATE CREATED:.... 06 SEP 07                                            *
 * DATE MODIFIED:... 06 SEP 07                                            *
 **************************************************************************
 *
 * REVISION HISTORY
 *
 **************************************************************************
 *
 * DESCRIPTION
 *
 * The codec and its public interface are described in codec.h
 *
 **************************************************************************
 */

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------

#include <stdlib.h> // free(), malloc()

#include "codec.h"
```

```
#include "sha1.h"

#define DECODE_LIMIT 0.05*(double)CLOCKS_PER_SEC

//---------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//---------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//---------------------------------------------------------------------------
// PRIVATE FUNCTION DEFINITIONS
//---------------------------------------------------------------------------

#define SHA1_HASH_DWORDS (5)

void ExportMessage(CONFIG *c, CODEC *codec, SINK *sink)
{
  DWORD i;
  DWORD bit;
  DWORD index, offset;

  BYTE *message;

  // Create pointer to next element in sink memory
  message = sink->v + (sink->samples * sink->sample_size_bytes);

  // Discard leading random bits
  for (bit = 0, i = 0; i < c->codec_random_bits; i++, bit++)
  {
    while (codec->checkbit[bit])
      bit++;
  }

  // Extract message bits and pack into byte string
  for (i = 0; i < c->codec_message_bits; i++, bit++)
  {
    while (codec->checkbit[bit])
      bit++;
    index  = i >> 3;
    offset = i & 0x00000007;
    message[index] &= ~c->bitmask[7-offset];
    if (codec->msg[bit])
      message[index] |= c->bitmask[7-offset];
  }

  // Zero pad remainder of last byte if necessary
  while (7 != offset)
  {
    i++;
    offset = i & 0x00000007;
    message[index] &= ~c->bitmask[7-offset];
  }

  // NUL terminate byte string
  index++;
  message[index] = '\0';
  //printf("\t\t%s\n",message);
  // Advance sink memory pointer
  sink->samples++;
}

//----------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//----------------------------------------------------------------------------

CODEC *CODEC_Del(CODEC *p)
{
  if (p)
  {
    if (p->state)     { free(p->state);     p->state = NULL;     }
    if (p->digest)    { free(p->digest);    p->digest = NULL;    }
    if (p->hashstate) { free(p->hashstate); p->hashstate = NULL; }
    if (p->msg)       { free(p->msg);       p->msg = NULL;       }
    if (p->checkbit)  { free(p->checkbit);  p->checkbit = NULL;  }
  }
  return NULL;
}

CODEC *CODEC_New(CONFIG *c, DWORD *errcode)
{
  CODEC *p;
  DWORD err;
  DWORD check;
  DWORD i, run;
  err = 0;
  p = (CODEC *) malloc(sizeof(CODEC));
  if (!p)
    err = 1 << 0;

  if (!err)
  {
    // State information
    p->state  = (SHA1Context *) malloc(sizeof(SHA1Context));
    p->digest = (SHA1Context *) malloc(sizeof(SHA1Context));

    // Decode Buffer
    p->msg       = (BYTE *) malloc(c->padded_message_bits);
    p->checkbit  = (BYTE *) malloc(c->padded_message_bits);
    p->hashstate = (SHA1Context *) malloc(c->padded_message_bits*sizeof(SHA1Context));

    // Verify successful memory allocation
    if (!(p->state))      err |= 1 << 1;
    if (!(p->digest))     err |= 1 << 2;
    if (!(p->msg))        err |= 1 << 3;
    if (!(p->checkbit))   err |= 1 << 4;
    if (!(p->hashstate))  err |= 1 << 5;
  }

  if (!err)
  {
    // Initialize checkbit markers
    for (check = TRUE, run = 0, i = 0; i < (c->padded_message_bits - c->codec_stop_bits); i++)
    {
      switch (check)
      {
        case TRUE:
          if (run >= c->codec_clamp_bits)
          {
            check = FALSE;
            run = 0;
          }
          break;
        case FALSE:
          if (run >= c->codec_fragment_bits)
          {
            check = TRUE;
            run = 0;
          }
          break;
        }
        p->checkbit[i] = check;
        run++;
      }

      for (i = 0; i < c->codec_stop_bits; i++)
        p->checkbit[c->padded_message_bits - c->codec_stop_bits + i] = TRUE;

    }

    if (c->diagnostics)
    {
      // Diagnostic Report
      printf("----------------------------------------\n");
      printf("CODEC\n");
      printf("  Creation:............... %s\n", ((err)? "FAILED":"SUCCEEDED"));
      printf("  Location:............... %p\n", (void *) p);
      printf("  Message bits:........... %li\n", (unsigned long) c->codec_message_bits);
      printf("  Random bits:............ %li\n", (unsigned long) c->codec_random_bits);
      printf("  Clamp bits:............. %li\n", (unsigned long) c->codec_clamp_bits);
      printf("  Fragment bits:.......... %li\n", (unsigned long) c->codec_fragment_bits);
      printf("  Stop bits:.............. %li\n", (unsigned long) c->codec_stop_bits);
      printf("  Padded message length:.. %li\n", (unsigned long) c->padded_message_bits);
      printf("  Packet expansion:....... %li\n", (unsigned long) c->codec_expansion);
      printf("  Packet load:............ %li messages\n", (unsigned long) c->codec_packet_load);
      printf("  Decode limit:........... %li messages\n", (unsigned long) c->codec_decode_limit);
      printf("  Message buffer at:...... %p\n", p->msg);
      printf("  Checksum buffer at:..... %p\n", p->checkbit);
      printf("  Hash buffer at:......... %p\n", p->hashstate);
      printf("  State buffer at:........ %p\n", p->state);
      printf("  Digest buffer at:....... %p\n", p->digest);
      printf("----------------------------------------\n");
    }

    if (err)
      p = CODEC_Del(p);

    *errcode = err;
    return p;
}

/* DECODER
 *
 * The decoder decodes all eight of the packets that start with each of the
 * eight bits in the byte located at the present "read" location of the buffer.
 *
 * The value of the variable "originbit" determines which of the eight offsets
 * from the beginning of the byte the present packet starts at. The variable
 * "location" refers to the location of the bit in question relative to the
 * beginning of the packet. Therefore, relative to the beginning of the byte
 * where the packet starts, the location is simply "origin + location". This
 * combined location must then be turned into an index and and offset. The
 * "index" refers to which byte within the buffer contains the bit of interest
 * while the "offset" identifies the bit within that byte. The "index" value
 * must further account for the fact that the first byte in the packet is
 * located at the "read" point within the index and that the buffer is circular.
 * The "offset" value must be used to mask the byte being examined so that only
 * the bit of interest is considered. For speed purposes, this mask is provided
 * by a lookup table "bitmask".
 *
 * Taking all of this into account, the following steps will check if a
 * particular packet bit is set:
 *
 * index  = {read + floor[(location + originbit)/8]} mod bufferlength
 * offset = (location + originbit) mod 8
 * status = buffer[index] & bitmask[offset]
 *
 * Since the buffer length is exactly 2^n long, the residue of the index can
 * be taken by simply retaining only the lower n bits. Similarly, the residue
 * of the offset modulo-8 can be taken by only retaining the lower 3 bits. Both
 * of these can be done by performing a bitwise-AND with an appropriate mask.
 * Finally, the division of the effective location within the packet can be
 * performed by right-shifting the sum by 3 bits. Hence we have the following
 * equations:
 *
 * index = (read + ((location + originbit) >> 3)) & buffermask;
 * offset = (location + originbit) & 0x00000007;
 * status = buffer[index] & bitmask[offset]
 *
 * The most challenging part of the decoding algorithm is the backtracking that
 * must take place when the present partial message is finished, either because
 * it was found to be a dead end or because it resulted in an actual message.
 * The basic task is to traverse the decoding tree backwards until the last
 * partial message bit that was a zero is found. Then that bit is changed to a one
 * and decoding moves forward again. Two special cases have to be taken into
 * account. First, if there are no message bits that are zero, then the decoding
 * of that packet is finished. Second, checksum bits are always zero and the
 * decoder must skip over them without turning them to ones.
 * index    0123456789....
 * check    1001001001....
 * msg       0010110110....
 *
 */

//-----------------------------------------------------------------------------

/*
 * The encoding function can be implemented in a more compact, efficient
 * way. The method used here is intended to mirror the decode operation
 * as closely as possible. This is reasonable because the encoding
 * operation requires constant time regardless of message and is therefore
 * well constrained.
 *
 */

void Encode(CONFIG *c, SOURCE *source, CODEC *codec, BUFFER *buffer)
{
  DWORD msg_bit, pmsg_bit, r, i, index, offset;
  unsigned int location;
  int bit_value;
  BYTE *msg;
  DWORD message_stop;
  clock_t ticks;
  DWORD marks;

  ticks = clock();

  marks = 0;
  message_stop = source->sample + c->codec_packet_load;
  if (message_stop > source->samples)
    message_stop = source->samples;

  // Place bookend marks
  location = 0;
  index = (buffer->write + (location >> 3)) & buffer->buffermask;
  offset  = location & 0x00000007;
  if (buffer->buffer[index] & c->bitmask[offset])
    marks--;
  buffer->buffer[index] |= c->bitmask[offset];
  marks++;

  location = c->last_packet_bit;
  index = (buffer->write + (location >> 3)) & buffer->buffermask;
  offset  = location & 0x00000007;
  if (buffer->buffer[index] & c->bitmask[offset])
    marks--;
  buffer->buffer[index] |= c->bitmask[offset];
  marks++;

  while (source->sample < message_stop)
  {
    if (c->diagnostics)
      printf(”Encoding message #%lu\n”, source->sample);
    // Compute pointer to beginning of present message in source buffer
    msg = (BYTE *) source->v + source->sample * source->sample_size_bytes;

    // Initialize Hash Function state to the Initial Vector
    SHA1Reset(codec->state);

    // Load message into the codec's message buffer
    for (pmsg_bit = 0, r = 0, msg_bit = 0 ; pmsg_bit < c->padded_message_bits; pmsg_bit++)
    {
      if (codec->checkbit[pmsg_bit])
        bit_value = 0;
      else
      {
        if (r < c->codec_random_bits)
        {
          bit_value = rand() < (RAND_MAX >> 1);
          r++;
        }
        else
        {
          index = msg_bit >> 3;
          offset = msg_bit & 0x00000007;
          bit_value = (msg[index] & c->bitmask[7-offset])? 1 : 0;
          msg_bit++;
        }
      }
      SHA1Input(codec->state, c->bitptr + bit_value, 1);

      // Compute hash result for present prefix
      *(codec->digest) = *(codec->state);
      SHA1Result(codec->digest);

      // Generate mark location for present prefix
      location = 0;
      for (i = 0; i < SHA1_HASH_DWORDS; i++)
        location += ((codec->digest)->Message_Digest[i])<<i;
      location %= c->packet_bits;
```

```
      // Place mark for present prefix
      index = (buffer->write + (location >> 3)) & buffer->buffermask;
      offset  = location & 0x00000007;
      if (buffer->buffer[index] & c->bitmask[offset])
        marks--;
      buffer->buffer[index] |= c->bitmask[offset];
      marks++;
    }
    source->sample++;
  }

  if (source->sample >= source->samples)
  {
    // Last packet has been encoded. Advance buffer past last packet.
    source->streaming = FALSE;
    c->buffer_advance = c->bufferbytes_per_packet;
  }

  // Advance buffer write pointer to next packet write location.
  buffer->write = (buffer->write + c->buffer_advance) & buffer->buffermask;
  buffer->margin -= c->buffer_advance;
  buffer->ready += c->buffer_advance;

  c->dec_ticks += clock() - ticks;
}

void Decode(CONFIG *c, BUFFER *buf, CODEC *codec, SINK *sink)
{


  SDWORD i, bit;
  DWORD location, index, offset, originbit;
  clock_t ticks;
  DWORD limit;
  ticks = clock();
  //if (c->diagnostics)
  //printf("Begining new Decode buf->read=[%i]\n", buf->read);
  // Process all 8 packets that begin within the byte at the front of the buffer
  for (originbit = 0; originbit < 8; originbit++ /*&& (clock() - ticks) < DECODE_LIMIT*/)
  {
    if ((sink->sample_limit - sink->samples) > c->codec_decode_limit)
      limit = (sink->sample_limit - sink->samples);
    else
      limit = c->codec_decode_limit;

    // Check for bookend marks
    index = (buf->read + (originbit >> 3)) & buf->buffermask;
    offset  = (originbit) & 0x00000007;
    if ( !(buf->buffer[index] & c->bitmask[offset]) )
      break;

    index = (buf->read + ((originbit + c->last_packet_bit) >> 3)) & buf->buffermask;
    offset  = (originbit + c->last_packet_bit) & 0x00000007;
    if ( !(buf->buffer[index] & c->bitmask[offset]) )
      break;

    // Initialize Hash Function state to the Initial Vector
    SHA1Reset(codec->state);
    bit = 0;
    codec->msg[bit] = 0;
    //printf("index=[%i] offset=[%i]\n",index, offset);
    while (TRUE) // Loop will terminate with a "break" call
    {
      /*if ((clock() - ticks) > DECODE_LIMIT)
        break;*/
      // Update the hash state for the new message bit
      SHA1Input(codec->state, c->bitptr + codec->msg[bit], 1);

      // Compute the packet bit location corresponding to the hash
      *(codec->digest) = *(codec->state);
      SHA1Result(codec->digest);
      location = 0;
      for (i = 0; i < SHA1_HASH_DWORDS; i++)
        location += ((codec->digest)->Message_Digest[i])<<i;
      location %= c->packet_bits;

      // Check for mark at calculated location
      index = (buf->read + ((originbit + location) >> 3)) & buf->buffermask;
      offset  = (originbit + location) & 0x00000007;
      //printf("\tindex=[%i] offset=[%i] location=[%i] (buf->buffer=[%i] & c->bitmask=[%i])=[%i]\n
          ",index, offset,location,buf->buffer[index], c->bitmask[offset],
      //(buf->buffer[index] & c->bitmask[offset]));
      if(buf->buffer[index] & c->bitmask[offset])
      {
        //printf("\t\tEnter\n");
        // Update hash state for present partial message
        codec->hashstate[bit] = *(codec->state);
        bit++;
        // IF a complete message hasn't been decoded yet
```

```
        if ((DWORD) bit < c->padded_message_bits)
        {
          // Start with 0 for next bit in partial message
          codec->msg[bit] = 0;
          //printf("\t\tContinue\n");
          continue;
        }
        // ELSE a complete message has been found
        //printf("\t\tComplete message found\n");
        c->message_count++;
        ExportMessage(c, codec, sink);
        bit--;
        limit--;
        if (0 == limit){
          //printf("\t\tlimit==0 break\n");
          break;
        }
      }

      // Backtrack to last message bit that is a zero
      while ( (bit >= 0) && (codec->checkbit[bit] || codec->msg[bit]) )
        bit--;

      // If no bits are zero, then decoding is finished
      if (bit < 0)
        break;

      // Change last zero bit to a one
      codec->msg[bit] = 1;

      // Reset hash state
      if (0 == bit) // to initial vector
        SHA1Reset(codec->state);
      else          // to vector of previous partial message
        *(codec->state) = codec->hashstate[bit-1];
    }
  }
  buf->read = (buf->read + 1) & buf->buffermask;
  buf->empty++;
  buf->margin--;
  //if (c->diagnostics)
  //  printf("\tDecode time: %0.05f\n", ((clock() - ticks)/(double)CLOCKS_PER_SEC));
  c->dec_ticks += clock() - ticks;;
}
```

```
//----------------------------------------------------------------------------
```

A.3.9 config.h

```
/****************************************************************************
 * Configuration Module for the Real-time BBC Codec/Modem                   *
 ****************************************************************************
 * William L. Bahn                                                          *
 * Academy Center for Information Security                                  *
 * Department of Computer Science                                           *
 * United States Air Force Academy                                          *
 * USAFA, CO 80840                                                          *
 ****************************************************************************
 * FILE:............ config.h                                               *
 * DATE CREATED:.... 03 SEP 07                                              *
 * DATE MODIFIED:... 08 SEP 07                                              *
 ****************************************************************************
 *
 * REVISION HISTORY
 *
 ****************************************************************************
 *
 * DESCRIPTION
 *
 * This module imports and manages the configuration information for the
 * modem and the codec.
 *
 */

#ifndef CONFIGdotH
#define CONFIGdotH

//----------------------------------------------------------------------------
// REQUIRED INCLUDES
//----------------------------------------------------------------------------

#include <time.h>

#include "dirtyd.h"

//----------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
//----------------------------------------------------------------------------

typedef struct CONFIG CONFIG;
```

```

//---------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//---------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct CONFIG
{
  int diagnostics;

  // Direction
  int scheduler_TX_notRX;
  int scheduler_realtime;

  // Source Parameters
  char *path;
  char *source_name;
  DWORD source_sample_size_bytes;
  DWORD source_sample_limit;
  WORD  source_id;

  // Codec Parameters
  DWORD codec_message_bits;
  DWORD codec_random_bits;
  DWORD codec_clamp_bits;
  DWORD codec_fragment_bits;
  DWORD codec_stop_bits;
  DWORD codec_expansion;
  DWORD codec_decode_limit;
  DWORD codec_packet_load;

  // Derived Codec Parameters
  DWORD fragments;      // Number of complete fragement in padded message
  DWORD padded_message_bits;      // Length of message after padding with random and check bits
  DWORD packet_bits;
  DWORD last_packet_bit;
  DWORD bytes_per_message;
  DWORD bytes_per_packet;
  DWORD bufferbytes_per_packet;

  // Buffer Parameters
  double buffer_packets;
  double buffer_lambda;
  DWORD  buffer_advance;

  // Modem Parameters
  DWORD  modem_packet_rate_bps;
  DWORD  modem_samples_per_bit;
  double modem_gain_dB;
  double modem_channel_loss_dB;
  double modem_threshold_pct;
  double modem_hysteresis_pct;
  double modem_jitter_bits;
  double modem_cushion_pct;
  // Derived Modem Parameters
  //DWORD bytes_per_sample;
  double nominal_tx_signal;
  double nominal_rx_signal;
  DWORD trx_bytes_per_packet_byte;
  DWORD cushion_bits;

  // Sink Parameters
  char *sink_name;
  DWORD sink_sample_size_bytes;
  DWORD sink_sample_limit;

  // Misc
  DWORD message_count;
  DWORD marks;

  // Lookup tables
  BYTE  bitptr[2];        // 0 and 1 represented as BYTEs that can be passed by reference
  BYTE  bitmask[8];  // Masks to pick off the bits within a byte

  // Tally Counters
  DWORD actual_trx_bytes;
  DWORD nominal_trx_bytes;
  double bytespertick;
  clock_t dem_ticks;
  clock_t dec_ticks;
  clock_t ticks;
  clock_t tot_ticks;

};
```

```
//-----------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//-----------------------------------------------------------------------------

CONFIG *CONFIG_Del(CONFIG *p);
CONFIG *CONFIG_New(char *filename, DWORD *errcode);

//-----------------------------------------------------------------------------
#endif
```

A.3.10 config.c

```
/****************************************************************************
 * Configuration Module for the Real-time BBC Codec/Modem                   *
 ****************************************************************************
 * William L. Bahn                                                          *
 * Academy Center for Information Security                                  *
 * Department of Computer Science                                           *
 * United States Air Force Academy                                          *
 * USAFA, CO 80840                                                          *
 ****************************************************************************
 * FILE:............ config.c                                               *
 * DATE CREATED:.... 03 SEP 07                                              *
 * DATE MODIFIED:... 03 SEP 07                                              *
 ****************************************************************************
 *
 * REVISION HISTORY
 *
 ****************************************************************************
 *
 * DESCRIPTION
 *
 * This module imports and manages the configuration information for the
 * modem and the codec.
 *
 */

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------
#include <stdlib.h> // malloc(), free()
#include <math.h>
#include <string.h>
#include <ctype.h>
```

```
#include "config.h"
#include "dirtyd.h"

#define USRP_SAMPLE_SIZE (2*sizeof(float))

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//-----------------------------------------------------------------------------
// PRIVATE FUNCTION DEFINITIONS
//-----------------------------------------------------------------------------

// Nominal String: xxx"filename"xxxx
// If both double quotes are not found, a NULL pointer is returned.

char *ExtractName(char *s)
{
  char *filename;
  char *t;
  int len;

  filename = NULL;

  // Advance s to first double quote or end of string
  while ((*s)&&('\"' != *s))
    s++;
  // If double quote found, advance to next character
  if (*s)
    s++;
  // Advance t to next double quote or end of string
  for (t = s; (*t)&&('\"' != *t); t++)
    EMPTYLOOP;

  // Calculate length of string between first pair of double quotes
  len = t - s;

  t = filename = malloc(len + 1);

  if (filename)
  {
    while (len--)
      *t++ = *s++;
    *t = '\0';
  }

  return filename;
}

// NOTE: The character string may me changed by this function.

void UpdateConfig(CONFIG *c, char *string)
{
  char *s, *v;
  DWORD vi;
  double vf;

  if ((!c)||(!string))
    return;
  // Advance into string to first non-whitespace character
  for (s = string; isspace(*s); s++)
    EMPTYLOOP;

  // Ignore blank or comment lines
  if ((NUL == *s)||('#' == *s))
    return;

  // Identify parameter keyword and convert to uppercase
  for (v = s; (*v) && (!((isspace(*v)) || (':' == *v) || ('=' == *v))); v++)
    *v = toupper(*v);
  // Terminate keyword and start value immediately after (if anything there)
  if (*v)
    *v++ = NUL;

  // Skip over whitespace, colons, and equal signs.
  while ( (isspace(*v)) || (':' == *v) || ('=' == *v) )
    v++;

  // Process those parameters that use string values
  if (!strcmp(s, "PATH"))            c->path = ExtractName(v);
  else if (!strcmp(s, "SOURCE_NAME"))            c->source_name = ExtractName(v);
  else if (!strcmp(s, "SINK_NAME"))              c->sink_name = ExtractName(v);
  else if (!strcmp(s, "DIAGNOSTICS"))
  /*{
    if(!strcmp(v,"True"))  */
      c->diagnostics = TRUE;
  //}
  else
  {
  // Process remaining parameters

  // Extract value from string
  vi = atoi(v);
  vf = atof(v);

  // SCHEDULER Configuration

  if (!strcmp(s, "SCHEDULER_TX_NOTRX"))     c->scheduler_TX_notRX = vi;
  else if (!strcmp(s, "SCHEDULER_REALTIME"))     c->scheduler_realtime = vi;

  // SOURCE Configuration
  else if (!strcmp(s, "SOURCE_ID"))              c->source_id = vi;
  // SOURCE_NAME processed above due to string value

  // CODEC Configuration
  else if (!strcmp(s, "CODEC_MESSAGE_BITS"))     c->codec_message_bits = vi;
  else if (!strcmp(s, "CODEC_RANDOM_BITS"))      c->codec_random_bits = vi;
  else if (!strcmp(s, "CODEC_CLAMP_BITS"))       c->codec_clamp_bits = vi;
  else if (!strcmp(s, "CODEC_FRAGMENT_BITS"))    c->codec_fragment_bits = vi;
  else if (!strcmp(s, "CODEC_STOP_BITS"))        c->codec_stop_bits = vi;
  else if (!strcmp(s, "CODEC_EXPANSION"))        c->codec_expansion = vi;
  else if (!strcmp(s, "CODEC_PACKET_LOAD"))      c->codec_packet_load = vi;
  else if (!strcmp(s, "CODEC_DECODE_LIMIT"))     c->codec_decode_limit = vi;

  // BUFFER Configuration
  else if (!strcmp(s, "BUFFER_PACKETS"))         c->buffer_packets = vi;
  else if (!strcmp(s, "BUFFER_LAMBDA"))          c->buffer_lambda = vf;

  // MODEM Configuration
  else if (!strcmp(s, "MODEM_PACKET_RATE_BPS")) c->modem_packet_rate_bps = vi;
  else if (!strcmp(s, "MODEM_SAMPLES_PER_BIT")) c->modem_samples_per_bit = vi;
  else if (!strcmp(s, "MODEM_GAIN_DB"))          c->modem_gain_dB = vf;
  else if (!strcmp(s, "MODEM_CHANNEL_LOSS_DB")) c->modem_channel_loss_dB = vf;
  else if (!strcmp(s, "MODEM_THRESHOLD_PCT"))    c->modem_threshold_pct = vf;
  else if (!strcmp(s, "MODEM_HYSTERESIS_PCT"))   c->modem_hysteresis_pct = vf;
  else if (!strcmp(s, "MODEM_JITTER_BITS"))      c->modem_jitter_bits = vf;
  else if (!strcmp(s, "MODEM_CUSHION_PCT"))      c->modem_cushion_pct = vf;

  // SINK Configuration
```

```
  // SOURCE_FILE_NAME processed above due to string value
  else if (!strcmp(s, "SINK_SAMPLE_LIMIT"))     c->sink_sample_limit = vi;
  }
}

//---------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//---------------------------------------------------------------------------

CONFIG *CONFIG_Del(CONFIG *p)
{
  if (p)
  {
    if (p->source_name)
    {
      free(p->source_name);
      p->source_name = NULL;
    }
    if (p->sink_name)
    {
      free(p->sink_name);
      p->sink_name = NULL;
    }
    free(p);
    p = NULL;
  }

  return p;
}

CONFIG *CONFIG_New(char *filename, DWORD *errcode)
{
  CONFIG *p;
  FILE *fp;
  DWORD err;
  int i;
  char *s;

  p = NULL;
  err = 0;
  s = NULL;

  p = (CONFIG *) malloc(sizeof(CONFIG));
  if (!p)
    err |= 1 << 0;

  if (!err)
  {
    //----------------------------------------------------------------------
    // NOTE: Establish default values and then overwrite with file data
    //----------------------------------------------------------------------

    p->diagnostics = FALSE;

    // Direction
    p->scheduler_TX_notRX = TRUE;
    p->scheduler_realtime = FALSE;

    // Source Parameters
    p->path = NULL;
    p->source_name = NULL;
    p->source_id = 0;

    // Codec Parameters
    p->codec_message_bits = 512;
    p->codec_random_bits = 0;
    p->codec_clamp_bits = 1;
    p->codec_fragment_bits = 1;
    p->codec_stop_bits = 100;
    p->codec_expansion = 100;
    p->codec_packet_load = 5;
    p->codec_decode_limit = 100;

    // Buffer Parameters
    p->buffer_packets = 2.0;
    p->buffer_lambda = 1.0;

    // Modem Parameters
    p->modem_packet_rate_bps = 500000;
    p->modem_cushion_pct = 10.0;
    p->modem_samples_per_bit = 4;
    p->modem_threshold_pct = 46.3744;
    p->modem_hysteresis_pct = 5.0;
    p->modem_gain_dB = 80.0;
    p->modem_channel_loss_dB = 3.0;
    p->modem_jitter_bits = 3.0;

    // Sink Parameters
    p->sink_name = NULL;
    p->sink_sample_limit = 0;
  p->sink_sample_size_bytes = 0;

  //-----------------------------------------------------------------------
  // Update values from configuration file
  //-----------------------------------------------------------------------

  if (filename)
  {
    fp = fopen(filename, "rt");
    if (fp)
    {
      while (!feof(fp))
      {
        s = fdgets(fp);
        UpdateConfig(p, s);
        if (s)
          free(s);
        s = NULL;
      }
      fclose(fp);
    }
    else
      err |= 1 << 1;
  }

  //-----------------------------------------------------------------------
  // Calculate derived parameters
  //-----------------------------------------------------------------------

  // bitmasks to mask bits within a byte.
  for (i = 0; i < 8; i++)
    p->bitmask[i] = ((BYTE) 1) << i;

  // Set USRP sample size to two floats (complex IQ)
  if (p->scheduler_TX_notRX)
    p->sink_sample_size_bytes = USRP_SAMPLE_SIZE;
  else
    p->source_sample_size_bytes = USRP_SAMPLE_SIZE;

  // Set sink sample limit
  if (!p->sink_sample_limit)
  {
    if (p->scheduler_TX_notRX)
      p->sink_sample_limit = 2000000;
    else
```

```
      p->sink_sample_limit = 1000;
  }

  // Set source filename to default if not set by config file
  if (!p->source_name)
  {
    if (p->scheduler_TX_notRX)
    {
      p->source_name = malloc(strlen("usrp.txd")+1);
      if (p->source_name)
        strcpy(p->source_name, "usrp.txd");
    }
    else
    {
      p->source_name = malloc(strlen("usrp.srp")+1);
      if (p->source_name)
        strcpy(p->source_name, "usrp.srp");
    }
  }

  // Set sink filename to default if not set by config file
  if (!p->sink_name)
  {
    if (p->scheduler_TX_notRX)
    {
      // Sink Parameters
      p->sink_name = malloc(strlen("usrp.srp")+1);
      if (p->sink_name)
        strcpy(p->sink_name, "usrp.srp");
    }
    else
    {
      // Sink Parameters
      p->sink_name = malloc(strlen("usrp.rxd")+1);
      if (p->sink_name)
        strcpy(p->sink_name, "usrp.rxd");
    }
  }

  // Calculate and store derived quantities
  p->bytes_per_message = p->codec_message_bits/8;
  if (p->bytes_per_message % 8)
    p->bytes_per_message++;
  p->packet_bits = (p->codec_message_bits * p->codec_expansion);
  p->last_packet_bit = p->packet_bits - 1;
```

```
p->bytes_per_packet = p->packet_bits/8;
if (p->bytes_per_packet % 8)
  p->bytes_per_packet++;
p->bufferbytes_per_packet = p->bytes_per_packet + 1;
p->buffer_advance = (DWORD) (p->bytes_per_packet * p->buffer_lambda);

p->cushion_bits = (DWORD) (p->packet_bits * p->modem_cushion_pct / 100.0);

p->nominal_tx_signal =  pow(10.0, (p->modem_gain_dB)/20.0);
p->nominal_rx_signal =  pow(10.0, (p->modem_gain_dB - p->modem_channel_loss_dB)/20.0);

// Compute storage requirments for BBC decode tree
if ((0 == p->codec_clamp_bits)||(0 == p->codec_fragment_bits))
{
  p->codec_clamp_bits = 0;
  p->codec_fragment_bits = p->codec_random_bits + p->codec_message_bits;
}
p->fragments = (p->codec_random_bits + p->codec_message_bits)/p->codec_fragment_bits;
p->padded_message_bits = p->fragments * (p->codec_clamp_bits + p->codec_fragment_bits);
if ((p->codec_random_bits + p->codec_message_bits) % p->codec_fragment_bits)
{
  p->padded_message_bits += p->codec_clamp_bits;
  p->padded_message_bits += (p->codec_random_bits + p->codec_message_bits)%p->
       codec_fragment_bits;
}
p->padded_message_bits += p->codec_stop_bits;

//Lookup tables
// 0 and 1 represented as BYTEs that can be passed by reference
p->bitptr[0] = 0;
p->bitptr[1] = 1;

// Tally counters;
p->message_count = 0;

// State information
p->marks = 0;

// Tally Counters
p->actual_trx_bytes = 0;
p->nominal_trx_bytes = 0;
p->dem_ticks = 0;
p->dec_ticks = 0;
p->ticks = 0;
```

```
    p->tot_ticks = 0;

    p->trx_bytes_per_packet_byte = 8 * p->modem_samples_per_bit * USRP_SAMPLE_SIZE;
    p->bytespertick = (  p->modem_packet_rate_bps
                       * p->modem_samples_per_bit
                 * USRP_SAMPLE_SIZE
               ) / ((double)CLOCKS_PER_SEC);
  }
  if (err)
    p = CONFIG_Del(p);


  *errcode = err;
  return p;
}

//------------------------------------------------------------------------
```

A.3.11 dirtyd.h

```
/*
 * =======================================================================
 * PROGRAMMER "BAHN, William"
 * TITLE      "Simple Utilty Functions"
 * CREATED    08 FEB 07
 * MODIFIED   08 FEB 07
 * FILENAME   "dirtyd.c"
 * =======================================================================
 */

/*
 * =======================================================================
 * GENERAL DESCRIPTION
 *
 * This file contains many useful functions - and more are added from time
 * to time.
 * =======================================================================
 */

/*
 * =======================================================================
 * REVISION HISTORY
 *
 * REV 2: 02 DEC 03
 * Added the GetBoundedInt() function
 * Added the InBounds() macro
 * Added the GetDouble() function
 *
 * REV 1: 28 NOV 03
 * Added the PI macro (good to 20 digits)
 * Added the StripCR() function.
 * Added the ClearBuffer() function.
 * Added the WaitForKey() function.
 *
 * REV 0: 09 NOV 03
 *
 * Initial Creation.
 * =====================================================================
 */


#ifndef _DirtyD_H
#define _DirtyD_H

// This directive prevents the prototypes and, most importantly, the
// function definitions (which would normally be in a separate .c file)
// from being included more than once.
//
// At the end of the excluded block of code, the identifier is defined.

//=========================================================================
//== INCLUDE FILES ========================================================
//=========================================================================
#include <stdio.h>  // FILE
#include "bytes.h"

//=========================================================================
//== MACRO DEFINITIONS ====================================================
//=========================================================================

#define FALSE (0)
#define TRUE  (!FALSE)

#define LO (FALSE)
#define HI (TRUE)

#define PI (3.1415926358979323846)

#define RET_DEFAULT (0)
#define RET_CLIPPED (1)
```

```
#define DD_CLIP_NONE   (0x00)
#define DD_CLIP_MIN    (0x01)
#define DD_CLIP_MAX    (0x02)
#define DD_CLIP_MINMAX (0x03)

#define BLANKLINE putc('\n', stdout);
#define EMPTYLOOP {}
#define NUL ('\0')

#define InBounds(min, test, max) ( ((min) <= (test)) && ((test) < (max)) )

//============================================================================
//== FUNCTION PROTOTYPES =====================================================
//============================================================================

// Get input from a stream
char      *fdgets(FILE *fp);
char       fdgetc(FILE *fp);
int        fdgeti(FILE *fp);
long int   fdgetl(FILE *fp);
float      fdgetf(FILE *fp);
double     fdgetd(FILE *fp);

// Get input from stdin
char      *dgets(void);
char       dgetc(void);
int        dgeti(void);
long int   dgetl(void);
float      dgetf(void);
double     dgetd(void);


void   PrintHeader(void);
char  *StripCR(char *s);
char  *GetFileName(char *name, char *ext, int size);
FILE  *OpenAndVerify(char *name, char *mode);
int    rand_int(int min, int max);
double rand_norm(void);
double rand_fp(double xmin, double max);
void   ExitIfError(int errcode);
int    GetBoundedInt(int min, int max, int def, int mode);
double GetBoundedDouble(double min, double max, double def, int mode);
double BoundedDouble(double x, double min, double max, int mode);
double StringToBoundedDouble(char *s, double def, double min, double max, int mode);
```

```

int    GetInt(int min, int max);
double GetDouble(double min, double max);

void *my_memory(FILE *log, void *p, size_t bytes, int action, char *s);
void free1D(void *p);
void *malloc1D(size_t cols, size_t size);
void free2D(void **p, size_t rows);
void **malloc2D(size_t rows, size_t cols, size_t size);
void free3D(void ***p, size_t sheets, size_t rows);
void ***malloc3D(size_t sheets, size_t rows, size_t cols, size_t size);

DWORD Bits2Bytes(DWORD bits);
BYTE *MemorySet(BYTE *p, DWORD bytes, BYTE v);
BYTE *MemoryCopy(BYTE *dest, BYTE *src, DWORD bytes);
void DisplayHEX(FILE *fp, BYTE *p, DWORD bytes, int mode);

BYTE GetBit(BYTE *d, size_t size, DWORD bit);
void SetBit(BYTE *d, size_t size, DWORD bit, int v);

char *ParseString(char *s, char* fdelim, char *tdelim);

DWORD rand_DWORD(DWORD max);

int memequal(char *s1, char *s2, DWORD bytes);

#endif
```

A.3.12 dirtyd.c

```
/*
 * =========================================================================
 * PROGRAMMER "BAHN, William"
 * TITLE      "Simple Utilty Functions"
 * CREATED    08 FEB 07
 * MODIFIED   08 FEB 07
 * FILENAME   "dirtyd.c"
 * =========================================================================
 */

/*
 * =========================================================================
 * GENERAL DESCRIPTION
 *
 * This file contains many useful functions - and more are added from time
 * to time.
 * ======================================================================
 */

/*
 * ======================================================================
 * REVISION HISTORY
 *
 * REV 2: 02 DEC 03
 * Added the GetBoundedInt() function
 * Added the InBounds() macro
 * Added the GetDouble() function

 * REV 1: 28 NOV 03
 * Added the PI macro (good to 20 digits)
 * Added the StripCR() function.
 * Added the ClearBuffer() function.
 * Added the WaitForKey() function.

 * REV 0: 09 NOV 03
 *
 *  Initial Creation.
 * ======================================================================
 */


//========================================================================
//== INCLUDE FILES =======================================================
//========================================================================
#include <stdlib.h> // exit()
#include <string.h> // strlen()
#include <ctype.h>
#include ”dirtyd.h”

//========================================================================
//== FUNCTION DEFINITIONS ================================================
//========================================================================

//========================================================================
// FUNCTION: PrintHeader()
// #include <stdio.h>  // printf()
//========================================================================
#ifdef PROGRAMMER

// This function assumes that the #define statements that create these
// identifiers are used, typically in the function where main() is defined.
//
// By checking if one of them is declared, this function can be skipped if
// necessary so that the other functions can be used. However, if this
// function IS to be available, then it is important that the compiler
// encounter the necessary #define statements before this file is included
// for the very first time.

void PrintHeader(void)
{
  printf("========================================"
         "=======================================\n");
  printf("Course....... %s-%i (%s %i)\n", COURSE, SECTION, TERM, YEAR);
  printf("Programmer... %s (%s)\n", PROGRAMMER, PROG_CODE);
  printf("Assignment... %s (Rev %i) (Source Code in %s)\n",
    ASSIGNMENT, REVISION, FILENAME);
  printf("Description.. %s\n", TITLE);
  printf("              %s\n", SUBTITLE);
  printf("========================================"
         "=======================================\n");
  return;
}
#endif

/*=========================================================================
 * INPUT FUNCTIONS
 *
 * In general, the use of scanf() and its sister functions is to be
 * avoided at nearly all costs. These functions can be quite useful and
 * certainly have their place, but for the vast majority of users, they
 * cause for more problems than they are worth.
 *
 * The preferred method is to use fgets(), which provides enough
 * information to permit quite robust input validation.
 *
 * char *fgets(char *s, int n, FILE *fp)
 *
 * The pointer 's' (which is also the return value of the function) must
 * point to a writeable string in memory containing at least 'n' bytes.
 *
 * The function will read from the input stream 'fp' until either (n-1)
 * bytes or a newline character has been read from the stream, which ever
 * comes first. All bytes read from the stream, including the newline
 * character, are copied to the string pointed to by 's'. The string is
 * then terminated by a NUL character.
 *
 * The reason that the newline character is copied is so that an inspection
 * of the returned string can determine if the entire line was retrieved
 * or if there were too many characters to fit into the available string.
 *
 * PHILOSOPHY
 *
 * Most of the time, Users want to get single items from the keyboard and
 * if something goes wrong (e.g., the User types a string longer than can
 * be handled) then it is usually sufficient to make that known to the
 * program and let the programmer worry about how to deal with it.
 *
 * The "input string longer than the input buffer" problem can be dealt
 * with by using a dynamically allocated buffer that grows to accommodate
 * the length of the string actually entered.
 *
 *===========================================================================
 */

//===========================================================================
//== FUNCTION DEFINITIONS ===================================================
//===========================================================================

char *fdgets(FILE *fp)
{
  char *s;         // Pointer to the dynamically growing string buffer.
  size_t size;     // Present length of the string buffer.
  size_t len;      // Present length of the string in the string buffer.
  int c;           // Character read from the input stream

  s = NULL;
  size = 1;        // Initial size that will be allocated.
  len = 0;

  if (NULL == fp)
    fp = stdin;

  while ( (NULL == s) || (NUL != s[len-1]) )
  {
    // Double the buffer size
    if (2*size < size) // Protect against wrap-around
    {
      if (s)
      {
        free(s);
        s = NULL;
      }
      return s;
    }

    size *= 2;
    s = (char *) realloc(s, size);
    if (NULL == s)
      return s;          // Failed to reallocate string buffer

    // Read in more characters up to the buffer capacity
    do
    {
      c = fgetc(fp);
      s[len++] = ((EOF == c)||('\n' == c))? NUL : (char) c;
    } while ( (len < size) && (NUL != s[len-1]) );
  }
  return s;
}

char fdgetc(FILE *fp)
{
  char *s;
  char n;

  s = fdgets(fp);
  n = 0;
  if (s)
  {
    n = *s;
    free(s);
  }

  return n;
}

int fdgeti(FILE *fp)
{
  char *s;
  int n;

  s = fdgets(fp);
  n = 0;
  if (s)
  {
    n = atoi(s);
    free(s);
  }

  return n;
}

long int fdgetl(FILE *fp)
{
  char *s;
  long int n;

  s = fdgets(fp);
  n = 0;
  if (s)
  {
    n = atol(s);
    free(s);
  }

  return n;
}

float fdgetf(FILE *fp)
{
  char *s;
  float n;

  s = fdgets(fp);
  n = 0;
  if (s)
  {
    n = (float) atof(s);
    free(s);
  }

  return n;
}

double fdgetd(FILE *fp)
{
  char *s;
  double n;

  s = fdgets(fp);
  n = 0;
  if (s)
  {
    n = atof(s);
    free(s);
  }

  return n;
}
// Functions that get only from stdin
char *dgets(void)
{
  return fdgets(stdin);
}

char dgetc(void)
{
  return fdgetc(stdin);
}

int dgeti(void)
{
  return fdgeti(stdin);
}

long int dgetl(void)
{
  return fdgetl(stdin);
}

float dgetf(void)
{
  return fdgetf(stdin);
}

double dgetd(void)
{
  return fdgetd(stdin);
}

/*=========================================================================
 * FUNCTION: StripCR()
 *=========================================================================
 * This functions strips any trailing Carriage Returns from the end of a
 * string. In order to catch carriage returns that might be embedded in
 * the middle of a string, it scans the string from the beginning and looks
 * for a Line Feed, or a Carriage Return and replaces the first occurance
 * with a NULL terminator. The use of a do/while() loop allows the test
 * to operate on the character just examined (and possibly modified) so
 * that is exits correctly regardless of the NULL terminitor found was
 * inserted by the loop or was part of the original string.
 *==========================================================================
 */

char *StripCR(char *s)
{
  int i;

  i = -1;
  do
  {
    switch(s[++i])
    {
      case 10: // Line Feed
      case 13: // Carriage Return
        s[i] = '\0';
    }
  } while('\0' != s[i]);

  return(s);
}

//==========================================================================
// FUNCTION: GetFileName()
//==========================================================================
// This functions gets the a file name from the standard input device and
// returns a string pointer to it. There are several modes in which it can
// be used.
//
// The simplest is to pass null arguments for the name and ext variables
// and 0 for the size. This tells the function to dynamically allocate
// enough memory to accommodate whatever is submitted and to return a
// pointer to the allocated memory.
//
// Example:
//
// char *filename;
//
// filename = GetFileName(NULL, NULL, 0);
//
```

```
// The next easiest way is to allocate memory yourself for the string and
// tell the function where that memory is located. This is most often done
// using a statically allocate character array but previously allocated
// dynamic memory will work the same way. Here you MUST tell the function
// how much memory is available for the string. The function will ensure
// that the string does not exceed the indicated size, including the null
// terminator.
//
// Example:
//
// char filename[13];
//
// GetFileName(filename, NULL, 13);
//
// If you provide a non-NULL pointer for name and you indicate a size of
// zero, the function will assume that the pointer is for previously
// allocated memory that is to be freed and then the pointer re-used to
// point to new memory. Therefore, do NOT pass the name of a static array
// under these conditions as a runtime error will result.
//
// The ext argument can be used to provide a default file extension. If
// the user enters an extension, this parameter will be ignored. If the
// user does not include an extension, the one supplied will be appended.
// Whether the user entered an extension is determined by checking for the
// presence of a period anywhere in the string.
//
// If the given value for ext is NULL, then no extension will be added
// even if the user does not supply one. If the value given for ext is
// a pointer to a null string (i.e., ""), then if an extension is not
// supplied by the user an empty extension will be added - meaning that
// the '.' delimiter will be added but nothing more.

char *GetFileName(char *name, char *ext, int size)
{
  int length;
  char c;
  int endloop;
  int extgiven;

  // Check if size is negative
  if(0>size)
    return(NULL);

  // Check to see if string is static or dynamic
  if((NULL == name) || (0 == size))
```

```
{
  // String is dynamic
  length = 0;
  name = realloc(name, length + 1);
  name[length] = '\0';
}
else
{
  // String is static or fixed length
  if(size < (int) (strlen(ext)+3) ) // Extension too long, ignore it.
    ext = NULL;
}

endloop = FALSE;
extgiven = FALSE;

while(!endloop)
{
  // Check if there is enough room for another character in string.
  if( (0 < size) && !(length < size) )
    endloop = TRUE;

  switch(c = getchar())
  {
    case EOF: // End of File found
    case 10:    // Form Feed encountered
    case 13:    // Carriage Return encountered

            endloop = TRUE;
            break;

    case '.':   // Extension Delimiter found
            extgiven = TRUE;

    default : // All characters (including delimiter above)
            name = realloc(name, length + 2);
            name[length++] = c;
            name[length] = '\0';
            break;
  }
}

// Check if user supplied an extension and use default if appropriate.
if( (!extgiven)&&(NULL != ext) )
{
    if(0 == size) // dynamic array
    {
      // Allocated additional memory for the extension
      name = realloc(name, length + strlen(ext) + 2);
    }
    else // static or fixed length array
    {
      // Ensure that the static array can take the extension
      name[size - strlen(ext) - 2] = '\0';
    }

    strcat(name, ".");
    strcat(name, ext);
  }

  return(name);
}

//=========================================================================
// FUNCTION: OpenAndVerify()
// #include <stdio.h>  // fopen(), FILE, printf()
// #include <stlib.h>  // exit()
//=========================================================================
FILE *OpenAndVerify(char *name, char *mode)
{
  FILE *fp;

  fp = fopen(name, mode);
  if(NULL == fp)
  {
    printf("ABORT! - Failed to open file <%s> (mode %s)\n", name, mode);
    exit(1);
  }
  return fp;
}

//=========================================================================
// FUNCTION: rand_int()
// #include <stlib.h>  // rand()
//=========================================================================
// This function returns a random integer value between min and max
// inclusive.

int rand_int(int min, int max)
{
  return( rand()%( (max-min) + 1) + min );
}

//=========================================================================
// FUNCTION: rand_norm()
// #include <stlib.h>  // rand(), RAND_MAX
//=========================================================================
// This function returns a random floating point value between 0.0 and 1.0
// inclusive.

double rand_norm(void)
{
  return( (double)rand()/(double)RAND_MAX );
}

//=========================================================================
// FUNCTION: rand_fp()
//=========================================================================
// This function returns a random floating point value between min and max
// inclusive.

double rand_fp(double min, double max)
{
  return(min + rand_norm()*(max-min));
}

//=========================================================================
// FUNCTION: ExitIfError()
//=========================================================================
void ExitIfError(int errcode)
{
  if(errcode)
  {
    printf("Abort! (Error #%i detected)\n", errcode);
    exit(errcode);
  }
  return;
}

//=========================================================================
// GetBoundedInt()
// This function gets an int from the keyboard and checks if it is within
// the specified limits. If it is, then that value is returned, otherwise
// the limit that is violated is returned if the mode is set RET_CLIPPED.
// Otherwise, the def(ault) value is returned (use RET_DEFAULT).
```

```
//=====================================================================
int GetBoundedInt(int min, int max, int def, int mode)
{
  int i;

  i = dgeti();

  if(i < min)
    i = (RET_CLIPPED == mode)? min : def;

  if(i > max)
    i = (RET_CLIPPED == mode)? max : def;

  return(i);
}

//=====================================================================
// GetBoundedDouble()
// This function gets a double from the keyboard and checks if it is within
// the specified limits. If it is, then that value is returned, otherwise
// the limit that is violated is returned if the mode is set RET_CLIPPED
// Otherwise, the def(ault) value is returned (use RET_DEFAULT).
//=====================================================================
double GetBoundedDouble(double min, double max, double def, int mode)
{
  double x;

  x = dgetd();

  if(x < min)
    x = (RET_CLIPPED == mode)? min : def;

  if(x > max)
    x = (RET_CLIPPED == mode)? max : def;

  return(x);
}

double BoundedDouble(double x, double min, double max, int mode)
{

  if ((DD_CLIP_MINMAX == mode)||(DD_CLIP_MIN == mode))
    if (x < min)
      x = min;
  if ((DD_CLIP_MINMAX == mode)||(DD_CLIP_MAX == mode))
```

```
    if (x > max)
      x = max;
  return x;
}

double StringToBoundedDouble(char *s, double def, double min, double max,int mode)
{
  double x;

  x = (s)? atof(s) : def;

  return BoundedDouble(x, min, max, mode);
}

//============================================================================
// GetInt()
// This function calls GetBoundedInt with an embedded CLIPPED option.
//============================================================================
int GetInt(int min, int max)
{
  return(GetBoundedInt(min, max, 0, RET_CLIPPED));
}

//============================================================================
// GetDouble()
// This function calls GetBoundedDouble with an embedded CLIPPED option.
//============================================================================
double GetDouble(double min, double max)
{
  return(GetBoundedDouble(min, max, 0, RET_CLIPPED));
}


//============================================================================
// DYNAMICALLY ALLOCATED ARRAYS
//============================================================================

#define MYMEM_MALLOC  (0)
#define MYMEM_FREE    (1)
#define MYMEM_CREATE  (2)
#define MYMEM_DESTROY (3)
#define MYMEM_LINES   (8192)

void *my_memory(FILE *log, void *p, size_t bytes, int action, char *s)
{
```

```
#ifdef MYMEM
  static FILE *memlog = NULL;
  static long int Allocations = 0;
  static long int Deallocations = 0;
  static long int NetAllocations = 0;
  static long int MaxAllocations = 0;
  static long int TotalBytes;

  static size_t *map_bytes = NULL;
  static void   **map_ptrs = NULL;
  static int map_entries = 0;

  int i;
#endif

switch (action)
{
  case MYMEM_CREATE:
    #ifdef MYMEM
      memlog = log;
      if (memlog)
      {
        fprintf(memlog, "%s\n", s);
      }
      map_bytes = (size_t *) my_memory(NULL, NULL, MYMEM_LINES*(sizeof(size_t)), MYMEM_MALLOC, "MAP - bytes");
      map_ptrs  = (void **)  my_memory(NULL, NULL, MYMEM_LINES*(sizeof(void *)), MYMEM_MALLOC, "MAP - ptrs");
      if (map_bytes && map_ptrs)
      {
        for (i = 0; i < MYMEM_LINES; i++)
        {
          map_ptrs[i] = NULL;
          map_bytes[i] = 0;
        }
        map_entries = 0;
      }
    #else
      break;
    #endif
    break;

  case MYMEM_MALLOC:
    p = malloc(bytes);

    #ifdef MYMEM
      Allocations++;
      NetAllocations++;
      TotalBytes += bytes;
      if (NetAllocations > MaxAllocations)
        MaxAllocations = NetAllocations;

      if (memlog)
      {
        fprintf(memlog, "REQUESTED: %6u bytes", bytes);

        if (p)
          fprintf(memlog, " [%p]", p);
        else
          fprintf(memlog, " [--------] DENIED!");

        fprintf(memlog, " Allocs:   %10li (%10li net - %10li)", Allocations, NetAllocations,
            TotalBytes);

        if (s)
          fprintf(memlog, " %s", s);

        fprintf(memlog, "\n");
        fflush(memlog);
      }

      if (map_bytes && map_ptrs)
      {
        if (map_entries < MYMEM_LINES)
        {
          map_ptrs[map_entries] = p;
          map_bytes[map_entries] = bytes;
          map_entries++;
        }
        else
        {
          fprintf(memlog, "Pointer Map entry limit exceeded\n");
        }
      }
    #endif

    break;

  case MYMEM_FREE:

```

```
#ifdef MYMEM
  Deallocations++;
  NetAllocations--;

  if (memlog)
  {
    fprintf(memlog, "FREEING..:                ");

    if (p)
      fprintf(memlog, " [%p]", p);
    else
      fprintf(memlog, " [--------] NULL PTR!");

    fprintf(memlog, " Deallocs: %10li (%10li net - %10li)", Deallocations, NetAllocations,
         TotalBytes);

    if (s)
      fprintf(memlog, " %s", s);

    fprintf(memlog, "\n");

    fflush(memlog);
  }
  if (p)
  {
    if (map_bytes && map_ptrs)
    {
      for (i = (map_entries-1); (i >= 0) && (p != map_ptrs[i]); i--)
        ; // EMPTY LOOP;
      if (i >= 0)
      {
        TotalBytes -= map_bytes[i];
        map_entries--;
        while (i < map_entries)
        {
          map_ptrs[i] = map_ptrs[i+1];
          map_bytes[i] = map_bytes[i+1];
          i++;
        }
      }
      else
      {
        fprintf(memlog, "Pointer Map entry not found!\n");
      }
    }
```

```
        }
      #endif

      if (p)
        free(p);

      break;

    case MYMEM_DESTROY:

      #ifdef MYMEM
        if (memlog)
        {
          fprintf(memlog, "============================================================\n");
          fprintf(memlog, "%s\n", s);
          fprintf(memlog, "RESIDUAL MEMORY ALLOCATIONS\n");
          fprintf(memlog, "============================================================\n");

          if (map_bytes && map_ptrs)
            for (i = 0; i < map_entries; i++)
            {
              if (map_ptrs[i] = NULL)
                fprintf(memlog, "[%p] %li bytes\n", map_ptrs[i], ((long int) map_bytes[i]));
            }
          fprintf(memlog, "============================================================\n");
        }

        my_memory(NULL, map_bytes, 0, MYMEM_FREE, "MAP - bytes");
        my_memory(NULL, map_ptrs,  0, MYMEM_FREE, "MAP - ptrs");
      #else
        break;
      #endif

      break;

    default:
      break;
  }
  return p;
}

void free1D(void *p)
{
  my_memory(NULL, p, 0, MYMEM_FREE, "1D");
}
```

```

void *malloc1D(size_t cols, size_t size)
{
  void *array;
  size_t bytes;

  bytes = cols*size;

  if (0 == bytes)
    return NULL;

  array = my_memory(NULL, NULL, bytes, MYMEM_MALLOC, "1D");

  return array;
}


void free2D(void **p, size_t rows)
{
  if(p)
    while (rows--)
      if (p[rows])
        my_memory(NULL, p[rows], 0, MYMEM_FREE, "2D - row");
  my_memory(NULL, p, 0, MYMEM_FREE, "2D - base");
}

void **malloc2D(size_t rows, size_t cols, size_t size)
{
  void **array;
  size_t i;
  size_t bytes;

  if (!(rows && cols && size))
    return NULL;

  bytes = rows * sizeof(void*);

  if (NULL == (array = my_memory(NULL, NULL, bytes, MYMEM_MALLOC, "2D - base")))
    return NULL;

  for (i = 0; i < rows; i++)
  {
    if (NULL == ( array[i] = malloc1D(cols, size) ))
    {
      while (i)
      {
        i--;
        my_memory(NULL, array[i], 0, MYMEM_FREE, "2D failed - row");
      }
      my_memory(NULL, array, 0, MYMEM_FREE, "2D failed - base");
      i = rows;
    }
  }

  return array;
}

void free3D(void ***p, size_t sheets, size_t rows)
{
  if (p)
    while (sheets--)
      if (p[sheets])
        free2D(p[sheets], rows);
  my_memory(NULL, p, 0, MYMEM_FREE, "3D - base");
}

void ***malloc3D(size_t sheets, size_t rows, size_t cols, size_t size)
{
  void ***array;
  size_t i;
  size_t bytes;

  if (!(rows && cols && size))
    return NULL;

  bytes = sheets * sizeof(void*);

  if (NULL == (array = my_memory(NULL, NULL, bytes, MYMEM_MALLOC, "3D - base")))
    return NULL;

  for (i = 0; i < sheets; i++)
  {
    if (NULL == ( array[i] = malloc2D(rows, cols, size) ))
    {
      while (i)
      {
        i--;
        my_memory(NULL, array[i], 0, MYMEM_FREE, "3D failed - row");
      }
      my_memory(NULL, array, 0, MYMEM_FREE, "3D failed - base");
      i = sheets;
    }
  }

  return array;
}

DWORD Bits2Bytes(DWORD bits)
{
  return (bits / (8*sizeof(BYTE))) + ((bits % (8*sizeof(BYTE)))? 1:0);
}

BYTE *MemorySet(BYTE *p, DWORD bytes, BYTE v)
{
  DWORD byte;

  if (p)
    for (byte = 0; byte < bytes; byte++)
      p[byte] = v;

  return p;
}

BYTE *MemoryCopy(BYTE *dest, BYTE *src, DWORD bytes)
{
  DWORD i;

  for (i = 0; i < bytes; i++)
    dest[i] = src[i];

  return dest;
}

void DisplayHEX(FILE *fp, BYTE *p, DWORD bytes, int mode)
{
  DWORD byte;
  DWORD line;
  WORD i;

  switch (mode)
  {
    case 0:
    case 1:
    default:
      fprintf(fp, "\n");
      fprintf(fp, "  --------------------------------------");
      fprintf(fp, "---------------------------------------");
      fprintf(fp, "\n");
      fprintf(fp, "           ");
      for (i = 0; i < 16; i++)
      {
        fprintf(fp, "%2X ", i);
      }
      fprintf(fp, "- ");
      for (i = 0; i < 16; i++)
      {
        fprintf(fp, "%1X", i);
      }
      fprintf(fp, "\n");
      fprintf(fp, "  --------------------------------------");
      fprintf(fp, "---------------------------------------");
      fprintf(fp, "\n");
      for (line = byte = 0; byte < bytes; line++, byte+=16)
      {
        fprintf(fp, "  [%06X] ", line);
        for (i = 0; i < 16; i++)
        {
          if (byte+i < bytes)
            fprintf(fp, "%02X ", p[byte+i]);
          else
            fprintf(fp, "-- ");
        }
        fprintf(fp, "- ");
        for (i = 0; i < 16; i++)
        {
          if (byte+i < bytes)
            fprintf(fp, "%1c", (isprint(p[byte+i])? p[byte+i]:'.'));
          else
            fprintf(fp, " ");
        }
        fprintf(fp, "\n");
      }
      fprintf(fp, "  --------------------------------------");
      fprintf(fp, "---------------------------------------");
      fprintf(fp, "\n");
      break;
  }
}
```

```

WORD GetBitIndex(DWORD bit)
{
  WORD index;

  index = bit/8;

  return index;
}

BYTE GetBitMask(DWORD bit)
{
  BYTE offset;
  BYTE mask;

  mask = 0x80;
  offset = bit%8;
  mask >>= offset;

  return mask;
}

BYTE GetBit(BYTE *d, size_t size, DWORD bit)
{
  WORD index;
  BYTE mask;

  index = GetBitIndex(bit);
  mask  = GetBitMask(bit);

  return (d[index] & mask)? 1 : 0;
}


void SetBit(BYTE *d, size_t size, DWORD bit, int v)
{
  WORD index;
  BYTE mask;

  index = GetBitIndex(bit);
  mask  = GetBitMask(bit);

  if (v)
    d[index] |= mask;
  else
```

```
    d[index] &= ~mask;

}

typedef struct STRINGPARSER STRINGPARSER;

struct STRINGPARSER
{
  char *string;
  int length;
  char *next;
};

int IsIn(char c, char *s)
{
  int i;

  for (i = 0; s[i] && (c != s[i]); i++)
    EMPTYLOOP;

  return s[i];
}

char *ParseString(char *s, char* fdelim, char *tdelim)
{
  static STRINGPARSER *p = NULL;
  int i, n;
  char *substring;

  if (!p)
  {
    p = (STRINGPARSER *) malloc(sizeof(STRINGPARSER));
    if (p)
    {
      p->string = NULL;
      p->length = 0;
      p->next = NULL;
    }
    else
      return NULL;
  }

  if (NULL == s)
  {
    free(p);
    p = NULL;
    return NULL;
  }

  if (p->string != s)
  {
    p->string = s;
    p->length = strlen(s);
    p->next = s;
  }

  for (; (p->next < (p->string + p->length)) && (IsIn(*(p->next), fdelim)); p->next++)
    EMPTYLOOP;

  if ((p->next - p->string) >= p->length)
  {
    p->string = NULL;
    p->length = 0;
    p->next = NULL;
    return NULL;
  }

  for (n = 0; (p->next+n < (p->string + p->length)) && (!IsIn(*(p->next+n), tdelim)); n++);
    EMPTYLOOP;

  substring = malloc(n+1);

  if (substring)
  {
    for (i = 0; i < n; i++)
      substring[i] = p->next[i];
    substring[n] = NUL;
  }

  p->next += n;

  return substring;
}

DWORD rand_DWORD(DWORD max)
{
  DWORD mask, value;

  for (mask = 1; mask < max; mask = (mask<<1) + 1)
    EMPTYLOOP;
  do
  {
    value = (rand()<<(8*sizeof(WORD))) + rand();
    value &= mask;
  } while (value > max);

  return value;
}

int memequal(char *s1, char *s2, DWORD bytes)
{
 DWORD i;

  for (i = 0; i < bytes; i++)
    if (s1[i] != s2[i])
      return FALSE;

  return TRUE;
}
```

A.3.13 modem.h

```
/*****************************************************************************
 * MODEM for the Real-time BBC Codec/Modem                                   *
 *****************************************************************************
 * William L. Bahn                                                           *
 * Academy Center for Information Security                                   *
 * Department of Computer Science                                            *
 * United States Air Force Academy                                           *
 * USAFA, CO 80840                                                           *
 *****************************************************************************
 * FILE:............ modem.h                                                 *
 * DATE CREATED:.... 06 SEP 07                                               *
 * DATE MODIFIED:... 06 SEP 07                                               *
 *****************************************************************************
 *
 * REVISION HISTORY
 *
 *****************************************************************************
 *
 * DESCRIPTION
 *
 * The modem converts baseband signal data to/from packet data.
 *
 */
```

```
#ifndef MODEMdotH
#define MODEMdotH

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------

#include <time.h>  // clock_t

#include "config.h"
#include "source.h"
#include "buffer.h"
#include "sink.h"
#include "dirtyd.h"

//-----------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
//-----------------------------------------------------------------------------

typedef struct MODEM MODEM;

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct MODEM
{
  // Derived quantities
  DWORD jitter_samples;
  double alpha;
  double t_hi, t_lo;

  // State information
  DWORD state;
  double integrator;
  SDWORD stamp;
};
```

```
//-----------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//-----------------------------------------------------------------------------

MODEM *MODEM_Del(MODEM *p);
MODEM *MODEM_New(CONFIG *c, DWORD *errcode);
void Modulate(CONFIG *c, BUFFER *buffer, MODEM *modem, SINK *sink);
void Demodulate(CONFIG *c, SOURCE *source, MODEM *modem, BUFFER *buf);

//-----------------------------------------------------------------------------
#endif
```

A.3.14 modem.c

```
/*************************************************************************
 * MODEM for the Real-time BBC Codec/Modem                               *
 *************************************************************************
 * William L. Bahn                                                       *
 * Academy Center for Information Security                               *
 * Department of Computer Science                                        *
 * United States Air Force Academy                                       *
 * USAFA, CO 80840                                                       *
 *************************************************************************
 * FILE:............ modem.c                                             *
 * DATE CREATED:.... 06 SEP 07                                           *
 * DATE MODIFIED:... 28 FEB 09                                           *
 *************************************************************************
 *
 * REVISION HISTORY
 *     Modified to support only the requirements of providing same symbol
 *     rate data as a means to create a jammer.
 *     2/28/2009 Derek Sanders
 *
 *************************************************************************
 *
 * DESCRIPTION
 *
 * The modem and its public interface is described in modem.h.
 *
 *************************************************************************
 */

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------
```

```
#include <stdlib.h> // malloc()
#include <math.h>   // exp()
#include "modem.h"

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//-----------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//-----------------------------------------------------------------------------

MODEM *MODEM_Del(MODEM *p)
{
  if (p)
  {
    free(p);
  }
  return NULL;
}

MODEM *MODEM_New(CONFIG *c, DWORD *errcode)
{
  MODEM *p;
  DWORD err;
  double nominal_steady_state_peak;

  p = NULL;
  err = 0;

  p = (MODEM *) malloc(sizeof(MODEM));
  if (!p)
    err |= 1 << 0;

  if (!err)
  {
    // Derived quantities
    p->jitter_samples = (int)(c->modem_samples_per_bit * c->modem_jitter_bits);
```

```c

    // Integrator parameter
    p->alpha = exp((2.0/c->modem_samples_per_bit) - 1.0);

    // Threshold parameters
    nominal_steady_state_peak = (c->nominal_rx_signal*c->nominal_rx_signal) * (1.0/(1.0-p->alpha));
    p->t_hi = nominal_steady_state_peak * ((c->modem_threshold_pct + c->modem_hysteresis_pct/2.0)/100.0);
    p->t_lo = nominal_steady_state_peak * ((c->modem_threshold_pct - c->modem_hysteresis_pct/2.0)/100.0);

    // State information
    p->state = 0;
    p->integrator = 0.0;
    p->stamp = 0;
  }

  if (err)
    p = MODEM_Del(p);

  if (c->diagnostics)
  {
    // Diagnostic Report
    printf("--------------------------------------------\n");
    printf("MODEM\n");
    printf("  Creation:.................. %s\n", ((err)? "FAILED":"SUCCEEDED"));
    printf("  Location:.................. %p\n", (void *) p);
    printf("  Integrator alpha:.......... %f\n", p->alpha);
    printf("  Jitter tolerance:.......... %f\n", p->jitter_samples);
    printf("  Modem gain:................ %f (%f dB)\n", c->nominal_tx_signal, c->modem_gain_dB);
    printf("  Nominal channel loss:...... %f dB\n", c->modem_channel_loss_dB);
    printf("  Nominal rx signal peak:.... %f (%f dB)\n", c->nominal_rx_signal, (c->modem_gain_dB-c->modem_channel_loss_dB));
    printf("  Nominal integrator peak:... %f\n", nominal_steady_state_peak);
    printf("  LO -> HI threshold:........ %f\n", p->t_hi);
    printf("  HI -> LO threshold:........ %f\n", p->t_lo);
    printf("--------------------------------------------\n");
  }

  *errcode = err;
  return p;
}

//----------------------------------------------------------------------------
```

```
/* MODEM
 *
 * The MODEM reads/writes USRP in bursts of samples corresponding to
 * 8 packet bits. The calling function is responsible for ensuring that
 * valid data and/or sufficient room for new data exists in the buffer.
 *
 */

/* MODULATOR
 *
 * The modulator reads one byte of packet data from the buffer and generates
 * USRP data for the entire set of 8 packet bits.
 *
 */

void Modulate(CONFIG *c, BUFFER *buffer, MODEM *modem, SINK *sink)
{
  DWORD originbit, sample;
  float signal;
  clock_t ticks;
  float *v;
  ticks = clock();

  // Push write pointer if packet byte is not available
  if (!buffer->ready)
  {
    buffer->write = (buffer->write + 1) & buffer->buffermask;
    buffer->ready++;
    buffer->margin--;
  }

  // For each bit in the packet byte at the buffer's read pointer
  for (originbit = 0; originbit < 8; originbit++)
  {
    // Determine if the bit is a mark or a space
    if (buffer->buffer[buffer->read] & c->bitmask[originbit])
    {
      c->marks++;
      signal = (float) c->nominal_tx_signal;
    }
    else
      signal = 0.0;

    // Determine if the sink can take all the samples for the present bit
    if (sink->samples + c->modem_samples_per_bit < sink->sample_limit)
    {
      // Establish the base location within the sink's buffer
      v = ((float *) sink->v) + (2 * sink->samples);

      // Generate and write the baseband samples to the sink
      for (sample = 0; sample < c->modem_samples_per_bit; sample++)
      {
        v[2*sample]     = signal; // I(t) (actual data)
        v[2*sample + 1] = 0.0;    // Q(t) (forced to zero)
      }
      sink->samples += c->modem_samples_per_bit;
    }
    else
      sink->streaming = FALSE;
  }

  buffer->buffer[buffer->read] = 0;
  buffer->read = (buffer->read + 1) & buffer->buffermask;
  buffer->ready--;
  buffer->margin++;

  c->actual_trx_bytes += c->trx_bytes_per_packet_byte;
  c->dem_ticks += clock() - ticks;
}

void Demodulate(CONFIG *c, SOURCE *source, MODEM *modem, BUFFER *buf)
{
  DWORD sample;
  DWORD originbit;
  clock_t ticks;
  float *v;
  double v2;

  ticks = clock();

  for (originbit = 0; originbit < 8; originbit++)
  {
    v = ((float *) source->v) + (2 * source->samples);
    for (sample = 0; sample < c->modem_samples_per_bit; sample++)
    {
      if (source->samples < source->sample_limit)
      {

        v2 = v[2*sample] * v[2*sample]  + v[2*sample+1] * v[2*sample+1];
        source->samples++;
      }
      else
      {
        v2 = 0;
        source->streaming = FALSE;
      }

      modem->integrator = v2 + modem->alpha*(modem->integrator - v2);

      switch (modem->state)
      {
        case 0:
          if (modem->integrator > modem->t_hi)
          {
            modem->state = 1;
          }
          break;
        case 1:
          if (modem->integrator < modem->t_lo)
          {
            modem->state = 2;
            modem->stamp = (SDWORD) (sample + modem->jitter_samples);
          }
          break;
        case 2:
          if (modem->integrator > modem->t_hi)
            modem->state = 1;
          else
            if (((SDWORD) sample > modem->stamp)&&(modem->integrator < modem->t_lo))
            {
              modem->state = 0;
            }
          break;
      }
    }
    modem->stamp -= c->modem_samples_per_bit;

    if (0 == buf->empty)
    {
      buf->read = (buf->read + 1) & buf->buffermask;
      buf->empty++;
      buf->margin--;
      buf->overflows++;
    }
```

```

      // Step packet forward and mark next location
      if (modem->state > 0)
      {
        c->marks++;
        buf->buffer[buf->write] |= c->bitmask[originbit];
      }
      else
        buf->buffer[buf->write] &= ~c->bitmask[originbit];
    }

    buf->write = (buf->write + 1) & buf->buffermask;
    buf->margin++;
    buf->empty--;

    c->actual_trx_bytes += c->trx_bytes_per_packet_byte;
    c->dem_ticks += clock() - ticks;
  }

  //---------------------------------------------------------------------------
```

A.3.15 sha1.h

```
/*
 *  sha1.h
 *
 *  Copyright (C) 1998
 *  Paul E. Jones <paulej@arid.us>
 *  All Rights Reserved
 *
 *****************************************************************************
 *  $Id: sha1.h,v 1.2 2004/03/27 18:00:33 paulej Exp $
 *****************************************************************************
 *
 *  Description:
 *      This class implements the Secure Hashing Standard as defined
 *      in FIPS PUB 180-1 published April 17, 1995.
 *
 *      Many of the variable names in the SHA1Context, especially the
 *      single character names, were used because those were the names
 *      used in the publication.
 *
 *      Please read the file sha1.c for more information.
 *
 */
```

```
#ifndef _SHA1_H_
#define _SHA1_H_

/*
 *  This structure will hold context information for the hashing
 *  operation
 */

typedef struct SHA1Context
{
    unsigned Message_Digest[5]; /* Message Digest (output)          */

    unsigned Length_Low;        /* Message length in bits           */
    unsigned Length_High;       /* Message length in bits           */

    unsigned char Message_Block[64]; /* 512-bit message blocks      */
    int Message_Block_Index;    /* Index into message block array   */

    int Computed;               /* Is the digest computed?          */
    int Corrupted;              /* Is the message digest corruped?  */
} SHA1Context;

/*
 *  Function Prototypes
 */
void SHA1Reset(SHA1Context *);
int SHA1Result(SHA1Context *);
void SHA1Input( SHA1Context *,
                const unsigned char *,
                unsigned);

#endif
```

A.3.16 sha1.c

```
/*
 *  sha1.c
 *
 *  Copyright (C) 1998
 *  Paul E. Jones <paulej@arid.us>
 *  All Rights Reserved
 *
 *****************************************************************************
 *  $Id: sha1.c,v 1.2 2004/03/27 18:00:33 paulej Exp $
 *****************************************************************************
 *
 *  Description:
 *      This file implements the Secure Hashing Standard as defined
 *      in FIPS PUB 180-1 published April 17, 1995.
 *
 *      The Secure Hashing Standard, which uses the Secure Hashing
 *      Algorithm (SHA), produces a 160-bit message digest for a
 *      given data stream.  In theory, it is highly improbable that
 *      two messages will produce the same message digest.  Therefore,
 *      this algorithm can serve as a means of providing a "fingerprint"
 *      for a message.
 *
 *  Portability Issues:
 *      SHA-1 is defined in terms of 32-bit "words".  This code was
 *      written with the expectation that the processor has at least
 *      a 32-bit machine word size.  If the machine word size is larger,
 *      the code should still function properly.  One caveat to that
 *      is that the input functions taking characters and character
 *      arrays assume that only 8 bits of information are stored in each
 *      character.
 *
 *  Caveats:
 *      SHA-1 is designed to work with messages less than 2^64 bits
 *      long. Although SHA-1 allows a message digest to be generated for
 *      messages of any number of bits less than 2^64, this
 *      implementation only works with messages with a length that is a
 *      multiple of the size of an 8-bit character.
 *
 */

#include "sha1.h"

/*
 *  Define the circular shift macro
 */
#define SHA1CircularShift(bits,word) \
                ((((word) << (bits)) & 0xFFFFFFFF) | \
                ((word) >> (32-(bits))))

/* Function prototypes */
void SHA1ProcessMessageBlock(SHA1Context *);
void SHA1PadMessage(SHA1Context *);

/*
```

```
 *  SHA1Reset
 *
 *  Description:
 *      This function will initialize the SHA1Context in preparation
 *      for computing a new message digest.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to reset.
 *
 *  Returns:
 *      Nothing.
 *
 *  Comments:
 *
 */
void SHA1Reset(SHA1Context *context)
{
    context->Length_Low             = 0;
    context->Length_High            = 0;
    context->Message_Block_Index    = 0;

    context->Message_Digest[0]      = 0x67452301;
    context->Message_Digest[1]      = 0xEFCDAB89;
    context->Message_Digest[2]      = 0x98BADCFE;
    context->Message_Digest[3]      = 0x10325476;
    context->Message_Digest[4]      = 0xC3D2E1F0;

    context->Computed   = 0;
    context->Corrupted  = 0;
}

/*
 *  SHA1Result
 *
 *  Description:
 *      This function will return the 160-bit message digest into the
 *      Message_Digest array within the SHA1Context provided
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to use to calculate the SHA-1 hash.
 *
 *  Returns:
 *      1 if successful, 0 if it failed.
 *
 *  Comments:
 *
 */
int SHA1Result(SHA1Context *context)
{

    if (context->Corrupted)
    {
        return 0;
    }

    if (!context->Computed)
    {
        SHA1PadMessage(context);
        context->Computed = 1;
    }

    return 1;
}

/*
 *  SHA1Input
 *
 *  Description:
 *      This function accepts an array of octets as the next portion of
 *      the message.
 *
 *  Parameters:
 *      context: [in/out]
 *          The SHA-1 context to update
 *      message_array: [in]
 *          An array of characters representing the next portion of the
 *          message.
 *      length: [in]
 *          The length of the message in message_array
 *
 *  Returns:
 *      Nothing.
 *
 *  Comments:
 *
 */
void SHA1Input(     SHA1Context         *context,
                    const unsigned char *message_array,
                    unsigned            length)
{
    if (!length)
    {
        return;
    }

    if (context->Computed || context->Corrupted)
    {
        context->Corrupted = 1;
        return;
    }

    while(length-- && !context->Corrupted)
    {
        context->Message_Block[context->Message_Block_Index++] =
                                                (*message_array & 0xFF);

        context->Length_Low += 8;
        /* Force it to 32 bits */
        context->Length_Low &= 0xFFFFFFFF;
        if (context->Length_Low == 0)
        {
            context->Length_High++;
            /* Force it to 32 bits */
            context->Length_High &= 0xFFFFFFFF;
            if (context->Length_High == 0)
            {
                /* Message is too long */
                context->Corrupted = 1;
            }
        }

        if (context->Message_Block_Index == 64)
        {
            SHA1ProcessMessageBlock(context);
        }

        message_array++;
    }
}

/*
 *  SHA1ProcessMessageBlock
 *
 *  Description:
 *      This function will process the next 512 bits of the message
 *      stored in the Message_Block array.
 *
 *  Parameters:
 *      None.
 *
 *  Returns:
 *      Nothing.
 *
 *  Comments:
 *      Many of the variable names in the SHAContext, especially the
 *      single character names, were used because those were the names
 *      used in the publication.
 *
 *
 */
void SHA1ProcessMessageBlock(SHA1Context *context)
{
    const unsigned K[] =            /* Constants defined in SHA-1   */
    {
        0x5A827999,
        0x6ED9EBA1,
        0x8F1BBCDC,
        0xCA62C1D6
    };
    int         t;                  /* Loop counter                */
    unsigned    temp;               /* Temporary word value        */
    unsigned    W[80];              /* Word sequence               */
    unsigned    A, B, C, D, E;      /* Word buffers                */

    /*
     *  Initialize the first 16 words in the array W
     */
    for(t = 0; t < 16; t++)
    {
        W[t] = ((unsigned) context->Message_Block[t * 4]) << 24;
        W[t] |= ((unsigned) context->Message_Block[t * 4 + 1]) << 16;
        W[t] |= ((unsigned) context->Message_Block[t * 4 + 2]) << 8;
        W[t] |= ((unsigned) context->Message_Block[t * 4 + 3]);
    }

    for(t = 16; t < 80; t++)
    {
       W[t] = SHA1CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
    }

    A = context->Message_Digest[0];
    B = context->Message_Digest[1];
    C = context->Message_Digest[2];
    D = context->Message_Digest[3];
    E = context->Message_Digest[4];

    for(t = 0; t < 20; t++)
    {
        temp =  SHA1CircularShift(5,A) +
                ((B & C) | ((~B) & D)) + E + W[t] + K[0];
        temp &= 0xFFFFFFFF;
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 20; t < 40; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
        temp &= 0xFFFFFFFF;
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 40; t < 60; t++)
    {
        temp = SHA1CircularShift(5,A) +
               ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
        temp &= 0xFFFFFFFF;
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    for(t = 60; t < 80; t++)
    {
        temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
        temp &= 0xFFFFFFFF;
        E = D;
        D = C;
        C = SHA1CircularShift(30,B);
        B = A;
        A = temp;
    }

    context->Message_Digest[0] =
                        (context->Message_Digest[0] + A) & 0xFFFFFFFF;
    context->Message_Digest[1] =
                        (context->Message_Digest[1] + B) & 0xFFFFFFFF;
    context->Message_Digest[2] =
                        (context->Message_Digest[2] + C) & 0xFFFFFFFF;
    context->Message_Digest[3] =
                        (context->Message_Digest[3] + D) & 0xFFFFFFFF;
    context->Message_Digest[4] =
                        (context->Message_Digest[4] + E) & 0xFFFFFFFF;

    context->Message_Block_Index = 0;
}

/*
 *  SHA1PadMessage
 *
 *  Description:
 *      According to the standard, the message must be padded to an even
 *      512 bits.  The first padding bit must be a '1'.  The last 64
 *      bits represent the length of the original message.  All bits in
 *      between should be 0.  This function will pad the message
 *      according to those rules by filling the Message_Block array
 *      accordingly.  It will also call SHA1ProcessMessageBlock()
 *      appropriately.  When it returns, it can be assumed that the
 *      message digest has been computed.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to pad
 *
 *  Returns:
 *      Nothing.
 *
 *  Comments:
 *
 */
void SHA1PadMessage(SHA1Context *context)
{
    /*
     *  Check to see if the current message block is too small to hold
     *  the initial padding bits and length.  If so, we will pad the
     *  block, process it, and then continue padding into a second
     *  block.
     */
    if (context->Message_Block_Index > 55)
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 64)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }

        SHA1ProcessMessageBlock(context);

        while(context->Message_Block_Index < 56)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }
    else
    {
        context->Message_Block[context->Message_Block_Index++] = 0x80;
        while(context->Message_Block_Index < 56)
        {
            context->Message_Block[context->Message_Block_Index++] = 0;
        }
    }

    /*
     *  Store the message length as the last 8 octets
     */
    context->Message_Block[56] = (context->Length_High >> 24) & 0xFF;
    context->Message_Block[57] = (context->Length_High >> 16) & 0xFF;
    context->Message_Block[58] = (context->Length_High >> 8) & 0xFF;
    context->Message_Block[59] = (context->Length_High) & 0xFF;
    context->Message_Block[60] = (context->Length_Low >> 24) & 0xFF;
    context->Message_Block[61] = (context->Length_Low >> 16) & 0xFF;
    context->Message_Block[62] = (context->Length_Low >> 8) & 0xFF;
    context->Message_Block[63] = (context->Length_Low) & 0xFF;

    SHA1ProcessMessageBlock(context);
}
```

A.3.17 sink.h

```
/**************************************************************************
 * Signal Sink Module for the Real-time BBC Codec/Modem                   *
 **************************************************************************
 * William L. Bahn                                                        *
 * Academy Center for Information Security                                *
 * Department of Computer Science                                         *
 * United States Air Force Academy                                        *
 * USAFA, CO 80840                                                        *
 **************************************************************************
 * FILE:............ sink.h                                               *
 * DATE CREATED:.... 08 SEP 07                                            *
 * DATE MODIFIED:... 08 SEP 07                                            *
 **************************************************************************
 *
 * REVISION HISTORY
 *
 **************************************************************************
 *
 * DESCRIPTION
 *
 * This module supports the signal sink for both the TX and the RX
 *
 */

#ifndef SINKdotH
#define SINKdotH

//-------------------------------------------------------------------------
// REQUIRED INCLUDES
//-------------------------------------------------------------------------

#include "config.h"
#include "dirtyd.h"

//-------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
//-------------------------------------------------------------------------

typedef struct SINK SINK;
```

```
//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct SINK
{
  FILE *fp;
  int    streaming;
  DWORD samples;
  DWORD sample_size_bytes;
  DWORD sample_limit;
  BYTE *v;

  size_t buffer_size;
};

//-----------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//-----------------------------------------------------------------------------

SINK *SINK_Del(SINK *p);
SINK *SINK_New(CONFIG *config, DWORD *errcode);
void SINK_Purge(CONFIG *config, SINK *p);

//-----------------------------------------------------------------------------
#endif
```

A.3.18 sink.c

```
/*****************************************************************************
 * Signal Sink Module for the Real-time BBC Codec/Modem                      *
 *****************************************************************************
 * William L. Bahn                                                           *
 * Academy Center for Information Security                                   *
 * Department of Computer Science                                            *
 * United States Air Force Academy                                           *
 * USAFA, CO 80840                                                           *
 *****************************************************************************
 * FILE:........... sink.c                                                   *
 * DATE CREATED:....  08 SEP 07                                              *
 * DATE MODIFIED:...  08 SEP 07                                              *
 ******************************************************************************
 *
 * REVISION HISTORY
 *
 ******************************************************************************
 *
 * DESCRIPTION
 *
 * This module supports the signal sink for both the TX and the RX
 *
 */

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------

#include <stdlib.h> // malloc(), free()
#include <string.h> // memmove()

#include ”sink.h”
#include ”bbcftp.h”
#include ”dirtyd.h”

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//-----------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//-----------------------------------------------------------------------------

SINK *SINK_Del(SINK *p)
{
  if (p)
  {
    if (p->fp)
      if (stdout != p->fp)
      {
        fclose(p->fp);
        p->fp = NULL;
      }
    if (p->v)      { free(p->v);   p->v = NULL;        }
    free(p);
    p = NULL;
  }

  return p;
}

// Sufficient memory is allocated up front
// to handle a maximum amount of data. However, the present
// contents of the buffer can be purged using SINK_Purge().

SINK *SINK_New(CONFIG *c, DWORD *errcode)
{
  SINK *p;
  DWORD err;

  p = NULL;
  err = 0;

  p = (SINK *) malloc(sizeof(SINK));
  if (!p)
    err |= 1 << 0;

  // Open Data Sink file
  if (!err)
  {
    /*p->fp = NULL;
    if (c->sink_name)
    {
      char path[256];
      strcpy(path, c->path);
      strcat(path,c->sink_name);
      p->fp = fopen(path, ”wb”);
      if (!p->fp)
        err |= 1 << 7;
    }
    else
      p->fp = stdout;*/
  }

```

```
// Initialize state information
if (!err)
{
  p->samples = 0;
  p->streaming = TRUE;

  if (c->sink_sample_limit)
    p->sample_limit = c->sink_sample_limit;
  else
  {
    if (c->scheduler_TX_notRX)
    {
      p->sample_limit = 4*c->modem_samples_per_bit*c->packet_bits;
    }
    else
    {
      p->sample_limit = 1000;
    }
  }

  if (c->sink_sample_size_bytes)
    p->sample_size_bytes = c->sink_sample_size_bytes;
  else
  {
    if (c->scheduler_TX_notRX)
    {
      p->sample_size_bytes = 2*sizeof(float);
    }
    else
    {
      // One byte for each eight full bits of message
      p->sample_size_bytes = c->codec_message_bits / 8;
      // Add a final byte, if necessary, to hold leftover bits
      if (c->codec_message_bits % 8)
        p->sample_size_bytes++;
      // Add one byte for terminating NUL character
      p->sample_size_bytes++;
    }
  }

}

// Allocate Memory for sink data
if (!err)
{
    p->buffer_size = p->sample_limit * p->sample_size_bytes;
    p->v = malloc(p->buffer_size);
    if (!p->v)
      err |= 1 << 1;
  }

  #ifdef DIAGNOSTICS
  // Diagnostic Report
  printf("-------------------------------------------\n");
  printf("SINK\n");
  printf("  Creation:............... %s\n", ((err)? "FAILED":"SUCCEEDED"));
  printf("  Location:............... %p\n", (void *) p);
  printf("  Sample size:............ %lu bytes\n", (unsigned long) p->sample_size_bytes);
  printf("  Sample limit:........... %lu\n", (unsigned long) p->sample_limit);
  printf("  Buffer size:............ %lu bytes\n", (unsigned long) p->buffer_size);
  printf("  Buffer location:........ %p\n", (void *) p->v);
  printf("-------------------------------------------\n");
  #endif

  if (err)
    SINK_Del(p);

  *errcode = err;
  return p;
}

void SINK_Purge(CONFIG *c, SINK *p)
{
  DWORD i, seq, missing, distinct;
  BYTE *base;
  int found, complete;
  WORD id, stream_id, last_stream_id;
  char filename[256];
  int filenamelen;
  FILE *fp;

  // Transmitter
  if (c->scheduler_TX_notRX)
  {
    p->fp = NULL;
    if (c->sink_name)
    {
      char path[256];
      strcpy(path, c->path);
      strcat(path,c->sink_name);
      p->fp = fopen(path, "wb");
      //if (!p->fp)
      //   err |= 1 << 7;
    }
    else
      p->fp = stdout;

    // Leading cushion
    for (i = 0; i < c->cushion_bits*c->modem_samples_per_bit; i++)
      fwrite(&c->bitptr[0], sizeof(float), 1, p->fp);

    // Buffer dump
    fwrite(p->v, p->sample_size_bytes, p->samples, p->fp);

    // Trailing cushion
    for (i = 0; i < c->cushion_bits*c->modem_samples_per_bit; i++)
      fwrite(&c->bitptr[0], sizeof(float), 1, p->fp);
  }
  // Receiver
  else
  {
    if (c->diagnostics)
    {
      for (i = 0; i < p->samples; i++)
      {
        base = p->v + i * p->sample_size_bytes;
        PrintMessage(base);
      }
    }

    // The assumption is that there are multiple message streams contained
    // in the data. So as to operate in fixed-memory, the message streams
    // are processed one at a time, starting with the lowest ID. This is
    // not an approach that is very consistent with the notion of a streaming
    // real-time system, but it is a start.

    // Stream ID's of zero will be ignored. They are used to push messages
    // that the decoder must receive and process and are assumed to be
    // discriminated against at the decoder level.

    stream_id = 0;
    fp = NULL;
    do
    {
      // Find next larger sequence ID that has a sequence number of zero.
```

```
last_stream_id = stream_id;
for (i = 0; i < p->samples; i++)
{
  base = p->v + i * p->sample_size_bytes;
  if (0 == GetMessageSeq(base))
  {
    id = GetMessageID(base);
    if (id > last_stream_id)
      if ((id < stream_id)||(stream_id == last_stream_id))
        stream_id = id;
  }
}

// Process the next stream (if one was found)
if (stream_id > last_stream_id)
{
  if (c->diagnostics)
    printf("Stream ID: %lu.\n", (unsigned int) stream_id);
  missing = 0;
  distinct = 0;
  complete = FALSE;
  for (seq = 0; (!complete) && (seq < p->samples); seq++)
  {
    found = FALSE;
    for (i = 0; (!found) && (i < p->samples); i++)
    {
      base = p->v + i * p->sample_size_bytes;
      if ( (seq == GetMessageSeq(base))&&(stream_id == GetMessageID(base)) )
      {
        found = TRUE;
        distinct++;
      }
    }
    if (found)
    {
      // Extract file name from header message and open file
      if (0 == seq)
      {
        filenamelen = GetMessageLoadBits(base)/8;
        if (filenamelen < 255)
        {
            memmove(filename, c->path, strlen(c->path));
          memmove(filename+strlen(c->path), GetMessagePayload(base), filenamelen);
          filename[filenamelen+strlen(c->path)] = NUL;
          fp = fopen(filename, "wb");
```

```
              }
            }
            // Process non-header messages
            else
            {
              // Check for terminal message
              if (0 == GetMessageLoadBits(base))
                complete = TRUE;
              // Transfer next data fragment to file
              else
                if (fp)
                  fwrite(GetMessagePayload(base), 1, GetMessageLoadBits(base)/8, fp);
            }
          }
          else
          {
            if (c->diagnostics)
              printf("*** Missing Sequence #: %lu\n", (unsigned int) seq);
            missing++;
          }
        }

        if (c->diagnostics)
        {
          if (!complete)
            printf("Terminal message not found.\n");
          printf("Total Missing Sequences: %lu\n", (unsigned int) missing);
          printf("Total Distinct Messages: %lu\n", (unsigned int) distinct);
        }
      }
      if (fp)
      {
        fclose(fp);
        fp = NULL;
        if(!complete || missing > 0) //delete the file{
        {
         printf("Removing the file %s\n",filename);
         remove(filename);
         }
      }

    } while (stream_id > last_stream_id);
  }
  p->samples = 0;
}
```

A.3.19 source.h

```
/**************************************************************************
 * Signal Source Module for the Real-time BBC Codec/Modem                  *
 **************************************************************************
 * William L. Bahn                                                         *
 * Academy Center for Information Security                                 *
 * Department of Computer Science                                          *
 * United States Air Force Academy                                         *
 * USAFA, CO 80840                                                         *
 **************************************************************************
 * FILE:............ source.h                                              *
 * DATE CREATED:.... 08 SEP 07                                             *
 * DATE MODIFIED:... 08 SEP 07                                             *
 **************************************************************************
 *
 * REVISION HISTORY
 *
 **************************************************************************
 *
 * DESCRIPTION
 *
 * This module supports the signal source for both the TX and the RX
 *
 */

#ifndef SOURCEdotH
#define SOURCEdotH

//------------------------------------------------------------------------
// REQUIRED INCLUDES
//------------------------------------------------------------------------

#include "config.h"
#include "dirtyd.h"

//------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
//------------------------------------------------------------------------

typedef struct SOURCE SOURCE;

//------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//------------------------------------------------------------------------
```

```
// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct SOURCE
{
   int    streaming;          // Buffer active flag
   DWORD sample;              // Number of samples that have been processed
   DWORD samples;             // Number of samples in buffer
   DWORD sample_size_bytes;   // Bytes required per sample
   DWORD sample_limit;        // Number of samples space is allocated for
   BYTE *v;                   // Buffer address

   DWORD file_bytes;          // File size based on seek test
   size_t chunk_size;         // File bytes bytes per message
   size_t buffer_size;        // Size of allocated source buffer
};

//------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//------------------------------------------------------------------------

SOURCE *SOURCE_Del(SOURCE *p);
SOURCE *SOURCE_New(CONFIG *c, DWORD *errcode);

//------------------------------------------------------------------------
#endif
```

A.3.20 source.c

```
/**************************************************************************
 * Data Source Module for the Real-time BBC Codec/Modem                   *
 **************************************************************************
 * William L. Bahn                                                        *
 * Academy Center for Information Security                                *
 * Department of Computer Science                                         *
 * United States Air Force Academy                                        *
 * USAFA, CO 80840                                                        *
 **************************************************************************
 * FILE:............ source.c                                             *
 * DATE CREATED:.... 08 SEP 07                                            *
 * DATE MODIFIED:... 08 SEP 07                                            *
 ***************************************************************************
 *
 * REVISION HISTORY
 *
 ***************************************************************************
 *
 * DESCRIPTION
 *
 * This module supports the data source for both the TX and the RX.
 *
 */

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------

#include <string.h>   // memmove()
#include <stdlib.h> // malloc(), free()

#include "bbcftp.h"
#include "source.h"
#include "dirtyd.h"

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//-----------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//-----------------------------------------------------------------------------

SOURCE *SOURCE_Del(SOURCE *p)
{
  if (p)
  {
    if (p->v)      { free(p->v);   p->v = NULL;         }
    free(p);
    p = NULL;
  }
  return p;
}

/*
 * Eventually the source and sink will be Gnu Radio and therefore very
 * little effort has been made to make this temporary source flexible
 * or sophisticated. The SOURCE_New() function opens the source file,
 * allocated memory for the entire contents, loads the entire contents
 * into memory, and then closes the source file.
 *
 * TX: If configured as a transmitter, the data file is assumed to be a
 * binary file that is to be transmitted across a BBC link. The file
 * is brought up into memory as a series of messages using the following
 * format:
 *
 * [Checksum] [SeqNum] [DataBits] [Data]
 *
 * The sequence number is a 16-bit number starting at 0 and incrementing
 * by one for each packet. The length field is also a 16-bit number that
 * contains the number of bits of actual data follows. The data field
 * contains a string of bits read directly from the file being transmitted.
 * It is a fixed width field and is zero padded if necessary. The checksum
 * field is the last 32-bits of the message and contains a CRC checksum for
 * message up to, but not including, the checksum field. At the present time,
 * the checksum field is set to all zeros.
 *
 */

DWORD SOURCE_NewTX(SOURCE *p, CONFIG *c)
{
  DWORD err;
  FILE *fp;
  BYTE *base;
  DWORD bytes_read;
  WORD seqnum, loadbits, id, length;
  BYTE *buffer;

  err = 0;

  // Initialize state information
  p->streaming = TRUE;
  p->sample = 0;
  p->samples = 0;
```

```
// Data Source
fp = NULL;
if (c->source_name)
{
  char path[256];
  strcpy(path, c->path);
  strcat(path,c->source_name);
  fp = fopen(path, ”rb”);
  if (!fp)
    err |= 1 << 7;

}

// Create Data Read Buffer
if (fp)
{
  // Determine the size of the file
  fseek(fp, 0, SEEK_END);
  p->file_bytes = ftell(fp);
  fseek(fp, 0, SEEK_SET);

  // How much memory each message needs in the Source Buffer
  p->sample_size_bytes = c->bytes_per_message;

  // Determine if each message can carry at least one file byte.
  if ( !((c->codec_message_bits/8) > BBC_FTP_HEADER_BYTES) )
    err |= 1 << 4;
}

if (!err)
{
  // Calculate how many bytes of the file each message can hold.
  p->chunk_size = (c->codec_message_bits/8) - BBC_FTP_HEADER_BYTES;      // File bytes per message
  p->sample_limit = p->file_bytes / p->chunk_size; // Messages needed for whole chunks
  if ( p->file_bytes % p->chunk_size )              // Plus one for any partial chunk
    p->sample_limit++;
  p->sample_limit+=2;                               // Plus one each for header/trailer

  p->buffer_size = p->sample_limit * p->sample_size_bytes;
  if (p->buffer_size)
    p->v = malloc(p->buffer_size);
  else
    err |= 1 << 2;
  if (!p->v)
```

```
      err |= 1 << 3;

    buffer = malloc(p->chunk_size);
    if (!buffer)
      err |= 1 << 5;
  }

  // Fill Data Buffer
  if (!err)
  {
    p->samples = 0;
    id = c->source_id;
    seqnum = 0;
    bytes_read = 0;
    do
    {
      base = p->v + p->samples * p->sample_size_bytes;

      length = p->chunk_size;
      if ((p->file_bytes - bytes_read) <= length)
        length = p->file_bytes - bytes_read;

      if (seqnum)
      {
        if (length)
          length = fread(buffer, 1, length, fp);
        SetMessagePayload(base, buffer, length, 0);
        bytes_read += length;
      }
      else
      {
        length = strlen(c->source_name);
        if (length > p->chunk_size)
          length = p->chunk_size;
        SetMessagePayload(base, c->source_name, length, 0);
      }

      loadbits = 8 * length;
      SetMessageChecksum(base, 0); // Force checksum to zero (temporary convenience)
      SetMessageSeq(base, seqnum);
      SetMessageLoadBits(base, loadbits);
      SetMessageID(base, c->source_id);

      seqnum++;
      if (c->diagnostics)
```

```
        PrintMessage(base);
      p->samples++;

    } while (length);

    fclose(fp);
    fp = NULL;
  }

  return err;
}

DWORD SOURCE_NewRX(SOURCE *p, CONFIG *c)
{
  DWORD err;
  FILE *fp;

  err = 0;

  // Initialize state information
  p->streaming = TRUE;
  p->samples = 0;

  // Data Source
  fp = NULL;
  if (c->source_name)
  {
    char path[256];
    strcpy(path, c->path);
    strcat(path,c->source_name);
    fp = fopen(path, "rb");
    if (!fp)
      err |= 1 << 7;
  }

  // Create Data Read Buffer
  if (fp)
  {
    // Determine the size of the file
    fseek(fp, 0, SEEK_END);
    p->file_bytes = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    p->sample_size_bytes = c->source_sample_size_bytes;
    // Determine number of complete samples in data file
    p->sample_limit = p->file_bytes / p->sample_size_bytes;
    // Adjust sample limit if initialization file sets a lower limit
    if ((c->source_sample_limit)&&(p->sample_limit > c->source_sample_limit))
      p->sample_limit = c->source_sample_limit;

    p->buffer_size = p->sample_limit * p->sample_size_bytes;
    if (p->buffer_size)
      p->v = malloc(p->buffer_size);
    else
      err |= 1 << 2;
    if (!p->v)
      err |= 1 << 3;
  }

  // Fill Data Buffer
  if (!err)
  {
    p->sample_limit = fread(p->v, p->sample_size_bytes, p->sample_limit, fp);
    fclose(fp);
    fp = NULL;
  }

  return err;
}

SOURCE *SOURCE_New(CONFIG *c, DWORD *errcode)
{
  DWORD err;
  SOURCE *p;

  p = NULL;
  err = 0;

  p = (SOURCE *) malloc(sizeof(SOURCE));
  if (!p)
    err |= 1 << 0;

  if (!err)
    if (c->scheduler_TX_notRX)
      err = SOURCE_NewTX(p, c);
    else
      err = SOURCE_NewRX(p, c);
  if (c->diagnostics)
  {
    // Diagnostic Report
    printf("----------------------------------------\n");
    if (c->scheduler_TX_notRX)
      printf("MESSAGE SOURCE\n");
    else
      printf("USRP SOURCE\n");
    printf("  File name:............. %s\n", c->source_name);
    printf("  Creation:.............. %s\n", ((err)? "FAILED":"SUCCEEDED"));
    printf("  Location:.............. %p\n", (void *) p);
    printf("  File size:............. %lu bytes\n", (unsigned long) p->file_bytes);
    printf("  Chunk size:............ %lu bytes\n", (unsigned long) p->chunk_size);
    printf("  Messages needed:....... %lu\n", (unsigned long) p->sample_limit);
    printf("  Message requirements:... %lu bytes\n", (unsigned long) p->sample_size_bytes);
    printf("  Buffer size:........... %lu bytes\n", (unsigned long) p->buffer_size);
    printf("  Buffer location:....... %p\n", (void *) p->v);
    printf("----------------------------------------\n");
  }

  if (err)
    SOURCE_Del(p);

  *errcode = err;
  return p;
}

/*
DWORD SOURCE_Run(BBCFTP *sys)
{
  // Load another block of data from the file if possible.
  while ( (sys->source->fp) && (sys->source->input_fifo_bytes <= sys->config->file_block_size) )
  {
    bytes_read = fread(sys->source->input_fifo + sys->source->fifo_write, 1, sys->config->
        file_block_size, sys->source->fp);
    sys->source->input_fifo_bytes += bytes_read;
    sys->source->input_fifo_write = (sys->source->input_fifo_write + bytes_read) & (sys->config->
        input_fifo_mask);
    if (bytes_read < sys->config->file_block_size)
    {
      fclose(sys->source->fp);
      sys->source->fp = NULL;
    }
  }

  // Process as much data from input FIFO to output FIFO as possible
  if( (sys->source->input_fifo_bytes > sys->source->input_chunk_size) && (sys->source->
      output_fifo_items < sys->source->output_fifo_size) )
  {
    // Process a chunk of data
    if (sys->config->scheduler_TX_notRX)
    {
      // Prepare a message for encoding
    }
    else
    {
      // Transfer raw USRP data for demodulation
    }

    sys->source->input_fifo_bytes -= sys->source->input_chunk_size;
    sys->source->input_fifo_read = (sys->source->input_fifo_bytes + sys->source->input_chunk_size) & (sys->config->input_fifo_mask);

    sys->source->output_fifo_items++;
    sys->source->output_fifo_write = (sys->source->output_fifo_write + sys->source->output_chunk_size) & (sys->config->output_fifo_mask);
  }

  // Determine if source can no longer stream data to its successor
  if (!sys->source->fp)
    if (sys->source->input_fifo_bytes < sys->source->input_chunk_size)
      if (0 == sys->source->output_fifo_items)
        sys->source->streaming = FALSE;

  return 0;
}
*/

//-----------------------------------------------------------------------------
```

A.3.21 usrp.c

```
/*************************************************************************
 * Main TX/RX Module for the Real-time BBC Codec/Modem                  *
 *************************************************************************
 * William L. Bahn                                                      *
 * Academy Center for Information Security                              *
 * Department of Computer Science                                       *
 * United States Air Force Academy                                      *
 * USAFA, CO 80840                                                      *
 *************************************************************************
 * FILE:............ usrp.c                                             *
 * DATE CREATED:.... 03 SEP 07                                          *
 * DATE MODIFIED:... 08 SEP 07                                           *
 *************************************************************************
 *
 * REVISION HISTORY
 *
 *************************************************************************
 *
 * DESCRIPTION
 *
 * This program implements a simple file transfer protocol using a BBC-encoded
 * data channel. Since the purpose of this code is to implement only specific
 * real-time components, and not all of them, the data source and sinks are
 * kept very simple. In particular, the transmitter reads the entire file into
 * memory, formatted as a series of BBC messages, before transmission begins
 * and, similarly, the receiver stores all of the received messages into memory
 * before dumping them to disk all at once. This is opposed to the streaming
 * source and sink modules that will be typical of the complete real-time
 * implementation.
 *
 * The basic, high-level, flow is as follows:
 *
 * TX: The Transmitter
 *
 * The transmitter uses the following signal flow:
 *
 * SOURCE -> ENCODER -> BUFFER -> MODULATOR -> SINK
 *
 *
 * T
 * RX: The Receiver
 *
 * The receiver used the following signal flow
 *
 * SOURCE -> MODEM -> BUFFER -> CODEC -> SINK
 *
 *
  //-----------------------------------------------------------------------

 * the module supports both the transmitter and recevier functions.
 *
 *
 */
 /* Real-time BBC CODEC
 *
 * This program is designed to process the raw USRP output data and decode
```

```
 * the resulting packets in real-time in a streaming fashion. Since it is
 * a real-time application, structural overhead has been minimized and
 * global variables have been used extensively.
 *
 * THE DATA BUFFER
 *
 * The data is stored in a circular buffer with the following variables:
 *     buffer: Pointer to the block of memory where the buffer starts.
 *     read:   Index of the first byte of the present packet.
 *     write:  Index of the next unused buffer location.
 *     fill:   How many bytes are in buffer beyond the scope of the CODEC.
 *     unused: How many unused bytes are available in the buffer.
 *
 * The buffer is seen by two functions, the one that is demodulating the
 * data packet and the one that is decoding the resulting data. The
 * demodulating function writes to the buffer at a nominally constant
 * rate dictated by the communications link. In this application, this is
 * simulated by reading the stored waveform data from a file and querying
 * the clock to determine how many bytes to add to the buffer each time
 * the function is called. The decoding function, on the other hand, always
 * to decodes eight packets each time it is called provided sufficient data
 * is available. Specifically, it decodes the eight packets that start with
 * the bits in the byte stored at the "read" pointer. Since it can't decode
 * packets that are not completely contained in the buffer, the decoding
 * function first checks to see if "fill" is non-negative. If it isn't, then
 * it returns immediately. At the other end of the spectrum, the MODEM
 * may run out of unused memory to write to. If this happens, data is going
 * to be lost. It is cleaner to throw away old data instead of introducing
 * a gap in present data, therefore the MODEM will push the "read"
 * pointer forward as it overwrites the beginning of the existing packet
 * data.
 *
 */

//-----------------------------------------------------------------------------
// FILE INCLUSIONS
//-----------------------------------------------------------------------------

#include <stdio.h>  // printf()
#include <stdlib.h> // exit(), EXIT_SUCCESS, EXIT_FAILURE
#include <time.h>   // clock(), CLOCKS_PER_SEC

#include "bbcftp.h"
#include <pthread.h>
#include "config.h"
```

```
#include "source.h"
#include "codec.h"
#include "buffer.h"
#include "modem.h"
#include "sink.h"



//--------------------------------------------------------------------------
// TRANSMITTER
//--------------------------------------------------------------------------

int tx(BBCFTP *sys)
{
  int state;

  //------------------------------------------------------------------------
  // Runtime scheduler
  //------------------------------------------------------------------------

  state = 0;
  sys->config->tot_ticks = clock();
  while ( (sys->sink->streaming) && ( sys->source->streaming || sys->buffer->ready ) )
  {
    switch (state)
    {
      case 0: // Scheduler
        if ( (sys->sink->streaming) && (sys->config->actual_trx_bytes < sys->config->
            nominal_trx_bytes) )
        {
          state = 1; // Run MODEM until sampling is caught up
        }
        else if ((sys->source->streaming) &&(0 <= sys->buffer->margin))
        {
          state = 2; // Encode packets subject to maximum amount of time.
        }
        else
        {
          if (sys->config->scheduler_realtime)
            sys->config->nominal_trx_bytes = (DWORD) ((clock() - sys->config->tot_ticks) * sys->
                config->bytespertick);
          else
            sys->config->nominal_trx_bytes += 1;//(DWORD) (config->bytespertick);
        }

        break;

      case 1: // Modulator
        if (sys->buffer->ready == 1000)
          state = 100;
        if (sys->buffer->ready == 100)
          state = 10;
        if (sys->buffer->ready == 10)
          state = 1;
        if (sys->buffer->ready == 1)
          state = 1;
        if (sys->buffer->ready == 0)
          state = 0;
        Modulate(sys->config, sys->buffer, sys->modem, sys->sink);
        state = 0;
        break;

      case 2: // Encoder
        Encode(sys->config, sys->source, sys->codec, sys->buffer);
        if (!sys->source->streaming)
          state = 100;
        state = 0;
        break;
    }
  }
  sys->config->tot_ticks = clock() - sys->config->tot_ticks;

  //---------------------------------------------------------------------------
  // POST RUN CODE
  //---------------------------------------------------------------------------

  printf("\n");
  printf("Marks: %li\n", sys->config->marks);
  printf("Total time:........... %0.3f sec.\n", ( (double)sys->config->tot_ticks / (double) CLOCKS_PER_SEC ));
  printf("MODEM time:........... %0.3f sec.\n", ( (double)sys->config->dem_ticks / (double) CLOCKS_PER_SEC ));
  printf("CODEC time:........... %0.3f sec.\n", ( (double)sys->config->dec_ticks / (double) CLOCKS_PER_SEC ));
  printf("Samples created.......%i.\n", sys->sink->samples);
  SINK_Purge(sys->config, sys->sink);

  return EXIT_SUCCESS;
}

```

```
//-----------------------------------------------------------------------------
// RECEVIER
//-----------------------------------------------------------------------------



int rx(BBCFTP *sys)
{
  int state;
  double vmax;

  //---------------------------------------------------------------------------
  // Runtime scheduler
  //---------------------------------------------------------------------------

  vmax = 0;
  state = 0;
  sys->config->tot_ticks = clock();
  while ( ((sys->source->streaming) || (0 <= sys->buffer->margin) ) /*&& ( (double)(clock() - sys
      ->config->tot_ticks) / (double)CLOCKS_PER_SEC ) < 25.0*/)
  {
    //printf("%i %i %i\n", sys->source->streaming, sys->buffer->margin, state);
    switch (state)
    {
      case 0: // Scheduler
        if ( (sys->source->streaming) && (sys->config->actual_trx_bytes < sys->config->
            nominal_trx_bytes) )
        {
          state = 1; // Run MODEM until sampling is caught up
        }
        else if (0 <= sys->buffer->margin)
        {
          state = 2; // Decode packets subject to maximum amount of time.
        }
        else
        {
          if (sys->config->scheduler_realtime)
            sys->config->nominal_trx_bytes = (DWORD) ((clock() - sys->config->tot_ticks) * sys->
                config->bytespertick);
          else
            sys->config->nominal_trx_bytes += (DWORD) (sys->config->bytespertick);
        }

        break;
      case 1: // MODEM
        Demodulate(sys->config, sys->source, sys->modem, sys->buffer);
        state = 0;
        break;

      case 2: // CODEC
        Decode(sys->config, sys->buffer, sys->codec, sys->sink);
        state = 0;
        break;

    }
    //printf("What's going on?\n");
  }
  sys->config->tot_ticks = clock() - sys->config->tot_ticks;

  //----------------------------------------------------------------------
  // POST RUN CODE
  //----------------------------------------------------------------------

  printf("\n");
  printf("Marks: %li\n", sys->config->marks);
  printf("Messages found: %lu\n", sys->config->message_count);
  printf("Packets lost: %lu\n", (DWORD) (sys->buffer->overflows * 8));
  printf("Total time:.......... %0.3f sec.\n", ( (double)sys->config->tot_ticks / (double)
      CLOCKS_PER_SEC ));
  printf("MODEM time:.......... %0.3f sec.\n", ( (double)sys->config->dem_ticks / (double)
      CLOCKS_PER_SEC ));
  printf("CODEC time:.......... %0.3f sec.\n", ( (double)sys->config->dec_ticks / (double)
      CLOCKS_PER_SEC ));

  SINK_Purge(sys->config, sys->sink);

  return EXIT_SUCCESS;
}

//------------------------------------------------------------------------
// MAIN PROGRAM
//------------------------------------------------------------------------

int main(int argc, char *argv[])
{
  BBCFTP *sys;

  char *config_file_name;
```

```
DWORD errcode;

int res;

//---------------------------------------------------------------------
// Read configuration information
//---------------------------------------------------------------------

config_file_name = NULL;
if (argc < 2)
{
  printf(”Mode (T or R): ”);
  res = getc(stdin);
  switch (res)
  {
    case 'T':
    case 't':
      config_file_name = ”tx.ini”;
      break;
    case 'R':
    case 'r':
    default :
      config_file_name = ”rx.ini”;
  }
  while ('\n' != res)
    res = getc(stdin);
}
else
  config_file_name = argv[1];

sys = BBCFTP_New(config_file_name, &errcode);
if (errcode)
{
  printf(”BBC FTP System Constructor exited with error code: %lu\n”, errcode);
  exit(EXIT_FAILURE);
}

//---------------------------------------------------------------------
// Launch transmitter or recever as appropriate
//---------------------------------------------------------------------

if (sys->config->scheduler_TX_notRX)
  tx(sys);
else
  rx(sys);

  //--------------------------------------------------------------------------
  // Runtime Scheduler
  //--------------------------------------------------------------------------

  // The components of the new scheduler are not yet complete.
  // while (sys->sink->streaming)
  // {
  //     SOURCE_Run(sys);
  //     CODEC_Run(sys);
  //     MODEM_Run(sys);
  //     SINK_Run(sys);
  // }

  //--------------------------------------------------------------------------
  // Final Housekeeping
  //--------------------------------------------------------------------------

  //BBCFTP_Del(sys);

  return EXIT_SUCCESS;
}
```

A.3.22 Makefile

```
#
# Real-time BBC Demodulator and Decoder
#
usrp: usrp.o bbcftp.o config.o source.o codec.o buffer.o modem.o sink.o sha1.o dirtyd.o bytes.o
  gcc -o usrp usrp.o bbcftp.o config.o source.o codec.o buffer.o modem.o sink.o sha1.o dirtyd.o bytes.o -lm

# Top Level Program

usrp.o: usrp.c
  gcc -c -O3 usrp.c

usrp.c: bbcftp.h config.h source.h codec.h buffer.h modem.h sink.h

# Application Module

bbcftp.o: bbcftp.c
  gcc -c -O3 bbcftp.c

bbcftp.c: bbcftp.h
bbcftp.h: config.h source.h codec.h buffer.h modem.h sink.h dirtyd.h

# Configuration Module

config.o: config.c
	gcc -c -O3 config.c

config.c: config.h dirtyd.h

config.h: dirtyd.h

# SOURCE Module

source.o: source.c
	gcc -c -O3 source.c

source.c: bbcftp.h source.h dirtyd.h

source.h: config.h dirtyd.h

# CODEC Module

codec.o: codec.c

codec.c: codec.h sha1.h
	gcc -c -O3 codec.c

codec.h: config.h source.h buffer.h sink.h sha1.h dirtyd.h

# BUFFER Module

buffer.o: buffer.c
	gcc -c -O3 buffer.c

buffer.c: buffer.h

buffer.h: config.h dirtyd.h

# MODEM Module

modem.o: modem.c

modem.c: modem.h sha1.h
	gcc -c -O3 codec.c
```

```

modem.h: config.h source.h buffer.h sink.h dirtyd.h

# SINK Module

sink.o: sink.c
  gcc -c -O3 sink.c

sink.c: sink.h dirtyd.h

sink.h: config.h dirtyd.h

# SHA1 Support Module

sha1.o: sha1.c
  gcc -c -O3 sha1.c

sha1.c: sha1.h

# DIRTY DEEDS Support Module

dirtyd.o: dirtyd.c
  gcc -c -O3 dirtyd.c

dirtyd.c: dirtyd.h

dirtyd.h: bytes.h

# BYTE Definitions Support Module

bytes.o: bytes.c
  gcc -c -O3 bytes.c

bytes.c: bytes.h

# HOUSEKEEPING TARGETS

clean:
  rm *.o
```

A.4 Jammer Source Code

A.4.1 Main Program Source (jammer.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "config.h"
#include "buffer.h"
#include "modem.h"
#include "sink.h"
#include <unistd.h>

// return a random integer in the range [0, n).
// n should be in the range [1, RAND_MAX].
unsigned long long randint(unsigned long long n)
{
    if (n <= 0)        return -1;
    if (n > RAND_MAX) return -1;
    unsigned long long r;
    // the trivial rand()%n implementation does not generate uniform
    // distributions, so we ignore the top section of the distribution that
    // would become nonuniform.
    do {
        r = rand();
    } while (r >= (RAND_MAX/n)*n);

    return r % n;
}

int main(int argc, char *argv[])
{
  extern char *optarg;
  extern int optind, opterr, optopt;
  char *config_file_name;
  DWORD errcode;
  CONFIG *config;
  BUFFER *buffer;
  MODEM *modem;
  SINK *sink;
  int jammer_level = 12;
  int samples = 1600;
  unsigned char marked[8*sizeof(unsigned long long)];
```

```
int i = 0;
int j = 0;
int c;
unsigned long long *ran_number = malloc(sizeof(unsigned long long));
unsigned long long *buf_number = malloc(sizeof(unsigned long long));
srand((unsigned)(time(0)));

config_file_name = ”tx.ini”;

while ((c = getopt(argc, argv, ”J:N:C:”)) != -1) {
  switch(c) {
          case ’J’:
            jammer_level = atoi(optarg);
            break;
          case ’N’:
            samples = atoi(optarg);
            break;
          case ’C’:
            config_file_name = optarg;
            break;
  }



}

config   = CONFIG_New(config_file_name, &errcode);
buffer   = BUFFER_New(config, &errcode);
modem    = MODEM_New(config, &errcode);
sink  = SINK_New(config, &errcode);

for(i = 0;i<(samples/(32*sizeof(unsigned long long)));i++){
  *buf_number = 0;
  *ran_number = 0;
  for(j=0;j < jammer_level;j++){
    *ran_number = rand()%(8*sizeof(unsigned long long));
    while (marked[*ran_number]==1){
      *ran_number =  rand()%(8*sizeof(unsigned long long));
    }
    marked[*ran_number] = 1;
    //set the bit at ran_number to 1
    *buf_number |= (1 << *ran_number);
  }
  memcpy(buffer->buffer+buffer->write, buf_number, sizeof(unsigned long long));
  buffer->write+=sizeof(unsigned long long);

```

```
    for(j=0;j<sizeof(unsigned long long);j++){
      buffer->ready = 1;
      Modulate(config, buffer, modem, sink);
    }
    memset(marked,0x00, sizeof(unsigned long long)*8);
  }

  printf("Samples...........: %i\n",sink->samples);
  SINK_Purge(config,sink);
  MODEM_Del(modem);
  BUFFER_Del(buffer);
  CONFIG_Del(config);
  SINK_Del(sink);

  free(ran_number);
  free(buf_number);
  return EXIT_SUCCESS;
}
```

A.4.2 Modified BBC modem.h Source

```
/*************************************************************************
 * MODEM for the Real-time BBC Codec/Modem                               *
 *************************************************************************
 * William L. Bahn                                                       *
 * Academy Center for Information Security                               *
 * Department of Computer Science                                        *
 * United States Air Force Academy                                       *
 * USAFA, CO 80840                                                       *
 *************************************************************************
 * FILE:........... modem.h                                              *
 * DATE CREATED:.... 06 SEP 07                                           *
 * DATE MODIFIED:... 06 SEP 07                                           *
 *************************************************************************
 *
 * REVISION HISTORY
 *     Modified to support only the requirements of providing same symbol
 *     rate data as a means to create a jammer.
 *     2/28/2009 Derek Sanders
 *
 *************************************************************************
 *
 * DESCRIPTION
 *
 * The modem converts baseband signal data to/from packet data.
 *
 */

#ifndef MODEMdotH
#define MODEMdotH

//----------------------------------------------------------------------------
// REQUIRED INCLUDES
//----------------------------------------------------------------------------

#include <time.h>  // clock_t

#include "config.h"
#include "buffer.h"
#include "sink.h"
#include "dirtyd.h"

//----------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
//----------------------------------------------------------------------------

typedef struct MODEM MODEM;

//----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct MODEM
{
   // Derived quantities
   DWORD jitter_samples;
   double alpha;
   double t_hi, t_lo;

   // State information
   DWORD state;
   double integrator;
   SDWORD stamp;
};
```

```
//----------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//----------------------------------------------------------------------------

MODEM *MODEM_Del(MODEM *p);
MODEM *MODEM_New(CONFIG *c, DWORD *errcode);
void Modulate(CONFIG *c, BUFFER *buffer, MODEM *modem, SINK *sink);

//----------------------------------------------------------------------------
#endif
```

A.4.3 Modified BBC modem.c Source

```
/***************************************************************************
 * MODEM for the Real-time BBC Codec/Modem                                 *
 ***************************************************************************
 * William L. Bahn                                                         *
 * Academy Center for Information Security                                 *
 * Department of Computer Science                                          *
 * United States Air Force Academy                                         *
 * USAFA, CO 80840                                                         *
 ***************************************************************************
 * FILE:............ modem.c                                               *
 * DATE CREATED:.... 06 SEP 07                                             *
 * DATE MODIFIED:... 06 SEP 07                                             *
 ***************************************************************************
 *
 * REVISION HISTORY
 *
 *     Modified to support only the requirements of providing same symbol
 *     rate data as a means to create a jammer.
 *     2/28/2009 Derek Sanders
 ***************************************************************************
 *
 * DESCRIPTION
 *
 * The modem and its public interface is described in modem.h.
 *
 ***************************************************************************
 */

//----------------------------------------------------------------------------
// REQUIRED INCLUDES
//----------------------------------------------------------------------------
```

```
#include <stdlib.h> // malloc()
#include <math.h>   // exp()
#include "modem.h"

//----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//----------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//----------------------------------------------------------------------------

MODEM *MODEM_Del(MODEM *p)
{
  if (p)
  {
    free(p);
  }
  return NULL;
}

MODEM *MODEM_New(CONFIG *c, DWORD *errcode)
{
  MODEM *p;
  DWORD err;
  double nominal_steady_state_peak;

  p = NULL;
  err = 0;

  p = (MODEM *) malloc(sizeof(MODEM));
  if (!p)
    err |= 1 << 0;

  if (!err)
  {
    // Derived quantities
    p->jitter_samples = (int)(c->modem_samples_per_bit * c->modem_jitter_bits);
    // Integrator parameter
    p->alpha = exp((2.0/c->modem_samples_per_bit) - 1.0);

    // Threshold parameters
    nominal_steady_state_peak = (c->nominal_rx_signal*c->nominal_rx_signal) * (1.0/(1.0-p->alpha));
    p->t_hi = nominal_steady_state_peak * ((c->modem_threshold_pct + c->modem_hysteresis_pct/2.0)/100.0);
    p->t_lo = nominal_steady_state_peak * ((c->modem_threshold_pct - c->modem_hysteresis_pct/2.0)/100.0);

    // State information
    p->state = 0;
    p->integrator = 0.0;
    p->stamp = 0;
  }

  if (err)
    p = MODEM_Del(p);

  if (c->diagnostics)
  {
    // Diagnostic Report
    printf("------------------------------------------\n");
    printf("MODEM\n");
    printf("  Creation:.................. %s\n", ((err)? "FAILED":"SUCCEEDED"));
    printf("  Location:.................. %p\n", (void *) p);
    printf("  Integrator alpha:.......... %f\n", p->alpha);
    printf("  Jitter tolerance:.......... %f\n", p->jitter_samples);
    printf("  Modem gain:................ %f (%f dB)\n", c->nominal_tx_signal, c->modem_gain_dB);
    printf("  Nominal channel loss:...... %f dB\n", c->modem_channel_loss_dB);
    printf("  Nominal rx signal peak:.... %f (%f dB)\n", c->nominal_rx_signal, (c->modem_gain_dB-c->modem_channel_loss_dB));
    printf("  Nominal integrator peak:... %f\n", nominal_steady_state_peak);
    printf("  LO -> HI threshold:........ %f\n", p->t_hi);
    printf("  HI -> LO threshold:........ %f\n", p->t_lo);
    printf("------------------------------------------\n");
  }

  *errcode = err;
  return p;
}

//--------------------------------------------------------------------------
```

```

/* MODEM
 *
 * The MODEM reads/writes USRP in bursts of samples corresponding to
 * 8 packet bits. The calling function is responsible for ensuring that
 * valid data and/or sufficient room for new data exists in the buffer.
 *
 */

/* MODULATOR
 *
 * The modulator reads one byte of packet data from the buffer and generates
 * USRP data for the entire set of 8 packet bits.
 *
 */

void Modulate(CONFIG *c, BUFFER *buffer, MODEM *modem, SINK *sink)
{
  DWORD originbit, sample;
  float signal;
  clock_t ticks;
  float *v;
  ticks = clock();

  // Push write pointer if packet byte is not available
  if (!buffer->ready)
  {
    buffer->write = (buffer->write + 1) & buffer->buffermask;
    buffer->ready++;
    buffer->margin--;
  }

  // For each bit in the packet byte at the buffer's read pointer
  for (originbit = 0; originbit < 8; originbit++)
  {
    // Determine if the bit is a mark or a space
    if (buffer->buffer[buffer->read] & c->bitmask[originbit])
    {
      c->marks++;
      signal = (float) c->nominal_tx_signal;
    }
    else
      signal = 0.0;

    // Determine if the sink can take all the samples for the present bit
    if (sink->samples + c->modem_samples_per_bit < sink->sample_limit)
    {
      // Establish the base location within the sink's buffer
      v = ((float *) sink->v) + (2 * sink->samples);

      // Generate and write the baseband samples to the sink
      for (sample = 0; sample < c->modem_samples_per_bit; sample++)
      {
        v[2*sample]     = signal; // I(t) (actual data)
        v[2*sample + 1] = 0.0;    // Q(t) (forced to zero)
      }
      sink->samples += c->modem_samples_per_bit;
    }
    else
      sink->streaming = FALSE;
  }

  buffer->buffer[buffer->read] = 0;
  buffer->read = (buffer->read + 1) & buffer->buffermask;
  buffer->ready--;
  buffer->margin++;

  c->actual_trx_bytes += c->trx_bytes_per_packet_byte;
  c->dem_ticks += clock() - ticks;
}
```

A.4.4 Modified BBC sink.h Source

```
/*************************************************************************
 * Signal Sink Module for the Real-time BBC Codec/Modem                  *
 *************************************************************************
 * William L. Bahn                                                       *
 * Academy Center for Information Security                               *
 * Department of Computer Science                                        *
 * United States Air Force Academy                                       *
 * USAFA, CO 80840                                                       *
 *************************************************************************
 * FILE:............ sink.h                                              *
 * DATE CREATED:.... 08 SEP 07                                           *
 * DATE MODIFIED:... 08 SEP 07                                           *
 *************************************************************************
 *
 * REVISION HISTORY
 *
 *************************************************************************
 *
 * DESCRIPTION
 *
 * This module supports the signal sink for both the TX and the RX
 *
 */

#ifndef SINKdotH
#define SINKdotH

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------

#include "config.h"
#include "dirtyd.h"

//-----------------------------------------------------------------------------
// STRUCTURE DECLARATIONS
//-----------------------------------------------------------------------------

typedef struct SINK SINK;

//-----------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//-----------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

struct SINK
{
  FILE *fp;
  int    streaming;
  DWORD samples;
  DWORD sample_size_bytes;
  DWORD sample_limit;
  BYTE *v;

  size_t  buffer_size;
};
```

```
//-----------------------------------------------------------------------------
// PUBLIC FUNCTION PROTOTYPES
//-----------------------------------------------------------------------------

SINK *SINK_Del(SINK *p);
SINK *SINK_New(CONFIG *config, DWORD *errcode);
void SINK_Purge(CONFIG *config, SINK *p);

//-----------------------------------------------------------------------------
#endif
```

A.4.5 Modified BBC sink.c Source

```
/**************************************************************************
 * Signal Sink Module for the Real-time BBC Codec/Modem                   *
 **************************************************************************
 * William L. Bahn                                                        *
 * Academy Center for Information Security                                *
 * Department of Computer Science                                         *
 * United States Air Force Academy                                        *
 * USAFA, CO 80840                                                        *
 **************************************************************************
 * FILE:............ sink.c                                               *
 * DATE CREATED:.... 08 SEP 07                                            *
 * DATE MODIFIED:... 28 FEB 09                                            *
 **************************************************************************
 *
 * REVISION HISTORY
 *     Modified to support only the requirements of providing same symbol
 *     rate data as a means to create a jammer.
 *     2/28/2009 Derek Sanders
 *
 **************************************************************************
 *
 * DESCRIPTION
 *
 * This module supports the signal sink for both the TX and the RX
 *
 */

//-----------------------------------------------------------------------------
// REQUIRED INCLUDES
//-----------------------------------------------------------------------------

#include <stdlib.h> // malloc(), free()
```

```
#include <string.h> // memmove()

#include "sink.h"
#include "dirtyd.h"

//---------------------------------------------------------------------------
// STRUCTURE DEFINITIONS
//---------------------------------------------------------------------------

// NOTE: Normally the structure definition would be in the *.c file to make
// the structure members inaccessible to outside functions except through
// public function calls. But for the real-time code it has been decided
// to make the structure members directly visible to the functions that
// manipulate them.

//---------------------------------------------------------------------------
// PUBLIC FUNCTION DEFINITIONS
//---------------------------------------------------------------------------

SINK *SINK_Del(SINK *p)
{
  if (p)
  {
    if (p->fp)
      if (stdout != p->fp)
      {
        fclose(p->fp);
        p->fp = NULL;
      }
    if (p->v)     { free(p->v);   p->v = NULL;       }
    free(p);
    p = NULL;
  }

  return p;
}

// Sufficient memory is allocated up front
// to handle a maximum amount of data. However, the present
// contents of the buffer can be purged using SINK_Purge().

SINK *SINK_New(CONFIG *c, DWORD *errcode)
{
  SINK *p;
  DWORD err;

  p = NULL;
  err = 0;

  p = (SINK *) malloc(sizeof(SINK));
  if (!p)
    err |= 1 << 0;

  // Open Data Sink file
  if (!err)
  {
    /*p->fp = NULL;
    if (c->sink_name)
    {
      char path[256];
      strcpy(path, c->path);
      strcat(path,c->sink_name);
      p->fp = fopen(path, "wb");
      if (!p->fp)
        err |= 1 << 7;
    }
    else
      p->fp = stdout;*/
  }

  // Initialize state information
  if (!err)
  {
    p->samples = 0;
    p->streaming = TRUE;

    if (c->sink_sample_limit)
      p->sample_limit = c->sink_sample_limit;
    else
    {
      if (c->scheduler_TX_notRX)
      {
        p->sample_limit = 4*c->modem_samples_per_bit*c->packet_bits;
      }
      else
      {
        p->sample_limit = 1000;
      }
    }

    if (c->sink_sample_size_bytes)
      p->sample_size_bytes = c->sink_sample_size_bytes;
    else
    {
      if (c->scheduler_TX_notRX)
      {
        p->sample_size_bytes = 2*sizeof(float);
      }
      else
      {
        // One byte for each eight full bits of message
        p->sample_size_bytes = c->codec_message_bits / 8;
        // Add a final byte, if necessary, to hold leftover bits
        if (c->codec_message_bits % 8)
          p->sample_size_bytes++;
        // Add one byte for terminating NUL character
        p->sample_size_bytes++;
      }
    }

  }

  // Allocate Memory for sink data
  if (!err)
  {
    p->buffer_size = p->sample_limit * p->sample_size_bytes;
    p->v = malloc(p->buffer_size);
    if (!p->v)
      err |= 1 << 1;
  }

  if(c->diagnostics)
  {
  // Diagnostic Report
  printf("--------------------------------------------\n");
  printf("SINK\n");
  printf("  Creation:............... %s\n", ((err)? "FAILED":"SUCCEEDED"));
  printf("  Location:.............. %p\n", (void *) p);
  printf("  Sample size:............ %lu bytes\n", (unsigned long) p->sample_size_bytes);
  printf("  Sample limit:.......... %lu\n", (unsigned long) p->sample_limit);
  printf("  Buffer size:............ %lu bytes\n", (unsigned long) p->buffer_size);
  printf("  Buffer location:........ %p\n", (void *) p->v);
  printf("--------------------------------------------\n");
  }
```

```
  if (err)
    SINK_Del(p);

  *errcode = err;
  return p;
}

void SINK_Purge(CONFIG *c, SINK *p)
{
  DWORD i, seq, missing, distinct;
  BYTE *base;
  int found, complete;
  WORD id, stream_id, last_stream_id;
  char filename[256];
  int filenamelen;
  FILE *fp;

  // Transmitter
  if (c->scheduler_TX_notRX)
  {
    p->fp = NULL;
    if (c->sink_name)
    {
      char path[256];
      strcpy(path, c->path);
      strcat(path,c->sink_name);
      p->fp = fopen(path, ”wb”);
      //if (!p->fp)
      //  err |= 1 << 7;
    }
    else
      p->fp = stdout;

    // Leading cushion
    for (i = 0; i < c->cushion_bits*c->modem_samples_per_bit; i++)
      fwrite(&c->bitptr[0], sizeof(float), 1, p->fp);

    // Buffer dump
    fwrite(p->v, p->sample_size_bytes, p->samples, p->fp);

    // Trailing cushion
    for (i = 0; i < c->cushion_bits*c->modem_samples_per_bit; i++)
      fwrite(&c->bitptr[0], sizeof(float), 1, p->fp);
  }

```

```
	p->samples = 0;
}
```

A.4.6 Jammer Makefile

```
#
# Real-time BBC Demodulator and Decoder
# Modified makefile for creating a jammer 2/28/2009 Derek T. Sanders
#

INCLUDES = ../usrp0A
jammer: jammer.o config.o buffer.o modem.o dirtyd.o bytes.o sink.o
	gcc -o jammer jammer.o config.o buffer.o modem.o dirtyd.o bytes.o sink.o -lm

# Top Level Program

jammer.o: jammer.c
	gcc -c -O3 jammer.c -I$(INCLUDES) -I.

jammer.c: ../usrp0A/config.h ../usrp0A/buffer.h modem.h sink.h

# Configuration Module

config.o: ../usrp0A/config.c
	gcc -c -O3 ../usrp0A/config.c -I$(INCLUDES) -I.

$(INCLUDES)/config.c: ../usrp0A/config.h ../usrp0A/dirtyd.h

$(INCLUDES)/config.h: ../usrp0A/dirtyd.h

# BUFFER Module

buffer.o: ../usrp0A/buffer.c
	gcc -c -O3 ../usrp0A/buffer.c -I$(INCLUDES) -I.

$(INCLUDES)/buffer.c: ../usrp0A/buffer.h

$(INCLUDES)/buffer.h: ../usrp0A/config.h ../usrp0A/dirtyd.h

# MODEM Module

modem.o: modem.c

modem.c: modem.h
	gcc -c -O3 modem.c -I$(INCLUDES) -I.
```

```
modem.h: ../usrp0A/config.h ../usrp0A/buffer.h sink.h ../usrp0A/dirtyd.h

# SINK Module

sink.o: sink.c
	gcc -c -O3 sink.c -I$(INCLUDES) -I.

sink.c: sink.h ../usrp0A/dirtyd.h

sink.h: ../usrp0A/config.h ../usrp0A/dirtyd.h

# DIRTY DEEDS Support Module

dirtyd.o: ../usrp0A/dirtyd.c
	gcc -c -O3 ../usrp0A/dirtyd.c -I$(INCLUDES) -I.

$(INCLUDES)/dirtyd.c: ../usrp0A/dirtyd.h

$(INCLUDES)/dirtyd.h: ../usrp0A/bytes.h

# BYTE Definitions Support Module

bytes.o: ../usrp0A/bytes.c
	gcc -c -O3 ../usrp0A/bytes.c -I$(INCLUDES) -I.

$(INCLUDES)/bytes.c: ../usrp0A/bytes.h

# HOUSEKEEPING TARGETS

clean:
	rm *.o
```

Appendix B

Miscellaneous Files

B.1 Data Frame Hexadecimal String

```
0014a54726b200146c1e70be0800450805dcced04000360622d7cc98bf25c0a80
1061f1b080df1d58447c92573ff501016d0b13900006f636b696e67206f6e746f
207468652072657175657374206c6f636b20696e207468652055544206472697
665722c20692e652e2020646f6e2774206c6f636b0a0974686520717565756520
7370696e6c6f636b207768656e2063616c6c65642066726f6d207468652072657
1756573742066756e6374696f6e2e0a090a09496e2064657461696c3a0a090a09
52656e616d65207562645f66696e697368282920746f205f5f7562645f66696e6
97368282920616e642072656d6f7665207562645f696f5f6c6f636b2066726f6d
2069742e20204164640a09777261707065722c207562645f66696e69736828292
c207768696368206772616273206c6f636b206265666f72652063616c6c696e67
205f5f7562645f66696e69736828292e20205570646174650a09646f5f7562645
f7265717565737420746f20757365207468652066c6f636b2066726565205f5f75
62645f66696e697368282920746f2061766f696420646561646c6f636b2e20204
16c736f2c0a096170706172656e746c7920707265706172655f72657175657374
2069732063616c6c65642077697468207562645f696f5f6c6f636b2068656c642
c20736f2072656d6f7665206c6f636b730a0974686572652e0a090a095369676e
65642d6f66662d62793a20436872697320577269676874203c636872697377406
```

```
f73646c2e6f72673e0a095369676e65642d6f66662d62793a2050616f6c6f2027
426c6169736f72626c616465272047696172727573736f203c626c6169736f726
26c6164655f7370616d407961686f6f2e69743e0a095369676e65642d6f66662d
62793a20416e64726577204d6f72746f6e203c616b706d406f73646c2e6f72673
e0a095369676e65642d6f66662d62793a204c696e757320546f7276616c647320
3c746f7276616c6473406f73646c2e6f72673e0a0a3c626c6169736f72626c616
4655f7370616d407961686f6f2e69743e0a095b50415443485d20756d6c3a2075
736520616c776179732061207365706172617465206f20746872656164206666
f72205542440a090a0943757272656e746c792c207562643d73796e6320697320
646966666572656e742066726f6d207265706c6163696e672075626423d20776
974682075626423733d2e20205468697320697730a09616761696e737420507269
6e6369706c65206f66204c656173742053757270726973652c20736f2072656d6
f7665207468697320646966666572656e63652e0a090a09416c736f2074686520
63757272656e74207562643d73796e63206265686176696f757220697320636f6
d706c6574656c79207573656c6573733a206974206973207420746f206d616b652073
7572650a0974686174207768656e20746865206b65726e656c2068617320737973796
e636865642069747320492f4f20746f20746865207669727475616c206469736b
2c2074686520686f737420646f65730a096e6f7420696e76616c6964617465207
46869732077697468206869732063616368696e673b2074686973206361757365
7320526569736572465320636f7272757074696f6e2e0a090a094275742073696
e636520616374756616c6c792077652063616c6c20656e645f7265717565737428
29206f6e6c7920616674657220746865206f5f746872656164206861732064646
```

f6e65206974730a09776f726b2c207765206e65766572206c696520746f207468

6520626c6f636b206c617965722e20205573696e67204f5f53594e43206173207

76520646f207768656e207265706c6163696e670a09756264233d207769746820

75626423733d20697320656e6f7567682e0a090a095369676e65642d6f66662d6

2793a2050616f6c6f2027426c6169736f72626c61646527204769617272757373

6f203c626c6169736f72626c6164655f7370616d407961686f6f2e69743e0a095

369676e65642d6f66662d62793a20416e64726577204d6f72746f6e203c616b70

6d406f73646c2e6f72673e0a095369676e6564

B.2 RTS Frame Hexadecimal String

08ae26b201f0a0000000b0c49741d26f026e5d866232382e34373535