Transmit Simulation and Receive Optimization for 802.11b Networks

by

Pascal F. Rettig

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requiements ofor the Degrees of Bachelor of
Science and Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
May 24, 2002

Author____
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by____
Chris Riddle
VI-A Company Thesis Supervisor

Certified by____
Muriel Medard
M.I.T. Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Transmit Simulation and Receive Optimization for 802.11b Networks

by

Pascal F. Rettig

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Electrical Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The simulation presented in this paper provides an implementation of a full simulated transmit chain from packet encoding through base band modulation for the 802.11b wireless networking standard. This forward transmit chain is coupled with a physical channel simulation that can introduce a number of different channel effects to simulate interference caused in the real world. Packets which the transmit simulation produces can be sent to a receive simulation to test design parameters or can be modulated and sent to 802.11b hardware to test hardware implementation. Using former procedure, this paper also evaluates implementations of a Phase lock loop used to track Frequency Doppler and a Time Tracking Loop used to track Code Doppler under various Signal to Noise levels. The results from these simulations can be used to optimize various receive parameters and algorithms.

Thesis Supervisor: Muriel Medard
Title: Assistant Professor Department of Electrical Engineering and Computer Science

Contents

Figures

Tables

1 Introduction

This paper describes the creation of an 802.11b arbitrary waveform generator (AWG) and channel simulator tool and the use of this tool to characterize the performance of the phase locked and time tracking loops on a simulated 802.11b receiver. The AWG can also be used to generate packets for commercial 802.11b networks to verify understanding of the standard as well as test actual hardware.

The AWG and channel simulator were coded in C++ that could be linked into MATLAB. Automated tests were written as MATLAB scripts which paired the output of the transmit chain with that of a separately coded receive chain. The basic receive chain was coded as a MATLAB script by another member of the project team. Modifications were made to the phase-tracking portion of the code and certain parts were recoded in C for added efficiency when simulating large numbers of packets.

1.1 Tools

The two primary tools used in the development of this project were Microsoft Visual C++ 6.0 and MATLAB version 6.0 release 12. MATLAB’s MEX toolkit was used to create Dynamic Link Libraries of C++ code that can be called as functions from inside the MATLAB’s command window. This allowed for a seamless integration with the MATLAB coded receive chain while still taking advantage of the speed of C.

This paper was written in Microsoft Word 2000. The figures were generated from MATLAB output figures or created with Microsoft Visio.

1.2 Abbreviations

The following is an alphabetical list of abbreviations used throughout this document

AFC	Automatic Frequency Control
AWG	Arbitrary waveform generator
AWGN	Additive White Gaussian Noise
BSS	Basic Service Set
CCK	Complementary Code Keying
Chipx8	The chipping rate times a factor of eight
CRC	Cyclic Redundancy Check
CSMA/CA	Collision Sense Multiple Access with Collision Avoidance
CSMA/CD	Collision Sense Multiple Access with Collision Detection
CTS	Clear to Send
dB	Decibel
DSS	Distribution System Services
DSSS	Direct Sequence Spread Spectrum
ESS	Extended Service Set
I	In-Phase portion of complex signal
ICV	Integrity Check Vector
ISP	Internet Service Provider
LAN	Local Area Network
MAC	Medium Access Control
Mcps	Mega-Chips per second
MEX	Matlab Extensions
MPDU	MAC Protocol Data Unit
MSDU	MAC Service Data Unit
Msps	Mega-Symbols per second
NAV	Network Allocation Vector
PHY	Physical layer
PLCP	Physical Layer Convergence Procedure

PLL	Phase Locked Loop
PMD	Physical Medium Dependant
PPDU	PHY Protocol Data Unit
PSDU	PHY Service Data Unit
Q	Quadrature phase portion of complex signal
RC4	Encryption Algorithm used in WEP
RF	Radio Frequency
RTS	Ready to Send
RV	Random variable
SFD	Start Frame Delimiter
Sinc	Figure created by the equation (sin x)/x
SS	Station Services
STL	C++ Standard Template Library
SYNC	Synchronization portion of the PLCP preamble
VCO	Voltage controlled oscillator
WEP	Wireless Equivalent Privacy
WLAN	Wireless Local Area Network

2 802.11b networks

2.1 Overview

Wireless local area networking is an emerging technology that appears poised to become a significant force in business and consumer connectivity products. It combines the productivity gains leveraged from network data with the convenience and versatility of wireless. Beyond the simple removal of wires, the merging of these two technologies has opened the door for greater access to information in many different areas. WLANs have been used to create high-speed wireless ISPs where formerly there was no low-cost, high bandwidth option [6]. The relatively low-cost and availability of components has also spawned the creation of mid-range Neighborhood area networks that can link entire neighborhoods together. Businesses are using WLANs to expand on their wired backbones with a large savings in infrastructure costs. Wireless phone providers are looking to add 802.11 to their phones to give their users more, higher-speed connectivity options when available.

The standardization of a WLAN protocol has been one of the factors in their recent growth. 802.11, the first WLAN standard, was created by IEEE in 1997 and later revised in 1999. It is part of the IEEE 802 family of local and metropolitan area network standards. It provides for data rates up to 2 Mbps when using Direct Sequence Spread Spectrum (one of the physical layer options). An extension to 802.11, called 802.11b followed later that year [10]. The extension allowed for two new data rates using DSSS, 5.5 Mbps and 11 Mbps, while at the same time staying compatible with the original standard. This project implements the 802.11b specification.

A user can create a wireless network using 802.11 in one of two ways. The first is to use an Access Point station (AP) that servers as a central router for all data packets. End user stations called terminals talk to each other and the outside world through the AP. Wireless LANs that are configured in this way are referred to as being built around a Basic Service Set (BSS). The other configuration option is to create a network that consists only of stations, without an AP. Each station can talk directly to all other stations in its range, and no centralized control is need; all management functionality is distributed. A wireless LAN set up this way is referred to as an Independent BSS

(IBSS).

The operation of a BSS is extended by connecting AP's together to form an Extended Service Set (ESS). APs are connected over what IEEE calls a distribution system (DS). How the DS is implemented is not defined in the standard, but is left to the individual vendors. The DS can take the form of a wired connection or a separate WLAN on a different channel. APs from different vendors generally cannot connect together to form an ESS. When connected in to a wired LAN, the ESS appears to be a single MAC-layer network with stationary terminals. The mobility provided by 802.11 networks is masked to any networks outside of the ESS, allowing communication over standard networks protocols without any modification.

The functionality of 802.11 is exposed through a set of nine services divided into two groups: four Station Services (SS) and five Distribution System Services (DSS). The station services, Authentication, Deauthentication, Privacy, and MSDU Delivery, are supported by all stations (terminals and the AP), and provide functions similar to wired networks. Authentication verifies that a potential terminal is a legitimate user of the system, while Deauthentication ensures that once a user has logged off, access is cut off. The privacy service, in its current iteration, uses Wired Equivalent Privacy (WEP), an implementation of the RC4 algorithm, to encrypt the data as it is sent over the air. The goal of WEP was to provide a level of security equal to that of a standard wired LAN, and not to provide a comprehensive security solution for WLANs. Recently, however, it has become clear that WEPs ability to fulfill even this minimal goal is suspect; numerous paper describing the ease of which WEP can be cracked have been circulating [14,15]. The last service MSDU delivery, is similar to the data delivery service of any LAN standard in the IEEE 802 family.

The remaining group, the Distribution System Services, is the group those services provided by the DS. Only those stations with direct access to the DS, specifically APs, offer the DSS. The five services are: Association, Disassociation, Distribution, Integration, Reassociation. The association service is initiated to create a logical connection between a station and an AP. The Disassociation service is used by either the station or the AP to signify that WLAN resources are either no longer needed or no longer available. The distribution service is responsible for correctly determining where to send frames along the DS, whether to another AP, back into the wireless medium, or to a portal. The final service, the integration service, consists of a portal to the outside world by linking to other wired or wireless LANs. It handles all data that comes into our goes out of the 802.11 ESS.

Like any of the other protocol standards in the 802 family, 802.11 defines two layers of the complete protocol stack: the Medium Access Control layer and the Physical layer. The Medium Access Control (MAC) layer is responsible for ensuring fair, protected, reliable and collision free network operation. The physical layer sits between the MAC and the wireless medium, and manages the transmission and reception of data over the air. Both are discussed in subsequent sections.

The transmit simulation that was created for this thesis implements most of the features of the MAC and Physical layer. A discussion of each follows.

2.2 MAC layer

The MAC layer has a three-fold duty. Firstly, it ensures that data is received reliably, so that when errors occur they are noted and the corrupted data is not passed up the protocol stack. Secondly, it controls access to the wireless medium in a way that both minimizes data collisions and fairly distributes available bandwidth among all the stations. Thirdly, it should protect data from decoding by unwanted listeners. Each of these tasks is made more difficult than in the case of wired networks because of the peculiarities of the wireless medium.

The ensure that transmitted data is received reliably, the MAC layer prepends control, duration, addressing, and sequencing information, and appends a 32-bit cyclic redundancy check (CRC) to the data it receives to transmit. The data itself is referred to as the MAC service data unit (MSDU). When it is surrounded with header and trailer information, it is called the MAC protocol data unit (MPDU). When ready to be sent, the MPDU is sent down to physical layer for transmission. Figure 2-1 shows the general frame format, although many frame types do not include all the available fields. The size in bytes of each field is indicated.

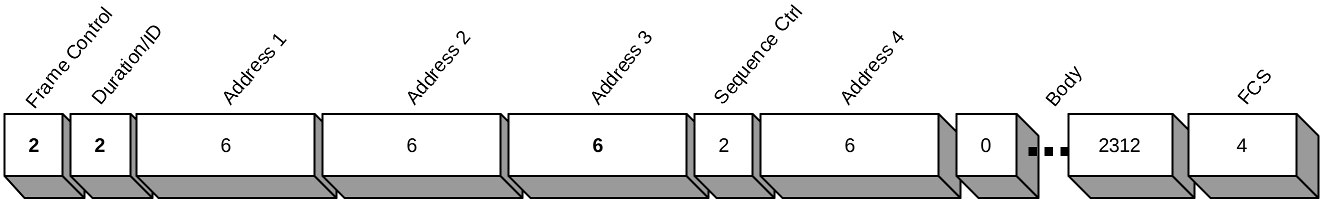

Figure 2-1 - MPDU Frame format

The frame control field is included in every frame. It is broken up further into a protocol revision subfield, a type subfield, a subtype subfield, and eight additional 1-bit flags. The 2-bit protocol version is used to ensure that a received frame is compatible and will be decoded correctly. The 2-bit type subfield characterizes the frame as a Management, Control, or Data frame. Management frames are used to broadcast information about the network as well as access the authentication, deauthentication, association, disassociation, and reassociation services. Control frame are used to affect the state of other terminals and include the acknowledgement (ACK), clear to send (CTS), and ready to send (RTS) frames. Data frames are used to transmit higher-level data payloads. Certain data frames also include control information to avoid the need to send additional separate control frames. The 4-bit subtype field determines the specific frame that is sent in each of the three types. The last eight bit flags are used to determine additional information needed to decode the frame correctly. “To DS” and “From DS” determine how the four addresses are interpreted. “More Frags” indicates that the current MSDU is only a fragment. “Retry” indicates whether this frame is a retransmission. “Power Mgt” is used to alert an AP that the station is going to sleep. “More Data” indicates that additional data frames are prepared to follow this one. “WEP” is used to enable WEP encoding. Finally “Order” indicates that a higher level of the protocol stack requested strictly ordered services. The entire frame control field is shown in Figure 2-2 below, with the size of each bit field indicated in the boxes.

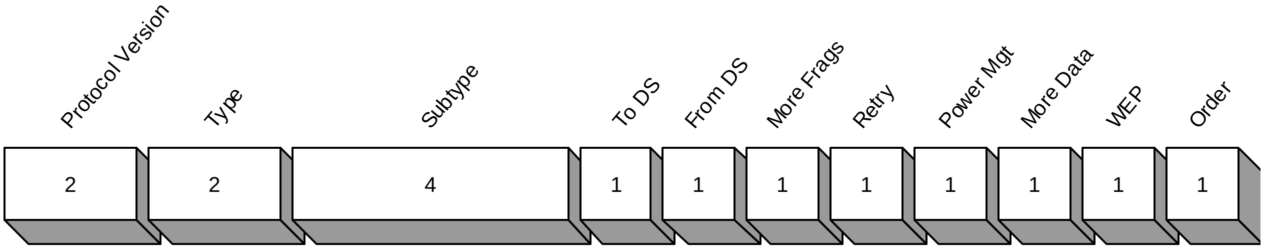

Figure 2-2 - Frame Control Fields

The Duration/ID Field has a dual purpose depending on the frame type. When functioning as a duration, it is used by all stations that receive the packet to update their Network Allocation Vector (NAV). The NAV stores the amount of times in milliseconds before the medium becomes free again. When functioning as an ID, it is used by a station to identify itself to an AP when in power save mode. The first address field is always the destination address, and stations use this address to determine if they need to continue decoding the packet. The remaining three addresses are optional, and their inclusion and decoding depends on the frame type and the To and From DS fields in the frame control. The sequence control field holds two subfields, a sequence number and a fragment number (if the frame is a fragment). The body of the frame contains fields that are specific to each of the frame types. In the case of a DATA frame, the body contains whatever information was passed down from higher in the protocol stack.

To achieve reliable transmission, the MAC must ensure three conditions. First, every packet must eventually reach its destination. Secondly, duplicate packets must be discarded. Thirdly, the data must be received intact. To address the first condition, 802.11 requires that each unicast packet be acknowledged with an ACK frame within 10us of the end of the received packet. If the station that originally transmitted the frame does not receive an ACK, it will retry the same frame a specified number of times. For dealing with the second condition, the sequence control field is used as an identifier for each frame. If a frame with the same sequence number as a previous frame is received, it will be acknowledged but then discarded. For the third condition, the FCS is used as a check on both the MPDU header information and the MSDU.

To control access to the network, the MAC layer uses a medium access scheme named collision sense multiple access with collision avoidance (CSMA/CA) and exponential backoff. This scheme is similar to the collision sense multiple access with collision detection (CSMA/CD) that is used in Ethernet (IEEE 802.3). Collision avoidance, however, must be used instead of collision detection because wireless devices cannot generally transmit and receive at the same time. Exponential backoff comes into play when a station detects that someone else is transmitting directly before the period of time that it wants to transmit. In this case the station sets a random backoff counter whose range increases exponentially with each detection of a potentially interfering transmission. The specifics of the algorithm can be found in chapter nine of the 802.11 standard.

In wired LANS, each node on a network branch is generally guaranteed to be able to communicate with every other node on that branch. In WLANs, however, because of the more rapid falloff in signal strength, a transmitted packet may not be received by remote nodes on the same BSS. This introduces a problem when two or more nodes that are outside each others range both wish to communicate with a third node. As neither of the two transmitting nodes can hear each, each could potentially start a transmission while the other is in the middle of sending a packet. The middle node, which can hear both, would end up receiving corrupted packets. To solve this problem, the standard defines two optional messages that expand on the standard DATA/ACK 2 frame sequence. The first message, Ready to Send (RTS), is a short message sent by the station that wishes to transmit a frame. Upon reception of a RTS message that is addressed to it, the receiving station will respond with a Clear to Send (CTS) frame. Both the RTS and CTS messages set the duration field to the total length of the four-frame sequence. This means that any station receiving either message will update its NAV, and thus know to keep the medium free for the right amount of time. Since both the original sending station and the receiving station send a message, any potentially interfering station that is within listening distance of either station will be notified to wait. The full frame sequence RTS/CTS/DATA/ACK is optional, and can be automatically enabled based on the length of the data packet.

To attempt to guarantee data security at least to the minimal extent provided by the physical wires in a wired LAN, the 802.11 working group adopted an optional private-key encryption algorithm called Wired Equivalent Privacy (WEP). WEP's shortcomings have already been mentioned, and as of this writing it appears that many users who are actually concerned with any sort of security measures are using other, higher level, encryption methods. WEP is based on the RC4 algorithm created by RSA Data Security, Inc. RC4 is variable length key symmetric stream cipher. IEEE chose a 40-bit length key for 802.11, although it seems that most implementations actually allow larger keys. The MSDU is the only portion of the MPDU that is actually encrypted; the header and the FCS are left in plain text. The standard supports the use of either up to 4 default keys, which would be shared among all stations on a BSS, or the creation of a key mapping relationship with a specific station. In the later case only the two stations communicating with each other have access to the key. WEP is enabled by toggling on the WEP flag in the Frame Control field of the MPDU header. Once enabled, three additional fields are added to the MSDU. The first, a 24-bit Initialization Vector is appended to the key when encoding or decoding the frame. After a 6-bit pad, come the 2-bit Key ID, which is used two select among the four available keys. These first two fields are necessarily left unencrypted. Next follow the MSDU and finally a 32-bit Integrity Check Vector (ICV), both of which are encrypted. The inclusion of the ICV ensures any tampering with the MSDU is noted.

2.3 Physical layer – 802.11b specific

The Physical layer (PHY) sits below the MAC and serves as an interface to the physical medium. 802.11 defines three different PHYs: direct sequence spread spectrum (DSSS), frequency hoping spread spectrum, and infrared. Each of these PHYs is broken into two sublayers. The higher Physical Layer Convergence Procedure (PLCP) sublayer provides a uniform interface to the MAC and passes data down to the lower Physical Medium Dependant (PMD) sublayer. The PMD performs the tasks necessary to actually transmit the frame. Only the 802.11 DSSS PHY and its 802.11b extensions will be discussed, as they are relevant to the project.

2.3.1 PLCP sublayer

Upon reception of a frame from the MAC, the PLCP sublayer first adds preamble and header information then passes the entire frame down to the PMD. The preamble consists of a synchronization field (SYNC) and start frame delimiter field (SFD). 802.11b supports two types of

SYNCs, a long version and a short version. The long version consists of a sequence of 128 bits all set to ‘1’, while the short version is 48 bits set to ‘0’. Support for a short preamble was added in 802.11b. The inclusion of a SYNC field allows the receiver time to acquire the signal and synchronize the demodulator before decoding each frame. Following the preamble is a 16-bit start of frame delimiter (SFD). This indicates to the receiver that the preamble is finished. When using a long preamble, the bit sequence: the SFD is [1111 0011 1010 0000]. In the case of the short preamble, the reverse, [0000 0101 110 0111], is sent.

Following the preamble, a 40-bit PLCP header is added. This header contains four fields: signal, service, length, and CRC. The byte long signal field indicates the speed the MPDU will be transmitted at divided by 100 kb/s. In 802.11b, there are four valid values: 0x0A for 1Mbps, 0x14 for 2Mbps, 0x37 for 5.5 Mbps, and 0x6E for 11 Mbps. The one byte service was unused in 802.11, but in 802.11b three bit flags are used: one for length field extension, one for choose CCK or PBCC coding, and one for indicating that the transmitting clock and frequency are locked to the same oscillator. Next is the 16-bit length field, which indicates the duration of the MPDU in microseconds. At 11 Mbps the length field extension bit must be used to decode the number of bits properly. Last is an 8-bit CRC that validates the header information.

The MPDU follows, here called the PLCP Service Data Unit (PSDU). The entire packet, with preamble, PLCP header and PSDU, combines to form the PLCP Protocol Data Unit (PPDU), and this entire frame is passed down to the PMD. Figure 2-3 shows the layout of the PPDU.

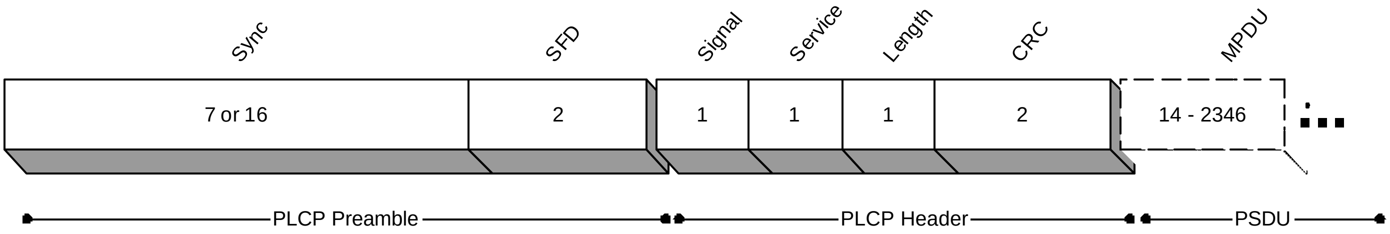

Figure 2-3 - PPDU Frame format

2.3.2 PMD sublayer

The PMD is responsible for transmitting the entire frame onto the wireless medium. Each of the three sections of the PPDU, however, must be handled separately as they might all be transmitted at different rates. Although the preamble is always sent at 1 Mbps, the PLCP header can be sent at either 1 Mbps or 2 Mbps depending on whether the preamble uses a long or short SYNC respectively. The PSDU is always sent at the rate defined in the PLCP header. To transmit at frame, the PMD goes through a four-step process for each section. First it whitens the PPDU by passing the entire frame through a scrambler. Second, it differentially phase shift key encodes the scrambled data to output orthogonal I and Q streams. Third, it spreads the encoded data. Finally, it passes the encoded I and Q streams through a transmit mask filter and then onto the RF chain

The first step, data whitening, is effected by the passing of the entire bit stream through a self-synchronizing 7-bit polynomial. The same 7-bit polynomial is used to unscramble the data. Because the scrambler is self-synchronizing, the receiver will be able to successfully unscramble the frame provided the demodulator is synchronized at least 7 bits before the start of the SFD. The polynomial used is shown in Equation 2-1 below:

$$G(z) = z^{-7} + z^{-4} + 1 \qquad \text{(Eq. 2-1)}$$

The method the PMD uses to PSK encode the data in the second step depends on the transmit speed of the section it is encoding. At 1 Mbps, it differentially binary phase shift key (DBPSK) encodes the entire section one bit at a time. At 2 Mbps, differential quadrature phase shift keying (DQPSK) is used, and the entire section is encoded two bits at a time. At the higher data rates, DQPSK is still used, but only a portion of the section's bits are used for encoding, while the rest are used to determine the spreading sequence. In addition, odd symbols at the higher rates are given an additional 180° rotation. At 5.5 Mbps, two bits of each incoming 4-bit nibble are used. At the highest rate, 11 Mbps, only two bits of each incoming byte are used. Because differential PSK is used in all cases, only the relative position of the symbol is important.

Table 2-1 - DQPSK Encoding

Input Bits	Phase change
00	0°
01	90°
11	180°
10	270°

Table 2-2 - DBPSK Encoding

Input Bit	Phase change
0	0°
1	180°

How the PMD performs the third step, spreading, also depends on the rate of current section. If the current section is sent at 1 Mbps or 2 Mbps, the section is spread with the 11-bit Barker word sequence shown below. This sequence is used because it has good correlation properties to help resolve multipath interference.

$$[\ 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1\ ] \qquad \text{(Eq. 2-2)}$$

At the higher speeds, Complementary Code Keying (CCK) is used. In CCK, the spreading sequence is dependent on the incoming data. At 5.5 Mbps, 2 bits out of every 4-bit nibble are used to choose one of 4 8-chip code words. At 11 Mbps, 6 bits out of every incoming byte are used to choose among 64 8-chip code words. Since CCK codes are already complex, the DQPSK encoding step is included to prevent the need for two separate encoding steps. At both 5.5 Mbps and 11 Mbps, the 8 chips of the complex code word, as transmitted sequentially in time, are determined by the following equations:

$$\begin{array}{ll} c_0 = e^{j(j_1+j_2+j_3+j_4)} & c_1 = e^{j(j_1+j_3+j_4)} \\ c_2 = e^{j(j_1+j_2+j_4)} & c_3 = -e^{j(j_1+j_4)} \\ c_4 = e^{j(j_1+j_2+j_3)} & c_5 = e^{j(j_1+j_3)} \\ c_6 = -e^{j(j_1+j_2)} & c_7 = e^{jj_1} \end{array} \qquad \text{(Eq. 2-3)}$$

At both rates, the angle f $_1$ is determined by the DQPSK encoding (see Table 2-1) of the first two bits, taking into account an extra 180° rotation on odd numbered symbols. At 5.5 Mbps, the remaining angles are defined at follows according to the remaining bits (b2, b3) of the 4-bit nibble.

$$\begin{array}{l} j_2 = (b_2 \cdot 180°) + 90° \\ j_3 = 0 \\ j_4 = b_3 \cdot 180° \end{array} \qquad \text{(Eq. 2-4)}$$

At 11 Mbps, the remaining three angles are defined by the QPSK encoding of the remaining 6 bits in groups of two. The QPSK encoding is shown in Table 2-3.

Table 2-3 - QPSK Encoding

Input Bits	Absolute Phase
00	0°
01	90°
10	180°
11	270°

The fourth and final task of the PMD is to pass the spread I and Q data streams through a transmit mask filter. The baseband filter was designed to meet the 802.11b spectral mask. The filter spreads out the signal and makes it more resistant to channel noise.

2.4 Channel Effects

With any network, the issues that prevent the successful transmission of data are generally caused by one of two categories of problems: either synchronization discrepancies between the transmitter and receiver or interference introduced over the medium that is used as a channel for the data. All these problems are here lumped under the title of Channel Effects. The program described in this paper simulates four Channel Effects that could be used to characterize and optimize the performance of a potential receiver. The first pair, Frequency Doppler and Code Doppler, falls into the first category of synchronization discrepancies. The second pair, Additive White Gaussian Noise (AWGN) and Multipath Fading, is a result of the second category of channel interference. Each effect is described in turn.

Frequency Doppler is introduced due to a difference in the modulation frequency of the transmitted signaland the frequency oscillator at the receiver. This difference can be caused by one of two factors. First, the transmitter might be in motion relative to the receiver. Depending on the direction and speed of the motion, this could result in a noticeable stretching or squashing of the signal, slightly changing the frequency of the signal. Doppler introduced in this way is significant in longer-range wireless application which are likely to be more mobile, such as cell phones, but is less important in 802.11 because of its shorter range. Users are unlikely to remain in range of a network if they are traveling at speeds fast enough to introduce significant Frequency Doppler. The second factor that can introduce Frequency Doppler is a slight difference in the frequency of the oscillator in transmitter and that in the receiver. Although the oscillators may nominally be the same frequency, discrepancies may result since they are independently generated. As a result, Voltage Controlled Oscillators (VCOs) are often used in the receiver to allow it to attempt to match in real time the frequency of the transmitter. This real-time matching is called Phase tracking or Frequency tracking [4, 5] and requires some sort of feedback mechanism in the receiver. To be compliant with 802.11, the standard dictates that a receiver must be tolerant of up to +/- 30 parts per million of frequency error.

Code Doppler is the result of a difference in the frequency in the receiver and in the transmitter of the oscillator that provides the timing reference for digital processing. To decode a transmitted 802.11 signal, a receiver must find the peaks of the incoming signal and decodes them at the chip rate. Unless the timing frequency between transmitter and receiver is synchronized exactly, then these peaks will slide forward or backwards over the course of the packet. If the receiver does not adjust for this, eventually a decoding error could result. Since the independent oscillators that control the timing in the transmitter and receiver cannot be exactly synchronized, the receiver often implements another feedback loop to track the slowly shifting peaks of the signal. This is referred to as time tracking and is implemented with a time tracking loop [4, 5]. The standard again dictates that a receiver must be tolerant of up to +/- 30 parts per million of frequency error.

Possible implementations of a PLL and TTL and their effects on performance are describe later in the paper.

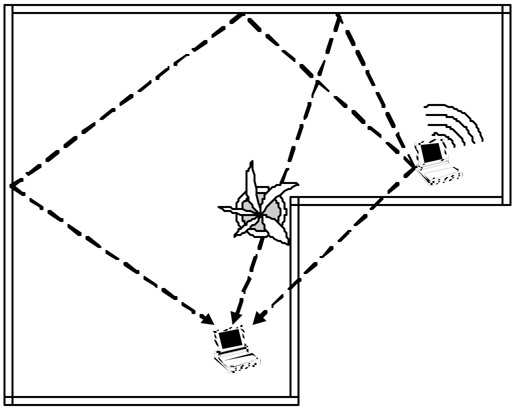

Figure 2-4 - Multipath reception of a transmitted signal

The first of the physical channel related problem, random environmental energy, is the most common interference problem in any communication medium. Since this noise is generally uncorrelated from sample to sample, is additive, and is distributed within a certain range from a zero-mean, it is modeled as Additive White Gaussian Noise (AWGN). The strength of the energy of the signal in relation to noise, called the signal to noise ratio (SNR) and measured in decibels (dB), can be varied to measure the effects of different levels of interference. There are two common ways to help reduce the effect of AWGN that are implemented in 802.11. The first is to pass the received signal through a matched filter, which removes out of band interference and thus lessens the effect of the noise. The second is to use a processing gain, which uses more than one incoming chip to make a signal decision, this takes advantage in the lack of correlation between adjacent sampled noise signals. [13]. The use of the 11-bit Barker code is an example of this, and results in about a 10.4 dB gain in noise tolerance.

The last channel effect, multi-path fading, is one of the most difficult to deal with. It is the result of the fact that from the location of one wireless device to another there are numerous routes a signal could take. Since each of these paths might be a different length and bounce off or pass through different types of objects, multiple copies of the transmitted signal could reach the receiver at different times and different strengths. If these multiple signals are not accounted for, they could cause serious interference and ruin the transmission. To combat this effect, a multi-fingered antenna called a rake receiver is often used [13]. The spacing of the fingers allows the antenna to pull in the different strength delayed signals constructively, not destructively. Additionally, an equalizer can be used correct larger delay spreads by matching the channel and filtering out interchip interference [4]. To ensure valid comparison across different implementations, IEEE 802.11 working group provided a standard mathematical channel model that can be used to quantify performance. It provides for a simulated channel filter with taps whose average power decays exponentially and whose phase is randomly uniformly distributed. Adjusting the T_{RMS} parameter of the model allows the RMS delay spread tolerance of a potential system to be measured. The model is shown below:

$$\begin{aligned} h_k &= N(\frac{1}{2} s_k^2) + jN(\frac{1}{2} s_k^2) \\ s_k^2 &= s_0^2 e^{-kT_s/T_{RMS}} \\ s_0^2 &= 1 - e^{-T_s/T_{RMS}} \end{aligned} \qquad \text{(Eq. 2-5)}$$

Where N(Var) is a zero-mean Gaussian random variable with variance Var, T_s is the sampling period, and T_{RMS} is the RMS delay spread. The number of taps should be large enough to allow for the tail of the filter to decay sufficiently. The IEEE 802.11 Handbook gives the example value for k_{max} equal to 10 times the ratio of T_s/T_{RMS}. Figure 2-5 shows the exponentially decaying channel model as well as a potential realization of the model.

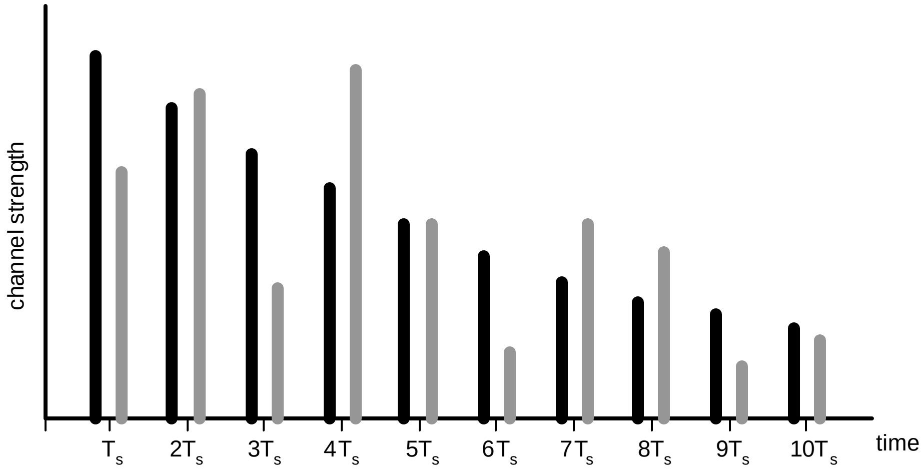

Figure 2-5 - Channel model and realization

3 Transmit Simulation

3.1 Overview

The transmit simulation is programmed as set of C++ classes with a file-configurable front end. The simulation is capable of creating real 802.11b control, data, and management frames. Options can be configured through a class interface. The bitstream can then be generated, and the resultant data is PSK encoded and spread according one of the available 802.11b rates (1, 2, 5.5, or 11 Mbps), creating I and Q streams at the chip rate (11 Mcps). The data streams are then passed through a combination transmit filter and upsampler, outputting data streams at eight times the chip rate (88 Mhz) that could be RF modulated and sent to an actual 802.11 station.

For Simulation purposes, however, this I and Q data can be passed through a simulated physical channel which supports AWGN, frequency Doppler, scaling, and the 802.11 multipath fading model. Code Doppler is also supported through the transmit filter.

To speed up receive processing, the output from the physical channel is passed through a receive filter matched to the transmit filter. The optimized filter in C++ provides a substantial performance gain over its MATLAB counterpart. The last frame generated can also be replayed with a new channel simulation. This feature is useful when a large number of frames need to be generated: it cuts out the processing steps need to parse the input file, created the bits, and transmit filter the frame.

The full interface to each class is listed in Appendix A.

3.2 Class descriptions

The classes that are used to generate and transmit the packet information are described below. The front end that uses them is described in detail in section 3.3.

Figure 3-1 shows each of the steps of the transmit simulation and the top-level classes that are used by the front end to create the actual frame. The main stages are shown with either boxes or symbols. The classes used in each stage are shown adjacent in gray. The top-level classes, helper classes, and the utility classes used to pass data among the top-level classes are all described below.

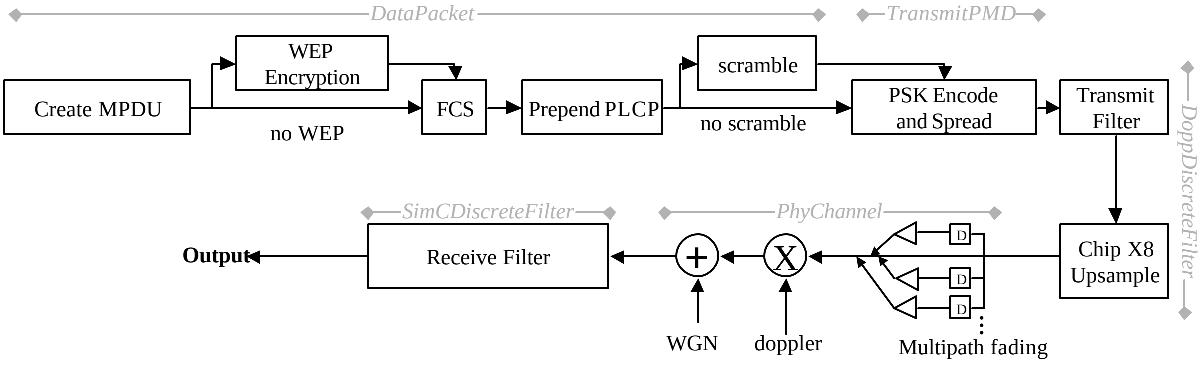

Figure 3-1 - Overview of transmit chain class flow

3.2.1 Basic Types

The simulation defines a few basic types to make the rest of the code more intuitive. The definitions are created primarily to make the exact length of variables explicit. Using known length variables is important because of the need to pack specific numbers of bits into data packets in a specific way. Since C++ does not guarantee the size of its pre-defined types on all architectures, these definitions also allow for easier porting of the code to different architectures. The three types of this sort defined are BYTE, WORD, and DWORD, which represent 8, 16, and 32 bits respectively. Additionally, a *sigtype* type is created to represent a signal sample. Currently *sigtype* is represented as a double precision floating point number. These types are defined “802Typedef.h”, listed in the appendix.

3.2.2 Utility classes

The simulation uses three main utility classes to store packet data and pass it from one class to another. There are each built around the C++ standard template library (STL) *vector* class. Using the *vector* class provides the two-fold benefit of allowing easy access to individual items and fast iteration through the entire collection while at the same time overcoming the inherent limitations of C arrays. Using C array requires either static sized array or dynamic allocation with often misused pointers. This architecture design allows pointers to be avoided almost completely. The utility classes are passed by reference among the top-level classes.

The first class, *PackedBits* (see appendix – “PackedBits.h”), is a bit level representation used when creating the data in a packet. It provides a number of *pack_msb* and *pack_lsb* functions that pack anything from a few bits to a double word of data in either most significant bit (MSB) or least significant bit (LSB) order respectively. Using a bit level representation allows the bits to be added to the packet in the correct order without the need for extensive explicit bit-masking. The *DataPacket* class described below uses these functions to pack bits into a packet. *PackedBits* inherits most of its functionality from the bool specialization of the *vector*.

The second class, *PMSignal* represents a simple, non-complex, physical medium signal. It is an instance of a STL *vector* specialized for the *sigtype* type, adding a few new constructors and utility functions. All the added functions were created in response to needs to the program. Although a number of additional operators and functions could have been added to increase its level of abstraction, for performance reasons most of the code that uses *PMSignal* directly accesses the inherited *vector* functions and iterators.

The last class is *CPMsignal*, is simply a structure of two *PMSignal’s* named I and Q. It is used to store the in-phase and quadrature parts of a signal together while still allowing each to be accessed separately. Using a complex vector to represent both I and Q was inefficient, as most operations could be performed on one and then the other separately. This removed the overhead needed to extract the inphase and quadrature parts for each iteration of an algorithm’s loop. Both *PMSignal* and *CPMSignal* are in the appendix in “PMSignal.h”.

3.2.3 Creating the Packet – DataPacket

The first stage of the transmit chain is to create the data to be transmitted. This task is handled almost entirely by the *DataPacket* class (see appendix – DataPacket.h and DataPacket.cpp). Conceptually, *DataPacket* encapsulates the MAC layer as well as the PLCP sublayer of physical layer of 802.11b. After setting the SYNC type and transmit speed in the constructor, DataPacket operates in three stages.

In the first, the parameters for the packet are set. These include setting the various frame control options, setting the MPDU addresses, enabling WEP, and setting the packet type and subtype. Because the simulation does not have the format of each frame type programmed into it, the fields that are to be included in the frame also need to be selected. All of the parameters have default values that will still result in a technically valid packet. By default, all fields are included. As no bounds checking is performed on parameters to the option functions, the user must be careful to only set values that conform to the standard. A number of fields are auto-generated in the next step and thus cannot be manually set. Since these parameters are always determined by the other settings, this is not a limitation, baring the desire to create intentional error frames. At the end of the first stage, *DataPacket* is prepared to create a packet, although the actual bits are not yet set.

During the second stage, the bits of the packet are created and packed as elements of a PackedBits collection. These steps take place with a sequence of three function calls. The first creates the MPDU, the second prepends the PLCP preamble and header, and the third scrambles the data. The third function call is optional, as the data can optionally be left unscrambled in order to facilitate examination of the bits. As mentioned, DataPacket also auto-generates a few fields. The SYNC bits are created, as is the appropriate SFD depending on the choice of preamble length. The PLCP length and length extension bit are generated for the PLCP header, as is the 16-bit CRC. If WEP encryption is enabled, the MSDU is scrambled and the ICV is generated. The 32-bit FCS is also created.

To create the MPDU, *DataPacket* needs to know what data is to be transmitted. For this purpose the function to create the MPDU takes in an object of the abstract class *DataSim*. *DataSim* defines only three functions: reset, length, and next. Abstracting the data passed to *DataPacket* allows for easier creation of arbitrary length filler data while at the same time not making it much more difficult to use real data. Currently three classes are derived from *DataSim*: *ConstDataSim* always returns one set byte of data; *RandomDataSim* produces a random byte each time; ArrayDataSim takes in a pointer and returns each byte of that array. (See appendix: DataSim.h and DataSim.cpp).

In case WEP is enabled, *DataPacket* uses the *rc4* class (see appendix “rc4.h” and “rc4.cpp”) to encrypt the MPDU data. To successfully enable WEP, the user must first set the key and the IV through the *DataPacket* functions mentioned in the first stage. To use the *rc4* class, *DataPacket* passes the user-supplied key and IV and calls the *PrepareKey* function to prepare the algorithm. Subsequently, each time *rc4*’s encrypt function is supplied with a byte of data it returns the encrypted version of that data. With WEP enabled, *DataPacket* automatically encodes the data, includes the IV and appends the ICV.

The third stage of *DataPacket’s* functionality is to return the bits of the packet. This is done with three accessor functions that separately return the PLCP preamble, PLCP header, and MPDU in the form of a constant *PackedBits* reference. Theoretically the entire packet could have been passed as one bit stream, however, as each part might need to be encoded at a different rate, their separation makes subsequent processing easier.

3.2.4 Encoding and spreading – TransmitPMD

The *TransmitPMD* class performs the tasks of the PHY PMD sublayer. Its tasks include phase shift keying and spreading according to the length of the preamble and the rate of the MPDU.

Following initialization with the MPDU rate and type of preamble to send, *TransmitPMD* requires three function calls before outputting a frame. Each function call corresponds to encoding and spreading one of three parts of the frame. They are sendPreamble, sendHeader, and sendMPDU. All take in a reference to a *PackedBits* class and append their output to a *CPMSignal* , the second argument of each function.

SendPreamble takes in the bits of the preamble, DBPSK encodes them, and spreads them using the Barker code spreading sequence. The result, a 1 Mbps signal, is the only rate supported for the preamble.

SendHeader's job is only slightly more difficult: It encodes the header bits either at 1 Mbps with DBPSK or 2 Mbps with DQPSK, and then spreads them with the Barker sequence. DBPSK is used with a long preamble (144 bits) and DQPSK with a short preamble (56 bits).

SendMPDU, the last function called in the above sequence, must cope with any of the four possible data rates available in 802.11b. When using 1Mbps, the data is encoded in the same way as the preamble, using DBPSK and the Barker sequence. With 2Mbps, the data is encoded at the rate of the faster header with DQPSK. With the faster rates, 5.5Mbps and 11Mbps, sendMPDU calls one of two private CCK encoding functions which handle both encoding and spreading.

3.2.5 Filtering and transmitting – DoppDiscreteFilter

Once the data has been encoded and spread, it passes to *DoppDiscreteFilter*, which has the threefold task of adding Code Doppler, up-sampling and transmit filtering the signal. *DoppDiscreteFilter* is a template class, which would allow it to filter a vector of any type. Although *DoppDiscretFilter* supports any filter length and upsampling rate, in this case a 22-tap filter resampled by a factor of 128 is used. The upsampling rate used is 8.

The signal is up-sampled and filtered by using the discrete definition of convolution. At eight chip intervals, a copy of the transmit filter multiplied by the chip value is combined additively with the last samples of the buffer. The buffer is then extended by the upsampling rate and the process is repeated through the end of the packet. This method works because the upsampled input signal only has information in one out of every eight positions, and so only one out of every eight multiplies is necessary. Filtering this way lessens the number of CPU multiplies necessary by a factor of eight compared to sequentially upsampling and then filtering.

Code Doppler is controlled by adjusting the offset into a transmit filter that has been up-sampled by a factor of 128. This allows for a fine-grained simulation of a time skew in the transmit clock while still keeping the overall sampling rate at eight times the chipping rate. When the overall skew reaches +128 or –128, the transmit pulse is moved one chipx8 symbol to the left or right respectively.

DoppDiscreteFilter sits in “DiscreteFilter.h” (listed in the appendix) along with the other filters.

3.2.6 Simulating the channel – PhyChannel

The *PhyChannel* class (See appendix: PhyChannel.h and PhyChannel.cpp) has the fourfold task of adding AWGN, Frequency Doppler, Scaling, and Multipath Fading. For speed purposes, the effects are additive to the input signal and do not require the creation of new signals. The class takes as input a configuration file that controls the activation and parameters of each of the effects. After instantiation, a call to the public method AddAll adds all of the effect. The effects can also be added individually by calling the appropriate Add[Effect] function. A template configuration file can be found in the appendix.

Additive White Gaussian Noise (AWGN) is added to the signal by first creating a pair of normalized, zero-mean Gaussian random values. These values are then scaled according to the configurable signal to noise ratio (SNR) and added to each sample of the signal. To make this SNR accurate, the signal constellation needed to have an energy of 2, like the output of *TransmitPMD*, and the transmit filter needed to be scaled so as not to effect the energy of the signals.

Frequency Doppler is controlled by a Frequency Error parameter, which is specified in parts per million (PPM) and assumes a 2.4 Ghz signal. The PPM parameter is converted into a time dependent phase and then a two-dimensional rotation is performed on each complex I/Q sample.

Scaling simply multiplies each sample by a user-configurable scaling factor. This can be used to test the receive chain AGC.

The multipath fading effect implements the model adopted by the IEEE 802.11 Working group [12] to simulate environmental multipath effects on a transmitted signal. As per the model, the program constructs a channel filter whose impulse response consists of a randomly distributed phase and a Rayleigh distributed magnitude with exponentially decaying average power [12]. The user can configure the number of Taps as well as the RMS delay spread. The transmitted signal is then simply passed through the randomly generated complex channel filter.

The configuration file supports a simple format for setting channel options of the form:

“VariableName=Value”. The support variable names are listed in the Table 3-7 below. The value is specified as either a floating point or integer number.

Table 3-1 - PhySignal supported configuration variables

Name	Description
CERR	Code Doppler error, specified in parts per million
FERR	Frequency error, specified in parts per million assuming a 2.4 GHz carrier frequency
FPH	Frequency phase. Initial rotated phase of the signal
SNR	Signal to Noise ration, specified in dB.
SCALE	Scaling factor, added after the other effects
TS	Sampling time, specified in seconds, used in creating the random channel filter
TRMS	RMS delay spread, specified in seconds, used in creating the random channel filter
TAPS	Number of complex taps to create in the channel filter.

3.2.7 ReceiveFiltering – SimCDiscreteFilter

When receive filtering is enabled, the last step of the simulation is to pass the complex samples through a receive filter. While this step is not technically part of the transmit chain or the physical channel, it is provided as an option to speed up the MATLAB coded receive chain. The receive filter is a simulated complex discrete filter, meaning that while it filters a complex signal, the filter itself only has a real component. This implementation enables the consolidation of the filtering into one loop while at the same time eliminating the additional two multiplications per sample necessary for a true complex filter.

3.3 Optimizations

Initial executions of the transmit chain ran around 8 seconds, indicating that optimizations were necessary prior to using the simulation to generate large amounts of packets. Some simple profiling was done to attempt to discover where the major speed bottlenecks. As could have been expected, transmit and receive filtering both occupied a substantial portion of the execution time. Additionally, the algorithm originally used to create a Gaussian random variable for AWGN occupied a noticeable portion of running time.

The optimization performed on the filtering classes has already been mentioned above. Combining transmit filtering and upsampling into one step realized the great performance gain. This step reduced the number of multiplications necessary for filtering a packet by eightfold without reducing the accuracy of the filter. The merging of the separate receive filtering loops for I and Q resulted in a significant, if not as dramatic, reduction in running time. The new receive filter ran for around 60% of the time of the original filter.

The creation of a Gaussian RV was originally performed by creating a rayleigh RV from a uniform distribution and multiplying it by the cosine or sine of another uniform RV. This method, while accurate, required a good deal of calculation for each RV and slowed the addition of AWGN to a signal. Another method (see appendix: 802Util.cpp) was substituted and this provided a substantial performance gain.

Should the need arise to further speed up the transmit simulation, a number of additional optimizations are possible, each, however, with diminishing returns. Currently, the C standard math library cos and sin functions are used to calculate the cosine and sine. Replacing these with a lookup table or a taylor series would result in a performance boost, especially when frequency Doppler is introduced in the channel. Additionally, the standard C math library rand function is used to create uniform distribution RV. Replacing this function with a new pseudorandom number generator could also result in a speed up. Any additional optimizations, unless they were to affect the simulation between transmit and receive filtering, would result in a negligible performance increase.

3.4 User interface and packaging

The simulation is compiled into a file configurable front end. This frontend has two interfaces: a command line interface, and a MATLAB native MEX interface.

The command line interface was created to allow for easier debugging and to prevent the need to load up and run MATLAB simply to generate a frame. The MEX interface, on the other hand, provided a close coupling with MATLAB and removed the additional step of loading the file data from disk to process it. When running simulations that created and processed a large number of frames, the approximately 500ms necessary to load a 1024kb data frame from disk proved to be a major time sink.

Originally, there were two frond ends, the now current file configurable one as well as a more basic parameter configurable chain that only generated a few types of packets at the different data rates. The parameter configurable chain had a performance advantage over the file configurable chain because it skipped the pre-processing step of parsing the frame file, but its existence required additional vigilance to ensure both chains were kept up to date. As it became clear that most of the simulations consisted of varying the channel conditions and not the composition of the packet, a last-packet replay feature was added to the file configurable chain. This feature was made possible because files compiled using the MEX toolkit are output as dynamic link libraries (DLL). The execution of a function in one of these DLLs results in that library being loaded into and remaining in memory. This feature allows for state variables to persist over more than call to library functions. Replay takes advantage of this persistence by storing the last packet it generated in memory before it is passed through the simulated channel. The result is that the file-configurable chain runs more quickly than the parameter configurable chain if the replay feature is used. Added flexibility in generating different types of packets thus came without a loss in performance and the parameter configurable chain was dropped.

3.4.1 Implementation

The entire front-end interface to the transmit chain is exposed through a single function called txchain_file (see appendix: “txchain_file.cpp” and “txchain_file.h”). This function takes in two input arguments and a reference to a CPMSignal as an output argument. The two input arguments are strings that hold the frame configuration filename and the channel configuration filename respectively. The channel configuration filename is passed down to an instance of PhyChannel and is handled there. The frame configuration filename, on the other hand, is dealt with directly by txchain_file and its helper functions. The function prototype is shown below:

```
bool txchain_file(string frameFile,string channelFile,CPMSignal &output)
```

Both the MEX interface and the command line interface use this function to interact with the simulation. The two interfaces differ only in the way that they extract the parameters and handle the output. The command line interface – see appendix: txchain_dos.cpp – pass the arguments from the argv parameter onto txchain_file and call a function to write the output out to disk. Since PMSignal already includes a write function to perform that task, outputting is also fairly simple. The MEX interface – see appendix: txchain_mex.cpp – is only slightly more complicated because of its need to call MEX toolkit function to extract parameters and assign the output.

Decoding of the frame configuration file is dealt with by a hierarchy of helper functions to txchain_file that are each responsible for a smaller, and smaller part of the file. txchain_file is responsible for opening the frame file and decoding the top-level tags, or initiating a frame replay. The second level functions, txchain_delay, txchain_frame, and txchain_replay, each handle one of the three possible top-level options. Further functional division is shown in the table below.

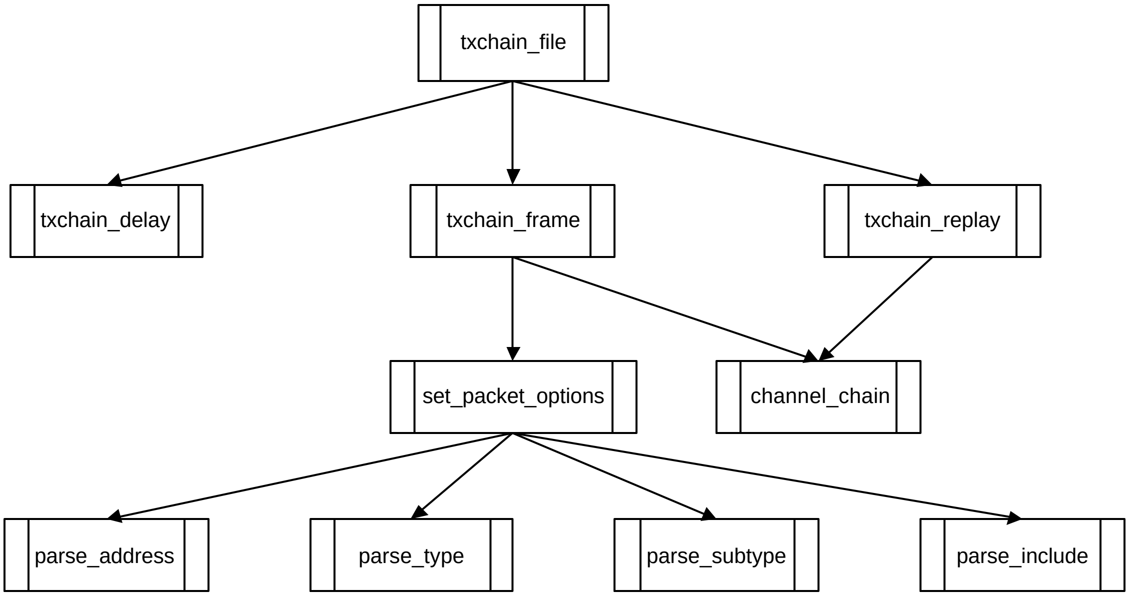

Figure 3-2 - txchain_file function call hierarchy

For aid in parsing the frame configuration file, txchain_file opens the file using the FileToken class. This class handles file input and returns file data in the form of tokens using the C++ ifstream class. FileToken provides a number of functions to read tokens and predigest the data. Tokens can be returned as strings, integer values, arrays, or untyped values. Untyped values can be up to 32-bits long. It can also skip over expected tokens. Single line MATLAB style comments are supported (beginning with a '%') as are multi-line C style comments. FileToken supports data in base-10, hexadecimal (with a leading '0x'), binary (using a trailing 'b'), as ASCII character codes (wrapped in single quotation marks), or as strings (wrapped in double quotation marks). Base-10 number and ASCII codes are always treated as byte long. Hexadecimal and binary numbers can be anywhere from 8-32 bits. FileToken uses C++ exception handling to deal with errors trying to tokenize the input. Exceptions are thrown whenever a token cannot be converted to the type required by a member function. For error reporting purposes, the class keeps track of the current column and row of the current token.

3.4.2 File format

An example of the frame configuration file format is shown below. An ACK frame is used because of its conciseness. Nevertheless, the ACK frame shows all the features of the frame format.

```
% Example ACK frame:
frame(longSync,1mbps)

    Type(Control)  % ACK is a control packet
    Subtype(1101b) % ACK subtype

    Include(FC,DURATION,ADR1,FCS) % Only include four fields
    Duration(0) % Set duration to zero, no frames follow
    Adr1(0xAB,0xCD,0x87,0x22,0x12,0xA1) % Address of sending station

    BitFile("ack.bits") % save bits for examination
end frame
```

Figure 3-3 - Example ACK Frame file

The frame format supports only two top-level tags, delay and frame. A delay tag is used to indicate a delay period where no data is sent. The delay tag must be followed by an integer value and then a time unit of measurement. Possible time units are microseconds, “us”, or milliseconds, “ms.” A frame tag is used to indicate the beginning of a frame. It must be followed by a set of parenthesis containing a comma delineated SYNC type and transmission rate. Possible SYNC types are “longSync” or “shortSync”. Possible transmission rates are “1Mbps”, “2Mbps”, “55Mbps”, or “11Mbps”. The decimal point is dropped from the 5.5 Mbps rate to ease parsing.

Once a frame is begun using a frame tag, any number of packet options can be set. The frame is then ended using a “end frame” tag. Packet options are set using a functional notation of the form: Option(Parameter1,Parameter2,…). By setting the options appropriately, any frame in the 802.11b standard can be created. None of the frames are built into the simulation, however, and so access to the standard is necessary in order to create them correctly. Frame options can roughly be broken down into five functional categories: Frame Control, Data, WEP, Field, and Miscellaneous. Frame Control options only affect the bits of the Frame Control field. Data options set the MSDU. WEP options enable and control WEP encryption. Field options set the included fields and the value of any fields not already set. Miscellaneous options set an assortment of other options such as scrambling and receive filtering. Table 3-2 shows all the options available divided into the five different groups.

The SetData option is used to pack specific data into the MSDU, and is therefore one of the most used options. It supports any number of parameters, each of any size from 8 to 32 bits in byte size increments. An example of the MSDU for an association request (see 7.2.3.4 of the 802.11 standard) is shown below in figure 3-5:

```
  SetData(0000000000010001b,     % Capability info field
         0x00FF,                 % Listen interval
         0x00,                   % SSID Field
         0x08,                   % Length 8
         '8','0','2','1','1','D','e','v',
         0x01,                   % Supported rates
         0x04,                   % 4 supported
         0x02,0x04,0x0B,0x16     % 1,2,5.5 and 11 Mbps
         `
```

Figure 3-4 - Association Request MPDU example

Table 3-2- Frame Configuration Options

Frame Control Options	**Default**	**Description**
`ToDs(b)`	0	Sets the FC ToDs bit to b.
`FromDs(b)`	0	Sets the FC FromDs bit to b.
`Retry(b)`	0	Sets the FC Retry bit to b.
`PowerMgmt(b)`	0	Sets the FC Power Management bit to b.
`Type(type)`	00b	Sets the FC Type to type. Supported types are: Control,Mangement, and Data.
`Subtype(subtype)`	0000b	Sets the FC Subtype to subtype. Subtypes can range from 0-16.
WEP Options	**Default**	**Description**
`WEP(b)`	0	Enables or DisablesWEP. Need to call the other WEP functions if enabled.
`WEPIV(B1,B2,B3)`	-	Sets the three bytes of the WEP IV.
`WEPKey(B1,...,Bn)`	-	Sets the WEP Key, can be any number of bytes upto 16.
`WEPKeyID(d)`	-	Sets the WEP Key ID from 0-3.
Data Options	**Default**	**Description**
`SetData(D1,...,Dn)`	-	Sets the MSDU to D1 through Dn.
`ConstData(L,B)`	-	Sets the MSDU to constant byte value B of length L.
`RandData(L)`	-	Sets the MSDU to random data of length L.
Field Options	**Default**	**Description**
`Include(F1,...,Fn)`	All	Indicates which fields should be included. Valid identifiers are: FC, DURATION, ADR1, ADR2, ADR3, SEQ, ADR4 BODY, FCS).
`Sequence(N)`	0	Sets the sequence number to N.
`Duration(N)`	0	Sets the Duration fields to N.
`Adr1(B1,B2,B3,B4,B5,B6)`	0:0:0:0:0:0	Sets Address 1 to the six bytes B1-B6 in hi-lo order.
`Adr2(B1,B2,B3,B4,B5,B6)`	0:0:0:0:0:0	Sets Address 2 to the six bytes B1-B6 in hi-lo order.
`Adr3(B1,B2,B3,B4,B5,B6)`	0:0:0:0:0:0	Sets Address 3 to the six bytes B1-B6 in hi-lo order.
`Adr4(B1,B2,B3,B4,B5,B6)`	0:0:0:0:0:0	Sets Address 4 to the six bytes B1-B6 in hi-lo order.
`Fragment(N,more)`	-	Makes the Frame a fragment, setting the fragment number to N, and the More fragments bit to more.
Miscellaneous Options	**Default**	**Description**
`Scramble(b)`	1	Enables or Disables scrambling of the bit information.
`RxFilter(b)`	0	Enables or Disable receive filtering of the frame.
`BitFile("filename")`	-	Outputs the bits of the frame to a file.
`Replay(b)`	0	Enables frame replaying on this frame.

The appendix lists additional configuration files (“ProbeReq.frame”, “AssocReq.frame”, “1KB.frame”).

4 Receive simulation

4.1 Receive chain

The receive chain was coded as a set of MATLAB scripts by another member of the project team. The algorithms it uses to decode the frame mimic those that will be implemented in digital hardware. After automatic gain control (AGC) and A to D conversion, the receive simulation runs through three sequential loops to decode the preamble, PLCP header, and MPDU. The channel match filter, phase tracking, and time tracking loops can each optionally be enabled or disabled. The preamble is decoded only if a legitimate SFD is detected, and the MPDU is decoded only if PLCP header CRC passes. The decoded MPDU is then returned to the calling function without checking its validity. To check whether a received packet is valid, any function calling the receive chain must check that a MPDU is returned (its length is greater than 1), and then that the last 32-bits are a valid FCS on the rest of the MPDU.

4.2 Optimizations to the receive chain

As it was originally coded, the fully MATLAB coded receive chain did not run at a fast enough speed to allow for efficient simulation. A 1 KB, 1 Mbps packet required around 60 seconds to decode on a Pentium III 450 Mhz. At this rate, one simulation run of one thousand packets required sixteen hours and forty minutes. Optimizations were therefore necessary. A trade-off had to be reached, however, between spending too many additional hours recoding a constantly changing simulation in C and wasting processing time. Analysis of the MATLAB profiler output showed that only two function classes in the receive chain used a significant percentage of processing time on a per-call basis. These calls were the receive filter and the 32-bit CRC. For the receive filter, it was removed from the receive chain, coded in C, and added as an optional feature of the transmit chain. For the CRC, the call was moved outside of the receive chain, as described in the section above, and was also recoded in C and linked into a MEX DLL (see appendix:crc_mex.c). The C version ran in well under one second.

As this first round of optimizations still left a decoding time of 45 seconds, further work needed to be done. Without taking into consideration the above optimizations, examination of the self time, the time a function spends doing its own calculation, of the top level receive function registered at almost 66%. This high percentage meant that to effect a significant speed up, a portion of the main function needed to be recoded. The third loop therein, MPDU decoding, was both the simplest and the most time consuming, and as such it was the best candidate for replacement. It was therefore rewritten in C. The loop was further simplified by supporting only 1 and 2 Mbps frames. This left a smaller number of subfunctions that needed to be coded: Barker match filtering, descrambling, Phase tracking, time tracking, and DPSK decoding. The recoded third loop is listed in the appendix: MPDU_decode.cpp. After inserting MPDU_decode into the receive chain, the total decoding time was brought down to 3.5 seconds: 1½ seconds for the preamble, ½ second for the header, and 1½ seconds for the MPDU. When combined with an increased higher level overhead due to the CRC, and transmit chain time due to the receive filter, the total time per frame settled around 5.5 seconds. While 5.5 seconds is not ideal, more than an eleven fold relative gain seemed enough of a gain to begin simulation runs.

5 PLL Receive Calculations

Figure 5-1 – 1Mbps and 2Mbps receive chain model

The digital portion of the receive chain for 1 Mbps and 2 Mbps can be modeled by the system shown in figure 5-1. The feedback loop shown above represents the Phase-Locked Loop component of the receiver. In a real system, the complex rotator would probably sit earlier in the chain and the form of a VCO in order to match the carrier frequency in the analog domain. The PLL tracks the differences in the received frequency of the system and the reference frequency on the receiver. As mentioned before, 802.11 requires a receiver to be able to support at least 30 PPM in frequency error. Because of its shortened range, most of the frequency error introduced in an 802.11 system is created by the differential between the oscillator of the sending system and the reference of the receiving one. This discrepancy generally results in a fixed frequency differential over the course of a packet: an input step in terms of frequency and a linear ramp in terms of phase.

As 802.11b uses either DBPSK or DQPSK, the goal of the PLL is to push the incoming complex signal to one of the axes. In the QPSK cases, it drives the signals to the In-Phase or Quadrature axis. For the BPSK case, only the In-Phase axis is used. Since 802.11b uses differential PSK, the symbol information can easily be stripped by removing rotations of 180° in BPSK, or of any 90° increments for QPSK in order to calculate the discriminator.

Since the PLL attempts to drive the symbols to one of the axes, the ideal discriminator would be the angle from the nearest axis after the signal information is removed. Actually calculating the angle would be expensive in hardware. By using a small angle approximation, however, we can approximate this angle: If we are driving the symbol towards the In-Phase axis, then the error angle can be approximated by the Quadrature portion of the signal and vice versa. This system works fairly well, but becomes extremely dependent on the AGC loop in the system and this dependence needs to be taken into consideration.

The system shown above can be approximated by abstracting away from the implementation details of the PLL to simply looking at the phase angle characteristics of the system. This can be seen in figure 5-2. From this figure one can derive the continuous time closed loop function in terms of the loop filter f(t).

Figure 5-2 – Continuous time PLL model

5.1 Continuous time derivation

From figure 5-2, the definition of the error angle, f (t) is given in terms of the input angle, q (t) and the estimated angle $\hat{q}(t)$:

$$f(t) = q(t) - \hat{q}(t) \qquad \text{(Eq. 5-1)}$$

The voltage-controlled oscillator used to generate the estimated angle to match the measured angle can be modeled as an integrator taking as its input the output of the loop filter:

$$\begin{aligned} \hat{q}(t) &= K_2 \int_0^t E(e)de \\ \frac{d\hat{q}(t)}{dt} &= K_2 E(t) \end{aligned} \qquad \text{(Eq. 5-2)}$$

The output of the loop filter E(t), can be given by a convolution of the input error measurement e(t) and the filter transfer function f(t):

$$E(t) = \int_0^t e(e) f(t - t) dt \qquad \text{(Eq. 5-3)}$$

Taking the derivative of Equation 5-1 and substituting in Equations 5-2 and 5-3, gives the following result:

$$\frac{df(t)}{dt} = \frac{dq(t)}{dt} - \frac{d\hat{q}(t)}{dt}$$
$$\frac{df(t)}{dt} = \frac{dq(t)}{dt} - K_2 E(t)$$
$$\frac{df(t)}{dt} = \frac{dq(t)}{dt} - K_2 \int_0^t e(t) f(t - t) dt$$

$$\frac{df(t)}{dt} = \frac{dq(t)}{dt} - K_2 e(t) * f(t) \qquad \text{(Eq. 5-4)}$$

In order to use the results of Equation 5-4, the error measurement, e(t) needs to be linearized. This linearization can be done by claiming that for small angles, the error angle is equal to the difference of the input and the VCO. If this simplification is done, the error measurement becomes very dependent on the scaling of the signals, and thus on the output of the Automatic Gain Controller (AGC). To represent a simplified model of the AGC, a scaling constant A_0 is used. An additional scaling constant K_1 is introduced to offset this value:

$$e(t) = A_0 K_1 f(t)$$
$$f(t) = \frac{e(t)}{A_0 K_1} \qquad \text{(Eq. 5-5)}$$
$$\Phi(s) = \frac{F_e(s)}{A_0 K_1}$$

If we take the transform of Equation 5-4 and combine it with that of equation 5-5, we get the closed loop transfer function of the system in terms of the loop filter:

$$s\Phi(s) = s\Theta(s) - K_2 F_e(s) F(s)$$

$$\frac{F_e(s)}{A_0 K_1} s = s\Theta(s) - K_2 F_e(s) F(s)$$

$$F_e(s)(s + A_0 K_1 K_2 F(s)) = A_0 K_1 s\Theta(s)$$

$$\frac{F_e(s)}{\Theta(s)} = \frac{A_0 K_1 s}{s + A_0 K_1 K_2 F(s)} \qquad \text{(Eq. 5-6)}$$

If we substitute the error angle $f(s)$ back in for the error measurement $F_e(s)$ in Equation 5-6, and plug in the transform of equation 5-1, the result is the final closed loop transfer function of the estimated angle over the input angle:

$$\Phi(s) = \frac{s}{s + A_0 K_1 K_2 F(s)} \Theta(s)$$

$$\Theta(s) - \hat{\Theta}(s) = \frac{s}{s + A_0 K_1 K_2 F(s)} \Theta(s)$$

$$\hat{\Theta}(s) = \Theta(s)(1 - \frac{s}{s + A_0 K_1 K_2 F(s)})$$

$$\frac{\hat{\Theta}(s)}{\Theta(s)} = \frac{A_0 K_1 K_2 F(s)}{s + A_0 K_1 K_2 F(s)} \qquad \text{(Eq. 5-7)}$$

These two final closed loop equations can be transformed into a discrete time transfer function by using the discrete time equivalent model shown in figure 5-3.

5.2 Discrete time derivation

Figure 5-3 – Discrete time PLL model

A discrete version of equation 5-1, and its transform can be extracted as below:

$$\begin{aligned} f(n\Delta t) &= q(n\Delta t) - \hat{q}(n\Delta t) \\ f[n] &= q[n] - \hat{q}[n] \\ \Phi(z) &= \Theta(z) - \hat{\Theta}(z) \end{aligned} \qquad \text{(Eq. 5-8)}$$

performing the same linearization as in equation 5-5 and passing the signal through a discrete loop filter with impulse response f[n] we can extract an equation and transform for the output of the loop filter in terms of the discrete error measurement:

$$\begin{aligned} E[n] &= A_0 K_1 \sum_{k=-\infty}^{\infty} f[k] f[n-k] \\ F_E(z) &= A_0 K_1 \Phi(z) F(z) \end{aligned} \qquad \text{(Eq. 5-9)}$$

The VCO can be discreteized by converting the integrator to a summation using a delay element:

$$\hat{q}(t) = K_2 \int_0^t E(e)de \Rightarrow \hat{q}[n] = K_2 \sum_{t=0}^{n} E[t]\Delta t \qquad \text{(Eq. 5-10)}$$

Equation 5-10 results in a simple difference equation and a first order transform:

$$\begin{aligned} \hat{q}[n+1] &= \hat{q}[n] + K_2 E[n]\Delta t \\ z\hat{\Theta}(z) &= \hat{\Theta}(z) + K_2 \Delta t F_E(z) \end{aligned} \qquad \text{(Eq. 5-11)}$$

Plugging the transform from equation 5-9 into the last equation, gives us the estimated angle in terms of the discrete error measurement:

$$\begin{aligned} \hat{\Theta}(z)(z-1) &= A_0 K_1 K_2 \Delta t \Phi(z) F(z) \\ \hat{\Theta}(z) &= \frac{A_0 K_1 K_2 \Delta t F(z)}{(z-1)} \Phi(z) \end{aligned} \qquad \text{(Eq. 5-12)}$$

We can expand this from the first discrete equation to give us the closed loop transfer function in terms of the discrete loop filter:

$$\hat{\Theta}(z) = \frac{A_0 K_1 K_2 \Delta t F(z)}{(z-1)}(\Theta(z) - \hat{\Theta}(z))$$
$$\frac{\hat{\Theta}(z)}{\Theta(z)} = \frac{A_0 K_1 K_2 \Delta t F(z)}{z - 1 + A_0 K_1 K_2 \Delta t F(z)} \qquad \text{(Eq. 5-13)}$$

5.3 Loop filter design

Given the transfer function in both continuous and discrete time, the design of the loop filter can be examined in terms of its effect on the resultant error measurement of the system. The final closed loop continuous time equation can be simplified by examining the output error Fe(s) as opposed to the output angle measurement:

$$\frac{F_e(s)}{\Theta(s)} = \frac{A_0 K_1 s}{s + A_0 K_1 K_2 F(s)}$$

The complexity of the loop filter that needs to be implemented can be determined by examining the performance of a few different filters under input stress. Equations 5-14 through 5-16 list the three different loop filters examined, in order of increasing complexity from first to third order:

$$f_1(t) = d(t) \qquad F_1(s) = 1 \qquad \text{(Eq.5-14)}$$

$$f_2(t) = d(t) + a \qquad F_2(s) = 1 + \frac{a}{s} \qquad \text{(Eq.5-15)}$$

$$f_3(t) = d(t) + a + bt \qquad F_1(s) = 1 + \frac{a}{s} + \frac{b}{s^2} \qquad \text{(Eq.5-16)}$$

As an estimate for $q(t)$, we can use an input signal that includes a step position, velocity, and acceleration [9]. This system can be modeled with the following equation:

$$q_{est}(t) = (q_c + u_c t + a_c t^2)u(t)$$
$$\Theta_{est}(s) = \frac{q_c}{s} + \frac{u_c}{s^2} + \frac{2a_c}{s^3} \qquad \text{(Eq.5-17)}$$

Since $q_{est}(t)$ is a linear, time-invariant system, the effect of each portion of this input can be evaluated separately without changing the effects of the other parts. We can use the final value theorem view the performance of the PLL with each of the different loop filters installed.

Table 5-1, shows the steady state error of each loop filter in relation to the different parts of the input signal. Appendix B contains the value evaluation of each of the nine cases. A sample case is listed below for the effect of the velocity step on a first order PLL.

$$
\begin{aligned}
F_e(s) &= \frac{A_0 K_1 s}{s + A_0 K_1 K_2}\Theta(s) \\
F_e(s) &= \frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2} \\
\lim_{t\to\infty} F_e(t) &= \lim_{s\to 0} s\frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2} \\
&= \frac{A_0 K_1 u_c}{A_0 K_1 K_2} = \frac{u_c}{K_2}
\end{aligned}
\tag{Eq.5-18}
$$

Here, the velocity step on the first order PLL results in a constant error dependent on the speed of the velocity step and the constant K_2.

Table 5-1 – PLL Input Response

Loop Filter:	Phase Step	Velocity Step	Acceleration Step
1st Order	0	$\frac{u_c}{K_2}$	Monotonically increasing
2nd Order	0	0	$\frac{2a_c}{aK_2}$
3rd Order	0	0	0

In the case of a 802.11b system, a short distance WLAN, the PLL is needed less for channel interference caused by relative motion of the transmitter and receiver than for differences in the respective reference oscillators. The reason for this is that a significant, sustained velocity differential would generally move the transmitter and receiver too far away to communicate in any case. The same holds true for acceleration. Short bursts of acceleration (caused, for instance, by dropping a computer) could cause significant interference, but the layers that sit on top of 802.11b, such as TCP/IP, can retransmit the lost frames. As the wireless medium used by 802.11b can have significant interference (for example by Bluetooth devices or microwave ovens) occasional lost frames are fairly routine.

As the 802.11b standard requires that any IEEE compliant system support up to 30 PPM of frequency error, using only a 1st order PLL would reduce performance considerably. For handling acceleration, because the steady state error is both linearly dependent on the magnitude of the acceleration and constant for constant acceleration, a second order Loop filter should be able to cope by slow movement, such as that caused by pedestrian motion, without overly affecting performance. The benefit in terms of performance of a third-order loop filter over a second order one would be fairly small. A second order PLL seems therefore to be the best tradeoff between hardware complexity and performance.

5.4 Continuous Time Second order PLL

Since we are using a second order loop filter, its transfer function and impulse response are as follows:

$$F(s)=1+\frac{a}{s} \qquad f(t)=d(t)+a \tag{Eq. 5-19}$$

Substituting this into the closed loop transfer function (Equation 5-7), we get the full transfer function of the system:

$$\frac{\hat{\Theta}(s)}{\Theta(s)}=\frac{?_0 ?_1 ?_2 F(s)}{s+ ?_0 ?_1 ?_2 F(s)}$$

$$\frac{\hat{\Theta}(s)}{\Theta(s)}=\frac{?_0 ?_1 ?_2 (1+{}^{a}\!/_{s})}{s+ ?_0 ?_1 ?_2 (1+{}^{a}\!/_{s})}\cdot\frac{s}{s}$$

$$\frac{\hat{\Theta}(s)}{\Theta(s)}=\frac{?_0 ?_1 ?_2 s+a ?_0 ?_1 ?_2}{s^2+ ?_0 ?_1 ?_2 s+a ?_0 ?_1 ?_2} \tag{Eq. 5-20}$$

This is a second order equation of the form shown below, in which the natural frequency (ω_n), damping ration (ζ), and loop bandwidth (B_L) are given by combination of the four parameters:

$$\frac{2zw_n k + w_n^2}{s^2 + 2zw_n s + w_n^2} \qquad \text{(Eq. 5-21)}$$

$$\begin{aligned} w_n^2 &= a?_0 ?_1 ?_2 \\ 2zw_n &= ?_0 ?_1 ?_2 \end{aligned} \qquad \text{(Eq. 5-22)}$$

It is also useful to extract the loop bandwidth of the system B$_L$:

$$B_L = \int_0^{\infty} \left| \frac{\hat{\Theta}(2pf)}{\Theta(2pf)} \right|^2 df$$

by using equation 5-23 [J-T Wu]:

$$B_L = \frac{1}{2} w_n \left(z + \frac{1}{4z}\right) \qquad \text{(Eq. 5-23)}$$

With the natural frequency and damping ration defined in terms of the constants above, the loop bandwidth can be defined as:

$$B_L = \frac{1}{4} (?_0 ?_1 ?_2 + a) \qquad \text{(Eq. 5-24)}$$

5.5 Discrete Time second order PLL

The first step to determining the derivation is to take the difference between the output of the loop filter over a time step:

$$E[n] - E[n-1] = \int_0^{n\Delta t} e(e)\, f(n\Delta t - e)\, de - \int_0^{(n-1)\Delta t} e(e)\, f((n-1)\Delta t - e)\, de \qquad \text{(Eq. 5-25)}$$

Substituting loop filter from the continuous time case (Equation 5-19) simplifies Equation 5-25 down to one integral and the difference of two input functions evaluated a sampling period apart:

$$\begin{aligned} E[n]-E[n-1] &= e(n\Delta t)+\int_{0}^{n\Delta t} e(e)\,ade - e((n-1)\Delta t)-\int_{0}^{(n-1)\Delta t} e(e)\,ade \\ E[n]-E[n-1] &= e(n\Delta t)-e((n-1)\Delta t)+a\int_{(n-1)\Delta t}^{n\Delta t} e(e)\,de \end{aligned} \qquad \text{(Eq. 5-26)}$$

Converting the entire equation to discrete time requires a discrete approximation of the integral. This can be done a number of ways, from a simple Euler integration to a more complicated interpolation. I chose to use the average value between the two samples as a good tradeoff between accuracy and complexity:

$$E[n]-E[n-1]=e[n]-e[n-1]+a\Delta t\,\frac{(e[n-1]+e[n])}{2} \qquad \text{(Eq. 5-27)}$$

Taking the Z-Transform of Equation 5-27 gives the loop filter transfer function:

$$\begin{aligned} E(z)-z^{-1}E(z) &= e(z)-z^{-1}e(z)+\frac{a\Delta t}{2}(z^{-1}+1)e(z) \\ E(z)(1-z^{-1}) &= (1+a\Delta t/2-z^{-1}(1-a\Delta t/2))e(z) \\ E(z) &= \frac{(1+a\Delta t/2-z^{-1}(1-a\Delta t/2))}{(1-z^{-1})}e(z) \end{aligned} \qquad \text{(Eq. 5-28)}$$

Plugging the discrete loop filter into Equation 5-13, and performing the same linearization on the phase angle gives a second order transfer function relating the input difference to the output angle:

$$\begin{aligned} \hat{\Theta}(z)(z-1) &= K_2\Delta t\,\frac{(1+a\Delta t/2-z^{-1}(1-a\Delta t/2))}{(1-z^{-1})}e(z) \\ \hat{\Theta}(z) &= A_1K_1K_2\,\frac{z^{-1}(1+a\Delta t/2-z^{-1}(1-a\Delta t/2))}{(1-z^{-1})^2}f(z) \end{aligned} \qquad \text{(Eq. 5-29)}$$

Taking a discrete version of equation 5-1 and its transform:

$$\begin{aligned} f(n\Delta t) &= q(n\Delta t) - \hat{q}(n\Delta t) \\ f[n] &= q[n] - \hat{q}[n] \\ \Phi(z) &= \Theta(z) - \hat{\Theta}(z) \end{aligned} \qquad \text{(Eq. 5-30)}$$

we can now solve for the final closed loop form of the discrete equation, using Equations 5-29 and 5-30,

$$\hat{\Theta}(z) = A_0 K_1 K_2 \frac{z^{-1}(1 + a\Delta t/2 - z^{-1}(1 - a\Delta t/2))}{(1 - z^{-1})^2} (\Theta(z) - \hat{\Theta}(z))$$

$$\hat{\Theta}(z)(1 + A_0 K_1 K_2 \frac{z^{-1}(1 + a\Delta t/2 - z^{-1}(1 - a\Delta t/2))}{(1 - z^{-1})^2} = A_0 K_1 K_2 \frac{z^{-1}(1 + a\Delta t/2 - z^{-1}(1 - a\Delta t/2))}{(1 - z^{-1})^2} \Theta(z)$$

$$\frac{\hat{\Theta}(z)}{\Theta(z)} = \frac{A_0 K_1 K_2 \Delta t \cdot z^{-1}(1 + a\Delta t/2 - z^{-1}(1 - a\Delta t/2))}{(1 - z^{-1})^2 + A_0 K_1 K_2 \Delta t \cdot z^{-1}(1a\Delta t/2 - z^{-1}(1 - a\Delta t/2))} \cdot \frac{z^2}{z^2}$$

$$\frac{\hat{\Theta}(z)}{\Theta(z)} = \frac{A_0 K_1 K_2 \Delta t(1 + a\Delta t/2) z - A_0 K_1 K_2 \Delta t(1 - a\Delta t/2))}{z^2 + (A_0 K_1 K_2 \Delta t + A_0 K_1 K_2 \Delta t^2 a/2 - 2) z + (A_0 K_1 K_2 \Delta t^2 a/2 - A_0 K_1 K_2 \Delta t + 1)}$$

(Eq. 5-31)

Using the quadratic formula confirms that the poles from Equation 8 are related to those of Equation 18 by the standard relationship. A pole at s = -a is matched by a pole at z = $e^{-a\Delta t}$.

In order to implement the system in simulation, the full system difference equation is needed. This can be extracted from equations 5-30 and 5-29:

$$\begin{aligned} \hat{q}[n+1] &= \hat{q}[n] + K_2 \Delta t E[n] \\ E[n] &= E[n-1] + A_0 K_1 (1 + a\Delta t/2) f[n] - A_0 K_1 (1 - a\Delta t/2) f[n-1] \end{aligned} \qquad \text{(Eq. 5-32)}$$

By introducing a new function w[n], the above can be rewritten:

$$\hat{q}[n+1] = \hat{q}[n] + K_2 \Delta t(E[n] + A_0 K_1 (1 + a\Delta t/2) f[n] - A_0 K_1 (1 + a\Delta t/2) f[n]))$$

$$w[n] = K_2 \Delta t E[n] - A_0 K_1 K_2 \Delta t(1 + a\Delta t/2) f[n] = K_2 \Delta t E[n-1] - A_0 K_1 K_2 \Delta t(1 - a\Delta t/2) f[n-1]$$

(Eq. 5-33)

All that is left to be in is to substitute w[n] in for E[n] and this gives the final form:

$$\begin{aligned}\hat{q}[n+1] &= \hat{q}[n] + w[n] + A_0 K_1 K_2 \Delta t(1 + a\Delta t/2) f[n] \\ w[n] &= w[n-1] + aA_0 K_1 K_2 \Delta t^2 f[n-1]\end{aligned} \qquad \text{(Eq. 5-34)}$$

If we substitute in the equations for the natural frequency and damping ratio from the continuous time derivation, then we get the final form of the difference equation expressed in terms of these two parameters and the sampling period:

$$\begin{aligned}\hat{q}[n+1] &= \hat{q}[n] + w[n] + (2zw_n\Delta t + (w_n\Delta t)^2/2) f[n] \\ w[n] &= w[n-1] + (w_n\Delta t)^2 f[n-1]\end{aligned} \qquad \text{(Eq. 5-35)}$$

This equation can be used in simulation to write a second order PLL

5.6 PLL Implementation

Implementation of the second order PLL requires setting the values for the damping ratio and natural frequency. To do this, we can look at the effects that these variables have on the noise performance and tracking time. This will be done for the simplest case of BPSK encoding.

With the continuous case, phase jitter of the system can be modeled by introducing two Gaussian variables onto the incoming I and Q samples:

$$\begin{aligned}I_n(t) &= I(t) + n_i(t) \\ Q_n(t) &= Q(t) + n_q(t)\end{aligned} \qquad \text{(Eq. 5-36)}$$

If we assume the system is already locked, then the Quadrature portion of the input signal can be assumed to be zero, because we are assuming a BPSK signal locked onto the In-Phase axis. Provided the amplitude of the input signal is a good deal larger than the amplitude of the noise, then using the small angle approximation, the phase jitter can be modeled as below:

$$I_n(t) = I(t) + n_i(t) \quad Q_n(t) = n_q(t)$$

$$q_n(t) \approx \frac{n_q(t)}{I(t)} \qquad \text{(Eq. 5-37)}$$

Assuming that the noise is circularly symmetric, then the average power of the phase jitter $\overline{q}_n^{\,2}$ can be modeled in terms of the signal to noise ratio:

$$\overline{q}_n^{\,2} = \frac{\overline{n}_q^{\,2}}{\overline{I}} = \frac{\overline{n}^{\,2}}{2\overline{I}^{\,2}} = \frac{1}{2SNR} \qquad \text{(Eq. 5-38)}$$

Since average power of the phase jitter $\overline{q}_n^{\,2}$ is equal to the sum of the spectral density of the noise over all frequencies, the effect of the phase jitter on the output of the VCO, $\hat{q}$ (t) can be modeled by integrating the spectral density of the noise $S_{q\,n}(f)$ multiplied by the power of the transfer function integrated over all frequencies:

$$\overline{q}_{out}^{\,2} = \int_0^\infty S_{q\,n}(f)\left|\frac{\hat{\Theta}(2pf)}{\Theta(2pf)}\right|^2 df \qquad \text{(Eq. 5-39)}$$

Since we are assuming AWGN, the spectral density is simply a constant over all frequencies, and so Equation 5-39 becomes:

$$\overline{q}_{out}^{\,2} = S_{q\,n}(f)B_L \qquad \text{(Eq.5-40)}$$

Thus the effect of the phase jitter on the output is dependent on the SNR of the signal and the loop bandwidth. Examining the loop bandwidth equation for a 2nd order PLL from equation 5-23, the minimum occurs at ? = 0.5. BL also stays at under 125% of the minimum of .25 < ? < 1 [J-T Wu].

If we set damping ration (ζ) to $2^{-½}$, then the poles of the system are equidistant from the real and the jw axis. This makes the system act like a 2nd order Butterworth filter, in which the magnitude of the filter remains flat over the widest bandwidth before decaying [11]. On account of this, the time constant for the system can be characterized by the following equation [4]:

$$T_c = \frac{3}{4B_l} \qquad \text{(Eq. 5-41)}$$

Depending on the value of the time constant, there are four possible outcomes when testing the system under the influence of frequency Doppler spread. First, the PLL will not be able to pull in the frequency error at all. In this case, the T_C is far too large, and the packet will fail. Second, the PLL will pull in the frequency error, but too slowly. In this case the packet preamble will expire without frequency lock having been achieved and symbol errors will occur. Third, T_C could be too large, and the PLL will pull in the frequency error but the large amount of residual error will result in symbol errors. Last, T_C could be set correctly and the frequency error will be pulled in while at the same time the residual error because of AWGN could be small enough not to result in symbol decision errors. The goal of the following simulations was to discover the ideal time constant, and whether using a sequence of time constants would help decoding.

6 Results

6.1 Commercial station interaction

One of the first tests performed with the transmit simulation was to pass its output through an arbitrary waveform generator (AWG) into the base band RF of a commercial 802.11b card set up as an AP. The input and output to and from the commercial 802.11b card was connected to an oscilloscope and so could be captured and decoded offline.

A probe request packet was created with a broadcast to address, and a sending address of another 802.11b card. A probe request packet was used because it would elicit a response packet (a probe response) regardless of the state of communication between two stations: A probe request is generally the first communication between a station and an AP. Figure 6-1 shows the results of the transmission of the probe request packet. Since the probe request is a broadcast packet, it is not acknowledged by the AP with an ACK, instead the AP simply responds with a probe response at its leisure. The addressed probe response is sent forth both over the cable to the oscilloscope and over the air. As a side effect of the transmission, due to the fact the from address of a real station was used to generate the request, that station responded with an acknowledgement to the AP's probe response. The ACK, as per the standard, was sent within 10us of the reception of the probe response. It can only be differentiated because of its much lower power level.

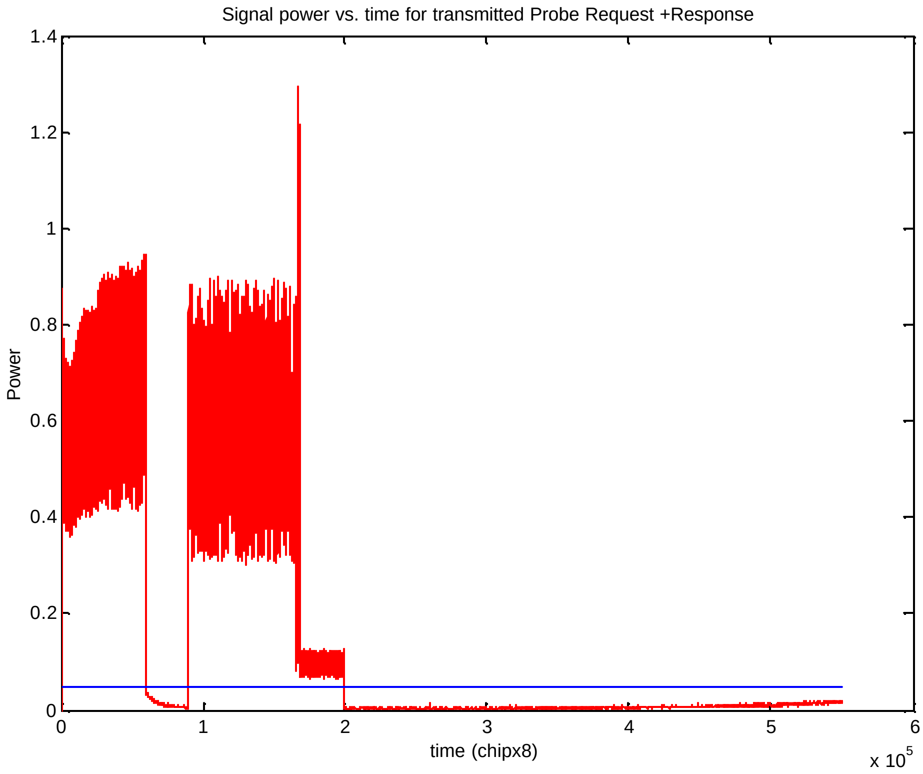

Figure 6-1 - Commercial station interaction: Probe request + response

6.2 Reference Curve – No Doppler or multipath channel

When optimally configured, the best performance level the receiver tracking loops could possible reach will never be better than the performance of the system without any frequency or code Doppler. A good measure of the performance of a phase tracking or time tracking loop, therefore, would be how close it approaches this ideal. To this end, a reference packet error rate curve was generated over a range of signal-to-noise ratios. This curve shows receiver performance in an AWGN channel with a perfectly rotated signal. The signal can be perfectly rotated because the transmitting constellation is known (the signal is encoded at a 45 degree offset). Figure 6-2 shows the result. In subsequent analysis, references will be made to this figure to compare the efficiency in dB that has been lost with the introduction of frequency or code error.

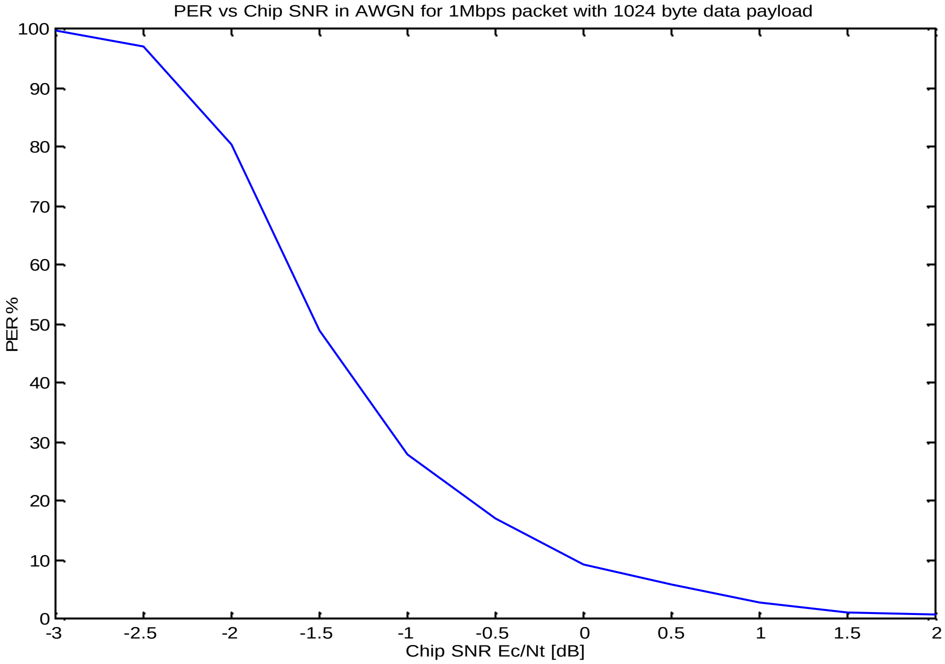

Figure 6-2 - PER vs. Chip SNR in AWGN for 1Mbps 1KB Packet

6.3 Automatic Frequency Correction and Phase locked loop optimizations

6.3.1 Separate, Coexisting AFC and PLL

The first simulation tests performed were on a receive chain that used Automatic Frequency Correction (AFC) to correct frequency error and a separate Phase Locked Loop (PLL) to remove residual phase. In theory, the AFC would remove almost all 1st order phase changes, leaving a nearly constant phase for the PLL.

The AFC operates by comparing the two dimensional dot and cross product of the two most recent samples. It uses this comparison to ignore any phase that is the result of the phase shift encoding, leaving only an incremental phase between the two samples. It calculates a new cross product from this incremental phase and then multiplies it by a gain value and adds that result to the current frequency error. This frequency error then modifies the current phase error.

The PLL works by calculating a phase discriminator, multiplying it by a gain value, and then using that result to push the phase to one of the axis.

The first test performed was the optimization of the AFC gain parameter under the stress of frequency error. For this test, an automated MATLAB script was created to link packet creation (the transmit chain) with packet decoding (the receive chain), see appendix – full_chain_kd.m.

The gain parameter (Kd), was swept logarithmically over a range of 500 to 5000 for eighteen different values. This set of Kd values was sent to the full chain, which ran 1000 trials for each Kd. The trials used the SNR of 1.5 db and a 10 part-per-million simulated frequency error. The results were similar to what was expected: an inverted bell shape Packet error rate vs. Kd graph. See figure 6-3. The lowest point on the curve was for a Kd value of around 1470.

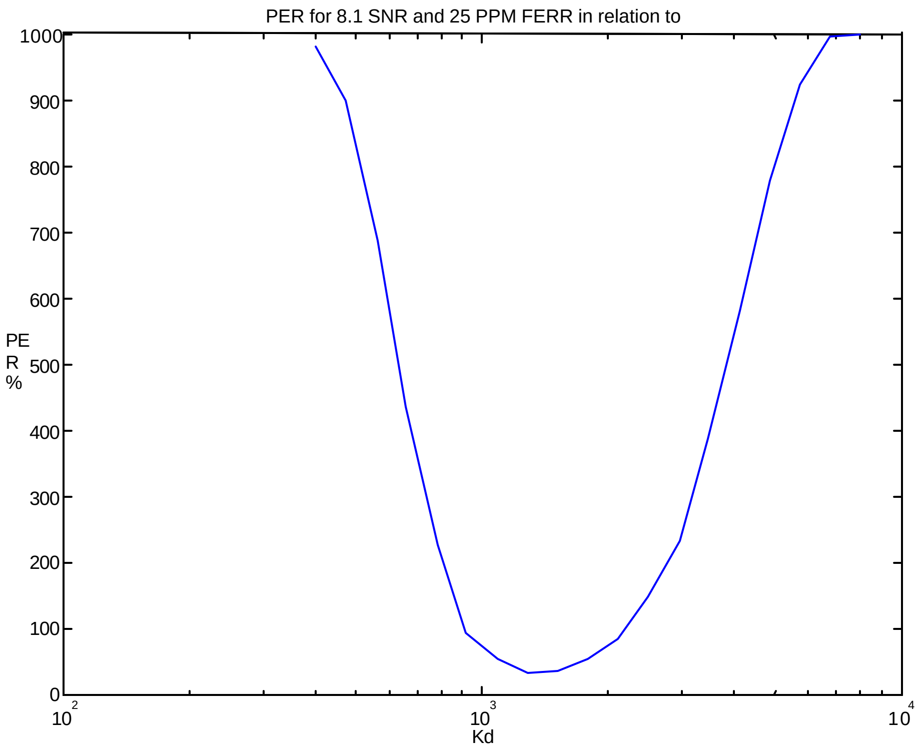

Figure 6-3 - PER vs Kd in 4 dB AWGN and 25 PPM FERR for 1KB Packet

The next step was to be an evaluation of the performance benefits from using a two step AFC gain progression: an high initial gain to quickly remove the error, and then a smaller gain to track the error more closely. Time series plots from the first tests, however, showed that the AFC and PLL were not interoperating well: each seemed to be fighting the other’s adjustments. Figure 6-4 shows the inverse relationship between the phase correction contributed by the PLL and that of the AFC. The PLL seems to be reacting to the corrections made by the AFC, countering any angle adjustment by the AFC with an inverse adjustment of its own.

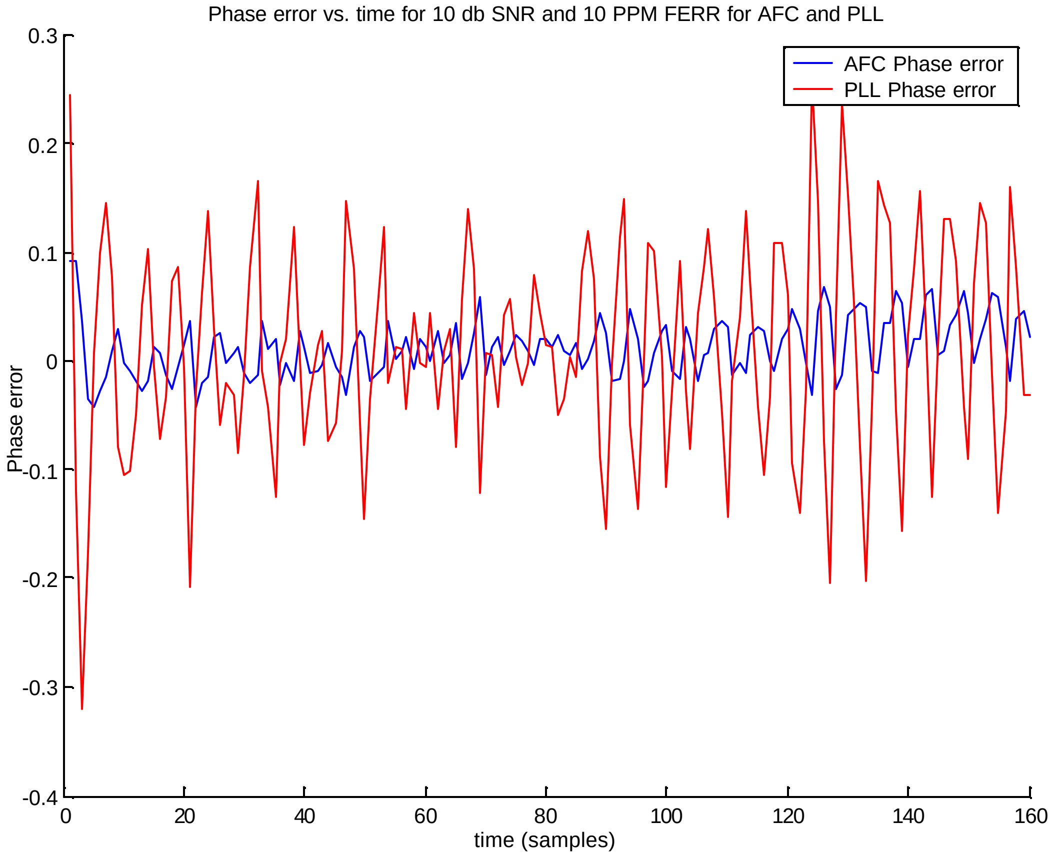

Figure 6-4 - Phase error vs. time in 10 dB SNR for AFC and PLL

6.3.2 Second order PLL

After the problems that were revealed using a separate AFC and PLL, the decision was made to examine the behavior of a second order phase locked loop. Another option would have been to look at the performance of the AFC and first order PLL operating separately in time. Hardware considerations, however, made the second order PLL a more attractive choice. Derivations of the continuous time and discrete time equations for a second order PLL are given in Section 5.

6.3.2.1 Single Loop Bandwidth Gain Step

The first step to determining the ideal time constant, prior to running a large scale simulation, was to determine some time constant values for which the receive simulation, dealing with 30 PPM frequency error, could at least decode the MPDU. As might have been expected, time constants in the single digits on the order of a microsecond seemed to fare best. A script was therefore set up to sweep T_C values over this range for a specific signal to noise ratio (see appendix full_chain_pll.m). The results, shown in figure 6-5 below, indicate a fairly steep decline on the lower range of scale and a somewhat more gradual incline on the higher range. This indicates that attacking the frequency error too quickly results in too much instability, while attacking too slowly is given a little more leeway, although not by much. The best operating range is from around 5us to around 10us. Operating in this range gives a little bit of leeway in the operation of the AGC, on which the PLL discriminator is dependent. The graph shows that varying the signal to noise ratio, while expanding the working range of the time constant, does not change the optimal value.

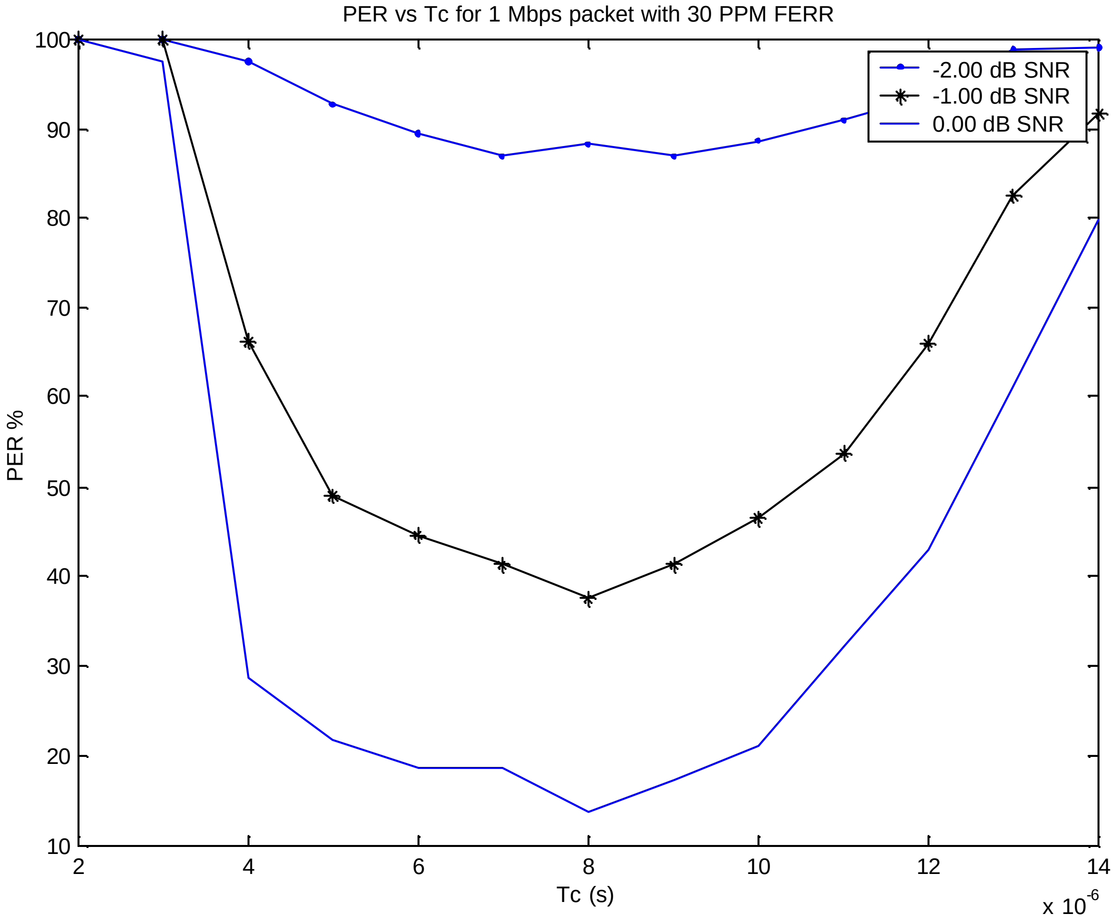

Figure 6-5 - PER vs. Tc in AWGN, 30 PPM FERR for 1 Mbps 1KB Packet

6.3.2.2 Multiple Loop Bandwidth Gain Steps

The next step to determining the optimum 2nd order PLL configuration was to determine how performance of the tracking loop is affected by using multiple gain steps. It was possible that using multiple gain steps would allow for a short time constant to attack the frequency error and then a longer time constant to slowly and accurately track the error over the course of the packet. More than two time constants could potentially be used to provide finer grained steps. The goal of using multiple gain steps was not only to achieve a better packet error rate, but additionally to achieve lock more quickly onto the frequency error so as to allow more time for the other synchronization tasks of the preamble such as channel matching.

Still using the first time constant as the variable, the proposed test was to proceed as follows: after having waited for three times the initial time constant, switch to a second tracking time constant. Given a standard exponential step response, after three time constants, around 95% of the frequency error should be removed. With an initial error of 30 PPM, this leaves about 1.5 PPM of frequency error remaining. A test similar to the one set up in the last section was run, only using 1.5 PPM of frequency error instead of 30 PPM. The results were somewhat unexpected, as lowering the frequency error does not seem to shift the time constant significantly, but rather simply increased the upper range of values at which the time constant could be set. Additionally, decreasing the frequency error from 30 PPM down to 1.5 PPM did not result in a dramatic decrease in packet error rate, regardless of the time constant that was used. Another batch of simulations was run, this time with a number of different frequency error ranges at one SNR. Figure 6-6 shows the results.

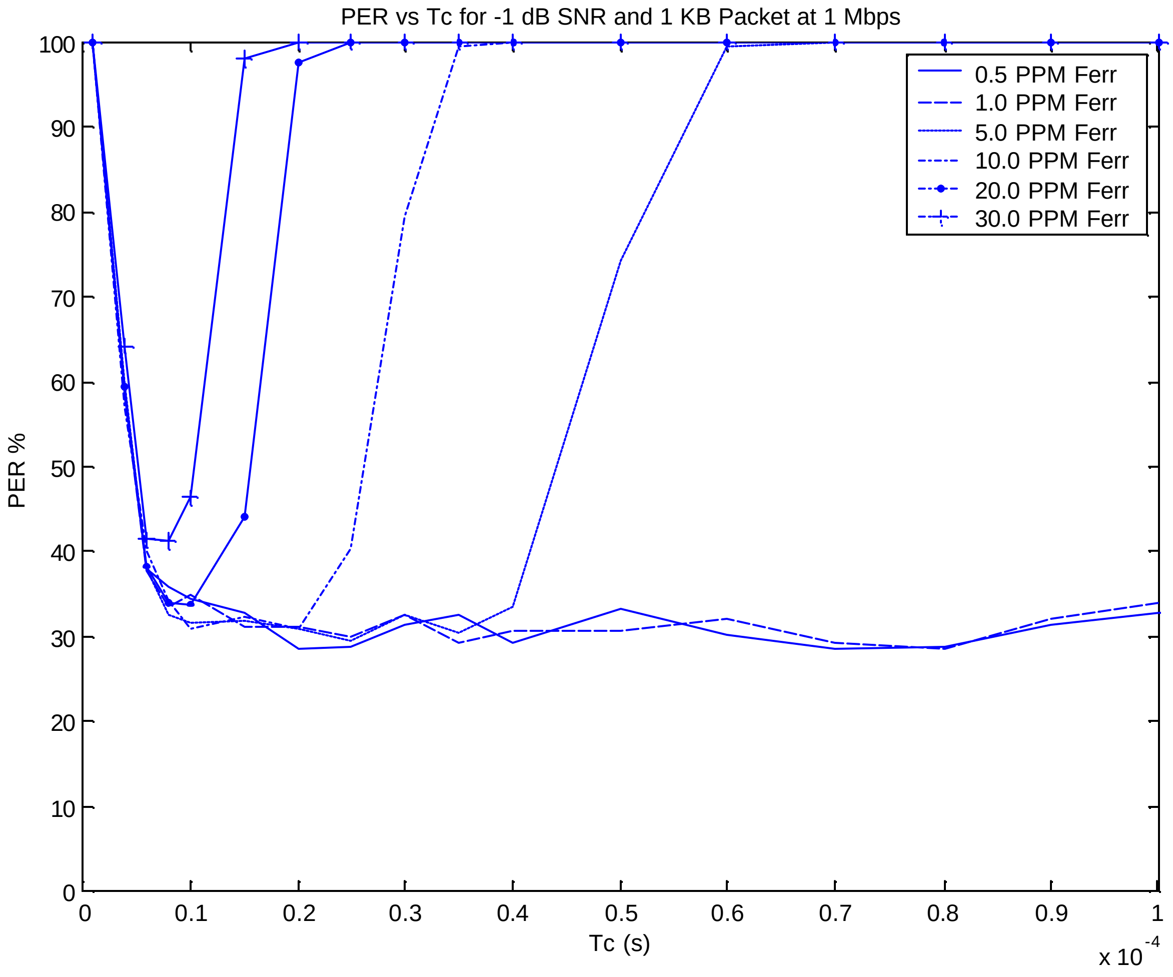

Figure 6-6 - PER vs. Tc for Ferr PPM range

One way to analyze these results is to point to the steep decline on the lower range of T_C as an indication of how little, past a certain point, packet errors are caused by decoding frequency fluctuations. This conclusion is aided by the maximum of 10% PER among the different levels of frequency error. These show that once a time constant is larger than a certain minimum, regardless of the SNR it does not lead to further packet errors. From figure 6-6, it can also be seen that while changing the frequency error affects the higher range of time constant values, the lower range in quite similar, with the plotted values sharing a near minimum value around 8us. The use of more than one gain step would therefore not aid in decoding the packet. Using more than one gain step does not decrease the amount of time needed to reach phase lock, as it seems that regardless of the amount of frequency error introduced, the ideal value for T_C remains the same.

6.4 Time Tracking Loop Optimizations

The 802.11 standard dictates that a standard compliant system must support up to 30 parts per million of code Doppler error. It turns out that 30 PPM of Code Doppler is much less difficult to track than the 30 PPM of Frequency Doppler. For this reason, the time tracking loop can be much simpler than the PLL, and only a 1^{st} order TTL is used.

6.4.1 TTL Description

The time tracking loop used in the receive simulation is a first order tracking loop with a maximum of 1/8 chip advance or retards. It takes the difference of the squares of the Barker matching of the previous and next 1/8 chips and multiplies this by a gain value. The result of this calculation is added to an error accumulator. If at any point this error accumulator reaches a set value, positive or negative, the error accumulator is reset and the tracker advanced or retarded 1/8 of a chip.

The slow time shift that code Doppler introduces over the course of a packet results in a gradual worsening of the decoded symbol to noise ratio. The effective signal loss in dB of this decay depends on the type of transmit and receive filters that are used. For bandlimited filters, the losses can be calculated as shown below [15].

$$loss(t) = 20\log_{10}\left(\frac{\sin(p\ {}^{t}\!/_{T_c})}{p\ {}^{t}\!/_{T_c}}\right) \qquad \text{(Eq. 6-1)}$$

The parameter t is the offset shift in time. The symbol T_c is the duration of one chip. The ratio ${}^{t}\!/_{T_c}$ therefore is the fractional shift over the course of one chip. Evaluating the equation with fractional shifts gives the results shown in Table 5-1.

Table 6-1- Chip offset signal loss

Fractional shift	dB signal loss
1/16	.06
1/8	.22
1/4	.91
1/2	3.92

A 30 PPM error results in approximately a five chip shift over the course of one packet. Without an active time tracking loop, an examination of the In-Phase symbol decoding of such a 1 Mbps packet would show the convergence of the symbols onto the mean less than a quarter of the way through the packet. This convergence is the result of the slow time shift and the accompanying signal strength loss. The reason the symbols do not spread back out is on account of the orthogonality properties of shifted Barker symbols. Since the receiver is match filtering entire Barker symbols, until the entire 11-chip symbol shifts through, no substantial spreading will result. If the code Doppler is increased to 90 PPM, this respreading could be seen.

6.4.2 1st Order TTL Optimization

Much like that for the PLL, the gain constant for the time tracking loop was optimized by sweeping it over a range of values at a set SNR. The results are again very dependent on the functioning of the AGC, because, as described above, the TTL uses the difference of two Barker-matched values to advance or retard the clock. The following simulations were run with a simulated AGC that scaled incoming data peaks to around 1. The gain value, Kdll, was used in correlation with an advance/retard threshold of ten. Any change in these values would make it necessary to rescale Kdll. The MATLAB function full_chain_ttl (See appendix: full_chain_ttl.m), was used to generate the results. Above the upper range of the graph, a switch is made every symbol, so evaluating Kdll above this range would result in PER values like the upper range.

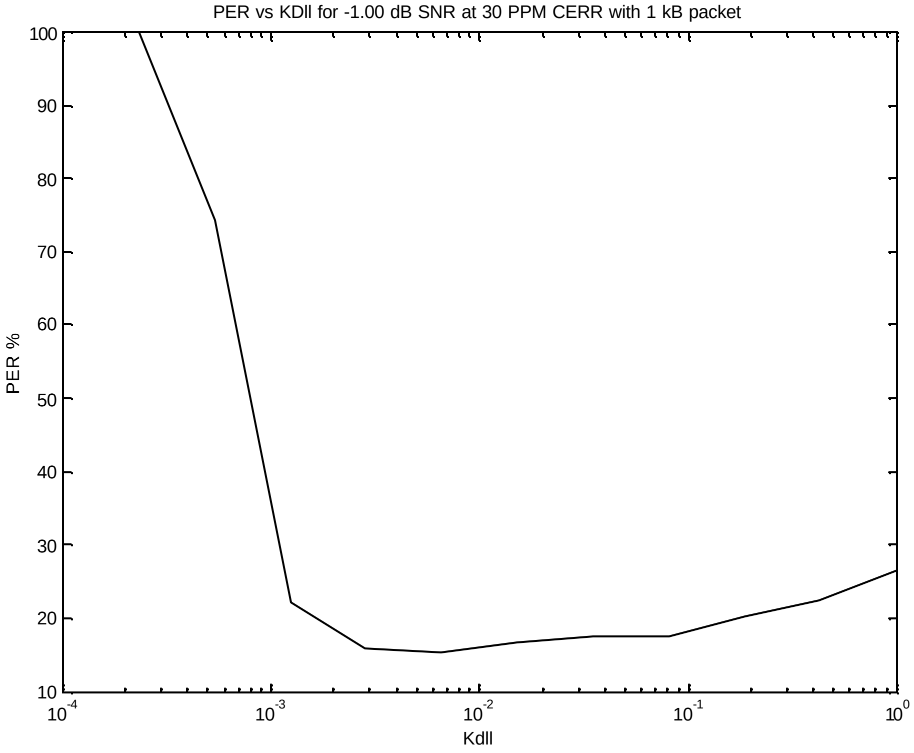

Figure 6-7 - PER vs. Kdll for -1.00 dB SNR

The minimum of the graph ends up around .006 with 25% packet error rate. Comparing the PER at this mark with the reference curve reveals a somewhat unexpected situation: The PER at that –1 dB SNR is almost 20% below that of the reference, which should be theoretically impossible.

6.4.3 Reference curve reevaluation

A reevaluation of the reference curve was necessary. The problems causing the PER discrepancy could be tracked to the peak detection algorithm used in the receiver. To perform peak detection the receiver only uses a correlation over the range of one Barker matched symbol to determine the peak offset. As a result, it is possible that at low signal to noise power levels, the real peak is obscured by the additive noise, and other peak-offset values have a higher correlation. As the original reference curve was run with the time tracking loop disabled, these peaks would not be corrected over time. This problem was exacerbated in the receiver because peak detection was only performed at a signal decimated to two times the chipping rate. Without the ability to shift within this range the detected peak could have been off by as much as ¼ of a chip, which would result in the loss of almost a full decibel of signal strength. The time tracking loop allows for 1/8 chip shifts. Figures 6-8 and 6-9 show where peak detection was at the end of the preamble over a sample size of 1000 packets. For Figure 6-8, time tracking was disabled, and for Figure 6-9 it was enabled.

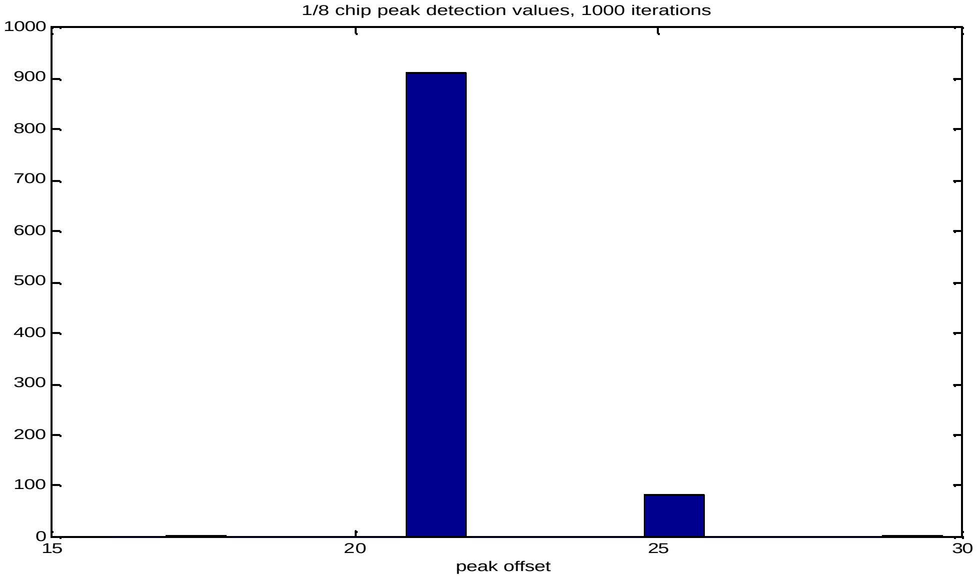

Figure 6-8 – Packet peak matches without TTL

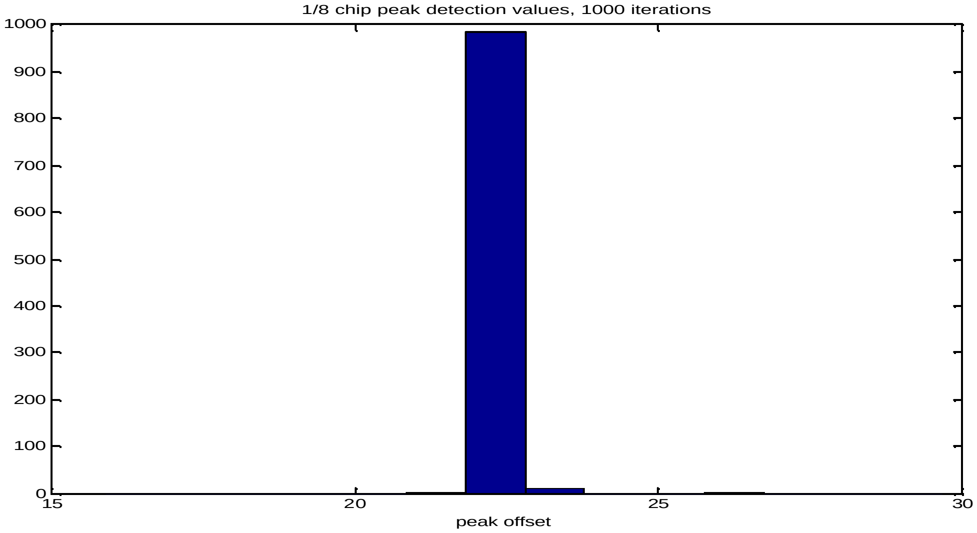

Figure 6-9 - Packet peak matches with TTL

Because the simulated packets always had the same ideal peak, a new reference curve was created with peak detection disabled and the peak offset set manually. This turned out to have the same result as when the TTL was in enabled with no code Doppler. The new reference graph now shows two curves: the original curve, and finally the ideal curve, with the peak offset set manually:

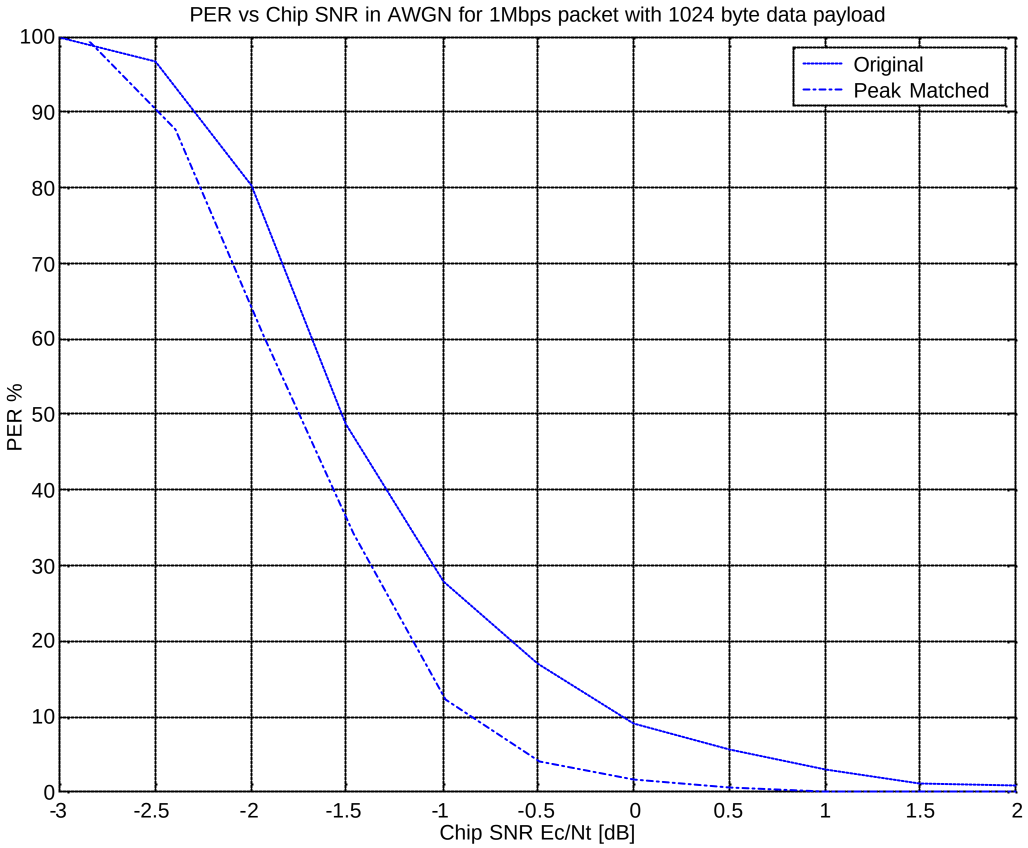

Figure 6-10 - New reference curve

6.4.4 Final TTL evaluation

Armed with the correct reference curve, it becomes clear that at only 30 PPM, the effect of the Code Doppler is essentially undetectable with the number of trials used in this simulation. A first order loop is here sufficient to provide the performance needed.

7 References

[1] ANSI/IEEE Std. 802.11, “Wireless LAN Medium Access Control and Physical Layer Specifications”, 1999.

[2] ANSI/IEEE Std. 802.11b, “Higher-Speed Physical Layer Extension in the 2.4 GHz Band”, 1999.

[3] R. Bagrodia, J. Short, L. Kleinrock, “Mobile wireless network system simulation”, Wireless Networks 1 (1995) 451-467.

[4] R. E. Best, “Phase-Locked Loops”, McGraw-Hill, Inc. 1984

[5] R. L. Bogush, “Digital Communications in Fading Channels: Tracking and Synchronization”, Mission Research Corporation, 1990.

[6] G.F. Franklin, A. Emani-Naeini, J. D. Powell, “Feedback Control of Dynamic Systems, Addison-Wesley”, Reading, MA, 1991

[7] G. Fleishman, “An 802.11 ISP on Maine's Rocky Coast”, O’Reilly Devcenter, 2000

[8] I.M. Jacobs, J. M. Wozercraft, “Principle of Communications Engineering”, John Wiley & Sons, New York, NY, 1965.

[9] T. Kaitz, “Channel and Interference Models for 802.16b”, BreezeCOM Ltd, 2000.

[10] K. Kaukonen and R.Thayer, “A Stream Cipher Encryption Algorithm ‘Arcfour’”, The Internet Society 1999.

[11] G. Nash, AN535, “Phase-Locked Loop Design Fundaments”, Motorola Inc., 1994

[12] B. O’Hara and A. Petrick, “IEEE 802.11 Handbook, A Designer’s Companion”, IEEE Press, New York, NY 1999.

[13] A.V. Oppenheim and R.W. Schafer, “Discrete-Time Signal Processing”, Prentice Hall, Englewood Cliffs, NJ, 1989.

[14] A. V. Oppenheim and A. S. Willsky, “Signals & Systems”, Prentice Hall, Upper Saddle River, NJ, 1997.

[15] A. J. Viterbi, CDMA “Principles of Spread Spectrum Communication”, Addison-Wesley, Reading, MA 1995.

[16] Arbaugh, Shankar, and Wan, “Your 802.11 Wireless Network has no Clothes”, University

of Maryland, 2001.

[17] S. Fluhrer, I. Mantin, and A. Shamir, “Weaknesses in the key scheduling algorithm of RC4.” Eight Annual Workshop on Selected Areas in Cryptography (August 2001).

A. Appendix – Transmit Class Interface

Table A-1 - PackedBits public interface

Function	Description

`void pack_msb(BYTE byte)`
`void pack_msb(WORD word)`
`void pack_msb(DWORD dword)`
	Three overloaded version of a packing function which packs 8, 16, or 32 bit variables into the end of the vector in most significant bit first order

`void pack_lsb(BYTE byte)`
`void pack_lsb(WORD word)`
`void pack_lsb(DWORD dword)`
	Three overloaded versions of a packing function which pack 8,16, or 32 bit variables into the end of the vector in least significant bit first order

`void packbits_msb(WORD word,int num)`	Same as pack_msb, but only packs the first *num* bits of *word*
`void packbits_lsb(WORD word,int num)`	Same as pack_lsb, but only packs the first *num* bits of *word*

Table A-2 - PMSignal public interface

Function	Description
`PMSignal(sigtype const *a,int n)`	Constructor that creates *PMSignal* from a normal C array
`PMSignal(int n,sigtype c)`	Constructor that creates a *n* length *PMSignal* with each element initialized to *c*
`PMSignal(PMSignal &inSig,int i1, int i2,int inc)`	Constructor that creates a *PMSignal* using every *inc* element, in the range from *i1* to *i2*, from and input *PMSignal*
`Void replace(int i1,int i2, PMSignal &repSig)`	Replaces the elements in the range of *i1* to *i2* of the *PMSignal* with the entirety of another *PMSignal*
`operator+=(const PMSignal& s)`	Appends *s* onto the current *PMSignal*
`void add(const PMSignal& s)`	Performs a vector addition using the elements of *s*

Table A-3 DataPacket public interface

Stage 1 Functions	**Description**
`DataPacket(BYTE signal,bool sync)`	Constructor, create packet of signal and sync type
`void Include(int includeMask)`	Includes a specific field
`void MakeFragment(int fragnum,bool lastFrag)`	Makes the frame a fragment
`void ResetIncluded()`	Resets the included fields to none
`void SetAdr(BYTE *adr,int num)`	Sets address *num* to the value of the byte array *Adr*
`void SetDuration(int duration)`	Sets the duration field
`void SetOrder(bool order)`	Sets the order frame control bit
`void SetPwrMgmt(bool pwr)`	Sets the power management frame control bit
`void SetRetry(bool retry)`	Sets the Retry Frame control bit
`void SetSequence(int seq)`	Sets the sequence number field
`void SetSubtype(BYTE subtype)`	Sets the frame subtype
`void SetToFromDS(bool toDS,bool fromDS)`	Sets the To and From DS frame control bits
`void SetType(BYTE type`	Sets the frame type
`void WEP(bool enable)`	Enables WEP Encryption
`void WEPIV(vector<BYTE> iv)`	Sets the WEP IV
`void WEPKey(vector<BYTE> key`	Sets the WEP secret key
`void WEPKeyID(char keyID)`	Sets the WEP keyed
Stage 2 Functions	**Description**
`bool CreateMPDU(DataSim &data)`	Creates the MPDU from the set options and *data*
`bool CreatePLCP()`	Creates the PLCP preamble and header
`bool Scramble()`	Scrambles all the bits in the packet
Stage 3 Functions	**Description**
`Const PackedBits &preamble()`	Returns the PLCP preamble
`Const PackedBits &header()`	Returns the PLCP header
`Const PackedBits &mpdu()`	Returns the MPDU

Table A-4 CPMSignal public interface

Function	Description
`PMSignal I,Q`	Public data members that allow access to inphase and quadrature portions of complex signal.
`CPMSignal(PMSignal pI,PMSignal pQ)`	Constructor that creates a *CPMSignal* from two *PMSignal*'s
`Double Energy()`	Calculates and returns the energy of the *CPMSignal*
`void Clear()`	Clears the both the inphase and quadrature portions of the signal
`operator+=(const CPMSignal& s)`	Appends *s* onto the current C*PMSignal*

Table A-5 - TransmitPMD public interface

Function	Description
`TransmitPMD(BYTE signal,bool sync)`	Initializes the class with signal and sync
`void signalEncode(const PackedBits packet,PMSignal &output)`	Encodes packet into a vector of 1's and –1
`void bitEncode(const PackedBits packet,PMSignal &output)`	Encodes packet into a vector of 1's and 0's, Used for saving the bit information
`void setEncoder(int s0,int s1)`	Sets the initial state of the encoder
`void sendPreamble(const PackedBits &preamble, CPMSignal &sig)`	Encodes the preamble at 1 Mbps
`void sendHeader(const PackedBits &header,CPMSignal &sig)`	Encodes the header at 1 or 2 Mbps
`void sendMPDU(const PackedBits &mpdu,CPMSignal &sig)`	Encodes the MPDU at appropriate rate

Table A-6 - DoppDiscreteFilter public interface

Function	Description

`DoppDiscreteFilter(`
`T &filter,int upsample,int taps,`
`DynamicValue &offset_delta)`
	Initializes the class with an oversampled filter vector, an upsampling multiplier, the number of actual taps, and a offset.

`bool filter(T &inVec,T &outVec)`	Filters an incoming signal and outputs a new vector

Table A-7 - PhyChannel public interface

Function	Description
`PhysicalChannel(const char *filename)`	Constructor that loads a configuration file
`void createChannel()`	Creates the channel filter according to configuration
`void loadCFG(const char *filename)`	Loads a configuration file
`void addFreqDoppler(CPMSignal &sig)`	Adds frequency Doppler according to configuration
`void addChannelEffects(CPMSignal &sig)`	Filters the incoming signal through the channel filter
`void addAWGN(CPMSignal &sig)`	Adds AWGN to the signal according to configuration
`void addScale(CPMSignal &sig)`	Multiplies the signal by a scaling factor
`void addAll(CPMSignal &sig)`	Adds all of the above effects to the signal

Table A-8 - SimCDiscreteFilter public interface

Function	Description
`SimCDiscreteFilter(T &filter_r)`	Initializes the class with a filter vector
`filter(T &inVecR,T &inVecI,T &outVecR,T &outVecI)`	Filters an incoming complex signal and outputs a set of new I and Q vectors

Table A-9 - FileToken public interface

Function	Description
`void Open(string filename)`	Opens file filename for reading.
`void Close()`	Closes the current open file.
`bool IsOpen()`	Returns whether there is currently a file open.
`String NextToken()`	Returns the next token in string form.
`String StringToken()`	Returns the next string token.
`Int NextValue()`	Returns the next byte sized token as an integer value, if it cannot be converted, throws an exception.
`string LastToken()`	Returns the last token successfully read.
`Int LastValue()`	Returns the last byte value token.
`bool AddNextTokenValues(vector<BYTE> &vec)`	Adds the last token (up to 32-bits) to vector *vec*
`Int LastTokenValue(int *num)`	Returns the size in bits of the last token read, and sets *num* to the value of that token.
`void EatToken(char *str)`	Eats the next token if it is *str*, otherwise throws an exception.
`string PositionString();`	Returns a string of the that indicates the position in the file of the last token read
`Bool IsDone()`	Returns if the file is done

B. Appendix – Links to Code

A	

C. Appendix – PLL Loop filter analysis

Phase Step with 1st order Loop filter:

$$F_e(s) = \frac{A_0 K_1 q_c s}{s^2 + A_0 K_1 K_2 s}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 q_c s}{s^2 + A_0 K_1 K_2 s}$$

$$= \lim_{s \to 0} \frac{A_0 K_1 q_c s^2}{A_0 K_1 K_2 s} = 0$$

Velocity Step with 1st order Loop filter:

$$F_e(s) = \frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2}$$

$$= \frac{A_0 K_1 u_c}{A_0 K_1 K_2} = \frac{u_c}{K_2}$$

Acceleration Step with 1st order Loop filter:

$$F_e(s) = \frac{A_0 K_1 2a_c s}{s^4 + A_0 K_1 K_2 s^3}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 2a_c s}{s^4 + A_0 K_1 K_2 s^3}$$

$$= \lim_{s \to 0} \frac{2A_0 K_1 a_c s^2}{A_0 K_1 K_2 s^3} = \infty$$

Phase Step with 2nd order Loop filter:

$$F_e(s) = \frac{A_0 K_1 q_c s}{s^2 + A_0 K_1 K_2 s + A_0 K_1 K_2 a}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 q_c s}{s^2 + A_0 K_1 K_2 s + A_0 K_1 K_2 a}$$

$$= \lim_{s \to 0} \frac{A_0 K_1 q_c s^2}{A_0 K_1 K_2 a} = 0$$

Velocity Step with 2nd order Loop filter:

$$F_e(s) = \frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2 + A_0 K_1 K_2 as}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2 + A_0 K_1 K_2 as}$$

$$= \lim_{s \to 0} \frac{A_0 K_1 u_c s^2}{A_0 K_1 K_2 as} = 0$$

Acceleration Step with 2nd order Loop filter:

$$F_e(s) = \frac{A_0 K_1 a_c 2s}{s^4 + A_0 K_1 K_2 s^3 + A_0 K_1 K_2 as^2}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{2A_0 K_1 a_c s}{s^4 + A_0 K_1 K_2 s^3 + A_0 K_1 K_2 as^2}$$

$$= \frac{2A_0 K_1 a_c}{A_0 K_1 K_2 a} = \frac{2a_c}{K_2 a}$$

Phase Step with 3rd order Loop filter:

$$F_e(s) = \frac{A_0 K_1 q_c s}{s^2 + A_0 K_1 K_2 s + A_0 K_1 K_2 a + \frac{A_0 K_1 K_2 b}{s}}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 q_c s}{s^2 + A_0 K_1 K_2 s + A_0 K_1 K_2 a + \frac{A_0 K_1 K_2 b}{s}}$$

$$= \lim_{s \to 0} \frac{A_0 K_1 q_c s^2}{\frac{A_0 K_1 K_2 b}{s}} = \lim_{s \to 0} \frac{A_0 K_1 q_c s^3}{A_0 K_1 K_2 b} = 0$$

Velocity Step with 2nd order Loop filter:

$$F_e(s) = \frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2 + A_0 K_1 K_2 as + A_0 K_1 K_2 b}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 u_c s}{s^3 + A_0 K_1 K_2 s^2 + A_0 K_1 K_2 as + A_0 K_1 K_2 b}$$

$$= \lim_{s \to 0} \frac{A_0 K_1 u_c s^2}{A_0 K_1 K_2 b} = 0$$

Acceleration Step with 2nd order Loop filter:

$$F_e(s) = \frac{A_0 K_1 a_c 2s}{s^3 + A_0 K_1 K_2 s^2 + A_0 K_1 K_2 as + A_0 K_1 K_2 b}$$

$$\lim_{t \to \infty} F_e(t) = \lim_{s \to 0} s \frac{A_0 K_1 a_c 2s}{s^4 + A_0 K_1 K_2 s^3 + A_0 K_1 K_2 as^2 + A_0 K_1 K_2 bs}$$

$$= \lim_{s \to 0} \frac{2 A_0 K_1 a_c s^2}{A_0 K_1 K_2 bs} = 0$$

D. Appendix – Sample Configuration File