Complete Communication System Modeling

Pranavi Boyalakuntla
Jonathan Kelley

I. Abstract

In this paper, we leveraged introductory-level skills in analog-digital communication (ADC) to design an end-to-end communication system in Simulink. The Simulink model of the system was used to then select appropriate physical components and understand a potential hardware implementation. The ADC system model was designed to use Binary Phase Shift Keying (BPSK) as the modulation scheme and optimized to fit within a handful of real-world constraints including power limitations, transmission rate, error rate, and availability of components for the hardware implementation. The final result of the Simulink model demonstrated the capability for a transmitter and receiver to send digital messages to one another utilizing a custom packet encoding scheme.

This project served as an excellent demonstration of introductory ADC skills and helped better inform and relate these skills to real-world system design.

> 💡 All code and schematic layup is available at https://github.com/jkelleyrtp/adc-bpsk-simulation

II. Introduction

Analog-to-digital and digital-to-analog communication techniques are the foundation of modern communication systems. Nearly every device used in today's society uses digital, programmable logic to control analog components that interact with real-world physics and limitations. These systems are ubiquitous - used everywhere from the networking the internet to connecting cell phones and enabling GPS.

Ultimately, the goal of such a system is to transmit a payload of data from one geographic zone to another. From there, system designers optimize for data rates, power requirements, security, and a whole host other relevant factors. For some systems, smoke signals might be adequate despite their limited range and limited bandwidth. For other systems, fiber optic cables and dedicated integrated circuits might be required to achieve system requirements.

For this project, we will be exploring a communication system designed to transmit digital payloads using radio frequency electromagnetic waves. As an academic exercise, the system will use binary phase shift keying (BPSK) as the modulation scheme, however we plan to implement a model flexible enough handle other frequency-shift-keying schemes. The ultimate goal for the system will be to transmit binary data between a receiver and transmitter, effectively laying the groundwork for a text-messaging system.

In order to properly bound the system designer parameters, we will be operating in the WiFi spectrum and roughly adhering to FCC regulations for WiFi. This means our carrier frequency will be 2.4 GHz. According to part 15 of the FCC rules [1],

> "Maximum Effective Isotropic Radiated Power (EIRP) is 36 dBm (4 watt)"

The FCC also provides the table shown in Figure 1:

Maximum Power from Intentional Radiator *1	Maximum Antenna Gain (dBi)	EIRP (dBm)	EIRP (watts)
30dBm or 1 watt	6	36	4
27dBm or 500mW	9	36	4
24dBm or 250mW	12	36	4
21dBm or 125mW	15	36	4
18dBm or 63mW	18	36	4
15dBm or 32mW	21	36	4
12dBm or 16mW	24	36	4

Figure 1. FCC Regulation Table.

It should be noted that these regulations inherently limit physical aspects of the system rather than the mathematical and theoretical inputs. Ultimately, our transmitter system will be limited by the power regulations and realistic values for WiFi antennas we can find available online. It should also be noted that system in this paper is relatively short-range with an emphasis on higher bandwidths, perhaps for sending large novels or watching Netflix.

III. System Design

A. Overview

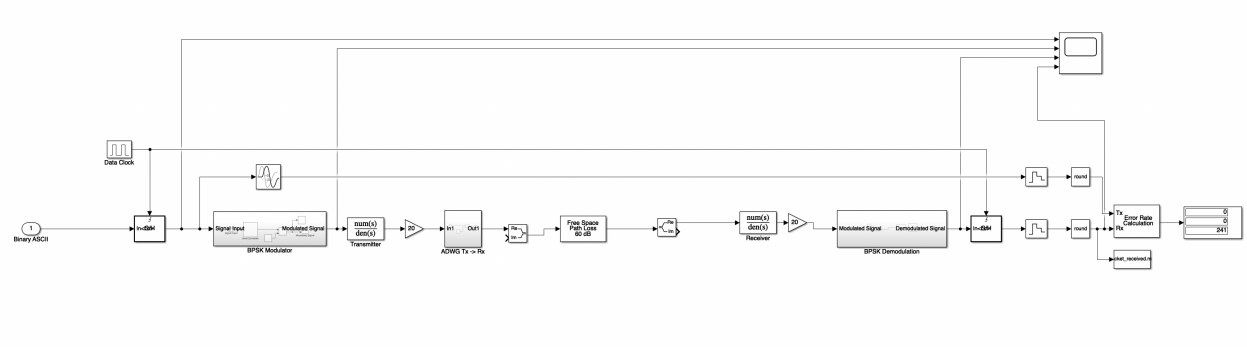

Figure 2. Overall simulink model

At a very high level, our binary transmission system is comprised of a handful of fundamental components:

- **Data source**: the message to be sent from the transmitter to the receiver
- **Encoder**: a way of converting the data buffer into a binary signal with error handling
- **Modulator**: component that takes in the data signal and converts it into a signal to be transmitted
- **Transmitter:** component that takes a signal and converts it to a radio frequency (RF) output
- **Environment:** the space RF signals travel through on the way to the receiver
- **Receiver:** an antenna and appropriate hardware to convert RF signals into an analog system
- **Demodulator:** converts an analog signal from the receiver into a digital signal
- **Decoder:** decodes the digital signal from the demodulator
- **Data sink:** the microcontroller or host where the decoded binary data gets sent

Each of these components plays an important role in the entire communication system with their own independent limitations. For example, the datasource and encoder need to be fast enough to generate the data to feed the transmission rate. The modulator needs to be accurate and stable enough not to add unnecessary distortion in the transmitted signal. The transmitter is power limited, both in input power and RF energy and EIRP. The environment also imposes significant challenges depending on

transmission distance, weather, and even solar activity. Ultimately, each component must be able to operate without burning up or drawing infinite amounts of power.

B. Data source

For any communication system, a user *must* supply a payload message. In our communication system, we consider this data to be loaded into an integrated circuit's memory. This could either be a register on a CPU, or a dedicated buffer of latches controlled by a microcontroller. For our data source in the Simulink model, we will construct a "Time Table" object which simulates a microcontroller changing the output of a data pin from 0 to 1 at given points in the simulation. This could be as simple as microcontroller like an STM32 turning off and on a GPIO pin or as complicated as an FPGA loading a message into RAM. In contrast, our data source is simply a list of 1's and 0's indexed by time. This configuration enables us to pre-encode the message before loading it into Simulink, vastly simplifying encoder logic.

We can then populate our datasource by encoding a list of ASCII characters into their respective binary representation, padding each character to one byte of memory. To send 4 characters, this would require 32 bits (4 bytes) of space in the buffer.

```
A 01000001
D 01000100
C 01000011
! 00100001
```

For the hardware design, we will simply default to the host microcontroller's flash memory as the datasource and encoder. This fundamentally limits the bit rate to that of the microcontroller's speed to set a data output pin HIGH/LOW, but will enable flexibility in the encoding phase.

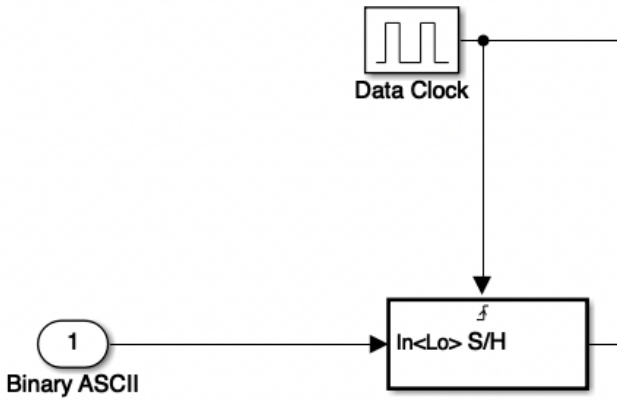

Figure 3. Implementation of sourcing data in Simulink.

C. Encoder

The encoder is one of the most interesting aspects of the communication system. Here, the payload data is converted into a format that adheres to the MAC layer specifications. For the decoder to be able to identify, synchronize, and read the payload data, we need to design a robust encoder.

First, the encoder needs to provide some signal to a decoder that a packet of data is being sent. This will enable listeners to prepare for the data, either by waking up or clearing existing buffers of data. This portion of the final payload is known as the "header." As such, we'll take inspiration from the ethernet specification and choose `10101010101010101010` as the preamble of this header.

> An Ethernet packet begins with the Ethernet preamble, 56 bits of alternating 1 and 0 bits, allowing the receiver to synchronize its clock to the transmitter, followed by a one-octet start frame delimiter byte and then the header.

In reality, the design of a MAC 802.11 packet is much more complicated with receiver address, transmitter address and destination address.

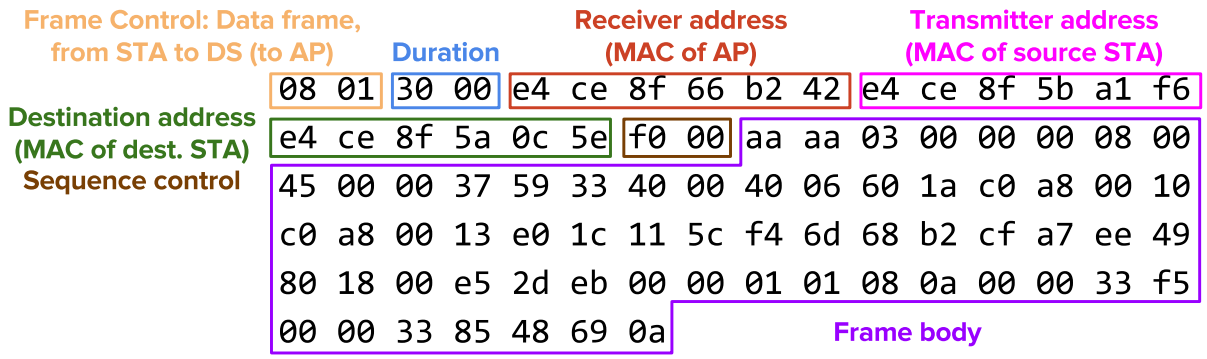

Figure 4. MAC 802.11 packet structure

For our simple simulation, we'll stick with the preamble, frame control `11001100`, duration, and body. The frame control informs listeners about the layout of the rest of the header and body. The 802.11 header also includes extra information in the frame body, but we'll chose to ignore that for our model. This leaves us with a packet layout as shown in Figure 5.

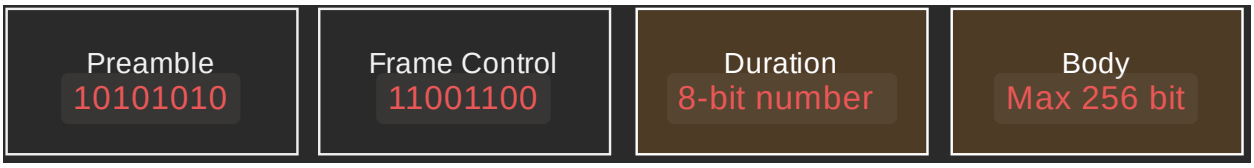

Figure 5. Custom packet construction protocol for this simulation.

The body component of the packet is encoded via Hamming(7,4) code error correction, increasing our redundancy in exchange for data throughput. For a system as short-range as ours, this might be overkill, but the high frequency of the 2.4 GHz carrier means we can send a high volume of data through a variety of indoor obstructions where error correction would be useful.

Finally, the encoded packet data is sampled via the data clock with period T_b. This is held in a 1-bit register via a zero-hold latch so the modulator can sample at period T_s.

D. Modulation

The modulator is a crucial component of the communication system. The modulator enables us to convert a digital signal from the encoder, convert it into an analog signal, mix it with a carrier frequency, and drive the transmitter. For this system, we chose to implement Binary Phase Shift Keying (BPSK) due to its widespread use, interesting

dynamics, and robustness in indoor settings. In fact, BPSK is so well suited for low-range, high throughput communication systems that it's commonly found in wireless routers, bluetooth, and RFID.

The theory of BPSK is simple: a signal can be offset in one of two directions with each direction corresponding to either a 1 or a 0. Visually, this would translate into a signal as shown in Figure 6, where the data line is Signal A and the modulated line is Signal B.

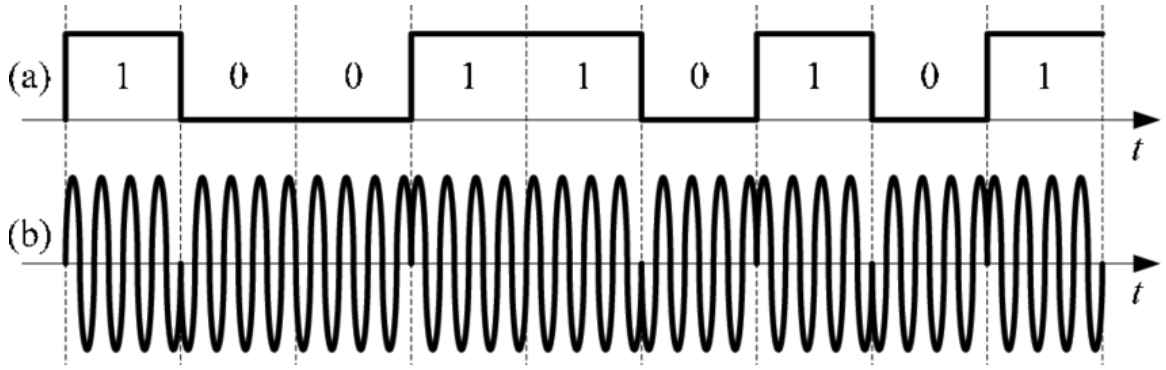

Figure 6. BPSK example.

To convert the digital signal into a phase offset, we can use a simple setup called a *level converter* which maps the signal like so:

```
Digital    Phase
Input   |  Offset
--------------------
1       |  1
0       | -1
```

In Simulink, we can implement this as a simple equation

$$y[k] = 2k - 1 \tag{1}$$

From there, we can multiply this signal by the carrier 2.4 GHz wave sampled at frequency F_s. Because multiple oscillations of F_s must fit within a single T_b , F_s will need to be a significantly higher frequency than F_b. Recall that T_b and T_s are the bit period and bit frequency, respectively. Using the WiFi 802.11an protocol as a base, we

can expect to achieve 11 Mbps with 2.4 GHz and BPSK. Therefore, T_b would be about 100 ns, so we would need to sample Fs with a period of around 10ns to achieve 10 samples per bit period. The resolution here is not terribly important for the simulated design, but does play an important role when noise might distort the modulated signal. The BPSK implementation is shown in Figure 7.

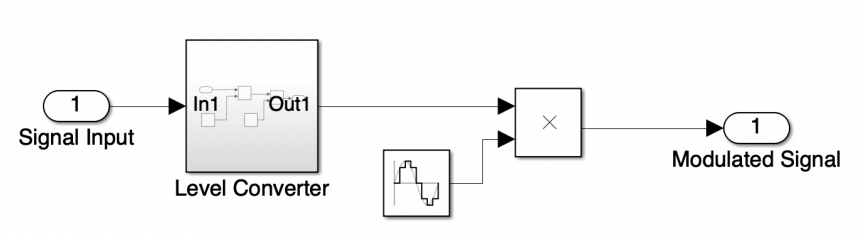

Figure 7. Modulator in Simulink

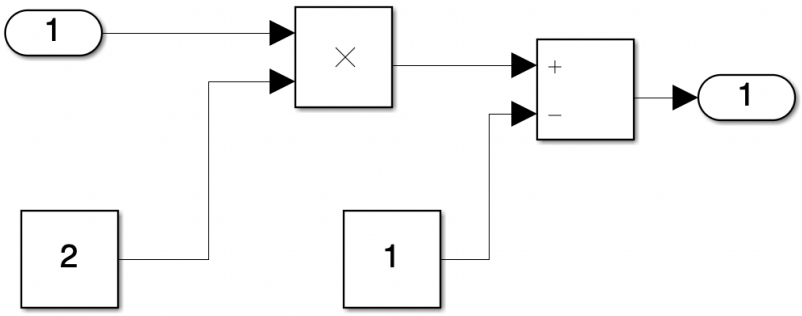

Figure 8. Level Converter in Simulink

E. Transceiver

Now that that data signals have been retrieved from the source, modulated, and encoded, we need to transmit the payload. The most basic design for a transmitter/receiver pair is shown in Figure 9. This transmitter and receiver pair serve as the foundational components for RF communication.

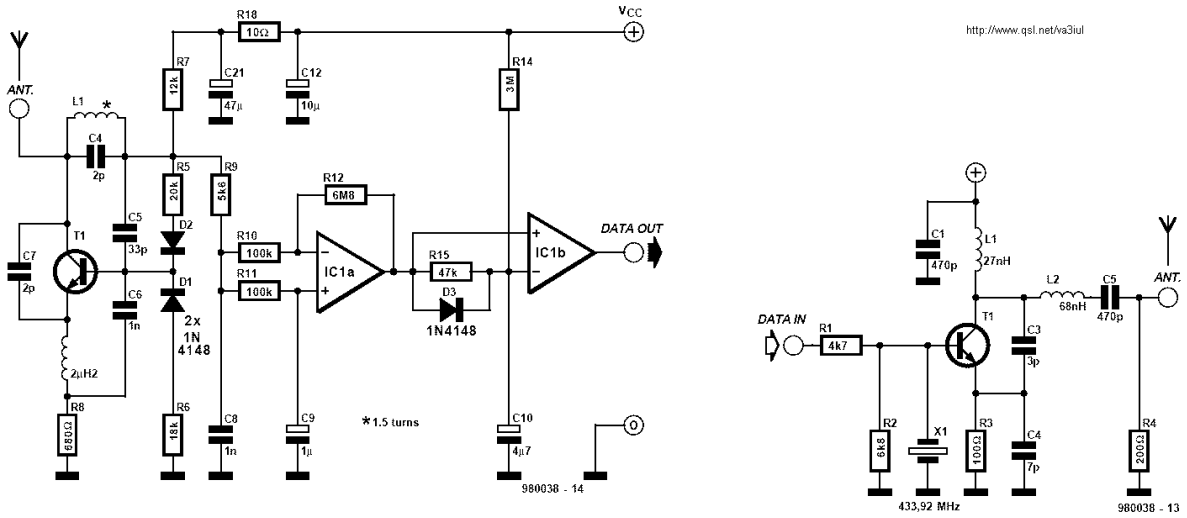

Figure 9. Most basic transmitter/receiver (QSL.net)

To include the transmitter characteristics in our Simulink model, we'll simply abstract the behavior as a low-pass filter. While this might not be entirely accurate, it does take into account the strong RLC (resistor-inductor-capacitor) nature of the design and should be a suitable approximation.

Figure 10. Transmitter Transfer Function in Simulink

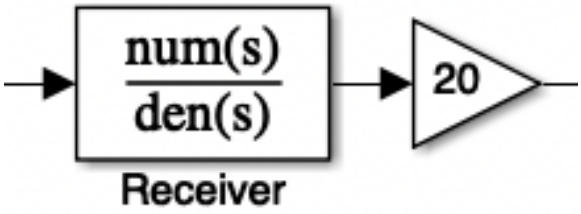

Figure 11. Receiver Transfer Function in Simulink

For specific model parameters, we can select an off-the-shelf 2.4 GHz transceiver an antenna: Nordic Semiconductor's nRF24L01 combined with Sparkfun's 2.4 GHz Reverse Polarized Large Duck Antenna. We will use the antenna's 5dBi gain and 50 ohm impedance as inputs to the simulation.

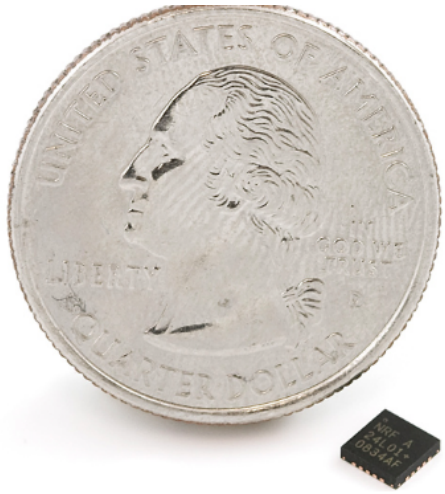

Figure 12. nRF24L01

Figure 13. 2.4 GHz Duck Antenna

Conveniently, Sparkfun also provides instructions on how to design the system schematic which we are able to replicate with little difficulty.

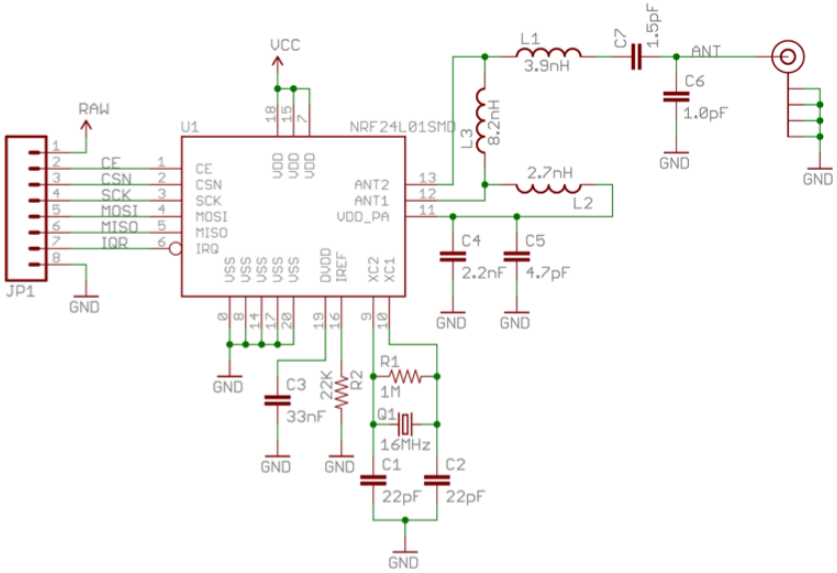

Figure 14. Transceiver schematic

F. Environment

We approximated the environment as general noise with power loss. Simulink provides many various options for simulating power loss, interference, and noise. For the system in question, we'll use the *mean free space power loss* block in conjunction with a

Gaussian white noise block to obtain a rough estimate of how transmission through an indoor environment might attenuate the payload.

Our free space power loss component follows an attenuation behavior shown in Eqn. 2.

$$FSPL = 20\log_{10}(d) + 20\log_{10}(f) + 20\log_{10}\left(\frac{4\pi}{c}\right) - G_t - G_r \qquad (2)$$

where

```
Gt: transmitter gain
Gr: receiver gain
d: distance
f: carrier frequency
```

At a distance of 10 meters, a carrier frequency of 2.4 GHz, a transmitter gain of 20dB, and a receiver gain of 20dB, our FSPL is therefore around 20 dB.

For the Gaussian white noise, we chose a power spectral density of 2e-20 Watts/Hz as an estimation of noise inside an average indoor environment. The noise and attenuation components of the Simulink model are shown in Figure 15.

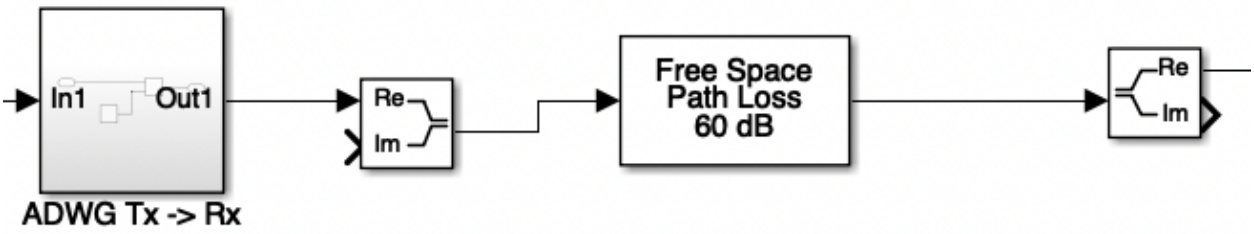

Figure 15. Environment model in Simulink

G. Demodulation

Now that the signal has been received from the receiver, it needs to be demodulated before any meaningful data before the payload can be decoded and displayed. First, we'll multiply it with a cosine function, re-centering the signal and then low-pass filtering to remove any ghosting artifacts not centered at 2.4 GHz. From there, we pass the signal into the "accumulator": the component instrumental in converting the phase-shifted receiver signals back into the digital domain. For this, we use a variation

of the "integrate and dump" approach via a windowed integrator. This component integrates the receiver signal for one bit-period at a time. If the received signal has a positive phase offset, then the integral of the signal will be positive, and if the signal has a negative phase offset, the integral of the signal will be negative.

The output of this integral is then fed into an "un-level" shifter which converts the +1/-1 phase offsets into a 1/0 digital signal. Once we have the digital signal, we can use a zero-order-hold synchronized with the bit period to sample the digital signal.

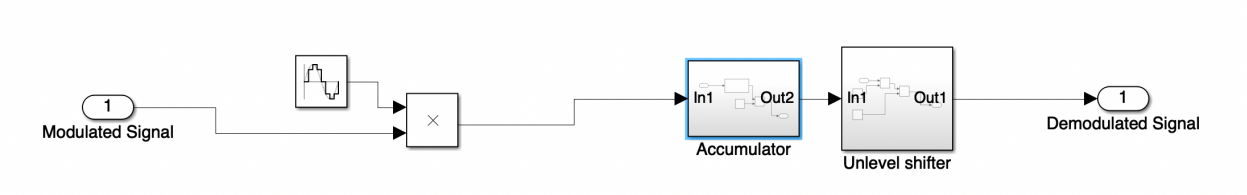

Figure 16. Demodulator system in Simulink

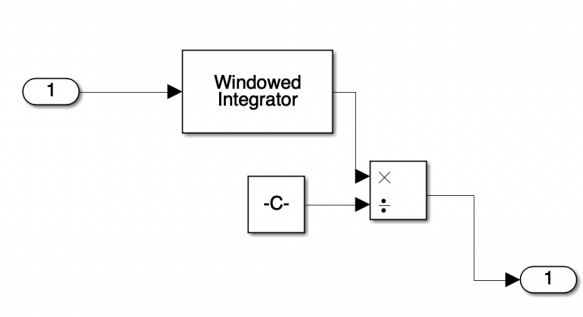

Figure 17. Normalized windowed integrator in Simulink

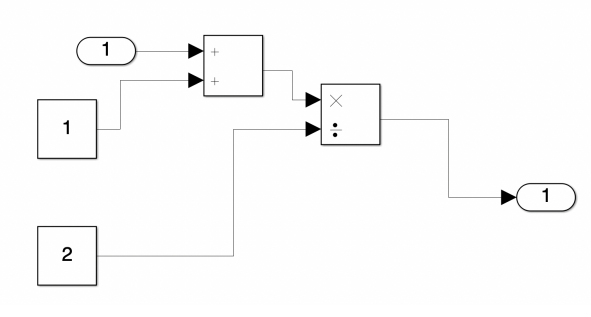

Figure 18. Un-level-shifter in Simulink

H. Decoder

The decoder is the final step in the communication system before the payload is passed on to a data sink. While we won't model a data sink in this project, the output of the decoder should still be suitable for transferring data into a sink like a microcontroller or ethernet-connected local area network. We'll implement a simple decoder that takes the demodulated signal and interprets that signal as data using Simulink callbacks. In our model, the decoder watches for the packet preamble. Once the preamble is found, a delayed callback is triggered, ending the simulation after the packet is completely

received. The decoding is done via a custom Matlab function whose contents is shown in Appendix X.

```
data = deconstructPacket(packet_received);
decoded = hammingDecode(data);
textMessage = get_text(decoded)
```

From there, we break apart the packet into the header, duration, and body, stripping any padding to accommodate for the (4,7) Hamming Code. Next, we finally decode the hamming-encoded duration and body contents.

```
decoded = [];
for i = 1:7:length(input_vector)
    decoding_portion = input_vector(i:i+6);
    decData = decode(decoding_portion,7,4,'hamming/binary');
    decoded = vertcat(decoded, decData');
end
```

Finally, the payload is passed onto the data sink where it is converted from binary to ASCII and the message is displayed. For the sake of testing, we will add an error rate calculator that compares the transmitted data to the received data. This works simply by comparing both signals and counting where the bits differ. However, this does not include error in messages after they've been Hamming decoded.

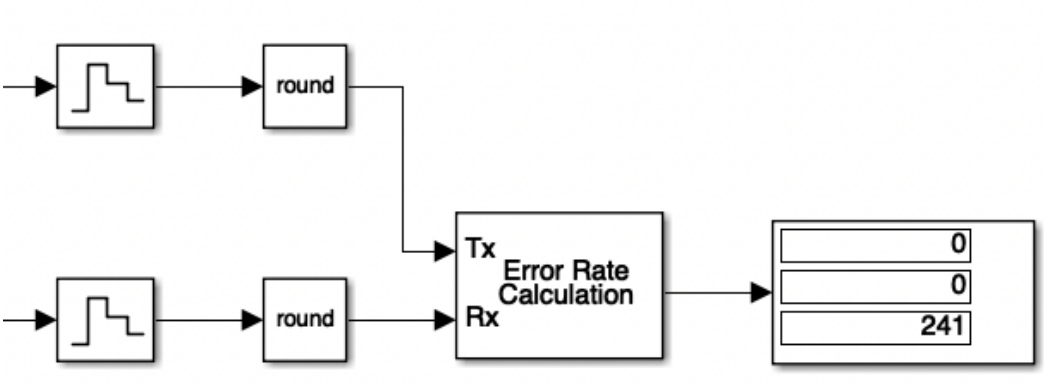

Figure 19: Error-rate calculator in Simulink

IV. Results

In order to verify the simulated model, let's simulate sending text messages across it as previously disussed. The phrase we will send is "Hello, World!".

We can set up the important variables of the simulation using A.1. Then, we can run the code in A.2 to construct the corresponding packet and convert it to a time table using the previously selected bit period.

Then, we can run this text through the Simulink model and view how the signal changes throughout. Let's focus in on the first few bit periods which will cover part of the preamble so that we can see how this model in action.

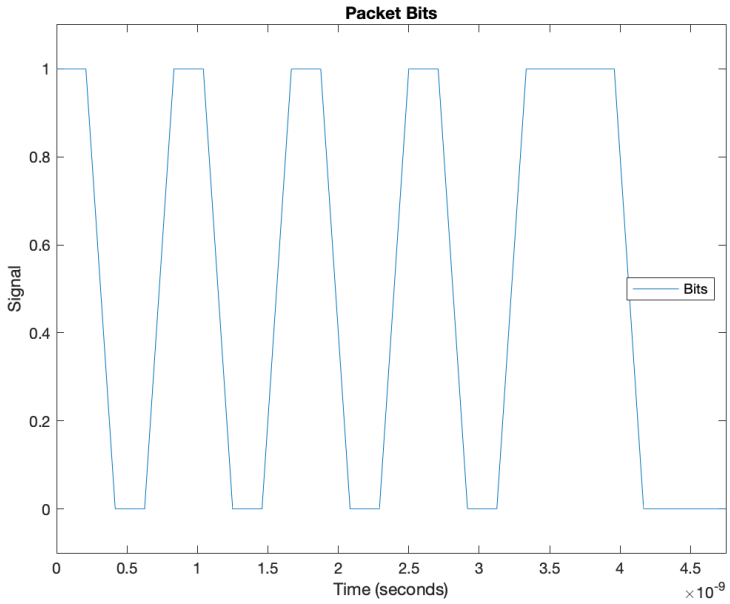

Figure 20. Bits of the packet turned into a signal

Using BPSK, we can modulate the signal. Looking at Figure 21, we can see the the BPSK modulation did in fact work. The repeated phase shifts accompany the "101010" of the bits with a bit of delay.

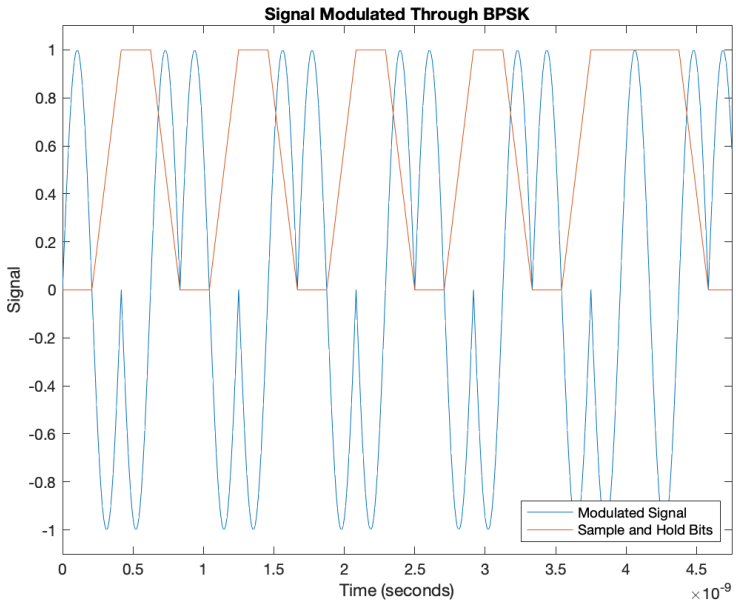

Figure 21. Sent bits compared against the BPSK modulated signal.

There is a gain component in the transmitter which is clearly seen in the comparison between the modulated signal and the transmitted signal in Figure 22. In addition, the receiver seems to have smoothed out some of the sharper turns in the BPSK. This is due to the frequency response of the receiver smoothing out frequencies above its frequency cutoff of 2.4GHz. Besides those difference, however, the phase shifts seems to align.

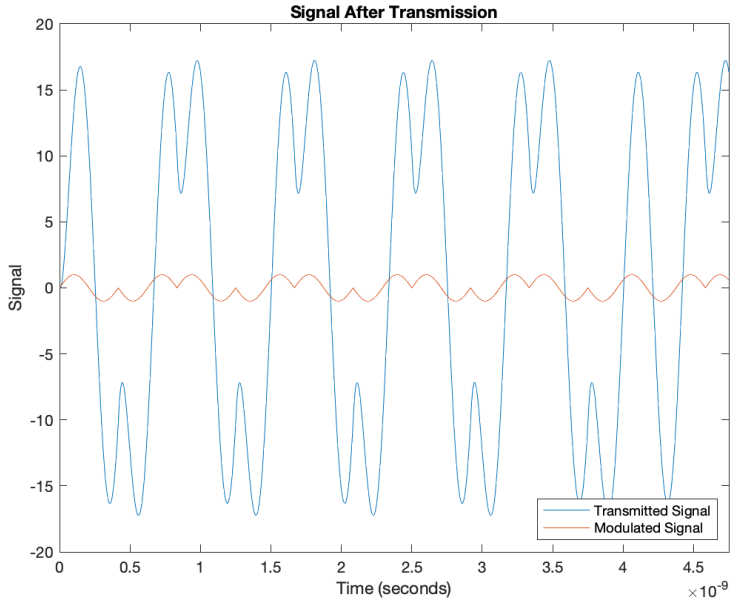

Figure 22: Signal after it has been sent through the transmitter.

Once the signal passes through space, we can see there is a significant loss in power in Figure 23 which can be attributed to the free space path loss discussed previously.

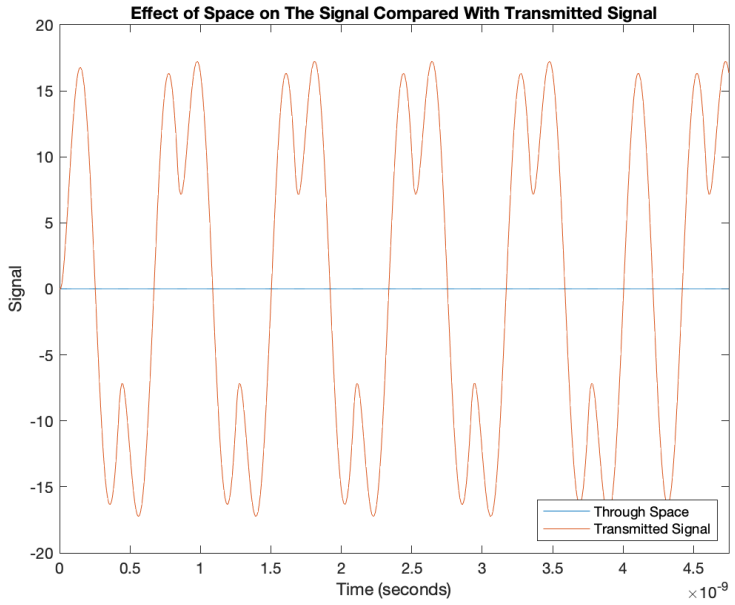

Figure 23: Comparison between the transmitted signal before and after it has been through space.

Once zoomed in, we can see in Figure 24 that even though there has been a loss of power through transmission, the phase shift are maintained in the signal.

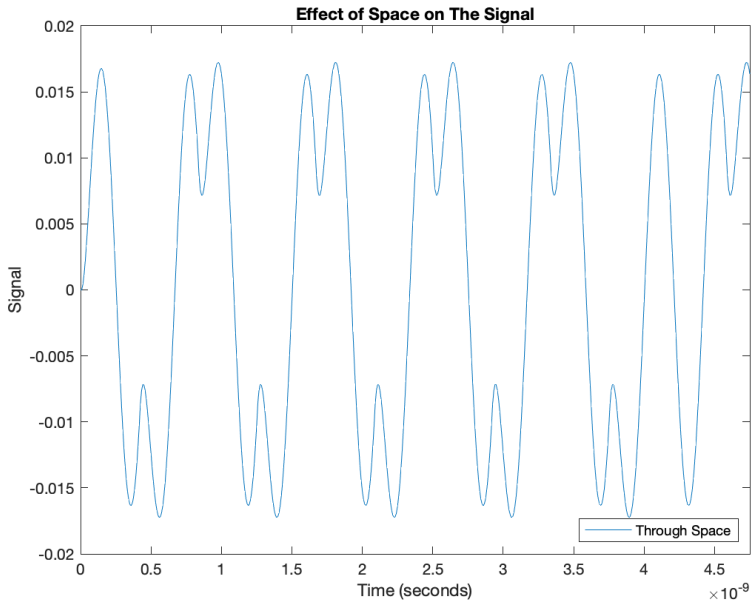

Figure 24: Zoomed in version of the signal after it has travelled through free space.

After the signal has passed through the receiver, we can see in Figure 25 how the gain of the receiver preserves the phase shifts.

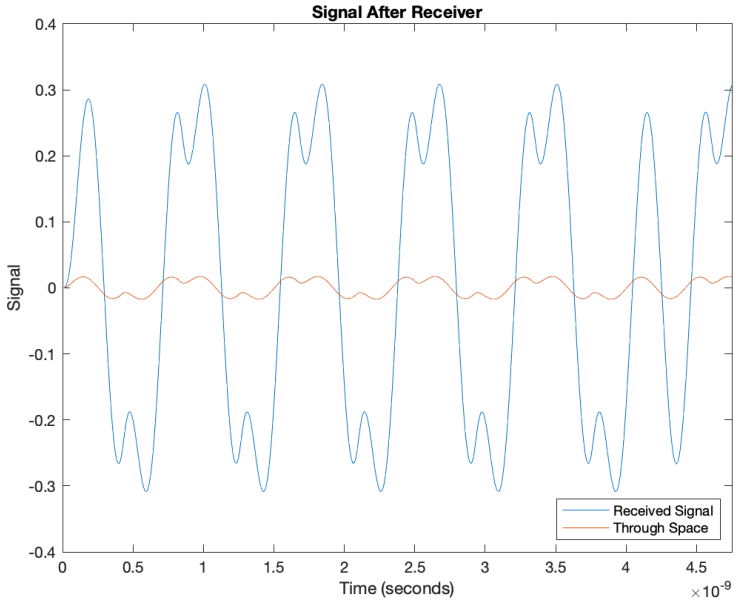

Figure 25: Signal after the gain of the receiver compared with the signal after it passed through space.

In Figure 26, we can see how the demodulated signal compares with the original modulated signal. There is a significant drop in power and some more smoothing of high frequency components, but the overall phase shifts are still preserved. This is what makes BPSK so robust.

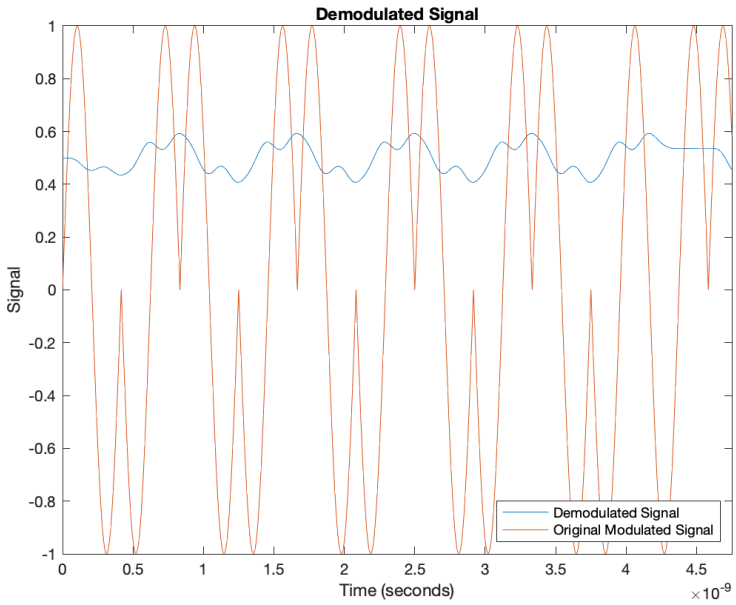

Figure 26: Demodulated signal compared with the original modulated signal.

We can see in Figure 27 that the the received bits emulate the phase shifts of the demodulated signal with a small delay.

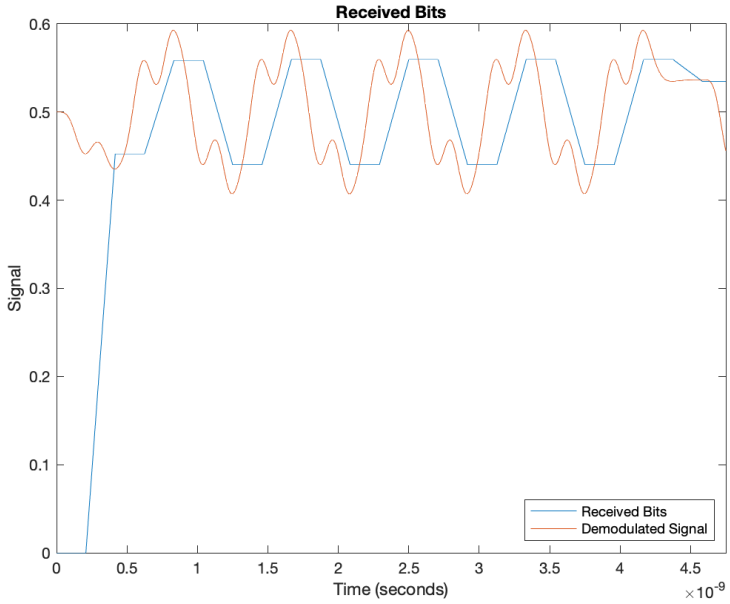

Figure 27: Demodulated signal compared with the same signal converted to bits.

Looking at the Figure 28, we can see that the received bits map to the sent bits with a small delay. We need to round the received bits to the nearest integer due to the power loss.

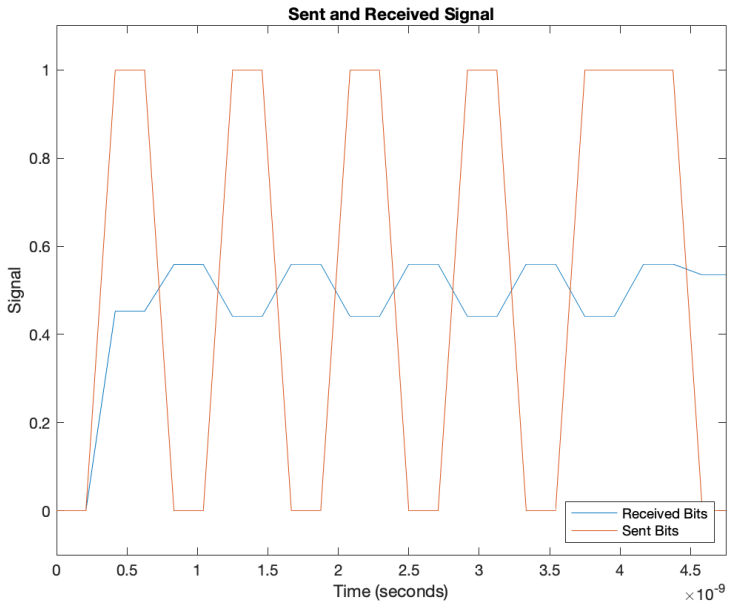

Figure 28: Comparison of the sent and received bits.

We can see in Figure 29 that once we round the received signal, we get the same bits that were sent with a small delay.

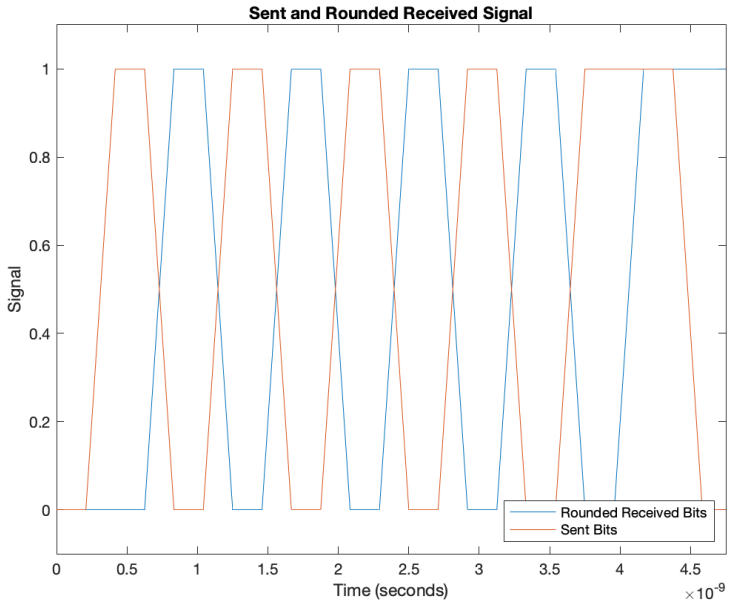

Figure 29: Comparison of the sent and rounded received bits.

When we decode the packet, we need to account for this delay and that is where the preamble comes into play. In our simulation, we can listen for when the preamble starts and adjust the bit frames accordingly. If we do this, we will receive the same message we sent as seem in Figure 30.

```
Decoded message!

textMessage =

    "Hello, world!"
```

Figure 30: Received text message.

V. Discussion and Future Work

Throughout this paper, we implemented an end-to-end communication system complete with error correction, a basic MAC layer, and a plan of work for the hardware implementation. This exercise, while academic in nature, serves as exposure to real-world communication system design. We considered many different tradeoffs that impacted hardware cost, power requirements, and even approach to signal modulation and demodulation.

Despite the learning progress made in this report, there are still many more topics in analog digital communications left unexplored. As a team, we would have enjoyed spending more time capturing schematics and developing hardware designs. Ultimately, we chose off-the-shelf components with parameters much better than what our original specifications required. Because the state-of-the art is beyond our expectations, we had no gauge of how to compare transmitters and receivers. This project facilitated a stronger understanding of various aspects of communication systems like noise, path loss, and carrier waves, but this understanding doesn't quite translate into determining the appropriate hardware components and schematic layup. A future exploration in complete systems design would certainly include the production of a physical transmitter system from basic components.

Another aspect of this communication system left unexplored was the usage of modulation schemes other than BPSK. In our research, BPSK was extremely popular for low-range, moderate bandwidth systems. However, newer protocol versions of WiFi, Bluetooth, and RFID include a wide variety of optimizations on top of the underlying BPSK algorithm. For instance, WiFi 6.0 was recently introduced with a maximum data throughput of 9.6 Gbps - much higher than our measly 11 Mbps targets. WiFi 6 leverages new techniques like *orthogonal frequency division multiple access* (OFDMA) which allows one transmission to deliver data to multiple devices at once. In our exploration of a communication system in the WiFi frequency range, we only ever considered a single transmitter and a single receiver with essentially with very little expectation that two devices might try to communicate simultaneously.

Another area ripe for exploration would be the concept of data channels and channel selection. For WiFi, the 2.4 GHz carrier wave actually comes in 11 different possible channels all operating at slightly different frequencies. During class, we lightly touched on this concept, but did not explore channel interference for phase-shift-keyed. It is unclear to us why a channel is 22 MHz in width, and why overlapping of these channels cause increased error rates.

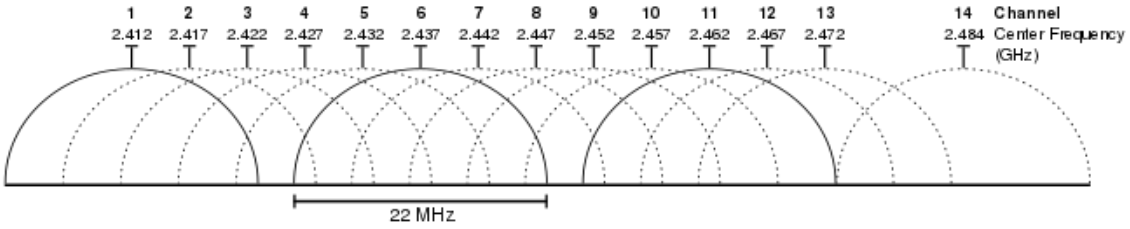

Figure 31: 2.4 GHz WiFi channels (Wikipedia - WLAN Channels)

Despite the time constraints, we were very pleased with final product and results of the Simulink model. Simulink certainly had a sharp learning curve as well as some performance issues that forced us to frequently save and restart the program. However, the ubiquity of resources, models, and inspiration made it easy to add in new components like the white noise block and error correction calculation blocks. We are extremely excited to continue our exploration of signals, systems, communication, and wireless into the future.

VI. Resources

[1] FCC ISM Band Regulation

[2] Understanding the 802.11 Wireless LAN MAC frame format

[3] Calculating Hamming Codes

[4] BPSK - Binary Phase Shift Keying - Auburn University

[5] Digital Phase Modulation

[6] nRF24L01 - Nordic Semiconductors

[7] Typical gain of smartphone WiFi receiver

[8] Top-Down Design of an RF Receiver

VII. Appendix A

All code and schematic layup is available at https://github.com/jkelleyrtp/adc-bpsk-simulation

Code

1. System Variables

```
%% Run this section to set up the BPSK
% Setting All Variables For Simulink
%
% Fc: Carrier frequency
% Tb: Bit period
% wperiod: integrating window period
% SampSin: Time Difference For Sampling Sine
% Tau: Component for LPF transfer function
% Loss: Frequency to handle loss
%

Fc = 2.4e9;
Tb = 1/(Fc);
```

```
wperiod = 50;
SampSin = Tb/wperiod;
tau = 1/(2*pi*(Fc+1e9));
loss = Fc*1000;
```

2. Timetable Construction

```
% Create Packet For Sending
random_additions = 'asdf';
bitlist = get_bits('Hello, world!');
encoded = hammingEncode(bitlist);
packet = createPacket(encoded, random_additions);

bits = createTimetable(Tb, packet)

disp('Set up complete!')
```

3. ASCII Converter

```
function bitlist = get_bits(string)
%
% String: Input string for ASCII conversion
%
% Converts a string into a sequence of bits for the data portion the
% packet.
%

char_array = [];

for k = 1:length(string)
    char_array = [char_array, string(k)];
end

bitlist = [];
for i = 1:length(char_array)
    decimals = pad(dec2bin(char_array(i)), 8, 'left', '0');
    X = str2num(reshape(decimals.', [], 1));
    bitlist = vertcat(bitlist, X);
end
```

4. Hamming Encoder

```
function encoded = hammingEncode(input_vector)
%
% input_vector: vector of integer bits
%
% Encodes bits with a Hamming(7,4) scheme.
%

encoded = [];
for i = 1:4:length(input_vector)
    encoding_portion = input_vector(i:i+3);
    encData = encode(encoding_portion,7,4,'hamming/binary');
    encoded = vertcat(encoded, encData);
end
```

5. Packet Construction

```
function packet = createPacket(bitstring, random_additions)
%
% bits: list of bits for the data section of the packet
%
% Build a packet as shown in the paper.
%
preamble = [1, 0, 1, 0, 1, 0, 1, 0]';
frame_control = [1, 1, 0, 0, 1, 1, 0, 0]';

calculated_length = length(bitstring)
len = pad(dec2bin( calculated_length ), 8, 'left', '0')
duration = str2num(reshape(len.', [], 1));

body = bitstring;

ending_error = get_bits(random_additions);

packet = vertcat(preamble, frame_control, duration, body, ending_error);
```

6. Timetable Conversion

```
function table = createTimetable(Tb, bits)
%
% Tb: Bit period
```

```
% Bits: Data for the signal in the timetable
%
% Generate a timetable given a bit period and a list of bits.
%

time_array = (0:(length(bits) - 1)) * Tb;
Time = seconds(time_array)';
table = array2timetable(bits,'RowTimes',Time);
```

7. Text Recovery

```
%% Decode Packet And Understand
load('packet_received.mat')
data = deconstructPacket(packet_received);
decoded = hammingDecode(data);
disp('Decoded message!')
textMessage = get_text(decoded)
```

8. Packet Breakdown

```
function data = deconstructPacket(packet)
%
% packet: Packet that has been assembled in the way described in the
% associated paper
%
% Get the data portion of the packet out and return it.
%
index = find(packet(2,:) ~= 0, 1, 'first')
unpadpacket = packet(2, index : end);

duration = int32(unpadpacket(17:24))
duration = bin2dec(num2str(duration))
data = unpadpacket(25:(25+(duration - 1)));
```

9. Hamming Decoder

```
function decoded = hammingDecode(input_vector)
%
% input_vector: Array of integer bits
```

```
%
% Decode message in the Hamming(7,4) protocol.
%
decoded = [];
for i = 1:7:length(input_vector)
    decoding_portion = input_vector(i:i+6);
    decData = decode(decoding_portion,7,4,'hamming/binary');
    decoded = vertcat(decoded, decData');
end
```

10. Decoder

```
function text = get_text(bits)
%
% bits: Integer array of bits
%
% Convert integer bit array to ASCII text.
%
to_string = num2str(bits);
text = "";
for i = 1:8:length(to_string)
    letter = to_string(i:i+7)';
    bits = append(letter);
    text = append(text, char(bin2dec(bits)));
end
```